RANCANG BANGUN GAME EDUKASI *JAVA CODING GAME* MENGGUNAKAN METODE *GENERATING TREE* PADA *CONTEXT FREE GRAMMAR*

SKRIPSI

Oleh :

SELAMET HARIADI
NIM. 06550073

JURUSAN TEKNIK INFORMATIKA
FAKULTAS SAINS DAN TEKNOLOGI
UNIVERSITAS ISLAM NEGERI (UIN)
MAULANA MALIK IBRAHIM MALANG
2013

RANCANG BANGUN GAME EDUKASI *JAVA CODING GAME*
MENGGUNAKAN METODE *GENERATING TREE*
PADA *CONTEXT FREE GRAMMAR*

SKRIPSI

Diajukan kepada :

Fakultas Sains dan Teknologi
Universitas Islam Negeri (UIN) Maulana Malik Ibrahim Malang
Sebagai Salah Satu Persyaratan Guna Memperoleh Gelar
Sarjana Teknik Informatika (S.Kom)

Oleh :

SELAMET HARIADI
NIM. 06550073

JURUSAN TEKNIK INFORMATIKA
FAKULTAS SAINS DAN TEKNOLOGI
UNIVERSITAS ISLAM NEGERI (UIN)
MAULANA MALIK IBRAHIM MALANG
2013

LEMBAR PERSETUJUAN

RANCANG BANGUN GAME EDUKASI *JAVA CODING GAME*
MENGGUNAKAN METODE *GENERATING TREE*
PADA *CONTEXT FREE GRAMMAR*

SKRIPSI

Oleh :

SELAMET HARIADI
06550073

Telah Disetujui,
6 Juli 2013

Pembimbing I	Pembimbing II
Fatchurrochman, M.Kom	**Dr. Ahmad Barizi, M.A**
NIP. 19700731 200501 1 002	**NIP. 19731212 199803 1 001**

Mengetahui,
Ketua Jurusan Teknik Informatika
Fakultas Sains dan Teknologi
Universitas Islam Negeri (UIN) Maulana Malik Ibrahim Malang

Ririen Kusumawati, M.Kom
NIP. 19720309 200501 2 002

HALAMAN PENGESAHAN

RANCANG BANGUN GAME EDUKASI *JAVA CODING GAME* MENGGUNAKAN METODE *GENERATING TREE* PADA *CONTEXT FREE GRAMMAR*

SKRIPSI

Dipersiapkan dan disusun oleh:

Selamet Hariadi
NIM. 06550073

Telah Dipertahankan Di Depan Dewan Penguji Skripsi
Dan Dinyatakan Diterima Sebagai Salah Satu Persyaratan
Untuk Memperoleh Gelar Sarjana Komputer (S.Kom)
Tanggal, 10 Juli 2013

Susunan Dewan Penguji :		**Tanda Tangan**
Penguji Utama	**: Ririen Kusumawati, M.Kom** **NIP. 19720309 200501 2 002**	( )
Ketua Penguji	**: Yunifa Miftachul Arif, M.T** **NIP. 19830616 201101 1 004**	( )
Sekretaris Penguji	**: Fatchurrochman, M.Kom** **NIP. 19700731 200501 1 002**	( )
Anggota Penguji	**: Dr. Ahmad Barizi, M.A** **NIP. 19731212 199803 1 001**	( )

Mengetahui dan Mengesahkan
Ketua Jurusan Teknik Informatika
Fakultas Sains dan Teknologi
Universitas Islam Negeri (UIN) Maulana Malik Ibrahim Malang

Ririen Kusumawati M.Kom
NIP. 19720309 200501 2 002

LEMBAR PERNYATAAN

Saya yang bertanda tangan dibawah ini :

Nama : Selamet Hariadi

NIM : 06550073

Jurusan : Teknik Informatika

Judul Skripsi : **RANCANG BANGUN GAME EDUKASI *JAVA CODING GAME* MENGGUNAKAN METODE *GENERATING TREE* PADA *CONTEXT FREE GRAMMAR***

Menyatakan dengan sebenar-benarnya bahwa hasil penelitian yang saya buat ini tidak terdapat unsur-unsur penjiplakan karya penelitian atau karya ilmiah yang pernah dibuat oleh orang lain baik sebagian maupun seluruhnya, kecuali dalam bentuk kutipan dan disebutkan dalam sumber kutipan tersebut didalam daftar pustaka.

Apabila ternyata hasil penelitian ini terbukti terdapat unsur-unsur jiplakan, maka semua akibat dari hal tersebut akan menjadi tanggung jawab saya sepenuhnya, tanpa melibatkan dosen pembimbing atau pihak lain.

Demikian pernyataan ini saya buat dengan penuh kesadaran.

Malang, 15 Juli 2013

Yang membuat pernyataan,

Selamet Hariadi

NIM. 06550073

MOTTO

إِنَّ مَعَ ٱلْعُسْرِ يُسْرًا ۝٦

"Sesungguhnya bersama kesulitan ada kemudahan" (Al-Insyirah [94] : 6)

"Ingat Hidup, Untuk Apa...? Ingat Mati, Mau Kemana...?"

Hidup Sekali, Hiduplah yang Berarti

Karena Setiap Gerak Sendi adalah Lillahi

PERSEMBAHAN

بِسۡمِ ٱللَّهِ ٱلرَّحۡمَٰنِ ٱلرَّحِيمِ

Syukur terhatur untuk Allah SWT yang telah memberikan petunjuk, bimbingan dan jalan yang terbaik agar skripsi ini dapat terselesaikan.

Sholawat dan salam semoga tetap tercurah kepada Sang Revulusioner Sejati dan Suri Tauladan Terbaik Nabi Besar Muhammad SAW, keluarga Sahabat dan seluruh penerus yang memperjuangkan sunnahnya.

Beribu terimakasih ku persembahkan untuk Bapak dan Ibu atas cinta, dukungan, semangat, nasehat dan kesabaran yang diberikan. Semoga dapat melakukan yang terbaik untuk bapak dan ibu serta keluarga.

Pak Fatchurrochman dan Pak Ahmad Barizi yang dengan sabar membimbing hingga terselesaikannya skripsi ini.

Ustadz Halimi Zuhdy dan juga Ustadz Saiful Mustofa yeng telah banyak memberikan nasehat untuk menjadi pribadi yang lebih baik.

Dosen Teknik Informatika Bu Ririen Kusumawati, Pak Syauqi, Pak Syahid, Pak Yaqin, Pak Suhartono, Pak Totok, Pak Yunifa, Pak Zainal Abidin, Pak Faisal sebagai dosen wali, Pak Cahyo, Bu Roro, Bu Hani, Pak Amin, Seluruh Dosen Teknik Informatika, laboran, serta admin dan seluruh civitas akademika Jurusan Teknik Informatika dan UIN Maulana Malik Ibrahim Malang.

Semua Dosen dan Guru yang memberikan banyak keilmuan dan pengetahuan.

Para sahabat Slamet Arif Billah, Syamsudin, Ramadhita, Hartanto, Zainal, Agung, Budi, Ruslan, Iqrok, Nurul Hasanah, Nurul Hidayati, Khoirul Hasanah, Lina, Fithri, Erik, Asad dan Sahabat AIR semua. Semoga kita semua dapat semakin lebih baik daripada sebelumnya, Aamiiiiiin.

Segenap sahabat di Teknik Informatika 2006, Terima Kasih untuk semuanya.

Serta semua pihak yang membantu terselesaikannya skripsi ini…

Terima kasih banyak, semoga kita semua dapat menjadi selalu lebih baik daripada sebelumnya

KATA PENGANTAR

Alhamdulillah, segaa puji bagi Allah SWT yang telah memberikan anugerah kemampuan dalam hidup pun juga pada semangat untuk menyelesaikan skripsi. Shalawat dan salam selalu terhatur pada sang revolusiuner sejati dan suri tauladan terbaik Nabi Muhammad SAW yang telah memberikan berbagai banyak pencerahan dan jalan terang dalam beragama Islam.

Dalam penyelesaian tugas akhir yang berjudul "Rancang Bangun *Java Coding Game* menggunakan Metode *Generating Tree* pada *Context Free Grammar*" ini telah cukup banyak pihak yang membantu dan memberikan semangat untuk segera selesai. Atas bantuan yang telah diberikan, penulis ingin menyampaikan penghargaan dan ucapan terima kasih yang sedalam-dalamnya kepada:

1. Prof Dr. Mudjia Raharjo, M.Si selaku Rektor Universitas Islam Negeri (UIN) Maulana Malik Ibrahim Malang yang memberikan banyak inspirasi kebaikan dan berfikir lebih baik lagi.
2. Dr. drh. Bayinatul Muchtarochmah, M.Si selaku Dekan Fakultas Sains dan Teknologi Universitas Islam Negeri (UIN) Maulana Malik Ibrahim Malang.
3. Ririen Kusumawati, M.Kom, selaku Ketua Jurusan Teknik Informatika Universitas Islam Negeri (UIN) Maulana Malik Ibrahim Malang; yang memberikan saran hingga semangat untuk segera menyelesaikan skripsi.

4. Fatchurrochman, M.Kom selaku Dosen Pembimbing yang telah memberi masukan, saran, bimbingan hingga bantuan dalam proses menyelesaikan skripsi ini.
5. Dr. Ahmad Barizi M.A selaku Dosen Pembimbing Integrasi Sains, Teknologi dan Islam yang telah memberi masukan, saran dalam proses menyelesaikan skripsi ini.
6. Zainal Abidin, M.Kom yang memberikan banyak masukan untuk segera menyelesaikan skripsi ini.
7. Semua Bapak dan Ibu Dosen Teknik Informatika UIN Malang yang telah mengajarkan dan memberikan banyak ilmu dengan tulus. Semoga Ilmu yang di berikan dapat bermanfaat di dunia dan akhirat.
8. Segenap sahabat *Azzam Islamic Research*, *Fun Java*, DNA, Pagar Nusa, JDFI, Buletin *Misykatul Afkar* hingga *Online Marketer Group* yang memberikan pengalaman berharga bagi penulis.
9. Semua pihak yang tak bisa disebutkan satu persatu dan telah menjadi motivator demi terselesaikannya penyusunan skripsi ini. Semoga karya sederhana ini dapat bermanfaat dan memberi arti bagi kemajuan bangsa.

Penulis berharap untuk riset selanjutnya dapat lebih baik dan lebih bermanfaat untuk khalayak ramai.

Malang 15 Juli 2013

Penulis

Selamet Hariadi

DAFTAR ISI

DAFTAR GAMBAR

DAFTAR KODE PROGRAM

DAFTAR TABEL

ABSTRAK

Hariadi, Selamet. 2013. Rancang Bangun *Game* Edukasi *Java Coding Game* Menggunakan Metode *Generating Tree* Pada *Context Free Grammar*. Jurusan Teknik Informatika, Fakultas Sains dan Teknologi, Universitas Islam Negeri Maulana Malik Ibrahim Malang. Pembimbing I: Fatchurrochman, M.Kom, Pembimbing II: Dr Ahmad Barizi M.A

Kata kunci: c*ontext free grammar*, ANTLR, grammar, GTGE, *generating tree*

Terinspirasi dari bahasa natural manusia, ilmuwan-ilmuwan ilmu komputer yang mengembangkan bahasa pemrograman, turut serta memberikan tata bahasa (pemrograman) secara formal. Tata bahasa ini diciptakan secara bebas konteks dan disebut CFG (*Context Free Grammar*). Hasilnya, dengan pendekatan formal ini, kompiler suatu bahasa pemrograman dapat dibuat lebih mudah dan menghindari ambiguitas ketika *parsing* bahasa tesebut.

ANTLR membantu melakukan inialisasi grammar java yang bisa bersifat *Domain Spesific Language* (DSL). Pembuatan grammar dengan bahasa *Extended Backnaus-Naur Form* mudah dipelajari dan dibuat, namun untuk lingkup bahasa yang lebih detail dan dalam perlu pembelajaran yang lebih mendalam pula.

Dengan bantuan ANTLRWorks dapat dilihat bagaimana dalam *grammar* gambaran pola *Generating Tree* sebagai dasar untuk melakukan *Lexer* dan *Parser* hingga membentuk sebuah DSL.

Generate code grammar setelah melalui pembuatan dan analisa menggunakan ANTLRWorks untuk melihat gambaran pola *Generating Tree*, ANLTR membantu pula untuk menghasilkan *Token*, *Lexer* dan *Parser*. Untuk versi ANTLR versi 4 menghasilkan *class* lain pula seperti *Listener*, *Visitor*, *Baselistener*, *Basevisitor* dan lainnya dengan ekstensi *.dot*.

Setelah grammar menghasilkan *Token*, *Lexer* dan *Parser* selanjutnya bisa diujicoba dengan program sederhana memanggil ketiga program ini. Setelah sukses selanjutnya bisa diintegrasikan dengan program lainnya.

GTGE merupakan *Game Engine* buatan Indonesia yang memudahkan bagi para pembuat game bersifat 2 dimensi. Dengan game yang sudah dihasilkan seperti *Warlock* yang di dalamnya ada *Level Builder* dapat membantu pembuatan halaman Level, mengatur tata letak, *enemy*, hingga waktu bermain dalam sebuah game lebih cepat.

ABSTRACT

Hariadi, Selamet. 2013. Educational Game Design Java Game Coding with Method Implementation Using Generating Tree In Context Free Grammar. Departement of Informatics Engineering, Faculty of Science and Technology, State Islamic University of Maulana Malik Ibrahim Malang. Advisor I: Fatchurrochman, M.Kom, Advisor II: Dr Ahmad Barizi, M.A

Keywords: c*ontext free grammar*, ANTLR, grammar, GTGE, *generating tree*

Inspired by the natural human language, computer science scientists who developed the programming language, participating gives grammar (programming) formally. This grammar is created independently of context and called CFG (Context Free Grammar). The result, with the formal approach, the compiler of a programming language can be made more easily and avoid ambiguity when parsing the language proficiency level.

ANTLR java grammar inialisasi help doing that could be Domain Specific Language (DSL). Making grammar with language-Naur Form Extended Backnaus easy to learn and made, however, for a more detailed scope of language and the need to study more in-depth as well.

With the help of ANTLRWorks can be seen how the grammar description Tree Generating patterns as a basis for Lexer and Parser to establish a DSL.

Generate code grammar after going through the manufacture and analysis using ANTLRWorks to see the picture of the pattern of Generating Tree, ANLTR help also to generate tokens, Lexer and Parser. ANTLR version 4 to version produces another class as well as Listener, Visitor, Baselistener, Basevisitor and other with the extension *.dot*.

After grammar generating tokens, Lexer and Parser can subsequently be tested with a simple program to call the three programs. After further success can be integrated with other programs.

GTGE Game Engine is made in Indonesia which makes it easier for game makers is 2-dimensional. With a game that has been generated as Warlock in which there is Level Builder can help to create page level, set the layout, enemy, until the time playing in a game faster.

BAB I

PENDAHULUAN

1.1 Latar Belakang

Perkembangan aplikasi berbasis komputer mulai banyak diminati oleh khalayak ramai, hal ini dikarenakan lebih interaktif dan memudahkan dalam memakainya. Dibalik aplikasi program dan sistem yang kompleks terdapat banyak aturan program yang terangkai dalam kode-kode program. Kode-kode program inilah yang akan terlihat tampilan sistem dan fungsionalitas dari program dapat dipergunakan oleh pengguna program tersebut. Di dalam Al-Qur'an Allah berfirman dalam Surat Al-An'am [6] : 101 sebagaimana berikut:

بَدِيعُ ٱلسَّمَٰوَٰتِ وَٱلۡأَرۡضِۖ أَنَّىٰ يَكُونُ لَهُۥ وَلَدٞ وَلَمۡ تَكُن لَّهُۥ صَٰحِبَةٗۖ وَخَلَقَ كُلَّ شَيۡءٖۖ وَهُوَ بِكُلِّ شَيۡءٍ عَلِيمٞ ١٠١

Artinya: *Dia (Allah) Pencipta langit dan bumi. bagaimana Dia mempunyai anak padahal Dia tidak mempunyai isteri. Dia menciptakan segala sesuatu; dan Dia mengetahui segala sesuatu. (Al-An'am [6] : 101)*

Dalam Ayat diatas menunjukkan kuasa Allah yang menciptakan segala sesuatu, Allah adalah pemrogram kehidupan ini. Semua yang ada di manapun merupakan kekuasaan Allah serta hanya Allah yang mengetahui segala sesuatu. Dalam kaitannya dengan pemrograman yang dibutuhkan pemeran utama seorang pemrogram untuk pemrogramannya maka Allah adalah pemrogram kehidupan ini yang segala hal dijalankan atas kehendak Allah. Allah memberikan alam sebagai

tanda-tanda, sebagaimana Allah berfirman dalam surat Ali Imran [3] : 190 berikut:

إِنَّ فِي خَلْقِ ٱلسَّمَٰوَٰتِ وَٱلْأَرْضِ وَٱخْتِلَٰفِ ٱلَّيْلِ وَٱلنَّهَارِ لَءَايَٰتٍ لِّأُو۟لِى ٱلْأَلْبَٰبِ ﴿١٩٠﴾

Artinya: *Sesungguhnya dalam penciptaan langit dan bumi, dan silih bergantinya malam dan siang terdapat tanda-tanda bagi orang-orang yang berakal. (Ali Imran [3] : 190)*

Dalam surat Ali Imran [3]: 190 ini dijelaskan adanya tanda-tanda dalam penciptaan serta pengaturan malam dan siang. Allah juga menurunkan tanda-tanda yang lain seperti gempa atau hujan yang bisa jadi adalahs ebuah peringatan agar manusia kembali ke jalan Allah. Tanda-tanda jika dikaitkan dalam pemrograman adalah kode-kode pemrograman yang membentuk sebuah aplikasi dari rangkaian aturan kode-kode pemrograman.

Dalam ayat di atas dijelaskan pula bagaimana tanda-tanda atau kode kehidupan ini sesuai dengan sistem atau bahasa lainnya adalah sunnatullah. Akibat dari sistem atau sunnatullah ini adalah tentang iman dan kafir, mereka yang beriman akan mendapatkan surga sedang yang kafir akan mendapat neraka. Jika dikaitkan dalam pemrograman yang sebelumnya tentang tanda-tanda yang bisa berarti kode-kode pemrograman, maka sistem atau sunnatulah dalam bahasa pemrograman adalah sistem pemrograman itu sendiri. Sedangkan untuk hasil adanya surga dan neraka jika dikaitkan dalam pembuatan pemrograman adalah adanya berhasilnya membentuk aplikasi atau gagalnya diakibatkan dari cara melakukan pemrograman. Pada hal lainnya seperti *game* bisa dikaitkan adanya

yang menang dan yang kalah dalam bermain *game*. Allah SWT berfirman dalam Surat Al-Jatsiyah [45] : 11 sebagaimana berikut ini.

هَٰذَا هُدًى ۖ وَٱلَّذِينَ كَفَرُوا۟ بِـَٔايَٰتِ رَبِّهِمْ لَهُمْ عَذَابٌ مِّن رِّجْزٍ أَلِيمٌ ﴿١١﴾

Artinya: *ini (Al Quran) adalah petunjuk. dan orang-orang yang kafir kepada ayat-ayat Tuhannya bagi mereka azab yaitu siksaan yang sangat pedih.*

Menurut *Tafsir Ibnu Katsir* kata *hadza* (ini), yakni berarti Al-Qur'an. Sedangkan *hudan* adalah petunjuk dari kesesatan. Kesesatan di sini adalah dari manusia, sedang petunjuk yang diturunkan inilah yang coba diungkapkan dalam Al-Qur'an, bahwa kita mempunyai petunjuk atas apa yang ada terjadi di dalam kehidupan kita. Pun begitu dengan yang ada di kode-kode program adalah petunjuk dari kehidupan sehari-hari kita karena apa yang ada dalam *grammar* kode program sebenarnya adalah sama dengan apa yang terjadi yang kita temui fenomena di alam. Seperti contohnya adalah jika setiap orang ataupun benda mempunya nama untuk mudah dikenali, maka di dalam pemrograman Java juga ada penamaan *class*. Hal lainnya seperti adanya *array* dalam program sama dengan gambaran padi yang ada di lumbung padi lalu disendirikan menurut jenis padi tersebut.

Melihat dari penjelasan ayat di atas tadi, petunjuk yang diberikan Allah adalah agar kita belajar lebih baik dari sebelumnya. Belajar adalah kebutuhan setiap manusia dalam berinteraksi dengan kehidupan dan untuk mencapai kehidupan yang lebih baik setelah meninggal. Menurut Barizi (2009 : 61), manusia sebagai pemangku khalifah Allah SWT wajib secara individual untuk mencari ilmu. Dalam berbagai bidang kehidupan ini ilmu memang sangat

diperlukan dalam interaksi sosial manusia. Allah SWT berfirman dalam Surat Al-Alaq [96] : 1-4 :

ٱقۡرَأۡ بِٱسۡمِ رَبِّكَ ٱلَّذِي خَلَقَ ﴿١﴾ خَلَقَ ٱلۡإِنسَٰنَ مِنۡ عَلَقٍ ﴿٢﴾ ٱقۡرَأۡ وَرَبُّكَ ٱلۡأَكۡرَمُ ﴿٣﴾ ٱلَّذِي
عَلَّمَ بِٱلۡقَلَمِ ﴿٤﴾

Artinya: *1. Bacalah dengan (menyebut) nama Tuhanmu yang Menciptakan,*
2. Dia telah menciptakan manusia dari segumpal darah.
3. Bacalah, dan Tuhanmulah yang Maha pemurah,
4. Yang mengajar (manusia) dengan perantaran kalam
(Al-Alaq [96] : 1-4)

Dalam *Tafsir Ibnu Katsir, Iqra'* (bacalah), maksudanya adalah bacalah, hai Muhammad! Inilah yang pertama kali diturunkan Jibril AS kepada beliau. Ini adalah ayat pertama kali yang diturunkan kepada nabi Muhammad SAW. *Bismi rabbika* (dengan Nama Tuhanmu) maksudnya adalah dengan Perintah Tuhanmu. *Alladzī khalaq* (yang telah Menciptakan) seluruh makhluk.

All adzī 'allama bil qalam (yang telah mengajar dengan pena), yakni mengajarkan menulis dengan pena. Banyak teguran yang diberikan Allah untuk manusia dengan berbagai macam contoh ummat masa lalu maupun sekarang.

Kata Belajar sudah tertanam begitu dalam di otak manusia, bahkan sejak dalam kandungan (Jaya, 2010 : viii). Belajar adalah hal yang penting dalam kehidupan kita karena hampir setiap bidang kehidupan yang kita lalui selalu membutuhkan keilmuan. Sedangkan keilmuan kita bisa dapatkan dengan belajar.

Proses mengajar juga penting dalam sebuah sistem pembelajaran. Menurut Novian Triwidia Jaya (2010 : 28) mengajat pada prinsipnya adalah mengkomunikasikan dan mengirimkan informasi dari pengajar ke pelajar.

Penyaluran informasi berupa pengetahuan inilah yang menjadi hal penting dalam proses belajar dan mengajar.

Sistem pembelajaran bahkan juga ada dengan model *Hypnosis Learning*. Metode yang mungkin baru bagi sebagian orang, menurut Novian Triwidia Jaya (2010 : 208) Metode *Hypnosis Learning* (HL) dikatan berhasil jika terciptanya sebuah fokus, keputusan dan kontrol dari anak-anak dengan tetap memperhatikan kesukaan dari anak dalam menggunakan kemampuan imajinasi mereka.

Pemahaman terhadap kode-kode program yang kuat dalam bahasa pemrograman sangat diperlukan bagi pembuat program atau sistem. Namun bagi pembuat program yang masih pemula rasanya memang cukup sulit memulai jika melihat banyak aturan bahasa program yang cukup banyak. Begitu juga bagi mereka yang telah mahir atau mereka yang bergelut di dunia IT yang kompleks namun tak hanya menggeluti dunia kode program namun juga bidang yang lainnya, kadang kita akan lupa pada sebagian aturan program dalam membentuk sebuah sistem program yang kompleks.

Banyak hal yang ditemui dalam kehidupan mahasiswa informatika, sistem informasi atau singkatnya yang berbasis di bidang komputer serta teknologi informasi adalah kurang kuatnya pemahaman akan *script coding* program. Padahal pemahaman yang mendalam terhadap *script coding* adalah yang paling tidak terpenting di dalam orang yang mendalami bidang berbasis IT. Hal ini dikarenakan dalam setiap sistem yang besar dalam sebuah program terdapat jaringan *script coding* yang saling berhubungan secara kompleks dalam menampilkan tampilan yang diinginkan pembuat program. Memang menjadi

programmer adalah bukan pilihan yang harus dibuat pilihan wajib bagi yang memilih bidang IT sebagai pilihan hidupnya untuk berkontribusi untuk dunianya. Masih banyak pilihan konsentrasi bidang lainnya, seperti: desainer, analis sistem, manajer proyek, dan lain sebagainya jika dijelaskan lebih dalam akan terlihat lebih banyak. Namun, seperti kata salah seorang pakar IT di Indonesia yakni Romi Satria Wahono, kemampuan *script coding* patutlah menjadi hal yang terpenting dalam mereka yang bergelut di dunia berbasis IT, apalagi bagi mereka yang membuat dan merancang sistem berbasis IT.

Melihat media pembelajaran yang cukup sukses membantu dalam mengetik sepuluh jari seperti *typing master*, maka media pembelajaran berupa *Game* atau permainan dalam pengenalan hingga pemahaman aturan-aturan program dirasa perlu guna mempermudah *programmer* awam atau pun mereka yang bergelut di dunia IT. Media yang dirancang dan dibangun ini akan sangat berbeda dengan media pembelajaran untuk mengetik karena dengan sistem yang lebih kompleks.

1.2 Rumusan Masalah

Berdasarkan latar belakang di atas maka diperoleh rumusan masalah yang butuh solusi untuk diselesikan, yakni:

1. Bagaimana membangun media pembelajaran edukasi untuk pemahaman aturan-aturan tata bahasa kode program?
2. Bagaimana membangun media pembelajaran yang menyenangkan?
3. Bagaimana membangun media pembelajaran mangatasi kejenuhan?

1.3 Tujuan Penelitian

Setelah menemukan rumusan masalah selanjutnya diperoleh Tujuan penelitian untuk dilakukan ini adalah sebagai berikut:

1. Membuat media pembelajaran edukatif "*Java Coding Game*" sebagai media pembelajaran yang menyenangkan.
2. Membuat media pembelajaran yang berbasis pada *Game*.
3. Membuat media pembelajaran dengan alur sekaligus melakukan *refresh* pada otak.

1.4 Manfaat Penelitian

Hasil dari penelitian ini diharapkan memberikan manfa'at bagi bagi pembuat program yang masih pemula rasanya memang cukup sulit memulai jika melihat banyak aturan bahasa program yang cukup banyak.

Begitu juga bagi mereka yang telah mahir atau mereka yang bergelut di dunia IT yang kompleks namun tak hanya menggeluti dunia kode program namun juga bidang yang lainnya, kadang kita akan lupa pada sebagian aturan program dalam membentuk sebuah sistem program yang kompleks.

1.5 Batasan Masalah

Dari permasalahan diatas, berikut ini diberikan batasan-batasan masalah untuk mencegah membiasnya masalah yang akan diselesaikan:

a. Bahasa pemrograman yang digunakan untuk membangun *Game* edukasi ini adalah *Java 2 Standard Edition* (J2SE).
b. Ruang pemahaman pembelajaran adalah inialisasi program terlebih dahulu.
c. Pengguna *Game* ini hanya pada mereka yang bisa menggunakan J2SE.
d. Materi *Game* dibatasi hanya pada materi dasar-dasar pemrograman.
e. Metode *Generation Tree* digunakan untuk mengeksekusi program.
f. Pengguna *Game* adalah mereka yang mengerti atau dalam proses belajar bahasa java pada range usia 17 sampai 30 tahun.

1.6 Sistematika Penulisan

Sebuah sistem memiliki bagian dari sistem yang cukup kompleks, dalam skripsi ini juga memiliki sistematika penulisan. Berikut penjelasan secara umum tentang sistematika penulisan skripsi ini:

1. Bab I Pendahuluan

   Bab I berisikan tentang pandahuluan skripsi mulai dari latar belakang penelitian skripsi berbasis program studi di Jurusan Teknik Informatika dengan melakukan integrasi Agama. Latar belakang berisikan alasan tentang yang terjadi di lingkungan membuat tergeraknya peneliti untuk melakukan penelitian skripsi ini.

   Bab I juga berisikan tentang Rumusan Masalah setelah melihat latar belakang penelitian untuk menghasilkan rumusan tentang masalah yang terjadi untuk diberikan solusi penyelesaiannya. Sub-bab berikutnya

adalah tentang Tujuan Penelitian yang menyangkut tentang tujuan dilakukannya penelitian skripsi dengan acuan melihat pada latar belakang serta rumusan masalah, sehingga solusi yang diinginkan untuk dilakukan penelitian dalam skripsi ini dapat tercapai.

Manfaat Penelitian juga merupakan bagian dari isi yang ada dalam Bab I. Penjelasan dari Manfaat Penelitian ini adalah pada kebermanfaat penelitian untuk sumbangsih dalam dunia keilmuan dan pelaksanaan kehidupan yang lebih baik.

Batasan Masalah juga cukup penting dalam sebuah penelitian agar penelitian yang dilakukan dapat terfokus pada solusi yang diharapkan dari latar belakang, lalu merumuskan masalah yang terciptanya tujuan penelitian hingga manfaat pentingnya penelitian skripsi ini dilakukan. Batasan masalah akan membatasi untuk kebaikan, yakni agar tak meluber ke berbagai bidang lain sehingga tujuan penelitian dari awal mulai terpinggirkan. Hal terakhir yang dibahas dalam Bab I adalah tentang Sistematika Penulisan skripsi untuk mempermudah mendalami skripsi penjelasan di Sistematika Penulisan ini dapat membantu sebagai acuan awal.

2. Bab II Tinjauan Pustaka

Bab II berisikan Tinjauan Pustaka penelitian yang meliputi tentang pengertian Edukasi yang merujuk pada judul penelitian menyangkut tentang edukasi. Hal lain yang menjadi pembahasan lebih dalam adalah

tentang serta alat-alat pendukung penelitian skripsi ini mulai dari ANTLR hingga yang lainnya.

3. Bab III Analisis dan Perancangan

   Baba III berisikan tentang Analisis dan Perancangan Sistem untuk mengenal lebih dalam tentang penelitian yag dilakukan berdasarkan pada Bab pendahuluan hingga tinjauan pustaka.

   Dalam Bab III ini dilakukan analisis yang diperlukan hingga pendalaman pada perancangan sistem dan kolerasinya dengan *Game*.

4. Bab IV Hasil dan Pembahasan

   Baba IV merupakan Bab Hasil dan pembahasan yang melihat dari Analisis dan Perancangan Sistem dari rujukan bab sebelumnya.

5. Bab V Kesimpulan dan Saran

   Bab V berisikan Kesimpulan dan Saran tentang penelitian *Game* Edukasi Java Coding *Game* yang dilakukan.

BAB II

TINJAUAN PUSTAKA

2.1 Pengertian Edukasi

Edukasi bisa diartikan sebagai rangkaian usaha yang bertujuan untuk mempengaruhi orang lain, mulai dari individu, kelompok, keluarga dan masyarakat luas. Saat ini perkembangan Edukasi mulai berkembang ke berbagai ranah bidang kehidupan sebagai sarana untuk menjadikan masyarakat lebih baik.

Umumnya orang mengartikan edukasi dari bahasa Inggris yakni *education*, dalam kamus besar bahasa Inggris *education* berarti pendidikan, pendidikan berasal dari kata didik, atau mendidik yang berarti memelihara dan membentuk latihan.

Sedangkan dalam *Kamus Besar Bahasa Indonesia* (1991) pendidikan diartikan sebagai proses pengubahan sikap dan tata laku seseorang atau sekelompok orang dalam usaha mendewasakan manusia melalui upaya pengajaran dan pelatihan. Dalam pengajaran dan pelatihan ini terdapat proses pembelajaran yang menjadikan setiap mereka yang melakukannya mendapatkan aktivitas belajar baik pribadi maupun secara kelompok.

Situasi belajar yang baik berarti mengupayakan penyampaian materi pelajaran yang baik sehingga mudah dimengerti dan dipahami. Konsep Pendidikan dalam Al-Qur'an melalui metode yang bernalar serta sarat dengan kegiatan meneliti, membaca, mempelajari, dan penelitian ilmiah terhadap

manusia sejak manusia masih dalam bentuk di dalam kandungan ibu yang berarti belum lahir ke dunia seperti dalam Surat Al-Alaq [96] : 1-5 seperti berikut ini:

ٱقۡرَأۡ بِٱسۡمِ رَبِّكَ ٱلَّذِي خَلَقَ ۝١ خَلَقَ ٱلۡإِنسَٰنَ مِنۡ عَلَقٍ ۝٢ ٱقۡرَأۡ وَرَبُّكَ ٱلۡأَكۡرَمُ ۝٣
ٱلَّذِي عَلَّمَ بِٱلۡقَلَمِ ۝٤ عَلَّمَ ٱلۡإِنسَٰنَ مَا لَمۡ يَعۡلَمۡ ۝٥

Artinya:

1. *Bacalah dengan (menyebut) nama Tuhanmu yang Menciptakan,*
2. *Dia Telah menciptakan manusia dari segumpal darah.*
3. *Bacalah, dan Tuhanmulah yang Maha pemurah,*
4. *Yang mengajar (manusia) dengan perantaran kalam,*
5. *Dia mengajar kepada manusia apa yang tidak diketahuinya. (Al-Alaq [96] : 1-5)*

Dalam *Tafsir Ibnu Abbas* dijelaskan bahwa turunnya ayat ini Nabi Muhammad SAW mendapatkan proses edukasi dari malaikat Jibril yang mana Nabi Muhammad SAW tak bisa membaca namun oleh Malaikat Jibril membacakan 4 ayat pertama surat Al-Alaq ini kepada Nabi Muhammad SAW. Dalam ayat ke-4 dijelaskan bahwa Allah mengajarkan manusia menulis dengan pena.

Di ayat lainnya dalam Al-Qur'an surat An-Nahl [16] : 125 juga telah menerangkan seperti berikut ini:

ٱدۡعُ إِلَىٰ سَبِيلِ رَبِّكَ بِٱلۡحِكۡمَةِ وَٱلۡمَوۡعِظَةِ ٱلۡحَسَنَةِۖ وَجَٰدِلۡهُم بِٱلَّتِي هِيَ أَحۡسَنُۚ إِنَّ
رَبَّكَ هُوَ أَعۡلَمُ بِمَن ضَلَّ عَن سَبِيلِهِۦۖ وَهُوَ أَعۡلَمُ بِٱلۡمُهۡتَدِينَ ۝١٢٥

Artinya:

Serulah (manusia) kepada jalan Tuhan-mu dengan hikmah dan pelajaran yang baik dan bantahlah mereka dengan cara yang baik. Sesungguhnya Tuhanmu Dialah yang lebih mengetahui tentang siapa yang tersesat dari jalan-Nya dan Dialah yang lebih mengetahui orang-orang yang mendapat petunjuk.. (An-Nahl [16] :125)

Menurut *Tafsir Ibnu Katsir* maksud Surat An-Nahl [16] : 125 ini adalah untuk seruan mengajak ke jalan Tuhan dengan hikmah dan nasihat yang baik serta perintah membantah mereka dengan yang lebih baik. Sesungguhnya Tuhan, Dia-lah yang lebih Mengetahui tentang siapa-siapa yang tersesat dari Jalan-Nya dan Dia-lah yang lebih Mengetahui orang-orang yang mendapat petunjuk.

Makna ayat *Ud'u ilā sabīli rabbika* menurut *Tafsir Ibnu Katsir* yang berarti ajaklah ke Jalan Tuhanmu, yakni ke dalam Agama Tuhanmu. Tujuan mengajak ke jalan Tuhan adalah agar manusia bertakwa kepada Allah SWT dengan jalan yang telah ditetapkan melalui ajaran Agama Islam. Norma kehidupan sebagai manusia sosial yang selayaknya saling menghargai bisa didapat dengan jalan aturan yang ada di dalam Agama. Tiap manusia memang memiliki kebebasan, namun kebebasan yang dimiliki setiap manusia tentu dibatasi dengan kebebasan manusia lainnya pula. Maka disinilah peran setiap manusia yang memilki Agama agar bisa saling berbuat dengan interaksi yang baik dengan manusia lainnya. Jika dielaborasi lagi mengajak atau menyeru ke Jalan Tuhan adalah sesuai ajaran yang ada dalam bingkai Agama, agama yang dimaksud adalah Agama Islam ang menjadi rahmat bagi sekalian alam.

Ibnu Katsir dalam tafsirnya lebih lanjut menerangkan metode yang dimaksud *Bil hikmati* (dengan hikmah), yakni dengan Al-Qur'an. Lalu makna *Wal mau'zhatil hasanati* (dan nasihat yang baik), yakni dan nasihatilah mereka dengan nasihat-nasihat Al-Qur'an. Jika dirangkai dari kalimat awal tetang perintah mengajak atau menyeru manusia ke jalan Tuhan, maka pada lanjutan kalimat di ayat ini menurut *Ibnu Katsir* menjelaskan tentang cara dengan hikmah yang

dimaksud adalah dengan Al-Qur'an sebagai rujukan. Al-Qur'an sebagai kitab suci umat Islam adalah rujukan utama dalam gerak sendi kehidupan umat Islam.

Dalam *Tafsir Ibnu Katsir* kalimat *Wa jādilhum bil latī hiya ahsan* yang artinya serta bantahlah mereka dengan yang lebih baik itu, maksudnya ialah dengan Al-Qur'an.

Inna rabbaka huwa a'lamu bi maη dlalla 'aη sabīlihī (sesungguhnya Tuhanmu, Dia-lah yang lebih Mengetahui siapa-siapa yang tersesat dari Jalan-Nya), yakni dari Agama-Nya. *Wa huwa a'lamu bil muhtadīn* (dan Dia-lah yang lebih Mengetahui orang-orang yang mendapat petunjuk) kepada Agama-Nya.

Dakwah dimaknai mengajak manusia kepada jalan Alloh. Berkembangnya Islam seperti sekarang ini tak lepas dari momentum gerakan dakwah yang menyebar ke seluruh belahan dunia. Dakwah bisa dijadikan gerakan massa, itu bisa dilihat dari banyaknya gerakan yang orientasinya massa banyak di belahan dunia ini. Dakwah merupakan seruan yang ditujukan kepada seluruh strata sosial dalam masyarakat. Tidak ada cara lain untuk gerakan dakwah yang bersifat massa selain ia sendiri terjun ke tengah masyarakat. Allah SWT berfirman dalam Surat Saba' [34] : 28.

وَمَآ أَرۡسَلۡنَٰكَ إِلَّا كَآفَّةٗ لِّلنَّاسِ بَشِيرٗا وَنَذِيرٗا وَلَٰكِنَّ أَكۡثَرَ ٱلنَّاسِ لَا يَعۡلَمُونَ

٢٨

Artinya:.

> *Dan kami tidak mengutus kamu, melainkan kepada umat manusia seluruhnya sebagai pembawa berita gembira dan sebagai pemberi peringatan, tetapi kebanyakan manusia tiada Mengetahui. (Saba' [34] : 28)*

Dalam *Tafsir Ibnu Katsir* yang dimaksud *Wa mā arsalnāka* (dan tidaklah Kami Mengutusmu) ialah Nabi Muhammad. Hal ini bisa dilihat karena Nabi Muhammad adalah sebagai Nabi penerima Al-Qur'an yang kemudian menyebarkannya ke manusia lainnya. Kalimat *Basyīran* (sebagai penyampai kabar gembira) ialah berupa surga bagi siapa saja yang beriman kepada Allah SWT. Maksud dari *Wa nadzīran* (dan pemberi peringatan) ialah berupa neraka bagi siapa pun yang kafir kepada-Nya.

Tak dapat dipungkiri pesatnya perkembangan dunia informasi & gaya kehidupan manusia meuntut hal yang lebih baik dengan kemasan yang menarik dari dakwah Islam. Melihat kepada abad 21 yang akan datang, gerakan dakwah massa haruslah dijadikan *trend* kembali yang akan menjadi *landmark* bagi arus baru kebangkitan Islam meskipun tekanan dan halangan masih lagi berleluasa. Ini tidak bermakna pukulan dan tekanan ke atas gerakan dakwah tidak wujud tetapi gerakan dakwah haruslah untuk menjadikan ia faktor yang akan mempercepat proses penyebaran Islam.

Dakwah bersama jamaah adalah dakwah yang paling efektif dan sangat bermanfaat bagi gerakan islam. Sebaliknya dakwah secara bersendirian akan kurang pengaruhnya dalam usaha menanamkan ajaran Islam pada umat manusia. Atas dasar ini Allah SWT mengisyaratkan dalam Al-Quran dengan firman-Nya:

وَلْتَكُن مِّنكُمْ أُمَّةٌ يَدْعُونَ إِلَى ٱلْخَيْرِ وَيَأْمُرُونَ بِٱلْمَعْرُوفِ وَيَنْهَوْنَ عَنِ ٱلْمُنكَرِ ۚ وَأُو۟لَٰٓئِكَ هُمُ ٱلْمُفْلِحُونَ ﴿١٠٤﴾

Artinya:

> *Dan hendaklah ada di antara kamu segolongan umat yang menyeru kepada kebajikan, menyuruh kepada yang ma'ruf dan mencegah dari yang munkar; merekalah orang-orang yang beruntung. (Ali Imran [3] : 104*)

Dalam ayat tersebut Allah telah mengisyaratkan tentang wajibnya melaksanakan dakwah secara bersama (berjamaah) atau melaksanakan aktivitas bersama. Pada Surah Ali Imran [3] : 104 yang dimaksud Ma'ruf adalah segala perbuatan yang mendekatkan kita kepada Allah SWT, sedangkan munkar ialah segala perbuatan yang menjauhkan kita dari pada-Nya.

Dakwah sekarang memang berdeda dengan cara dakwah dahulu walau substansinya tetap. Sambutan terhadap Islam sudah mulai membaik. Gerakan dakwah yang sahulu lebih pada internal untuk pembentukan karakter Muslim, sekarang memang sudah selayaknya dikembangan menjadi gerakan dakwah yang ke masyarakat luas.

Cara penyampaian informasi di media juga mempengaruhi cara penyampaian dakwah Islam secara lebih baik. Bila penyampaian media massa untuk penyampaian informasi dikenal baik, maka dakwah pun masuk ke ranah media massa untuk saling mengingatkan & mengajak kepada keislaman yang benar. Internet yang mulai menjalar ke berbagai pelosok dunia juga dimanfaatkan para aktivis dakwah dalam mengembangkan dakwah Islamnya. Seperti munculnya situs tentang keislaman & orgasisasi atau jama'ah keislaman di dunia maya ini untuk mengimbangan informsi yang negatif pula dari internet. Situs pertemanan jaringan sosial pun dimanfa'atkan guna mengajak manusia ke jalan yang benar.

Jaman yang berkembang mengharuskan manusia juga memperbaharui peradabannya hingga tak lupa pula pada proses edukasi. Upaya tersebut didukung dengan adanya usaha-usaha dari pihak yang berwenang untuk mengembangkan dan mencari terobosan baru demi tercapainya tujuan pembelajaran tersebut. Salah satu objek pembelajaran yang saat ini dikembangkan yaitu pemanfaatan teknologi pendidikan.

Edukasi merupakan rangkaian proses pembelajaran yang didapat oleh setiap manusia, dalam hal ini menjadikan manusia lebih baik daripada waktu sebelumnya. Edukasi dapat dirumuskan sebagai tuntunan pertumbuhan manusia sejak lahir hingga tercapai kedewasaan jasmani dan rohani, dalam interaksi alam dan lingkungan masyarakatnya. Edukasi merupakan proses yang terus menerus, tidak berhenti sampai di penghujung usia.

Saat ini Edukasi dapat didapat secara formal maupun non -formal. Edukasi secara formal diperoleh dari suatu pembelajaran yang terstruktur yang telah dirancang oleh institusi pendidikan. Hal ini seperti contohnya sekolah formal. Sedangkan pendidikan non-formal adalah pengetahuan yang didapat manusia dalam kehidupan sehari-hari baik yang dialami atau yang dipelajari dari kehidupan dan masyarakat.

Berdasarkan uraian di atas maka dapat disimpulkan bahwa edukasi adalah suatu usaha yang secara terus menerus yang dilakukan manusia untuk menjadi lebih baik keilmuannya daripada sebelumnya..

2.1.1 Tujuan Edukasi

Di Indonesia berdasarkan undang-undang (UU) nomor 2 tahun 1989 disebutkan bahwa pendidikan nasional bertujuan mencerdaskan kehidupan bangsa dan mengembangkan mannusia Indonesia seutuhnya, yaitu manusia yang beriman dan bertaqwa terhadap Tuhan Yang Maha Esa dan berbudi pekerti luhur, memiliki pengetahuan dan ketrampilan, kesehatan jasmani dan rohani, kepribadian yang mantap dan mandiri serta rasa tanggungjawab kemasyarakatan dan kebangsaan.

Dari landasan undang-undang ini edukasi tak hanya pada peningkatan pengetahuan namun juga keterampilan hingga kepribadian yang lebih baik yang mana menjadi tugas masyarakat bersama. Hal ini seperti usaha Pemerintah menjadikan masa anak-anak diasupi dengan edukasi untuk menjadikan masa remaja-nya dapat lebih berpengetahuan untuk menyongsong masa kehidupan di tengah pergaulan dunia internasional.

Tujuan edukasi seperti di dalam termuat di dalam Surat Ar-Rahman [55] : 1-8 sebagaimana berikut ini:

ٱلرَّحۡمَٰنُ ۝١ عَلَّمَ ٱلۡقُرۡءَانَ ۝٢ خَلَقَ ٱلۡإِنسَٰنَ ۝٣ عَلَّمَهُ ٱلۡبَيَانَ ۝٤ ٱلشَّمۡسُ وَٱلۡقَمَرُ بِحُسۡبَانٖ ۝٥ وَٱلنَّجۡمُ وَٱلشَّجَرُ يَسۡجُدَانِ ۝٦ وَٱلسَّمَآءَ رَفَعَهَا وَوَضَعَ ٱلۡمِيزَانَ ۝٧ أَلَّا تَطۡغَوۡاْ فِي ٱلۡمِيزَانِ ۝٨

Artinya:

1. *(Tuhan) yang Maha pemurah,*
2. *Yang Telah mengajarkan Al Quran.*
3. *Dia menciptakan manusia.*
4. *Mengajarnya pandai berbicara.*
5. *Matahari dan bulan (beredar) menurut perhitungan.*

6. Dan tumbuh-tumbuhan dan pohon-pohonan kedua-duanya tunduk kepada nya.
7. Dan Allah Telah meninggikan langit dan dia meletakkan neraca (keadilan).
8. Supaya kamu jangan melampaui batas tentang neraca itu. (Ar-Rahman [55] : 1-6)

Dalam *Tafsir Ibnu Abbas*, Matahari dan bulan (beredar) menurut perhitungan, yakni tempat peredaran keduanya menurut perhitungan. Ada yang berpendapat, keduanya tergantung di antara langit dan bumi. Ada pula yang mengemukakan, keduanya menurut perhitungan dan mempunyai batas waktu seperti halnya manusia.

Lebih lanjut Allah berfirman dalam ayat ke-8 menjelaskan bahwa Allah memberikan pengetahuan kepada manusia agar manusia tidak melampaui batas dengan berlebih-lebihan. Oleh karenanya banyak nasehat tentang Zakat atau sedekah untuk membagi harta yang dimiliki kepada mereka yang berhak. Dalam Surah Ar-Rahman Allah sering mengulang ayat tentang pertanyaan untuk berpikir nikmat mana yang didustakan manusia. Ayat-ayat ini mmebuat manusia lebih menggunakan pikirannya sebagai bentu edukasi memahami kenikmatan Allah yang telah diberikan kepada manusia.

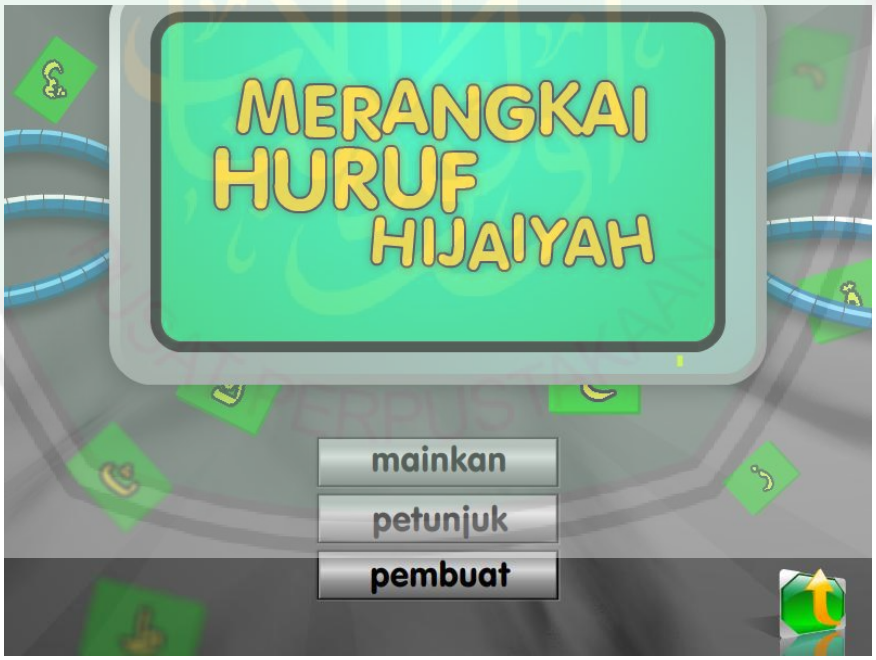
Media edukasi merupakan hal penting dalam proses mencapai tujuan edukasi, oleh karenanya mulai banyak media edukasi yang berkembang saat ini. Mulai dari pemanfaatan buku bergambar yang digunakan pada tahun 1920-an, TV edukasi, *game* edukasi hingga pemanfaatan internet sebagai basis pembelajaran. Cara edukasi juga mulai terkesan tak terasa prosesnya yakni

dengan secara tak sadar menjadikan manusia mengerti namun bukan dengan cara yang konvensional.

2.1.2 Pemanfaatan G*ame* Edukasi

Media Edukasi memang cukup banyak ragamnya, salah satunya yang berkembang cukup pesat adalah adanya *Game* edukasi. *Game* edukasi dibuat sesuai permintaan pasar yang menginginkan sebuah aplikasi yang menyenangkan namun juga ada sisi menariknya yakni Edukasi.

Game dengan motif edukasi umumnya lebih cenderug mempermudah interface kepada pemain *game* . Berikut ini contoh *Game.*



Gambar 2.1: *Game* Edukasi Huruf Hijaiyah

Perkembangan *game* telah mencapai berbagai perkembangan sejak awal awal dibuatnya. Pendekatan *game* untuk kepentingan pedidikan tersebut dapat membuka peluang bagi tim dan lintas disiplin dalam proses belajar mengajar.

Beberapa orang ahli dengan latar belakang kepakaran yang berbeda dapat saling melengkapi pengembangan media untuk menyajikan pesan dalam *game* edukasi.

Isi/konten dapat disusun oleh mereka yang memiliki kompetensi di bidang tersebut baik dari guru, dosen atau adviso dari ahli pendidikan dan psikologi, untuk menyusun metode pembelajaran, materi, modul, latihan dan evaluasi. Konten ini kemudian dikemas dan disajikan dengan memanfaatkan keahlian dari bidang desain, informatika dan mereka yang memahami produksi media baru ini. (Ramok, 2010). Penggunaan *game* semakin berkembang lebih besar lagi setelah berkembangnya dunia internet, oleh karenanya social media yang awalnya hanya sebagai media untuk bersosialisasi diri mulai berkembangan pula *game* di *social media*.

2.2 Pengertian *Game*

Permainan dalam kamus bahasa Inggris berarti *game* . Definisi *game* bisa diartikan sebagai permainan. Selain dampak positif dan negatif yang ditimbulkan, semua *game* elektronik yang dimainkan saat ini, baik itu versi *console*, HP atau PC masing-masing mempunyai elemen yang hampir sama. Sifat *game* edukasi, kadang ada pula yang memasukkan dalam gabungan *education* dan *entertainment*, yakni edukasi dan hiburan.

2.2.1 *Elemen Game*

Pada setiap *game* memiliki elemen yang hampir sama satu dengan yang lainnya. Hal ini merupakan sebuah informasi berharga dalam mengamati berbagai macam *game* .

a. Judul *Game* (*Game Title*)

Judul *game* atau *game title* ibarat nama pada manusia. Judul *game* ini umumnya wajib karena untuk dikenal orang banyak maka *game* perlu diberi nama. Judul *game* ini seperti *Football Manager* yang merupakan *game* tentang menjadi manajer sepakbola.

Dalam kehidupan kita antar manusia nama ini digunakan untuk mengenali dirinya dan juga orang lain. Pembahasan tentang nama ini disebutkan dalam Al-Qur'an seperti pada Surat Maryam [19] : 7 sebagaimana berikut.

يَٰزَكَرِيَّآ إِنَّا نُبَشِّرُكَ بِغُلَٰمٍ ٱسْمُهُۥ يَحْيَىٰ لَمْ نَجْعَل لَّهُۥ مِن قَبْلُ سَمِيًّا ۝٧

Artinya:

> *Hai Zakaria, Sesungguhnya kami memberi kabar gembira kepadamu akan (beroleh) seorang anak yang namanya Yahya, yang sebelumnya kami belum pernah menciptakan orang yang serupa dengan Dia.(Maryam [19] : 7)*

Dalam *Tafsir Ibnu Katsir* kalimat *Yā zakariyyā innā nubasy-syiruka bi ghulāmin* (hai Zakariyya, sesungguhnya Kami Memberimu kabar gembira dengan [kelahiran] seorang anak), yakni anak laki-laki. Lalu *Ismuhū yahyā* (yang bernama Yahya), yakni anak tersebut akan diberi nama Yahya karena ia memberi kehidupan pada rahim ibunya.

Kalimat *Lam naj'al lahū ming qablu samiyyā* (yang sebelumnya Kami belum pernah Memberikan nama itu kepadanya), maksudanya ialah sebelumnya Kami tidak pernah memberi Zakariyya seorang anak yang diberi nama Yahya. Ada juga yang berpendapat, belum pernah ada seseorang yang diberi nama dengan nama itu sebelum Yahya AS.

Dalam Surat Maryam [19] : 7 ini dijelaskan seorang Nabi diberi Allah anak dengan nama Yahya. Dari sini dapat ditarik kesimpulan bahwa untuk mengenali setiap manusia ataupun benda perlu nama.

b. *Credit*

Credit merupakan sebuah halaman khusus yang menampilkan informasi pembuat ataupun pengembang *game* .

c. *Intro Game*

Intro game adalah bagian awal sebuah *game* untuk dimulai, umumnya dalam sebuah *game* dimulai dengan cerita terlebih dahulu agar seorang pemain agme dapat mungkin merasakan dampak yang berbeda jika melihat atau membaca cerita *game* .

d. *Control Panel*

Control panel seperti namanya adalah sebuah panel untuk melakukan kontrol. *Control panel* ini biasanya berisikan pengaturan *volume*, musik dan berbagai hal yang dibutuhkan dalam sebuah *game*

e. Game Board

game board adalah halaman utama yang selalu aktif meski pemain *game* telah memulai permainannya. Biasanya halaman ini digunakan untuk membantu pemain *game* jika di tengan permainan membutuhkan bantuan.

f. Level game

Tiap *game* yang bertema sejenis petualangan tentu memiliki level, stage atau dengan kata lain adalah tingkatan *game* . Biasanya pemain *game* yang biasa di level tertentu ingin melanjutkan ke level yang lebih menantang dan lebih seru, inilah hal yang diperlukan untuk tetap mempertahankan pemain *game* di *game* tersebut dengan adanya level yang lebih berat jika seorang pemain *game* telah menyelesaikan level yang mudah baginya.

g. Documentation

Documentation atau daam bahasa Indonesia adalah dokumentasi yakni penyimpanan elemen *game* yang dibutuhkan, seperti cara bermain hingga penyimpanan nilai tertinggi pencapaian level *game* .

h. Copyright

Copyright merupakan tambahan dalam sebuah *game* jika ingin *game* yang disebaran mengandung hak cipta.

2.2.2 Dampak *Game*

game dengan berbagai macam tujuan dibuatnya memilki dampak tersendiri yang berbeda satu sama lain. *game* umumnya digunakan untuk menghilangkan kejenuhan atau mendapatkan kenyamanan dan kesenangan,

namun jika bermain *game* malah terjadi hal yang berdampak buruk berarti intensitas dalam bermain *game* perlu dikoreksi kembali.

2.3 Perkembangan *Game*

Game mengalami perkembangan dari waktu ke waktu, hal dengan berbagai alasan seperti mengikuti kebutuhan khalayak ramai hingga alasan perkembangan teknologi yang lebih baru jadi perlu pengembangan sistem *game* yang berbeda dari sebelumnya.

2.3.1 *Platform*

Platform merupakan pijakan atau lingkungan pengembangan *game* . *Platform* ini seperti antara lain Windows, *Console*, Linux, Mac OS dan Handled/ HP. Perkembangan *platform* yang kian berkembang maka dibutuhkan pula *game* yang berkembang di p*latform* tersebut, hal ini seperti perkembangan *game* di p*latform smartphone* seperti *Android* dan *Blackberry*.

a. Windows

Pengembang *game* di *Platform windows* ini bisa jadi cukup banyak karena perkembangan ini disertai pengguna plataform ini untuk kebuthan lain selain *game* seperti bekerja, sekolah atau sekedar pemakai pasif.

b. Console

Contoh g*ame* console yang cukup terkenal adalah playstation dengan berbagai versi perkembangannya sampai saat ini.

c. Linux

Linux yang memiliki tema *operating system* yang gratis juga cukup banyak pengembang *game* di linux.

d. Mac

Mac atau atau dikenal pula dengan nama lain machintos juga memiliki *game* yang dikembangkan.

e. HandHeld/HP

Perkembangan *game* di *Platform* HP (*handphone*) juga cukup banyak saat ini, hal ini seiring banyaknya minat pada pemrograman Java untuk mobile.

2.3.2 *Genre* atau Jenis *Game*

Membangun sebuah *game* layaknya sebuah lagu yang memilki genre atau jenis *game* . Berikut ini kelompok *genre game* yang berkembang menjadi beberapa kelompok yaitu:

a. *Sport Game*s: Jenis ini merupakan simulasi kegiatan atau permainan olahraga seperti Sepakbola, *badminton*, basket, catur dan olahraga lainnya. Menurut Samuel Henry (2005 : 53) selama game itu mengetengahkan genre olehraga maka disebut *genre sport*. Tom Meigs (2003 : 109) menjelaskan aspek desain untuk *Sport Games* adalah:

- *Motion capture versus* animasi tangan
- Peraturan yang telah ditetapkan
- Simulasi *versus* fantasi
- Lisensi

- *Heavy tuning*
- Script bertahan
- *Player anda arena look and feel*
- ststistik

Gambar 2.2: FIFA 13 – contoh *Sport Games*

b. ***Fighting Games***: *Game* jenis ini adalah *game* bersifat ketangkasan, menurut Samuel Henry (2005 : 48) sesuai namanya game ini mengetengahkan pertarungan. Game ini memberikan kesempatan untuk pemain dalam bertarung menggunakan berbagai kombinasi gerakan dalam pertarungan. Jenis game genre ini diantaranya seperti *Tekken 4, Street Fighter 2* dan *Kungfu*.

c. ***Puzzle Games*:** *Game* jenis ini merupakan genre atau jenis *game* yang mengandalkan puzzle atau teka-teki untuk dipecahkan. Menurut Samuel Henry (2005 : 46), game jenis ini memberikan tantangan kepada

pemainnya dengan cara menjatuhkan sesuatu dari sisi sebelah atas ke bawah.

Gambar 2.3 : *Rubik's Cube* – Contoh *Puzzle Games*

d. ***Real-Time Strategy Games*:** *Game* ini merupakan *game* yang senantiasa menggunakan strategi dalam permainannya. Samuel Henry (2005 : 50), pada jenis game ini pemain harus melakukan berbagai gerakan sesuai dengan strategi pembuat game. Game jenis genre ini seperti contohnya *Warcraft* di mana pemain harus secepat mungkin mengumpulkan emas untuk membangun kekuatan.

Gambar 2.4 : *FootBall Manager* – Contoh *Real-Time Strategy Games*

e. ***Role Playing Games (RPG):*** *Game* jenis ini hampir sama dengan *game* petualangan. Menurut Samuel Henry (2005 : 51), genre game ini pemain berperan menjadi sebuah karakter; pemain menjalankan peran dengan berbagai atribut seperti: kesehatan, ientelegensi, kekuatan dan keahlian. Salah satu game yang tekenal dengan RPG pada masa awal adalah *Ultima*, kini game ini berkembang menjadi beberapa jenis variasi RPG seperti *action RPG* dengan contoh game *Legacy Of kain, Blade Sword dan Beyond Divinity*.

f. ***First Person Shooter (FPS):*** Menurut Samuel Henry (2005 : 51), disebut First Person Shooter karena pandangan pemain adalah pandangan pertama (fisrt person), kita melihat tampilan di layar seperti kita melihat dari mata kita sendiri. Seperti namanya Shooter, game ini mengandung tentang baku tembak dengan mengutamakan kecepatan gerakan dalam permainan.

g. ***First Person 3D Vehicle Based:*** Menurut Samuel Henry (2005 : 51), genre game ini sama dengan genre FPS hanya bedanya pandangannya bukan dari mata tetapi dari sudut pandangan kendaraan atau mesin yang

dinaiki. Gerakan mungkin akan lebih lambat karena pemain game berada di dalam kendaraan.

h. ***Third Person 3D Games*****:** Menurut Samuel Henry (2005 : 51), genre game ini sama dengan FPS anya berbeda sudut pandanganya. Kalau FPS melihat dari sudut pandang orang pertama maka genre game ini melihat dari sudut pandanga orang ketiga.

game jenis ini contohnya seperti *Counter Strike* yang sebelumnya dimasukkan dalam genre game FPS, namun *Counter Strike* bisa juga dimasukkan dalam jenis ganre ini karena pemain bisa melihat peran yang dimainkannya sendiri.

2.4 *Game Engine*

game engine merupakan alat pendukung membuat *game* . Komponen yang umumnya dimiliki oleh *game engine* adalah sebagai berikut:

a. ***Graphic Engine****:* Berfungsi untuk megambar grafis obyek 2D atau 3D ke dalam layar *game* .

b. ***Physic Engine****:* komponen ini mempunyai fungsi mendeteksi benturan fisik dalam *game* .

c. ***Platform Abstraction***: Berfungsi abstraksi dari *Platform* .

d. ***Integrated Development Environment*** **(IDE)**: komponen IDE berfungsi mempercepat proses pembuatan *game*

2.4.1 GTGE *Game Engine*

Menurut Andi Taru (2010 : 15), GTGE (*Golden T game Engine*) merupakan game engine bebasis bahasa pemrograman java. GTGE *Game* Engine dapat membantu kita mempermudah pengembangan *game* 2 Dimensi. Karena dengan *Game* Engine kita akan dengan mudah melakukan pembuatan *game*

Gambar 2.5 : Logo GTGE

Dengan GTGE *game* engine untuk membuat *game* sederhana akan lebih mudah dengan adanya kemudahan masukan *sprite*/gerakan atau *collision*/tumbukan pada *game* .

2.5 *Game* untuk Edukasi dalam Islam

Hadits yang diriwayatkan oleh Bukrori dan Muslim ini bisa jadi acun dalam pemanfaatan *game* untuk Edukasi, *"Permudahlah jangan dipersulit, berilah kabar gembira dan jangan menakut nakuti"(Shahih Bukhari-Muslim).*

Dalam hadits ini adanya anjuran mempermudah tentu harus ada batasannya pula, *game* sebagai media untuk eduksi atau pembelajaran asal tak mengganggu atau malah melanggar syariat tentu menjadi hal yang dilarang.

2.6 ANTLR

ANTLR digunakan dalam penelitian ini sebagai penyeleksi kebenaran tata bahasa pemrograman. ANTLR merupakan kepanjangan sebenarnya dari *ANother Tool for Language Recognition*. ANTLR adalah generator *Parser* yang dapat di gunakan untuk membaca, mengolah, melaksanakan, atau menerjemahkan teks terstruktur atau file biner. Ini banyak digunakan dalam dunia akademis dan industri untuk membangun segala macam bahasa, alat, dan kerangka kerja. (Terence Parr, 2012 : xi).

Gambar 2.6: Logo ANTLR

2.6.1 Kegunaan ANTLR

ANTLR dapat membantu membuat sebuah DSL (*Domain Spesific Language*) atau bahasa yang spesifik untuk hal tertentu. Penjelasan di dalam situs resminya antlr.org; Twitter dalam hal mengolah data untuk pencarian menggunakan ANTLR untuk *parsing query*, dengan lebih dari 2 milyar permintaan per hari. Bahasa untuk Hive, gudang data dan sistem analisis untuk Hadoop, keduanya menggunakan ANTLR. Machina lex menggunakan ANTLR untuk ekstraksi informasi dari teks-teks hukum. Oracle menggunakan ANTLR dalam SQL *Developer* IDE dan alat migrasi mereka. NetBeans IDE mengurai

C++ dengan ANTLR. Bahasa HQL dalam rangka pemetaan objek-relasional Hibernate dibangun dengan ANTLR.

Selain dari nama besar, proyek profil tinggi, ANTLR juga dapat membangun segala macam alat yang berguna seperti pembaca file konfigurasi, warisan kode konverter, wiki *markup* penyaji, dan *JSON Parser*.

Dari deskripsi bahasa formal yang disebut tata bahasa, ANTLR menghasilkan *Parser* untuk itu bahasa yang secara otomatis dapat membangun pohon *parse (tree parse)*, yang merupakan struktur data yang mewakili bagaimana tata bahasa cocok dengan input. ANTLR juga secara otomatis menghasilkan pejalan kaki pohon yang dapat Anda gunakan untuk mengunjungi node dari pohon-pohon untuk mengeksekusi kode aplikasi-spesifik.

Ada banyak orang menggunakan ANTLR untuk mempermudah aktivitasnya hal ini juga yang mengakibatkan banyak yang melakukan download ANTLR hingga ribuan kali dan itu termasuk pada semua Linux dan distribusi OS X. ANTLR banyak digunakan karena mudah untuk memahami, kuat, fleksibel, menghasilkan output terbaca-manusia, dilengkapi dengan sumber lengkap di bawah lisensi BSD, dan secara aktif didukung.

2.6.2 Alur ANTLR

ANTLR sebagai alat untuk membantu memudahkan pekerjaan menciptakan sebuah DSL (*Domain Spesific Language*) atau bahasa yang spesifik untuk hal tertentu memiliki alur dimulai dari membuat sebuah grammar.

Grammar dalam ANTLR ini menggunakan bahasa *Extended Backus–Naur Form* (EBNF). Dalam ilmu komputer, BNF (*Backus Normal Form* atau Formulir *Backus Naur*-) menurut wikipedia adalah salah satu dari dua teknik notasi utama untuk tata bahasa bebas konteks, sering digunakan untuk menggambarkan sintaks dari bahasa yang digunakan dalam komputasi, seperti bahasa pemrograman komputer, format dokumen, instruksi set dan protokol komunikasi, teknik utama lainnya untuk menulis tata bahasa bebas konteks adalah bentuk van Wijngaarden. Mereka diterapkan dimanapun deskripsi yang tepat dari bahasa yang diperlukan: misalnya, dalam spesifikasi bahasa resmi, dalam buku pedoman, dan buku mengenai teori bahasa pemrograman.

Dalam Grammar inilah Metode *Generating Tree* dilakukan dengan berbagai macam alur. Cara membuat Grammar ini agar terlihat proses Generating Tree dapat menggunakan alat pendukung seperti ANTLRWorks atau bisa juga Netbeans.

Dalam versi ANTLR sebelum generasi ke-4 ANTLRWorks merupakan alat terpisah layaknya editor compiler, namun dalam versi ANTLR generasi versi yang ke-4 ANTLRWorks sudah berupa plugin yang masuk dalam Netbeans. Namun masuknya plugin ANTLRWorks ini hanya bisa dilakukan pada Netbeans versi 7.3 sebagai Netbenas penerima plugin ini.

Setelah *Grammar* telah benar sesuai keinginan, maka tahap selanjutnya adalah melakukan *Generate Recognizer* untuk menghasilkan program pendukung yakni Token, *Lexer* dan *Parser*. Untuk versi ANTLR generasi ke-4 terdapat pula program *Visitor*, *Listener* dan juga program .dot.

Untuk proses dimulai dari Token. Token adalah kumpulan karakter yang menyusun suatu bahasa. Dalam ilmu bahasa, token dapat disamakan dengan kata. Token- token ini dibentuk dari karakter - karakter berdasarkan suatu aturan tertentu yang disebut sebagai *Lexer rule*.

Kode sumber bahasa pemrograman merupakan kumpulan karakter-karakter yang masih belum bermakna. Untuk memperoleh maknanya karakter-karakter itu harus dikelompokan. Pengelompokan ini mempunyai aturan tertentu sehingga akan menghasilkan tipe tipe token , misalnya: *keywords, operator, identifier* dan komentar. Token- token ini kemudian dikirimkan kepada *Parser* untuk kemudian digabungkan dengan token - token lainnya untuk membentuk *statement* (kalimat) (Grune 2001). Token ini akan diproses ke tahap berikutnya yakni dengan menggunakan *Lexer*.

Lexer melakukan proses dengan mengelompokkan karakter-karakter yang mempunyai makna hingga membuang karakter-karekter yang tak mempunyai arti atau makna. Hal ini karena dalam sebuah tata bahasa diperlukan sebuah makna yang jelas sehingga dapat diketahui maksud tujuan penyampaian bahasa tersebut.

Dari *Lexer* akan menghasilkan aliran karakter yang menjadi aliran token untuk diperoses pada *Parser*.*Parser* lalau memperoses aliran karakter dari *Lexer* ini dengan menggunakan tata bahasa tertentu yang menunjukkan aliran token tersebut benar atau tidak.

2.7 *Context Free Grammar*

Menurut Bambang Hariyanto (2004 : 233) semula *Context Free Grammar* (CFG) ditemukan untuk membantu menspesifikasikan bahasa manusia dan ternyata sangat cocok untuk mendefinisikan bahasa komputer, memformulasikan pengertian parsing, menyederhanakan penerjemah bahasa komputer dan aplikasi-aplikasi pengolah string lainnya. (*Context Free Grammar* (CFG) adalah tata bahasa yang hampir sama tujuannya pada tata bahasa reguler seperti bahasa Inggris, bahasa Indonesia atau bahasa Arab yang menghasilkan untaian sebuah bahasa.

Terinspirasi dari bahasa natural manusia, ilmuwan-ilmuwan ilmu komputer yang mengembangkan bahasa pemrograman, turut serta memberikan tata bahasa (pemrograman) secara formal. Tata bahasa ini diciptakan secara bebas konteks dan disebut CFG (*Context Free Grammar*). Hasilnya, dengan pendekatan formal ini, kompiler suatu bahasa pemrograman dapat dibuat lebih mudah dan menghindari ambiguitas ketika *parsing* bahasa tesebut.

Semula CFG ditemukan untuk membantu menspesifikasikan bahasa manusia dan ternyata sangat cocok untuk mendefinisikan bahasa komputer, menformulasikan pengertian *parsing*, menyederhanakan penerjemahan bahasa komputer dan aplikasi-aplikasi pengolahan string lainnya. CFG digunakan untuk menspesifikasikan struktur sintaks bahasa pemrograman serta beragam basis data.

CFG adalah sekumpulan berhingga variabel yang juga disebut nonterminal atau kategori sintaks, dimana masing-masing mempresentasikan bahasa. Bahasa-bahasa yang direpresentasikan dengan variabel-variabel itu yang dideskripsikan

secara rekursif dalam bentukan lain dan symbol-simbol primitif disebut terminal. Aturan-aturan yang berhubungan dengan variabel-variabel itu disebut produksi.

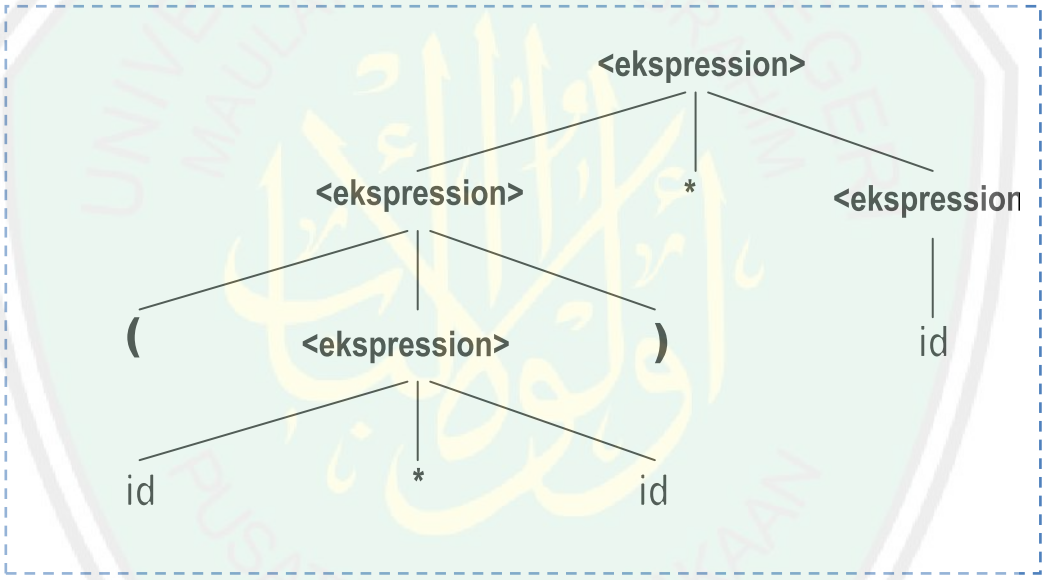
Contoh penggunaan CFG adalah pada dokumen HTML yang berbentuk bagus selalu menyertakan tag-tag berpasangan secara lengkap. Contoh pada Struktur HTML sebagai berikut:

```
<HTML>
<HEAD>
<TITLE> Komunitas Pecinta Teknologi Komputer </TITLE>
<H1>Home Page bagi orang yang terus ingin maju</H1>
<H2>Komunitas Informatika yang tak hanya dari 1 sisi pandangan</H2>
<P>
<CENTER>
<IMG src=”kptk_view.jpg”>
</CENTER>
</P>
</HEAD>
</HTML>
```

Pada dokumen HTML yang berbentuk bagus, ekspresi <HTML> selalu diikuti </HTML>, <HEAD> selalu akan diikuti dengan </HEAD>, lalu <CENTER> selalu akan diikuti pula dengan </CENTER>, dan begitulah seterusnya. Pola-pola ini serupa pasangan kurung. Dengan demikian, pengenalan string-string yang termasuk dalam *Context Free Grammar* (CFG) merupakan salah satu tugas yang dilaksanakan *web browser* (Hariyanto, 2004 : 234).

2.8 Generating Tree

Generation Tree juga disebut diagram *parsing*, Pohon Pembangkitan atau phrase marker (Hariyanto, 2004 : 240). Pada *Generation* atau *Generating Tree* ini berguna untuk memecah grammar dari bahasa pemrograman menjadi diturunkan lebih kecil, hal ini digunakan untuk mengecek kebenaran dengan yang diminta pada *Java Coding game* ini. *Generating Tree* digunakan dalam pembentukan *Grammar* di ANTLR dengan bahasa EBNF.



Gambar 2.7: *Generating Tree*

2.9 Bahasa *Pemrograman* Java

Menurut Rangsang Purnama (2003 : i), Java adalah bahasa pemrograman yang sedang naik daun. Java merupakan salah satu bahasa pemrograman yang cukup menjadi idola di waktu-waktu ini pada pembelajaran di komunitas programmer. Walau mulai muncul tantangan dari Ruby yang dikembangankan di Jepang, java yang namanya diambil dari kopi jawa sepertinya tak dapat

disingkirkan di dunia saat ini. Salah satu buktinya pengembangan java telah sampai pada alat komunikasi antar sesama manusia.

Setelah mengetahui betapa pentingnya java dan itulah kenapa mahasiswa pada jurusan Teknik Informatika di sebagian perguruan tinggi semenjak awal telah dicekoki oleh pengenalan terhadap java. Algoritma, logika, *psydocode* hingga pada aksi konkret hasil dari pemrograman java. Oleh karenanyalah, kemampuan *coding* seorang programmer sangat penting, karena menentukan kualitas hasil *project* mereka juga. Mengenai hal ini, Romi Satria Wahono seorang praktisi komputer pernah mengkritisi bagaimana lemahnya *coding* lulusan infomatika, padahal entah anda dari teknik, manajemen, sistem informasi informatika pengetahuan atau kuatnya *coding* sangatlah penting.

BAB III

ANALISIS DAN PERANCANGAN SISTEM

3.1 Analisis Sistem

3.1.1 *Storyboard Game*

Storyboard Game adalah hal yang dibutuhkan pertama kali dalam merancang sebuah aplikasi Game. Untuk jenis game versi pendek yang tetap cerita seperti Game ular tangga, catur, *tetris, winning eleven*, dan tata bata. Sedangkan alur cerita game yang panjang dengan berbagai tantangan membutuhkan waktu yang relatif panjang pula untuk menyelesaikan Game, hal ini seperti contohnya Game *Football Manager, Gran Turismo*.

a. Judul Game

Judul Game adalah: "*Java Coding Game*"

b. Deskripsi Game

Game ini merupakan game yang mengkolaborasikan 2 bagian halaman game di tiap level, yakni tentang kesabaran dan keuletan menemukan kunci untuk masuk ke pintu *exit* serta sisi edukasi *coding* setelah melewati pintu *exit*.

c. Cerita Singkat

Game ini dimulai dari cerita seorang tokoh dalam game yang tiba di setiap daerah yang mana tantangan di setiap daerah yang dia kunjungi ini adalah menemukan kunci-kunci. Selain kunci ada pula poin dan tambahan nyawa dalam game. Semakin bertambah nyawa akan semakin bisa lebih lama pula untuk bertahan dalam mengarungi game ini. Hal ini karena setiap tokh dalam

game tertabrak oleh musuh (*Enemy*), waktu habis untuk menuntaskan tantangan dalam tiap level, atau hal lai yang menyebabkan terjadinya kematian maka cadangan nyawa yang masih ada akan digunakan untuk bisa bermain kembali.

Pada halaman di tiap level pemain game akan dihadapkan pada tantangan untuk menemukan kunci-kunci ditambah lagi tantangan untuk mengatasi musuh (*Enemy*) serta adanya poin untuk menambah pundi poin sehingga di akhir game nanti akan ditentukan peringkat pemain game berdasarkan skor tertinggi (*hi-score*)

Dalam setiap level setelah melewati halaman game untuk menemukan kunci-kunci, meningkatkan poin higga mengatasi musuh (*Enemy*) untuk menemukan pintu *exit* maka halaman di irisan halaman game kedua tiap level adalah pada halaman memecahkan tantangan tentang kode-kode Java. Setelah memecahkan tantangan ini, maka akan berlanjut ke level berikutnya.

d. Skenario Game

Musuh (*Enemy*) menyebar tantangan dan menyebar untuk menghalangi pemain dalam game mendapatkan poin, kunci-kunci, bonus hingga pintu *exit.* Pemain dalam game berusaha melewati musuh dengan menemukan kunci-kunci, tambahan poin, bunus untuk menemukan pintu *exit* setelah itu akan masuk pada halaman tantangan tentang kode-kode java untuk dipecahkan. Perjalanan game akan lebih cepat dengan menekan tombol *shift* untuk pilihan *Turbo Mode.*

Pemain dalam game akan mati jika menabrak musuh (*Enemy*) atau kehabisan waktu untuk melewati tantangan di tiap level. Permainan masih dapat dilanjutkan jika pemain dalam game masih memiliki nyawa, akan terjadi *game over* jika nyawa sudah habis yang kemudian akan masuk ke halaman *hi-score* game untuk mengisi nama jika *score* atau poin yang dikumpulkan masuk jajaran *top hi-score.*

Jika permainan disudahi karena adanya permintaan dari pemain menemekan tombol Q untuk keluar (*Quit*), maka pemain akan diarahkan ke halaman *hi-score* jika poin atau *score*-nya masuk jajaran top *hi-score*. Permainan juga dapat di-ulang kembali atau *restart* jika pemain menekan tombol R (*restart*).

Gambar 3.1: Plot *Storyboard Game*

e. Karakter Tokoh

Gambar Tokoh	Nama Tokoh	Karakter Tokoh
	Jacod	Jacod merupakan tokoh sentral dalam game pada stage pencarian kunci dan menemukan pintu. Jacod akan hampir ada di setiap level di game ini.

	Master	Master adalah tokoh *Enemy* (musuh) dalam game ini. Muncul pada level-level awal game dengan pergerakan 1 arah bolak-balik. Jika Jacod mengenainya maka akan mengurangi 1 nyawa dalam game.
	Glany	Glany adalah tokoh *Enemy* (musuh) dalam game ini. Pergerakannya seperti peran nenek sishir yang menggurakan sapu yang bisa terbang.
	Toy	Toy adalah tokoh *Enemy* (musuh) dalam game ini. Peran Toy bergerak berdasar pada keinginan pembuat game, dalam level awal pergerakannya satu arah bolak-balik menjaga agar Jacod tidak lewat.
	Rob Do	Rob Do adalah tokoh *Enemy* (musuh) dalam game ini. Pergerakannya bisa menyilang mengikuti perintah pada level builder, perannya menjaga wilayah tertentu untuk menghalangi Jacod.
	Bado	Bado adalah *Enemy* (musuh) dalam game ini. Bado bergerak mengikuti perintah pada pembuatannya, Bado memiliki peran menjaga daerahya dari pencapaian yang dilakukan oleh Jacod.
	Baromo	Baromo adalah tokoh *Enemy* (musuh) dalam game ini. Baromo memiliki bentuk agak besar untuk berperan menjaga wilayah dari perjalanan Jacod menemukan kunci dan pintu keluar.

	Dinocro	Dinocro adalah tokoh *Enemy* (musuh) dalam game ini. Bentuk Dinocro hampir seperti dinosaurus. Dinocro memiliki peran penjagaan daerah yang pergerakannya lebih luas

Tabel 3.1: Karakter Tokoh *Stage* Halaman Awal

3.1.2 Kebutuhan Sistem

Setiap aplikasi selalu dibangun dari kebutuhan sistem yang membangun aplikasi tersebut; Meigs (2003 : 2) menjelaskan dalam tahapan awal pembuatan Game yakni *Previsualization Process* yang di dalamnya dimulai dengan kontruksi level game, yaitu dengan mengetahui lingkungan dari kebutuhan sistem untuk membangun game.

Game yang lebih menitik beratkan pada edukasi maka berbeda konsep tujuan pembuatannya dengan game yang dibuat hanya sebagai alat atau media untuk memberi kepuasan. Game dengan konsep edukasi memiliki tujuan utama pada bidang edukasi penggunanya dengan jalan yang lebih santai, tegang hingga nyaman yang tak terasa menikmati game sekaligus memperoleh atau menguatkan keilmuan.

Java Coding Game dibangun dengan tujuan untuk memberikan edukasi tentang bahasa *Grammar* pemrograman yang ternyata hampir sama dengan konsep di kehidupan kita sehari-hari, sehingga pemain game akan merasa mudah mengingat dan menerapkan pemrograman karena ternyata setiap manusia pun besinggungan dengan logika pemrograman dalam kehidupan ini.

a. *Grammar*

Grammar adalah sistem matematis untuk mendefinisikan bahasa. Bahasa yang didefinisikan oleh *Grammar* adalah himpunan string yang hanya berisi terminal dan diturunkan mulai dari simbol tertentu yang dikhususkan yang disebut S atau simbol mula (*starting symbol*) (Harianto, 2004 : 64).

Grammar adalah elemen utama dalam kebutuhan sistem pertama kali, hal ini karena dari *Grammar* inilah inti nilai edukasi dalam *Java Coding Game* ini. Dari *Grammar* yang merupakan tata bahasa atau aturan bahasa, maka akan ditemui bahasa pemrograman yang benar sesuai kaidah *Grammar* pemrograman Java.

Penggunaan *Grammar* pada Game ini menggunakan Bahasa EBNF atau *Extended Backus-Naur Form*. Bahasa EBNF dengan metode Generating Tree diinisialisasikan dalam pemrograman.ANTLR membantu pemebentukan *Grammar* ini dengan menyediakan *resource* sebagai *parse generator*.

b. Pengenalan *Grammar*

Grammar yang sebagai acuan aturan tata bahasa perlu penyambung untuk mengenali mereka yang memakai suatu bahasa, hal ini karena berkaitan erat tentang aturan bahasa untuk mepermudahkan komunikasi.

Seperti layaknya komunikasi di dunia nyata yang mana setiap manusia di suatu negeri memiliki aturan bahasa tersendiri di dalam kelompoknya. Aturan bahasa yang menjadi alat komunikasi ini turun temurun dijaga hingga ada yang pula merumuskan dalam sebuah tata bahasa atau biasa disebut *Grammar*.

Grammar satu bahasa dengan bahasa lainnya kadang berbeda namun kadang pula sama dalam polanya. Oleh karenanya *Grammar* di dalam bahasa Inggris kadang ada yang sama dengan bahasa Indonesia. Hal ini seperti penerapan Subyek, Obyek dan Predikat dalam bahasa Indonesia yang hampir sama maksuda kaidah letak atau posisinya dalam *Grammar* bahasa Inggris.

Di dalam bahasa pemrograman juga memiliki aturan bahasa *Grammar* seperti pula aturan *Grammar* di bahasa komunikasi manusia. Hal ini tak lepas dari pembuata bahasa pemrograman adalah manusia yang memebawa kaidah dalam kehidupan yang nyata di dalam aturan tata bahasa pemrograman agar lebih mudah dimengerti.

Java Coding Game menggunakan pengenalan ini dimulai pada kebutuhan untuk mengenali kebenaran *Grammar* dan menganalisanya menggunakan metode *Generating tree*. *Generating tree* dapat dilihat pada syntax diagram di netbeans melalui plugin tambahan yang dikembangkan oleh ANTLR dengan Tunnel visions untuk menghasilkan ANTLRWorks versi 2 karena versi sebelumnya ANTLRWorks berdiri sendiri sebagai sebuah alat untuk mengenali *Grammar* hingga melakukan generate code untuk menghasilkan *Lexer* dan *Parser*.

3.2 *Finite State Machine* (FSM)

Menurut Iwan Setiawan (2006 : 1), *Finite State Machines* (FSM) adalah sebuah metodologi perancangan sistem kontrol yang menggambarkan tingkah laku atau prinsip kerja sistemdengan menggunakan tiga hal berikut: *State*

(Keadaan), *Event* (kejadian) dan *Action* (aksi). FSM cukup banyak dipakai sebagai basis perancangan aplikasi yang mempunyai kontinuitas seperti *Game*.

Gambar 3.2: *Finite State Machine Java Coding Game*

3.3 Perancangan Sistem

Sebuah sistem yang besar umumnya diperlukan perancangan pembangunan sistem yang baik untuk dapat bisa digunakan dengan baik untuk khalyak ramai. Setelah melakukan analisa sistem dengan melihat alur game hingga kebutuhan sistem pembentuk maka selanjutnya adalah dengan perancangan sistem.

Sebelum masuk ke tahapan game, yang diperlukan adalah melakukan analisis hingga perancangan sistem yang baik. Hal ini karena sistem yang akan

diintegrasikan dengan game nantinya merupakan sebuah sistem yang bisa bediri sendiri, namun karena alasan untuk melakukan perubahan agar lebih bersifat edukasi maka media *Game* diperlukan untuk melakukan pembelajaran.

Setiap sistem akan dimulai dengan dasar sistem atau bisa disebut juga otak sistem yang mana apapun yang ada di dalam sistem tersebut letak yang paling penting ada di dalam otaknya ini. Begitu pula dengan *Game* Edukasi *Java Coding Game* memiliki otak yang merupakan inti untuk menjalankan game. Perancangan sistem awal seperti merujuk pada analisa sistem untuk dikembangkan sebagai sebuah desain sistem yang baik agar bisa digunakan sebagai rujukan untuk implementasi hingga pengujian sistem lebih lanjut.

Gambar 3.3: Alur Sistem Sebelum *Game*

Gambar di atas merupakan gambaran alur sistem sebelum dimasukkan dalam ruang lingkup *Game*. Dimulai dari inialisasi *Grammar* yang dalam contohnya adalah inialisasi karakter sp = 100;

Sistem kemudian melakukan proses *Language Recognizer* yang sumber datanya diambil dari *Grammar* yang berisi karakter. Secara mendalam *Language Recognizer* ini melakukan proses *Lexer* lalu *Tokenizing* hingga *Parser*.

Setelah melewati proses pada *Language Recognizer* selanjutnya akan menghasilkan *parse tree* yang berisikan penjelasan lebih dalam tentang

percabangan pohon yang merupakan gambaran dari inialisasi *Grammar*. Melihat lebih dalam sebuah program dibentuk melalui proses ini akan dapat diketahui bahwa sistem seperti bahasa pemrograman dibentuk dari kumpulan kata untuk membantuk kalimat bahasa pemrograman yang dikenali oleh *Grammar* bahasa pemrograman.

3.3.1 Perancangan *Grammar*

Grammar yang merupakan inti dari sistem yang dibangun ini merupakan awal untuk melakukan perancangan sistem. Seperti yang telah dijelaskan di atas tadi bahwa setiap sistem mempunyai otak sistem yang merupakan landasan terbentuknya sistem tersebut. Perancangan *Grammar* merupakan inti atau otak utuk melakukan pembangunan sistem Game berbasis edukasi pemrograman bahasa Java ini.

Seperti merujuk pada gambar alur sistem sebelum game, perancangan *Grammar* merupakan langkah awal sebelum melakukan perancangan lain dalam membentuk sistem. Perancangan *Lexer*, *Token*s hingga *Parser* dimulai dari perancangan pada *Grammar*. Hal ini karena *Grammar* akan menjadi rujukan utama tentang aturan yang ada dalam langkah sistem berikutnya.

Grammar yang dibangun dalam Game Edukasi *Java Coding Game* ini merupakan *Grammar* dengan basis bahasa EBNF (*Extend Back naous Form*). EBNF merupakan pengembangan lanjut dari bahasa sebelumnya yakni BNF (*Back Nous Form*).

Konsep *Generating Tree* akan terlihat pula di dalam inialisasi program *Grammar*. Hal ini bisa dilihat nantinya pada proses inialisasi rule atau aturan yang menggunakan konsep bercabang dalam istilahnya di dalam pemrograman *Grammar* EBNF.

```
Grammar ArrayInit;

init :'{'value(','value)*'}';

value : init
      | INT
      ;

INT: [0-9]+;
WS : [\t\r\n]+->skip;
```

Kode Program 3.1: Contoh *Grammar ArrayInit*

Kode Program di atas merupakan contoh kode program *Grammar* yang dalam hal ini mempunyai nama untuk program `Grammar ArrayInit`. Judul dan merupakan nama *Grammar* ini adalah *ArrayInit*. Seperti dalam tata bahasa pemrograman java yang nama class harus pula ditulis pada struktur pemrograman, maka pun begitu pula dengan penulisan kode program untuk *Grammar*.

Setelah melakukan inialisasi nama *Grammar*, langkah selanjutnya dalam penulisan *Grammar* adalah melaukan inialisasi pada rule atau aturan *Grammar* lebih lanjut. Seperti pada contoh inialisasi `init :'{'value(','value)*'}';`

`init :` merupakan inialisasi bahwa aturan *Grammar* yang dibuat ini bernama init. Sesuai bahasa pemrograman EBNF sebagai pembentuk *Grammar* ini, `init :` nantinya akan dipakai sebagai inti dalam *Grammar*. Hampir semua method akan dipanggil melalui program utama ini.

`init :` akan memanggil rule atau aturan di dalam aturan *Grammar* yang ada di bawahnya untuk inialisasi. Namun pemanggilan ini juga harus disertai inialisasi pula pada aturan turunan tersebut agar dapat dipanggil di aturan utama dalam bingkai `init :`

Aturan `init :` sendiri berisikan aturan`'{'value(','value)*'}';` yang merupakan method yang dipanggil dari aturan dibawah init. Aturan `'{'value(','value)*'}';` menginialisasikan sebuah aturan yang mana ada sisipan aturan `value`.

Aturan `'value` akan diinialisasi lagi pada setelah inialisasi init. Aturan init ini juga berisikan kode program kuruang kurawal ("{}") dan koma (","). Secara luas method init ini beriskan aturan tentang nilai `value`.

```
...
      value : init
            | INT
            ;
...
```

Kode Program 3.2: Aturan `value`

Kode program `value` berisikan aturan `value` . aturan ini mempunyai isi INT. Isi dari INT nantinya akan dijelaskan kembali pada aturan program *Grammar* di baris berikurtnya

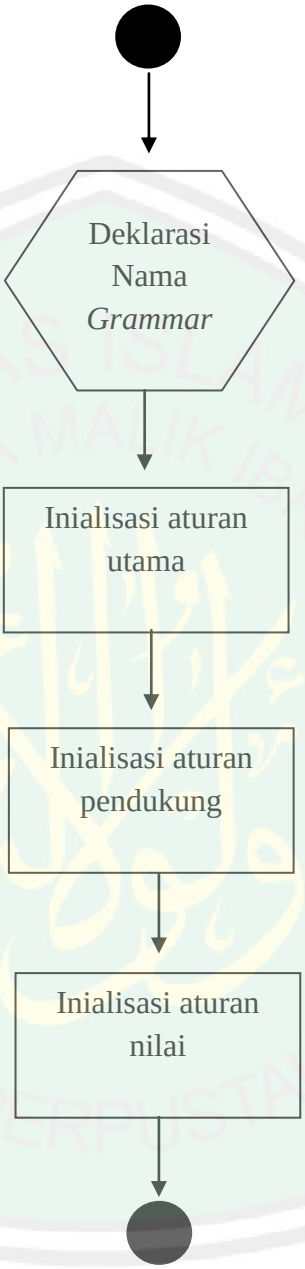
```
...
      INT: [0-9]+;
      WS : [\t\r\n]+->skip;
```

Kode Program 3.3: Aturan Tambahan *Grammar ArrayInit*

Pada kode program yang berisikan aturan tambahan pada gramar *ArrayInit* ini merupakan aturan terakhir dalam sistem *Grammar*. Hampir setiap *Grammar* yang diinialisasikan dengan EBNF untuk keperluan bahasa menggunakan ANTLR menggunakan cara penulisan dengan penjelasan rinskas aturan tambahan *Grammar* ini untuk menunjang aturan *Grammar* secara keseluruhan.

Aturan tambahan ini beriskan `INT: [0-9]+;` yang maksudnya adalah nilai atau maksud dari method INT adalah bilangan 0 (nol) hingga 9 (sembilan) yang mana modifikasi dengan memilih kata sesuai kebutuhan dalam tahapan *Lexer*, parsing hingga penentukan kebenaran inputan berdasarkan *Grammar* ini.

Pada aturan terakhir yakni `WS : [\t\r\n]+->skip;` merupakan aturan yang mendeskripsikan tentang menentukan aturan WS atau White Space untuk `[\t\r\n]`.



Gambar 3.4: Proses pembentukan *Grammar*

Proses pembentukan *Grammar* seperti digambarkan pada Gambar merupakan bentuk alur sederhana dari proses pembuatan dan alur proses dari aturan yang ada dalam *Grammar*.

Proses yang diawali oleh pemberian nama *Grammar* dilanjutkan dengan inialisasi yang saling mendukung dalam sebuah aturan. Hal ini bisa dilihat dari gambar bahwa inialisasi aturan utama mengambil pada inialisasi yang diberikan pada aturan pendukung serta mengambil pula aturan pada inialisasi aturan nilai seperti umumnya WS atau *white space*.

3.3.2 Perancangan *Lexer* dan *Parser*

Hal yang cukup penting setelah melakukan perancangan *Grammar* adalah merumuskan bagaimana *Lexer*, *Token* serta *Parser* untuk selanjutnya dapat didapat aturan pembangunan sistem untuk mengecek kebenaran *Grammar* yang dimasukkan dengan proses bantuan *Lexer* dan *Parser*.

Token merupakan kumpulan karakter yang menyusun suatu bahasa. Dalam ilmu bahasa, *Token* dapat disamakan dengan kata yang berasal dari kumulan huruf. *Token- Token* ini dibentuk dari karakter - karakter berdasarkan suatu aturan tertentu yang disebut sebagai *Lexer* rule. Kode sumber bahasa pemrograman merupakan kumpulan karakter-karakter yang masih belum bermakna. Untuk memperoleh maknanya karakter-karakter itu harus dikelompokan. Pengelompokan ini mempunyai aturan tertentu sehingga akan menghasilkan tipe tipe *Token* , misalnya: *keywords, operator, identifier* dan komentar (Fathan, 2007 : 4).

Dari *Token* yang didapat dari aturan *Grammar* maka proses *Lexer* dan *Parser* akan berjalan saling mendukung.Potongan *Token* yang ibarat “kata” akan

diproses oleh *Lexer* untuk mendapatkan makna dari potongan kata sesuai aturan spesifikasi dalam *Grammar*.

```
WS=5
INT=4
T__1=2
T__0=3
T__2=1
'}'=3
```

Kode Program 3.4: *Token ArrayInit*

Seperti dalam Kode program contoh pada *ArrayInit* yang aturan *Grammar* telah dibahas di atas tadi, maka aturan dalam *Token* adalah seperti kode program di atas. Dalam kode program tersebut digambarkan bahwa *Token ArrayInit* berisikan tentang kode aturan program berdasarkan aturan dari *Grammar*.

Selnjutnya mengenai *Lexer*, *Lexer* merupakan bagian awal dari sebuah recognition , tugasnya adalah mengubah aliran karakter yang tidak bermakna menjadi potongan potongan *Token* yang sesuai dengan spesifikasi *Grammar* dari domain bahasanya. *Lexer* juga membuang karakter- karakter yang tidak bermakna misalnya komentar, spasi, tab, dan karakter baris baru (*new line*) (Fathan, 2007 : 4).

Lexer membantu proses berikutnya yakni *Parser* untuk memproses program dengan lebih baik. Aliran karakter menjadi lairan *Token* yang diperoses oleh *Lexer* akan menjadi kemudhan proses dan juga dimungkinkan bisa menjadikan proses program lebih cepat. Karena jika *Parser* memproses langsung

tentunya akan lebih memakan banyak waktu karena fungsi *Parser* yang berbeda dengan *Lexer*.

```
public class ArrayInitLexer extends Lexer {
      protected static final DFA[] _decisionToDFA;
      protected static final PredictionContextCache
_sharedContextCache =
            new PredictionContextCache();
      public static final int
            T__2=1, T__1=2, T__0=3, INT=4, WS=5;
      public static String[] modeNames = {
            "DEFAULT_MODE"
      };
      public static final String[] tokenNames = {
            "<INVALID>",
            "'{'", "','", "'}'", "INT", "WS"
      };
      public static final String[] ruleNames = {
            "T__2", "T__1", "T__0", "INT", "WS"
      };
      public ArrayInitLexer(CharStream input) {
            super(input);
            _interp = new
LexerATNSimulator(this,_ATN,_decisionToDFA,_sharedContextC
ache);
      }
```

Kode Program 3.5: *Lexer* pada *ArrayInit*

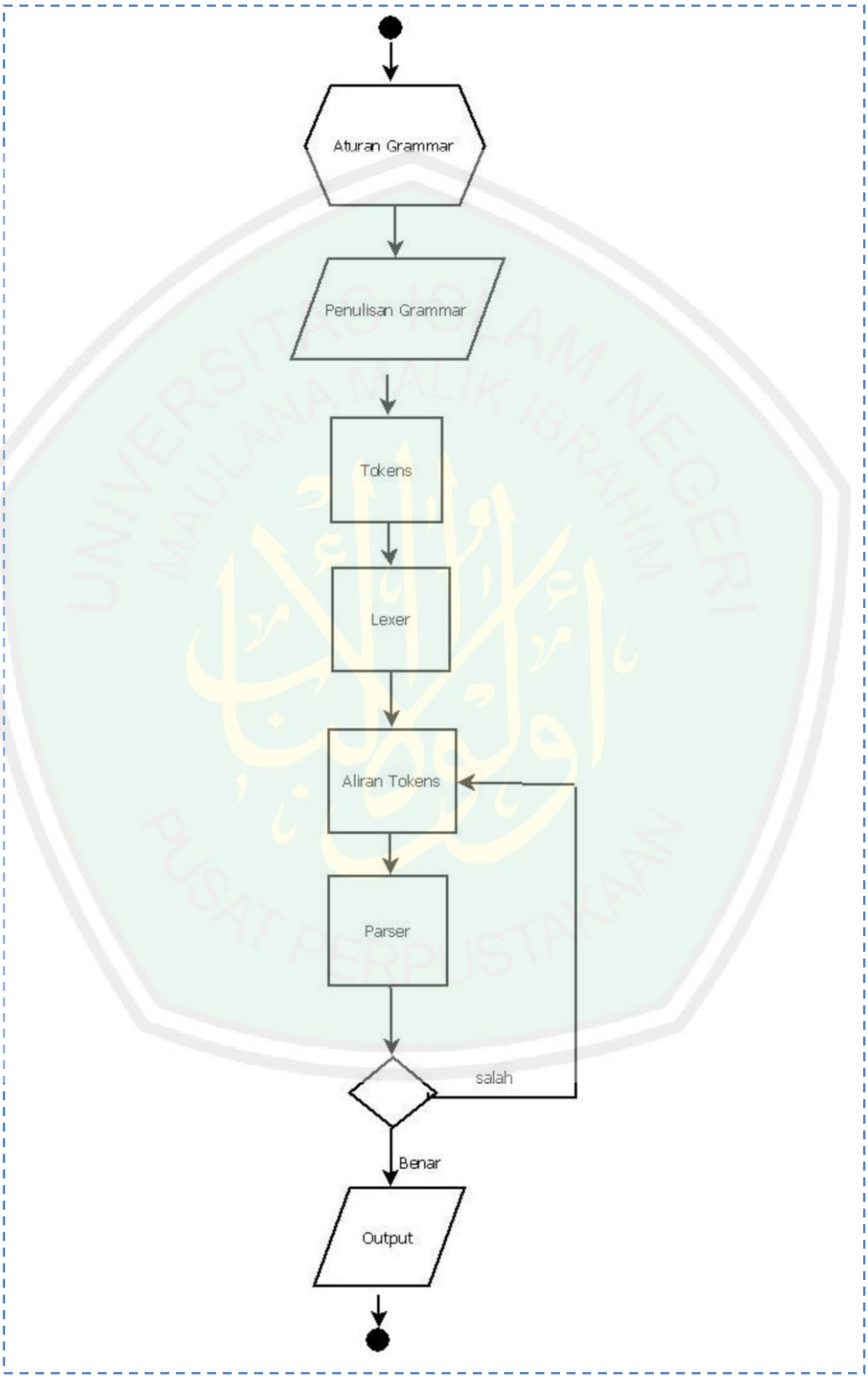
Pada kode program di atas digambarkan sebuah program *Lexer* untu *ArrayInit*. Dalam program tersebut digambarkan jika *Lexer* juga menggil aturan yang ada dalam *Token*s.

Parser memeriksa struktur kode sumber dengan menggunakan aturan - aturan dalam tata bahasa (*Grammar*) tertentu. *Grammar* inilah yang menentukan apakah suatu urutan *Token* dikatakan valid atau tidak.. (Bima, 2007 : 4).

Parser mendapat aliran data *Token* yang telah diproses oleh *Lexer*. Aliran *Token*s ini diproses oleh *Parser* untuk mengetahui valid data yangs esuai dengan aturan *Grammar* yang telah dideklarasaikan sebelumnya. Adanya kesalahan pada aliran *Token* akan dikembalikan pada *Lexer*.

Aturan *Parser* ini jika di dalam aturan *Grammar* bahasa Indonesia adalah penempatan yang tepat pada Subyek, Predikat, Objek hingga keterangan. Sedangkan *Lexer* merupakan proses yang memeriksa kebenaran kata sebelum dijadikan proses kebenaran kalimat oleh *Parser*.

Aliran *Token*s dari *Lexer* bisa digambarkan sebagai aliran kata yang awalnya tak berbentuk kemudian dipilah menjadi bermakna. Namun bila makna kata tak sesuai aturan *Grammar* yang diproses oleh *Parser*, maka kata tersebut seolah dikembalikan lagi pada *Lexer*.



Gambar 3.5: Alur proses Data *Grammar*

Gambar di atas megambarkan tentang alur proses yang dilalui untuk melakukan pemrosesan aturan bahasa *Grammar* mulai dari mempersiapkan aturan *Grammar* lalu penulisan *Grammar* yang akan menghasilkan *Token*s, *Lexer* dan proses *Parser*. Aliran *Token*s akan dikembalikan pada *Lexer* jika pada proses *Parser* ditemui kesalahan.

Pada *Lexer* pula akan bisa ditambahkan sistem penjelas yang mana akan melakukan klarifikasi atau mengeluarkan pemberitahuan jika terjadi kesalahan pada proses setelah melewati *Parser*. Kesalahan ini seperti tidak cocoknya kalimat yang dimasukkan dengan aturan penulisan *Grammar* yang telah dibuat.

3 2.3 Pengujian *Grammar*

Dalam sebuah perancangan sistem tentunya butuh pengujian internal dulu sebelum dihubungkan dengan program lainnya atau diimplementasikan pada khalayak ramai. Pengujian juga dibutuhkan untuk pengujian *Grammar* dalam rangakaian perancangan sistem, sehingga pada tahapan selenjutnya tinggal pada pengujian internal untuk proram game.

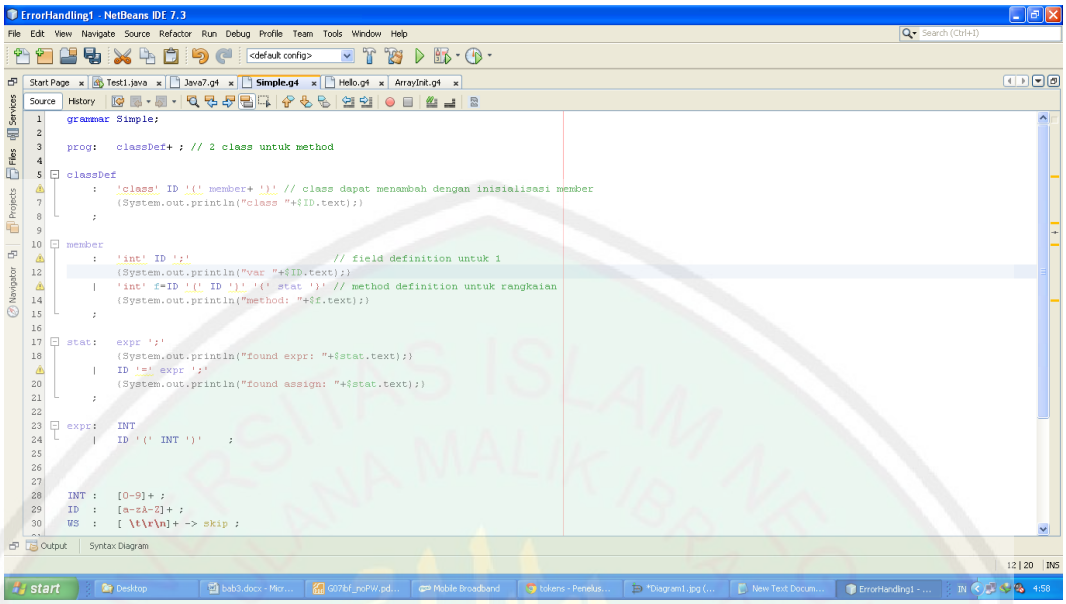
Pengujian *Grammar* ini dimaksudkan agar saat dilakukan implementasi atau integrasi dengan program grame akan lebih cepat prosesnya pada tahapan penyatuan bukan lagi pengujian internal program *Grammar*.

Dalam pengujian *Grammar* ini dilakukan pengujian menggunakan ANTLR untuk mempercepat proses. Hal ini karena ANTLR yang merupakan *parse generator* memiliki kemampuan untuk melakukan pemebentukan *Token*s, *Lexer* hingga *Parser* yang dibutuhkan dalam pengujian *Grammar* ini.

Pada versi yang lama pengujian *Grammar* dilakukan manual dengan membuat *Token*s, *Lexer* dan *Parser* yang kesemuanya dengan manual. Setelah adanya *parse generator* seperti ANTLR ini akan mempercepat proses seorang pro*Grammar* melakukan pemrograman *Grammar*. Memang cukup banyak aplikasi lain yang mendukung selain ANTLR, namun kemudahan interface hingga dukungan berbagai kompilator cukup memebantu mengembangkan pembuatan *Grammar* menggunakan ANTLR.

Langkah awal untuk pengujian *Grammar* adalah dengan meng-install aplikasi ANTLR. Untuk versi ANTLR sebelumnya ada namanya ANTLRWorks yang mendukung pembuatan *Grammar* secara lebih mudah dengan tampilan *Gnenerating Tree*.

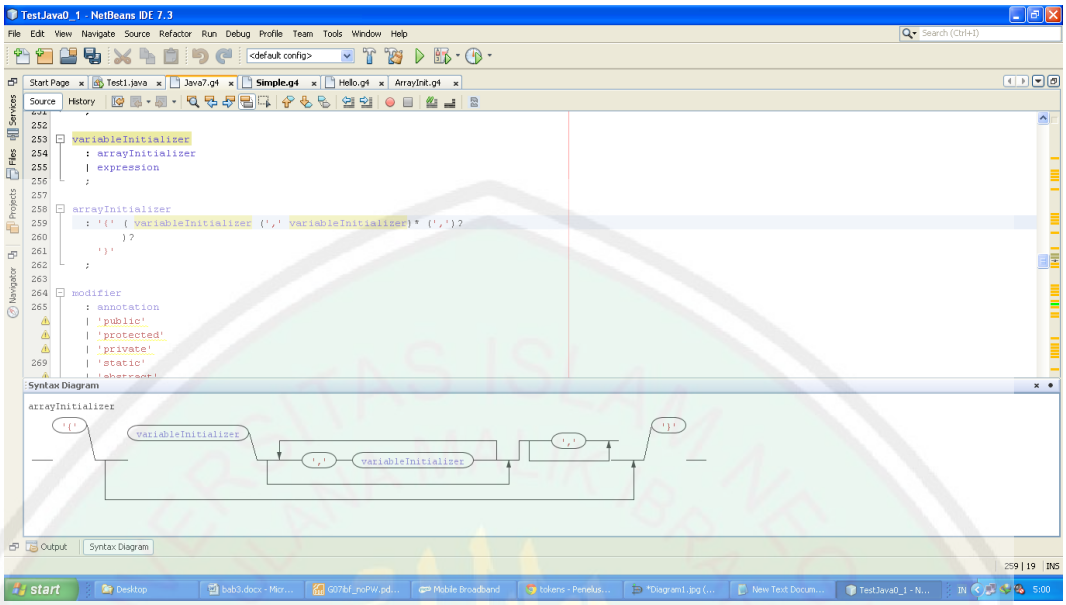
Untuk Versi ANTLR ke-4 dengan pengembangan ANTLRWorks generasi ke-2 didapatkan bahwa ANTLRWorks sudah berdiri sebagai *plugin* di Netbeans 7.3 sebagai wadah, untuk versi di bawah Netbeans 7.3 setelah dilakukan percobaan dan riset ternyata ANTLRWorks lebih kompatibel untuk Netbeans 7.3 daripada versi di bawahnya.



Gambar 3.6: Proses Pembuatan *Grammar* di Netbeans

Pada gambar Proses pembuatan *Grammar* di Netbeans terdapat kode pemrograman untuk membuat *Grammar*. Untuk mempermudah *Grammar* ini akan terhubung dengan *library* ANTLR sehingga kesalahan yang terjadi akan terlihat.

Bila sebuah editor belum dimasukkan library ANTLR dan juga *plugin* ANLTRWorks maka program belum bisa berjalan dengan baik.

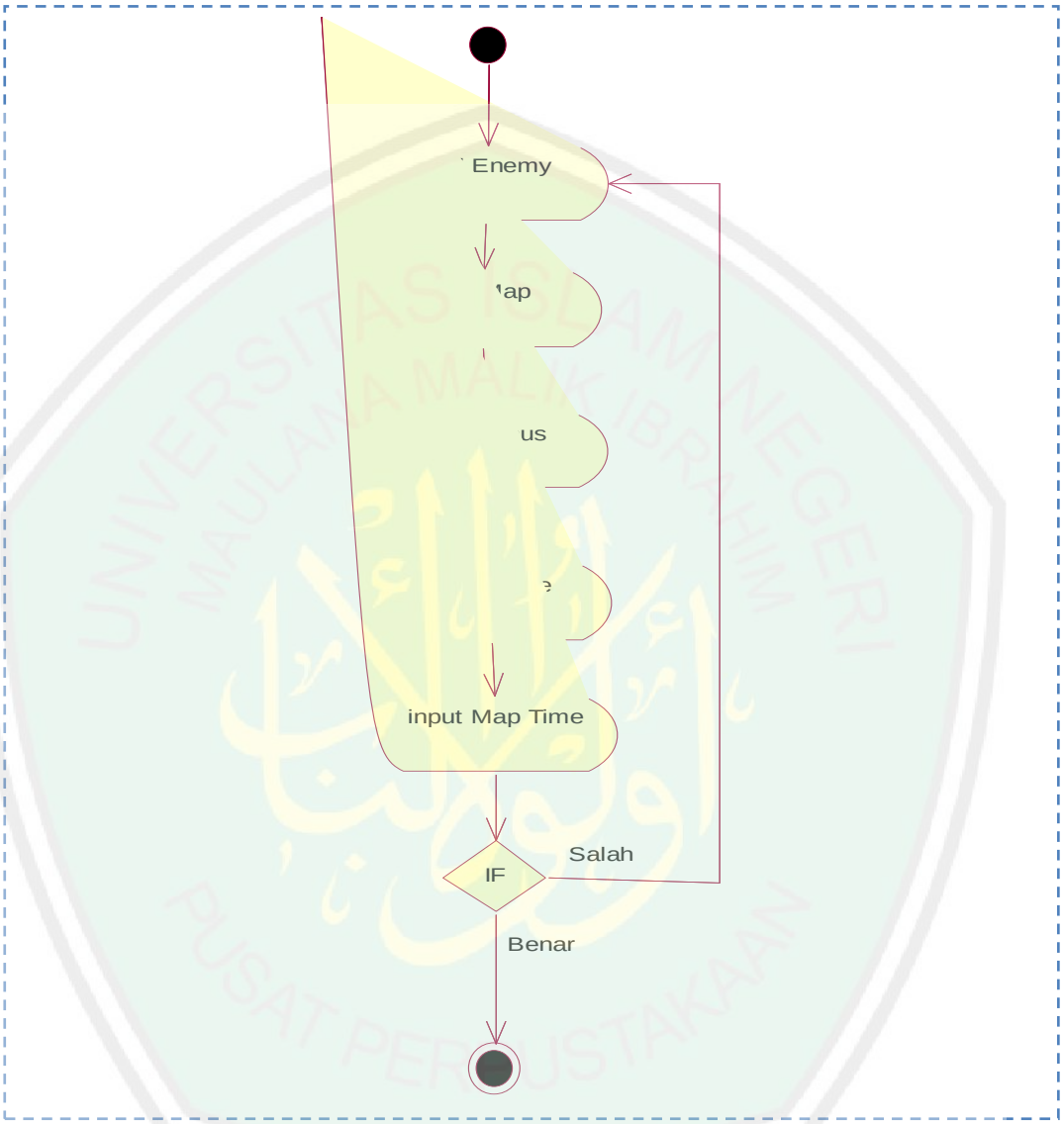


Gambar 3.7: *Syntax Diagram Grammar*

3.4 Perancangan Game

Perancangan game merupakan proses desain untuk game agar lebih mudah diimplementasikan bersama dengan sistem yang dari *Grammar*. Perancangan Game ini meliputi desain untuk sistem game dari pembuatan hingga proses serta prosedur permainan untuk mempermudah implementasi pemain game yang akan menggunakan.

3.4.1 Desain Pembuatan Game

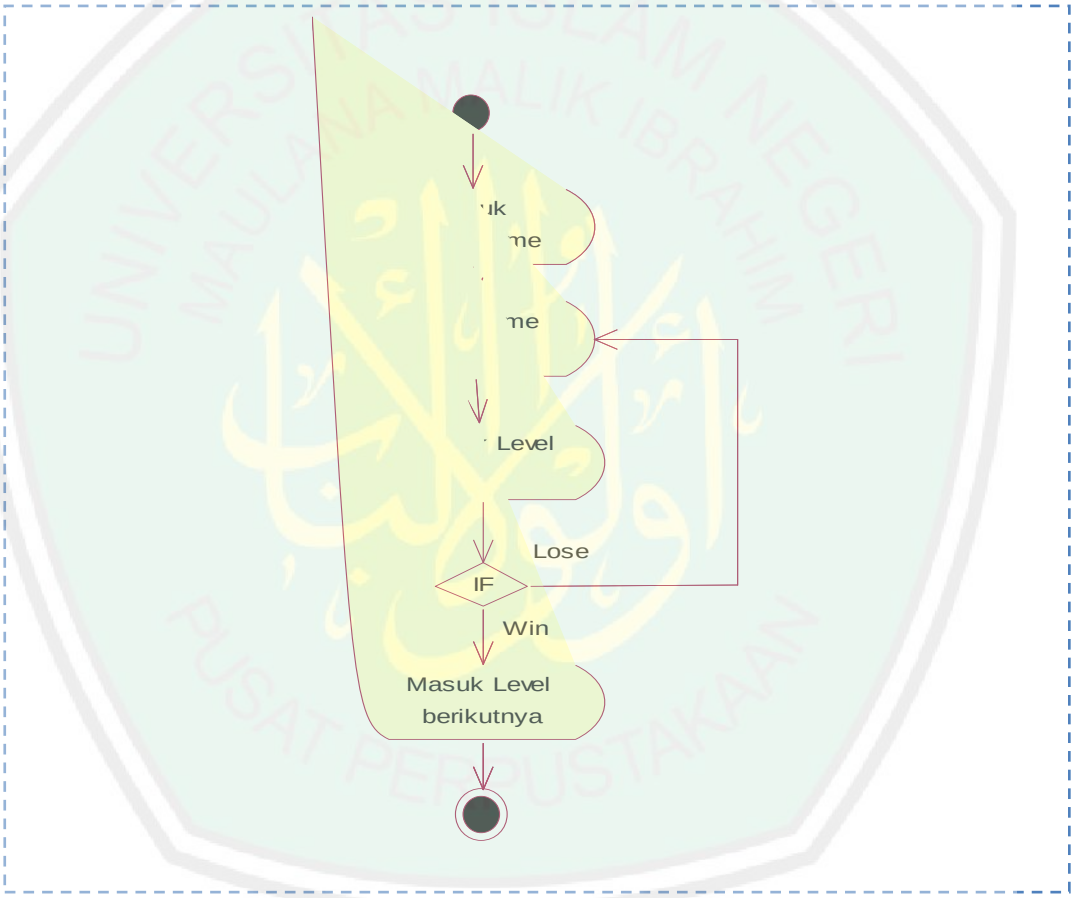


Gambar 3.8: *Activity Diagram* Pembuatan Game

Pada Gambar *Activity Diagram* Pembuatan Game ini proses pada Game menggunakan *LevelBuilder* yang telah disediakan GTGE untuk mempermudah pembuat game. Jadi pembuatan halaman *Game* dapat lebih cepat, namun untuk membuat *stage coding* tidak menggunakan fasilitas *LevelBuilder* ini.

Proses ini dimulai dari penambahan *Enemy* atau musuh, suasana atau lokasi tempat permainan hingga penamaan tempat permainan dengan waktu bermainnya.

3.4.2 Desain Alur Game

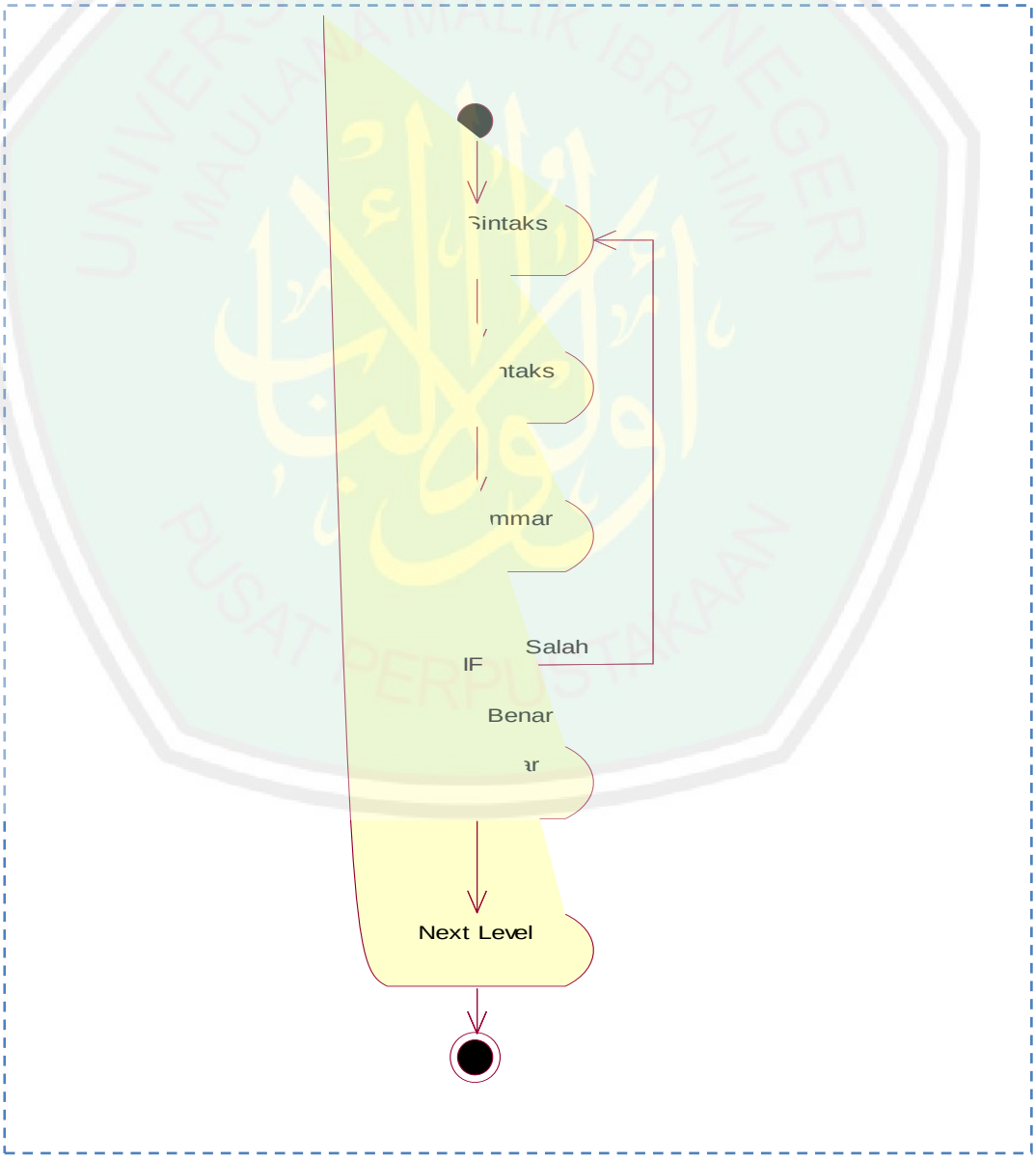


Gambar 3.9: *Activity Diagram* Alur Game

Pada Gambar Desain Alur Game dijelaskan alur masuk masuk *Java Coding Game* akan dilanjutkan pada Start Game. Inialisasi masuk *Java Coding Game* adalah pada tampilan halaman muka dengan pilihan layar *full screen* atau tidak disertai pilihan untuk menggunakan *BufferedReader*.

Dari gambar desain alur game terdapat gambaran proses dengan pilihan "*WIN*" atau "*LOSE*". Pilihan akan lanjut jika pemain menang serta permaikan akan kembali ke awal jika pemain kalah dalam tantangan game yang dengan kata lain *game over*.

3.4.3 Desain Integrasi Sistem



Gambar 3.10: *Activity Diagram* Intergrasi Sistem

Pada proses integrasi sistem akan ditemukan kondisi dimana inputan dari pemain game akan dicek oleh *Grammar*. Inputan yang ditangani dengan bahasa pemrograman game di GTGE akan dilanjutkan proses *Grammar* oleh ANTLR, jika jawaban benar maka permainan akan dilanjutkan.

3.4.4 Prosedur Permainan

Prosedur permainan dalam Game Edukasi *Java Coding Game* ini diperlukan untuk mempermudah pengguna dalam menggunakan. Prosedur Permainan game ini adalaha sebagai berikut:

a) Untuk ruang lingkup game pengguna dapat membuat halaman permainan atau peta melalui level builder yang sudah disediakan oleh GTGE.

b) Untuk edit halaman dan peta permainan juga bisa melalui level builder dengan melakukan perintah load map pada halaman level builder.

c) Menambah stage atau level game dapat dilakukan lewat level builder dengan melakukan input *Enemy* , bonus, Map Title, lingkungan level hingga lama waktu bermain.

d) Permainan dimulai sejak awal dengan nilai skor yang masih nol (“0”).

e) Bonus adalah sebuah tujuan yang berupa berlian atau gambar pion untuk mendapatkan poin hingga bertambahnya nyawa sehingga dapat lebih lama bermain jika terjadi kekalahan dalam satu level.

f) Permainan berakhir jika: waktu habis, keluar dengan menekan persetujuan tombol “Q” atau kadar nyawa yang dimili sudah habis.

BAB IV

HASIL DAN PEMBAHASAN

4.1 Implementasi

Setelah dilakukan perancangan dan pembuatan program, dalam bab ini akan dipaparkan implementasi dan analisa hasil pada program yang telah dibuat. Tujuan dari pengujian ini adalah untuk mengetahui apakah aplikasi yang telah dibuat sesuai dengan perancangannya. Selain itu juga untuk mengetahui detail jalannya aplikasi serta kesalahan yang ada untuk pengembangan dan perbaikan lebih lanjut. Pada proses pengujian ini untuk mendapatkan hasil yang optimal dibutuhkan beberapa peralatan-peralatan baik berupa perangkat keras dan perangkat lunak.

4.1.1 *Hardware* dan *Software* Untuk Pembuatan Aplikasi

a. *Hardware* yang Digunakan

Hardware atau Perangkat keras yang digunakan untuk membantu riset dalam pembuatan *Game* Edukasi *Java Coding Game* ini adalah sebagai berikut:

1. Prosesor Intel versi Core 2 Duo
2. RAM DDR 2 sebesar 3 GB
3. *Hardisk* Dengan Kapasistas 320 GB
4. LCD 14” dengan resolusi 1366 x 768 *pixels*
5. *Keyboard*

b. *Software* yang Digunakan

Alat pedukung aplikasi merupakan alat yang mendukung pembuatan dengan berbagai macam keperluan. Alat-alat ini berintegrasi mendukung tercipnyata aplikasi dengan hasil yang terbaik.

Software atau perangkat lunak yang digunakan dalam riset pembuatan *Game* Edukasi *Java Coding Game* ini adalah sebagai berikut:

1. Sistem operasi Windows XP SP3
2. Bahasa Pemrograman Java J2SE JDK 6.2
3. Netbeans 7.3
4. ANTLR 4
5. ANTLRWorks 2
6. GTGE *Game Engine* 2D
7. Notepad ++

4.1.2 Integrasi Aplikasi

Integrasi aplikasi dibutuhkan karena adanya bebarapa alat pendukung yang disatukan membentuk sebuah aplikasi. Integrasi aplikasi ini menjadi sangat penting karena dengan adanya integrasi dalam implementasi maka dapat terjadi sebuah aplikasi yang bisa dinikmati dan juga bermanfaat.

Integrasi dalam aplikasi *Game* ini pada intinya ada 2 yakni pada integrasi *Coding Grammar* dari ANTLR dengan GTGE *Game* engine. *Coding Grammar* digunakan untuk inti proses edukasi dalam *Game* yakni adanya pengecekan *grammar* yang dimasukkan oleh pemain dalam *Java Coding Game*.

Proses pengecekan ini menggunakan Tokens, *Lexer* dan *Parser* yang dihasilkan dari proses pembuatan *grammar* sebagai sebuah aturan. Aturan inilah menjadi patokan edukasi, proses edukasi disini dibuat lebih nyaman dengan menggunakan media permainan yang umumnya disukai sebagian orang.

Aturan yang ada dalam *Grammar* kemudian tergambarkan menyambung di dalam class-class yang dihasilkan melalui generate code dari *Grammar* ini akan berintegrasi dengan GTGE *Game* Engine. Tugas GTGE *Game* Engine melalui halaman *Game*-nya ialah mendapatkan inputan pemain *Game* yang kemudian diteruskan prosesnya oleh class dari ANTLR ini untuk diberikan jawaban.

4.2 Penjelasan Sistem Aplikasi

Setiap aplikasi memiliki sistem yang di dalamnya terdapat banyak yang saling berkesinambungan membentuk Aplikasi. Dalam Aplikasi berupa *Game* edukasi ini menggunakan GTGE *Game* Engine sebagai halaman untuk menampilkan kesan *Game*.

GTGE *Game* Engine mempermudah kerja dengan menyediakan level builder yang dapat membuat halaman untuk permainan. Sehingga proses *coding* sebenarnya lebih pada integrasi sistem yang memiliki gaya bahasa yang berbeda.

Jika GTGE *Game* Engine menggunakan bahasa Java yang telah dimodifikasi dengan standar kebutuhan dalam *Game* Engine ini, maka ANTLR memilki kaidah bahasa EBNF dalam membuat *grammar* . 2 jenis ini tenu berbeda, namun ANLTR dengan bantuan generate code *grammar* dapat berubah

menjadi banyak class yang saling berkesinambungan melakukan proses lexing, tokenizing hingga parsing untuk dapat menemukan makna sesuai aturan daam *grammar* yang telah dibuat.

Berikut ini adalah aturan pada bagian untuk mendapatkan input proses kebenaran pada *grammar* :

a) *Grammar* .g atau *grammar* .g4

Grammar memilki class tersendiri untuk membuat sebuah aturan. Dalam versi ANTLR terbaru terdapat tambahan nomor empat (“4”) dalam akhiran nama *grammar* yang bisanyanya berekstensi “.g”.

Di dalam *grammar* ini sistem menggunakan bahasa EBNF sebagai bahasa acuan dari ANTLR. Bahasa ini menginialisasi untuk mebentuk aturan yang nantinya akan menghasilkan program.

b) Generating Tree

Dalam *grammar* ANTLR ini terdapat sistem yang bila menggunakan fasilitas *syntax diagram* akan ditemui Gambar pohon atau tree yang terbentuk dari bahasa EBNF (*Extended Backus-Naur Form*) di dalam *grammar* ANTLR.

Pengecekan kebenran Generating Tree dalam *grammar* ANTLR menggunakan pengenalan dengan engine ANTLR. Sehingga dengan engine ANTLR ini dpat dihasilkan berbagai macam class dari *generate*-nya.

c) *Grammar* .tokens

Setelah mengetahui Generating Tree di dalam *grammar* ANTLR lalu melakukan generate code *grammar* yang menghasilkan class pendukugn seperti *Lexer* da *Parser*, maka token merupakan kebutuhan dalam sistem bahasa menggunakan ANTLR.

Token ini akan menjadi bahan rujukan dengannya proses lexing hingga parsing akan dijalankan, mulai dari penentukan kata yang tepat pada lezer hingga penentuan aliran token yang tepat pada *parser*.

d) *Lexer*

Lexer akan bekerja untuk mengecek kebenaran alisan Token setelah inputan dari pengguna *Game* di awal tadi dari GTGE *Game* Engine. *Lexer* melakukan proses yang kemudian akan diserahkan proses berikutnya oleh *parser*.

Lexer juga menerima kesalahan aliran token yang telah diberikan dari *parser* untuk dikembalikan lagi ke *lexer*. Jika ada tambahan perintah untuk menampilkan kesalahan maka *lexer* akan bertugas menyalurkan penerusan perintah agar ditampilkan, namun bila tak ada tambahan di dalam *lexer* maka yang ditampilkan adalah kesalahan biasa seperti kesalahan pada pengetikan program di *kompilator*.

e) *Parser*

Parser mendapat aliran token kemudian melakukan pemrosesan dengan rujukan *grammar* , jika terjadi kesalahan pada aliran token maka akan dikembalikan lagi ke *lexer*.

Alur sistem aplikasi berikutnya adalah pada GTGE *Game* Engine yang inti dari edukasi dalam *Game* ini adalah mendapatkan inputan dari pengguna atau pemain *Game*. GTGE *Game* Engine juga dimanfaatkan untuk menangi proses grafis dalam sebuah *Game*.

GTGE *Game* Engine memiliki aturan penulisan program tersendiri sebagai engine *Game*, oleh karenanya penting untuk mengetahui aturan penulisan program pada GTGE Geme Engine ini adalah sebagai berikut:

a) **Class jacod extend Jacod***Game***:** Class ini merupakan class utama dari GTGE *Game* Engine yang digunakan sebagai *host* untuk menempatkan class yang lain

b) **Public *method* initResources():** *Method* ini berfungsi untuk meletakkan *background*, music dan pemicu awal permainan, karena didalam GTGE class initResources inilah yang akan dieksekusi pertama kali

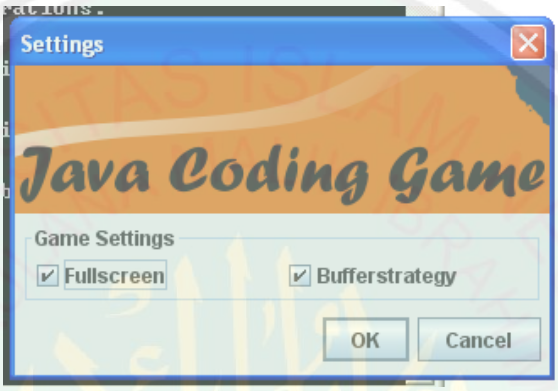
c) **Public *method update*():** *Method* ini merupakan *method* kedua yang berfungsi untuk meng*update* perubahan yang terjadi didalam *Game*, sehingga proses permainan dan sebagainya akan diletakkan disini

d) ***Method render*():** Dalam *method* ini semua *image* yang sudah diinisialisasikan di *initResource* dan semua perubahan *Game* tiap detik akan ditampilkan

4.2.1 Halaman *Setting*

Halaman ini merupakan tampilan awal yang muncul ketika *Game* edukasi *Java Coding Game* dijalankan. Halaman setting ini berfungsi sebagai pengatur

dari *Game* apakah *Game* akan dijalankan dengan mode fullscreen atau tidak. Selain itu di halaman ini juga mengatur pemanfaatan *buffer strategy*. Mode yang disediakan oleh GTGE ini berguna untuk memberikan pilihan kepada pemain dalam *Game* untuk memaksimalkan memory untuk *Game* atau tidak.



Gambar 4.1: Halaman *Setting Java Coding Game*

Pada gambar di atas merupakan halaman awal untuk memasuki Java Coding Game. Ada pilihan *Fullscreen* yang merupakan pilihan untuk menampilkan *Game* dalam format layar besar jika dicentang, sedang jika tak dicentang maka *Game* akan tampil dalam format lebih kecil. Di bawah ini adalah Kode program 4.1 yang digunakan untuk menampilkan halaman pada Gambar 4.1 di atas dengan mengambil gambar *Java Coding Game* dan aturan kode program lainnya memanggil fungsi *Game setting*.

```
public static void main(String[] args) {
            GameSettings settings = new
GameSettings(JacodGame.class.getResource("images/title.jpg")
) {
                  public void start() {
                        GameLoader game = new GameLoader();
                        game.setup(new JacodGame(), new
Dimension(640,480),
                                          fullscreen.isSelected(),
bufferstrategy.isSelected());
                        game.start();
                  }

                  protected JPanel initSettings() {
                        JPanel pane = super.initSettings();
                        pane.remove(sound);

                        return pane;
                  }
            };
      }

      { distribute = false; } // Menentukan tampilan Setting

}
```

Kode program 4.1: Penampil Halaman Pengaturan

Dalam kode program di atas diketahui bahwa pengaturan resolusi *Game*, pemilihan mode fullscreen dan bufferstrategy diatur di sini, fitur tersebut merupakan fasilitas yang disediakan oleh *library* GTGE.

4.2.2 *Input Font* pada *Game*

Dalam sebuah game, *font* cukup penting untuk melakukan inialisasi kata dalam berbagai *state*. Berikut ini kode program untuk *font* manager yang mengambil objek *image* menjadi bentuk *font*.

```
            // load font dari image

      fontManager.getFont(getImages("images/font.png", 16,
6),
                                                " !\"#$%&'()*+,-./"
+
                                                "0123456789:;<=>?" +
                                                "@ABCDEFGHIJKLMNO" +
                                                "PQRSTUVWXYZ['\\]^"
+
                                                "_abcdefghijklmno" +
                                                "pqrstuvwxyz{|}~");
      }
```

Kode program 4.2: *Font Manager*

Font yang diambil untuk masuk dalam *Game* berbantuk objek *image*, jadi perlu dipotong-potong untuk dapat mendapatkan kata per kata dengan *BufferedImage*.

4.2.3 *Input Buffered Image*

Buffered Image adalah sebuah *method* dalam pembuatan *Java Coding Game* ini untuk melakukan *buffering* gambar. Hal ini karena gambar atau image yang ada adalah merupakan image bersatu. Berikut ini Kode Program *BufferedImage*.

```
BufferedImage JacodImage;
      private BufferedImage[]     lowerImages;
      private BufferedImage[]     upperImages;
      private BufferedImage[]     coins;
      private BufferedImage[] jacodImage;

      private GameFont font;
```

Kode program 4.3: *BufferedImage*

Pada kode program di atas dilakukan pilihan *BufferedImage* pada image yang dibutuhkan dalam game. Untuk memanggil inisialisasi *BufferedImage*, maka dibutuhkan Kode Program untuk memanggilnya seperti berikut ini.

```
public void initResources() {
            JacodImage = getImage("images/title.jpg");
            lowerImages =
getImages("images/lowertileset.png", 10, 5);
            upperImages =
getImages("images/uppertileset.png", 11, 1);
            coins = getImages("images/coins.png", 3, 1);
            jacodImage = getImages("images/jacod.png", 12,
1);
```

Kode program 4.4: Memanggil *BufferedImage*

4.2.4 *Game* Objek

Game Objek merupakan kode program untuk memanggil objek *Game*, berikut ini Kode Program yang memanggil objek *Game*.

```
public GameObject getGame(int GameID) {
            switch (GameID) {
                  case MAIN_MENU        : return new
MainMenu(this);
                  case JACOD_GAME : return new Jacod(this);
            }

            return null;
      }
```

Kode program 4.5: *Game* Objek

4.2.5 Halaman Menu *Game*

Pada halaman ini, *Game* akan menampilkan beberapa pilihan tombol yang dapat dipilih oleh pemain dalam *Game* edukasi *Java Coding Game*.

Gambar 4.2 : Menu Game

Gambar di atas merupakan gambar untuk menu *Game* awal, dimana isi dari menu *Game* awal ini berbagai macam. Tombol-tombol tersebut didetailkan sebagai berikut.

a) Tombol *Start Game* : Sebagai pemicu untuk memasuki permainan. Pemain akan dibawa ke *level* satu setelah menekan tombol ini

b) Tombol *Instructions* :Berisikan instruksi *Game* dari mengenali pemain yang dijalankan, musuh hingga tombol cara menjalankan *Game*.

c) Tombol *Highscore*: Tombol ini untuk melihat hasil sepuluh skor tertinggi yang mampu diraih oleh pemain dalam *Game* edukasi *Java Coding Game*

d) Tombol *Sound*: Pemain dapat mematikan atau menghidupkan suara (*sound track*) yang sedang berjalan dalam *Java Coding Game*

e) Tombol *Level* : Pemain dapat memilih tombol level yang diinginkan sebagai tantangan kemampuan pemain. Hal ini karena tiap orang punya karakter kesukaan tersendiri dalam tantangan yang ditawarkan *Game*. Perlu diketahui untuk level *Easy* pemain yang dijalankan akan memiliki 4 nyawa.

f) Tombol *Credits* : Tombol ini digunakan sebagai halaman profil dan ucapan terima kasih untuk pihak yang membantu membuat *Game* ini

g) Tombol *Quit Game*: Tombol ini berada pada urutan terakhir dari list tombol yang ada. Tombol ini berfungsi sebagai pemicu untuk keluar dari program.

```
        String lvl = game.levelDesc[game.level];
        font.drawString(g, "Start Game",
GameFont.CENTER, 0, 150, getWidth());
        font.drawString(g, "Instructions",
GameFont.CENTER, 0, 190, getWidth());
        font.drawString(g, "Hi-Scores",
GameFont.CENTER, 0, 230, getWidth());
        font.drawString(g, "Sound: "+sound,
GameFont.CENTER, 0, 270, getWidth());
        font.drawString(g, "Level: "+lvl,
GameFont.CENTER, 0, 310, getWidth());
        font.drawString(g, "Creditvs",
GameFont.CENTER, 0, 350, getWidth());
        font.drawString(g, "Quit Game",
GameFont.CENTER, 0, 410, getWidth());
```

Kode program 4.6: Menampilkan Tombol *MainMenu*

Kode program di atas merupakan kode program untuk menampilkan menu *Game* utama. Dari program tersebut ada kode program `font.drawString` yang mengatur tata letak penempatan tiap pilihan dalam menu utama.

```
if (keyPressed(KeyEvent.VK_ENTER)) {
                playSound("sounds/switch.wav");

                switch (option) {
                      // start game
                      case 0:
                            parent.nextGameID =
JacodGame.JACOD_GAME;
                            finish();
                      break;

                      // instructions
                      case 1:
                            Instructions instructions = new
Instructions(parent);
                            instructions.start();
                      break;

                      // high score
                      case 2:
                            HighScore hiscore = new
HighScore(parent);
                            hiscore.start();
                      break;

                      // sound on/off
                      case 3:

     bsSound.setActive(!bsSound.isActive());

     bsMusic.setActive(!bsMusic.isActive());

                            if (bsMusic.isActive()) {

     bsMusic.play(bsMusic.getLastAudioFile());
                            }
                      break;

                      // level
                      case 4:
                            game.level++;
                            if (game.level > 2) {
                                  game.level = 0;
                            }
                      break;
```

```
                        // credits
                        case 5:
                              Credits credits = new
Credits(parent);
                              credits.start();
                        break;

                        // keluar dari game
                        case 6:
                              finish();
                        break;
                  }
```

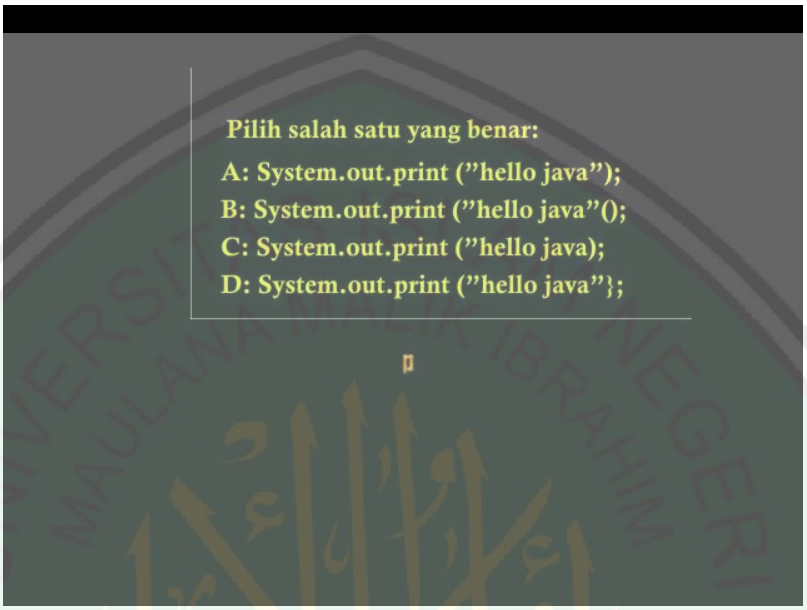
Kode program 4.7: Instruksi Tombol *MainMenu*

Kode Program di atas merupakan kode program yang berisi instruksi untuk tombol pada menu utama. Ketika pemain dalam *Game* memilih menu maka akan diatrahkan menuju menu yang dipilih. Berikut ini tampilan untuk menu yang jika dilih adalah memulai permainan dengan menekan tombol start.

Gambar 4.3 : Halaman *Game Stage* Awal

Dalam setiap level akan ada halaman *Game* yang hampir sama dengan gambar di atas, namun dengan rintangan yang berbeda. Setelah melewati stage di

halaman *Game* tersebut dengan menemukan pintu *exit*, maka pemain dalam *Game* akan menuju ke stage tentang *coding* seperti gambar di bawah ini.



Gambar 4.4 : *Coding Stage*

Pada Gambar di atas yang merupaan *coding stage*, pemain dalam *Game* diminta memilih jawaban mana yang tepat dari pertanyaan yang diajukan.

4.2.6 Halaman *Hi-Score*

Setiap permainan selesai maka *Game* akan segera mengalihkan layar ke halaman penyimpanan *Hi-Score*. Berikut ini kode program untuk mengambil data *score*.

```
...
                    DataInputStream din = new
DataInputStream(new FileInputStream(f));
                    for (int i=0;i < hiscore.length;i++) {
                         hiscore[i] = new
HiScoreData(din.readUTF(), din.readUTF(),
...
```

Kode program 4.8: Mengambil Data *Score*

Kode program di atas merupakan kode program untuk mengambil data score sebelumnya. Data score diambil untuk ditampilkan di layar permainan sehingga pemain dalam *Game* akan tahu, sistem akan menempatkan atau melakukan peringkat dimana pemain dalam *Game* berada sesuai dengan score yang telah dikumpulkan. Setelah mengambil data *score*, maka selanjutnya program *Game* akan menyimpan *score* dengan menambahkan nama sesuai keinginan pemain.

```
if (addscore != null) {
                    newHiScore = false;
                    for (int i=0;i < hiscore.length;i++) {
                         if
(Integer.parseInt(addscore.score) >
Integer.parseInt(hiscore[i].score)) {
                              // new high-score
                              for (int j=hiscore.length-1;j
> i;j--) {
                                   hiscore[j] = hiscore[j-
1];
                              }
                              hiscore[i] = addscore;
                              newHiScore = true;
                              gameState = INSERT_SCORE;
                              break;
                         }
                    }
```

Kode program 4.9: Menyimpan *Score*

Kode program di atas merupakan kode program untuk menyimpan score yang telah dikumpulkan pemain dalam *Game* di dalam sepanjang permainan yang dilakukannya. Sistem melakukan peringkat sesuai dengan jumlah *score*, semakin besar daripada *score* pemain sebelumnya sesuai data maka akan berada semakin di atas.

4.2.7 Halaman *Credits*

Halaman tentang kami berfungsi sebagai halaman untuk keterangan profil *developer*, instansi pendukung, alamat situs dan keterangan lain yang berhubungan dengan *Game*. Kode program di bawah ini bertugas untuk menampilkan Gambar tersebut ke layar dan menjadi non aktif ketika terjadi penekanan tombol enter.

```
public void render(Graphics2D g) {
          g.setColor(Color.BLACK);
          g.fillRect(0, 0, getWidth(), getHeight());
          g.drawImage(titleImage, 10, 10, null);

          font.drawString(g, "Development Game By:",
GameFont.CENTER, 0, 125, getWidth());
          g.drawImage(creditsImage, 120, 160, null);
```

Kode program 4.10: *Credits Game*

Kode program di atas mengambil data dan image untuk menampilkan tampian credits sebagaimana berikut ini:

Gambar 4.5: Tampilan *Credits*

4.2.8 Halaman *Quit Level*

Ini halaman yang digunakan pemain *Game* untuk keluar dari halaman utama ketika permainan masih berjalan. Halaman ini muncul jika *player* menekan tombol ESC. Pemain dapat memilih antara melanjutkan *Game* dengan menekan huruf C atau keluar dari permainan dengan menekan huruf Q. Setelah keluar dari halaman ini pemain akan kembali ke halaman menu. Berikut ini Kode program untuk keluar dari *Game*.

```
if (confirmExit) {
                  if (keyPressed(KeyEvent.VK_Q) ||
keyPressed(KeyEvent.VK_ESCAPE)) {
                      showHiScore();
```

Kode program 4.11: Keluar dari *Game*

Gambar 4.6: Tampilan *Quit Level*

Tampilan di atas merupakan tampilan gambar untuk *Quit Level* yang berisi berberapa pilihan yakni *Continue*, *Restart* ataupun *Quit*. Dengan menekan tombol Q berarti keluar, jika R berarti mengulangi permainan dari awal serta jika C berarti pilihan untuk melanjutkan permainan.

4.3 Pengujian Sistem

Sistem yang baik perlu dilakukan pengujian kepada sebagian orang untuk mendapat gamabar umum pada pemakaian sistem tersebut nantinya kepada khalayak yag lebih luas lagi. Implementasi *Generating Tree* dimulai dari penetapan *grammar* , lalu dengan bantuan ANTLR menghasilkan berbagai class yang saling berhubungan.

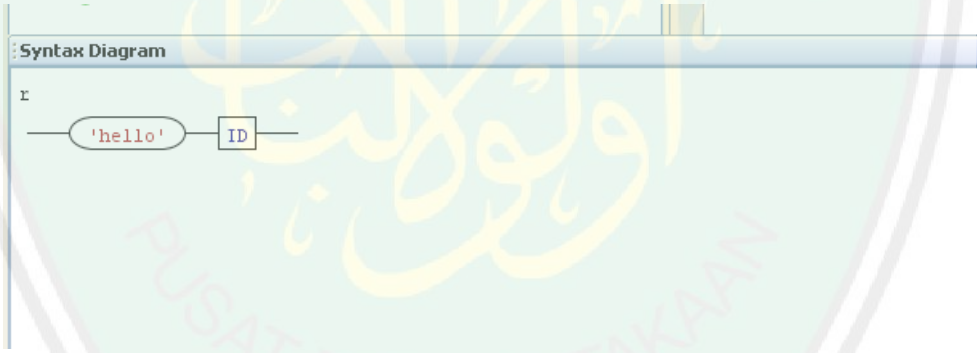
4.3.1 Pemaparan *Rule Grammar*

Berikut ini contoh *coding Game* pada *Grammar* untuk penentukan "hello world" sebagai inputan dari pemain *Game*. Inputan tersebut pertama kali diinisialisasikan melalui *coding grammar* berikut ini.

```
grammar Hello;
r  : 'hello' ID ;
ID : [a-z]+ ;
WS : [ \t\r\n]+ -> skip ;
```

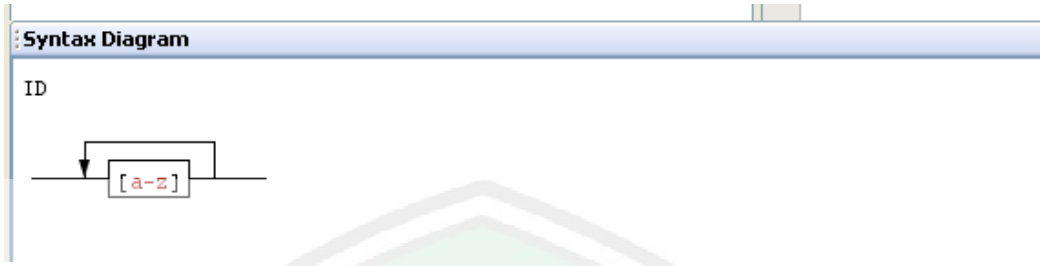
Kode program 4.12: *Grammar Hello*

Dengan menggunakan plugin *ANTLRWorks* di Netbeans 7.3 akan ditemui tampilan *Syntax Diagram* untuk *grammar Hello* sebagaimana berikut ini.



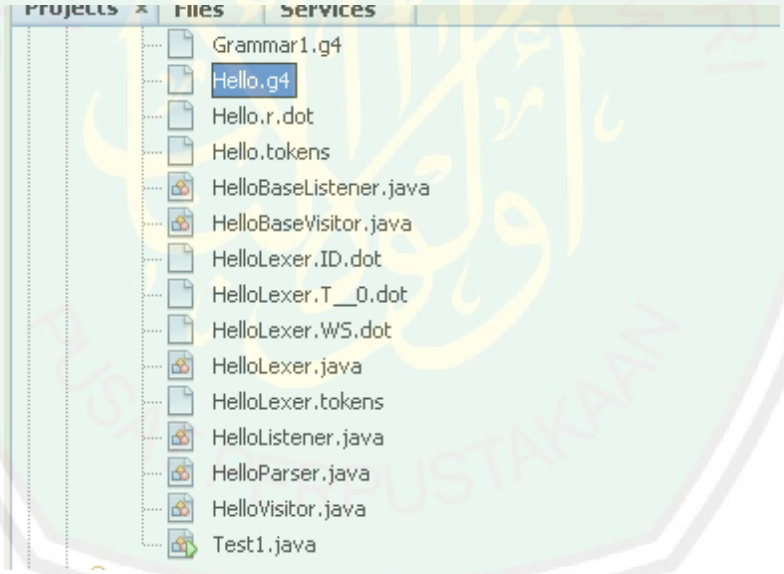
Gambar 4.7: *Syntax Diagram Hello*

Setiap baris di dalam *grammar* ini bisa dilehat di *Syntax Diagram* tentang alur kerjanya. Seperti Kode program di bawah ini tentang *Syntax Diagram* ID a-z.



Gambar 4.8: *Syantax Diagram ID a-z*

Setelah melakukan generating Code *Grammar* , maka akan ditemui berbagai macam class untuk mendukung proses selanjutnya. Berikut ini class yang dihasilkan dari Hello.g4



Gambar 4.9: *Class* yang dihasilkan dari Hello.g4

Untuk mengujinya dibutuhkan *class* tambahan, berikut ini potongan kode untuk menguji sementara *grammar* .

```
NTLRInputStream input = new ANTLRInputStream(//System.in);
                            amasuk       );
        // create a lexer that feeds off of input
CharStream
        HelloLexer lexer = new HelloLexer(input);

        // create a buffer of tokens pulled from the lexer
        CommonTokenStream tokens = new
CommonTokenStream(lexer);

        // create a parser that feeds off the tokens
buffer
        HelloParser parser = new HelloParser(tokens);

        ParseTree tree = parser.r(); // begin parsing at
init rule
        String s = tree.toStringTree(parser);
        System.out.println(s);
```

Kode program 4.13: Kode Pengujian *Grammar*

4.3.2 Rekapitulasi Hasil Angket

Game ini juga dilakukan pengujian kepada kalangan terbatas untuk mengetahui kualitas *Game* secara umum untuk diluncurkan kepada kalangan yang lebih luas. Oelh karena itulah dilakukan proses angket yang mana pilihan dalam angket adalah gambaran pada kulitas *Game* setelah melakukan pemakaian oleh sebagian pengguna.

Ada 4 pilihan dalam angket yang diberikan untuk menjawab pertanyaan setelah merasakan permainan, yakni SB untuk sangat baik, B untuk baik, C untuk cukup dan K untuk kurang.

Kemudahan dalam *Game* ini emrupaan pilihan anget tentang konfigurasi tombol dalam game, adanya penyajian persoalan hingga permainan secara keseluruhan. Mengenai lebih lengkapnya hasil angket sebagai berikut:

No	Uraian	SB	B	C	K
Mudahkah memainkan *Java Coding Game*?					
1	Konfigurasi tombol	3	6	1	0
2	Penyajian Persoalan	1	7	2	0
3	Permainan Keseluruhan	2	8	0	0
Bagaimana tanggapan mengenai *Java Coding Game*?					
4	Bentuk Tampilan	7	2	1	0
5	Pilihan Warna	1	8	1	0
6	Penggunaan Font	3	7	0	0
7	Penggunaan Bahasa	2	4	4	0
8	Sound	4	1	5	0
9	Tampilan Animasi	3	7	0	0
10	Ide / Konsep Cerita	4	5	1	0
Total Nilai		30	55	15	0

Tabel 4.1: Hasil Angket

Dari hasil angket di atas didapat berbagai gambaran yang terjadi pada berbagai orang yang melakukan pengujian atau percobaan pada *Java Coding Game*. Hal ini terjadi karena banyak karakter yang tentunya berbeda pula dlam menyikapi sebuah game.

Untuk lebih mendapat gambaran umum perlu disimpulkan sebuah hasil angket berdasarkan dari hasil angket yang telah diambil dari pengujian atau percobaan *Java Coding Game*. Berikut ini adalah tabel kesimpulan berdasarkan hasil angket. Pada kesimpulan ini terkadapat hasil angket yang sama pada kolom penggunaan bahasa, namun dimasukkan pada kecondongan yang lebih berat pada baik karena nilai untuk baik dekat dengan sangat baik yang berbeda dengan cukup dimana nilai kurang terbilang tidak ada. Berikut ini tabel kesimpulan hasil angket:

No	Uraian	SB	B	C	K
Mudahkah memainkan *Java Coding Game*?					
1	Konfigurasi tombol	30 %	60 %	10 %	0 %
2	Penyajian Persoalan	10 %	70 %	20 %	0 %
3	Permainan Keseluruhan	20 %	80 %	0 %	0 %
Bagaimana tanggapan mengenai *Java Coding Game*?					
4	Bentuk Tampilan	70 %	20 %	10 %	0 %
5	Pilihan Warna	10 %	80 %	10 %	0 %
6	Penggunaan Font	30 %	70 %	0 %	0 %
7	Penggunaan Bahasa	20 %	40 %	40 %	0 %
8	Sound	40 %	10 %	50 %	0 %
9	Tampilan Animasi	30 %	70 %	0 %	0 %
10	Ide / Konsep Cerita	40 %	50 %	10 %	0 %
Total Nilai		30 %	55 %	15 %	0 %

Tabel 4.2: Persentase Hasil Angket

Dari tabel kesimpulan hasil angket didapatkan kesimpulan cukup beragam, namun secara keseluruhan *Java Coding Game* dinilai Baik untuk digunakan sebagai *Game* edukasi. Dari tabel di atas terlihat bahwa dari hasil angket didapat 20 % yang melakukan percobaan kemudahan memainkan *Java Coding Game* secara keseluruhan sangat baik, untuk 80% menyatakan permainan secara keseluruhan Baik. Untuk yang menilai cukup atau kurang sebanyak 0 %.

Dalam sebuah *Game* tampilan adalah hal yang cukup penting untuk membuat pemain dalam *Game* nyaman bermain, selain itu juga *Game* akan menjadi betah bermain. Menurut Samuel Henry (2005 : 34), gambar memang salah satu bagian yang tidak terpisahkan dari sebuah game. Dari hasil angket didapat 70 % yang menyatakan bahwa bentuk tampilan sangat baik, untuk 20 %

menyatakan bentuk tampilan baik, sedang untuk 10 % berpendapat bentuk tampilan cukup.

Menurut Novian Triwidia Jaya (2010 : 97), Imajinasi adalah menciptakan gambat di pikiran. Di dalam pikiran, semua menjadi tidak ada batas. Imajinasi membuat pikiran bekerja lebih luas daripada biasanya. Hal ini seperti membayangkan pemandangan sangat indah di pinggir pantai dengan matahari yang tenggelam, angin semilir berhembus dan berbagai hal yang terbayang disana maka akan terasa seketika hal itu pada diri namun sebenarnya tidak nyata. Demikin juga pada *Game Java Coding Game* dengan kolaborasi antara bentuk tampilan, pemilihan warna hingga bentuk font merupakan komponen penting membentuk kenyamanan. Dari hasil angket didapat 80 % memberikan penilaian pada pilihan warna baik dalam *Game* ini, sedang 10 % memberikan penilaiannya sangat baik serta 10 % lainnya menyetakan pilihan warna cukup. Sedangkan untuk penggunaan font dalam *Game* sebanyak 30 % memberikan penilain sangat baik dan 70 % memberikan penilaian baik. Dari sini bisa disimpulkan tanggapan kolaborasi dalam tampilan, penggunaan warna dan penggunaan font antara sangat baik dan baik namun kebanyakan menilai sebagai kolaborasi yang baik. Di lain hal untuk penggunaan bahasa sebanyak 20 % memberikan penilaian sangat baik, untuk nilai baik dan cukup peserta angket sebanyak sama-sama 40%.

Menurut Samuel Henry (2005 : 65), semua *Game* saat ini sudah sewajarnya dilengkapi musik dan suara yang apik. Hal ini tentunya sudah tidak asing lagi bagi pemakai walaupun mereka sering mematikannya dan mendengar musik mereka sendiri. Namun jika *Game* tidak memiliki suara dan musik akan

terkesan tidak prfesional. Oleh karena hal itulah *Java Coding Game* dilengkapi dengan musik atau *sound* yang mendukung berjalannya *Game* lebih menarik. Dari hasil angket didapat sebanyak 50 % memberikan penilaian bahwa *Java Coding Game* memiliki sound yang cukup, sedang 40 % memberikan penilain *sound* sangat baik dan untuk 10 % memberikan penilaian *sound* baik.

Tampilan animasi merupakan komponen pendukung yang cukup penting juga, sebanyak 70 % dari hasil angket memberikan tampilan animasi pada *Java Coding Game* baik, sedangkan untuk 30 % memberikan penilaian bahwa tampilan animasi *Game* sangat baik.

Menurut Samuel Henry (2005 : 65), alur cerita dapat menambah semarak game. Oleh karena hal itu ide atau konsep cerita dalam *Java Coding Game* mungkin berbeda pada *Game* pada umumnya karena menyisipkan konsep edukasi kode-kode java. Dari hasil angket sebanyak 10 % memberikan penilaian cukup, sedangkan 50 % dari hasil angket memberikan penilaian baik dan 40 % memberikan penilaian sangat baik.

4.4 Kajian Agama

Semua ciptaan di dunia memiliki pelajaran penting bagi ulu al-albab, seperti perputaran siang dan malam, kisah-kisah sejarah dan pergolakan dunia tidak pernah putus (Barizi, 2009 : 63). Penelitian ini menggunakan Al-Qur'an dan Hadits sebagai landasan utamanya.

Hal ini penting untuk dilakukan agar penelitian yang terjadi tidak bertentangan dengan hukum syariat agama. Selain itu, tujuan dari penelitian ini

yang bermaksud untuk membantu pelajar memahami materi keilmuan merupakan hal yang dianjurkan di dalam Al-Qur'an maupun As-Sunnah.

4.4.1 Keutamaan Orang yang Menuntut Ilmu

Memiliki Keilmuan adalah sebuah kebutuhan bagi setiap manusia, ha ini seperti layaknya minum yang sealu haus akan keilmuan. Keilmuan memang penting karena untuk melakukan sesuatu misalnya mengendarai sepeda motor kita harus memiliki keilmuan tentang cara berkendara yang baik menggunakan sepeda motor.

Allah memberikan keistimewaan orang yang memiliki keilmuan. Sebab Ilmu berpengaruh bagi tegaknya suatu bangsa (Barizi, 2009 : 61). Hal ini seperti difirmankan Allah SWT seperti pada Surat Al-Mujadilah [58] : 11 berikut ini:

يَٰٓأَيُّهَا ٱلَّذِينَ ءَامَنُوٓاْ إِذَا قِيلَ لَكُمۡ تَفَسَّحُواْ فِي ٱلۡمَجَٰلِسِ فَٱفۡسَحُواْ يَفۡسَحِ ٱللَّهُ لَكُمۡۖ وَإِذَا قِيلَ ٱنشُزُواْ فَٱنشُزُواْ يَرۡفَعِ ٱللَّهُ ٱلَّذِينَ ءَامَنُواْ مِنكُمۡ وَٱلَّذِينَ أُوتُواْ ٱلۡعِلۡمَ دَرَجَٰتٖۚ وَٱللَّهُ بِمَا تَعۡمَلُونَ خَبِيرٞ ١١

Artinya:

> *Hai orang-orang beriman apabila kamu dikatakan kepadamu: "Berlapang-lapanglah dalam majlis", Maka lapangkanlah niscaya Allah akan memberi kelapangan untukmu. dan apabila dikatakan: "Berdirilah kamu", Maka berdirilah, niscaya Allah akan meninggikan orang-orang yang beriman di antaramu dan orang-orang yang diberi ilmu pengetahuan beberapa derajat. dan Allah Maha mengetahui apa yang kamu kerjakan.(Al-Mujadilah [58] : 11)*

Dalam *Tafsir Ibnu Katsir* turunnya ayat ini berkenaan dengan Tsabit bin Qais bin Syammas yang kisahnya terdapat dalam surat al-Hujurāt. Ada yang

berpendapat, ayat ini diturunkan berhubungan dengan beberapa orang ahli Badr (shahabat yang ikut serta dalam perang Badr), di antaranya Tsabit bin Qais bin Syamms yang datang menemui Nabi saw. ketika beliau sedang duduk di rumah Shafiyyah pada hari Jum'at. Tetapi para ahli Badr itu tidak mendapatkan tempat duduk, hingga akhirnya mereka berdiri di depan majelis. Melihat hal itu, Nabi saw. berkata kepada orang-orang yang bukan ahli Badr, Hal fulan, hai fulan, pindahlah dari tempatmu agar para ahli Badr itu bisa duduk. Nabi saw. sangat memuliakan ahli Badar. Nabi saw. pun mengetahui kalau orang-orang yang disuruhnya pindah itu merasa tidak senang. Sekaitan dengan kejadian itulah Allah Ta'ala Menurunkan ayat tersebut.

Maksud derajat dalam ayat ini yakni beberapa keutamaan di dalam surga, mengungguli derajat orang-orang yang diberi iman tanpa ilmu. Sebab seorang Mukmin yang berilmu lebih utama daripada orang Mukmin yang tak berilmu.

Kalimat *Wallāhu bimā ta'malūna khabīr* (dan Allah Maha Mengetahui apa yang kalian perbuat) maksudanya yakni kebaikan dan keburukan yang kalian perbuat. Allah mengetahui segala tingkah laku kita dimanapun, baik kita sembunyi bagaimanapun.

Keilmuan tentunya adalah keilmuan yang bersumber pada Agama yang menjadi rahmat bagi sekalian alam. Keilmuan yang dipunyai selayaknya dijadikan media untuk menjadikan lebih baik lagi dalam beribadah dan menyebarkan kebaikan di muka bumi ini.

Informatika merupakan keilmuan yang membahas tentang teknologi informasi punya peran penting dalam perbaikan kualitas kehidupan manusia.

Oleh karenanya pemanfaatan keilmuan bidang informatika dengan baik sesuai garis Agama Islam akan menjadikan pemilik ilmu seharusnya lebih baik pula dalam beribadah.

4.4.2 Dasar Hukum *Game* /Permainan Menurut Fiqh

Pendekatan dalam pembelajaran seharusnya berangkat dari konsep dasar manusia: fitrah (Barizi, 2009 : 75). Keilmuan yang dicari atau dimiliki memang menjadi kebutuhan setiap manusia, namun di sisi lain tak dapat dipungkiri juga bahwa manusia membutuhkan pembelajaran yang baik dan mudah diterima.

Pembelajaran yang mudah diterima ialah pembelajaran yang sesuai konsep dasar manusia yakni fitrahnya. Hampir setiap manusia memiliki rasa suka akan hal yang menyenangkan, Islam pun memberikan batasan akan rasa suka akan hal yang menyenangkan agar tak terjerumus ke dalam hal yang kurang baik.

Hal yang menyenangkan yang diatur ini tentu menjadikan ada batasan untuk menyenangkan suatu aktivitas. Permainan adalah termasuk hal yeng menyenangkan itu pula harus ada batasannya. Hukum permainan ini bisa dirujuk dari hadtis Aisyah. Dari ‘Aisyah radhiyallahu ‘anha, beliau menceritakan bahwa, *“Ia pernah bersama Nabi shallallahu ‘alaihi wa sallam dalam safar. ‘Aisyah lantas berlomba lari bersama beliau dan ia mengalahkan Nabi shallallahu ‘alaihi wa sallam. Tatkala ‘Aisyah sudah bertambah gemuk, ia berlomba lari lagi bersama Rasul shallallahu ‘alaihi wa sallam, namun kala itu ia kalah. Lantas Nabi shallallahu ‘alaihi wa sallam bersabda, “Ini balasan untuk kekalahanku*

dahulu." (HR. Abu Daud no. 2578 dan Ahmad 6: 264. Syaikh Al Albani mengatakan bahwa hadits ini *shahih*)

Dalam hadits di atas dijelaskan Nabi Muhammad SAW berlomba dengan Aisyah dengan berlari. Sosok Rosulullah yang menyempatkan diri berlomba denga Aisyah bisa digambarkan bagaimana sikap Rosullah yang bisa menyempatkan diri untuk berlomba.

Dalam hal lomba inilah yang biasa disebut juga permainan. Namun sebagai sebuah kegiatan atau aktivitas tentunya permainan tak mencederai asas pada ajaran keislaman. Perlombaan yang disertai hal yang kurang baik seperti memperolok orang lain selayaknya tak dilakukan. Hal ini seperti pada Surat Al-Hujurat [49] : 11 sebagai berikut.

يَٰٓأَيُّهَا ٱلَّذِينَ ءَامَنُواْ لَا يَسۡخَرۡ قَوۡمٞ مِّن قَوۡمٍ عَسَىٰٓ أَن يَكُونُواْ خَيۡرٗا مِّنۡهُمۡ وَلَا نِسَآءٞ مِّن نِّسَآءٍ عَسَىٰٓ أَن يَكُنَّ خَيۡرٗا مِّنۡهُنَّۖ وَلَا تَلۡمِزُوٓاْ أَنفُسَكُمۡ وَلَا تَنَابَزُواْ بِٱلۡأَلۡقَٰبِۖ بِئۡسَ ٱلِٱسۡمُ ٱلۡفُسُوقُ بَعۡدَ ٱلۡإِيمَٰنِۚ وَمَن لَّمۡ يَتُبۡ فَأُوْلَٰٓئِكَ هُمُ ٱلظَّٰلِمُونَ ١١ يَٰٓأَيُّهَا ٱلَّذِينَ ءَامَنُواْ لَا يَسۡخَرۡ قَوۡمٞ مِّن قَوۡمٍ عَسَىٰٓ أَن يَكُونُواْ خَيۡرٗا مِّنۡهُمۡ وَلَا نِسَآءٞ مِّن نِّسَآءٍ عَسَىٰٓ أَن يَكُنَّ خَيۡرٗا مِّنۡهُنَّۖ وَلَا تَلۡمِزُوٓاْ أَنفُسَكُمۡ وَلَا تَنَابَزُواْ بِٱلۡأَلۡقَٰبِۖ بِئۡسَ ٱلِٱسۡمُ ٱلۡفُسُوقُ بَعۡدَ ٱلۡإِيمَٰنِۚ وَمَن لَّمۡ يَتُبۡ فَأُوْلَٰٓئِكَ هُمُ ٱلظَّٰلِمُونَ ١١

Artinya:

> *Hai orang-orang yang beriman, janganlah sekumpulan orang laki-laki merendahkan kumpulan yang lain, boleh jadi yang ditertawakan itu lebih baik dari mereka. dan jangan pula sekumpulan perempuan merendahkan kumpulan lainnya, boleh jadi yang direndahkan itu lebih baik. dan janganlah suka mencela dirimu sendiri dan jangan memanggil dengan gelaran yang mengandung ejekan. seburuk-buruk panggilan adalah (panggilan) yang buruk*

sesudah iman dan barangsiapa yang tidak bertobat, Maka mereka Itulah orang-orang yang zalim. (Al-Hujurat [49] : 11)

Menurut *Tafsir Ibnu Katsir* turunnya ayat ini berkenaan dengan Tsabit bin Qais bin Syammas yang kisahnya terdapat dalam surah al-Hujurāt. Ada yang berpendapat, ayat ini diturunkan berhubungan dengan beberapa orang ahli Badr (shahabat yang ikut serta dalam perang Badr), di antaranya Tsabit bin Qais bin Syamms yang datang menemui Nabi saw. ketika beliau sedang duduk di rumah Shafiyyah pada hari Jum'at. Tetapi para ahli Badr itu tidak mendapatkan tempat duduk, hingga akhirnya mereka berdiri di depan majelis. Melihat hal itu, Nabi saw. berkata kepada orang-orang yang bukan ahli Badr, Hal fulan, hai fulan, pindahlah dari tempatmu agar para ahli Badr itu bisa duduk. Nabi saw. sangat memuliakan ahli Badar. Nabi saw. pun mengetahui kalau orang-orang yang disuruhnya pindah itu merasa tidak senang. Sekaitan dengan kejadian itulah Allah Ta'ala Menurunkan ayat tersebut.

Kalimat *Yarfa'illāhul ladzīna āmanū mingkum* (niscaya Allah akan Meninggikan orang-orang yang beriman di antara kalian), menurut Ibnu Katsir ialah baik secara diam-diam maupun se cara terang-terangan.

Sedangkan kata *Darajāt* (beberapa derajat), masudnya beberapa keutamaan di dalam surga, mengungguli derajat orang-orang yang diberi iman tanpa ilmu. Sebab seorang Mukmin yang berilmu lebih utama daripada orang Mukmin yang tak berilmu.

Lebih lanjut menurut *Tafsir Ibnu Katsir* kalimat *Wallāhu bimā ta'malūna khabīr*

(dan Allah Maha Mengetahui apa yang kalian perbuat) ialah yakni kebaikan dan keburukan yang kalian perbuat.

Dalam elaborasi ayat ini kita dapat menemuka makna bahwa memperolok yang lainnya karena belum tentu yang diperolok itu lebih rendah dari apa yang diperoloknya. Dalam hal permainan, selayaknya dalam norma permainan perlu memegang kaidah hal ini agar permainan adalah sebagai ajang menyenangkan dan menambah rasa persaudaraan.

Permainan yang menyenangkan akan kurang terasa bermanfaat jika ada kata-kata yang kurang pantas yang dihujamkan ke yang lainnya. Manfaat dan semangat permainan akan terkikis dengan hal ini. Sebaiknya dalam hal permainan kita tetap menghargai perbedaan dengan menjunjung tinggi kaidah dalam ajaran

4.4.3 Ulul Albab

Ada dua hal fundamental yang bisa diidentifikasi sebagai *ulu al-albab*, yaitu zikir dan pikir (Barizi, 2009 : 63). Zikir dan pikir memang menjadi hal yang saling berhubungan dalam diri pembelajar yang beragama Islam. Konsep *ulul albab* sebagai gambaran mereka yang berilmu seperti pada Surat Ali Imran [3] : 189 – 190 sebagaimana berikut ini.

وَلِلَّهِ مُلْكُ ٱلسَّمَٰوَٰتِ وَٱلْأَرْضِ ۗ وَٱللَّهُ عَلَىٰ كُلِّ شَىْءٍ قَدِيرٌ ﴿١٨٩﴾ إِنَّ فِى خَلْقِ ٱلسَّمَٰوَٰتِ وَٱلْأَرْضِ وَٱخْتِلَٰفِ ٱلَّيْلِ وَٱلنَّهَارِ لَءَايَٰتٍ لِّأُو۟لِى ٱلْأَلْبَٰبِ ﴿١٩٠﴾

Artinya:

> *189. Kepunyaan Allah-lah kerajaan langit dan bumi, dan Allah Maha Perkasa atas segala sesuatu.*

> *190. Sesungguhnya dalam penciptaan langit dan bumi, dan silih bergantinya malam dan siang terdapat tanda-tanda bagi orang-orang yang berakal, (Ali Imran [3] : 189 – 190)*

Menurut *Tafsir Ibnu Katsir* kalimat *Wa lillāhi mulkus samāwāti wal ardl* (Kepunyaan Allah-lah kerajaan langit dan bumi) maksudnya adalah perbendaharaan langit yang berupa hujan, dan perbendaharaan bumi yang berupa tumbuh-tumbuhan.

Sedangkan kalimat *Wallāhu 'alā kulli syai-ing qadīr* (dan Allah Maha Kuasa terhadap segala sesuatu), yakni terhadap penghuni langit dan bumi serta perbendaharaan keduanya. Selanjutnya Allah Ta'ala Menjelaskan tanda-tanda Kekuasan-Nya kepada orang-orang kafir Mekah. Hal ini berkaitan dengan ucapan mereka, *"Hai Muhammad, tunjukkanlah kepada kami suatu tanda yang menunjukkan benarnya apa yang kamu katakan!"*

Dalam ayat berikutnya yakni Ali Imran [3] : 190 terdapat kalimat *Inna fī khalqis samāwāti* (sesungguhnya dalam penciptaan langit) maksudnya yakni sesungguhnya dalam penciptaan segala makhluk yang ada di langit, yaitu malaikat, matahari, bulan, bintang, dan awan.

Sedang kata *Wal ardli* (dan bumi), yakni dan dalam penciptaan bumi beserta segala sesuatu yang ada padanya berupa gunung, lautan, pepohonan, dan hewan. Maksudnya apa yang ada di dalam bumi termasuk di dalamnya.

Kalimat *Wakh tilāfil laili wan nahāri* La āyātin (serta silih bergantinya malam dan siang terdapat tanda-tanda), yakni dan juga dalam pertukaran malam dan siang. terdapat tanda-tanda yang menunjukkan Keesaan-Nya. Tanda-tanda inilah yang dibutuhkan insan bercirikan Ulul Albab untuk dapat membacanya.

Lebih lanjut kalimat *Li ulil albāb* (bagi orang-orang yang berakal), maksudanya ialah bagi manusia yang memiliki pikiran. Kemudian Allah SWT Mengemukakan sifat-sifat orang yang berakal dengan Firman-Nya.

Gambar 4.10: Logo UIN Maulana Malik Ibrahim Malang

Universitas Islam Negeri (UIN) Maulana Malik Ibrahim Malang yang menyadarkan logo universitas dengan kata Ulul Albab mengisyarakatkan akan keinginan civitas akademika Universitas ini dapat menjadi insan Ulul Albab yang mampu berpikir membaca tanda-tanda dari Allah SWT. Di Surat Ali Imran [3] : 191 Allah menerangkan tentang Ulul Albab sebagaimana berikut.

ٱلَّذِينَ يَذۡكُرُونَ ٱللَّهَ قِيَٰمٗا وَقُعُودٗا وَعَلَىٰ جُنُوبِهِمۡ وَيَتَفَكَّرُونَ فِي خَلۡقِ ٱلسَّمَٰوَٰتِ وَٱلۡأَرۡضِ رَبَّنَا مَا خَلَقۡتَ هَٰذَا بَٰطِلٗا سُبۡحَٰنَكَ فَقِنَا عَذَابَ ٱلنَّارِ ١٩١

Artinya:

> *(yaitu) orang-orang yang mengingat Allah sambil berdiri atau duduk atau dalam keadan berbaring dan mereka memikirkan tentang penciptaan langit dan bumi (seraya berkata): "Ya Tuhan Kami, Tiadalah Engkau menciptakan ini dengan sia-sia, Maha suci Engkau, Maka peliharalah Kami dari siksa neraka (Qs. Ali-Imran: 191)*

Dalam *Tafsir Ibnu Katsir* kalimat *Alladzīna yadzkurūnallāha* ([yaitu] orang-orang yang mengingat Allah), yakni orang-orang yang shalat. Sedangkan *Qiyāman* (sambil berdiri) yakni bila mereka mampu. *Wa qu'ūdan* (atau duduk) bila tidak mampu berdiri. *Wa 'alā junūbihim* (atau dalam keadaan berbaring) bila tidak mampu berdiri dan duduk.

Lebih lanjut kalimat *Wa yatafakkarūna fī khalqis samāwāti wal ardli* (dan mereka memikirkan tentang penciptaan langit dan bumi) ialah yang berkaitan dengan segala hal yang mengagumkan.

Ayat ini mengisyaratkan pula Allah menciptakan segalanya dengan mengagumkan, lalu di akhir ayat terdapat doa bahwa segala yang mengagumkan pasti ada maksudnya.

Dalam hal edukasi selayaknya mereka yang meraih ilmu memikirkan tanda-tanda kekuasaan Allah sebagai bentuk pelajaran dalam keilmuan. Banyak mereka yang bisa membaca tanda ini menjadikan dirinya lebih baik. Insan Ulul Albab selayaknya bisa menjadikan kehidupan lebih bermakna.

BAB V

PENUTUP

5.1 Kesimpulan

Dari penelitian dan implikasi uji coba penelitian ini, dapat disimpulkan bahwa penggunaan metode *Generating Tree* mempunyai kemudahan dalam inialisasi program untuk menghasilkan *grammar* yang baik untuk dapat dibaca pada pengujian yang menghasilkan pengecekan kebenaran *grammar* yang dimasukkan.

ANTLR membantu melakukan inialisasi *grammar* java yang bisa bersifat *Domain Spesific Language* (DSL). Pembuatan *grammar* dengan bahasa *Extended Backnaus-Naur Form* mudah dipelajari dan dibuat, namun untuk lingkup bahasa yang lebih detail dan dalam perlu pembelajaran yang lebih mendalam pula.

Dengan bantuan ANTLRWorks dapat dilihat bagaimana dalam *grammar* gambaran pola *Generating Tree* sebagai dasar untuk melakukan *Lexer* dan *Parser* hingga membentuk sebuah DSL.

Generate code grammar setelah melalui pembuatan dan analisa menggunakan ANTLRWorks untuk melihat gambaran pola *Generating Tree*, ANLTR membantu pula untuk menghasilkan *Token, Lexer* dan *Parser*. Untuk versi ANTLR versi 4 menghasilkan *class* lain pula seperti *Listener, Visitor, Baselistener, Basevisitor* dan lainnya dengan ekstensi *.dot*. Setelah *grammar* menghasilkan *Token, Lexer* dan *Parser* selanjutnya bisa diujicoba dengan program sederhana memanggil ketiga program ini. Setelah sukses selanjutnya bisa diintegrasikan dengan program lainnya.

GTGE merupakan *Game Engine* buatan Indonesia yang memudahkan bagi para pembuat game bersifat 2 dimensi. Dengan game yang sudah dihasilkan seperti *Warlock* yang di dalamnya ada *Level Builder* dapat membantu pembuatan halaman Level, mengatur tata letak, *enemy*, hingga waktu bermain dalam sebuah game lebih cepat.

Dari hasil angket didapat 20 % yang melakukan percobaan kemudahan memainkan *Java Coding Game* secara keseluruhan sangat baik, untuk 80% menyatakan permainan secara keseluruhan Baik. Dari sini dapat disimpulkan bahwa penggunakan *Java Coding Game* sebagai game edukasi mendapat respon yang baik. Sedang untuk ide atau konsep cerita dalam *Java Coding Game* mungkin berbeda pada *Game* pada umumnya karena menyisipkan konsep edukasi kode-kode java. Dari hasil angket sebanyak 10 % memberikan penilaian cukup, sedangkan 50 % dari hasil angket memberikan penilaian baik dan 40 % memberikan penilaian sangat baik.

5.2 Saran

Ada bebarapa hal yang perlu dikembangkan dari penelitian ini, yakni seperti penggunaan sistem penjelas untuk memudahkan pemain game dalam proses edukasi, hal ini karena dengan adanya sistem penjelas dapat membantu pemain game menemukan jawaban yang tepat.

Alur cerita game juga bisa ditambah lebih menarik dengan berbagai warna dan kondisi yang tak lepas dari proses edukasi bahasa pemrograman java lebih menarik.

DAFTAR PUSTAKA

Almatin, Isma. 2010. *Dahsyatnya Hypnosis Learning untuk Guru dan Orang Tua*. Yogyakarta: Pustaka Widyatama.

Barizi, Ahmad. 2009. *Menjadi Guru Unggul Bagaimana Menciptakan Pembelajaran yang Produktif & Profesional*. Jogjakarta: Ar-Ruzz Media.

Fakultas Sains dan Tenologi UIN Maulana Malik Ibrahim Malang. 2011. *Pedoman Penulisan Tugas Akhir/Skripsi Fakultas Sains dan Teknologi*. Malang: Fakultas Sains dan Tenkologi UIN Maulana Malik Ibrahim Malang.

Fathan, Ibnu Bima. 2007. Penyusunan Model Evaluasi Pada sistem pembelejaran Cerdas. *Skripsi* Tidak Diterbitkan. Bogor: Departemen Ilmu Komputer Fakultas MIPA Institut Pertanian Bogor.

Hariyanto, Bambang. 2004. *Teori Bahasa, Otomata dan Komputasi serta terapannya*. Bandung: Informatika.

Henry, Samuel. 2005. *Panduan Praktis Membuat Game 3D*. Yogyakarta: Graha Ilmu.

Jaya. Novian T. 2010. *Hypno Teaching Bukan Sekedar Mengajar*. Bekasi: D-Brain

Katsir, Ibnu. 2009. *Al-Kalam Digital versi 1.0: Tafsir Ibnu Katsir*. Bandung: Penerbit Diponegoro.

Meigs, Tom. 2003. *Ultimate Game Design: Building Game Worlds*. California: McGraw-Hill.

Parr, Terence. 2007. *The Definitive ANTLR Reference Building Domain-Specific Languages*. Texas, USA: The Pragmatic Bookshelf.

Parr, Terence. 2009. *Language Implementation Patterns: Creating Your Own Domain-Specific and General Programming Languages.*. Texas, USA: The Pragmatic Bookshelf.

Parr, Terence. 2012. *The Definitive ANTLR 4 Reference*. Texas, USA: The Pragmatic Bookshelf.

Purnama, Rangsang. 2003. *Pemrograman Java Jilid 1*. Jakarta: Prestasi Pustaka Publishere

Rachman, Anung dkk. 2010. Agen Cerdas Animasi Wajah Untuk Game Tebak Kata. *Jurnal* Tidak Diterbitkan. Semarang : Pascasarjana Teknik Informatika Unibersitas Dian Nuswantoro

Setiawan, Iwan. 2006. Perancangan Software Embeded System Berbasis FSM. *Laporan* Tidak Diterbitkan. Semarang: Elektro FT Universitas Diponegoro

Syamsudin, Ahmad. 2011. Rancang Bangun Game Edukasi Cisco Router "Melacak Koruptor" Menggunakan Metode Top-Down Parsing. *Skripsi* Tidak Diterbitkan. Malang: Jurusan TI Fakultas Saintek UIN Maulana Malik Ibrahim Malang

Sykes, Edward F dan Franya Franek. Web-Based Architecture of an Intelligent Tutoring System for Remote Students Learning to Program Java. *Sheridan College & McMaster University*

Taru, Andi NNW. 2010. *Pemrograman Game dengan Java dan GTGE*. Yogyakarta: Penerbit Andi.

Tuerah, Paulus. http://goldenstudios.or.id/products/GTGE/

Utdirartatmo, Firrar. 2005. *Teknik Kompilasi*. Yogyakarta: Graha Ilmu.

Utdirartatmo, Firrar. 2001. *Teori Bahasa dan Otomata*. Yogyakarta: Graha Ilmu.

The Pragmatic Programmers

The Definitive ANTLR 4 Reference

Terence Parr

Edited by Susannah Davidson Pfalzer

UNIVERSITAS ISLAM NEGERI
MAULANA MALIK IBRAHIM
PUSAT PERPUSTAKAAN

Early Praise for *The Definitive ANTLR 4 Reference*

Parr's clear writing and lighthearted style make it a pleasure to learn the practical details of building language processors.

➤ **Dan Bornstein**
Designer of the Dalvik VM for Android

ANTLR is an exceptionally powerful and flexible tool for parsing formal languages. At Twitter, we use it exclusively for query parsing in our search engine. Our grammars are clean and concise, and the generated code is efficient and stable. This book is our go-to reference for ANTLR v4—engaging writing, clear descriptions, and practical examples all in one place.

➤ **Samuel Luckenbill**
Senior manager of search infrastructure, Twitter, Inc.

ANTLR v4 really makes parsing easy, and this book makes it even easier. It explains every step of the process, from designing the grammar to making use of the output.

➤ **Niko Matsakis**
Core contributor to the Rust language and researcher at Mozilla Research

I sure wish I had ANTLR 4 and this book four years ago when I started to work on a C++ grammar in the NetBeans IDE and the Sun Studio IDE. Excellent content and very readable.

➤ **Nikolay Krasilnikov**
Senior software engineer, Oracle Corp.

This book is an absolute requirement for getting the most out of ANTLR. I refer to it constantly whenever I'm editing a grammar.

➤ **Rich Unger**
Principal member of technical staff, Apex Code team, Salesforce.com

I have been using ANTLR to create languages for six years now, and the new v4 is absolutely wonderful. The best news is that Terence has written this fantastic book to accompany the software. It will please newbies and experts alike. If you process data or implement languages, do yourself a favor and buy this book!

➤ **Rahul Gidwani**
Senior software engineer, Xoom Corp.

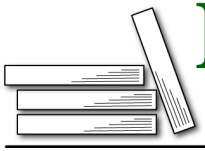
Never have the complexities surrounding parsing been so simply explained. This book provides brilliant insight into the ANTLR v4 software, with clear explanations from installation to advanced usage. An array of real-life examples, such as JSON and R, make this book a must-have for any ANTLR user.

➤ **David Morgan**
Student, computer and electronic systems, University of Strathclyde

The Definitive ANTLR 4 Reference

Terence Parr

The Pragmatic Bookshelf
Dallas, Texas • Raleigh, North Carolina



Pragmatic Bookshelf

Our Pragmatic courses, workshops, and other products can help you and your team create better software and have more fun. For more information, as well as the latest Pragmatic titles, please visit us at *http://pragprog.com*.

The team that produced this book includes:

Susannah Pfalzer (editor)
Potomac Indexing, LLC (indexer)
Kim Wimpsett (copyeditor)
David J Kelly (typesetter)
Janet Furlow (producer)
Juliet Benda (rights)
Ellie Callahan (support)

Printed in the United States of America.
ISBN-13: 978-1-93435-699-9
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—January 2013

Contents

Part II — Developing Language Applications with ANTLR Grammars

Part III — Advanced Topics

Part IV — ANTLR Reference

Acknowledgments

It's been roughly 25 years since I started working on ANTLR. In that time, many people have helped shape the tool syntax and functionality, for which I'm most grateful. Most importantly for ANTLR version 4, Sam Harwell[1] was my coauthor. He helped write the software but also made critical contributions to the Adaptive *LL(*)* grammar analysis algorithm. Sam is also building the ANTLRWorks2 grammar IDE.

The following people provided technical reviews: Oliver Ziegermann, Sam Rose, Kyle Ferrio, Maik Schmidt, Colin Yates, Ian Dees, Tim Ottinger, Kevin Gisi, Charley Stran, Jerry Kuch, Aaron Kalair, Michael Bevilacqua-Linn, Javier Collado, Stephen Wolff, and Bernard Kaiflin. I also appreciate those people who reported errors in beta versions of the book and v4 software. Kim Shrier and Graham Wideman deserve special attention because they provided such detailed reviews. Graham's technical reviews were so elaborate, voluminous, and extensive that I wasn't sure whether to shake his hand vigorously or go buy a handgun.

Finally, I'd like to thank Pragmatic Bookshelf editor Susannah Davidson Pfalzer, who has stuck with me through three books! Her suggestions and careful editing really improved this book.

1. http://tunnelvisionlabs.com

Welcome Aboard!

ANTLR v4 is a powerful parser generator that you can use to read, process, execute, or translate structured text or binary files. It's widely used in academia and industry to build all sorts of languages, tools, and frameworks. Twitter search uses ANTLR for query parsing, with more than 2 billion queries a day. The languages for Hive and Pig and the data warehouse and analysis systems for Hadoop all use ANTLR. Lex Machina[1] uses ANTLR for information extraction from legal texts. Oracle uses ANTLR within the SQL Developer IDE and its migration tools. The NetBeans IDE parses C++ with ANTLR. The HQL language in the Hibernate object-relational mapping framework is built with ANTLR.

Aside from these big-name, high-profile projects, you can build all sorts of useful tools such as configuration file readers, legacy code converters, wiki markup renderers, and JSON parsers. I've built little tools for creating object-relational database mappings, describing 3D visualizations, and injecting profiling code into Java source code, and I've even done a simple DNA pattern matching example for a lecture.

From a formal language description called a *grammar*, ANTLR generates a parser for that language that can automatically build parse trees, which are data structures representing how a grammar matches the input. ANTLR also automatically generates tree walkers that you can use to visit the nodes of those trees to execute application-specific code.

This book is both a reference for ANTLR v4 and a guide to using it to solve language recognition problems. You're going to learn how to do the following:

- Identify grammar patterns in language samples and reference manuals in order to build your own grammars.

1. http://lexmachina.com

- Build grammars for simple languages like JSON all the way up to complex programming languages like R. You'll also solve some tricky recognition problems from Python and XML.
- Implement language applications based upon those grammars by walking the automatically generated parse trees.
- Customize recognition error handling and error reporting for specific application domains.
- Take absolute control over parsing by embedding Java actions into a grammar.

Unlike a textbook, the discussions are example-driven in order to make things more concrete and to provide starter kits for building your own language applications.

Who Is This Book For?

This book is specifically targeted at any programmer interested in learning how to build data readers, language interpreters, and translators. This book is about how to build things with ANTLR specifically, of course, but you'll learn a lot about lexers and parsers in general. Beginners and experts alike will need this book to use ANTLR v4 effectively. To get your head around the advanced topics in Part III, you'll need some experience with ANTLR by working through the earlier chapters. Readers should know Java to get the most out of the book.

The Honey Badger Release

ANTLR v4 is named the "Honey Badger" release after the fearless hero of the YouTube sensation *The Crazy Nastyass Honey Badger*.[a] It takes whatever grammar you give it; it doesn't give a damn!

a. http://www.youtube.com/watch?v=4r7wHMg5Yjg

What's So Cool About ANTLR V4?

The v4 release of ANTLR has some important new capabilities that reduce the learning curve and make developing grammars and language applications much easier. The most important new feature is that ANTLR v4 gladly accepts every grammar you give it (with one exception regarding indirect left recursion, described shortly). There are no grammar conflict or ambiguity warnings as ANTLR translates your grammar to executable, human-readable parsing code.

If you give your ANTLR-generated parser valid input, the parser will always recognize the input properly, no matter how complicated the grammar. Of course, it's up to you to make sure the grammar accurately describes the language in question.

ANTLR parsers use a new parsing technology called *Adaptive LL(*)* or *ALL(*)* ("all star") that I developed with Sam Harwell.[2] *ALL(*)* is an extension to v3's *LL(*)* that performs grammar analysis dynamically at runtime rather than statically, before the generated parser executes. Because *ALL(*)* parsers have access to actual input sequences, they can always figure out how to recognize the sequences by appropriately weaving through the grammar. Static analysis, on the other hand, has to consider all possible (infinitely long) input sequences.

In practice, having *ALL(*)* means you don't have to contort your grammars to fit the underlying parsing strategy as you would with most other parser generator tools, including ANTLR v3. If you've ever pulled your hair out because of an ambiguity warning in ANTLR v3 or a reduce/reduce conflict in `yacc`, ANTLR v4 is for you!

The next awesome new feature is that ANTLR v4 dramatically simplifies the grammar rules used to match syntactic structures like programming language arithmetic expressions. Expressions have always been a hassle to specify with ANTLR grammars (and to recognize by hand with recursive-descent parsers). The most natural grammar to recognize expressions is invalid for traditional top-down parser generators like ANTLR v3. Now, with v4, you can match expressions with rules that look like this:

```
expr : expr '*' expr  // match subexpressions joined with '*' operator
     | expr '+' expr  // match subexpressions joined with '+' operator
     | INT            // matches simple integer atom
     ;
```

Self-referential rules like `expr` are recursive and, in particular, *left recursive* because at least one of its alternatives immediately refers to itself.

ANTLR v4 automatically rewrites left-recursive rules such as `expr` into non-left-recursive equivalents. The only constraint is that the left recursion must be direct, where rules immediately reference themselves. Rules cannot reference another rule on the left side of an alternative that eventually comes back to reference the original rule without matching a token. See Section 5.4, *Dealing with Precedence, Left Recursion, and Associativity*, on page 69 for more details.

2. http://tunnelvisionlabs.com

In addition to those two grammar-related improvements, ANTLR v4 makes it much easier to build language applications. ANTLR-generated parsers automatically build convenient representations of the input called *parse trees* that an application can walk to trigger code snippets as it encounters constructs of interest. Previously, v3 users had to augment the grammar with tree construction operations. In addition to building trees automatically, ANTLR v4 also automatically generates parse-tree walkers in the form of *listener* and *visitor pattern* implementations. Listeners are analogous to XML document handler objects that respond to SAX events triggered by XML parsers.

ANTLR v4 is much easier to learn because of those awesome new features but also because of what it does not carry forward from v3.

- The biggest change is that v4 deemphasizes embedding actions (code) in the grammar, favoring listeners and visitors instead. The new mechanisms decouple grammars from application code, nicely encapsulating an application instead of fracturing it and dispersing the pieces across a grammar. Without embedded actions, you can also reuse the same grammar in different applications without even recompiling the generated parser. ANTLR still allows embedded actions, but doing so is considered advanced in v4. Such actions give the highest level of control but at the cost of losing grammar reuse.
- Because ANTLR automatically generates parse trees and tree walkers, there's no need for you to build tree grammars in v4. You get to use familiar design patterns like the visitor instead. This means that once you've learned ANTLR grammar syntax, you get to move back into the comfortable and familiar realm of the Java programming language to implement the actual language application.
- ANTLR v3's *LL(*)* parsing strategy is weaker than v4's *ALL(*)*, so v3 sometimes relied on backtracking to properly parse input phrases. Backtracking makes it hard to debug a grammar by stepping through the generated parser because the parser might parse the same input multiple times (recursively). Backtracking can also make it harder for the parser to give a good error message upon invalid input.

ANTLR v4 is the result of a minor detour (twenty-five years) I took in graduate school. I guess I'm going to have to change my motto slightly.

> Why program by hand in five days what you can spend *twenty-five* years of your life automating?

ANTLR v4 is exactly what I want in a parser generator, so I can finally get back to the problem I was originally trying to solve in the 1980s. Now, if I could just remember what that was.

What's in This Book?

This book is the best, most complete source of information on ANTLR v4 that you'll find anywhere. The free, online documentation provides enough to learn the basic grammar syntax and semantics but doesn't explain ANTLR concepts in detail. Only this book explains how to identify grammar patterns in languages and how to express them as ANTLR grammars. The examples woven throughout the text give you the leg up you need to start building your own language applications. This book helps you get the most out of ANTLR and is required reading to become an advanced user.

This book is organized into four parts.

- Part I introduces ANTLR, provides some background knowledge about languages, and gives you a tour of ANTLR's capabilities. You'll get a taste of the syntax and what you can do with it.
- Part II is all about designing grammars and building language applications using those grammars in combination with tree walkers.
- Part III starts out by showing you how to customize the error handling of ANTLR-generated parsers. Next, you'll learn how to embed actions in the grammar because sometimes it's simpler or more efficient to do so than building a tree and walking it. Related to actions, you'll also learn how to use *semantic predicates* to alter the behavior of the parser to handle some challenging recognition problems.

  The final chapter solves some challenging language recognition problems, such as recognizing XML and context-sensitive newlines in Python.
- Part IV is the reference section and lays out all of the rules for using the ANTLR grammar meta-language and its runtime library.

Readers who are totally new to grammars and language tools should definitely start by reading Chapter 1, *Meet ANTLR*, on page 3 and Chapter 2, *The Big Picture*, on page 9. Experienced ANTLR v3 users can jump directly to Chapter 4, *A Quick Tour*, on page 31 to learn more about v4's new capabilities.

The source code for all examples in this book is available online. For those of you reading this electronically, you can click the box above the source code, and it will display the code in a browser window. If you're reading the paper version of this book or would simply like a complete bundle of the code, you

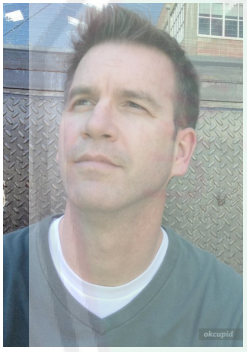
can grab it at the book website.[3] To focus on the key elements being discussed, most of the code snippets shown in the book itself are partial. The downloads show the full source.

Also be aware that all files have a copyright notice as a comment at the top, which kind of messes up the sample input files. Please remove the copyright notice from files, such as t.properties in the listeners code subdirectory, before using them as input to the parsers described in this book. Readers of the electronic version can also cut and paste from the book, which does not display the copyright notice, as shown here:

listeners/t.properties
```
user="parrt"
machine="maniac"
```

Learning More About ANTLR Online

At the http://www.antlr.org website, you'll find the ANTLR download, the ANTLRWorks2 graphical user interface (GUI) development environment, documentation, prebuilt grammars, examples, articles, and a file-sharing area. The tech support mailing list[4] is a newbie-friendly public Google group.



Terence Parr
University of San Francisco, November 2012

3. http://pragprog.com/titles/tpantlr2/source_code
4. https://groups.google.com/d/forum/antlr-discussion

Part I

Introducing ANTLR and Computer Languages

In Part I, we'll get ANTLR installed, try it on a simple "hello world" grammar, and look at the big picture of language application development. With those basics down, we'll build a grammar to recognize and translate lists of integers in curly braces like {1, 2, 3}. Finally, we'll take a whirlwind tour of ANTLR features by racing through a number of simple grammars and applications.

CHAPTER 1

Meet ANTLR

Our goals in this first part of the book are to get a general overview of ANTLR's capabilities and to explore language application architecture. Once we have the big picture, we'll learn ANTLR slowly and systematically in Part II using lots of real-world examples. To get started, let's install ANTLR and then try it on a simple "hello world" grammar.

1.1 Installing ANTLR

ANTLR is written in Java, so you need to have Java installed before you begin.[1] This is true even if you're going to use ANTLR to generate parsers in another language such as C# or C++. (I expect to have other targets in the near future.) ANTLR requires Java version 1.6 or newer.

Why This Book Uses the Command-Line Shell

Throughout this book, we'll be using the command line (shell) to run ANTLR and build our applications. Since programmers use a variety of development environments and operating systems, the operating system shell is the only "interface" we have in common. Using the shell also makes each step in the language application development and build process explicit. I'll be using the Mac OS X shell throughout for consistency, but the commands should work in any Unix shell and, with trivial variations, on Windows.

Installing ANTLR itself is a matter of downloading the latest jar, such as antlr-4.0-complete.jar,[2] and storing it somewhere appropriate. The jar contains all dependencies necessary to run the ANTLR tool and the runtime library

1. http://www.java.com/en/download/help/download_options.xml
2. See http://www.antlr.org/download.html, but you can also build ANTLR from the source by pulling from https://github.com/antlr/antlr4.

needed to compile and execute recognizers generated by ANTLR. In a nutshell, the ANTLR tool converts grammars into programs that recognize sentences in the language described by the grammar. For example, given a grammar for JSON, the ANTLR tool generates a program that recognizes JSON input using some support classes from the ANTLR runtime library.

The jar also contains two support libraries: a sophisticated tree layout library[3] and StringTemplate,[4] a template engine useful for generating code and other structured text (see the sidebar *The StringTemplate Engine*, on page 4). At version 4.0, ANTLR is still written in ANTLR v3, so the complete jar contains the previous version of ANTLR as well.

The StringTemplate Engine

StringTemplate is a Java template engine (with ports for C#, Python, Ruby, and Scala) for generating source code, web pages, emails, or any other formatted text output. StringTemplate is particularly good at multitargeted code generators, multiple site skins, and internationalization/localization. It evolved over years of effort developing jGuru.com. StringTemplate also generates that website and powers the ANTLR v3 and v4 code generators. See the About[a] page on the website for more information.

a. http://www.stringtemplate.org/about.html

You can manually download ANTLR from the ANTLR website using a web browser, or you can use the command-line tool curl to grab it.

```
$ cd /usr/local/lib
$ curl -O http://www.antlr.org/download/antlr-4.0-complete.jar
```

On Unix, /usr/local/lib is a good directory to store jars like ANTLR's. On Windows, there doesn't seem to be a standard directory, so you can simply store it in your project directory. Most development environments want you to drop the jar into the dependency list of your language application project. There is no configuration script or configuration file to alter—you just need to make sure that Java knows how to find the jar.

Because this book uses the command line throughout, you need to go through the typical onerous process of setting the CLASSPATH[5] environment variable. With CLASSPATH set, Java can find both the ANTLR tool and the runtime library. On Unix systems, you can execute the following from the shell or add it to the shell startup script (.bash_profile for bash shell):

3. http://code.google.com/p/treelayout
4. http://www.stringtemplate.org
5. http://docs.oracle.com/javase/tutorial/essential/environment/paths.html

```
$ export CLASSPATH=".:/usr/local/lib/antlr-4.0-complete.jar:$CLASSPATH"
```

It's critical to have the dot, the current directory identifier, somewhere in the CLASSPATH. Without that, the Java compiler and Java virtual machine won't see classes in the current directory. You'll be compiling and testing things from the current directory all the time in this book.

You can check to see that ANTLR is installed correctly now by running the ANTLR tool without arguments. You can either reference the jar directly with the java -jar option or directly invoke the org.antlr.v4.Tool class.

```
$ java -jar /usr/local/lib/antlr-4.0-complete.jar  # launch org.antlr.v4.Tool
ANTLR Parser Generator  Version 4.0
 -o ___              specify output directory where all output is generated
 -lib ___            specify location of .tokens files
...
$ java org.antlr.v4.Tool    # launch org.antlr.v4.Tool
ANTLR Parser Generator  Version 4.0
-o ___              specify output directory where all output is generated
-lib ___            specify location of .tokens files
...
```

Typing either of those java commands to run ANTLR all the time would be painful, so it's best to make an alias or shell script. Throughout the book, I'll use alias antlr4, which you can define as follows on Unix:

```
$ alias antlr4='java -jar /usr/local/lib/antlr-4.0-complete.jar'
```

Or, you could put the following script into /usr/local/bin (readers of the ebook can click the install/antlr4 title bar to get the file):

install/antlr4

```
#!/bin/sh
java -cp "/usr/local/lib/antlr4-complete.jar:$CLASSPATH" org.antlr.v4.Tool $*
```

On Windows you can do something like this (assuming you put the jar in C:\libraries):

install/antlr4.bat

```
java -cp C:\libraries\antlr-4.0-complete.jar;%CLASSPATH% org.antlr.v4.Tool %*
```

Either way, you get to say just antlr4.

```
$ antlr4
ANTLR Parser Generator  Version 4.0
-o ___              specify output directory where all output is generated
-lib ___            specify location of .tokens files
...
```

If you see the help message, then you're ready to give ANTLR a quick test-drive!

1.2 Executing ANTLR and Testing Recognizers

Here's a simple grammar that recognizes phrases like hello parrt and hello world:

install/Hello.g4

```
grammar Hello;            // Define a grammar called Hello
r  : 'hello' ID ;         // match keyword hello followed by an identifier
ID : [a-z]+ ;             // match lower-case identifiers
WS : [ \t\r\n]+ -> skip ; // skip spaces, tabs, newlines, \r (Windows)
```

To keep things tidy, let's put grammar file Hello.g4 in its own directory, such as /tmp/test. Then we can run ANTLR on it and compile the results.

```
$ cd /tmp/test
$ # copy-n-paste Hello.g4 or download the file into /tmp/test
$ antlr4 Hello.g4  # Generate parser and lexer using antlr4 alias from before
$ ls
Hello.g4               HelloLexer.java     HelloParser.java
Hello.tokens           HelloLexer.tokens
HelloBaseListener.java HelloListener.java
$ javac *.java     # Compile ANTLR-generated code
```

Running the ANTLR tool on Hello.g4 generates an executable recognizer embodied by HelloParser.java and HelloLexer.java, but we don't have a main program to trigger language recognition. (We'll learn what parsers and lexers are in the next chapter.) That's the typical case at the start of a project. You'll play around with a few different grammars before building the actual application. It'd be nice to avoid having to create a main program to test every new grammar.

ANTLR provides a flexible testing tool in the runtime library called TestRig. It can display lots of information about how a recognizer matches input from a file or standard input. TestRig uses Java reflection to invoke compiled recognizers. Like before, it's a good idea to create a convenient alias or batch file. I'm going to call it grun throughout the book (but you can call it whatever you want).

```
$ alias grun='java org.antlr.v4.runtime.misc.TestRig'
```

The test rig takes a grammar name, a starting rule name kind of like a main() method, and various options that dictate the output we want. Let's say we'd like to print the tokens created during recognition. Tokens are vocabulary symbols like keyword hello and identifier parrt. To test the grammar, start up grun as follows:

```
⇒ $ grun Hello r -tokens     # start the TestRig on grammar Hello at rule r
⇒ hello parrt                # input for the recognizer that you type
⇒ EOF                        # type ctrl-D on Unix or Ctrl+Z on Windows
```

```
❮ [@0,0:4='hello',<1>,1:0]   # these three lines are output from grun
  [@1,6:10='parrt',<2>,1:6]
  [@2,12:11='<EOF>',<-1>,2:0]
```

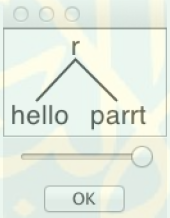
After you hit a newline on the grun command, the computer will patiently wait for you to type in hello parrt followed by a newline. At that point, you must type the end-of-file character to terminate reading from standard input; otherwise, the program will stare at you for eternity. Once the recognizer has read all of the input, TestRig prints out the list of tokens per the use of option -tokens on grun.

Each line of the output represents a single token and shows everything we know about the token. For example, [@1,6:10='parrt',<2>,1:6] indicates that the token is the second token (indexed from 0), goes from character position 6 to 10 (inclusive starting from 0), has text parrt, has token type 2 (ID), is on line 1 (from 1), and is at character position 6 (starting from zero and counting tabs as a single character).

We can print the parse tree in LISP-style text form (*root children*) just as easily.

```
⇛ $ grun Hello r -tree
⇛ hello parrt
⇛ EOF
❮ (r hello parrt)
```

The easiest way to see how a grammar recognizes the input, though, is by looking at the parse tree visually. Running TestRig with the grun -gui option, grun Hello r -gui, produces the following dialog box:



Running TestRig without any command-line options prints a small help message.

```
$ grun
java org.antlr.v4.runtime.misc.TestRig GrammarName startRuleName
  [-tokens] [-tree] [-gui] [-ps file.ps] [-encoding encodingname]
  [-trace] [-diagnostics] [-SLL]
  [input-filename(s)]
Use startRuleName='tokens' if GrammarName is a lexer grammar.
Omitting input-filename makes rig read from stdin.
```

As we go along in the book, we'll use many of those options; here's briefly what they do:

-tokens prints out the token stream.

-tree prints out the parse tree in LISP form.

-gui displays the parse tree visually in a dialog box.

-ps file.ps generates a visual representation of the parse tree in PostScript and stores it in file.ps. The parse tree figures in this chapter were generated with -ps.

-encoding encodingname specifies the test rig input file encoding if the current locale would not read the input properly. For example, we need this option to parse a Japanese-encoded XML file in Section 12.4, *Parsing and Lexing XML*, on page 224.

-trace prints the rule name and current token upon rule entry and exit.

-diagnostics turns on diagnostic messages during parsing. This generates messages only for unusual situations such as ambiguous input phrases.

-SLL uses a faster but slightly weaker parsing strategy.

Now that we have ANTLR installed and have tried it on a simple grammar, let's take a step back to look at the big picture and learn some important terminology in the next chapter. After that, we'll try a simple starter project that recognizes and translates lists of integers such as {1, 2, 3}. Then, we'll walk through a number of interesting examples in Chapter 4, *A Quick Tour*, on page 31 that demonstrate ANTLR's capabilities and that illustrate a few of the domains where ANTLR applies.

CHAPTER 2

The Big Picture

Now that we have ANTLR installed and some idea of how to build and run a small example, we're going to look at the big picture. In this chapter, we'll learn about the important processes, terminology, and data structures associated with language applications. As we go along, we'll identify the key ANTLR objects and learn a little bit about what ANTLR does for us behind the scenes.

2.1 Let's Get Meta!

To implement a language, we have to build an application that reads sentences and reacts appropriately to the phrases and input symbols it discovers. (A language is a set of valid sentences, a sentence is made up of phrases, and a phrase is made up of subphrases and vocabulary symbols.) Broadly speaking, if an application computes or "executes" sentences, we call that application an *interpreter*. Examples include calculators, configuration file readers, and Python interpreters. If we're converting sentences from one language to another, we call that application a *translator*. Examples include Java to C# converters and compilers.

To react appropriately, the interpreter or translator has to recognize all of the valid sentences, phrases, and subphrases of a particular language. Recognizing a phrase means we can identify the various components and can differentiate it from other phrases. For example, we recognize input `sp = 100;` as a programming language assignment statement. That means we know that `sp` is the assignment target and `100` is the value to store. Similarly, if we were recognizing English sentences, we'd identify the parts of speech, such as the subject, predicate, and object. Recognizing assignment `sp = 100;` also means that the language application sees it as clearly distinct from, say, an import statement. After recognition, the application would then perform a suitable operation such as `performAssignment("sp", 100)` or `translateAssignment("sp", 100)`.

Programs that recognize languages are called *parsers* or *syntax analyzers*. *Syntax* refers to the rules governing language membership, and in this book we're going to build ANTLR *grammars* to specify language syntax. A grammar is just a set of rules, each one expressing the structure of a phrase. The ANTLR tool translates grammars to parsers that look remarkably similar to what an experienced programmer might build by hand. (ANTLR is a program that writes other programs.) Grammars themselves follow the syntax of a language optimized for specifying other languages: ANTLR's *meta-language*.

Parsing is much easier if we break it down into two similar but distinct tasks or stages. The separate stages mirror how our brains read English text. We don't read a sentence character by character. Instead, we perceive a sentence as a stream of words. The human brain subconsciously groups character sequences into words and looks them up in a dictionary before recognizing grammatical structure. This process is more obvious if we're reading Morse code because we have to convert the dots and dashes to characters before reading a message. It's also obvious when reading long words such as *Humuhumunukunukuapua'a*, the Hawaiian state fish.

The process of grouping characters into words or symbols (*tokens*) is called *lexical analysis* or simply *tokenizing*. We call a program that tokenizes the input a *lexer*. The lexer can group related tokens into token classes, or *token types*, such as `INT` (integers), `ID` (identifiers), `FLOAT` (floating-point numbers), and so on. The lexer groups vocabulary symbols into types when the parser cares only about the type, not the individual symbols. Tokens consist of at least two pieces of information: the token type (identifying the lexical structure) and the text matched for that token by the lexer.

The second stage is the actual parser and feeds off of these tokens to recognize the sentence structure, in this case an assignment statement. By default, ANTLR-generated parsers build a data structure called a *parse tree* or *syntax tree* that records how the parser recognized the structure of the input sentence and its component phrases. The following diagram illustrates the basic data flow of a language recognizer:

The interior nodes of the parse tree are phrase names that group and identify their children. The root node is the most abstract phrase name, in this case *stat* (short for "statement"). The leaves of a parse tree are always the input tokens. Sentences, linear sequences of symbols, are really just serializations of parse trees we humans grok natively in hardware. To get an idea across to someone, we have to conjure up the same parse tree in their heads using a word stream.

By producing a parse tree, a parser delivers a handy data structure to the rest of the application that contains complete information about how the parser grouped the symbols into phrases. Trees are easy to process in subsequent steps and are well understood by programmers. Better yet, the parser can generate parse trees automatically.

By operating off parse trees, multiple applications that need to recognize the same language can reuse a single parser. The other choice is to embed application-specific code snippets directly into the grammar, which is what parser generators have done traditionally. ANTLR v4 still allows this (see Chapter 10, *Attributes and Actions*, on page 175), but parse trees make for a much tidier and more decoupled design.

Parse trees are also useful for translations that require multiple passes (tree walks) because of computation dependencies where one stage needs information from a previous stage. In other cases, an application is just a heck of a lot easier to code and test in multiple stages because it's so complex. Rather than reparse the input characters for each stage, we can just walk the parse tree multiple times, which is much more efficient.

Because we specify phrase structure with a set of rules, parse-tree subtree roots correspond to grammar rule names. As a preview of things to come, here's the grammar rule that corresponds to the first level of the *assign* subtree from the diagram:

```
assign : ID '=' expr ';' ; // match an assignment statement like "sp = 100;"
```

Understanding how ANTLR translates such rules into human-readable parsing code is fundamental to using and debugging grammars, so let's dig deeper into how parsing works.

2.2 Implementing Parsers

The ANTLR tool generates *recursive-descent parsers* from grammar rules such as `assign` that we just saw. Recursive-descent parsers are really just a collection of recursive methods, one per rule. The *descent* term refers to the fact that parsing begins at the root of a parse tree and proceeds toward the leaves

(tokens). The rule we invoke first, the *start symbol*, becomes the root of the parse tree. That would mean calling method stat() for the parse tree in the previous section. A more general term for this kind of parsing is *top-down parsing*; recursive-descent parsers are just one kind of top-down parser implementation.

To get an idea of what recursive-descent parsers look like, here's the (slightly cleaned up) method that ANTLR generates for rule assign:

```
// assign : ID '=' expr ';' ;
void assign() {     // method generated from rule assign
    match(ID);      // compare ID to current input symbol then consume
    match('=');
    expr();         // match an expression by calling expr()
    match(';');
}
```

The cool part about recursive-descent parsers is that the call graph traced out by invoking methods stat(), assign(), and expr() mirrors the interior parse tree nodes. (Take a quick peek back at the parse tree figure.) The calls to match() correspond to the parse tree leaves. To build a parse tree manually in a handbuilt parser, we'd insert "add new subtree root" operations at the start of each rule method and an "add new leaf node" operation to match().

Method assign() just checks to make sure all necessary tokens are present and in the right order. When the parser enters assign(), it doesn't have to choose between more than one *alternative*. An alternative is one of the choices on the right side of a rule definition. For example, the stat rule that invokes assign likely has a list of other kinds of statements.

```
/** Match any kind of statement starting at the current input position */
stat: assign          // First alternative ('|' is alternative separator)
    | ifstat          // Second alternative
    | whilestat
   ...
    ;
```

A parsing rule for stat looks like a switch.

```
void stat() {
    switch ( «current input token» ) {
        CASE ID    : assign(); break;
        CASE IF    : ifstat(); break; // IF is token type for keyword 'if'
        CASE WHILE : whilestat(); break;
        ...
        default    : «raise no viable alternative exception»
    }
}
```

Method stat() has to make a *parsing decision* or *prediction* by examining the next input token. Parsing decisions predict which alternative will be successful. In this case, seeing a WHILE keyword predicts the third alternative of rule stat. Rule method stat() therefore calls whilestat(). You might've heard the term *lookahead token* before; that's just the next input token. A lookahead token is any token that the parser sniffs before matching and consuming it.

Sometimes, the parser needs lots of lookahead tokens to predict which alternative will succeed. It might even have to consider all tokens from the current position until the end of file! ANTLR silently handles all of this for you, but it's helpful to have a basic understanding of decision making so debugging generated parsers is easier.

To visualize parsing decisions, imagine a maze with a single entrance and a single exit that has words written on the floor. Every sequence of words along a path from entrance to exit represents a sentence. The structure of the maze is analogous to the rules in a grammar that define a language. To test a sentence for membership in a language, we compare the sentence's words with the words along the floor as we traverse the maze. If we can get to the exit by following the sentence's words, that sentence is valid.

To navigate the maze, we must choose a valid path at each fork, just as we must choose alternatives in a parser. We have to decide which path to take by comparing the next word or words in our sentence with the words visible down each path emanating from the fork. The words we can see from the fork are analogous to lookahead tokens. The decision is pretty easy when each path starts with a unique word. In rule stat, each alternative begins with a unique token, so stat() can distinguish the alternatives by looking at the first lookahead token.

When the words starting each path from a fork overlap, a parser needs to look further ahead, scanning for words that distinguish the alternatives. ANTLR automatically throttles the amount of lookahead up-and-down as necessary for each decision. If the lookahead is the same down multiple paths to the exit (end of file), there are multiple interpretations of the current input phrase. Resolving such ambiguities is our next topic. After that, we'll figure out how to use parse trees to build language applications.

2.3 You Can't Put Too Much Water into a Nuclear Reactor

An ambiguous phrase or sentence is one that has more than one interpretation. In other words, the words fit more than one grammatical structure. The section title "You Can't Put Too Much Water into a Nuclear Reactor" is an

ambiguous sentence from a *Saturday Night Live* sketch I saw years ago. The characters weren't sure if they should be careful *not* to put too much water into the reactor or if they should put lots of water into the reactor.

For Whom No Thanks Is Too Much

One of my favorite ambiguous sentences is on the dedication page of my friend Kevin's Ph.D. thesis: "To my Ph.D. supervisor, for whom no thanks is too much." It's unclear whether he was grateful or ungrateful. Kevin claimed it was the latter, so I asked why he had taken a postdoc job working for the same guy. His reply: "Revenge."

Ambiguity can be funny in natural language but causes problems for computer-based language applications. To interpret or translate a phrase, a program has to uniquely identify the meaning. That means we have to provide unambiguous grammars so that the generated parser can match each input phrase in exactly one way.

We haven't studied grammars in detail yet, but let's include a few ambiguous grammars here to make the notion of ambiguity more concrete. You can refer to this section if you run into ambiguities later when building a grammar.

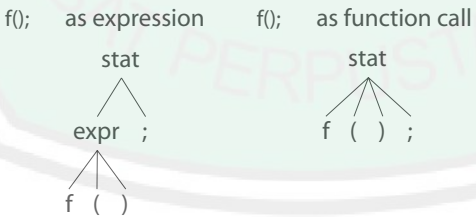
Some ambiguous grammars are obvious.

```
stat: ID '=' expr ';'  // match an assignment; can match "f();"
    | ID '=' expr ';'  // oops! an exact duplicate of previous alternative
    ;
expr: INT ;
```

Most of the time, though, the ambiguity will be more subtle, as in the following grammar that can match a function call via both alternatives of rule stat:

```
stat: expr ';'        // expression statement
    | ID '(' ')' ';'  // function call statement
    ;
expr: ID '(' ')'
    | INT
    ;
```

Here are the two interpretations of input f(); starting in rule stat:



The parse tree on the left shows the case where f() matches to rule expr. The tree on the right shows f() matching to the start of rule stat's second alternative.

Since most language inventors design their syntax to be unambiguous, an ambiguous grammar is analogous to a programming bug. We need to reorganize the grammar to present a single choice to the parser for each input phrase. If the parser detects an ambiguous phrase, it has to pick one of the viable alternatives. ANTLR resolves the ambiguity by choosing the first alternative involved in the decision. In this case, the parser would choose the interpretation of f(); associated with the parse tree on the left.

Ambiguities can occur in the lexer as well as the parser, but ANTLR resolves them so the rules behave naturally. ANTLR resolves lexical ambiguities by matching the input string to the rule specified first in the grammar. To see how this works, let's look at an ambiguity that's common to most programming languages: the ambiguity between keywords and identifier rules. Keyword begin (followed by a nonletter) is also an identifier, at least lexically, so the lexer can match b-e-g-i-n to either rule.

```
BEGIN : 'begin' ;  // match b-e-g-i-n sequence; ambiguity resolves to BEGIN
ID    : [a-z]+ ;   // match one or more of any lowercase letter
```

For more on this lexical ambiguity, see *Matching Identifiers*, on page 74.

Note that lexers try to match the longest string possible for each token, meaning that input beginner would match only to rule ID. The lexer would not match beginner as BEGIN followed by an ID matching input ner.

Sometimes the syntax for a language is just plain ambiguous and no amount of grammar reorganization will change that fact. For example, the natural grammar for arithmetic expressions can interpret input such as 1+2*3 in two ways, either by performing the operations left to right (as Smalltalk does) or in precedence order like most languages. We'll learn how to implicitly specify the operator precedence order for expressions in Section 5.4, *Dealing with Precedence, Left Recursion, and Associativity*, on page 69.

The venerable C language exhibits another kind of ambiguity, which we can resolve using context information such as how an identifier is defined. Consider the code snippet i*j;. Syntactically, it looks like an expression, but its meaning, or semantics, depends on whether i is a type name or variable. If i is a type name, then the snippet isn't an expression. It's a declaration of variable j as a pointer to type i. We'll see how to resolve these ambiguities in Chapter 11, *Altering the Parse with Semantic Predicates*, on page 189.

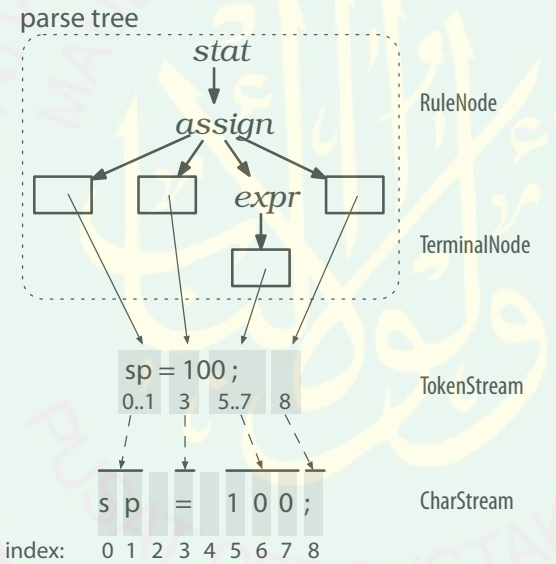
Parsers by themselves test input sentences only for language membership and build a parse tree. That's crucial stuff, but it's time to see how language applications use parse trees to interpret or translate the input.

2.4 Building Language Applications Using Parse Trees

To make a language application, we have to execute some appropriate code for each input phrase or subphrase. The easiest way to do that is to operate on the parse tree created automatically by the parser. The nice thing about operating on the tree is that we're back in familiar Java territory. There's no further ANTLR syntax to learn in order to build an application.

Let's start by looking more closely at the data structures and class names ANTLR uses for recognition and for parse trees. A passing familiarity with the data structures will make future discussions more concrete.

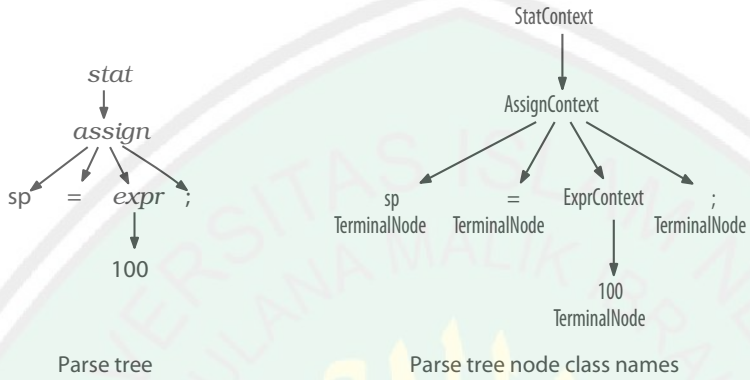
Earlier we learned that lexers process characters and pass tokens to the parser, which in turn checks syntax and creates a parse tree. The corresponding ANTLR classes are `CharStream`, `Lexer`, `Token`, `Parser`, and `ParseTree`. The "pipe" connecting the lexer and parser is called a `TokenStream`. The diagram below illustrates how objects of these types connect to each other in memory.



These ANTLR data structures share as much data as possible to reduce memory requirements. The diagram shows that leaf (token) nodes in the parse tree are containers that point at tokens in the token stream. The tokens record start and stop character indexes into the `CharStream`, rather than making copies

of substrings. There are no tokens associated with whitespace characters (indexes 2 and 4) since we can assume our lexer tosses out whitespace.

The figure also shows ParseTree subclasses RuleNode and TerminalNode that correspond to subtree roots and leaf nodes. RuleNode has familiar methods such as getChild() and getParent(), but RuleNode isn't specific to a particular grammar. To better support access to the elements within specific nodes, ANTLR generates a RuleNode subclass for each rule. The following figure shows the specific classes of the subtree roots for our assignment statement example, which are StatContext, AssignContext, and ExprContext:



These are called *context* objects because they record everything we know about the recognition of a phrase by a rule. Each context object knows the start and stop tokens for the recognized phrase and provides access to all of the elements of that phrase. For example, AssignContext provides methods ID() and expr() to access the identifier node and expression subtree.

Given this description of the concrete types, we could write code by hand to perform a depth-first walk of the tree. We could perform whatever actions we wanted as we discovered and finished nodes. Typical operations are things such as computing results, updating data structures, or generating output. Rather than writing the same tree-walking boilerplate code over again for each application, though, we can use the tree-walking mechanisms that ANTLR generates automatically.

2.5 Parse-Tree Listeners and Visitors

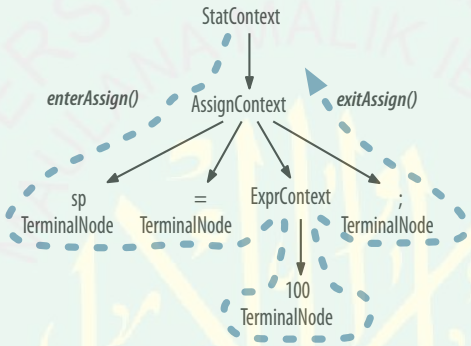
ANTLR provides support for two tree-walking mechanisms in its runtime library. By default, ANTLR generates a parse-tree *listener* interface that responds to events triggered by the built-in tree walker. The listeners themselves are exactly like SAX document handler objects for XML parsers. SAX listeners receive notification of events like startDocument() and endDocument(). The

methods in a listener are just callbacks, such as we'd use to respond to a checkbox click in a GUI application. Once we look at listeners, we'll see how ANTLR can also generate tree walkers that follow the visitor design pattern.[1]

Parse-Tree Listeners

To walk a tree and trigger calls into a listener, ANTLR's runtime provides class ParseTreeWalker. To make a language application, we build a ParseTreeListener implementation containing application-specific code that typically calls into a larger surrounding application.

ANTLR generates a ParseTreeListener subclass specific to each grammar with enter and exit methods for each rule. As the walker encounters the node for rule assign, for example, it triggers enterAssign() and passes it the AssignContext parse-tree node. After the walker visits all children of the assign node, it triggers exitAssign(). The tree diagram shown below shows ParseTreeWalker performing a depth-first walk, represented by the thick dashed line.



It also identifies where in the walk ParseTreeWalker calls the enter and exit methods for rule assign. (The other listener calls aren't shown.)

And the diagram in Figure 1, *ParseTreeWalker call sequence*, on page 19 shows the complete sequence of calls made to the listener by ParseTreeWalker for our statement tree.

The beauty of the listener mechanism is that it's all automatic. We don't have to write a parse-tree walker, and our listener methods don't have to explicitly visit their children.

1. http://en.wikipedia.org/wiki/Visitor_pattern

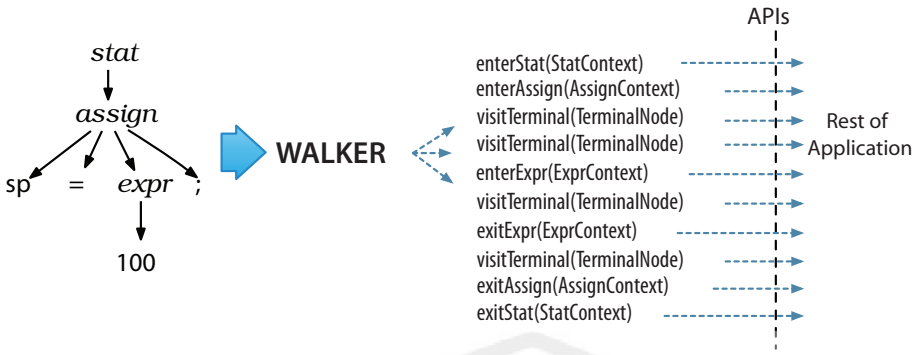


Figure 1—**ParseTreeWalker call sequence**

Parse-Tree Visitors

There are situations, however, where we want to control the walk itself, explicitly calling methods to visit children. Option `-visitor` asks ANTLR to generate a visitor interface from a grammar with a visit method per rule. Here's the familiar visitor pattern operating on our parse tree:

The thick dashed line shows a depth-first walk of the parse tree. The thin dashed lines indicate the method call sequence among the visitor methods. To initiate a walk of the tree, our application-specific code would create a visitor implementation and call `visit()`.

```
ParseTree tree = ... ; // tree is result of parsing
MyVisitor v = new MyVisitor();
v.visit(tree);
```

ANTLR's visitor support code would then call `visitStat()` upon seeing the root node. From there, the `visitStat()` implementation would call `visit()` with the children as arguments to continue the walk. Or, `visitMethod()` could explicitly call `visitAssign()`, and so on.

ANTLR gives us a leg up over writing everything ourselves by generating the visitor interface and providing a class with default implementations for the visitor methods. This way, we avoid having to override every method in the interface, letting us focus on just the methods of interest. We'll learn all about visitors and listeners in Chapter 7, *Decoupling Grammars from Application-Specific Code*, on page 109.

Parsing Terms

This chapter introduced a number of important language recognition terms.

Language A language is a set of valid sentences; sentences are composed of phrases, which are composed of subphrases, and so on.

Grammar A grammar formally defines the syntax rules of a language. Each rule in a grammar expresses the structure of a subphrase.

Syntax tree or parse tree This represents the structure of the sentence where each subtree root gives an abstract name to the elements beneath it. The subtree roots correspond to grammar rule names. The leaves of the tree are symbols or tokens of the sentence.

Token A token is a vocabulary symbol in a language; these can represent a category of symbols such as "identifier" or can represent a single operator or keyword.

Lexer or tokenizer This breaks up an input character stream into tokens. A lexer performs lexical analysis.

Parser A parser checks sentences for membership in a specific language by checking the sentence's structure against the rules of a grammar. The best analogy for parsing is traversing a maze, comparing words of a sentence to words written along the floor to go from entrance to exit. ANTLR generates top-down parsers called *ALL(*)* that can use all remaining input symbols to make decisions. Top-down parsers are goal-oriented and start matching at the rule associated with the coarsest construct, such as `program` or `inputFile`.

Recursive-descent parser This is a specific kind of top-down parser implemented with a function for each rule in the grammar.

Lookahead Parsers use lookahead to make decisions by comparing the symbols that begin each alternative.

So, now we have the big picture. We looked at the overall data flow from character stream to parse tree and identified the key class names in the ANTLR runtime. And we just saw a summary of the listener and visitor mechanisms used to connect parsers with application-specific code. Let's make this all more concrete by working through a real example in the next chapter.

CHAPTER 3

A Starter ANTLR Project

For our first project, let's build a grammar for a tiny subset of C or one of its derivatives like Java. In particular, let's recognize integers in, possibly nested, curly braces like {1, 2, 3} and {1, {2, 3}, 4}. These constructs could be int array or struct initializers. A grammar for this syntax would come in handy in a variety of situations. For one, we could use it to build a source code refactoring tool for C that converted integer arrays to byte arrays if all of the initialized values fit within a byte. We could also use this grammar to convert initialized Java short arrays to strings. For example, we could transform the following:

```
static short[] data = {1,2,3};
```

into the following equivalent string with Unicode constants:

```
static String data = "\u0001\u0002\u0003"; // Java char are unsigned short
```

where Unicode character specifiers, such as \u0001, use four hexadecimal digits representing a 16-bit character value, that is, a short.

The reason we might want to do this translation is to overcome a limitation in the Java .class file format. A Java class file stores array initializers as a sequence of explicit array-element initializers, equivalent to data[0]=1; data[1]=2; data[2]=3;, instead of a compact block of packed bytes.[1] Because Java limits the size of initialization methods, it limits the size of the arrays we can initialize. In contrast, a Java class file stores a string as a contiguous sequence of shorts. Converting array initializers to strings results in a more compact class file and avoids Java's initialization method size limit.

By working through this starter example, you'll learn a bit of ANTLR grammar syntax, what ANTLR generates from a grammar, how to incorporate the

1. To learn more about this topic, check out a video of my JVM Language Summit presentation: http://www.mefeedia.com/watch/24642856.

generated parser into a Java application, and how to build a translator with a parse-tree listener.

3.1 The ANTLR Tool, Runtime, and Generated Code

To get started, let's peek inside ANTLR's jar. There are two key ANTLR components: the ANTLR tool itself and the ANTLR runtime (parse-time) API. When we say "run ANTLR on a grammar," we're talking about running the ANTLR tool, class org.antlr.v4.Tool. Running ANTLR generates code (a parser and a lexer) that recognizes sentences in the language described by the grammar. A lexer breaks up an input stream of characters into tokens and passes them to a parser that checks the syntax. The runtime is a library of classes and methods needed by that generated code such as Parser, Lexer, and Token. First we run ANTLR on a grammar and then compile the generated code against the runtime classes in the jar. Ultimately, the compiled application runs in conjunction with the runtime classes.

The first step to building a language application is to create a grammar that describes a language's syntactic rules (the set of valid sentences). We'll learn how to write grammars in Chapter 5, *Designing Grammars*, on page 57, but for the moment, here's a grammar that'll do what we want:

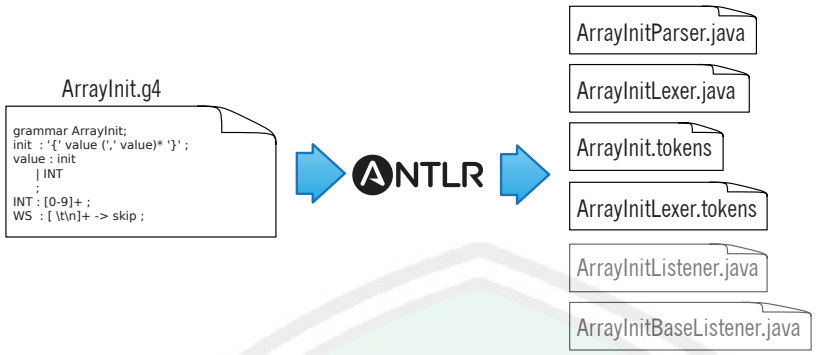
starter/ArrayInit.g4

```
/** Grammars always start with a grammar header. This grammar is called
 *  ArrayInit and must match the filename: ArrayInit.g4
 */
grammar ArrayInit;

/** A rule called init that matches comma-separated values between {...}. */
init  : '{' value (',' value)* '}' ;  // must match at least one value

/** A value can be either a nested array/struct or a simple integer (INT) */
value : init
      | INT
      ;

// parser rules start with lowercase letters, lexer rules with uppercase
INT :   [0-9]+ ;             // Define token INT as one or more digits
WS  :   [ \t\r\n]+ -> skip ; // Define whitespace rule, toss it out
```

Let's put grammar file ArrayInit.g4 in its own directory, such as /tmp/array (by cutting and pasting or downloading the source code from the book website). Then, we can run ANTLR (the tool) on the grammar file.

```
$ cd /tmp/array
$ antlr4 ArrayInit.g4  # Generate parser and lexer using antlr4 alias
```

From grammar ArrayInit.g4, ANTLR generates lots of files that we'd normally have to write by hand.



At this point, we're just trying to get the gist of the development process, so here's a quick description of the generated files:

ArrayInitParser.java This file contains the parser class definition specific to grammar ArrayInit that recognizes our array language syntax.

```
public class ArrayInitParser extends Parser { ... }
```

It contains a method for each rule in the grammar as well as some support code.

ArrayInitLexer.java ANTLR automatically extracts a separate parser and lexer specification from our grammar. This file contains the lexer class definition, which ANTLR generated by analyzing the lexical rules INT and WS as well as the grammar literals '{', ',', and '}'. Recall that the lexer tokenizes the input, breaking it up into vocabulary symbols. Here's the class outline:

```
public class ArrayInitLexer extends Lexer { ... }
```

ArrayInit.tokens ANTLR assigns a token type number to each token we define and stores these values in this file. It's needed when we split a large grammar into multiple smaller grammars so that ANTLR can synchronize all the token type numbers. See *Importing Grammars*, on page 36.

ArrayInitListener.java, ArrayInitBaseListener.java By default, ANTLR parsers build a tree from the input. By walking that tree, a tree walker can fire "events" (callbacks) to a listener object that we provide. ArrayInitListener is the interface that describes the callbacks we can implement. ArrayInitBaseListener is a set of empty default implementations. This class makes it easy for us to override just the callbacks we're interested in. (See Section 7.2, *Implementing Applications with Parse-Tree Listeners*, on page 112.) ANTLR can also

generate tree visitors for us with the -visitor command-line option. (See *Traversing Parse Trees with Visitors*, on page 119.)

We'll use the listener classes to translate short array initializers to String objects shortly (sorry about the pun), but first let's verify that our parser correctly matches some sample input.

ANTLR Grammars Are Stronger Than Regular Expressions

Those of you familiar with regular expressions[a] might be wondering if ANTLR is overkill for such a simple recognition problem. It turns out that we can't use regular expressions to recognize initializations because of nested initializers. Regular expressions have no memory in the sense that they can't remember what they matched earlier in the input. Because of that, they don't know how to match up left and right curlies. We'll get to this in more detail in *Pattern: Nested Phrase*, on page 65.

a. http://en.wikipedia.org/wiki/Regular_expression

3.2 Testing the Generated Parser

Once we've run ANTLR on our grammar, we need to compile the generated Java source code. We can do that by simply compiling everything in our /tmp/array directory.

```
$ cd /tmp/array
$ javac *.java     # Compile ANTLR-generated code
```

If you get a ClassNotFoundException error from the compiler, that means you probably haven't set the Java CLASSPATH correctly. On UNIX systems, you'll need to execute the following command (and likely add to your start-up script such as .bash_profile):

```
$ export CLASSPATH=".:/usr/local/lib/antlr-4.0-complete.jar:$CLASSPATH"
```

To test our grammar, we use the TestRig via alias grun that we saw in the previous chapter. Here's how to print out the tokens created by the lexer:

```
⇒ $ grun ArrayInit init -tokens
⇒ {99, 3, 451}
⇒ EOF
❮ [@0,0:0='{',<1>,1:0]
  [@1,1:2='99',<4>,1:1]
  [@2,3:3=',',<2>,1:3]
  [@3,5:5='3',<4>,1:5]
  [@4,6:6=',',<2>,1:6]
  [@5,8:10='451',<4>,1:8]
  [@6,11:11='}',<3>,1:11]
  [@7,13:12='<EOF>',<-1>,2:0]
```

After typing in array initializer {99, 3, 451}, we have to hit ^{EOF}[2] on a line by itself. By default, ANTLR loads the entire input before processing. (That's the most common case and the most efficient.)

Each line of the output represents a single token and shows everything we know about the token. For example, [@5,8:10='451',<4>,1:8] indicates that it's the token at index 5 (indexed from 0), goes from character position 8 to 10 (inclusive starting from 0), has text 451, has token type 4 (INT), is on line 1 (from 1), and is at character position 8 (starting from zero and counting tabs as a single character). Notice that there are no tokens for the space and newline characters. Rule WS in our grammar tosses them out because of the -> skip directive.

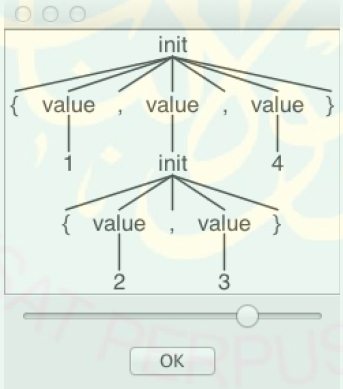
To learn more about how the parser recognized the input, we can ask for the parse tree with the -tree option.

```
⇛ $ grun ArrayInit init -tree
⇛ {99, 3, 451}
⇛ EOF
❮ (init { (value 99) , (value 3) , (value 451) })
```

Option -tree prints out the parse tree in LISP-like text form (*root children*). Or, we can use the -gui option to visualize the tree in a dialog box. Try it with a nested group of integers as input: {1,{2,3},4}.

```
⇛ $ grun ArrayInit init -gui
⇛ {1,{2,3},4}
⇛ EOF
```

Here's the parse tree dialog box that pops up:



2. The end-of-file character is Ctrl+D on Unix and Ctrl+Z on Windows.

In English, the parse tree says, "The input is an initializer with three values surrounded by curly braces. The first and third values are the integers 1 and 4. The second value is itself an initializer with two values surrounded by curly braces. Those values are integers 2 and 3."

Those interior nodes, init and value, are really handy because they identify all of the various input elements by name. It's kind of like identifying the verb and subject in an English sentence. The best part is that ANTLR creates that tree automatically for us based upon the rule names in our grammar. We'll build a translator based on this grammar at the end of this chapter using a built-in tree walker to trigger callbacks like enterInit() and enterValue().

Now that we can run ANTLR on a grammar and test it, it's time to think about how to call this parser from a Java application.

3.3 Integrating a Generated Parser into a Java Program

Once we have a good start on a grammar, we can integrate the ANTLR-generated code into a larger application. In this section, we'll look at a simple Java main() that invokes our initializer parser and prints out the parse tree like TestRig's -tree option. Here's a boilerplate Test.java file that embodies the overall recognizer data flow we saw in Section 2.1, *Let's Get Meta!*, on page 9:

starter/Test.java

```
// import ANTLR's runtime libraries
import org.antlr.v4.runtime.*;
import org.antlr.v4.runtime.tree.*;

public class Test {
    public static void main(String[] args) throws Exception {
        // create a CharStream that reads from standard input
        ANTLRInputStream input = new ANTLRInputStream(System.in);

        // create a lexer that feeds off of input CharStream
        ArrayInitLexer lexer = new ArrayInitLexer(input);

        // create a buffer of tokens pulled from the lexer
        CommonTokenStream tokens = new CommonTokenStream(lexer);

        // create a parser that feeds off the tokens buffer
        ArrayInitParser parser = new ArrayInitParser(tokens);

        ParseTree tree = parser.init(); // begin parsing at init rule
        System.out.println(tree.toStringTree(parser)); // print LISP-style tree
    }
}
```

The program uses a number of classes like CommonTokenStream and ParseTree from ANTLR's runtime library that we'll learn more about starting in Section 4.1, *Matching an Arithmetic Expression Language*, on page 32.

Here's how to compile everything and run Test:

```
⇛ $ javac ArrayInit*.java Test.java
⇛ $ java Test
⇛ {1,{2,3},4}
⇛ EOF
❮ (init { (value 1) , (value (init { (value 2) , (value 3) })) , (value 4) })
```

ANTLR parsers also automatically report and recover from syntax errors. For example, here's what happens if we enter an initializer that's missing the final curly brace:

```
⇛ $ java Test
⇛ {1,2
⇛ EOF
❮ line 2:0 missing '}' at '<EOF>'
  (init { (value 1) , (value 2) <missing '}'>)
```

At this point, we've seen how to run ANTLR on a grammar and integrate the generated parser into a trivial Java application. An application that merely checks syntax is not that impressive, though, so let's finish up by building a translator that converts short array initializers to String objects.

3.4 Building a Language Application

Continuing with our array initializer example, our next goal is to translate not just recognize initializers. For example, let's translate Java short arrays like {99,3,451} to "\u0063\u0003\u01c3" where 63 is the hexadecimal representation of the 99 decimal.

To move beyond recognition, an application has to extract data from the parse tree. The easiest way to do that is to have ANTLR's built-in parse-tree walker trigger a bunch of callbacks as it performs a depth-first walk. As we saw earlier, ANTLR automatically generates a listener infrastructure for us. These listeners are like the callbacks on GUI widgets (for example, a button would notify us upon a button press) or like SAX events in an XML parser.

To write a program that reacts to the input, all we have to do is implement a few methods in a subclass of ArrayInitBaseListener. The basic strategy is to have each listener method print out a translated piece of the input when called to do so by the tree walker.

The beauty of the listener mechanism is that we don't have to do any tree walking ourselves. In fact, we don't even have to know that the runtime is walking a tree to call our methods. All we know is that our listener gets notified at the beginning and end of phrases associated with rules in the grammar. As we'll see in Section 7.2, *Implementing Applications with Parse-Tree Listeners*, on page 112, this approach reduces how much we have to learn about ANTLR—we're back in familiar programming language territory for anything but phrase recognition.

Starting a translation project means figuring out how to convert each input token or phrase to an output string. To do that, it's a good idea to manually translate a few representative samples in order to pick out the general phrase-to-phrase conversions. In this case, the translation is pretty straightforward.

In English, the translation is a series of "*X goes to Y*" rules.

1. Translate { to ".
2. Translate } to ".
3. Translate integers to four-digit hexadecimal strings prefixed with \u.

To code the translator, we need to write methods that print out the converted strings upon seeing the appropriate input token or phrase. The built-in tree walker triggers callbacks in a listener upon seeing the beginning and end of the various phrases. Here's a listener implementation for our translation rules:

starter/ShortToUnicodeString.java

```
/** Convert short array inits like {1,2,3} to "\u0001\u0002\u0003" */
public class ShortToUnicodeString extends ArrayInitBaseListener {
    /** Translate { to " */
    @Override
    public void enterInit(ArrayInitParser.InitContext ctx) {
        System.out.print('"');
    }

    /** Translate } to " */
    @Override
    public void exitInit(ArrayInitParser.InitContext ctx) {
        System.out.print('"');
    }
```

```
    /** Translate integers to 4-digit hexadecimal strings prefixed with \\u */
    @Override
    public void enterValue(ArrayInitParser.ValueContext ctx) {
        // Assumes no nested array initializers
        int value = Integer.valueOf(ctx.INT().getText());
        System.out.printf("\\u%04x", value);
    }
}
```

We don't need to override every enter/exit method; we do just the ones we care about. The only unfamiliar expression is ctx.INT(), which asks the context object for the integer INT token matched by that invocation of rule value. Context objects record everything that happens during the recognition of a rule.

The only thing left to do is to create a translator application derived from the Test boilerplate code shown earlier.

starter/Translate.java

```
// import ANTLR's runtime libraries
import org.antlr.v4.runtime.*;
import org.antlr.v4.runtime.tree.*;

public class Translate {
    public static void main(String[] args) throws Exception {
        // create a CharStream that reads from standard input
        ANTLRInputStream input = new ANTLRInputStream(System.in);
        // create a lexer that feeds off of input CharStream
        ArrayInitLexer lexer = new ArrayInitLexer(input);
        // create a buffer of tokens pulled from the lexer
        CommonTokenStream tokens = new CommonTokenStream(lexer);
        // create a parser that feeds off the tokens buffer
        ArrayInitParser parser = new ArrayInitParser(tokens);
        ParseTree tree = parser.init(); // begin parsing at init rule

        // Create a generic parse tree walker that can trigger callbacks
        ParseTreeWalker walker = new ParseTreeWalker();
        // Walk the tree created during the parse, trigger callbacks
        walker.walk(new ShortToUnicodeString(), tree);
        System.out.println(); // print a \n after translation
    }
}
```

The only difference from the boilerplate code is the highlighted section that creates a tree walker and asks it to walk the tree returned from the parser. As the tree walker traverses, it triggers calls into our ShortToUnicodeString listener.

Please note: To focus our attention and to reduce bloat, the remainder of the book will typically show just the important or novel bits of code rather than entire files. If you're reading the electronic version of this book, you can always

click the code snippet titles; the title bars are links to the full source code on the Web. You can also grab the full source code bundle on the book's website.[3]

Let's build the translator and try it on our sample input.

```
⇛ $ javac ArrayInit*.java Translate.java
⇛ $ java Translate
⇛ {99, 3, 451}
⇛ EOF
❮ "\u0063\u0003\u01c3"
```

It works! We've just built our first translator, without even touching the grammar. All we had to do was implement a few methods that printed the appropriate phrase translations. Moreover, we can generate completely different output simply by passing in a different listener. Listeners effectively isolate the language application from the grammar, making the grammar reusable for other applications.

In the next chapter, we'll take a whirlwind tour of ANTLR grammar notation and the key features that make ANTLR powerful and easy to use.

3. http://pragprog.com/titles/tpantlr2/source_code

CHAPTER 4

A Quick Tour

So far, we have learned how to install ANTLR and have looked at the key processes, terminology, and building blocks needed to build a language application. In this chapter, we're going to take a whirlwind tour of ANTLR by racing through a number of examples that illustrate its capabilities. We'll be glossing over a lot of details in the interest of brevity, so don't worry if things aren't crystal clear. The goal is just to get a feel for what you can do with ANTLR. We'll look underneath the hood starting in Chapter 5, *Designing Grammars*, on page 57. For those with experience using previous versions of ANTLR, this chapter is a great way to retool.

This chapter is broken down into four broad topics that nicely illustrate the feature set. It's a good idea to download the code[1] for this book (or follow the links in the ebook version) and work through the examples as we go along. That way, you'll get used to working with grammar files and building ANTLR applications. Keep in mind that many of the code snippets you see interspersed in the text aren't complete files so that we can focus on the interesting bits.

First, we're going to work with a grammar for a simple arithmetic expression language. We'll test it initially using ANTLR's built-in test rig and then learn more about the boilerplate main program that launches parsers shown in Section 3.3, *Integrating a Generated Parser into a Java Program*, on page 26. Then, we'll look at a nontrivial parse tree for the expression grammar. (Recall that a parse tree records how a parser matches an input phrase.) For dealing with very large grammars, we'll see how to split a grammar into manageable chunks using grammar imports. Next, we'll check out how ANTLR-generated parsers respond to invalid input.

1. http://pragprog.com/titles/tpantlr2/source_code

Second, after looking at the parser for arithmetic expressions, we'll use a visitor pattern to build a calculator that walks expression grammar parse trees. ANTLR parsers automatically generate visitor interfaces and blank method implementations so we can get started painlessly.

Third, we'll build a translator that reads in a Java class definition and spits out a Java interface derived from the methods in that class. Our implementation will use the tree listener mechanism that ANTLR also generates automatically.

Fourth, we'll learn how to embed *actions* (arbitrary code) directly in the grammar. Most of the time, we can build language applications with visitors or listeners, but for the ultimate flexibility, ANTLR allows us to inject our own application-specific code into the generated parser. These actions execute during the parse and can collect information or generate output like any other arbitrary code snippets. In conjunction with *semantic predicates* (Boolean expressions), we can even make parts of our grammar disappear at runtime! For example, we might want to turn the `enum` keyword on and off in a Java grammar to parse different versions of the language. Without semantic predicates, we'd need two different versions of the grammar.

Finally, we'll zoom in on a few ANTLR features at the lexical (token) level. We'll see how ANTLR deals with input files that contain more than one language. Then we'll look at the awesome `TokenStreamRewriter` class that lets us tweak, mangle, or otherwise manipulate token streams, all without disturbing the original input stream. Finally, we'll revisit our interface generator example to learn how ANTLR can ignore whitespace and comments during Java parsing but retain them for later processing.

Let's begin our tour by getting acquainted with ANTLR grammar notation. Make sure you have the `antlr4` and `grun` aliases or scripts defined, as explained in Section 1.2, *Executing ANTLR and Testing Recognizers*, on page 6.

4.1 Matching an Arithmetic Expression Language

For our first grammar, we're going to build a simple calculator. Doing something with expressions makes sense because they're so common. To keep things simple, we'll allow only the basic arithmetic operators (add, subtract, multiply, and divide), parenthesized expressions, integer numbers, and variables. We'll also restrict ourselves to integers instead of allowing floating-point numbers.

Here's some sample input that illustrates all language features:

tour/t.expr
```
193
a = 5
b = 6
a+b*2
(1+2)*3
```

In English, a *program* in our expression language is a sequence of statements terminated by newlines. A statement is either an expression, an assignment, or a blank line. Here's an ANTLR grammar that'll parse those statements and expressions for us:

tour/Expr.g4
```
grammar Expr;

/** The start rule; begin parsing here. */
prog:   stat+ ;

stat:   expr NEWLINE
    |   ID '=' expr NEWLINE
    |   NEWLINE
    ;

expr:   expr ('*'|'/') expr
    |   expr ('+'|'-') expr
    |   INT
    |   ID
    |   '(' expr ')'
    ;

ID  :   [a-zA-Z]+ ;      // match identifiers
INT :   [0-9]+ ;         // match integers
NEWLINE:'\r'? '\n' ;     // return newlines to parser (is end-statement signal)
WS  :   [ \t]+ -> skip ; // toss out whitespace
```

Without going into too much detail, let's look at some of the key elements of ANTLR's grammar notation.

- Grammars consist of a set of rules that describe language syntax. There are rules for syntactic structure like stat and expr as well as rules for vocabulary symbols (tokens) such as identifiers and integers.
- Rules starting with a lowercase letter comprise the parser rules.
- Rules starting with an uppercase letter comprise the lexical (token) rules.
- We separate the alternatives of a rule with the | operator, and we can group symbols with parentheses into *subrules*. For example, subrule ('*'|'/') matches either a multiplication symbol or a division symbol.

We'll tackle all of this stuff in detail when we get to Chapter 5, *Designing Grammars*, on page 57.

One of ANTLR v4's most significant new features is its ability to handle (most kinds of) left-recursive rules. A left-recursive rule is one that invokes itself at the start of an alternative. For example, in this grammar, rule `expr` has alternatives on lines 11 and 12 that recursively invoke `expr` on the left edge. Specifying arithmetic expression notation this way is dramatically easier than what we'd need for the typical top-down parser strategy. In that strategy, we'd need multiple rules, one for each operator precedence level. For more on this feature, see Section 5.4, *Dealing with Precedence, Left Recursion, and Associativity*, on page 69.

The notation for the token definitions should be familiar to those with regular expression experience. We'll look at lots of lexical (token) rules in Chapter 6, *Exploring Some Real Grammars*, on page 83. The only unusual syntax is the `-> skip` operation on the `WS` whitespace rule. It's a directive that tells the lexer to match but throw out whitespace. (Every possible input character must be matched by at least one lexical rule.) We avoid tying the grammar to a specific target language by using formal ANTLR notation instead of an arbitrary code snippet in the grammar that tells the lexer to skip.

OK, let's take grammar `Expr` out for a joy ride. Download it either by clicking the `tour/Expr.g4` link on the previous code listing, if you're viewing the ebook version, or by cutting and pasting the grammar into a file called `Expr.g4`.

The easiest way to test grammars is with the built-in `TestRig`, which we can access using alias `grun`. For example, here is the build and test sequence on a Unix box:

```
$ antlr4 Expr.g4
$ ls Expr*.java
ExprBaseListener.java   ExprListener.java
ExprLexer.java          ExprParser.java
$ javac Expr*.java
$ grun Expr prog -gui t.expr # launches org.antlr.v4.runtime.misc.TestRig
```

Because of the `-gui` option, the test rig pops up a window showing the parse tree, as shown in Figure 2, *Window showing the parse tree*, on page 35.

The parse tree is analogous to the function call tree our parser would trace as it recognizes input. (ANTLR generates a function for each rule.)

It's OK to develop and test grammars using the test rig, but ultimately we'll need to integrate our ANTLR-generated parser into an application. The main

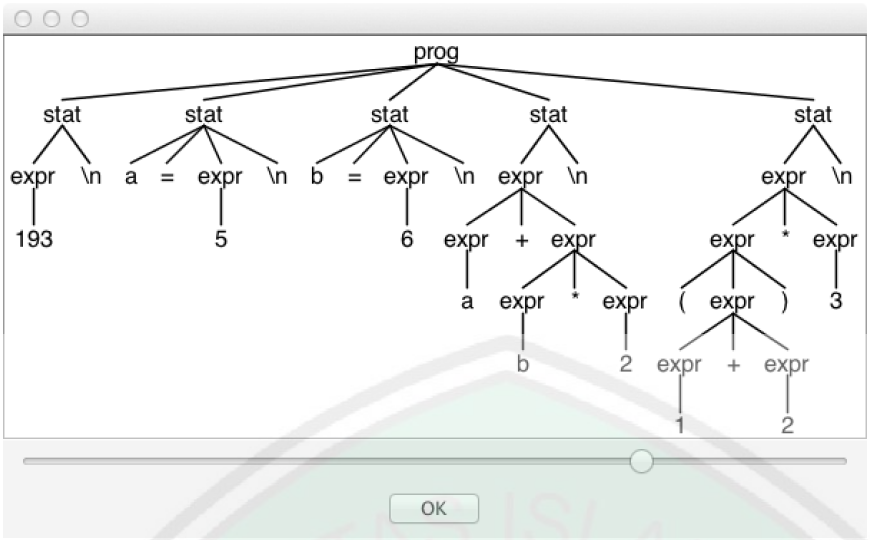


Figure 2—Window showing the parse tree

program given below shows the code necessary to create all necessary objects and launch our expression language parser starting at rule `prog`:

tour/ExprJoyRide.java

```
Line 1 import org.antlr.v4.runtime.*;
  - import org.antlr.v4.runtime.tree.*;
  - import java.io.FileInputStream;
  - import java.io.InputStream;
  5 public class ExprJoyRide {
  -     public static void main(String[] args) throws Exception {
  -         String inputFile = null;
  -         if ( args.length>0 ) inputFile = args[0];
  -         InputStream is = System.in;
 10         if ( inputFile!=null ) is = new FileInputStream(inputFile);
  -         ANTLRInputStream input = new ANTLRInputStream(is);
  -         ExprLexer lexer = new ExprLexer(input);
  -         CommonTokenStream tokens = new CommonTokenStream(lexer);
  -         ExprParser parser = new ExprParser(tokens);
 15         ParseTree tree = parser.prog(); // parse; start at prog
  -         System.out.println(tree.toStringTree(parser)); // print tree as text
  -     }
  - }
```

Lines 7..11 create an input stream of characters for the lexer. Lines 12..14 create the lexer and parser objects and a token stream "pipe" between them. Line 15 actually launches the parser. (Calling a rule method is like invoking

that rule; we can call any parser rule method we want.) Finally, line 16 prints out the parse tree returned from the rule method prog() in text form.

Here is how to build the test program and run it on input file t.expr:

```
⇛ $ javac ExprJoyRide.java Expr*.java
⇛ $ java ExprJoyRide t.expr
❮ (prog
    (stat (expr 193) \n)
    (stat a = (expr 5) \n)
    (stat b = (expr 6) \n)
    (stat (expr (expr a) + (expr (expr b) * (expr 2))) \n)
    (stat (expr (expr ( (expr (expr 1) + (expr 2)) )) * (expr 3)) \n)
)
```

The (slightly cleaned up) text representation of the parse tree is not as easy to read as the visual representation, but it's useful for functional testing.

This expression grammar is pretty small, but grammars can run into the thousands of lines. In the next section, we'll learn how to keep such large grammars manageable.

Importing Grammars

It's a good idea to break up very large grammars into logical chunks, just like we do with software. One way to do that is to split a grammar into parser and lexer grammars. That's not a bad idea because there's a surprising amount of overlap between different languages lexically. For example, identifiers and numbers are usually the same across languages. Factoring out lexical rules into a "module" means we can use it for different parser grammars. Here's a lexer grammar containing all of the lexical rules:

tour/CommonLexerRules.g4

```
lexer grammar CommonLexerRules; // note "lexer grammar"

ID  :   [a-zA-Z]+ ;      // match identifiers
INT :   [0-9]+ ;         // match integers
NEWLINE:'\r'? '\n' ;     // return newlines to parser (end-statement signal)
WS  :   [ \t]+ -> skip ; // toss out whitespace
```

Now we can replace the lexical rules from the original grammar with an import statement.

tour/LibExpr.g4

```
grammar LibExpr;         // Rename to distinguish from original
import CommonLexerRules; // includes all rules from CommonLexerRules.g4
/** The start rule; begin parsing here. */
prog:   stat+ ;
```

```
stat:   expr NEWLINE
    |   ID '=' expr NEWLINE
    |   NEWLINE
    ;

expr:   expr ('*'|'/') expr
    |   expr ('+'|'-') expr
    |   INT
    |   ID
    |   '(' expr ')'
    ;
```

The build and test sequence is the same as it was without the import. We do not run ANTLR on the imported grammar itself.

```
⇒ $ antlr4 LibExpr.g4 # automatically pulls in CommonLexerRules.g4
⇒ $ ls Lib*.java
❮ LibExprBaseListener.java        LibExprListener.java
  LibExprLexer.java               LibExprParser.java
⇒ $ javac LibExpr*.java
⇒ $ grun LibExpr prog -tree
⇒ 3+4
⇒ EOF
❮ (prog (stat (expr (expr 3) + (expr 4)) \n))
```

So far, we've assumed valid input, but error handling is an important part of almost all language applications. Let's see what ANTLR does with erroneous input.

Handling Erroneous Input

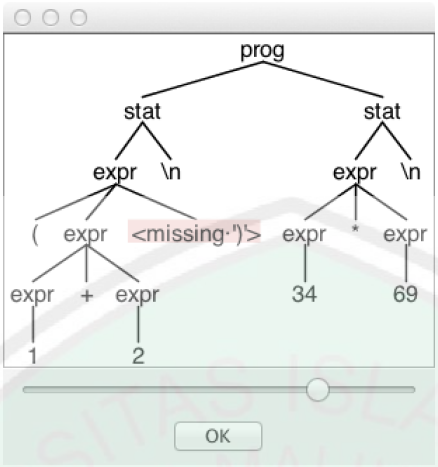
ANTLR parsers automatically report and recover from syntax errors. For example, if we forget a closing parenthesis in an expression, the parser automatically emits an error message.

```
⇒ $ java ExprJoyRide
⇒ (1+2
⇒ 3
⇒ EOF
❮ line 1:4 mismatched input '\n' expecting {')', '+', '*', '-', '/'}
  (prog
    (stat (expr ( (expr (expr 1) + (expr 2)) <missing ')'>) \n)
    (stat (expr 3) \n)
  )
```

Equally important is that the parser recovers to correctly match the second expression (the 3).

When using the -gui option on grun, the parse-tree dialog automatically highlights error nodes in red.

```
⇒ $ grun LibExpr prog -gui
⇒ (1+2
⇒ 34*69
⇒ EOF
```



Notice that ANTLR successively recovered from the error in the first expression again to properly match the second.

ANTLR's error mechanism has lots of flexibility. We can alter the error messages, catch recognition exceptions, and even alter the fundamental error handling strategy. We'll cover this in Chapter 9, *Error Reporting and Recovery*, on page 149.

That completes our quick tour of grammars and parsing. We've looked at a simple expression grammar and how to launch it using the built-in test rig and a sample main program. We also saw how to get text and visual representations of parse trees that show how our grammar recognizes input phrases. The import statement lets us break up grammars into *modules*. Now, let's move beyond language recognition to interpreting expressions (computing their values).

4.2 Building a Calculator Using a Visitor

To get the previous arithmetic expression parser to compute values, we need to write some Java code. ANTLR v4 encourages us to keep grammars clean and use parse-tree visitors and other walkers to implement language applications. In this section, we'll use the well-known visitor pattern to implement our little calculator. To make things easier for us, ANTLR automatically generates a visitor interface and blank visitor implementation object.

Before we get to the visitor, we need to make a few modifications to the grammar. First, we need to label the alternatives of the rules. (The labels can be any identifier that doesn't collide with a rule name.) Without labels on the alternatives, ANTLR generates only one visitor method per rule. (Chapter 7, *Decoupling Grammars from Application-Specific Code*, on page 109 uses a similar grammar to explain the visitor mechanism in more detail.) In our case, we'd like a different visitor method for each alternative so that we can get different "events" for each kind of input phrase. Labels appear on the right edge of alternatives and start with the # symbol in our new grammar, LabeledExpr.

tour/LabeledExpr.g4

```
stat:   expr NEWLINE                # printExpr
    |   ID '=' expr NEWLINE         # assign
    |   NEWLINE                     # blank
    ;

expr:   expr op=('*'|'/') expr      # MulDiv
    |   expr op=('+'|'-') expr      # AddSub
    |   INT                         # int
    |   ID                          # id
    |   '(' expr ')'                # parens
    ;
```

Next, let's define some token names for the operator literals so that, later, we can reference token names as Java constants in the visitor.

tour/LabeledExpr.g4

```
MUL :   '*' ; // assigns token name to '*' used above in grammar
DIV :   '/' ;
ADD :   '+' ;
SUB :   '-' ;
```

Now that we have a properly enhanced grammar, let's start coding our calculator and see what the main program looks like. Our main program in file Calc.java is nearly identical to the main() in ExprJoyRide.java from earlier. The first difference is that we create lexer and parser objects derived from grammar LabeledExpr, not Expr.

tour/Calc.java

```
LabeledExprLexer lexer = new LabeledExprLexer(input);
CommonTokenStream tokens = new CommonTokenStream(lexer);
LabeledExprParser parser = new LabeledExprParser(tokens);
ParseTree tree = parser.prog(); // parse
```

We also can remove the print statement that displays the tree as text. The other difference is that we create an instance of our visitor class, EvalVisitor, which we'll get to in just a second. To start walking the parse tree returned from method prog(), we call visit().

tour/Calc.java
```
EvalVisitor eval = new EvalVisitor();
eval.visit(tree);
```

All of our supporting machinery is now in place. The only thing left to do is implement a visitor that computes and returns values by walking the parse tree. To get started, let's see what ANTLR generates for us when we type.

```
⇒ $ antlr4 -no-listener -visitor LabeledExpr.g4
```

First, ANTLR generates a visitor interface with a method for each labeled alternative name.

```
public interface LabeledExprVisitor<T> {
    T visitId(LabeledExprParser.IdContext ctx);          # from label id
    T visitAssign(LabeledExprParser.AssignContext ctx);  # from label assign
    T visitMulDiv(LabeledExprParser.MulDivContext ctx);  # from label MulDiv
    ...
}
```

The interface definition uses Java generics with a parameterized type for the return values of the visit methods. This allows us to derive implementation classes with our choice of return value type to suit the computations we want to implement.

Next, ANTLR generates a default visitor implementation called LabeledExprBaseVisitor that we can subclass. In this case, our expression results are integers and so our EvalVisitor should extend LabeledExprBaseVisitor<Integer>. To implement the calculator, we override the methods associated with statement and expression alternatives. Here it is in its full glory. You can either cut and paste or save the tour/EvalVisitor link (ebook version).

tour/EvalVisitor.java
```
import java.util.HashMap;
import java.util.Map;

public class EvalVisitor extends LabeledExprBaseVisitor<Integer> {
    /** "memory" for our calculator; variable/value pairs go here */
    Map<String, Integer> memory = new HashMap<String, Integer>();

    /** ID '=' expr NEWLINE */
    @Override
    public Integer visitAssign(LabeledExprParser.AssignContext ctx) {
        String id = ctx.ID().getText();  // id is left-hand side of '='
        int value = visit(ctx.expr());   // compute value of expression on right
        memory.put(id, value);           // store it in our memory
        return value;
    }
```

```
    /** expr NEWLINE */
    @Override
    public Integer visitPrintExpr(LabeledExprParser.PrintExprContext ctx) {
        Integer value = visit(ctx.expr()); // evaluate the expr child
        System.out.println(value);         // print the result
        return 0;                          // return dummy value
    }

    /** INT */
    @Override
    public Integer visitInt(LabeledExprParser.IntContext ctx) {
        return Integer.valueOf(ctx.INT().getText());
    }

    /** ID */
    @Override
    public Integer visitId(LabeledExprParser.IdContext ctx) {
        String id = ctx.ID().getText();
        if ( memory.containsKey(id) ) return memory.get(id);
        return 0;
    }

    /** expr op=('*'|'/') expr */
    @Override
    public Integer visitMulDiv(LabeledExprParser.MulDivContext ctx) {
        int left = visit(ctx.expr(0));  // get value of left subexpression
        int right = visit(ctx.expr(1)); // get value of right subexpression
        if ( ctx.op.getType() == LabeledExprParser.MUL ) return left * right;
        return left / right; // must be DIV
    }

    /** expr op=('+'|'-') expr */
    @Override
    public Integer visitAddSub(LabeledExprParser.AddSubContext ctx) {
        int left = visit(ctx.expr(0));  // get value of left subexpression
        int right = visit(ctx.expr(1)); // get value of right subexpression
        if ( ctx.op.getType() == LabeledExprParser.ADD ) return left + right;
        return left - right; // must be SUB
    }

    /** '(' expr ')' */
    @Override
    public Integer visitParens(LabeledExprParser.ParensContext ctx) {
        return visit(ctx.expr()); // return child expr's value
    }
}
```

And here is the build and test sequence that evaluates expressions in t.expr:

```
⇒ $ antlr4 -no-listener -visitor LabeledExpr.g4   # -visitor is required!!!
⇒ $ ls LabeledExpr*.java
❮ LabeledExprBaseVisitor.java     LabeledExprParser.java
  LabeledExprLexer.java           LabeledExprVisitor.java
⇒ $ javac Calc.java LabeledExpr*.java
⇒ $ cat t.expr
❮ 193
  a = 5
  b = 6
  a+b*2
  (1+2)*3
⇒ $ java Calc t.expr
❮ 193
  17
  9
```

The takeaway is that we built a calculator without having to insert raw Java actions into the grammar, as we would need to do in ANTLR v3. The grammar is kept application independent and programming language neutral. The visitor mechanism also keeps everything beyond the recognition-related stuff in familiar Java territory. There's no extra ANTLR notation to learn in order to build a language application on top of a generated parser.

Before moving on, you might take a moment to try to extend this expression language by adding a `clear` statement. It's a great way to get your feet wet and do something real without having to know all of the details. The `clear` command should clear out the `memory` map, and you'll need a new alternative in rule `stat` to recognize it. Label the alternative with `# clear` and then run ANTLR on the grammar to get the augmented visitor interface. Then, to make something happen upon `clear`, implement visitor method `visitClear()`. Compile and run `Calc` following the earlier sequence.

Let's switch gears now and think about translation rather than evaluating or interpreting input. In the next section, we're going to use a variation of the visitor called a *listener* to build a translator for Java source code.

4.3 Building a Translator with a Listener

Imagine your boss assigns you to build a tool that generates a Java interface file from the methods in a Java class definition. Panic ensues if you're a junior programmer. As an experienced Java developer, you might suggest using the Java reflection API or the `javap` tool to extract method signatures. If your Java tool building kung fu is very strong, you might even try using a bytecode library such as ASM.[2] Then your boss says, "Oh, yeah. Preserve whitespace

2. http://asm.ow2.org

and comments within the bounds of the method signature." There's no way around it now. We have to parse Java source code. For example, we'd like to read in Java code like this:

tour/Demo.java
```
import java.util.List;
import java.util.Map;
public class Demo {
        void f(int x, String y) { }
        int[ ] g(/*no args*/) { return null; }
        List<Map<String, Integer>>[] h() { return null; }
}
```

and generate an interface with the method signatures, preserving the whitespace and comments.

tour/IDemo.java
```
interface IDemo {
        void f(int x, String y);
        int[ ] g(/*no args*/);
        List<Map<String, Integer>>[] h();
}
```

Believe it or not, we're going to solve the core of this problem in about fifteen lines of code by listening to "events" fired from a Java parse-tree walker. The Java parse tree will come from a parser generated from an existing Java grammar included in the source code for this book. We'll derive the name of the generated interface from the class name and grab method signatures (return type, method name, and argument list) from method definitions. For a similar but more thoroughly explained example, see Section 8.3, *Generating a Call Graph*, on page 134.

The key "interface" between the grammar and our listener object is called JavaListener, and ANTLR automatically generates it for us. It defines all of the methods that class ParseTreeWalker from ANTLR's runtime can trigger as it traverses the parse tree. In our case, we need to respond to three events by overriding three methods: when the walker enters and exits a class definition and when it encounters a method definition. Here are the relevant methods from the generated listener interface:

```
public interface JavaListener extends ParseTreeListener {
    void enterClassDeclaration(JavaParser.ClassDeclarationContext ctx);
    void exitClassDeclaration(JavaParser.ClassDeclarationContext ctx);
    void enterMethodDeclaration(JavaParser.MethodDeclarationContext ctx);
    ...
}
```

The biggest difference between the listener and visitor mechanisms is that listener methods are called by the ANTLR-provided walker object, whereas visitor methods must walk their children with explicit visit calls. Forgetting to invoke visit() on a node's children means those subtrees don't get visited.

To build our listener implementation, we need to know what rules classDeclaration and methodDeclaration look like because listener methods have to grab phrase elements matched by the rules. File Java.g4 is a complete grammar for Java, but here are the two methods we need to look at for this problem:

tour/Java.g4
```
classDeclaration
    :   'class' Identifier typeParameters? ('extends' type)?
        ('implements' typeList)?
        classBody
    ;
```

tour/Java.g4
```
methodDeclaration
    :   type Identifier formalParameters ('[' ']')* methodDeclarationRest
    |   'void' Identifier formalParameters methodDeclarationRest
    ;
```

So that we don't have to implement all 200 or so interface methods, ANTLR generates a default implementation called JavaBaseListener. Our interface extractor can then subclass JavaBaseListener and override the methods of interest.

Our basic strategy will be to print out the interface header when we see the start of a class definition. Then, we'll print a terminating } at the end of the class definition. Upon each method definition, we'll spit out its signature. Here's the complete implementation:

tour/ExtractInterfaceListener.java
```
import org.antlr.v4.runtime.TokenStream;
import org.antlr.v4.runtime.misc.Interval;

public class ExtractInterfaceListener extends JavaBaseListener {
    JavaParser parser;
    public ExtractInterfaceListener(JavaParser parser) {this.parser = parser;}
    /** Listen to matches of classDeclaration */
    @Override
    public void enterClassDeclaration(JavaParser.ClassDeclarationContext ctx){
        System.out.println("interface I"+ctx.Identifier()+" {");
    }
    @Override
    public void exitClassDeclaration(JavaParser.ClassDeclarationContext ctx) {
        System.out.println("}");
    }
```

```
    /** Listen to matches of methodDeclaration */
    @Override
    public void enterMethodDeclaration(
        JavaParser.MethodDeclarationContext ctx
    )
    {
        // need parser to get tokens
        TokenStream tokens = parser.getTokenStream();
        String type = "void";
        if ( ctx.type()!=null ) {
            type = tokens.getText(ctx.type());
        }
        String args = tokens.getText(ctx.formalParameters());
        System.out.println("\t"+type+" "+ctx.Identifier()+args+";");
    }
}
```

To fire this up, we need a main program, which looks almost the same as the others in this chapter. Our application code starts after we've launched the parser.

tour/ExtractInterfaceTool.java

```
JavaLexer lexer = new JavaLexer(input);
CommonTokenStream tokens = new CommonTokenStream(lexer);
JavaParser parser = new JavaParser(tokens);
ParseTree tree = parser.compilationUnit(); // parse

ParseTreeWalker walker = new ParseTreeWalker(); // create standard walker
ExtractInterfaceListener extractor = new ExtractInterfaceListener(parser);
walker.walk(extractor, tree); // initiate walk of tree with listener
```

We also need to add `import org.antlr.v4.runtime.tree.*;` at the top of the file.

Given grammar Java.g4 and our `main()` in `ExtractInterfaceTool`, here's the complete build and test sequence:

```
⇒ $ antlr4 Java.g4
⇒ $ ls Java*.java ExtractInterface*.java
❮ ExtractInterfaceListener.java  JavaBaseListener.java   JavaListener.java
  ExtractInterfaceTool.java      JavaLexer.java          JavaParser.java
⇒ $ javac Java*.java Extract*.java
⇒ $ java ExtractInterfaceTool Demo.java
❮ interface IDemo {
        void f(int x, String y);
        int[ ] g(/*no args*/);
        List<Map<String, Integer>>[] h();
}
```

This implementation isn't quite complete because it doesn't include in the interface file the import statements for the types referenced by the interface

methods such as List. As an exercise, try handling the imports. It should convince you that it's easy to build these kinds of extractors or translators using a listener. We don't even need to know what the importDeclaration rule looks like because enterImportDeclaration() should simply print the text matched by the entire rule: parser.getTokenStream().getText(ctx).

The visitor and listener mechanisms work very well and promote the separation of concerns between parsing and parser application. Sometimes, though, we need extra control and flexibility.

4.4 Making Things Happen During the Parse

Listeners and visitors are great because they keep application-specific code out of grammars, making grammars easier to read and preventing them from getting entangled with a particular application. For the ultimate flexibility and control, however, we can directly embed code snippets (actions) within grammars. These actions are copied into the recursive-descent parser code ANTLR generates. In this section, we'll implement a simple program that reads in rows of data and prints out the values found in a specific column. After that, we'll see how to make special actions, called *semantic predicates*, dynamically turn parts of a grammar on and off.

Embedding Arbitrary Actions in a Grammar

We can compute values or print things out on-the-fly during parsing if we don't want the overhead of building a parse tree. On the other hand, it means embedding arbitrary code within the expression grammar, which is harder; we have to understand the effect of the actions on the parser and where to position those actions.

To demonstrate actions embedded in a grammar, let's build a program that prints out a specific column from rows of data. This comes up all the time for me because people send me text files from which I need to grab, say, the name or email column. For our purposes, let's use the following data:

tour/t.rows

```
parrt   Terence Parr    101
tombu   Tom Burns       020
bke     Kevin Edgar     008
```

The columns are tab-delimited, and each row ends with a newline character. Matching this kind of input is pretty simple grammatically.

```
file : (row NL)+ ; // NL is newline token: '\r'? '\n'
row  : STUFF+ ;
```

It gets mucked up, though, when we add actions. We need to create a constructor so that we can pass in the column number we want (counting from 1), and we need an action inside the (...)+ loop in rule row.

tour/Rows.g4
```
grammar Rows;

@parser::members { // add members to generated RowsParser
    int col;
    public RowsParser(TokenStream input, int col) { // custom constructor
        this(input);
        this.col = col;
    }
}

file: (row NL)+ ;

row
locals [int i=0]
    : (   STUFF
          {
          $i++;
          if ( $i == col ) System.out.println($STUFF.text);
          }
      )+
    ;

TAB   :  '\t' -> skip ;   // match but don't pass to the parser
NL    :  '\r'? '\n' ;     // match and pass to the parser
STUFF:  ~[\t\r\n]+ ;      // match any chars except tab, newline
```

The STUFF lexical rule matches anything that's not a tab or newline, which means we can have space characters in a column.

A suitable main program should be looking pretty familiar by now. The only thing different here is that we're passing in a column number to the parser using a custom constructor and telling the parser not to build a tree.

tour/Col.java
```
RowsLexer lexer = new RowsLexer(input);
CommonTokenStream tokens = new CommonTokenStream(lexer);
int col = Integer.valueOf(args[0]);
RowsParser parser = new RowsParser(tokens, col); // pass column number!
parser.setBuildParseTree(false); // don't waste time bulding a tree
parser.file(); // parse
```

There are a lot of details in there that we'll explore in Chapter 10, *Attributes and Actions*, on page 175. For now, actions are code snippets surrounded by curly braces. The members action injects that code into the member area of

the generated parser class. The action within rule row accesses $i, the local variable defined with the locals clause. It also uses $STUFF.text to get the text for the most recently matched STUFF token.

Here's the build and test sequence, one test per column:

```
⇒ $ antlr4 -no-listener Rows.g4  # don't need the listener
⇒ $ javac Rows*.java Col.java
⇒ $ java Col 1 < t.rows          # print out column 1, reading from file t.rows
❮ parrt
  tombu
  bke
⇒ $ java Col 2 < t.rows
❮ Terence Parr
  Tom Burns
  Kevin Edgar
⇒ $ java Col 3 < t.rows
❮ 101
  020
  008
```

These actions extract and print values matched by the parser, but they don't alter the parse itself. Actions can also finesse how the parser recognizes input phrases. In the next section, we'll take the concept of embedded actions one step further.

Altering the Parse with Semantic Predicates

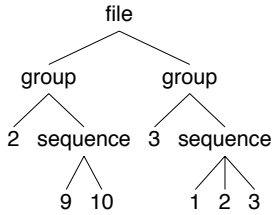
Until we get to Chapter 11, *Altering the Parse with Semantic Predicates*, on page 189, we can demonstrate the power of semantic predicates with a simple example. Let's look at a grammar that reads in sequences of integers. The trick is that part of the input specifies how many integers to group together. We don't know until runtime how many integers to match. Here's a sample input file:

tour/t.data
```
2 9 10 3 1 2 3
```

The first number says to match the two subsequent numbers, 9 and 10. The 3 following the 10 says to match three more as a sequence. Our goal is a grammar called Data that groups 9 and 10 together and then 1, 2, and 3 like this:

```
⇒ $ antlr4 -no-listener Data.g4
⇒ $ javac Data*.java
⇒ $ grun Data file -tree t.data
❮ (file (group 2 (sequence 9 10)) (group 3 (sequence 1 2 3)))
```

The parse tree clearly identifies the groups.

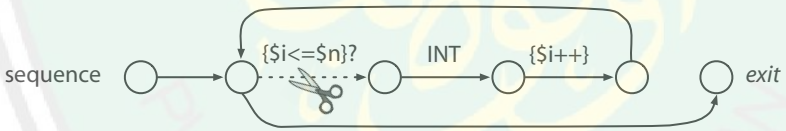


The key in the following Data grammar is a special Boolean-valued action called a *semantic predicate*: {$i<=$n}?. That predicate evaluates to true until we surpass the number of integers requested by the sequence rule parameter n. False predicates make the associated alternative "disappear" from the grammar and, hence, from the generated parser. In this case, a false predicate makes the (...)* loop terminate and return from rule sequence.

tour/Data.g4

```
grammar Data;

file : group+ ;

group: INT sequence[$INT.int] ;

sequence[int n]
locals [int i = 1;]
     : ( {$i<=$n}? INT {$i++;} )* // match n integers
     ;

INT :   [0-9]+ ;              // match integers
WS  :   [ \t\n\r]+ -> skip ; // toss out all whitespace
```

Visually, the internal grammar representation of rule sequence used by the parser looks something like this:



The scissors and dashed line indicate that the predicate can snip that path, leaving the parser with only one choice: the path to the exit.

Most of the time we won't need such micromanagement, but it's nice to know we have a weapon for handling pathological parsing problems.

During our tour so far, we've focused on parsing features, but there is a lot of interesting stuff going on at the lexical level. Let's take a look.

4.5 Cool Lexical Features

ANTLR has three great token-related features that are worth demonstrating in our tour. First, we'll see how to deal with formats like XML that have different lexical structures (inside and outside tags) in the same file. Next, we'll learn how to insert a field into a Java class by tweaking the input stream. It'll show how to generate output that is very similar to the input with minimal effort. And, lastly, we'll see how ANTLR parsers can ignore whitespace and comments without having to throw them out.

Island Grammars: Dealing with Different Formats in the Same File

All the sample input files we've seen so far contain a single language, but there are common file formats that contain multiple languages. For example, the `@author` tags and so on inside Java document comments follow a mini language; everything outside the comment is Java code. Template engines such as StringTemplate[3] and Django[4] have a similar problem. They have to treat all of the text surrounding the template expressions differently. These are often called *island grammars*.

ANTLR provides a well-known lexer feature called *lexical modes* that lets us deal easily with files containing mixed formats. The basic idea is to have the lexer switch back and forth between modes when it sees special sentinel character sequences.

XML is a good example. An XML parser treats everything other than tags and entity references (such as `&pound;`) as text chunks. When the lexer sees <, it switches to "inside" mode and switches back to the default mode when it sees > or />. The following grammar demonstrates how this works. We'll explore this in more detail in Chapter 12, *Wielding Lexical Black Magic*, on page 203.

tour/XMLLexer.g4
```
lexer grammar XMLLexer;

// Default "mode": Everything OUTSIDE of a tag
OPEN        :   '<'                 -> pushMode(INSIDE) ;
COMMENT     :   '<!--' .*? '-->'    -> skip ;
EntityRef   :   '&' [a-z]+ ';' ;
TEXT        :   ~('<'|'&')+ ;           // match any 16 bit char minus < and &

// ----------------- Everything INSIDE of a tag --------------------
mode INSIDE;
```

3. http://www.stringtemplate.org
4. https://www.djangoproject.com

```
CLOSE       :   '>'                 -> popMode ; // back to default mode
SLASH_CLOSE :   '/>'                -> popMode ;
EQUALS      :   '=' ;
STRING      :   '"' .*? '"' ;
SlashName   :   '/' Name ;
Name        :   ALPHA (ALPHA|DIGIT)* ;
S           :   [ \t\r\n]           -> skip ;

fragment
ALPHA       :   [a-zA-Z] ;

fragment
DIGIT       :   [0-9] ;
```

Let's use the following XML file as a sample input to that grammar:

tour/t.xml

```
<tools>
        <tool name="ANTLR">A parser generator</tool>
</tools>
```

Here's how to do a build and launch the test rig:

```
⇛ $ antlr4 XMLLexer.g4
⇛ $ javac XML*.java
⇛ $ grun XML tokens -tokens t.xml
❮ [@0,0:0='<',<1>,1:0]
  [@1,1:5='tools',<10>,1:1]
  [@2,6:6='>',<5>,1:6]
  [@3,7:8='\n\t',<4>,1:7]
  [@4,9:9='<',<1>,2:1]
  [@5,10:13='tool',<10>,2:2]
  [@6,15:18='name',<10>,2:7]
  [@7,19:19='=',<7>,2:11]
  [@8,20:26='"ANTLR"',<8>,2:12]
  [@9,27:27='>',<5>,2:19]
  [@10,28:45='A parser generator',<4>,2:20]
  [@11,46:46='<',<1>,2:38]
  [@12,47:51='/tool',<9>,2:39]
  [@13,52:52='>',<5>,2:44]
  [@14,53:53='\n',<4>,2:45]
  [@15,54:54='<',<1>,3:0]
  [@16,55:60='/tools',<9>,3:1]
  [@17,61:61='>',<5>,3:7]
  [@18,62:62='\n',<4>,3:8]
  [@19,63:62='<EOF>',<-1>,4:9]
```

Each line of that output represents a token and contains the token index, the start and stop character, the token text, the token type, and finally the line and character position within the line. This tells us how the lexer tokenized the input.

On the test rig command line, the XML tokens sequence is normally a grammar name followed by the start rule. In this case, we use the grammar name followed by special rule name tokens to tell the test rig it should run the lexer but not the parser. Then, we use test rig option -tokens to print out the list of matched tokens.

Knowledge of the token stream flowing from the lexer to the parser can be pretty useful. For example, some translation problems are really just tweaks of the input. We can sometimes get away with altering the original token stream rather than generating completely new output.

Rewriting the Input Stream

Let's build a tool that processes Java source code to insert serialization identifiers, serialVersionUID, for use with java.io.Serializable (like Eclipse does automatically). We want to avoid implementing every listener method in a JavaListener interface, generated from a Java grammar by ANTLR, just to capture the text and print it back out. It's easier to insert the appropriate constant field into the original token stream and then print out the altered input stream. No fuss, no muss.

Our main program looks exactly the same as the one in ExtractInterfaceTool.java from Section 4.3, *Building a Translator with a Listener*, on page 42 except that we print the token stream out when the listener has finished (highlighted with an arrow).

tour/InsertSerialID.java

```
ParseTreeWalker walker = new ParseTreeWalker(); // create standard walker
InsertSerialIDListener extractor = new InsertSerialIDListener(tokens);
walker.walk(extractor, tree); // initiate walk of tree with listener

// print back ALTERED stream
System.out.println(extractor.rewriter.getText());
```

To implement the listener, we need to trigger an insertion when we see the start of a class.

tour/InsertSerialIDListener.java

```
import org.antlr.v4.runtime.TokenStream;
import org.antlr.v4.runtime.TokenStreamRewriter;

public class InsertSerialIDListener extends JavaBaseListener {
    TokenStreamRewriter rewriter;
    public InsertSerialIDListener(TokenStream tokens) {
        rewriter = new TokenStreamRewriter(tokens);
    }
```

```
    @Override
    public void enterClassBody(JavaParser.ClassBodyContext ctx) {
        String field = "\n\tpublic static final long serialVersionUID = 1L;";
        rewriter.insertAfter(ctx.start, field);
    }
}
```

The key is the TokenStreamRewriter object that knows how to give altered views of a token stream without actually modifying the stream. It treats all of the manipulation methods as "instructions" and queues them up for lazy execution when traversing the token stream to render it back as text. The rewriter executes those instructions every time we call getText().

Let's build and test the listener on the Demo.java test file we used before.

```
⇛ $ antlr4 Java.g4
⇛ $ javac InsertSerialID*.java Java*.java
⇛ $ java InsertSerialID Demo.java
❮ import java.util.List;
  import java.util.Map;
  public class Demo {
          public static final long serialVersionUID = 1L;
          void f(int x, String y) { }
          int[ ] g(/*no args*/) { return null; }
          List<Map<String, Integer>>[] h() { return null; }
  }
```

With only a few lines of code, we were able to tweak a Java class definition without disturbing anything outside of our insertion point. This strategy is very effective for the general problem of source code instrumentation or refactoring. The TokenStreamRewriter is a powerful and extremely efficient means of manipulating a token stream.

One more lexical goodie before finishing our tour involves a mundane issue but one that is a beast to solve without a general scheme like ANTLR's token channels.

Sending Tokens on Different Channels

The Java interface extractor we looked at earlier magically preserves whitespace and comments in method signatures such as the following:

```
int[ ] g(/*no args*/) { return null; }
```

Traditionally, this has been a nasty requirement to fulfill. For most grammars, comments and whitespace are things the parser can ignore. If we don't want to explicitly allow whitespace and comments all over the place in a grammar, we need the lexer to throw them out. Unfortunately, that means the whitespace

and comments are inaccessible to application code and any subsequent processing steps. The secret to preserving but ignoring comments and whitespace is to send those tokens to the parser on a "hidden channel." The parser tunes to only a single channel and so we can pass anything we want on the other channels. Here's how the Java grammar does it:

tour/Java.g4

```
COMMENT
    :   '/*' .*? '*/'    -> channel(HIDDEN) // match anything between /* and */
    ;
WS  :   [ \r\t\u000C\n]+ -> channel(HIDDEN)
    ;
```

The `-> channel(HIDDEN)` is a lexer command like the `-> skip` we discussed before. In this case, it sets the channel number of these tokens so that it's ignored by the parser. The token stream still maintains the original sequence of tokens but skips over the off-channel tokens when feeding the parser.

With these lexical features out of the way, we can wrap up our ANTLR tour. This chapter covered all of the major elements that make ANTLR easy to use and flexible. We didn't cover any of the details, but we saw ANTLR in action solving some small but real problems. We got a feel for grammar notation. We implemented visitors and listeners that let us calculate and translate without embedding actions in the grammar. We also saw that, sometimes, embedded actions are exactly what we want in order to satisfy our inner control freak. And, finally, we looked at some cool things we can do with ANTLR lexers and token streams.

It's time to slow down our pace and revisit all of the concepts explored in this chapter with the goal of learning all of the details. Each chapter in the next part of the book will take us another step toward becoming language implementers. We'll start by learning ANTLR notation and figuring out how to derive grammars from examples and language reference manuals. Once we have those fundamentals, we'll build some grammars for real-world languages and then learn the details of the tree listeners and visitors we just raced through. After that, we'll move on to some virtuoso topics in Part III.

Part II

Developing Language Applications with ANTLR Grammars

In Part II, we'll learn how to derive grammars from language specifications and sample inputs. We'll build grammars for comma-separated values, JSON, the DOT graphics format, a simple programming language, and R. Once we know how to design grammars, we'll dig into the details of building language applications by walking parse trees.

CHAPTER 5

Designing Grammars

In Part I, we got acquainted with ANTLR and saw a high-level view of grammars and language applications. Now we're going to slow down and learn the details needed to perform useful tasks such as building internal data structures, extracting information, and generating a translation of the input. The first step on our journey, though, is to learn how to build grammars. In this chapter, we'll look at the most commonly used lexical and syntactic language structures and figure out how to express them in ANTLR notation. Armed with these ANTLR building blocks, we'll combine them to build some real grammars in the next chapter.

To learn how to build grammars, we can't just wade through the various ANTLR constructs. First, we need to study the common language patterns and learn to identify them in computer language sentences. That is how we get a picture of the overall language structure. (A language pattern is a recurring grammatical structure, such as a sequence like subject–verb–object in English or subject–object–verb in Japanese.) Ultimately, we need the ability to divine a language's structure from a set of representative input files. Once we identify a language's structure, we can express it formally with an ANTLR grammar.

The good news is that there are relatively few fundamental language patterns to deal with, despite the vast number of languages invented over the past fifty years. This makes sense because people tend to design languages that follow the constraints our brains place on natural language. We expect token order to matter and expect dependencies between tokens. For example, {(} is ungrammatical because of the token order, and (1+2 drives us crazy looking for the matching). Languages also tend to be similar because designers follow common notation from mathematics. Even at the lexical level, languages tend to reuse the same structures, such as identifiers, integers, strings, and so on.

The constraints of word order and dependency, derived from natural language, blossom into four abstract computer language patterns.

- *Sequence*: This is a sequence of elements such as the values in an array initializer.
- *Choice*: This is a choice between multiple, alternative phrases such as the different kinds of statements in a programming language.
- *Token dependence*: The presence of one token requires the presence of its counterpart elsewhere in a phrase such as matching left and right parentheses.
- *Nested phrase*: This is a self-similar language construct such as nested arithmetic expressions or nested statement blocks in a programming language.

To implement these patterns, we really only need grammar rules comprised of alternatives, token references, and rule references (*Backus-Naur-Format* [BNF]). For convenience, though, we'll also group those elements into *subrules*. Subrules are just in-lined rules wrapped in parentheses. We can mark subrules as optional (?) and as zero-or-more (*) or one-or-more (+) loops to recognize the enclosed grammar fragments multiple times (*Extended Backus-Naur-Format* [EBNF]).

No doubt most readers will have seen some form of grammar or at least regular expressions during their career, but let's start at the very beginning so we're all on the same page.

5.1 Deriving Grammars from Language Samples

Writing a grammar is a lot like writing software except that we work with rules instead of functions or procedures. (Remember that ANTLR generates a function for each rule in your grammar.) But, before focusing on the rule innards, it's worth discussing the overall anatomy of a grammar and how to form an initial grammar skeleton. That's what we'll do in this section because it's an important first step in any language project. If you're itching to build and execute your first parser, you can revisit Chapter 4, *A Quick Tour*, on page 31 or jump to the first example in the next chapter: Section 6.1, *Parsing Comma-Separated Values*, on page 84. Feel free to pop back and forth to the examples in the next chapter as we learn the fundamentals here.

Grammars consist of a header that names the grammar and a set of rules that can invoke each other.

```
grammar MyG;
rule1 : «stuff» ;
rule2 : «more stuff» ;
...
```

Just like writing software, we have to figure out which rules we need, what «*stuff*» is, and which rule is the *start rule* (analogous to a main() method).

To figure all this out for a given language, we either have to know that language really well or have a set of representative input samples. Naturally, it helps to have a grammar for the language from a reference manual or even in the format of another parser generator. But for now, let's assume we don't have an existing grammar as a guide.

Proper grammar design mirrors functional decomposition or top-down design in the programming world. That means we work from the coarsest to the finest level, identifying language structures and encoding them as grammatical rules. So, the first task is to find a name for the coarsest language structure, which becomes our start rule. In English, we could use sentence. For an XML file, we could use document. For a Java file, we could use compilationUnit.

Designing the contents of the start rule is a matter of describing the overall format of the input in English pseudocode, kind of like we do when writing software. For example, "a comma-separated-value (CSV) file is a sequence of rows terminated by newlines." The essential word *file* to the left of *is a* is the rule name, and everything to the right of *is a* becomes the «*stuff*» on the right side of a rule definition.

```
file : «sequence of rows that are terminated by newlines» ;
```

Then we step down a level in granularity by describing the elements identified on the right side of the start rule. The nouns on the right side are typically references to either tokens or yet-to-be-defined rules. The tokens are elements that our brain normally latches onto as words, punctuation, or operators. Just as words are the atomic elements in an English sentence, tokens are the atoms in a parser grammar. The rule references, however, refer to other language structures that need to be broken down into more detail like *row*.

Stepping down another level of detail, we could say that a row is a sequence of *field*s separated by commas. Then, a *field* is a number or string. Our pseudocode looks like this:

```
file  : «sequence of rows that are terminated by newlines» ;
row   : «sequence of fields separated by commas» ;
field : «number or string» ;
```

When we run out of rules to define, we have a rough draft of our grammar.

Let's see how this technique works for describing some of the key structures in a Java file. (We can make the rule names stand out by italicizing them.) At the coarsest level, a Java *compilation unit* is an optional *package specifier*, followed by one or more *class definition*s. Stepping down a level, a class definition is keyword `class`, followed by an identifier, followed optionally by a *superclass specifier*, followed optionally by an *implements clause*, followed by a *class body*. A *class body* is a series of *member*s enclosed in curly braces. A *member* is a nested class definition, a *field*, or a *method*. From here, we would describe *fields* and *methods* and then the *statements* within *method*s. You get the idea. Start at the highest possible level and work your way down, treating even large subphrases like Java class definitions as rules to define later. In grammar pseudocode, we'd start out like this:

```
compilationUnit  : «optional packageSpec then classDefinitions» ;
packageSpec      : 'package' identifier ';' ;
classDefinition  :
    'class' «optional superclassSpec optional implementsClause classBody» ;
superclassSpec   : 'super' identifier ;
implementsClause :
    'implements' «one or more identifiers separated by comma» ;
classBody        : '{' «zero-or-more members» '}' ;
member           : «nested classDefinition or field or method» ;
...
```

Designing a grammar for a large language like Java is a lot easier if we have access to a grammar that we can use as a reference, but be careful. Blindly following an existing grammar can lead you astray, as we'll discuss next.

5.2 Using Existing Grammars as a Guide

Having access to existing grammar in non-ANTLR format is a great way to figure out how somebody else decided to break down the phrases in a language. At the very least, an existing grammar gives us a nice list of rule names to use as a guide. A word of caution, though. I recommend against cutting and pasting a grammar from a reference manual into ANTLR and massaging it until it works. Treat it as a guide rather than a piece of code.

Reference manuals are often pretty loose for grammar clarity reasons, meaning that the grammar recognizes lots of sentences not in the language. Or, the grammar might be ambiguous, able to match the same input sequence in more than one way. For example, a grammar might say that an expression can invoke a constructor or call a function. The problem is that input like `T(i)` could match both. Ideally, there would be no such ambiguities in our grammar.

We really need a single interpretation of each input sentence to translate it or perform some other task.

At the opposite extreme, grammars in reference manuals sometimes overspecify the rules. There are some constraints that we should enforce after parsing the input, rather than trying to enforce the constraints with grammatical structure. For example, when working on Section 12.4, *Parsing and Lexing XML*, on page 224, I scanned through the W3C XML language definition and got lost in all the details. As a trivial example, the XML grammar explicitly specifies where we must have whitespace in a tag and where it's optional. That's good to know, but we can simply have a lexer strip out whitespace inside of tags before sending it to the parser. Our grammar need not have tests for whitespace everywhere.

The specification also says that the `<?xml ...>` tag can have two special attributes: `encoding` and `standalone`. We need to know that constraint, but it's easier to allow any attribute name and then, after parsing, inspect the parse tree to ensure all such constraints are satisfied. In the end, XML is just a bunch of tags embedded in text, so its grammatical structure is fairly straightforward. The only challenge is treating what's inside and outside the tags differently. We'll look at this more in Section 12.3, *Islands in the Stream*, on page 219.

Identifying the grammar rules and expressing their right sides in pseudocode is challenging at first but gets easier and easier as you build grammars for more languages. You'll get lots of practice as you go through the examples in this book.

Once we have pseudocode, we need to translate it to ANTLR notation to get a working grammar. In the next section, we'll define four language patterns found in just about any language and see how they map to ANTLR constructs. After that, we'll figure out how to define the tokens referenced in our grammars, such as integer and identifier. Remember that we're looking at grammar development fundamentals in this chapter. It will give us the solid footing we need in order to tackle the real-world examples in the next chapter.

5.3 Recognizing Common Language Patterns with ANTLR Grammars

Now that we have a general top-down strategy for roughing out a grammar, we need to focus on the common language patterns: sequence, choice, token dependence, and nested phrase. We saw a few examples of these patterns in the previous section, but now we're going to see many more examples from a variety of languages. As we go along, we'll learn basic ANTLR notation by

expressing the specific patterns as formal grammar rules. Let's start with the most common language pattern.

Pattern: Sequence

The structure you'll see most often in computer languages is a sequence of elements, such as the sequence of methods in a class definition. Even simple languages like the HTTP, POP, and SMTP network protocols exhibit the sequence pattern. Protocols expect a sequence of commands. For example, here's the sequence to log into a POP server and get the first message:

```
USER parrt
PASS secret
RETR 1
```

Even the commands themselves are sequences. Most commands are a keyword (reserved identifier), such as `USER` and `RETR`, followed by an operand and then newline. For example, in a grammar we'd say that the retrieve command is a keyword followed by an integer followed by a newline token. To specify such a sequence in a grammar, we simply list the elements in order. In ANTLR notation, the retrieve command is just sequence `'RETR' INT '\n'`, where `INT` represents the integer token type.

```
retr  : 'RETR' INT '\n' ; // match keyword integer newline sequence
```

Notice that we can include any simple sequence of letters, such as keywords or punctuation, directly as string literals like `'RETR'` in the grammar. (We'll explore lexical structures such as `INT` in Section 5.5, *Recognizing Common Lexical Structures*, on page 72.)

We use grammar rules to label language structures just like we label statement lists as functions in a programming language. In this case, we're labeling the `RETR` sequence as the `retr` rule. Elsewhere in the grammar, we can refer to the `RETR` sequence with the rule name as a shorthand.

Let's look at an arbitrarily long sequence such as the simple list of integers in a Matlab vector like `[1 2 3]`. As with a finite sequence, we want one element to follow the next, but we can't list all possible integer lists with rule fragments like `INT INT INT INT INT INT INT INT INT`....

To encode a sequence of one or more elements, we use the `+` subrule operator. For example, `(INT)+` describes an arbitrarily long sequence of integers. As a shorthand, `INT+` is OK too. To specify that a list can be empty, we use the zero-or-more `*` operator: `INT*`. This operator is analogous to a loop in a programming language, which of course is how ANTLR-generated parsers implement them.

Variations on this pattern include the *sequence with terminator* and *sequence with separator*. CSV files demonstrate both nicely. Here's how we can express the pseudocode grammar from the previous section in ANTLR notation:

```
file : (row '\n')* ;        // sequence with a '\n' terminator
row  : field (',' field)* ; // sequence with a ',' separator
field: INT ;                // assume fields are just integers
```

Rule file uses the list with terminator pattern to match zero or more row '\n' sequences. The '\n' token terminates each element of the sequence. Rule row uses the list with separator pattern by matching a field followed by zero or more ',' field sequences. The ',' separates the fields. row matches sequences like 1 and 1,2 and 1,2,3, and so on.

We see the same constructs in programming languages. For example, here's how to recognize statement sequences in a programming language like Java where each statement is terminated by a semicolon:

```
stats : (stat ';')* ; // match zero or more ';'-terminated statements
```

And here is how to specify a comma-separated list of expressions such as we'd find in a function call argument list:

```
exprList : expr (',' expr)* ;
```

Even ANTLR's metalanguage uses sequence patterns. Here's partially how ANTLR expresses rule definition syntax in its own syntax:

```
// match 'rule-name :' followed by at least one alternative followed
// by zero or more alternatives separated by '|' symbols followed by ';'
rule : ID ':' alternative ('|' alternative )* ';' ;
```

Finally, there is a special kind of zero-or-one sequence, specified with the ?, that we use to express optional constructs. In a Java grammar, for example, we might find a sequence like ('extends' identifier)? that matches the optional superclass specification. Similarly, to match an optional initializer on a variable definition, we could say ('=' expr)?. The optional operator is kind of like a choice between something and nothing. As a preview to the next section, ('=' expr)? is the same as ('=' expr |).

Pattern: Choice (Alternatives)

A language with only one sentence would be pretty boring. Even the simplest languages, such as network protocols, have multiple valid sentences such as the USER and RETR commands of POP. This brings us to the choice pattern. We've already seen a choice in the Java grammar pseudocode «*nested class-Definition or field or method*» for rule member.

To express the notion of choice in a language, we use | as the "or" operator in ANTLR rules to separate grammatical choices called *alternatives* or *productions*. Grammars are full of choices.

Returning to our CSV grammar, we can make a more flexible field rule by allowing the choice of integers or strings.

```
field : INT | STRING ;
```

Looking through the grammars in the next chapter, we'll find lots of choice pattern examples, such as the list of type names in rule type from Section 6.4, *Parsing Cymbol*, on page 98.

```
type:   'float' | 'int' | 'void' ; // user-defined types
```

In Section 6.3, *Parsing DOT*, on page 93, we'll see the list of possible statements in a graph description.

```
stmt:   node_stmt
    |   edge_stmt
    |   attr_stmt
    |   id '=' id
    |   subgraph
    ;
```

Any time you find yourself saying "language structure *x* can be either this or that," then you've identified a choice pattern. Use | in rule *x*.

Grammar sequences and choices let us encode lots of language constructs, but there are two key remaining patterns to look at: token dependency and phrase nesting. They're typically used together in grammars, but let's start with token dependencies in isolation for simplicity.

Pattern: Token Dependency

Previously, we used INT+ to express the nonempty sequence of integers in a Matlab vector, [1 2 3]. To specify a vector with the surrounding square brackets, we need a way to express dependencies between tokens. If we see one symbol in a sentence, we must find its matching counterpart elsewhere in the sentence. To express this with a grammar, we use a sequence that specifies both symbols, usually enclosing or grouping other elements. In this case, we completely specify vectors like this:

```
vector : '[' INT+ ']' ; // [1], [1 2], [1 2 3], ...
```

Glance at any nontrivial program in your favorite language, and you'll see all sorts of grouping symbols that must occur in pairs: (...), {...}, and [...]. From

Section 6.4, *Parsing Cymbol*, on page 98, we find token dependencies between the parentheses of a method call and the square brackets of an array index.

```
expr:   expr '(' exprList? ')'  // func call like f(), f(x), f(1,2)
    |   expr '[' expr ']'       // array index like a[i], a[i][j]
   ...
    ;
```

We also see token dependencies between left and right parentheses in method declarations.

examples/Cymbol.g4

```
functionDecl
    :   type ID '(' formalParameters? ')' block // "void f(int x) {...}"
    ;
formalParameters
    :   formalParameter (',' formalParameter)*
    ;
formalParameter
    :   type ID
    ;
```

The grammar from Section 6.2, *Parsing JSON*, on page 86 matches up curly braces around object definitions such as { "name":"parrt", "passwd":"secret" }.

examples/JSON.g4

```
object
    :   '{' pair (',' pair)* '}'
    |   '{' '}' // empty object
    ;
pair:   STRING ':' value ;
```

See Section 6.5, *Parsing R*, on page 102 for more examples of matching tokens.

Keep in mind that dependent symbols don't necessarily have to match. C-derived languages also have the a?b:c ternary operator where the ? sets up a requirement to see : later in the phrase.

Also, just because we have matching tokens doesn't necessarily imply a nested phrase. For example, a vector might not allow nested vectors. In general, though, subphrases enclosed in matching symbols typically nest. We get constructs like a[(i)] and {while (b) {i=1;}}. This brings us to the final language pattern we're likely to need.

Pattern: Nested Phrase

A nested phrase has a self-similar language structure, one whose subphrases conform to that same structure. Expressions are the quintessential self-similar language structure and are made up of nested subexpressions separated

by operators. Similarly, a while's code block is a code block nested within an outer code block. We express self-similar language structures using recursive rules in grammars. So, if the pseudocode for a rule references itself, we are going to need a recursive (self-referencing) rule.

Let's see how nesting works for code blocks. A while statement is the keyword while followed by a condition expression in parentheses followed by a statement. We can also treat multiple statements as a single block statement by wrapping them in curly braces. Expressing that grammatically looks like this:

```
stat: 'while' '(' expr ')' stat  // match WHILE statement
    | '{' stat* '}'              // match block of statements in curlies
   ...                           // and other kinds of statements
    ;
```

The looping statement, stat, of the while can be a single statement or a group of statements if we enclose them in {...}. Rule stat is *directly recursive* because it refers to itself in the first (and second) alternatives. If we moved the second alternative to its own rule, rules stat and block would be mutually *indirectly recursive*.

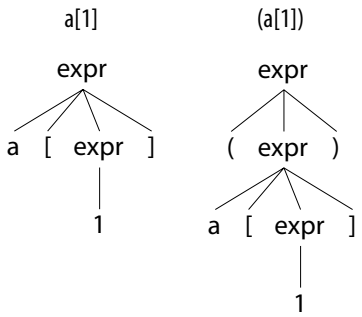
```
stat: 'while' '(' expr ')' stat  // match WHILE statement
    | block                      // match a block of statements
   ...                           // and other kinds of statements
    ;
block: '{' stat* '}' ;           // match block of statements in curlies
```

Most nontrivial languages have multiple self-similar constructs, leading to lots of recursive rules. Let's look at an expression rule for a simple language that has just three kinds of expressions: indexed array references, parenthesized expressions, and integers. Here's how we would express that in ANTLR notation:

```
expr: ID '[' expr ']'  // a[1], a[b[1]], a[(2*b[1])]
    | '(' expr ')'     // (1), (a[1]), (((1))), (2*a[1])
    | INT              // 1, 94117
    ;
```

Notice how the recursion happens naturally. The index component of an array index expression is itself an expression, so we just reference expr in that alternative. The fact that the array index alternative is itself an expression shouldn't bother us. The nature of the language construct dictates the use of a rule reference that just happens to be recursive.

Here are the parse trees for two sample inputs:



As we saw in Section 2.1, *Let's Get Meta!*, on page 9, the internal tree nodes are rule references, and the leaves are token references. A path from the root of the tree to any node represents the rule invocation stack for that element (or call stack for an ANTLR-generated recursive-descent parser). Paths representing recursive, nested subtrees have multiple references to the same rule. I like to think of the rule nodes as labeling the subtrees underneath. The root is `expr`, so the entire tree is an expression. The subtree with `expr` before `1` labels that integer as an expression.

Not all languages have expressions, such as data formats, but most languages you'll run into have fairly complex expressions (see Section 6.5, *Parsing R*, on page 102). Moreover, expression grammar specifications are not always obvious, so it's worth spending some time digging into the details of recognizing expressions. We'll do that next.

For future reference, here's a table summarizing ANTLR's core grammar notation:

Syntax	Description
x	Match token, rule reference, or subrule *x*.
x y ... z	Match a sequence of rule elements.
(... | ... | ...)	Subrule with multiple alternatives.
***x*?**	Match *x* or skip it.
x*	Match *x* zero or more times.
***x*+**	Match *x* one or more times.
r : ... ;	Define rule r.
r : ... | ... | ... ;	Define rule r with multiple alternatives.

Table 1—ANTLR Core Notation

And here is a table summarizing what we've learned so far about common computer language patterns:

Pattern Name	Description
Sequence	This is a finite or arbitrarily long sequence of tokens or subphrases. Examples include variable declarations (type followed by identifier) and lists of integers. Here are some sample implementations:

```
x y ... z           // x followed by y, ..., z
'[' INT+ ']'        // Matlab vector of integers
```

Sequence with terminator

This is an arbitrarily long, potentially empty sequence of tokens or subphrases separated by a token, usually a semicolon or newline. Examples include statement lists from C-like languages and rows of data terminated with newlines. Here are some sample implementations:

```
(statement ';')*    // Java statement list
(row '\n')*         // Lines of data
```

Sequence with separator

This is a nonempty arbitrarily long sequence of tokens or subphrases separated by a token, usually a comma, semicolon, or period. Examples include function argument definition lists, function call argument lists, languages where statements are separated but not terminated, and directory names. Here are some sample implementations:

```
expr (',' expr)*        // function call arguments
( expr (',' expr)* )?   // optional function call arguments
'/'? name ('/' name)*   // simplified directory name
stat ('.' stat)*        // SmallTalk statement list
```

Choice

This is a set of alternative phrases. Examples include the different kinds of types, statements, expressions, or XML tags. Here are some sample implementations:

```
type : 'int' | 'float' ;
stat : ifstat | whilestat | 'return' expr ';' ;
expr : '(' expr ')' | INT | ID ;
tag  : '<' Name attribute* '>' | '<' '/' Name '>' ;
```

Token dependency

The presence of one token requires the presence of one or more subsequent tokens. Examples include matching parentheses, curly braces, square bracket, and angle brackets. Here are some sample implementations:

```
'(' expr ')'          // nested expression
ID '[' expr ']'       // array index
'{' stat* '}'         // statements grouped in curlies
'<' ID (',' ID)* '>' // generic type specifier
```

Pattern Name	Description
Nested phrase	This is a self-similar language structure. Examples include expressions, nested Java classes, nested code blocks, and nested Python function definitions. Here are some sample implementations:

```
expr : '(' expr ')' | ID ;
classDef : 'class' ID '{' (classDef|method|field) '}' ;
```

5.4 Dealing with Precedence, Left Recursion, and Associativity

Expressions have always been a hassle to specify with top-down grammars and to recognize by hand with recursive-descent parsers, first because the most natural grammar is ambiguous and second because the most natural specification uses a special kind of recursion called *left recursion*. We'll discuss the latter in detail later, but for now, keep in mind that top-down grammars and parsers cannot deal with left recursion in their classic form.

To illustrate the problem, imagine a simple arithmetic expression language that has multiply and addition operators and integer "atoms." Expressions are self-similar, so it's natural for us to say that a multiplicative expression is two subexpressions joined by the * operator. Similarly, an additive expression is two subexpressions joined by a +. We can also have simple integers as expressions. Literally encoding this as a grammar leads to the following reasonable-looking rule:

```
expr : expr '*' expr  // match subexpressions joined with '*' operator
     | expr '+' expr  // match subexpressions joined with '+' operator
     | INT            // matches simple integer atom
     ;
```

The problem is that this rule is ambiguous for some input phrases. In other words, this rule can match a single input stream in more than one way, which you might recall from Section 2.3, *You Can't Put Too Much Water into a Nuclear Reactor*, on page 13. It's fine for simple integers and for single-operator expressions such as `1+2` and `1*2` because there is only one way to match them. For example, the rule can match `1+2` only using the second alternative, as shown by the left parse tree in the diagram in Figure 3, *Parse tree interpretations*, on page 70

The problem is that the rule as specified can interpret input such as `1+2*3` in the two ways depicted by the middle and right parse trees. They're different because the middle tree says to add 1 to the result of multiplying 2 and 3, whereas the tree on the right multiplies 3 by the result of adding 1 and 2.

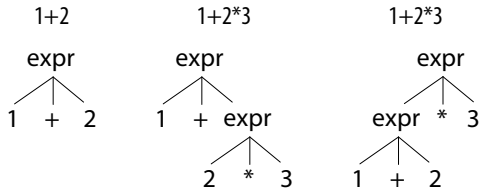


Figure 3—Parse tree interpretations

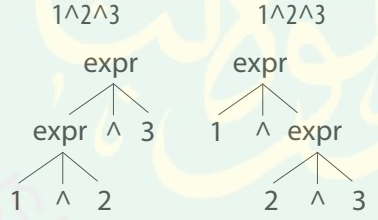
This is a question of operator precedence, and conventional grammars simply have no way to specify precedence. Most grammar tools, such as Bison,[1] use extra notation to specify the operator precedence.

Instead, ANTLR resolves ambiguities in favor of the alternative given first, implicitly allowing us to specify operator precedence. Rule `expr` has the multiplication alternative before the addition alternative, so ANTLR resolves the operator ambiguity for `1+2*3` in favor of the multiplication.

By default, ANTLR associates operators left to right as we'd expect for * and +. Some operators like exponentiation group right to left, though, so we have to manually specify the associativity on the operator token using option `assoc`. Here's an expression rule that properly interprets input like `2^3^4` as `2^(3^4)`:

```
expr : expr '^'<assoc=right> expr // ^ operator is right associative
     | INT
     ;
```

The following parse trees illustrates the difference between left and right associative versions of ^. The parse tree on the right is the usual interpretation, but language designers are free to use either associativity.



To combine all three operators in a single rule, we place the ^ alternative before the others because it has higher precedence than * and + (`1+2^3` is 9).

1. http://dinosaur.compilertools.net/bison/bison_8.html#SEC71

```
expr : expr '^'<assoc=right> expr  // ^ operator is right associative
     | expr '*' expr  // match subexpressions joined with '*' operator
     | expr '+' expr  // match subexpressions joined with '+' operator
     | INT            // matches simple integer atom
     ;
```

Readers familiar with ANTLR v3 have been waiting patiently for me to point out that ANTLR, like all conventional top-down parser generators, cannot handle left-recursive rules. However, one of ANTLR v4's major improvements is that it can now handle direct left recursion. A left-recursive rule is one that either directly or indirectly invokes itself on the left edge of an alternative. The expr rule is directly left recursive because everything but the INT alternative starts with a reference to the expr rule itself. (It's also *right recursive* because of the expr references on the right edges of some alternatives.)

While ANTLR v4 can handle *direct* left recursion, it can't handle *indirect* left recursion. That means we can't factor expr into the grammatically equivalent rules.

```
expr : expo  // indirectly invokes expr left recursively via expo
     | ...
     ;
expo : expr '^'<assoc=right> expr ;
```

Precedence Climbing Expression Parsing

Experienced compiler writers often build recursive-descent parsers by hand to squeeze out every last drop of performance and to allow complete control over error recovery. Instead of writing code for the long chain of expression rules, however, they often use *operator precedence parsers*.[a]

ANTLR uses a similar but more powerful strategy than operator precedence that follows work done by Keith Clarke[b] from 1986. Theodore Norvell subsequently coined the term *precedence climbing*.[c] Similarly, ANTLR replaces direct left recursion with a predicated loop that compares the precedence of the previous and next operators. We'll get into predicates in Chapter 11, *Altering the Parse with Semantic Predicates*, on page 189.

a. http://en.wikipedia.org/wiki/Operator-precedence_parser
b. http://antlr.org/papers/Clarke-expr-parsing-1986.pdf
c. http://www.engr.mun.ca/~theo/Misc/exp_parsing.htm

To recognize expressions with ANTLR v3, we had to unravel the left recursion we saw in the earlier expr into multiple rules, one for each precedence level. For example, we'd use rules like the following for expressions with multiplication and addition operators:

```
expr    : addExpr ;
addExpr : multExpr ('+' multExpr)* ;
multExpr: atom ('*' atom)* ;
atom    : INT ;
```

Expressions for languages such as C and Java end up with about fifteen such rules, which is a hassle whether we're building a top-down grammar or building a recursive-descent parser by hand.

ANTLR v4 makes short work of (directly) left-recursive expression rules. Not only is the new mechanism more efficient, expression rules are much smaller and easier to understand. For example, in a Java grammar, the number of lines dedicated to expressions dropped in half (from 172 to 91 lines).

In practice, we can handle all of the language structures we care about with direct left recursion. For example, here's a rule that matches a subset of the C declarator language including input such as *(*a)[][]:

```
decl : decl '[' ']'  // match [] suffixes using direct left-recursion
     | '*' decl      // *x, *x[], **x
     | '(' decl ')'  // (x), (x[]), (*x)[]
     | ID
     ;
```

To learn more about how ANTLR supports direct left recursion (using grammar transformations), please see Chapter 14, *Removing Direct Left Recursion*, on page 247.

At this point, we've studied the common patterns found in computer languages and figured out how to express them in ANTLR notation. But, before we can dive into some complete examples, we need to figure out how to describe the tokens referenced in our grammar rules. Just as there are a few key grammatical language patterns, we'll find that there are some extremely common lexical structures. Creating a complete grammar is a matter of combining grammatical rules from this section and lexical rules from the next section.

5.5 Recognizing Common Lexical Structures

Computer languages look remarkably similar lexically. For example, if I scramble up the order to obscure grammatical information, tokens) 10 (f could be combined into valid phrases from the earliest languages to the most recent. Fifty years ago, we'd see (f 10) in LISP and f(10) in Algol. Of course, f(10) is also valid in virtually all programming languages from Prolog to Java to the new Go language.[2] Lexically, then, functional, procedural, declarative, and object-oriented languages look pretty much the same. Amazing!

2. http://golang.org/

That's great because we have to learn only how to describe identifiers and integers once and, with little variation, apply them to most programming languages. As with parsers, lexers use rules to describe the various language constructs. We get to use essentially the same notation. The only difference is that parsers recognize grammatical structure in a token stream and lexers recognize grammatical structure in a character stream.

Since lexing and parsing rules have similar structures, ANTLR allows us to combine both in a single grammar file. But since lexing and parsing are two distinct phases of language recognition, we must tell ANTLR which phase is associated with each rule. We do this by starting lexer rule names with uppercase letters and parser rule names with lowercase letters. For example, `ID` is a lexical rule name, and `expr` is a parser rule name.

When starting a new grammar, I typically cut and paste rules from an existing grammar, such as Java,[3] for the common lexical constructs: *identifiers*, *numbers*, *strings*, *comments*, and *whitespace*. A few tweaks, and I'm up and running. Almost all languages, even nonprogramming languages like XML and JSON, have a variation of those tokens. For example, despite being very distinct grammatically, a lexer for C would have no problem tokenizing the following JSON:

```
{
  "title":"Cat wrestling",
  "chapters":[ {"Intro":"..."}, ... ]
}
```

As another example, consider block comments. In Java, they are bracketed by `/* ... */`, and in XML, comments are bracketed by `<!-- ... -->`, but they are more or less the same lexical construct except for the start and stop symbols.

For keywords, operators, and punctuation, we don't need lexer rules because we can directly reference them in parser rules in single quotes like `'while'`, `'*'`, and `'++'`. Some developers prefer to use lexer rule references such as `MULT` instead of literal `'*'`. That way, they can change the multiply operator character without altering the references to `MULT` in the parser rules. Having both the literal and lexical rule `MULT` is no problem; they both result in the same token type.

To demonstrate what lexical rules look like, let's build simple versions of the common tokens, starting with our friend the humble identifier.

3. http://www.antlr.org/grammar/java

Matching Identifiers

In grammar pseudocode, a basic identifier is a nonempty sequence of uppercase and lowercase letters. Using our newfound skills, we know to express the sequence pattern using notation (...)+. Because the elements of the sequence can be either uppercase or lowercase letters, we also know that we'll have a choice operator inside the subrule.

```
ID : ('a'..'z'|'A'..'Z')+ ; // match 1-or-more upper or lowercase letters
```

The only new ANTLR notation here is the range operator: 'a'..'z' means any character from a to z. That is literally the ASCII code range from 97 to 122. To use Unicode code points, we need to use '\uXXXX' literals where XXXX is the hexadecimal value for the Unicode character code point value.

As a shorthand for character sets, ANTLR supports the more familiar regular expression set notation.

```
ID : [a-zA-Z]+ ; // match 1-or-more upper or lowercase letters
```

Rules such as ID sometimes conflict with other lexical rules or literals referenced in the grammar such as 'enum'.

```
grammar KeywordTest;
enumDef : 'enum' '{' ... '}' ;
...
FOR : 'for' ;
...
ID : [a-zA-Z]+ ; // does NOT match 'enum' or 'for'
```

Rule ID could also match keywords such as enum and for, which means there's more than one rule that could match the same string. To make this clearer, consider how ANTLR handles combined lexer/parser grammars such as this. ANTLR collects and separates all of the string literals and lexer rules from the parser rules. Literals such as 'enum' become lexical rules and go immediately after the parser rules but before the explicit lexical rules.

ANTLR lexers resolve ambiguities between lexical rules by favoring the rule specified first. That means your ID rule should be defined after all of your keyword rules, like it is here relative to FOR. ANTLR puts the implicitly generated lexical rules for literals before explicit lexer rules, so those always have priority. In this case, 'enum' is given priority over ID automatically.

Because ANTLR reorders the lexical rules to occur after the parser rules, the following variation on KeywordTest results in the same parser and lexer:

```
grammar KeywordTestReordered;
FOR : 'for' ;
ID : [a-zA-Z]+ ; // does NOT match 'enum' or 'for'
...
enumDef : 'enum' '{' ... '}' ;
...
```

This definition of an identifier doesn't allow numbers, but you can peek ahead to Section 6.3, *Parsing DOT*, on page 93, Section 6.4, *Parsing Cymbol*, on page 98, and Section 6.5, *Parsing R*, on page 102 for full-blown ID rules.

Matching Numbers

Describing integer numbers such as 10 is easy because it's just a sequence of digits.

```
INT : '0'..'9'+ ; // match 1 or more digits
```

or

```
INT : [0-9]+ ; // match 1 or more digits
```

Floating-point numbers are much more complicated, unfortunately, but we can make a simplified version easily if we ignore exponents. (See Section 6.5, *Parsing R*, on page 102 for lexical rules that match full floating-point numbers and even complex numbers like 3.2i.) A floating-point number is a sequence of digits followed by a period and then optionally a fractional part, or it starts with a period and continues with a sequence of digits. A period by itself is not legal. Our floating-point rule therefore uses a choice and a few sequence patterns.

```
FLOAT:  DIGIT+ '.' DIGIT*  // match 1. 39. 3.14159 etc...
    |          '.' DIGIT+  // match .1 .14159
    ;

fragment
DIGIT   :   [0-9] ;        // match single digit
```

Here we're also using a helper rule, DIGIT, so we don't have to write [0-9] everywhere. By prefixing the rule with fragment, we let ANTLR know that the rule will be used only by other lexical rules. It is not a token in and of itself. This means that we could not reference DIGIT from a parser rule.

Matching String Literals

The next token that computer languages tend to have in common is the string literal like "Hello". Most use double quotes, but some use single quotes or even both (Python). Regardless of the choice of delimiters, we match them using a

rule that consumes everything between the delimiters. In grammar pseudocode, a string is a sequence of any characters between double quotes.

```
STRING : '"' .*? '"' ; // match anything in "..."
```

The dot *wildcard* operator matches any single character. Therefore, .* would be a loop that matches any sequence of zero or more characters. Of course, that would consume until the end of file, which is not very useful. Instead, ANTLR provides support for *nongreedy subrules* using standard regular expression notation (the ? suffix). Nongreedy means essentially to "scarf characters until you see what follows the subrule in the lexer rule." To be more precise, nongreedy subrules match the fewest number of characters while still allowing the entire surrounding rule to match. See Section 15.6, *Wildcard Operator and Nongreedy Subrules*, on page 283 for more details. In contrast, the .* is considered *greedy* because it greedily consumes characters that match the inside of the loop (wildcard in this case). If .*? is confusing, don't worry about it. Just remember it as a pattern for matching stuff inside quotes or other delimiters. We'll see nongreedy loops again shortly when we look at comments.

Our STRING rule isn't quite good enough yet because it doesn't allow double quotes inside strings. To support that, most languages define escape sequences starting with a backslash. To get a double quote inside a double-quoted string, we use \". To support the common escape characters, we need something like the following:

```
STRING: '"' (ESC|.)*? '"' ;
fragment
ESC : '\\"' | '\\\\' ; // 2-char sequences \" and \\
```

ANTLR itself needs to escape the escape character, so that's why we need \\ to specify the backslash character.

The loop in STRING now matches either an escape character sequence, by calling fragment rule ESC, or any single character via the dot wildcard. The *? subrule operator terminates the (ESC|.)*? loop upon seeing what follows, an unescaped double-quote character.

Matching Comments and Whitespace

When a lexer matches the tokens we've defined so far, it emits them via the token stream to the parser. The parser then checks the grammatical structure of the stream. But when the lexer matches comment and whitespace tokens, we'd like it to toss them out. That way, the parser doesn't have to worry about matching optional comments and whitespace everywhere. For example, the

following parser rule would be very awkward and error-prone where WS is a whitespace lexical rule:

```
assign : ID (WS|COMMENT)? '=' (WS|COMMENT)? expr (WS|COMMENT)? ;
```

Defining these discarded tokens is the same as for nondiscarded tokens. We just have to indicate that the lexer should throw them out using the skip command. For example, here is how to match both single-line and multiline comments for C-derived languages:

```
LINE_COMMENT : '//' .*? '\r'? '\n' -> skip ; // Match "//" stuff '\n'
COMMENT      : '/*' .*? '*/'       -> skip ; // Match "/*" stuff "*/"
```

In LINE_COMMENT, .*? consumes everything after // until it sees a newline (optionally preceded by a carriage return to match Windows-style newlines). In COMMENT, .*? consumes everything after /* and before the terminating */.

The lexer accepts a number of commands following the -> operator; skip is just one of them. For example, we have the option to pass these tokens to the parser on a "hidden channel" by using the channel command. See Section 12.1, *Broadcasting Tokens on Different Channels*, on page 204 for more on token channels.

Let's deal with whitespace, our final common token. Most programming languages treat whitespace characters as token separators but otherwise ignore them. (Python is an exception because it uses whitespace for particular syntax purposes: newlines to terminate commands and indent level, with initial tabs or spaces to indicate nesting level.) Here is how to tell ANTLR to throw out whitespace:

```
WS : (' '|'\t'|'\r'|'\n')+ -> skip ; // match 1-or-more whitespace but discard
```

or

```
WS : [ \t\r\n]+ -> skip ; // match 1-or-more whitespace but discard
```

When newline is both whitespace to be ignored and the command terminator, we have a problem. Newline is context-sensitive. In one grammatical context, we should throw out newlines, and in another, we should pass it to the parser so that it knows a command has finished. For example, in Python, f() followed by newline executes the code, calling f(). But we could also insert an extra newline between the parentheses. Python waits until the newline after the) before executing the call.

```
⇛ $ python
⇛ >>> def f(): print "hi"
❮ ...
⇛ >>> f()
```

```
❮ hi
⇛ >>> f(
⇛ ... )
❮ hi
```

For a detailed discussion of the problem and solutions, see *Fun with Python Newlines*, on page 214.

So, now we know how to match basic versions of the most common lexical constructs: identifiers, numbers, strings, comments, and whitespace. Believe it or not, that's a great start on a lexer for even a big programming language. Here's a lexer starter kit we can use as a reference later:

Token Category	Description and Examples

Punctuation	The easiest way to handle operators and punctuation is to directly reference them in parser rules.
	`call : ID '(' exprList ')' ;`
	Some programmers prefer to define token labels such as `LP` (left parenthesis) instead.
	`call : ID LP exprList RP ;`
	`LP : '(' ;`
	`RP : ')' ;`

Keywords	Keywords are reserved identifiers, and we can either reference them directly or define token types for them. `returnStat : 'return' expr ';'`

Identifiers	Identifiers look almost the same in every language, with some variation about what the first character can be and whether Unicode characters are allowed.
	`ID : ID_LETTER (ID_LETTER | DIGIT)* ; // From C language`
	`fragment ID_LETTER : 'a'..'z'|'A'..'Z'|'_' ;`
	`fragment DIGIT : '0'..'9' ;`

Numbers	These are definitions for integers and simple floating-point numbers.
	`INT : DIGIT+ ;`
	`FLOAT`
	`  : DIGIT+ '.' DIGIT*`
	`  | '.' DIGIT+`
	`  ;`

Token Category	Description and Examples

Strings	Match double-quoted strings.
	`STRING :  '"' ( ESC | . )*? '"' ;`
	`fragment ESC : '\\' [btnr"\\] ; // \b, \t, \n etc...`

Comments	Match and discard comments.
	`LINE_COMMENT : '//' .*? '\n' -> skip ;`
	`COMMENT      : '/*' .*? '*/' -> skip ;`

Whitespace	Match whitespace in the lexer and throw it out. `WS : [ \t\n\r]+ -> skip ;`

At this point, we have a strategy to go from sample input files to parser and lexer rules and are ready to tackle the examples in the next chapter. Before we move on, though, there are two important issues to consider. First, it's not always obvious where to draw the line between what we match in the parser and what we match in the lexer. Second, ANTLR places a few constraints on our grammar rules that we should know about.

5.6 Drawing the Line Between Lexer and Parser

Because ANTLR lexer rules can use recursion, lexers are technically as powerful as parsers. That means we could match even grammatical structure in the lexer. Or, at the opposite extreme, we could treat characters as tokens and use a parser to apply grammatical structure to a character stream. (These are called *scannerless parsers*. See `code/extras/CSQL.g4` for a grammar matching a small mix of C + SQL.)

Where to draw the line between the lexer and the parser is partially a function of the language but also a function of the intended application. Fortunately, a few rules of thumb will get us pretty far.

- Match and discard anything in the lexer that the parser does not need to see at all. Recognize and toss out things like whitespace and comments for programming languages. Otherwise, the parser would have to constantly check to see whether there are comments or whitespace in between tokens.
- Match common tokens such as identifiers, keywords, strings, and numbers in the lexer. The parser has more overhead than the lexer, so we shouldn't burden the parser with, say, putting digits together to recognize integers.

- Lump together into a single token type those lexical structures that the parser does not need to distinguish. For example, if our application treats integer and floating-point numbers the same, then lump them together as token type `NUMBER`. There's no point in sending separate token types to the parser.
- Lump together anything that the parser can treat as a single entity. For example, if the parser doesn't care about the contents of an XML tag, the lexer can lump everything between angle brackets into a single token type called `TAG`.
- On the other hand, if the parser needs to pull apart a lump of text to process it, the lexer should pass the individual components as tokens to the parser. For example, if the parser needs to process the elements of an IP address, the lexer should send individual tokens for the IP components (integers and periods).

When we say that the parser doesn't need to distinguish between certain lexical structures or doesn't care about what's inside a structure, we really mean that our application doesn't care. Our application performs the same action or translation on those lexical structures.

To see how the intended application affects what we match in the lexer vs. the parser, imagine processing a log file from a web server that has one record per line. We'll gradually increase the application requirements to see how it shifts the boundary between the lexer and the parser. Let's assume each row has a requesting IP address, HTTP protocol command, and result code. Here's a sample log entry:

```
192.168.209.85 "GET /download/foo.html HTTP/1.0" 200
```

Our brain naturally picks out the various lexical elements, but if all we want to do is count how many lines there are in the file, we can ignore everything but the sequence of newline characters.

```
file  : NL+ ;              // parser rule matching newline (NL) sequence
STUFF : ~'\n'+ -> skip ;   // match and discard anything but a '\n'
NL    : '\n' ;             // return NL to parser or other invoking code
```

The lexer doesn't have to recognize much in the way of structure, and the parser matches a sequence of newline tokens. (The `~x` operator matches anything but `x`.)

Next, let's say that we need to collect a list of IP addresses from the log file. This means we need a rule to recognize the lexical structure of an IP address, and we might as well provide lexer rules for the other record elements.

```
IP    : INT '.' INT '.' INT '.' INT ; // 192.168.209.85
INT   : [0-9]+ ;        // match IP octet or HTTP result code
STRING: '"' .*? '"' ;   // matches the HTTP protocol command
NL    : '\n' ;          // match log file record terminator
WS    : ' ' -> skip ;   // ignore spaces
```

With a complete set of tokens, we can make parser rules that match the records in a log file.

```
file  : row+ ;              // parser rule matching rows of log file
row   : IP STRING INT NL ;  // match log file record
```

Stepping up our processing needs a little bit, let's say we need to convert the text IP addresses to 32-bit numbers. With convenient library functions like split('.'), we could pass IP addresses as strings to the parser and process them there. But, it's better to have the lexer match the IP address lexical structure and pass the components to the parser as tokens.

```
file  : row+ ;                // parser rule matching rows of log file
row   : ip STRING INT NL ;    // match log file record
ip    : INT '.' INT '.' INT '.' INT ; // match IPs in parser

INT   : [0-9]+ ;        // match IP octet or HTTP result code
STRING: '"' .*? '"' ;   // matches the HTTP protocol command
NL    : '\n' ;          // match log file record terminator
WS    : ' ' -> skip ;   // ignore spaces
```

Switching lexer rule IP to parser rule ip shows how easily we can shift the dividing line. (Converting the four INT tokens to a 32-bit number would require some application code embedded in the grammar, which we haven't looked at yet in depth, so we'll leave them out.)

If the requirements call for processing the contents of the HTTP protocol command string, we would follow a similar thought process. If our application doesn't need to examine the parts of the string, then the lexer can pass the whole string to the parser as a single token. But, if we need to pull out the various pieces, it's better to have the lexer recognize those pieces and pass the components to the parser.

It doesn't take long to get a good feel for drawing the line according to a language's symbols and the needs of an application. The examples in the next chapter will help you internalize the rules of thumb from this section. Then, with that solid foundation, we'll examine a few nasty lexical problems in Chapter 12, *Wielding Lexical Black Magic*, on page 203. For example, Java compilers need to both ignore and process Javadoc comments, and XML files have different lexical structures inside and outside of tags.

In this chapter, we learned how to work from a representative sample of the language, or language documentation, to create grammar pseudocode and then a formal grammar in ANTLR notation. We also studied the common language patterns: sequence, choice, token dependency, and nested phrase. In the lexical realm, we looked at implementations for the most common tokens: identifiers, numbers, strings, comments, and whitespace. Now it's time to put this knowledge to work building grammars for some real-world languages.

CHAPTER 6

Exploring Some Real Grammars

In the previous chapter, we studied common lexical and grammatical structures and learned how to express them as snippets of ANTLR grammars. Now it's time to put that knowledge to use building some real-world grammars. Our primary goal is to learn how to assemble full grammars by sifting through reference manuals, sample input files, and existing non-ANTLR grammars. We'll tackle five languages, ramping up gradually in complexity. You don't have to build all of them now. Just work through the ones you're comfortable with and come back as you encounter more complicated problems in practice. Also feel free to pop back to look at the patterns and ANTLR snippets in the previous chapter.

The first language we'll look at is the comma-separated-value (CSV) format used by spreadsheets and databases. CSV is a great place to start because it's simple and yet widely applicable. The second language is also a data format, called JSON,[1] that has nested data elements, which lets us explore the use of rule recursion in a real language.

Next, we'll look at a declarative language called DOT[2] for describing graphs (networks). In a declarative language, we express logical constructions without specifying control flow. DOT lets us explore more complicated lexical structures, such as case-insensitive keywords.

Our fourth language is a simple non-object-oriented programming language called Cymbol (also discussed in Chapter 6 of *Language Implementation Patterns [Par09]*). It's a prototypical grammar we can use as a reference or starting point for other imperative programming languages (those composed of functions, variables, statements, and expressions).

1. http://www.json.org
2. http://www.graphviz.org

Finally, we'll build a grammar for the R functional programming language.[3] (Functional languages compute by evaluating expressions.) R is a statistical programming language increasingly used for data analysis. I chose R because its grammar consists primarily of a jumbo expression rule. It's a great opportunity to reinforce our understanding of operator precedence and associativity for a real language.

Once we have a firm grip on building grammars, we can move beyond recognition and get down to the business of triggering actions when the application sees input phrases of interest. In the next chapter, we'll create parser listeners that build data structures, manage symbol tables that track variable and function definitions, and perform language translations.

We begin with a grammar for CSV files.

6.1 Parsing Comma-Separated Values

While we have already seen a basic CSV grammar in *Pattern: Sequence*, on page 62, let's beef it up with the notion of a header row and allow empty columns. Here's a representative input file:

examples/data.csv
```
Details,Month,Amount
Mid Bonus,June,"$2,000"
,January,"""zippo"""
Total Bonuses,"","$5,000"
```

Header rows are no different from regular rows; we simply interpret the column value as the column header name. Rather than using a `row+` ANTLR fragment to match rows as well as the header row, we match it separately. When building a real application based upon this grammar, we'd probably want to treat the header differently. This way, we can get a handle on the first special row. Here's the first part of the grammar:

examples/CSV.g4
```
grammar CSV;

file : hdr row+ ;
hdr : row ;
```

Note that we've introduced an extra rule called `hdr` for clarity. Grammatically it's just a `row`, but we've made its role clearer by separating it out. Compare this to using just `row+` or `row row*` on the right-side rule `file`.

3. http://www.r-project.org

Rule row is the same as before: a list of fields separated by commas and terminated by a newline.

examples/CSV.g4
```
row : field (',' field)* '\r'? '\n' ;
```

To make our fields more flexible than they were in the previous chapter, let's allow arbitrary text, strings, and even empty fields in between commas.

examples/CSV.g4
```
field
    :   TEXT
    |   STRING
    |
    ;
```

The token definitions aren't too bad. TEXT tokens are a sequence of characters until we hit the next comma field separator or the end of the line. Strings are any characters in between double quotes. Here are the two token definitions we've used so far:

examples/CSV.g4
```
TEXT : ~[,\n\r"]+ ;
STRING : '"' ('""'|~'"')* '"' ; // quote-quote is an escaped quote
```

To get a double quote inside a double-quoted string, the CSV format generally uses two double quotes in a row. That's what the ('""'|~'"')* subrule does in rule STRING. We can't use a nongreedy loop with the wildcard, ('""'|.)*?, because it would stop at the first " it saw after the start of the string. Input like "x""y" would match two strings, not one string with "" inside it. Remember that nongreedy subrules match the fewest characters possible that still results in a match for the surrounding rule.

Before testing our parser rules, let's take a look at the token stream just to make sure that our lexer breaks up the character stream properly. Using TestRig via alias grun with option -tokens, we get the following:

```
⇛ $ antlr4 CSV.g4
⇛ $ javac CSV*.java
⇛ $ grun CSV file -tokens data.csv
❮ [@0,0:6='Details',<4>,1:0]
  [@1,7:7=',',<1>,1:7]
  [@2,8:12='Month',<4>,1:8]
  [@3,13:13=',',<1>,1:13]
  [@4,14:19='Amount',<4>,1:14]
  [@5,20:20='\n',<2>,1:20]
  [@6,21:29='Mid Bonus',<4>,2:0]
  [@7,30:30=',',<1>,2:9]
  [@8,31:34='June',<4>,2:10]
```

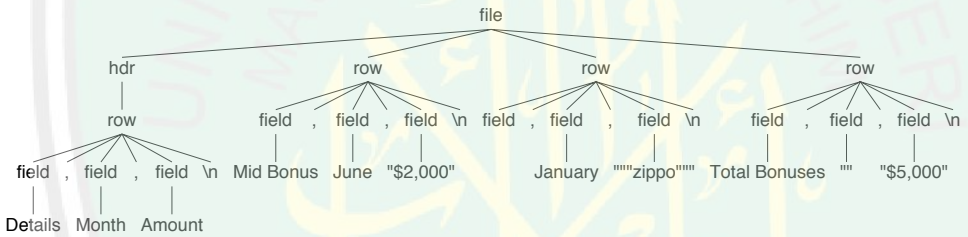
```
[@9,35:35=',',<1>,2:14]
[@10,36:43='"$2,000"',<5>,2:15]
[@11,44:44='\n',<2>,2:23]
[@12,45:45=',',<1>,3:0]
[@13,46:52='January',<4>,3:1]
...
```

Those tokens look fine. The punctuation, text, and strings all come through as expected.

Now, let's see how our grammar recognizes grammatical structure in the input stream. Using option -tree, the test rig prints out a text form of the parse tree (cleaned up for the book).

```
⇒ $ grun CSV file -tree data.csv
❮ (file
   (hdr (row (field Details) , (field Month) , (field Amount) \n))
   (row (field Mid Bonus) , (field June) , (field "$2,000") \n)
   (row field , (field January) , (field """zippo""") \n)
   (row (field Total Bonuses) , (field "") , (field "$5,000") \n)
  )
```

The root node represents everything that start rule file matched. It has a number of rows as children, starting with the header row. Here's what the parse tree looks like visually (obtained with the -ps file.ps option):



CSV is nice because of its simplicity, but it breaks down when we need a single field to hold multiple values. For that, we need a data format that allows nested elements.

6.2 Parsing JSON

JSON is a text data format that captures a collection of name-value pairs, and since values can themselves be collections, JSON can include nested structures. Designing a parser for JSON gives us an opportunity to derive a grammar from a language reference manual[4] and to work with some more complicated lexical rules. To make things more concrete, here's a simple JSON data file:

4. http://json.org

examples/t.json
```
{
        "antlr.org": {
                "owners" : [],
                "live" : true,
                "speed" : 1e100,
                "menus" : ["File", "Help\nMenu"]
        }
}
```

Our goal is to build an ANTLR grammar by reading the JSON reference manual and looking at its syntax diagram and existing grammar. We'll pull out key phrases from the manual and figure out how to encode them as ANTLR rules, starting with the grammatical structures.

JSON Grammatical Rules

The language reference says that a JSON file can be either an object, as shown earlier, or an array of values. Grammatically, that's just a choice pattern, which we can specify formally with this rule:

examples/JSON.g4
```
json:   object
    |   array
    ;
```

The next step is to drill down into the rule references in `json`. For objects, the reference says the following:

> An *object* is an unordered set of name-value pairs. An object begins with a left brace ({) and ends with a right brace (}). Each name is followed by a colon (:), and the name-value pairs are separated by a comma (,).

The syntax diagram at the JSON website also indicates that names have to be strings.

To convert this English description into grammar constructs, we break it apart looking for key phrases that represent one of our patterns: sequence, choice, token dependency, and nested phrase. The start of the sentence "An *object* is" clearly indicates we should create a rule called `object`. Next, an "unordered set of name-value pairs" is really just a sequence of pairs. The "unordered set" is referring to the semantics, or meaning, of the names; specifically, the order of the names has no meaning. That means we can just match any old list of pairs since we are just parsing.

The second sentence introduces a token dependency because an object starts and ends with curly braces. The final sentence refines our sequence of pairs

to be a sequence with a comma separator. Altogether, we get something like this in ANTLR notation:

examples/JSON.g4

```
object
    :   '{' pair (',' pair)* '}'
    |   '{' '}' // empty object
    ;
pair:   STRING ':' value ;
```

For clarity and to reduce code duplication, it's a good idea to break out the name-value pairs into their own rule. Otherwise, the first alternative of object would look like this:

```
object : '{' STRING ':' value (',' STRING ':' value)* '}' | ... ;
```

Notice that we have STRING as a token and not a grammatical rule. It's almost certainly the case that an application reading JSON files would want to treat strings as complete entities, rather than character sequences. Per our rules of thumb from Section 5.6, *Drawing the Line Between Lexer and Parser*, on page 79, strings should be tokens.

The JSON reference also has some informal grammatical rules, so let's see how our ANTLR rules compare. Here's the grammar verbatim from the reference:

```
object
    {}
    { members }

members
    pair
    pair , members

pair
    string : value
```

The reference has also broken out the pair rule, but there's a members rule that we don't have. As described in the sidebar *Loops vs. Tail Recursion*, on page 89, it's how a grammar expresses sequences without (...)* loops.

Turning to arrays, the other high-level construct, the reference manual says this:

> An *array* is an ordered collection of values. An array begins with a left bracket ([) and ends with a right bracket (]). Values are separated by a comma (,).

Like rule object, array has a comma-separated sequence and a token dependency between the left and right square brackets.

Loops vs. Tail Recursion

Rule members from the JSON reference manual looks strange because there is nothing in the English description that seems to fit a choice of pair or pair followed by a comma and reference to itself.

```
members
    pair
    pair , members
```

The difference lies in that ANTLR supports Extended BNF (EBNF) grammars and the informal rules from the JSON reference manual use straight BNF. BNF does not support subrules like our (...)* loop so they simulate looping with tail recursion (a rule invocation to itself as the last element in an alternative).

To see the relationship between the English sequence description and this tail recursive rule, here's how members derives one, two, and three pairs:

```
members => pair

members => pair , members
        => pair , pair

members => pair , members
        => pair , pair , members
        => pair , pair , pair
```

This reinforces the warning in Section 5.2, *Using Existing Grammars as a Guide*, on page 60 that existing grammars should be used as a guide, not the gospel truth.

examples/JSON.g4

```
array
    :   '[' value (',' value)* ']'
    |   '[' ']' // empty array
    ;
```

Stepping down a level of granularity, we get to value, which is a choice pattern according to the reference.

> A *value* can be a *string* in double quotes or a *number* or true or false or null or an *object* or an *array*. These structures can be nested.

The term *nested* naturally indicates a nested phrase pattern for which we should expect some recursive rule references. In ANTLR notation, value looks like Figure 4, *value in ANTLR notation*, on page 90.

By referencing object or array, rule value becomes (indirectly) recursive. By invoking either rule from value, we would eventually get back to rule value.

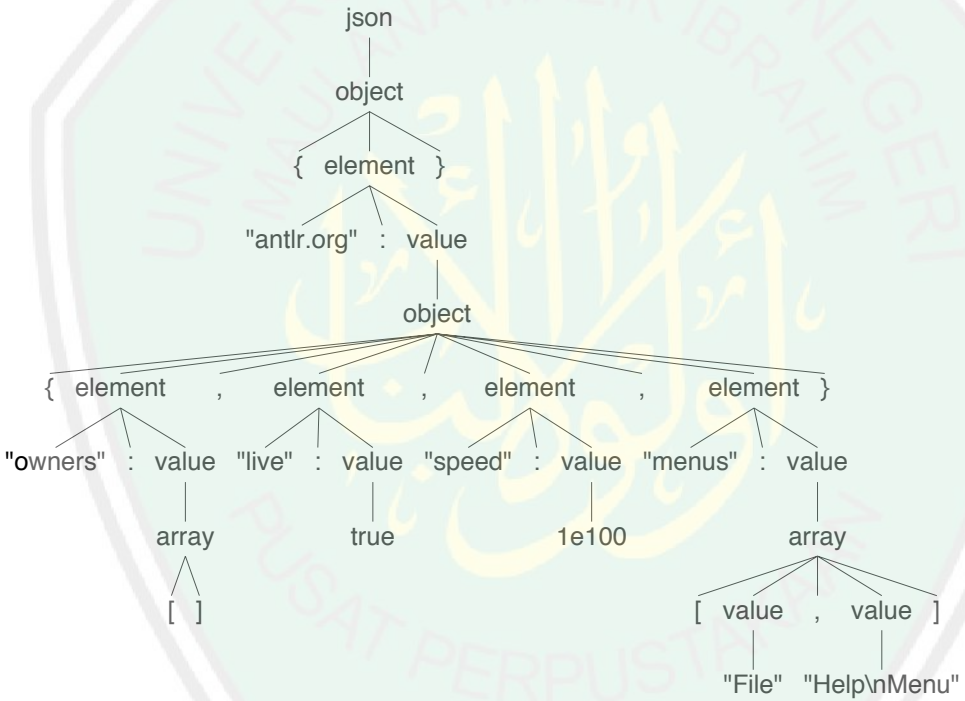
examples/JSON.g4

```
value
    :   STRING
    |   NUMBER
    |   object  // recursion
    |   array   // recursion
    |   'true'  // keywords
    |   'false'
    |   'null'
    ;
```

Figure 4—value in ANTLR notation

Rule value directly references string literals to match the JSON keywords. We also treat numbers as tokens for the same reason as strings: applications will treat numbers as complete entities.

That's it for the grammatical rules. We've completely specified the structure of a JSON file. Here's how our grammar parses the sample input from earlier:



Of course, we can't yet run our program to generate the image in that figure until we've completed the lexer grammar. We need rules for the two key tokens: STRING and NUMBER.

JSON Lexical Rules

According to the JSON reference, strings are defined as follows:

> A *string* is a sequence of zero or more Unicode characters, wrapped in double quotes, using backslash escapes. A character is represented as a single character string. A string is very much like a C or Java string.

As we discussed in the previous chapter, strings in most languages are pretty much the same. JSON's strings are similar to what we did in *Matching String Literals, on page 75*, with the addition of Unicode escapes. Looking at the existing JSON grammar, we can tell that the written description is incomplete. The grammar says this:

```
char
    any-Unicode-character-except-"-or-\-or-control-character
    \"
    \\
    \/
    \b
    \f
    \n
    \r
    \t
    \u four-hex-digits
```

This specifies what all of the escapes are and that we should match any Unicode character except for the double quote and backslash, which we can specify with a ~["\\] inverted character set. (Operator ~ means "not.") Our STRING definition looks like this:

examples/JSON.g4

```
STRING :  '"' (ESC | ~["\\])* '"' ;
```

The ESC rule matches either a predefined escape or a Unicode sequence.

examples/JSON.g4

```
fragment ESC :   '\\' (["\\/bfnrt] | UNICODE) ;
fragment UNICODE : 'u' HEX HEX HEX HEX ;
fragment HEX : [0-9a-fA-F] ;
```

Rather than repeat the definition of a hex digit multiple times in UNICODE, we use the HEX fragment rule as a shorthand. (Rules prefixed with fragment can be called only from other lexer rules; they are not tokens in their own right.)

The last token used by the parser is NUMBER. The JSON reference defines them as follows:

> A *number* is very much like a C or Java number, except that the octal and hexadecimal formats are not used.

The JSON reference's existing grammar has some fairly complicated rules for numbers, but we can pack it all into three main alternatives.

examples/JSON.g4

```
NUMBER
    :   '-'? INT '.' INT EXP?   // 1.35, 1.35E-9, 0.3, -4.5
    |   '-'? INT EXP            // 1e10 -3e4
    |   '-'? INT                // -3, 45
    ;
fragment INT :   '0' | [1-9] [0-9]* ; // no leading zeros
fragment EXP :   [Ee] [+\-]? INT ; // \- since - means "range" inside [...]
```

Again, using fragment rules INT and EXP reduces duplication and makes the grammar easier to read.

We know that INT should not match integers beginning with digit 0 from the informal JSON grammar.

```
int
    digit
    digit1-9 digits
    - digit
    - digit1-9 digits
```

We deal with the - negation operator in NUMBER so we can focus on just the first two choices: digit and digit1-9 digits. The first choice matches any single digit, so 0 is cool all by itself. The second choice starts with digit1-9, which is any digit but 0.

Unlike the CSV example in the previous section, JSON has to worry about whitespace.

> Whitespace can be inserted between any pair of tokens.

That is the typical meaning of whitespace, so we can reuse a rule from the lexer "starter kit" found at the end of the previous chapter.

examples/JSON.g4

```
WS  :   [ \t\n\r]+ -> skip ;
```

Now that we have a complete set of grammatical and lexical rules, we can try it. Let's start by printing out the tokens from sample input [1,"\u0049",1.3e9].

```
⇛ $ antlr4 JSON.g4
⇛ $ javac JSON*.java
⇛ $ grun JSON json -tokens
⇛ [1,"\u0049",1.3e9]
⇛ EOF
❮ [@0,0:0='[',<5>,1:0]
  [@1,1:1='1',<11>,1:1]
  [@2,2:2=',',<4>,1:2]
```

```
[@3,3:10='"\u0049"',<10>,1:3]
[@4,11:11=',',<4>,1:11]
[@5,12:16='1.3e9',<11>,1:12]
[@6,17:17=']',<1>,1:17]
[@7,19:18='<EOF>',<-1>,2:0]
```

Our lexer correctly breaks up the input stream into tokens, so let's try the grammar rules.

```
⇒ $ grun JSON json -tree
⇒ [1,"\u0049",1.3e9]
⇒ EOF
❮ (json (array [ (value 1) , (value "\u0049") , (value 1.3e9) ]))
```

The grammar properly interprets the token stream as an array of three values, so everything looks great. For a more complicated grammar, we'd want to try lots of other input files to verify correctness.

At this point, we've seen grammars for two data languages (CSV and JSON), so let's move on to a declarative language called DOT that ratchets up the grammatical complexity and introduces us to a new lexical pattern: case-insensitive keywords.

6.3 Parsing DOT

DOT[5] is a declarative language for describing graphs such as network diagrams, trees, or state machines. (DOT is a declarative language because we say what the graph connections are, not how to build the graph.) It's a generically useful graphing tool, particularly if you have a program that needs to generate images. For example, ANTLR's `-atn` option uses DOT to generate state machine visualizations.

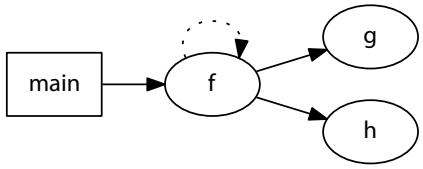
To get a feel for the language, imagine that we want to visualize a call tree graph for a program with four functions. We could draw this by hand, or we could specify the relationships with DOT as follows (either by hand or automatically by building a language application that computed the relationships from a program source):

examples/t.dot

```
digraph G {
    rankdir=LR;
    main [shape=box];
    main -> f -> g;         // main calls f which calls g
    f -> f [style=dotted] ; // f is recursive
    f -> h;                 // f calls h
}
```

5. http://www.graphviz.org/Documentation/dotguide.pdf

Here's the resulting diagram created using the DOT visualizer, graphviz:[6]



We're in a bit of luck because the DOT reference guide has syntax rules that we can reuse almost verbatim, if we convert them to ANTLR syntax. Unfortunately, we're on our own for the lexical rules. We'll have to read through the documentation and some examples to figure out the exact rules. Our easiest path is to start with the grammatical rules.

DOT Grammatical Rules

Here's the core grammar in ANTLR notation translated from the primary language reference:[7]

examples/DOT.g4

```
graph       :   STRICT? (GRAPH | DIGRAPH) id? '{' stmt_list '}' ;
stmt_list   :   ( stmt ';'? )* ;
stmt        :   node_stmt
            |   edge_stmt
            |   attr_stmt
            |   id '=' id
            |   subgraph
            ;
attr_stmt   :   (GRAPH | NODE | EDGE) attr_list ;
attr_list   :   ('[' a_list? ']')+ ;
a_list      :   (id ('=' id)? ','?)+ ;
edge_stmt   :   (node_id | subgraph) edgeRHS attr_list? ;
edgeRHS     :   ( edgeop (node_id | subgraph) )+ ;
edgeop      :   '->' | '--' ;
node_stmt   :   node_id attr_list? ;
node_id     :   id port? ;
port        :   ':' id (':' id)? ;
subgraph    :   (SUBGRAPH id?)? '{' stmt_list '}' ;
id          :   ID
            |   STRING
            |   HTML_STRING
            |   NUMBER
            ;
```

The only deviation we make from the reference grammar is rule port. The reference provides this.

6. http://www.graphviz.org
7. http://www.graphviz.org/pub/scm/graphviz2/doc/info/lang.html

```
port:   ':' ID [ ':' compass_pt ]
    |   ':' compass_pt
compass_pt
    :   (n | ne | e | se | s | sw | w | nw)
```

These rules would normally be fine if the compass points were keywords and not legal as identifiers. But, the text in the reference alters the meaning of the grammar.

> Note also that the allowed compass point values are not keywords, so these strings can be used elsewhere as ordinary identifiers....

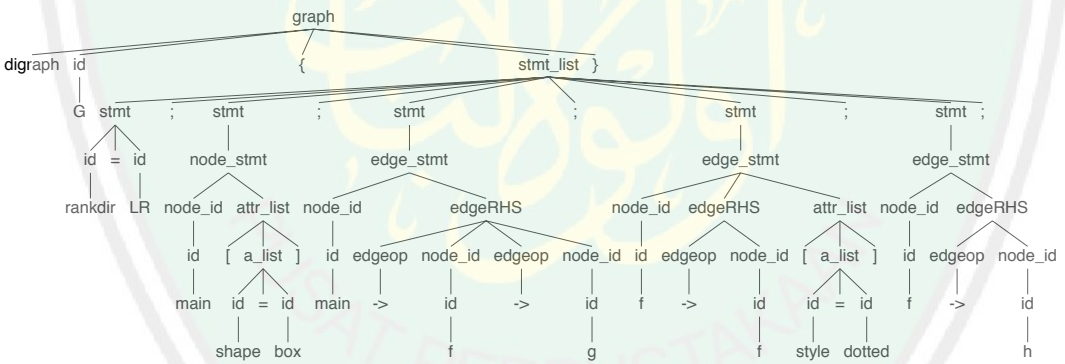
That means we have to accept edge statements such as `n -> sw;` where `n` and `sw` are identifiers, not compass points. The reference goes on to say this: "...conversely, the parser will actually accept any identifier." That's not entirely clear, but it sounds like the parser accepts any identifier for a compass point. If that's true, we don't have to worry about compass points at all in the grammar; we can replace references to rule `compass_pt` with `id`.

```
port:   ':' id (':' id)? ;
```

Just to be sure, it never hurts to try suppositions with a DOT viewer like those found on the Graphviz website.[8] DOT does, in fact, accept the following graph definition, so our `port` rule is OK:

```
digraph G { n -> sw; }
```

At this point, we have our grammatical rules in place. Pretending that our token definitions are finished, let's look at the parse tree for sample input `t.dot` (using `grun DOT graph -gui t.dot`).



Now let's try to define those tokens.

8. http://www.graphviz.org

DOT Lexical Rules

Since the reference guide does not provide formal lexical rules, we have to derive them from the written description. The keywords are the simplest, so let's start with those.

The reference says that "the keywords node, edge, graph, digraph, subgraph, and strict are case-independent." If they weren't case insensitive, we could simply use literals in the grammar like 'node'. To allow variations such as nOdE, we need to lay out lexical rules with uppercase and lowercase variations at each character position.

examples/DOT.g4

```
STRICT      :   [Ss][Tt][Rr][Ii][Cc][Tt] ;
GRAPH       :   [Gg][Rr][Aa][Pp][Hh] ;
DIGRAPH     :   [Dd][Ii][Gg][Rr][Aa][Pp][Hh] ;
NODE        :   [Nn][Oo][Dd][Ee] ;
EDGE        :   [Ee][Dd][Gg][Ee] ;
SUBGRAPH    :   [Ss][Uu][Bb][Gg][Rr][Aa][Pp][Hh] ;
```

Identifiers are similar to most programming languages.

> Any string of alphabetic ([a-zA-Z\200-\377]) characters, underscores ('_'), or digits ([0-9]), not beginning with a digit.

The \200-\377 octal range is 80 through FF in hex, so our ID rule looks like this:

examples/DOT.g4

```
ID          :   LETTER (LETTER|DIGIT)*;
fragment
LETTER      :   [a-zA-Z\u0080-\u00FF_] ;
```

Helper rule DIGIT is one of the lexical rules we need to match numbers. The reference guide says numbers follow this regular expression:

> [-]?(.[0-9]+ | [0-9]+(.[0-9]*)?)

Replacing [0-9] with DIGIT, DOT numbers in ANTLR notation look like this:

examples/DOT.g4

```
NUMBER      :   '-'? ('.' DIGIT+ | DIGIT+ ('.' DIGIT*)? ) ;
fragment
DIGIT       :   [0-9] ;
```

DOT strings are pretty basic.

> any double-quoted string ("...") possibly containing escaped quotes (\")

To match anything inside the string, we use the dot wildcard operator to consume characters until it sees the final double quote. We can also match the escaped double quote as an alternative of the subrule loop.

examples/DOT.g4
```
STRING      :   '"' ('\\"'|.)*? '"' ;
```

DOT also has what it calls an HTML string, which is, as far as I can tell, exactly like a string except it uses angle brackets instead of double quotes. The reference uses notation <...> and goes on to say this:

> ... in HTML strings, angle brackets must occur in matched pairs, and unescaped newlines are allowed. In addition, the content must be legal XML, so that the special XML escape sequences for ", &, <, and > may be necessary in order to embed these characters in attribute values or raw text.

That description tells us most of what we need but doesn't answer whether we can have > inside an HTML comment. Also, it seems to imply we should wrap sequences of tags in angle brackets like this: <<i>hi</i>>. From experimentation with a DOT viewer, that is the case. DOT seems to accept anything between angle brackets as long as the brackets match. So, > inside HTML comments are not ignored like they would be by an XML parser. HTML string <foo<!--ksjdf > -->> gets treated like string "foo<!--ksjdf > --".

To accept anything in angle brackets, we can use ANTLR construct '<' .*? '>'. But that doesn't allow for angle brackets nested inside because it will associate the first > with the first < rather than the most recent <. The following rules do the trick:

examples/DOT.g4
```
/** "HTML strings, angle brackets must occur in matched pairs, and
 *  unescaped newlines are allowed."
 */
HTML_STRING :   '<' (TAG|~[<>])* '>' ;
fragment
TAG         :   '<' .*? '>' ;
```

The HTML_STRING rule allows a TAG within a pair of angle brackets, implementing the single level of nesting. The ~[<>] set takes care of matching XML character entities such as <. It matches anything other than a left or right angle bracket. We can't use wildcard and a nongreedy loop here. (TAG|.)*? would match invalid input such as <<foo> because the wildcard inside the loop can match <foo. HTML_STRING in that case wouldn't have to call TAG to match a tag or portion of a tag.

You might be tempted to use recursion to match up the angle brackets like this:

```
HTML_STRING : '<' (HTML_STRING|~[<>])* '>' ;
```

But, that matches nested tags instead of just balancing the start and stop angle brackets. A nested tag would look like <<i
>>, which we don't want.

DOT has one last lexical structure we haven't seen before. DOT matches and discards lines starting with # because it considers them C preprocessor output. We can treat them just like the single-line comment rules we've seen.

examples/DOT.g4

```
PREPROC     :   '#' .*? '\n' -> skip ;
```

That's it for the DOT grammar (except for rules we're very familiar with). We've made it through our first moderately complex language! Aside from the more complex grammatical and lexical structures, this section emphasizes that we often have to look at multiple sources to unmask an entire language. The larger a language is, the more we need multiple references and multiple representative input samples. Sometimes poking and prodding an existing implementation is the only way to ferret out the edge cases. No language reference is ever perfectly comprehensive.

We also have to decide what is properly part of the parsing process and what should be processed later as a separate phase. For example, we treat the special port names, such as `ne` and `sw`, as simple identifiers in the parser. We also don't interpret the HTML inside <...> strings. A full DOT implementation would have to verify and process these elements at some point, but the parser gets to treat them as chunks.

Now it's time to tackle some programming languages. In the next section, we'll build a grammar for a traditional imperative programming language that looks like C. After that, we'll take on our biggest challenge yet with the R functional programming language.

6.4 Parsing Cymbol

To demonstrate how to parse a programming language with syntax derived from C, we're going to build a grammar for a language I conjured up called Cymbol. Cymbol is a simple non-object-oriented programming language that looks like C without `structs`. A grammar for this language serves as a good prototype for other new programming languages, if you'd like to build one. We won't see any new ANTLR syntax, but our grammar will demonstrate how to build a simple left-recursive expression rule.

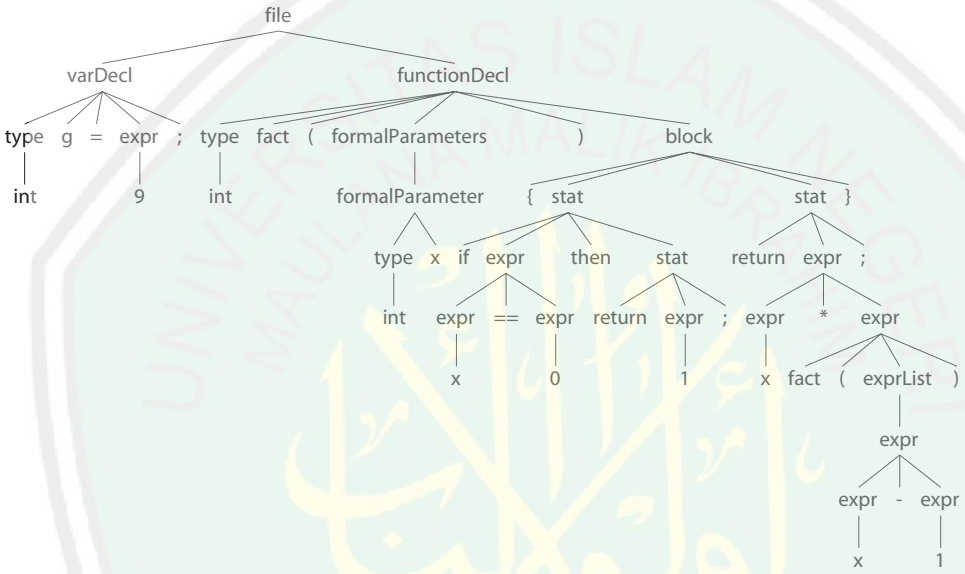
When designing a language, there's no formal grammar or language reference manual to work from. Instead, we start by conjuring up representative samples of the language. From there, we derive a grammar as we did in Section 5.1, *Deriving Grammars from Language Samples*, on page 58. (This is also how

we'd deal with an existing language for which we had no formal grammar or language reference.) Here's a program with a global variable and recursive function declaration that shows what Cymbol code looks like:

examples/t.cymbol

```
// Cymbol test
int g = 9;        // a global variable
int fact(int x) { // factorial function
    if x==0 then return 1;
    return x * fact(x-1);
}
```

Just like on a cooking show, let's see what the final product looks like so we have a target in mind. Here's the parse tree that illustrates how our grammar should interpret the input (via grun Cymbol file -gui t.cymbol):



Looking at that Cymbol program at the coarsest level, we see a sequence of global variable and function declarations.

examples/Cymbol.g4

```
file:   (functionDecl | varDecl)+ ;
```

Variable declarations look like they do in all C derivatives, with a type followed by an identifier optionally followed by an initialization expression.

examples/Cymbol.g4

```
varDecl
    :   type ID ('=' expr)? ';'
    ;
type:   'float' | 'int' | 'void' ; // user-defined types
```

Functions are basically the same: a type followed by the function name followed by a parenthesized argument list followed by a function body.

examples/Cymbol.g4
```
functionDecl
    :   type ID '(' formalParameters? ')' block // "void f(int x) {...}"
    ;
formalParameters
    :   formalParameter (',' formalParameter)*
    ;
formalParameter
    :   type ID
    ;
```

A function body is a block of statements surrounded by curly braces. Let's make six kinds of statements: nested block, variable declaration, if statement, return statement, assignment, and function call. We can encode that in ANTLR syntax as follows:

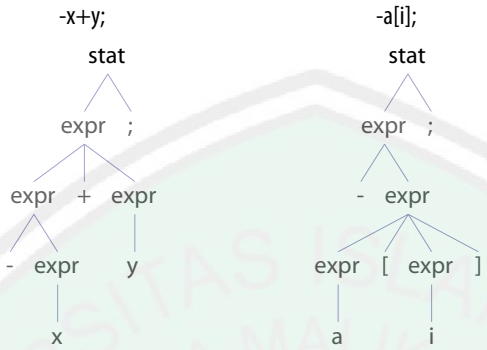
examples/Cymbol.g4
```
block:  '{' stat* '}' ;   // possibly empty statement block
stat:   block
    |   varDecl
    |   'if' expr 'then' stat ('else' stat)?
    |   'return' expr? ';'
    |   expr '=' expr ';' // assignment
    |   expr ';'          // func call
    ;
```

The last major chunk of the language is the expression syntax. Because Cymbol is really just a prototype or stepping-stone for building other programming languages, it's OK to avoid a big list of operators. Let's say we have unary negate, Boolean not, multiplication, addition, subtraction, function calls, array indexing, equality comparison, variables, integers, and parenthesized expressions.

examples/Cymbol.g4
```
expr:   ID '(' exprList? ')'    // func call like f(), f(x), f(1,2)
    |   expr '[' expr ']'       // array index like a[i], a[i][j]
    |   '-' expr                // unary minus
    |   '!' expr                // boolean not
    |   expr '*' expr
    |   expr ('+'|'-') expr
    |   expr '==' expr          // equality comparison (lowest priority op)
    |   ID                      // variable reference
    |   INT
    |   '(' expr ')'
    ;
exprList : expr (',' expr)* ;   // arg list
```

The most important lesson here is that we generally have to list the alternatives in order from highest to lowest precedence. (For an in-depth discussion of how ANTLR removes left recursion and handles operator precedence, see Chapter 14, *Removing Direct Left Recursion*, on page 247.)

To see the precedence in action, take a look at the parse trees for input `-x+y;` and `-a[i];` starting at rule `stat` (instead of `file` to reduce clutter).



The parse tree on the left shows the unary minus binding most tightly to the `x` since it has higher priority than addition. The unary minus alternative appears before the addition alternative. On the other hand, unary minus has lower precedence than the array index suffix operator because the alternative with unary minus appears after the array index alternative. The parse tree on the right illustrates that the negation is applied to the `a[i]`, not just identifier `a`. We'll see a more complicated expression rule in the next section.

We don't have to move on to the lexical rules like we usually do. Examining them wouldn't introduce anything new and interesting. The rules are pulled almost verbatim from the lexical patterns in the previous chapter. Our focus here is really on exploring the grammatical structure of an imperative programming language.

Using our intuitive sense of what a `struct`-less or `class`-less Java language might look like made building the Cymbol grammar pretty easy. And, if you've wrapped your head around this grammar, you're ready to build grammars for your own moderately complex imperative programming languages.

Next, we're going to tackle a language at the other extreme. To get a decent R grammar together, we'll have to deduce precise language structure by trudging through multiple references, examining sample programs, and testing the existing R implementation.

6.5 Parsing R

R is an expressive domain-specific programming language for describing statistical problems. For example, it's easy to create vectors, apply functions to them, and filter them (shown here using the R interactive shell).

```
⇛ x <- seq(1,10,.5)     # x = 1, 1.5, 2, 2.5, 3, 3.5, ..., 10
⇛ y <- 1:5              # y = 1, 2, 3, 4, 5
⇛ z <- c(9,6,2,10,-4)   # z = 9, 6, 2, 10, -4
⇛ y + z                 # add two vectors
❮ [1] 10  8  5 14  1    # result is 1-dimensional vector
⇛ z[z<5]                # all elements in z < 5
❮ [1]  2 -4
⇛ mean(z)               # compute the mean of vector z
❮ [1] 4.6
⇛ zero <- function() { return(0) }
⇛ zero()
❮ [1] 0
```

R is a medium-sized but complicated programming language, and most or all of us have a handicap: we don't know R. That means we can't just write down the grammar from our internal sense of the language structure like we could for Cymbol in the previous section. We must deduce R's structure by wading through reference manuals, examples, and a formal yacc grammar[9] from the existing implementation.

To get started, it's a good idea to skim through some language overviews.[10,11] We should also look at R examples to get a feel for the language and then pick some files to serve as an "acceptance test." Ajay Shah has built a fine set of examples[12] we can use. Covering those examples would mean that our grammar handles a large percentage of R code. (Getting a perfect grammar without knowing the language intimately is unlikely.) To help us in our quest to build an R grammar, there's a lot of documentation available at the main website,[13] but let's focus on R-intro[14] and language definition R-lang.[15]

Construction of our grammar begins at the coarsest level, as usual. From the language overviews, it's clear that R programs are a series of expressions or assignments. Even function definitions are assignments; we assign a function

9. http://svn.r-project.org/R/trunk/src/main/gram.y
10. http://www.stat.wisc.edu/~deepayan/SIBS2005/slides/language-overview.pdf
11. http://www.stat.lsa.umich.edu/~kshedden/Courses/Stat600/Notes/R_introduction.pdf
12. http://www.mayin.org/ajayshah/KB/R/index.html
13. http://www.r-project.org/
14. http://cran.r-project.org/doc/manuals/R-intro.pdf
15. http://cran.r-project.org/doc/manuals/R-lang.html

literal to a variable. The only unfamiliar thing is that there are three assignment operators: <-, =, and <<-. For our purposes, we don't have to care about their meaning because we're building only a parser, not an interpreter or compiler. Our first whack at the program structure looks something like this:

```
prog : (expr_or_assign '\n')* EOF ;

expr_or_assign
    :   expr ('<-' | '=' | '<<-' ) expr_or_assign
    |   expr
    ;
```

After reading some examples, though, it looks like we can also put more than one expression on a line by separating them with semicolons. R-intro confirms this. Also, while not in the manuals, the R shell allows and ignores blank lines. Tweaking our rules accordingly leads us to the beginning of our grammar.

examples/R.g4

```
prog:   (   expr_or_assign (';'|NL)
        |   NL
        )*
        EOF
    ;

expr_or_assign
    :   expr ('<-'|'='|'<<-') expr_or_assign
    |   expr
    ;
```

We use token NL rather than literal '\n' because we should allow Windows (\r\n) and Unix newlines (\n), which is easier to define as a lexical rule.

examples/R.g4

```
// Match both UNIX and Windows newlines
NL      :   '\r'? '\n' ;
```

Notice that NL doesn't say to discard those tokens as is customary. The parser uses them as expression terminators, like semicolons in Java, so the lexer must pass them to the parser.

The majority of R syntax relates to expressions, so that's what we'll focus on for the rest of the section. There are three main kinds: statement expressions, operator expressions, and function-related expressions. Since R statements are very similar to their imperative language counterparts, let's start with those to get them out of the way. Here are the alternatives dealing with statements from rule expr (which physically appear after the operator alternatives in expr):

examples/R.g4
```
|   '{' exprlist '}' // compound statement
|   'if' '(' expr ')' expr
|   'if' '(' expr ')' expr 'else' expr
|   'for' '(' ID 'in' expr ')' expr
|   'while' '(' expr ')' expr
|   'repeat' expr
|   '?' expr // get help on expr, usually string or ID
|   'next'
|   'break'
```

The first alternative matches the group of expressions per R-intro: "Elementary commands can be grouped together into one compound expression by braces ('{' and '}')." Here is exprlist:

examples/R.g4
```
exprlist
    :   expr_or_assign ((';'|NL) expr_or_assign?)*
    |
    ;
```

Most of the R expression language handles the plentiful operators. To get their correct forms, our best bet is to rely on the yacc grammar. Executable code is often, but not always, the best guide to a language author's intentions. To get the precedence, we need to look at the precedence table, which explicitly lays out the relative operator precedence. For example, here is what the yacc grammar says for the arithmetic operators (%left precedence commands listed first have lower priority):

```
%left           '+' '-'
%left           '*' '/'
```

The R-lang document also has a section called "Infix and prefix operators" that gives the operator precedence rules, but it seems to be missing the ::: operator found in the yacc grammar. Putting it all together, we can use the following alternatives for the binary, prefix, and suffix operators:

examples/R.g4
```
expr:   expr '[[' sublist ']' ']'  // '[[' follows R's yacc grammar
    |   expr '[' sublist ']'
    |   expr ('::'|':::') expr
    |   expr ('$'|'@') expr
    |   expr '^'<assoc=right> expr
    |   ('-'|'+') expr
    |   expr ':' expr
    |   expr USER_OP expr // anything wrappedin %: '%' .* '%'
    |   expr ('*'|'/') expr
    |   expr ('+'|'-') expr
    |   expr ('>'|'>='|'<'|'<='|'=='|'!=') expr
```

```
|   '!' expr
|   expr ('&'|'&&') expr
|   expr ('|'|'||') expr
|   '~' expr
|   expr '~' expr
|   expr ('->'|'->>'|':=') expr
```

We don't have to care about what the operators mean since we care only about recognition in this example. We just have to ensure that our grammar matches the precedence and associativity correctly.

The one unusual feature of the expr rule is the use of '[[' instead of '[' '[' in alternative expr '[[' sublist ']' ']'. ([[...]] selects a single element, whereas [...] yields a sublist.) I took '[[' directly from R's yacc grammar. This probably enforces a "no space between two left brackets" rule, but there was no obvious specification of this in the reference manual.

The ^ operator has token suffix <assoc=right> because R-lang indicates the following:

> The exponentiation operator '^' and the left assignment operators '<- = <<-' group right to left, all other operators group left to right. That is, 2 ^ 2 ^ 3 is 2 ^ 8, not 4 ^ 3.

With the statement and operator expressions out of the way, let's look at the last major chunk of the expr rule: defining and calling functions. We can use the following two alternatives:

examples/R.g4

```
|   'function' '(' formlist? ')' expr // define function
|   expr '(' sublist ')'             // call function
```

Rules formlist and sublist define the formal argument definition lists and call site argument expressions, respectively. The rule names mirror what the yacc grammar uses to make it easier to compare the two grammars.

The formal function arguments expressed by formlist follow the specification in R-lang.

> ... a comma-separated list of items each of which can be an identifier, or of the form '*identifier* = *default*', or the special token '...'. The *default* can be any valid expression.

We can encode that using an ANTLR rule that is similar to formlist in the yacc grammar (see Figure 5, *ANTLR rule for a formlist-like rule*, on page 106).

Now, to call a function instead of defining one, R-lang describes the argument syntax as shown in Figure 6, *R-lang argument syntax*, on page 106.

examples/R.g4
```
formlist : form (',' form)* ;

form:   ID
    |   ID '=' expr
    |   '...'
    ;
```

Figure 5—ANTLR rule for a formlist-like rule

> Each argument can be tagged (tag=expr), or just be a simple expression. It can also be empty or it can be one of the special tokens '...', '..2', etc.

Figure 6—R-lang argument syntax to call a function

A peek at the yacc grammar tightens this up a little bit for us; it indicates we can also have things like "n"=0, n=1, and NULL = 2. Combining specifications, we arrive at the following rules for function call arguments:

examples/R.g4
```
sublist : sub (',' sub)* ;
sub :   expr
    |   ID '='
    |   ID '=' expr
    |   STRING '='
    |   STRING '=' expr
    |   'NULL' '='
    |   'NULL' '=' expr
    |   '...'
    |
    ;
```

You might be wondering where in rule sub we match special tokens like ..2. It turns out we don't have to explicitly match them because our lexer can treat them as identifiers. According to R-lang:

> Identifiers consist of a sequence of letters, digits, the period ('.'), and the underscore. They must not start with a digit nor underscore, nor with a period followed by a digit. ... Notice that identifiers starting with a period [like] '...' and '..1', '..2', etc., are special.

To encode all of that, we use the following identifier rule:

examples/R.g4
```
ID  :   '.' (LETTER|'_'|'.') (LETTER|DIGIT|'_'|'.')*
    |   LETTER (LETTER|DIGIT|'_'|'.')*
    ;
fragment LETTER  : [a-zA-Z] ;
```

The first alternative separates out the first case, where an identifier starts with a period. We have to make sure that a digit is not the next character. We can ensure that with subrule (LETTER|'_'|'.'). To ensure that an identifier does not start with a digit or underscore, we start the second alternative with a reference to help rule LETTER. To match ..2, we use the first alternative. The initial '.' reference matches the first dot, the (LETTER|'_'|'.') subrule matches the second dot, and the last subrule matches digit 2.

The remainder of the lexer rules are direct copies or small extensions of rules we've seen before, so we can leave them out of our discussion here.

Let's take a look our handiwork now by using grun on the following input:

examples/t.R
```
addMe <- function(x,y) { return(x+y) }
addMe(x=1,2)
r <- 1:5
```

Here's how to build and bring up the parse tree visually (Figure 7, *Parse tree for input t.R*, on page 108) for input t.R:

```
$ antlr4 R.g4
$ javac R*.java
$ grun R prog -gui t.R
```

Our R grammar works well as long as each expression fits on a line, such as function addMe(). Unfortunately, that assumption is too restrictive because R allows functions and other expressions to span multiple lines. Nonetheless, we'll wrap up here because we've covered the R grammatical structure itself. In source directory code/extras, you'll find a solution to the persnickety problem of ignoring newlines within expressions; see R.g4, RFilter.g4, and TestR.java. It filters tokens from the lexer to keep or toss out newlines appropriately, according to syntax.

Our goal in this chapter was to solidify our knowledge of ANTLR syntax and to learn how to derive grammars from language reference manuals, examples, and existing non-ANTLR grammars. To that end, we looked at two data languages (CSV, JSON), a declarative language (DOT), an imperative language (Cymbol), and a functional language (R). These examples cover all of the skills you'll need to build grammars for most moderately complex languages. Before you move on, though, it's a good idea to lock in your new expertise by downloading the grammars and trying some simple modifications to alter the languages. For example, you might try adding more operators and statements to the Cymbol grammar. Use the TestRig to see the relationship between your altered grammars and sample inputs.

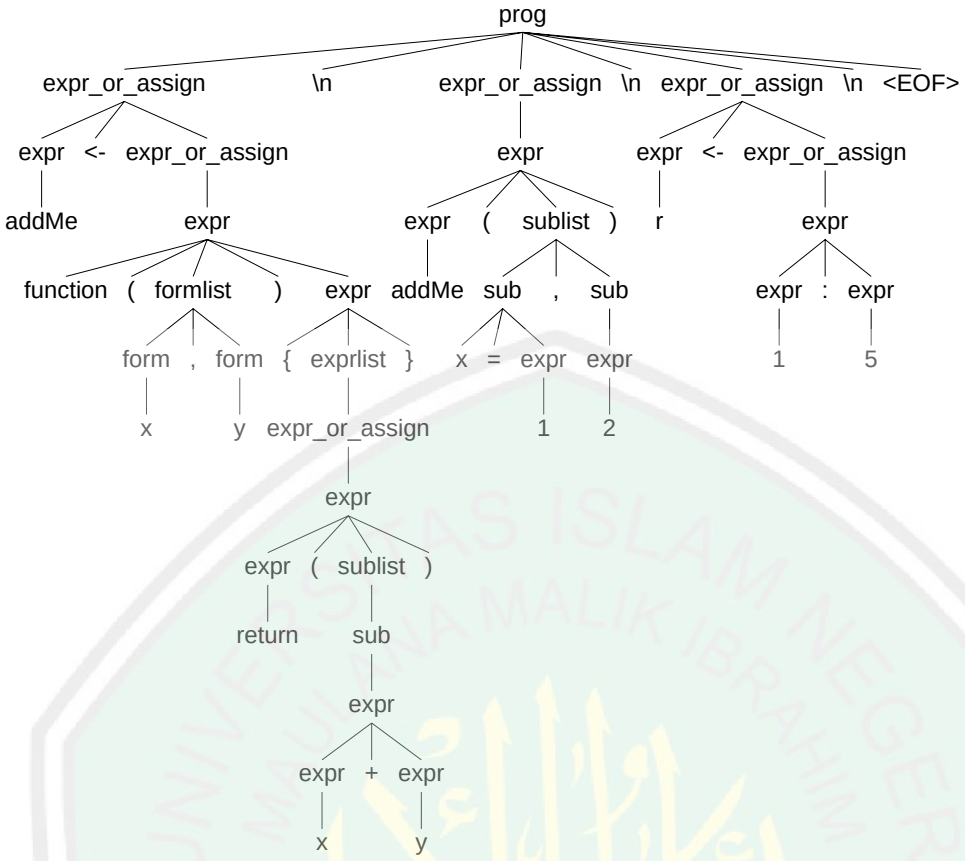


Figure 7—Parse tree for input t.R

So far in this book, we have focused on language recognition. But, grammars by themselves can indicate only whether the input conforms to the language. Now that we are parsing wizards, we're ready to learn about attaching application-specific code to the parsing mechanism, which we'll do in the next chapter. Following that, we'll build some real language applications.

CHAPTER 7

Decoupling Grammars from Application-Specific Code

Now that we know how to formally define languages using ANTLR grammar syntax, it's time to breathe some life into our grammars. By itself, a grammar isn't that useful because the associated parser tells us only whether an input sentence conforms to a language specification. To build language applications, we need the parser to trigger specific actions when it sees specific input sentences, phrases, or tokens. The collection of phrase → action pairs represents our language application or at least the interface between the grammar and a larger surrounding application.

In this chapter, we're going to learn how to use parse-tree *listeners* and *visitors* to build language applications. A listener is an object that responds to rule entry and exit events (phrase recognition events) triggered by a parse-tree walker as it discovers and finishes nodes. To support situations where an application must control how a tree is walked, ANTLR-generated parse trees also support the well-known tree visitor pattern.

The biggest difference between listeners and visitors is that listener methods aren't responsible for explicitly calling methods to walk their children. Visitors, on the other hand, must explicitly trigger visits to child nodes to keep the tree traversal going (as we saw in Section 2.5, *Parse-Tree Listeners and Visitors*, on page 17). Visitors get to control the order of traversal and how much of the tree gets visited because of these explicit calls to visit children. For convenience, I'll use the term *event method* to refer to either a listener callback or a visitor method.

Our goal in this chapter is to understand exactly what tree-walking facilities ANTLR builds for us and why. We'll start by looking at the origins of the

listener mechanism and how we can keep application-specific code out of our grammars using listeners and visitors. Next, we'll learn how to get ANTLR to generate more precise events, one for each alternative in a rule. Once we know a little more about ANTLR's tree walking, we'll look at three calculator implementations that illustrate different ways to pass around subexpression results. Finally, we'll discuss the advantages and disadvantages of the three approaches. At that point, we'll be ready to tackle the real examples in the next chapter.

7.1 Evolving from Embedded Actions to Listeners

If you're used to previous versions of ANTLR or other parser generators, you'll be surprised to hear that we can build language applications without embedding actions (code) in the grammars. The listener and visitor mechanisms decouple grammars from application code, providing some compelling benefits. Such decoupling nicely encapsulates an application instead of fracturing it and dispersing the pieces across a grammar. Without embedded actions, we can reuse the same grammar in different applications without even recompiling the generated parser.

ANTLR can also generate parsers in different programming languages for the same grammar if it's bereft of actions. (I anticipate supporting different target languages after the 4.0 release.) Integrating grammar bug fixes or updates is also easy because we don't have to worry about merge conflicts because of embedded actions.

In this section, we're going to investigate the evolution from grammar with embedded actions to completely decoupled grammar and application. The following property file grammar with embedded actions sketched in with *«...»* reads property files, one property assignment per line. Actions like *«start file»* are just stand-ins for appropriate Java code.

```
grammar PropertyFile;
file : {«start file»} prop+ {«finish file»} ;
prop : ID '=' STRING '\n' {«process property»} ;
ID   : [a-z]+ ;
STRING : '"' .*? '"' ;
```

Such a tight coupling ties the grammar to one specific application. A better approach is to create a subclass of PropertyFileParser, the parser generated by ANTLR, and convert the embedded actions to methods. The refactoring leaves only trivial method call actions in the grammar that trigger the newly created methods. Then, by subclassing the parser, we can implement any number of different applications without altering the grammar. One such refactoring looks like this:

```
grammar PropertyFile;
@members {
    void startFile() { }  // blank implementations
    void finishFile() { }
    void defineProperty(Token name, Token value) { }
}
file : {startFile();} prop+ {finishFile();} ;
prop : ID '=' STRING '\n' {defineProperty($ID, $STRING)} ;
ID   : [a-z]+ ;
STRING : '"' .*? '"' ;
```

This decoupling makes the grammar reusable for different applications, but the grammar is still tied to Java because of the method calls. We'll deal with that shortly.

To demonstrate the reusability of the refactored grammar, let's build two different "applications," starting with one that just prints out the properties as it encounters them. The process is simply to extend the parser class generated by ANTLR and override one or more of the methods triggered by the grammar.

```
class PropertyFilePrinter extends PropertyFileParser {
    void defineProperty(Token name, Token value) {
        System.out.println(name.getText()+"="+value.getText());
    }
}
```

Notice that we don't have to override startFile() or finishFile() because of the default implementations in the PropertyFileParser superclass generated by ANTLR.

To launch this application, we need to create an instance of our special PropertyFilePrinter parser subclass instead of the usual PropertyFileParser.

```
PropertyFileLexer lexer = new PropertyFileLexer(input);
CommonTokenStream tokens = new CommonTokenStream(lexer);
PropertyFilePrinter parser = new PropertyFilePrinter(tokens);
parser.file(); // launch our special version of the parser
```

As a second application, let's load the properties into a map instead of printing them out. All we have to do is create a new subclass and put different functionality in defineProperty().

```
class PropertyFileLoader extends PropertyFileParser {
    Map<String,String> props = new OrderedHashMap<String, String>();
    void defineProperty(Token name, Token value) {
        props.put(name.getText(), value.getText());
    }
}
```

After the parser executes, field props will contain the name-value pairs.

This grammar still has the problem that the embedded actions restrict us to generating parsers only in Java. To make the grammar reusable and language neutral, we need to completely avoid embedded actions. The next two sections show how to do that with a listener and a visitor.

7.2 Implementing Applications with Parse-Tree Listeners

To build language applications without entangling the application and the grammar, the key is to have the parser create a parse tree and then walk it to trigger application-specific code. We can walk the tree using our favorite technique, or we can use one of the tree-walking mechanisms that ANTLR generates. In this section, we're going to use ANTLR's built-in ParseTreeWalker to build a listener-based version of the property file application from the previous section.

Let's start with a denuded version of the property file grammar.

listeners/PropertyFile.g4

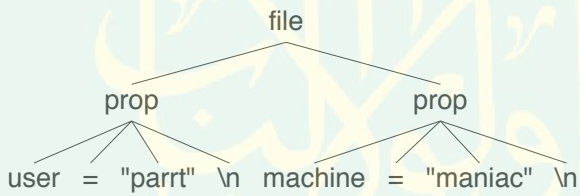
```
file : prop+ ;
prop : ID '=' STRING '\n' ;
```

Here's a sample property file:

listeners/t.properties

```
user="parrt"
machine="maniac"
```

From the grammar, ANTLR generates PropertyFileParser, which automatically builds the following parse tree:



Once we have a parse tree, we can use ParseTreeWalker to visit all of the nodes, triggering enter and exit methods.

Let's take a look at listener interface PropertyFileListener that ANTLR generates from grammar PropertyFile. ANTLR's ParseTreeWalker triggers enter and exit methods for each rule subtree as it discovers and finishes nodes, respectively. Because there are only two parser rules in grammar PropertyFile, there are four methods in the interface.

listeners/PropertyFileListener.java
```
import org.antlr.v4.runtime.tree.*;
import org.antlr.v4.runtime.Token;

public interface PropertyFileListener extends ParseTreeListener {
        void enterFile(PropertyFileParser.FileContext ctx);
        void exitFile(PropertyFileParser.FileContext ctx);
        void enterProp(PropertyFileParser.PropContext ctx);
        void exitProp(PropertyFileParser.PropContext ctx);
}
```

The FileContext and PropContext objects are implementations of parse-tree nodes specific to each grammar rule. They contain useful methods that we'll explore as we go along.

As a convenience, ANTLR also generates class PropertyFileBaseListener with default implementations that mimic the blank methods we manually wrote in the @members area of the grammar in the previous section.

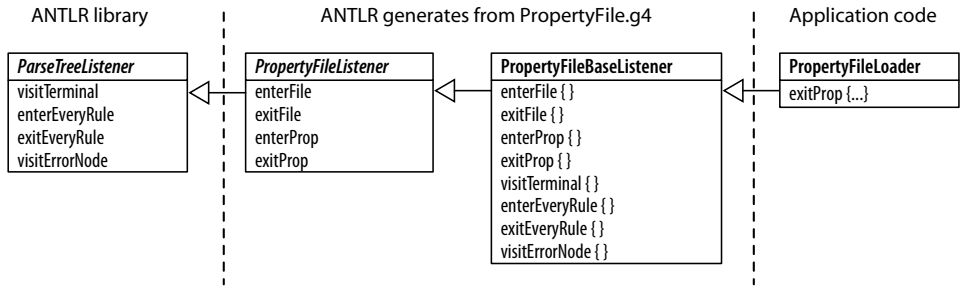
```
public class PropertyFileBaseVisitor<T> extends AbstractParseTreeVisitor<T>
    implements PropertyFileVisitor<T>
{
    @Override public T visitFile(PropertyFileParser.FileContext ctx) { }
    @Override public T visitProp(PropertyFileParser.PropContext ctx) { }
}
```

The default implementations let us override and implement only those methods we care about. For example, here's a reimplementation of the property file loader that has a single method like before, but using the listener mechanism:

listeners/TestPropertyFile.java
```
public static class PropertyFileLoader extends PropertyFileBaseListener {
    Map<String,String> props = new OrderedHashMap<String, String>();
    public void exitProp(PropertyFileParser.PropContext ctx) {
        String id = ctx.ID().getText(); // prop : ID '=' STRING '\n' ;
        String value = ctx.STRING().getText();
        props.put(id, value);
    }
}
```

The main differences are that this version extends the base listener instead of the parser and that the listener methods get triggered after the parser has completed.

There are a lot of interfaces and classes in flight here, so let's look at the inheritance relationship between the key players (interfaces in italics).



Interface ParseTreeListener is in the ANTLR runtime library and dictates that every listener respond to events visitTerminal(), enterEveryRule(), exitEveryRule(), and (upon syntax errors) visitErrorNode(). ANTLR generates interface PropertyFileListener from grammar PropertyFile and default implementations for all methods in class PropertyFileBaseListener. The only thing that we're building is the PropertyFileLoader, which inherits all of the blank functionality from PropertyFileBaseListener.

Method exitProp() has access to the rule context object, PropContext, associated with rule prop. That context object has methods for each of the elements mentioned in rule prop (ID and STRING). Because those elements are token references in the grammar, the methods return TerminalNode parse-tree nodes. We can either directly access the text of the token payload via getText(), as we've done here, or get the Token payload first via getSymbol().

And now for the exciting conclusion. Let's walk the tree, listening in with our new PropertyFileLoader.

listeners/TestPropertyFile.java
```
// create a standard ANTLR parse tree walker
ParseTreeWalker walker = new ParseTreeWalker();
// create listener then feed to walker
PropertyFileLoader loader = new PropertyFileLoader();
walker.walk(loader, tree);        // walk parse tree
System.out.println(loader.props); // print results
```

Here's a refresher on how to run ANTLR on a grammar, compile the generated code, and launch a test program to process an input file:

```
$ antlr4 PropertyFile.g4
$ ls PropertyFile*.java
PropertyFileBaseListener.java   PropertyFileListener.java
PropertyFileLexer.java          PropertyFileParser.java
$ javac TestPropertyFile.java PropertyFile*.java
$ cat t.properties
user="parrt"
machine="maniac"
$ java TestPropertyFile t.properties
{user="parrt", machine="maniac"}
```

Our test program successfully reconstitutes the property assignments from the file into a map data structure in memory.

A listener-based approach is great because all of the tree walking and method triggering is done automatically. Sometimes, though, automatic tree walking is also a weakness because we can't control the walk itself. For example, we might want to walk a parse tree for a C program, ignoring everything inside functions by skipping the function body subtrees. Listener event methods also can't use method return values to pass data around. When we need to control the walk or want to return values with event-method return values, we use a visitor pattern. Let's build a visitor-based version of this property file loader to compare the approaches.

7.3 Implementing Applications with Visitors

To use a visitor instead of a listener, we ask ANTLR to generate a visitor interface, implement that interface, and then create a test rig that calls visit() on the parse tree. We don't have to touch the grammar at all.

When we use the -visitor option on the command line, ANTLR generates interface PropertyFileVisitor and class PropertyFileBaseVisitor, which has the following default implementations:

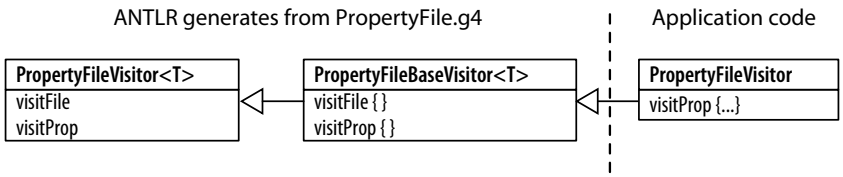
```
public class PropertyFileBaseVisitor<T> extends AbstractParseTreeVisitor<T>
    implements PropertyFileVisitor<T>
{
    @Override public T visitFile(PropertyFileParser.FileContext ctx) { ... }
    @Override public T visitProp(PropertyFileParser.PropContext ctx) { ... }
}
```

We can copy the map functionality from exitProp() in the listener and paste it into the visitor method associated with rule prop.

listeners/TestPropertyFileVisitor.java

```
public static class PropertyFileVisitor extends
    PropertyFileBaseVisitor<Void>
{
    Map<String,String> props = new OrderedHashMap<String, String>();
    public Void visitProp(PropertyFileParser.PropContext ctx) {
        String id = ctx.ID().getText(); // prop : ID '=' STRING '\n' ;
        String value = ctx.STRING().getText();
        props.put(id, value);
        return null; // Java says must return something even when Void
    }
}
```

For comparison to the listener version in the previous section, here's the inheritance relationship between the visitor's interfaces and classes:



Visitors walk parse trees by explicitly calling interface ParseTreeVisitor's visit() method on child nodes. That method is implemented in AbstractParseTreeVisitor. In this case, the nodes created for prop invocations don't have children, so visitProp() doesn't have to call visit(). We'll look at visitor generic type parameters in *Traversing Parse Trees with Visitors*, on page 119.

The biggest difference between a visitor and listener test rig (such as TestPropertyFile) is that visitor test rigs don't need a ParseTreeWalker. They just ask the visitor to visit the tree created by the parser.

listeners/TestPropertyFileVisitor.java

```
PropertyFileVisitor loader = new PropertyFileVisitor();
loader.visit(tree);
System.out.println(loader.props); // print results
```

With all of that in place, here's the build and test sequence:

```
$ antlr4 -visitor PropertyFile.g4  # create visitor as well this time
$ ls PropertyFile*.java
PropertyFileBaseListener.java   PropertyFileListener.java
PropertyFileBaseVisitor.java    PropertyFileParser.java
PropertyFileLexer.java          PropertyFileVisitor.java
$ javac TestPropertyFileVisitor.java
$ cat t.properties
user="parrt"
machine="maniac"
$ java TestPropertyFileVisitor t.properties
{user="parrt", machine="maniac"}
```

We can build just about anything we want with visitors and listeners. Once we're in the Java space, there's no more ANTLR stuff to learn. All we have to know is the relationship between a grammar, its parse tree, and the visitor or listener event methods. Beyond that, it's just code. In response to recognizing input phrases, we can generate output, collect information (as we've done here), validate phrases in some way, or perform computations.

This property file example is small enough that we didn't run into an issue regarding rules with alternatives. By default, ANTLR generates a single kind of event for each rule no matter which alternative the parser matches. That's pretty inconvenient because the listener and visitor methods have to figure

out which alternative was matched by the parser. In the next section, we'll see how to get events at a finer granularity.

7.4 Labeling Rule Alternatives for Precise Event Methods

To illustrate the event granularity problem, let's try to build a simple calculator with a listener for the following expression grammar:

listeners/Expr.g4

```
grammar Expr;
s : e ;
e : e op=MULT e    // MULT is '*'
  | e op=ADD e     // ADD is '+'
  | INT
  ;
```

As it stands, rule e yields a fairly unhelpful listener because all alternatives of e result in a tree walker triggering the same enterE() and exitE() methods.

```
public interface ExprListener extends ParseTreeListener {
    void enterE(ExprParser.EContext ctx);
    void exitE(ExprParser.EContext ctx);
    ...
}
```

The listener methods would have to test to see which alternative the parser matched for each e subtree using the op token label and the methods of ctx.

listeners/TestEvaluator.java

```
public void exitE(ExprParser.EContext ctx) {
    if ( ctx.getChildCount()==3 ) { // operations have 3 children
        int left = values.get(ctx.e(0));
        int right = values.get(ctx.e(1));
        if ( ctx.op.getType()==ExprParser.MULT ) {
            values.put(ctx, left * right);
        }
        else {
            values.put(ctx, left + right);
        }
    }
    else {
        values.put(ctx, values.get(ctx.getChild(0))); // an INT
    }
}
```

The MULT field referenced in exitE() is generated by ANTLR in ExprParser:

```
public class ExprParser extends Parser {
    public static final int MULT=1, ADD=2, INT=3, WS=4;
    ...
}
```

If we look at class EContext in class ExprParser, we can see that ANTLR packed all elements from all three alternatives into the same context object.

```
public static class EContext extends ParserRuleContext {
    public Token op;                    // derived from label op
    public List<EContext> e() { ... }   // get all e subtrees
    public EContext e(int i) { ... }    // get ith e subtree
    public TerminalNode INT() { ... }   // get INT node if alt 3 of e
    ...
}
```

To get more precise listener events, ANTLR lets us label the outermost alternatives of any rule using the # operator. Let's derive grammar LExpr from Expr and label e's alternatives. Here's the modified e rule:

listeners/LExpr.g4

```
e : e MULT e            # Mult
  | e ADD e             # Add
  | INT                 # Int
  ;
```

Now ANTLR generates a separate listener method for each alternative of e. Consequently, we don't need the op token label anymore. For alternative label X, ANTLR generates enterX() and exitX().

```
public interface LExprListener extends ParseTreeListener {
    void enterMult(LExprParser.MultContext ctx);
    void exitMult(LExprParser.MultContext ctx);
    void enterAdd(LExprParser.AddContext ctx);
    void exitAdd(LExprParser.AddContext ctx);
    void enterInt(LExprParser.IntContext ctx);
    void exitInt(LExprParser.IntContext ctx);
    ...
}
```

Note also that ANTLR generates specific context objects (subclasses of EContext) for the alternatives, named after the labels. The getter methods of the specialized context objects are limited to just those applicable to the associated alternatives. For example, IntContext has only an INT() method. We can ask for ctx.INT() in enterInt() but not in enterAdd().

Listeners and visitors are great. We get reusable and retargetable grammars as well as encapsulated language applications just by fleshing out event methods. ANTLR even automatically generates the skeleton code for us. It turns out, though, that the applications we've built so far are so simple we haven't run into a common implementation issue, which is that event methods sometimes need to pass around partial results or other information.

7.5 Sharing Information Among Event Methods

Whether we're collecting information or computing values, it's often most convenient and good programming practice to pass around arguments and return values, rather than using fields or other "global variables." The problem is that ANTLR automatically generates the signature of the listener methods without application-specific return values or arguments. ANTLR also generates visitor methods without application-specific arguments.

In this section, we're going to explore mechanisms that let event methods pass data around without altering the event method signatures. We'll build three different implementations of the same simple calculator based upon the LExpr expression grammar from the previous section. The first implementation uses visitor method return values, the second defines a field shared among event methods, and the third annotates parse tree nodes to squirrel away values of interest.

Traversing Parse Trees with Visitors

To build a visitor-based calculator, the easiest approach is to have the event methods associated with rule expr return subexpression values. For example, visitAdd() would return the result of adding two subexpressions. visitInt() would return the value of the integer. Conventional visitors don't specify return values for their visit methods. Adding a return type is easy when we implement a class for our specific application's needs, extending LExprBaseVisitor<T> and supplying Integer as the <T> type parameter. Here's what our visitor looks like:

listeners/TestLEvalVisitor.java

```
public static class EvalVisitor extends LExprBaseVisitor<Integer> {
    public Integer visitMult(LExprParser.MultContext ctx) {
        return visit(ctx.e(0)) * visit(ctx.e(1));
    }

    public Integer visitAdd(LExprParser.AddContext ctx) {
        return visit(ctx.e(0)) + visit(ctx.e(1));
    }

    public Integer visitInt(LExprParser.IntContext ctx) {
        return Integer.valueOf(ctx.INT().getText());
    }
}
```

EvalVisitor inherits the general visit() method from ANTLR's AbstractParseTreeVisitor class, which our visitor uses to concisely trigger subtree visits.

Notice that EvalVisitor doesn't have a visitor method for rule s. The default implementation of visitS() in LExprBaseVisitor calls predefined method ParseTreeVisitor.visitChildren(). visitChildren() returns the value returned from the visit of the last child. In this case, visitS() returns the value of the expression returned from visiting its only child (the e node). We can use this default behavior.

In test rig TestLEvalVisitor.java, we have the usual code to launch LExprParser and print the parse tree. Then we need code to launch EvalVisitor and print out the expression value computed while visiting the tree.

listeners/TestLEvalVisitor.java
```
EvalVisitor evalVisitor = new EvalVisitor();
int result = evalVisitor.visit(tree);
System.out.println("visitor result = "+result);
```

To build our calculator, we tell ANTLR to generate the visitor, using the -visitor option as we did for the property file grammar earlier. (If we no longer want to generate a listener, we also use option -no-listener.) Here's the complete build and test sequence:

```
⇛ $ antlr4 -visitor LExpr.g4
⇛ $ javac LExpr*.java TestLEvalVisitor.java
⇛ $ java TestLEvalVisitor
⇛ 1+2*3
⇛ EOF
❮ (s (e (e 1) + (e (e 2) * (e 3))))
visitor result = 7
```

Visitors work very well if we need application-specific return values because we get to use the built-in Java return value mechanism. If we prefer not having to explicitly invoke visitor methods to visit children, we can switch to the listener mechanism. Unfortunately, that means giving up the cleanliness of using Java method return values.

Simulating Return Values with a Stack

ANTLR generates listener event methods that return no values (void return types). To return values to listener methods executing on nodes higher in the parse tree, we can store partial results in a field of our listener. A stack of values comes to mind, just as the Java runtime uses the CPU stack to store method return values temporarily. The idea is to push the result of computing a subexpression onto the stack. Methods for subexpressions further up the parse tree pop operands off the stack. Here's the full Evaluator calculator listener (physically in file TestLEvaluator.java):

listeners/TestLEvaluator.java
```
public static class Evaluator extends LExprBaseListener {
    Stack<Integer> stack = new Stack<Integer>();

    public void exitMult(LExprParser.MultContext ctx) {
        int right = stack.pop();
        int left = stack.pop();
        stack.push( left * right );
    }

    public void exitAdd(LExprParser.AddContext ctx) {
        int right = stack.pop();
        int left = stack.pop();
        stack.push(left + right);
    }

    public void exitInt(LExprParser.IntContext ctx) {
        stack.push( Integer.valueOf(ctx.INT().getText()) );
    }
}
```

To try this, we can create and use a ParseTreeWalker in test rig TestLEvaluator, following what we did in TestPropertyFile earlier in this chapter.

```
⇛ $ antlr4 LExpr.g4
⇛ $ javac LExpr*.java TestLEvaluator.java
⇛ $ java TestLEvaluator
⇛ 1+2*3
⇛ EOF
❮ (s (e (e 1) + (e (e 2) * (e 3))))
stack result = 7
```

Using a stack field is a bit awkward but works fine. We have to make sure that the event methods push and pop things in the correct order across listener events. Visitors with return values do away with the awkwardness of the stack but require that we manually visit the nodes of the tree. The third option is to capture partial results by stashing them in tree nodes.

Annotating Parse Trees

Instead of using temporary storage to share data between event methods, we can store those values in the parse tree itself. We can use this tree annotation approach with both a listener and a visitor, but we'll demonstrate how it works using a listener here. Let's start by looking at the LExpr grammar parse tree for 1+2*3 annotated with partial results (Figure 8, *The LExpr grammar parse tree for 1+2*3*, on page 122).

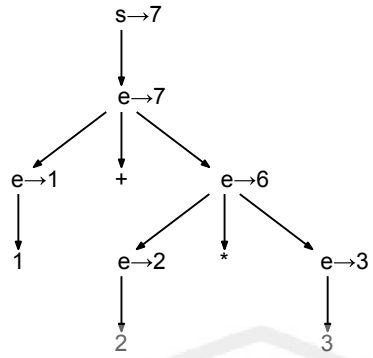


Figure 8—The LExpr grammar parse tree for 1+2*3

Each subexpression corresponds to a subtree root (and to an e rule invocation). The numbers pointed at by horizontal right arrows emanating from e nodes are the partial results we'd like to "return."

Adding Fields to Nodes via Rule Arguments and Return Values

If we didn't care about tying our grammar to a particular language, we could simply add a return value to the rule specification of interest.

```
e returns [int value]
  : e '*' e             # Mult
  | e '+' e             # Add
  | INT                 # Int
  ;
```

ANTLR places all parameters and return values of rules into the associated context object and so value ends up as a field of EContext.

```
public static class EContext extends ParserRuleContext {
    public int value;
    ...
}
```

Because the alternative-specific context is derived from EContext, all listener methods have access to this value. For example, listener methods could say ctx.value = 0;.

The approach shown here involves specifying that a rule method produces a result value, which is stored in the rule's context object. The specification uses target language syntax and consequently ties the grammar to a particular target language. However, the approach doesn't necessarily tie the grammar to a particular application, assuming that other applications can use the same rule result values. On the other hand, if we need more than one return value or a return value of a different type for a different application, this approach would not work.

Let's see how the node annotation strategy would work for rule e from our LExpr grammar.

listeners/LExpr.g4
```
e : e MULT e            # Mult
  | e ADD e             # Add
  | INT                 # Int
  ;
```

The listener methods for the alternatives of e would each store a result in the corresponding e parse-tree node. Any subsequent add or multiply events on nodes higher up in the parse tree would grab subexpression values by looking at the values stored in their corresponding children.

If we assume, for the moment, that each parse-tree node (each rule context object) has a field called value, then exitAdd() would look like this:

```
public void exitAdd(LExprParser.AddContext ctx) {
    // e(0).value is the subexpression value of the first e in the alternative
    ctx.value = ctx.e(0).value + ctx.e(1).value; // e '+' e # Add
}
```

That looks pretty reasonable, but unfortunately, we can't extend class ExprContext to add field value dynamically in Java (like Ruby and Python can). To make parse-tree annotation work, we need a way to annotate the various nodes without manually altering the associated node classes generated by ANTLR. (Otherwise, ANTLR would overwrite our changes the next time it generated code.)

The easiest way to annotate parse-tree nodes is to use a Map that associates arbitrary values with nodes. For that reason, ANTLR provides a simple helper class called ParseTreeProperty. Let's build another calculator version called EvaluatorWithProps in file TestLEvaluatorWithProps.java that associates partial results with LExpr parse-tree nodes using a ParseTreeProperty instance. Here's the appropriate definition at the start of our listener:

listeners/TestLEvaluatorWithProps.java
```
public static class EvaluatorWithProps extends LExprBaseListener {
    /** maps nodes to integers with Map<ParseTree,Integer> */
    ParseTreeProperty<Integer> values = new ParseTreeProperty<Integer>();
```

Caution: If you want to use your own field of type Map instead of ParseTreeProperty, make sure to derive it from IdentityHashMap, not the usual HashMap. We need to annotate specific nodes, testing by identity instead of equals(). Two e nodes might be equals() but not the same physical node in memory.

To annotate a node, we say values.put(node, value). To get a value associated with a node, we say values.get(node). This is OK, but let's create some helper methods with obvious names to make the code easier to read.

listeners/TestLEvaluatorWithProps.java
```
public void setValue(ParseTree node, int value) { values.put(node, value); }
public int getValue(ParseTree node) { return values.get(node); }
```

Let's start the listener methods with the simplest expression alternative, Int. We want to annotate its parse-tree e node with the integer value of the INT token it matches.

listeners/TestLEvaluatorWithProps.java
```
public void exitInt(LExprParser.IntContext ctx) {
    String intText = ctx.INT().getText(); // INT   # Int
    setValue(ctx, Integer.valueOf(intText));
}
```

For addition subtrees, we get the value of the two subexpression children (operands) and annotate the subtree root with the sum.

listeners/TestLEvaluatorWithProps.java
```
public void exitAdd(LExprParser.AddContext ctx) {
    int left = getValue(ctx.e(0)); // e '+' e   # Add
    int right = getValue(ctx.e(1));
    setValue(ctx, left + right);
}
```

Method exitMult() is the same except that the calculation uses multiply instead of add.

Our test rig starts parsing at rule s, so we have to make sure that the parse-tree root has the value of the e subtree. (Or, we could start parsing at e instead of s.) To bubble up the value from the e node to the root s node, we implement exitS().

listeners/TestLEvaluatorWithProps.java
```
/** Need to pass e's value out of rule s : e ; */
public void exitS(LExprParser.SContext ctx) {
    setValue(ctx, getValue(ctx.e())); // like: int s() { return e(); }
}
```

Here's how to launch the listener and print out the expression value from the parse-tree root node:

listeners/TestLEvaluatorWithProps.java
```
ParseTreeWalker walker = new ParseTreeWalker();
EvaluatorWithProps evalProp = new EvaluatorWithProps();
walker.walk(evalProp, tree);
System.out.println("properties result = " +evalProp.getValue(tree));
```

And here's the build and test sequence:

```
⇛ $ antlr4 LExpr.g4
⇛ $ javac LExpr*.java TestLEvaluatorWithProps.java
⇛ $ java TestLEvaluatorWithProps
⇛ 1+2*3
⇛ EOF
❮ (s (e (e 1) + (e (e 2) * (e 3))))
properties result = 7
```

Now we've seen three implementations of the same calculator, and we're almost ready to put our knowledge to use building real examples. Because each approach has its strengths and weaknesses, let's review what we've learned so far in this chapter and compare the different techniques.

Comparing Information Sharing Approaches

To get reusable and retargetable grammars, we need to keep them completely clear of user-defined actions. This means putting all of the application-specific code into some kind of listener or visitor external to the grammar. Listeners and visitors operate on parse trees, and ANTLR automatically generates appropriate tree-walking interfaces and default implementations. Since the event method signatures are fixed and not application specific, we looked at three ways event methods can share information.

- *Native Java call stack*: Visitor methods return a value of user-defined type. If a visitor needs to pass parameters, it must also use one of the next two techniques.
- *Stack-based*: A stack field simulates parameters and return values like the Java call stack.
- *Annotator*: A map field annotates nodes with useful values.

All three are completely decoupled from the grammar itself and nicely encapsulated in specialized objects. Beyond that, there are advantages and disadvantages to each. The needs of our problem and personal taste dictate which approach to take. There's also no reason that we can't use a variety of solutions even within the same application.

Visitor methods read nicely because they directly call other visitor methods to get partial results and can return values like any other method. That is also their negative. The visitor methods must explicitly visit their children, whereas the listeners do not. Because the visitor has a general interface, it can't define arguments. Visitors must use one of the other solutions to pass arguments to visitor methods it calls on children. A visitor's space efficiency

is good because it has to keep around only a few partial results at any one time. There are no partial results hanging around after the tree walk. While visitor methods can return values, each value must be of the same type, unlike the other solutions.

The stack-based solution can simulate arguments and return values with a stack, but there's a chance of a disconnect when manually managing the stack. This can occur because the listener methods aren't calling each other directly. As programmers, we have to make sure that what we push is appropriately popped off by future event method calls. The stack can pass multiple values and multiple return values. The stack-based solution is also space efficient because it does not attach anything to the tree. All partial results storage goes away after the tree walk.

The annotator is my default solution because it allows me to arbitrarily provide information to event methods operating on nodes above and below in the parse tree. I can pass multiple values around, and they can be of arbitrary types. Annotation is better than using a stack with fleeting values in many cases. There is less chance of a disconnect between the data-passing expectations of the various methods. Annotating the tree with `setValue(ctx, value)` is less intuitive than saying `return value` in a programming language but is more general. The only disadvantage of this approach over the other two is that the partial results are kept around during the tree walk and so it has a larger memory footprint.

On the other hand, being able to annotate the tree is precisely what we need in some applications, such as Section 8.4, *Validating Program Symbol Usage*, on page 138. That application requires multiple passes over the tree, and it's convenient for the first pass to compute and squirrel away data in the tree. The second pass then has easy access to the data as the parse-tree walker rewalks the tree. All in all, tree annotation is extremely flexible and has an acceptable memory burden.

Now that we know how to implement some basic language applications using parse-tree listeners and visitors, it's time to build some real tools based upon these techniques. That's exactly what we'll do in the next chapter.

CHAPTER 8

Building Some Real Language Applications

Now that we know how to trigger application code via listeners and visitors, it's time to build some useful applications. We'll construct four listeners of increasing complexity based upon the CSV, JSON, and Cymbol grammars from Chapter 6, *Exploring Some Real Grammars*, on page 83. (We could just as easily use visitors.)

The first real application is a loader for CSV files that constructs a two-dimensional list data structure. Then, we'll figure out how to translate JSON text files into XML text files. Next, we'll read in Cymbol programs and visualize the function call dependency graph using DOT/graphviz. Finally, we'll build a real symbol table for Cymbol programs that checks for undefined variables or functions and verifies that variables and functions are used properly. The checker needs to make multiple passes over the parse tree and, therefore, demonstrates how to collect information in one pass and use it in the next.

Let's get started with the simplest application.

8.1 Loading CSV Data

Our goal is to build a listener that loads comma-separated-value (CSV) data into a nice "list of maps" data structure. This is the kind of thing that any data format reader or even a configuration file reader would do. We'll collect the fields of each row into a map that associates a header name with a value. So, given the following input:

listeners/t.csv

```
Details,Month,Amount
Mid Bonus,June,"$2,000"
,January,"""zippo"""
Total Bonuses,"","$5,000"
```

we'd like to see the following list of maps printed out:

```
[{Details=Mid Bonus, Month=June, Amount="$2,000"},
 {Details=, Month=January, Amount="""zippo"""},
 {Details=Total Bonuses, Month="", Amount="$5,000"}]
```

To get precise methods within our listener, let's label each of the alternatives in rule field from the CSV grammar we built in Section 6.1, *Parsing Comma-Separated Values*, on page 84.

listeners/CSV.g4

```
grammar CSV;

file : hdr row+ ;
hdr : row ;

row : field (',' field)* '\r'? '\n' ;

field
    :   TEXT    # text
    |   STRING  # string
    |           # empty
    ;

TEXT : ~[,\n\r"]+ ;
STRING : '"' ('""'|~'"')* '"' ;
```

Other than that, the CSV grammar is the same as before.

We can start the implementation of our listener by defining the data structures we'll need. First, we need the main data structure, a list of maps called rows. We also need a list of the column names found in the header row, header. To process a row, we'll read the field values into a temporary list, currentRowFieldValues, and then map the column names to those values as we finish each row.

Here is the start of our listener:

listeners/LoadCSV.java

```
public static class Loader extends CSVBaseListener {
    public static final String EMPTY = "";
    /** Load a list of row maps that map field name to value */
    List<Map<String,String>> rows = new ArrayList<Map<String, String>>();
    /** List of column names */
    List<String> header;
    /** Build up a list of fields in current row */
    List<String> currentRowFieldValues;
```

The following three rule methods process field values by computing the appropriate string and adding it to the currentRowFieldValues:

listeners/LoadCSV.java
```
public void exitString(CSVParser.StringContext ctx) {
    currentRowFieldValues.add(ctx.STRING().getText());
}

public void exitText(CSVParser.TextContext ctx) {
    currentRowFieldValues.add(ctx.TEXT().getText());
}

public void exitEmpty(CSVParser.EmptyContext ctx) {
    currentRowFieldValues.add(EMPTY);
}
```

Before we can process the rows, we need to get the list of column names from the first row. The header row is just another row syntactically, but we need to treat it differently than a regular row of data. That means we need to check context. For now, let's assume that after exitRow() executes, currentRowFieldValues contains a list of the column names. To fill in header, we just have to capture the field values of that first row.

listeners/LoadCSV.java
```
public void exitHdr(CSVParser.HdrContext ctx) {
    header = new ArrayList<String>();
    header.addAll(currentRowFieldValues);
}
```

Turning to the rows themselves, we need two operations: one when we start a row and one when we finish. When we start a row, we need to allocate (or clear) currentRowFieldValues to prepare for getting a fresh set of data.

listeners/LoadCSV.java
```
public void enterRow(CSVParser.RowContext ctx) {
    currentRowFieldValues = new ArrayList<String>();
}
```

At the end of a row, we have to consider context. If we just loaded the header row, we don't want to alter the rows field. The column names aren't data. In exitRow(), we can test context by looking at the getRuleIndex() value of our parent node in the parse tree (or by asking if the parent is of type HdrContext). If the current row is instead a data row, we create a map using values obtained by simultaneously walking the column names in header and the values in currentRowFieldValues.

listeners/LoadCSV.java
```
public void exitRow(CSVParser.RowContext ctx) {
    // If this is the header row, do nothing
    // if ( ctx.parent instanceof CSVParser.HdrContext ) return; OR:
    if ( ctx.getParent().getRuleIndex() == CSVParser.RULE_hdr ) return;
```

```
    // It's a data row
    Map<String, String> m = new LinkedHashMap<String, String>();
    int i = 0;
    for (String v : currentRowFieldValues) {
        m.put(header.get(i), v);
        i++;
    }
    rows.add(m);
}
```

And that's all there is to loading the CSV data into a nice data structure. After using a ParseTreeWalker to traverse the tree, our main() in LoadCSV can print out the rows field.

listeners/LoadCSV.java

```
ParseTreeWalker walker = new ParseTreeWalker();
Loader loader = new Loader();
walker.walk(loader, tree);
System.out.println(loader.rows);
```

Here's the build and test sequence:

```
$ antlr4 CSV.g4
$ javac CSV*.java LoadCSV.java
$ java LoadCSV t.csv
[{Details=Mid Bonus, Month=June, Amount="$2,000"}, {Details=, Month=January,
 Amount="""zippo"""}, {Details=Total Bonuses, Month="", Amount="$5,000"}]
```

The other thing we might want to do after reading some data is to translate it to a different language, which is what we'll do in the next section.

8.2 Translating JSON to XML

Lots of web services return JSON data, and we might run into a situation where we want to feed some JSON data into existing code that accepts only XML. Let's use our JSON grammar from Section 6.2, *Parsing JSON*, on page 86 as a foundation to build a JSON to XML translator. Our goal is to read in JSON text like this:

listeners/t.json

```
{
    "description" : "An imaginary server config file",
    "logs" : {"level":"verbose", "dir":"/var/log"},
    "host" : "antlr.org",
    "admin": ["parrt", "tombu"],
    "aliases": []
}
```

and emit XML in an equivalent form, like this:

```
<description>An imaginary server config file</description>
<logs>
    <level>verbose</level>
    <dir>/var/log</dir>
</logs>
<host>antlr.org</host>
<admin>
    <element>parrt</element>
    <element>tombu</element>
</admin>
<aliases></aliases>
```

where <element> is a tag we need to conjure up during translation.

As we did with the CSV grammar, let's label some of the alternatives in the JSON grammar to get ANTLR to generate more precise listener methods.

listeners/JSON.g4

```
object
    :   '{' pair (',' pair)* '}'    # AnObject
    |   '{' '}'                     # EmptyObject
    ;

array
    :   '[' value (',' value)* ']'  # ArrayOfValues
    |   '[' ']'                     # EmptyArray
    ;
```

We'll do the same thing for rule value, but with a twist. All but three of the alternatives just have to return the text of the value matched by the alternative. We can use the same label for all of them, causing the parse-tree walker to trigger the same listener method for those alternatives.

listeners/JSON.g4

```
value
    :   STRING          # String
    |   NUMBER          # Atom
    |   object          # ObjectValue
    |   array           # ArrayValue
    |   'true'          # Atom
    |   'false'         # Atom
    |   'null'          # Atom
    ;
```

To implement our translator, it makes sense to have each rule return the XML equivalent of the input phrase matched by the rule. To track these partial results, we'll annotate the parse tree using field xml and two helper methods.

listeners/JSON2XML.java
```
public static class XMLEmitter extends JSONBaseListener {
    ParseTreeProperty<String> xml = new ParseTreeProperty<String>();
    String getXML(ParseTree ctx) { return xml.get(ctx); }
    void setXML(ParseTree ctx, String s) { xml.put(ctx, s); }
```

We'll attach the translated string for each subtree to the root of that subtree. Methods working on nodes further up the parse tree can grab those values to compute larger strings. The string attached to the root node is then the complete translation.

Let's start with the easiest translation. The Atom alternatives of value "return" (annotate the Atom node with) the text of the matched token (ctx.getText() gets the text matched by that rule invocation).

listeners/JSON2XML.java
```
public void exitAtom(JSONParser.AtomContext ctx) {
    setXML(ctx, ctx.getText());
}
```

Strings are basically the same except we have to strip off the double quotes (stripQuotes() is a helper method in the file).

listeners/JSON2XML.java
```
public void exitString(JSONParser.StringContext ctx) {
    setXML(ctx, stripQuotes(ctx.getText()));
}
```

If the value() rule method finds an object or array, it can copy the partial translation for that composite element to its own parse-tree node. Here's how to do it for objects:

listeners/JSON2XML.java
```
public void exitObjectValue(JSONParser.ObjectValueContext ctx) {
    // analogous to String value() {return object();}
    setXML(ctx, getXML(ctx.object()));
}
```

Once we can translate all of the values, we need to worry about name-value pairs and converting them to tags and text. The tag name for the resulting XML is derived from the STRING in the STRING ':' value alternative. The text in between the open and close tags is derived from the text attached to the value child.

listeners/JSON2XML.java
```
public void exitPair(JSONParser.PairContext ctx) {
    String tag = stripQuotes(ctx.STRING().getText());
    JSONParser.ValueContext vctx = ctx.value();
    String x = String.format("<%s>%s</%s>\n", tag, getXML(vctx), tag);
    setXML(ctx, x);
}
```

JSON objects consist of name-value pairs. So, for every pair found by object in the alternative marked by AnObject, we append the results computed here in the parse tree:

listeners/JSON2XML.java
```
public void exitAnObject(JSONParser.AnObjectContext ctx) {
    StringBuilder buf = new StringBuilder();
    buf.append("\n");
    for (JSONParser.PairContext pctx : ctx.pair()) {
        buf.append(getXML(pctx));
    }
    setXML(ctx, buf.toString());
}
public void exitEmptyObject(JSONParser.EmptyObjectContext ctx) {
    setXML(ctx, "");
}
```

Processing arrays follows a similar pattern, simply joining the list of results from child nodes and then wrapping them in <element> tags.

listeners/JSON2XML.java
```
public void exitArrayOfValues(JSONParser.ArrayOfValuesContext ctx) {
    StringBuilder buf = new StringBuilder();
    buf.append("\n");
    for (JSONParser.ValueContext vctx : ctx.value()) {
        buf.append("<element>"); // conjure up element for valid XML
        buf.append(getXML(vctx));
        buf.append("</element>");
        buf.append("\n");
    }
    setXML(ctx, buf.toString());
}

public void exitEmptyArray(JSONParser.EmptyArrayContext ctx) {
    setXML(ctx, "");
}
```

Finally, we need annotate the root of the parse tree with the overall translation, collected from an object or array.

listeners/JSON.g4
```
json:   object
    |   array
    ;
```

We can do that in our listener with a simple set operation.

listeners/JSON2XML.java
```
public void exitJson(JSONParser.JsonContext ctx) {
    setXML(ctx, getXML(ctx.getChild(0)));
}
```

Here's the build and test sequence:

```
$ antlr4 JSON.g4
$ javac JSON*.java
$ java JSON2XML t.json

<description>An imaginary server config file</description>
<logs>
<level>verbose</level>
...
```

Translations are not always as straightforward as JSON to XML. But, this example shows us how to approach the problem of sentence translation by piecing together partially translated phrases. (If you look in the source code directory, you'll also see a version that uses StringTemplate,[1] JSON2XML_ST.java, to generate output and another that builds XML DOM trees, JSON2XML_DOM.java.)

OK, enough playing around with data. Let's do something interesting with a programming language.

8.3 Generating a Call Graph

Software is hard to write and maintain, which is why we try to build tools to increase our productivity and effectiveness. For example, over the past decade we've seen an explosion of testing frameworks, code coverage tools, and code analyzers. It's also nice to see a class hierarchy visually as a tree, and most development environments support this. The other visualization I like is called a *call graph*, which has functions as nodes and function calls as directed edges between the nodes.

In this section, we're going to build a call graph generator using the Cymbol grammar from Section 6.4, *Parsing Cymbol*, on page 98. I think you'll be surprised at how simple it is, especially given how cool the results are. To give you an idea of what we're trying to achieve, consider the following set of functions and function calls:

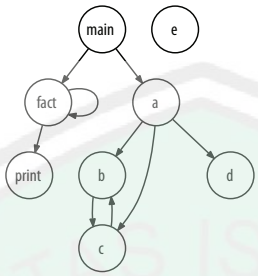
listeners/t.cymbol

```
int main() { fact(); a(); }

float fact(int n) {
  print(n);

  if ( n==0 ) then return 1;
  return n * fact(n-1);
}
```

1. http://www.stringtemplate.org

```
void a() { int x = b(); if false then {c(); d();} }
void b() { c(); }
void c() { b(); }
void d() { }
void e() { }
```

We'd like to visualize the call graph like this:



The good thing about visualizations is that the human eye can easily pick out aberrations. For example, the e() node is an orphan, which means that no one calls it and it is therefore dead code. At a glance, we found a function to jettison. We can also detect recursion very easily by looking for cycles in the graph such as fact() → fact() and b() → c() → b().

To visualize a call graph, we need to read in a Cymbol program and create a DOT file (and then view it with graphviz). For example, here is the DOT we need to generate for file t.cymbol from the earlier example:

```
digraph G {
  ranksep=.25;
  edge [arrowsize=.5]
  node [shape=circle, fontname="ArialNarrow",
        fontsize=12, fixedsize=true, height=.45];
  main; fact; a; b; c; d; e;
  main -> fact;
  main -> a;
  fact -> print;
  fact -> fact;
  a -> b;
  a -> c;
  a -> d;
  b -> c;
  c -> b;
}
```

The output consists of boilerplate setup statements such as ranksep=.25; and then a list of nodes and edges. To catch orphan nodes, we need to make sure to generate a node definition for each function name, even if it has no incoming or outgoing edges. It would not appear in the graph otherwise. Note the e on the end of the node definition line.

```
main; fact; a; b; c; d; e;
```

Our strategy is straightforward. When the parser finds a function declaration, our application will add the name of the current function to a list and set a field called currentFunctionName. When the parser sees a function call, our application will record an edge from the currentFunctionName to the callee's function name.

To get started, let's label some rule alternatives in Cymbol.g4 to get more precise listener methods.

listeners/Cymbol.g4

```
expr:   ID '(' exprList? ')'    # Call
    |   expr '[' expr ']'       # Index
    |   '-' expr                # Negate
    |   '!' expr                # Not
    |   expr '*' expr           # Mult
    |   expr ('+'|'-') expr     # AddSub
    |   expr '==' expr          # Equal
    |   ID                      # Var
    |   INT                     # Int
    |   '(' expr ')'            # Parens
    ;
```

Then, as a foundation for our language application, let's encapsulate all of the graph-related stuff into a class.

listeners/CallGraph.java

```
static class Graph {
    // I'm using org.antlr.v4.runtime.misc: OrderedHashSet, MultiMap
    Set<String> nodes = new OrderedHashSet<String>(); // list of functions
    MultiMap<String, String> edges =                  // caller->callee
        new MultiMap<String, String>();
    public void edge(String source, String target) {
        edges.map(source, target);
    }
```

From the collection of nodes and edges, we can dump out the appropriate DOT code using a little bit of Java in toDOT() of class Graph.

listeners/CallGraph.java

```
public String toDOT() {
    StringBuilder buf = new StringBuilder();
    buf.append("digraph G {\n");
```

```
        buf.append("  ranksep=.25;\n");
        buf.append("  edge [arrowsize=.5]\n");
        buf.append("  node [shape=circle, fontname=\"ArialNarrow\",\n");
        buf.append("        fontsize=12, fixedsize=true, height=.45];\n");
        buf.append("  ");
        for (String node : nodes) { // print all nodes first
            buf.append(node);
            buf.append("; ");
        }
        buf.append("\n");
        for (String src : edges.keySet()) {
            for (String trg : edges.get(src)) {
                buf.append("  ");
                buf.append(src);
                buf.append(" -> ");
                buf.append(trg);
                buf.append(";\n");
            }
        }
        buf.append("}\n");
        return buf.toString();
    }
```

Now all we have to do is fill in those data structures using a listener. The listener needs two fields for bookkeeping.

listeners/CallGraph.java

```
static class FunctionListener extends CymbolBaseListener {
    Graph graph = new Graph();
    String currentFunctionName = null;
```

And our application only needs to listen for two events. First, it has to record the current function name as the parser finds function declarations.

listeners/CallGraph.java

```
public void enterFunctionDecl(CymbolParser.FunctionDeclContext ctx) {
    currentFunctionName = ctx.ID().getText();
    graph.nodes.add(currentFunctionName);
}
```

Then, when the parser detects a function call, the application records an edge from the current function to the invoked function.

listeners/CallGraph.java

```
public void exitCall(CymbolParser.CallContext ctx) {
    String funcName = ctx.ID().getText();
    // map current function to the callee
    graph.edge(currentFunctionName, funcName);
}
```

Notice that the function calls can't hide inside nested code blocks or declarations such as in a().

```
void a() { int x = b(); if false then {c(); d();} }
```

The tree walker triggers listener method exitCall() regardless of where it finds function calls.

With the parse tree and class FunctionListener, we can launch a walker with our listener to generate output.

listeners/CallGraph.java

```
ParseTreeWalker walker = new ParseTreeWalker();
FunctionListener collector = new FunctionListener();
walker.walk(collector, tree);
System.out.println(collector.graph.toString());
System.out.println(collector.graph.toDOT());
```

Before dumping the DOT string, that code prints out the list of functions and edges.

```
$ antlr4 Cymbol.g4
$ javac Cymbol*.java CallGraph.java
$ java CallGraph t.cymbol
edges: {main=[fact, a], fact=[print, fact], a=[b, c, d], b=[c], c=[b]},
functions: [main, fact, a, b, c, d, e]
digraph G {
  ranksep=.25;
  edge [arrowsize=.5]
...
```

Naturally, to view the call graph, cut and paste just the output starting with digraph G {.

With very little code, we were able to build a call graph generator in this section. To demonstrate that our Cymbol grammar is reusable, we're going to use it again without modification in the next section to build a totally different application. Not only that, but we'll make two passes over the same tree with two different listeners.

8.4 Validating Program Symbol Usage

To build an interpreter, compiler, or translator for a programming language such as Cymbol, we'd need to verify that Cymbol programs used symbols (identifiers) properly. In this section, we're going to build a Cymbol validator that checks the following conditions:

- Variable references have corresponding definitions that are visible to them (in scope).

- Function references have corresponding definitions (functions can appear in any order).
- Variables are not used as functions.
- Functions are not used as variables.

To check all these conditions, we have a bit of work to do, so this example is going to take a little bit longer than the others to absorb. But, our reward will be a great base from which to build real language tools.

Let's get started by taking a look at some sample Cymbol code with lots of different references, some of which are invalid.

listeners/vars.cymbol

```
int f(int x, float y) {
    g();   // forward reference is ok
    i = 3; // no declaration for i (error)
    g = 4; // g is not variable (error)
    return x + y; // x, y are defined, so no problem
}

void g() {
    int x = 0;
    float y;
    y = 9; // y is defined
    f();   // backward reference is ok
    z();   // no such function (error)
    y();   // y is not function (error)
    x = f; // f is not a variable (error)
}
```

To verify that everything is OK within a program according to the previous conditions, we should print out the list of functions and their local variables plus the list of global symbols (functions and global variables). Further, we should emit an error when we find a problem. For example, for the previous input, let's build an application called CheckSymbols that generates the following:

```
⇒ $ java CheckSymbols vars.cymbol
❮ locals:[]
  function<f:tINT>:[<x:tINT>, <y:tFLOAT>]
  locals:[x, y]
  function<g:tVOID>:[]
  globals:[f, g]
  line 3:4 no such variable: i
  line 4:4 g is not a variable
  line 13:4 no such function: z
  line 14:4 y is not a function
  line 15:8 f is not a variable
```

The key to implementing this kind of problem is an appropriate data structure called a *symbol table*. Our application will store symbols in the symbol table and then check identifier references for correctness by looking them up in the symbol table. In the next section, we'll take a peek at what the data structure looks like and then use it to solve the validation problem at hand.

A Crash Course in Symbol Tables

Language implementers typically call the data structure that holds symbols a *symbol table*. The language being implemented dictates a symbol table's structure and complexity. If a programming language allows the same identifier to mean different things in different contexts, the symbol table groups symbols into *scopes*. A scope is just a set of symbols such as a list of parameters for a function or the list of variables and functions in a global scope.

The symbol table by itself is just a repository of symbol definitions—it doesn't do any checking. To validate code, we need to check the variable and function references in expressions against the rules we set up earlier. There are two fundamental operations for symbol validation: *defining* symbols and *resolving* symbols. Defining a symbol means adding it to a scope. Resolving a symbol means figuring out which definition the symbol refers to. In some sense, resolving a symbol means finding the "closest" matching definition. The closest scope is the closest enclosing scope. For example, let's look at another Cymbol example that has symbol definitions in different scopes (labeled with circumscribed numbers).

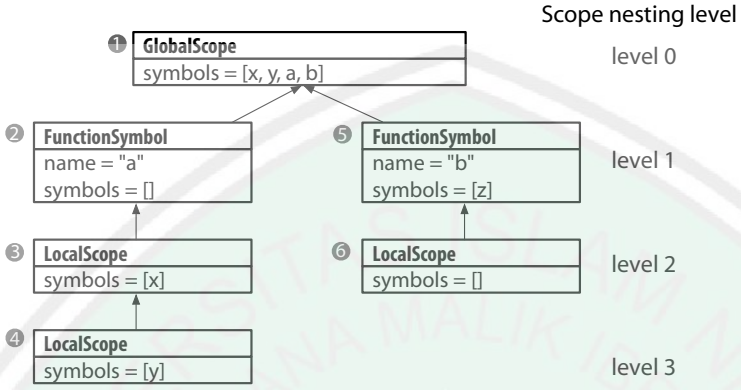
listeners/vars2.cymbol

```
1 int x;
  int y;
2 void a()
3 {
      int x;
      x = 1;  // x resolves to current scope, not x in global scope
      y = 2;  // y is not found in current scope, but resolves in global
4     { int y = x; }
  }
5 void b(int z)
6 { }
```

The global scope, ❶, contains variables `x` and `y` as well as the functions `a()` and `b()`. Functions live in the global scope, but they also constitute new scopes that hold the functions' parameters, if any: ❷ and ❺. Also nested within a function scope is the function's local code block (❸ and ❻), which constitutes

another new scope. Local variables are held in local scopes (❸, ❹, and ❻) nested within the function scopes.

Because symbol x is defined twice, we can't just stuff all of our identifiers into a single set without collision. That's where scopes come in. We keep a set of scopes and allow only a single definition for each identifier in a scope. We also keep a pointer to the parent scope so that we can find symbol definitions in outer scopes. The scopes form a tree.



The circled numbers refer to the scope from the source code. The nodes along the path from any scope to the root (global scope) form a stack of scopes. To find a symbol's definition, we start in the scope surrounding the reference and walk up the scope tree until we find its definition.

Rather than reimplement an appropriate symbol table just for this example, I've copied the symbol table source code[2] from Chapter 6 of *Language Implementation Patterns [Par09]*. I encourage you to look through the source for BaseScope, GlobalScope, LocalScope, Symbol, FunctionSymbol, and VariableSymbol to get a feel for the implementation. Together, those classes embody the symbol table, and we'll assume that they just work. With a symbol table in place, we're ready to build our validator.

Validator Architecture

To begin building our validator, let's think about the big picture and form an overall strategy. We can break this problem down according to the key operations: *define* and *resolve*. For definitions, we need to listen for variable and function definition events and insert Symbol objects into the scope surrounding that definition. At the start of a function, we need to "push" a new scope and then pop it at the end of that function definition.

2. http://pragprog.com/book/tpdsl/language-implementation-patterns

To resolve and check symbol references, we need to listen for variable and function name references within expressions. For each reference, we'll verify that there is a matching definition and that the reference uses the symbol properly.

That seems straightforward, but there's a complication: a Cymbol program can call a function defined after it in the source file. We call that a *forward reference*. To support them, we need to make two passes over the parse tree. The first pass, or phase, defines the symbols including the functions, and the second pass does the resolutions. In this way, the second pass can see all functions in the file. Here's the code in the test rig that triggers both passes over the parse tree:

listeners/CheckSymbols.java

```
ParseTreeWalker walker = new ParseTreeWalker();
DefPhase def = new DefPhase();
walker.walk(def, tree);
// create next phase and feed symbol table info from def to ref phase
RefPhase ref = new RefPhase(def.globals, def.scopes);
walker.walk(ref, tree);
```

During the definition phase, we'll be creating lots of scopes. Unless we keep references to them, the garbage collector will throw the scopes out. For the symbol table to survive the transition from the definition to the resolution phase, we need to track those scopes. The most logical place to squirrel them away is in the parse tree itself (or, technically, using an annotation map that associates values with tree nodes). The reference phase can then simply pick up the current scope pointer as it descends the parse tree. Tree nodes associated with functions and local blocks will get pointers to their scopes.

Defining and Resolving Symbols

With our general strategy in mind, let's build our validator, starting with the DefPhase. Our phase classes need three fields: a reference to a global scope, a parse tree annotator to track the scopes we create, and a pointer to the current scope. The listener code for enterFile() starts off the activity, creating the global scope. At the end, exitFile() is responsible for printing the results.

listeners/DefPhase.java

```
public class DefPhase extends CymbolBaseListener {
    ParseTreeProperty<Scope> scopes = new ParseTreeProperty<Scope>();
    GlobalScope globals;
    Scope currentScope; // define symbols in this scope
    public void enterFile(CymbolParser.FileContext ctx) {
        globals = new GlobalScope(null);
        currentScope = globals;
    }
```

```
    public void exitFile(CymbolParser.FileContext ctx) {
        System.out.println(globals);
    }
```

When the parser finds a function declaration, our application needs to create a FunctionSymbol object. FunctionSymbol objects do double duty as a symbol and as the scope containing the arguments. To nest the function scope within the global scope, we "push" the function scope. We do that by setting the function's enclosing scope to be the current scope and resetting the current scope.

listeners/DefPhase.java

```
public void enterFunctionDecl(CymbolParser.FunctionDeclContext ctx) {
    String name = ctx.ID().getText();
    int typeTokenType = ctx.type().start.getType();
    Symbol.Type type = CheckSymbols.getType(typeTokenType);

    // push new scope by making new one that points to enclosing scope
    FunctionSymbol function = new FunctionSymbol(name, type, currentScope);
    currentScope.define(function); // Define function in current scope
    saveScope(ctx, function);      // Push: set function's parent to current
    currentScope = function;       // Current scope is now function scope
}

void saveScope(ParserRuleContext ctx, Scope s) { scopes.put(ctx, s); }
```

Method saveScope() annotates the functionDecl rule node with the function scope so our reference phase can pick it up later. As we leave a function, we pop the function scope so that the current scope is again the global scope.

listeners/DefPhase.java

```
public void exitFunctionDecl(CymbolParser.FunctionDeclContext ctx) {
    System.out.println(currentScope);
    currentScope = currentScope.getEnclosingScope(); // pop scope
}
```

Local scopes work in a similar way. We push a scope in listener method enterBlock() and pop it in exitBlock().

Now that we've taken care of the scopes and function definitions, let's define the arguments and variables.

listeners/DefPhase.java

```
public void exitFormalParameter(CymbolParser.FormalParameterContext ctx) {
    defineVar(ctx.type(), ctx.ID().getSymbol());
}

public void exitVarDecl(CymbolParser.VarDeclContext ctx) {
    defineVar(ctx.type(), ctx.ID().getSymbol());
}
```

```
void defineVar(CymbolParser.TypeContext typeCtx, Token nameToken) {
    int typeTokenType = typeCtx.start.getType();
    Symbol.Type type = CheckSymbols.getType(typeTokenType);
    VariableSymbol var = new VariableSymbol(nameToken.getText(), type);
    currentScope.define(var); // Define symbol in current scope
}
```

That finishes up the definition phase.

To build our reference phase, let's start by setting the current scope to the global scope passed to us from the definition phase.

listeners/RefPhase.java

```
public RefPhase(GlobalScope globals, ParseTreeProperty<Scope> scopes) {
    this.scopes = scopes;
    this.globals = globals;
}
public void enterFile(CymbolParser.FileContext ctx) {
    currentScope = globals;
}
```

Then, as the tree walker triggers enter and exit events for Cymbol functions and blocks, our listener methods keep currentScope up-to-date by accessing values stored in the tree during the definition phase.

listeners/RefPhase.java

```
public void enterFunctionDecl(CymbolParser.FunctionDeclContext ctx) {
    currentScope = scopes.get(ctx);
}
public void exitFunctionDecl(CymbolParser.FunctionDeclContext ctx) {
    currentScope = currentScope.getEnclosingScope();
}

public void enterBlock(CymbolParser.BlockContext ctx) {
    currentScope = scopes.get(ctx);
}
public void exitBlock(CymbolParser.BlockContext ctx) {
    currentScope = currentScope.getEnclosingScope();
}
```

With the scopes set appropriately as the walker proceeds, we can resolve symbols by implementing listener methods for variable references and function calls. When the walker encounters a variable reference, it calls exitVar(), which uses resolve() to try to find the name in the current scope's symbol table. If resolve() doesn't find the symbol in the current scope, it looks up the enclosing scope chain. If necessary, resolve() will look all the way up to the global scope. It returns null if it can't find a suitable definition. If, however, resolve() finds a symbol but it's a function, not a variable, we need to generate an error message.

listeners/RefPhase.java
```
public void exitVar(CymbolParser.VarContext ctx) {
    String name = ctx.ID().getSymbol().getText();
    Symbol var = currentScope.resolve(name);
    if ( var==null ) {
        CheckSymbols.error(ctx.ID().getSymbol(), "no such variable: "+name);
    }
    if ( var instanceof FunctionSymbol ) {
        CheckSymbols.error(ctx.ID().getSymbol(), name+" is not a variable");
    }
}
```

Handling function calls is basically the same. We emit an error if we can't find a definition or we find a variable, not a function.

Finally, here's the build and test sequence that shows the desired output from earlier:

```
$ antlr4 Cymbol.g4
$ javac Cymbol*.java CheckSymbols.java *Phase.java *Scope.java *Symbol.java
$ java CheckSymbols vars.cymbol
locals:[]
function<f:tINT>:[<x:tINT>, <y:tFLOAT>]
...
```

With both passes complete, we've finished our symbol validator. We had to cover a lot ground, but the effort is worthwhile because this example is a great starting point for creating your own language tools. The implementation of the listeners is only about 150 lines of Java, with the symbol table support code coming in at another 100. If you're not actively building a tool that needs a symbol table at the moment, don't sweat the details here. The takeaway is that there is a well-known solution to tracking and validating symbols that's not rocket science. To learn more about symbol table management, I shamelessly suggest you purchase and dig through *Language Implementation Patterns [Par09]*.

If you've been following along pretty well so far in this section of the book, you're in great shape! Not only can you build grammars by digging through language reference manuals, you can bring those grammars to life to perform useful tasks by implementing listeners. Certainly there are language problems out there you might have difficulty with at this point, but your kung fu is very strong.

This chapter finishes Part II of the book. Once you have some experience using these key ANTLR skills, you'll want to jump into the next part to learn about advanced ANTLR usage.

Part III

Advanced Topics

In Part II, we learned how to abstract language structure (syntax) from language samples and reference manuals and then how to formally describe syntax with ANTLR grammars. To develop language applications, we built several tree listeners and visitors that operated on automatically generated parse trees. Now we have the keys to effectively use ANTLR for most problems.

Part III is about advanced usages of ANTLR. First, we'll examine ANTLR's automatic error handling mechanism. Then, we'll explore how to embed code snippets directly within a grammar in order to generate output or perform computations on-the-fly during the parse. Then we'll see how to dynamically turn alternatives in the grammar on and off based upon runtime information using semantic predicates. Finally, we'll perform some lexical black magic.

CHAPTER 9

Error Reporting and Recovery

As we develop a grammar, there will be lots of mistakes to fix just like any piece of software. The resulting parser won't recognize all valid sentences until we finish (and debug) our grammar. In the meantime, informative error messages help us track down grammar problems. Once we have a correct grammar, we then have to deal with ungrammatical sentences entered by users or even ungrammatical sentences generated by other programs gone awry.

In both situations, the manner in which our parser responds to ungrammatical input is an important productivity consideration. In other words, a parser that responds with "Eh?" and bails out upon the first syntax error isn't very useful for us during development or for our users during deployment.

Developers using ANTLR get a good error reporting facility and a sophisticated error recovery strategy for free. ANTLR generates parsers that automatically emit rich error messages upon syntax error and successfully resynchronize much of the time. The parsers even avoid generating more than a single error message for each syntax error.

In this chapter, we'll learn about the automatic error reporting and recovery strategy used by ANTLR-generated parsers. We'll also see how to alter the default error handling mechanism to suit atypical needs and how to customize error messages for a specific application domain.

9.1 A Parade of Errors

The best way to describe ANTLR's error recovery strategy is to watch an ANTLR-generated parser respond to erroneous input. Let's look at a grammar for a simple Java-like language containing class definitions with field and method members. The methods have simple statements and expressions.

We'll use it as the core of the examples in this section and the remainder of the chapter.

errors/Simple.g4

```
grammar Simple;

prog:   classDef+ ; // match one or more class definitions

classDef
    :   'class' ID '{' member+ '}' // a class has one or more members
        {System.out.println("class "+$ID.text);}
    ;

member
    :   'int' ID ';'                        // field definition
        {System.out.println("var "+$ID.text);}
    |   'int' f=ID '(' ID ')' '{' stat '}' // method definition
        {System.out.println("method: "+$f.text);}
    ;

stat:   expr ';'
        {System.out.println("found expr: "+$stat.text);}
    |   ID '=' expr ';'
        {System.out.println("found assign: "+$stat.text);}
    ;

expr:   INT
    |   ID '(' INT ')'
    ;

INT :   [0-9]+ ;
ID  :   [a-zA-Z]+ ;
WS  :   [ \t\r\n]+ -> skip ;
```

The embedded actions print out elements as the parser finds them. We're using embedded actions instead of a parse-tree listener for simplicity and brevity. We'll learn more about actions in Chapter 10, *Attributes and Actions*, on page 175.

First, let's run the parser with some valid input to observe the normal output.

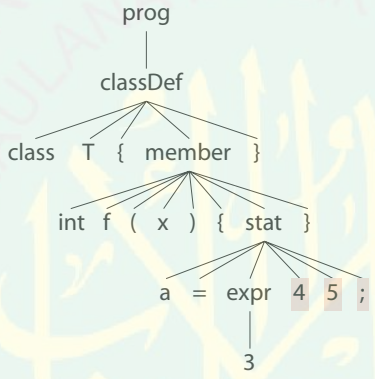
```
⇒ $ antlr4 Simple.g4
⇒ $ javac Simple*.java
⇒ $ grun Simple prog
⇒ class T { int i; }
⇒ EOF
❮ var i
  class T
```

We get no errors from the parser, and it executes the print statements to report proper recognition of variable i and class definition T.

Now, let's try a class with a method definition containing a bogus assignment expression.

```
⇛ $ grun Simple prog -gui
⇛ class T {
⇛   int f(x) { a = 3 4 5; }
⇛ }
⇛ EOF
❮ line 2:19 mismatched input '4' expecting ';'
  method: f
  class T
```

At the 4 token, the parser doesn't find the ; it was expecting and reports an error. The line 2:19 indicates that the offending token was found on the second line at the twentieth character position (character positions start from zero). Because of the -gui option, we also see the parse tree with error nodes highlighted (more on this in a moment).



In this case, there are two extra tokens, and the parser gives a generic error message about the mismatch. If there is just a single extra token, however, the parser can be a little bit smarter, indicating it's an extraneous token. In the following test run, there is an extraneous ; after the class name and before the start of the class body:

```
⇛ $ grun Simple prog
⇛ class T ; { int i; }
⇛ EOF
❮ line 1:8 extraneous input ';' expecting '{'
  var i
  class T
```

The parser reports an error at the ; but gives a slightly more informative answer because it knows that the next token is what it was actually looking for. This feature is called *single-token deletion* because the parser can simply pretend the extraneous token isn't there and keep going.

Language Theory Humor

Apparently, the great Niklaus Wirth[a] had an excellent sense of humor. He used to joke that in Europe people called him by "reference" (properly pronouncing his name "Ni-klaus Virt") and that in America people called him by "value" (pronouncing his name "Nickle-less Worth").

At the Compiler Construction 1994 conference, Kristen Nygaard[b] (inventor of Simula) told a story about how, while teaching a language theory course, he commented that "Strong typing is fascism," referring to his preference for languages that are loose with types. A student came up to him afterward and asked why typing hard on the keyboard was fascism.

a. See http://en.wikipedia.org/wiki/Niklaus_Wirth.
b. See http://en.wikipedia.org/wiki/Kristen_Nygaard.

Similarly, the parser can do *single-token insertion* when it detects a missing token. Let's chop off the closing } to see what happens.

```
⇛ $ grun Simple prog
⇛ class T {
⇛   int f(x) { a = 3; }
⇛ EOF
❮ found assign: a=3;
  method: f
  line 3:0 missing '}' at '<EOF>'
  class T
```

The parser reports that it couldn't find the required ending } token.

Another common syntax error occurs when the parser is at a decision point and the remaining input isn't consistent with any of the alternatives of that rule or subrule. For example, if we forget the variable name in a field declaration, neither of the alternatives in rule member will match. The parser reports that there is no viable alternative.

```
⇛ $ grun Simple prog
⇛ class T { int ; }
⇛ EOF
❮ line 1:14 no viable alternative at input 'int;'
  class T
```

There is no space between int and ; because we told the lexer to skip() in the whitespace WS() rule.

If there are lexical errors, ANTLR also emits an error message indicating the character or characters it could not match as part of a token. For example, if we send in a completely unknown character, we get a token recognition error.

```
⇛ $ grun Simple prog
⇛ class # { int i; }
⇛ EOF
❮ line 1:6 token recognition error at: '#'
  line 1:8 missing ID at '{'
  var i
  class <missing ID>
```

Since we did not give a valid class name, the single-token insertion mechanism conjured up the name missing ID so that the class name token was non-null. To take control of how the parser conjures up tokens, override getMissingSymbol() in DefaultErrorStrategy (see Section 9.5, *Altering ANTLR's Error Handling Strategy*, on page 171).

You might have noticed that the sample runs in this section show the actions executing as expected, despite the presence of errors. Aside from producing good error messages and resynchronizing the input by consuming tokens, parsers also have to bounce to an appropriate location in the generated code.

For example, when matching members via rule member in rule classDef, the parser should not bail out of classDef upon a bad member definition. That's why the parser is still able to execute those actions—a syntax error does not cause the parser to exit the rule. The parser tries really hard to keep looking for a valid class definition. We'll learn all about this topic in Section 9.3, *Automatic Error Recovery Strategy*, on page 158. But first, let's look at altering standard error reporting to help with grammar debugging and to provide better messages for our users.

9.2 Altering and Redirecting ANTLR Error Messages

By default, ANTLR sends all errors to standard error, but we can change the destination and the content by providing an implementation of interface ANTLRErrorListener. The interface has a syntaxError() method that applies to both lexers and parsers. Method syntaxError() receives all sorts of information about the location of the error as well as the error message. It also receives a reference to the parser, so we can query it about the state of recognition.

For example, here's an error listener (in test rig TestE_Listener.java) that prints out the rule invocation stack followed by the usual error message augmented with offending token information:

errors/TestE_Listener.java

```
public static class VerboseListener extends BaseErrorListener {
    @Override
    public void syntaxError(Recognizer<?, ?> recognizer,
                            Object offendingSymbol,
                            int line, int charPositionInLine,
                            String msg,
                            RecognitionException e)
    {
        List<String> stack = ((Parser)recognizer).getRuleInvocationStack();
        Collections.reverse(stack);
        System.err.println("rule stack: "+stack);
        System.err.println("line "+line+":"+charPositionInLine+" at "+
                           offendingSymbol+": "+msg);
    }

}
```

With this definition, our application can easily add an error listener to the parser before invoking the start rule.

errors/TestE_Listener.java

```
SimpleParser parser = new SimpleParser(tokens);
parser.removeErrorListeners(); // remove ConsoleErrorListener
parser.addErrorListener(new VerboseListener()); // add ours
parser.prog(); // parse as usual
```

Right before we add our error listener, we need to remove the standard console error listener so that we don't get repeated error messages.

Let's see what the error messages look like now for a class definition containing an extra class name and missing field name.

```
⇛ $ javac TestE_Listener.java
⇛ $ java TestE_Listener
⇛ class T T {
⇛   int ;
⇛ }
⇛ EOF
❮ rule stack: [prog, classDef]
  line 1:8 at [@2,8:8='T',<9>,1:8]: extraneous input 'T' expecting '{'
  rule stack: [prog, classDef, member]
  line 2:6 at [@5,18:18=';',<8>,2:6]: no viable alternative at input 'int;'
  class T
```

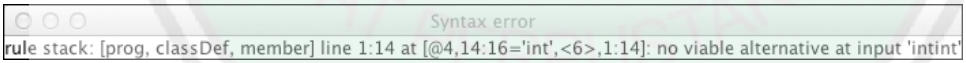
Stack [prog, classDef] indicates that the parser is in rule classDef, which was called by prog. Notice that the token information contains the character position within the input stream. This is useful for highlighting errors in the input like development environments do. For example, token [@2,8:8='T',<9>,1:8] indicates that it is the third token in the token stream (index 2 from 0), ranges from characters 8 to 8, has token type 9, resides on line 1, and is at character position 8 (counting from 0 in treating tabs as one character).

We can just as easily send that message to a dialog box using Java Swing by altering the syntaxError() method.

errors/TestE_Dialog.java

```
public static class DialogListener extends BaseErrorListener {
    @Override
        public void syntaxError(Recognizer<?, ?> recognizer,
                                Object offendingSymbol,
                                int line, int charPositionInLine,
                                String msg,
                                RecognitionException e)
    {
        List<String> stack = ((Parser)recognizer).getRuleInvocationStack();
        Collections.reverse(stack);
        StringBuilder buf = new StringBuilder();
        buf.append("rule stack: "+stack+" ");
        buf.append("line "+line+":"+charPositionInLine+" at "+
                   offendingSymbol+": "+msg);

        JDialog dialog = new JDialog();
        Container contentPane = dialog.getContentPane();
        contentPane.add(new JLabel(buf.toString()));
        contentPane.setBackground(Color.white);
        dialog.setTitle("Syntax error");
        dialog.pack();
        dialog.setLocationRelativeTo(null);
        dialog.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);
        dialog.setVisible(true);
    }
}
```

Running test rig TestE_Dialog on input class T { int int i; } pops up a dialog box such as the following:



As another example, let's build an error listener, TestE_Listener2.java, that prints out the line with the offending token underlined, as in the following sample run:

```
⇛ $ javac TestE_Listener2.java
⇛ $ java TestE_Listener2
⇛ class T XYZ {
⇛   int ;
⇛ }
⇛ EOF
❮ line 1:8 extraneous input 'XYZ' expecting '{'
  class T XYZ {
          ^^^
  line 2:6 no viable alternative at input 'int;'
    int ;
        ^
  class T
```

To make things easier, we'll ignore tabs—charPositionInLine isn't the column number because tab size isn't universally defined. Here's an error listener implementation that underlines error locations in the input like we just saw:

errors/TestE_Listener2.java

```
public static class UnderlineListener extends BaseErrorListener {
        public void syntaxError(Recognizer<?, ?> recognizer,
                                Object offendingSymbol,
                                int line, int charPositionInLine,
                                String msg,
                                RecognitionException e)
    {
        System.err.println("line "+line+":"+charPositionInLine+" "+msg);
        underlineError(recognizer,(Token)offendingSymbol,
                       line, charPositionInLine);
    }

    protected void underlineError(Recognizer recognizer,
                                  Token offendingToken, int line,
                                  int charPositionInLine) {
        CommonTokenStream tokens =
            (CommonTokenStream)recognizer.getInputStream();
        String input = tokens.getTokenSource().getInputStream().toString();
        String[] lines = input.split("\n");
        String errorLine = lines[line - 1];
        System.err.println(errorLine);
        for (int i=0; i<charPositionInLine; i++) System.err.print(" ");
        int start = offendingToken.getStartIndex();
        int stop = offendingToken.getStopIndex();
        if ( start>=0 && stop>=0 ) {
            for (int i=start; i<=stop; i++) System.err.print("^");
        }
        System.err.println();
    }
}
```

There's one final thing to know about error listeners. When the parser detects an ambiguous input sequence, it notifies the error listener. The default error listener, ConsoleErrorListener, however, doesn't print anything to the console. As we saw in Section 2.3, *You Can't Put Too Much Water into a Nuclear Reactor*, on page 13, ambiguous input likely indicates an error in our grammar; the parser should not inform our users. Let's look back at the ambiguous grammar from that section that can match input f(); in two different ways.

errors/Ambig.g4

```
grammar Ambig;

stat: expr ';'        // expression statement
    | ID '(' ')' ';'  // function call statement
    ;

expr: ID '(' ')'
    | INT
    ;

INT :   [0-9]+ ;
ID  :   [a-zA-Z]+ ;
WS  :   [ \t\r\n]+ -> skip ;
```

If we test the grammar, we don't see a warning for the ambiguous input.

```
⇛ $ antlr4 Ambig.g4
⇛ $ javac Ambig*.java
⇛ $ grun Ambig stat
⇛ f();
⇛ EOF
```

To hear about it when the parser detects an ambiguity, tell the parser to use an instance of DiagnosticErrorListener using addErrorListener().

```
parser.removeErrorListeners(); // remove ConsoleErrorListener
parser.addErrorListener(new DiagnosticErrorListener());
```

You should also inform the parser that you're interested in all ambiguity warnings, not just those it can detect quickly. In the interest of efficiency, ANTLR's decision-making mechanism doesn't always chase down full ambiguity information. Here's how to make the parser report all ambiguities:

```
parser.getInterpreter()
    .setPredictionMode(PredictionMode.LL_EXACT_AMBIG_DETECTION);
```

If you're using TestRig via the grun alias, use option -diagnostics to have it use DiagnosticErrorListener instead of the default console error listener (and turn on LL_EXACT_AMBIG_DETECTION).

```
⇒ $ grun Ambig stat -diagnostics
⇒ f();
⇒ EOF
❮ line 1:3 reportAttemptingFullContext d=0, input='f();'
  line 1:3 reportAmbiguity d=0: ambigAlts={1, 2}, input='f();'
```

The output shows that the parser also calls reportAttemptingFullContext(). ANTLR calls this method when *SLL(*)* parsing fails and the parser engages the more powerful full *ALL(*)* mechanism. See Section 13.7, *Maximizing Parser Speed*, on page 243.

It's a good idea to use the diagnostics error listener during development since the ANTLR tool can't warn you about ambiguous grammar constructs statically (when generating parsers). Only the parser can detect ambiguities in ANTLR v4. It's the difference between static typing in Java, say, and the dynamic typing in Python.

Improvements in ANTLR v4

There are two error-related important improvements in v4: ANTLR does much better inline error recovery and makes it much easier for programmers to alter the error handling strategy. When Sun Microsystems was building a parser for JavaFX with ANTLR v3, it noticed that a single misplaced semicolon could force the parser to stop looking for a list of, say, class members (via member+). Now, v4 parsers attempt to resynchronize before and during subrule recognition instead of gobbling tokens and exiting the current rule. The second improvement lets programmers specify an error handling mechanism following the Strategy pattern.

Now that we have a good idea about the kinds of messages ANTLR parsers generate and how to tweak and redirect them, let's explore error recovery.

9.3 Automatic Error Recovery Strategy

Error recovery is what allows the parser to continue after finding a syntax error. In principle, the best error recovery would come from the human touch in a handwritten recursive-descent parser. In my experience, though, it's really tough to get good error recovery by hand because it's so tedious and easy to screw up. In this latest version of ANTLR, I've incorporated every bit of jujitsu I've learned and picked up over the years to provide good error recovery automatically for ANTLR grammars.

ANTLR's error recovery mechanism is based upon Niklaus Wirth's early ideas in *Algorithms + Data Structures = Programs [Wir78]* (as well as Rodney Topor's *A Note on Error Recovery in Recursive Descent Parsers [Top82]*) but also includes

Josef Grosch's good ideas from his CoCo parser generator (*Efficient and Comfortable Error Recovery in Recursive Descent Parsers [Gro90]*).

Here is how ANTLR uses those ideas together in a nutshell: parsers perform *single-token insertion* and *single-token deletion* upon mismatched token errors if possible. If not, parsers gobble up tokens until they find a token that could reasonably follow the current rule and then return, continuing as if nothing had happened. In this section, we'll see what those terms mean and explore how ANTLR recovers from errors in various situations. Let's begin with the fundamental recovery strategy that ANTLR uses.

Recovery by Scanning for Following Tokens

When faced with truly borked-up input, the current rule can't continue, so the parser recovers by gobbling up tokens until it thinks that it has resynchronized and then returns to the calling rule. We can call this the *sync-and-return* strategy. Some people call this "panic mode," but it works remarkably well. The parser knows it can't match the current input with the current rule. It can throw out tokens only until the lookahead is consistent with something that should match after the parser exits from the rule. For example, if there is a syntax error within an assignment statement, it makes a great deal of sense to throw out tokens until the parser sees a semicolon or other statement terminator. Drastic, but effective. As we'll see, ANTLR tries to recover within the rule before falling back on this basic strategy.

Each ANTLR-generated rule method is wrapped in a try-catch that responds to syntax errors by reporting the error and attempting to recover before returning.

```
try {
    ...
}
catch (RecognitionException re) {
    _errHandler.reportError(this, re);
    _errHandler.recover(this, re);
}
```

We'll look at the error handling strategy in more detail in Section 9.5, *Altering ANTLR's Error Handling Strategy*, on page 171, but, for now, we can summarize recover() as consuming tokens until it finds one in the *resynchronization set*. The resynchronization set is the union of rule reference *following sets* for all the rules on the invocation stack. The following set for a rule reference is the set of tokens that can match immediately following that reference and without leaving the current rule. So, for example, given alternative assign ';', the following

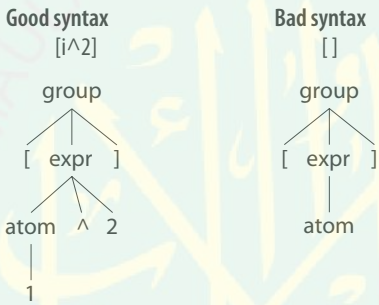
set for rule reference assign is {';'}. If the alternative were just assign, the following set would be empty.

It's worthwhile going through an example to lock in what resynchronization sets contain. Consider the following grammar, and imagine that, at each rule invocation, the parser tracks the following set for every rule invocation:

errors/F.g4

```
grammar F;
group
    : '[' expr ']'      // Tokens following ref to expr: {']'}
    | '(' expr ')'      // Tokens following ref to expr: {')'}
    ;
expr: atom '^' INT ;    // Tokens following ref to atom: {'^'}
atom: ID
    | INT
    ;

INT :   [0-9]+ ;
ID  :   [a-zA-Z]+ ;
WS  :   [ \t\r\n]+ -> skip ;
```

Consider the parse tree on the left for input [1^2] in the following diagram:



When matching token 1 in rule atom, the call stack is [group, expr, atom] (group called expr, which called atom). By looking at the call stack, we know precisely the set of tokens that can follow every rule the parser has called to get us to the current position. Following sets consider tokens only within the current rule so that, at runtime, we can combine just the sets associated with the current call stack. In other words, we can't get to rule expr from both alternatives of group at the same time.

Combining the following sets pulled from the comments in grammar F, we get a resynchronization set of {'^', ']'}. To see why this is the set we want, let's watch what happens when the parser encounters erroneous input []. We get the parse tree shown on the right in the earlier side-by-side diagram. In atom, the parser discovers that the current token,], isn't consistent with either

alternative of atom. To resynchronize, the parser consumes tokens until it finds a token from the resynchronization set. In this case, current token] starts out as a member of the resynchronization set so the parser doesn't actually consume any tokens to resynchronize in atom.

After finishing the recovery process in rule atom, the parser returns to rule expr but immediately discovers that it doesn't have the ^ token. The process repeats itself, and the parser consumes tokens until it finds something in the resynchronization set for rule expr. The resynchronization set for expr is the following set for the expr reference in the first alternative of group: { ']' }. Again, the parser does not consume anything and exits expr, returning to the first alternative of rule group. Now, the parser finds exactly what it is looking for following the reference to expr. It successfully matches the ']' in group, and the parser is now properly resynchronized.

During recovery, ANTLR parsers avoid emitting cascading error messages (an idea borrowed from Grosch). That is, parsers emit a single error message for each syntax error until they successfully recover from that error. Through the use of a simple Boolean variable, set upon syntax error, the parser avoids emitting further errors until the parser successfully matches a token and resets the variable. (See field errorRecoveryMode in class DefaultErrorStrategy.)

FOLLOW Sets vs. Following Sets

Those familiar with language theory will wonder whether the resynchronization set for rule atom is just *FOLLOW*(atom), the set of all viable tokens that can follow references to atom in some context. It isn't that simple, unfortunately, and the resynchronization sets must be computed dynamically to get the set of tokens that can follow the rule in a particular context rather than in all contexts. *FOLLOW*(expr) is { ')', ']' }, which includes all tokens that can follow references to expr in both contexts (alternatives 1 and 2 of group). Clearly, though, at runtime the parser can call expr from only one location at a time. Note that *FOLLOW*(atom) is '^', and if the parser resynchronized to that token instead of resynchronization set {'^', ']'}, it would consume until the end of file because there is no '^' on the input stream.

In many cases, ANTLR can recover more intelligently than consuming until the resynchronization set and returning from the current rule. It pays to attempt to "repair" the input and continue within the same rule. Over the next few sections, we'll look at how the parser recovers from mismatched tokens and errors within subrules.

Recovering from Mismatched Tokens

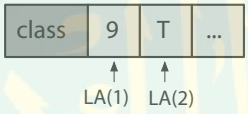
One of the most common operations during parsing is "match token." For every token reference, T, in the grammar, the parser invokes match(T). If the current token isn't T, match() notifies the error listener(s) and attempts to resynchronize. To resynchronize, it has three choices. It can delete a token, it can conjure one up, or it can punt and throw an exception to engage the basic sync-and-return mechanism.

Deleting the current token is the easiest way to resynchronize, if it makes sense to do so. Let's revisit rule classDef from our simple class definition language in grammar Simple.

errors/Simple.g4

```
classDef
    :   'class' ID '{' member+ '}' // a class has one or more members
        {System.out.println("class "+$ID.text);}
    ;
```

Given input class 9 T { int i; }, the parser will delete 9 and keep going within the rule to match the class body. The following image illustrates the state of the input after the parser has consumed class:



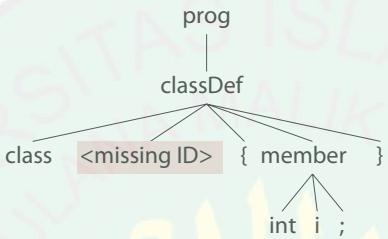
The LA(1) and LA(2) labels mark the first token of lookahead (the current token) and the second token of lookahead. The match(ID) expects LA(1) to be an ID, but it's not. However, the next token, LA(2), is in fact an ID. To recover, we just have to delete the current token (as noise), consume the ID we were expecting, and exit match().

If the parser can't resynchronize by deleting a token, it attempts to insert a token instead. Let's say we forgot the ID so that classDef sees input class { int i; }. After matching class, the input state looks like this:

The parser invokes match(ID) but instead of an identifier finds {. In this situation, the parser knows that the { is what it will need next since that is what follows the ID reference in classDef. To resynchronize, the match() can pretend to see the identifier and return, thus allowing the next match('{') call to succeed.

That works great if we ignore embedded actions, such as the print statement that references the class name identifier. The print statement references the missing token via $ID.text and will cause an exception if the token is null. Rather than simply pretending the token exists, the error handler conjures one up; see getMissingSymbol() in DefaultErrorStrategy. The conjured token has the token type that the parser expected and takes line and character position information from the current input token, LA(1). This conjured token also prevents exceptions in listeners and visitors that reference the missing token.

The easiest way to see what's going on is to look at the parse tree, which shows how the parser recognizes all the tokens. In the case of errors, the parse tree highlights in red the tokens that the parser deletes or conjures up during resynchronization. For input class { int i; } and grammar Simple, we get the following parse tree:



The parser also executes the embedded print actions without throwing an exception since error recovery conjures up a valid Token object for $ID.

```
⇛ $ grun Simple prog -gui
⇛ class { int i; }
⇛ EOF
❮ line 1:6 missing ID at '{'
  var i
  class <missing ID>
```

Naturally, an identifier with text <missing ID> isn't really useful for whatever goal we're trying to accomplish, but at least error recovery doesn't induce a bunch of null pointer exceptions.

Now that we know how ANTLR does in-rule recovery for simple token references, let's explore how it recovers from errors before and during subrule recognition.

Recovering from Errors in Subrules

Years ago the JavaFX group at Sun contacted me because their ANTLR-generated parser didn't recover well in certain cases. It turns out that the parser was bailing out of subrule loops like member+ at the first whiff of trouble,

forcing sync-and-return recovery for the surrounding rule. A small error in a member declaration like `var width Number;` (missing a colon after `width`) would force the parser to skip all of the remaining members.

Jim Idle, an ANTLR mailing list contributor and consultant, came up with what I call "Jim Idle's magic sync" error recovery. His solution was to manually insert references to an empty rule into the grammar that contained a special action that triggered error recovery when necessary. ANTLR v4 now automatically inserts synchronization checks at the start and at the loop continuation test to avoid such drastic recovery. The mechanism looks like this:

Subrule start At the start of any subrule, parsers attempt single-token deletion. But, unlike token matches, parsers don't attempt single-token insertion. ANTLR would have a hard time conjuring up a token because it would have to guess which of several alternatives would ultimately be successful.

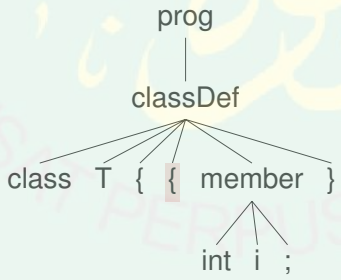
Looping subrule continuation test If the subrule is a looping construct, `(...)*` or `(...)+`, the parser tries to recover aggressively upon error to stay in the loop. After successfully matching an alternative of the loop, the parser consumes until it finds a token consistent with one of these sets:

(a) Another iteration of the loop

(b) What follows the loop

(c) The resynchronization set of the current

Let's look at single-token deletion in front of a subrule first. Consider the looping `member+` construct in rule `classDef` of grammar `Simple`. If we stutter and type an extra {, the `member+` subrule will delete the extra token before jumping into `member`, as shown in the following parse tree:



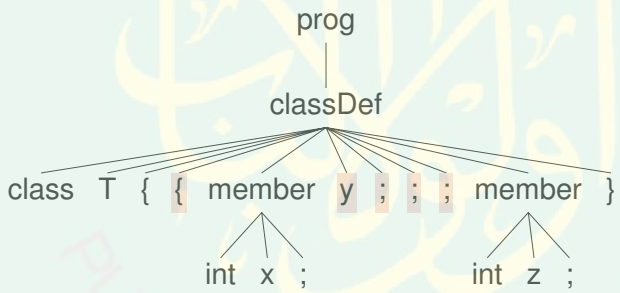
The following session confirms proper recovery because it correctly identifies variable `i`:

```
⇛ $ grun Simple prog
⇛ class T {{ int i; }
⇛ EOF
❮ line 1:9 extraneous input '{' expecting 'int'
  var i
  class T
```

Now let's try some really messed up input and see whether the member+ loop can recover and continue looking for members.

```
⇛ $ grun Simple prog
⇛ class T {{
⇛   int x;
⇛   y;;;
⇛   int z;
⇛ }
⇛ EOF
❮ line 1:9 extraneous input '{' expecting 'int'
  var x
  line 3:2 extraneous input 'y' expecting {'int', '}'}
  var z
  class T
```

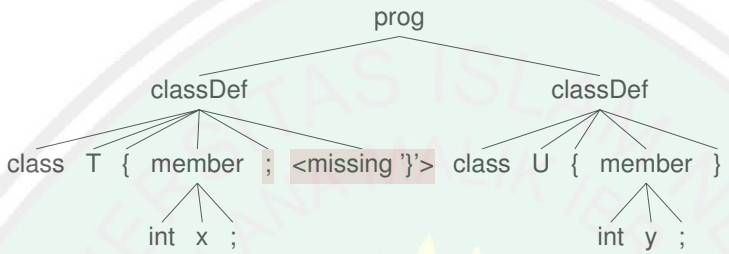
We know that the parser resynchronized and stayed within the loop because it identified variable z. The parser gobbles up y;;; until it sees the start of another member (set *(c)* earlier) and then loops back to member. If the input did not include int z;, the parser would have gobbled until it had seen } (set *(b)* above) and exited the loop. The parse tree highlights the deleted tokens and illustrates that the parser still interpreted int z; as a valid member.



If the user provides rule member with bad syntax and also forgets the closing } of a class, we wouldn't want the parser to scan until it finds }. Parser resynchronization could throw out an entire following class definition looking for }. Instead, the parser stops gobbling if it sees a token in set *(c)*, as shown the following session:

```
⇒ $ grun Simple prog
⇒ class T {
⇒   int x;
⇒   ;
⇒ class U { int y; }
⇒ EOF
❮ var x
  line 3:2 extraneous input ';' expecting {'int', '}'}
  class T
  var y
  class U
```

The parser stops resynchronization when it sees keyword class, as we can see from the parse tree.



Besides the recognition of tokens and subrules, parsers can also fail to match semantic predicates.

Catching Failed Semantic Predicates

We've gotten only a taste of semantic predicates at this point, but it's appropriate to discuss what happens upon failed predicates in this error handling chapter. We'll look at predicates in depth in Chapter 11, *Altering the Parse with Semantic Predicates*, on page 189. For now, let's treat semantic predicates like assertions. They specify conditions that must be true at runtime for the parser to get past them. If a predicate evaluates to false, the parser throws a FailedPredicateException exception, which is caught by the catch of the current rule. The parser reports an error and does the generic sync-and-return recovery.

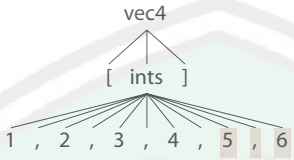
Let's look at an example that uses a semantic predicate to restrict the number of integers in a vector, very similar to the grammar in *Altering the Parse with Semantic Predicates*, on page 48. Rule ints matches up to max integers.

errors/Vec.g4
```
vec4:   '[' ints[4] ']' ;
ints[int max]
locals [int i=1]
    :   INT ( ',' {$i++;} {$i<=$max}? INT )*
    ;
```

Given one too many integers, as in the following session, we see an error message and get error recovery that throws out the extra comma and integers:

```
⇒ $ antlr4 Vec.g4
⇒ $ javac Vec*.java
⇒ $ grun Vec vec4
⇒ [1,2,3,4,5,6]
⇒ EOF
❮ line 1:9 rule ints failed predicate: {$i<=$max}?
```

The parse tree shows that the parser detected the error at the fifth integer.



The {$i<=$max} error message might be helpful to us as grammar designers, but it's certainly not helpful to our users. We can change the message from a chunk of code to something a little more readable by using the fail option on the semantic predicate. For example, here is the ints rule again but with an action that computes a readable string:

errors/VecMsg.g4

```
ints[int max]
locals [int i=1]
    :   INT ( ',' {$i++;} {$i<=$max}?<fail={"exceeded max "+$max}> INT )*
    ;
```

Now we get a better message for the same input.

```
⇒ $ antlr4 VecMsg.g4
⇒ $ javac VecMsg*.java
⇒ $ grun VecMsg vec4
⇒ [1,2,3,4,5,6]
⇒ EOF
❮ line 1:9 rule ints exceeded max 4
```

The fail option takes either a string literal in double quotes or an action that evaluates to a string. The action is handy if you'd like to execute a function when a predicate fails. Just use an action that calls a function such as {...}?<fail={failedMaxTest()}>.

A word of caution about using semantic predicates to test for input validity. In the vector example, the predicate enforces syntactic rules, so it's OK to throw an exception and try to recover. If, on the other hand, we have a syntactically valid but semantically invalid construct, it's not a good idea to use a semantic predicate.

Imagine that, in some language, we can assign any value to a variable except 0. That means assignment x = 0; is syntactically valid but semantically invalid. Certainly we have to emit an error to the user, but we should not trigger error recovery. x = 0; is perfectly legal syntactically. In a sense, the parser will automatically "recover" from the error. Here's a simple grammar that demonstrates the issue:

errors/Pred.g4

```
assign
    : ID '=' v=INT {$v.int>0}? ';'
      {System.out.println("assign "+$ID.text+" to ");}
    ;
```

If the predicate in rule assign throws an exception, the sync-and-return behavior will throw out the ; after the predicate. This might work out just fine, but we risk an imperfect resynchronization. A better solution is to emit an error manually and let the parser continue matching the correct syntax. So, instead of the predicate, we should use a simple action with a conditional.

```
{if ($v.int==0) notifyListeners("values must be > 0");}
```

Now that we've looked at all the situations that can trigger error recovery, it's worth pointing out a potential flaw in the mechanism. Given that the parser sometimes doesn't consume any tokens during a single recovery attempt, it's possible that overall recovery could go into an infinite loop. If we recover without consuming a token and get back to the same location in the parser, we will recover again without consuming a token. In the next section, we'll see how ANTLR avoids this pitfall.

Error Recovery Fail-Safe

ANTLR parsers have a built-in fail-safe to guarantee error recovery terminates. If we reach the same parser location and have the same input position, the parser forces a token consumption before attempting recovery. Returning to the simple Simple grammar from the start of this chapter, let's look at a sample input that trips the fail-safe. If we add an extra int token in a field definition, the parser detects an error and tries to recover. As we'll see in the next test run, the parser will call recover() and try to restart parsing multiple times before correctly resynchronizing (Figure 9, *Parser resynchronization*, on page 169).

The right parse tree in the diagram in Figure 10, *Parse trees for good and bad syntax*, on page 169 shows that there are three invocations of member from classDef.

```
⇒ $ grun Simple prog
⇒ class T {
⇒   int int x;
⇒ }
⇒ EOF
❮ line 2:6 no viable alternative at input 'intint'
  var x
  class T
```

Figure 9—Parser resynchronization

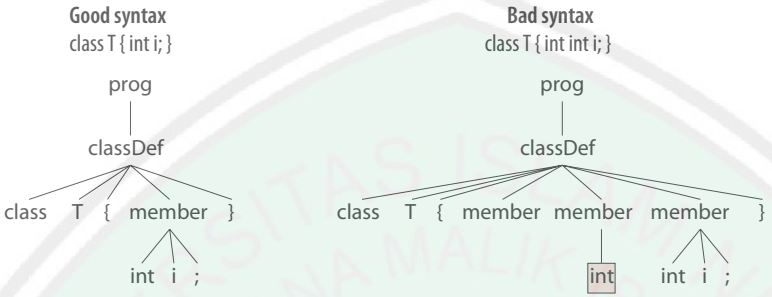


Figure 10—Parse trees for good and bad syntax

The first reference doesn't match anything, but the second one matches the extraneous int token. The third attempt at matching a member matches the proper int x; sequence.

Let's walk through the exact sequence of events. The parser is in rule member when it detects the first error.

errors/Simple.g4

```
member
    :   'int' ID ';'                        // field definition
        {System.out.println("var "+$ID.text);}
    |   'int' f=ID '(' ID ')' '{' stat '}' // method definition
        {System.out.println("method: "+$f.text);}
    ;
```

Input int int doesn't fit either alternative of member, so the parser engages the sync-and-return error recovery strategy. It emits the first error message and consumes until it sees a token in the resynchronization set for call stack [prog, classDef, member].

Because of the classDef+ and member+ loops in the grammar, computing the resynchronization set is a little complicated. Following the call to member, the parser could loop back and find another member or exit the loop and find

the '}' that closes the class definition. Following the call to classDef, the parser could loop back to see the start of another class or simply exit prog. So, for call stack [prog, classDef, member], the resynchronization set is {'int', '}', 'class'}.

At this point, the parser recovers without consuming a token because the current input token, int, is in the resynchronization set. It simply returns to the caller: the member+ loop in classDef. The loop then tries to match another member. Unfortunately, since it has not consumed any tokens, the parser detects another error when it returns to member (though it hushes the spurious error message, by virtue of the errorRecovery flag).

During recovery for this second error, the parser trips the fail-safe because it has arrived at the same parser location and input position. The fail-safe forces a token consumption before attempting resynchronization. Since int is in the resynchronization set, it doesn't consume a second token. Fortunately, that's exactly what we want because the parser is now properly resynchronized. The next three tokens represent a valid member definition: int x;. The parser returns once again from member to the loop in classDef. For the third time, we go back to member, but now parsing will succeed.

So, that's the story with ANTLR's automatic error recovery mechanism. Now let's look at a manual mechanism that can sometimes provide better error recovery.

9.4 Error Alternatives

Some syntax errors are so common that it's worth treating them specially. For example, programmers often have the wrong number of parentheses at the end of a function call with nested arguments. To handle these cases specially, all we have to do is add alternatives to match the erroneous but common syntax. The following grammar recognizes function calls with a single argument, possibly with nested parentheses in the argument. Rule fcall has two so-called *error alternatives*.

errors/Call.g4

```
stat:   fcall ';' ;
fcall
    :   ID '(' expr ')'
    |   ID '(' expr ')' ')' {notifyErrorListeners("Too many parentheses");}
    |   ID '(' expr         {notifyErrorListeners("Missing closing ')'");}
    ;

expr:   '(' expr ')'
    |   INT
    ;
```

While these error alternatives can make an ANTLR-generated parser work a little harder to choose between alternatives, they don't in any way confuse the parser. Just like any other alternative, the parser matches them if they are consistent with the current input. For example, let's try some input sequences that match the error alternatives, starting with a valid function call.

```
⇛ $ antlr4 Call.g4
⇛ $ javac Call*.java
⇛ $ grun Call stat
⇛ f(34);
⇛ EOF
⇛ $ grun Call stat
⇛ f((34);
⇛ EOF
❮ line 1:6 Missing closing ')'
⇛ $ grun Call stat
⇛ f((34)));
⇛ EOF
❮ line 1:8 Too many parentheses
```

At this point, we've learned quite a bit about the error messages that ANTLR parsers can generate and also how parsers recover from errors in lots of different situations. We've also seen how to customize error messages and redirect them to different error listeners. All of this functionality is controlled and encapsulated in an object that specifies ANTLR's error handling strategy. In the next section, we'll look at that strategy in detail to learn more about customizing how parsers respond to errors.

9.5 Altering ANTLR's Error Handling Strategy

The default error handling mechanism works very well, but there are a few atypical situations in which we might want to alter it. First, we might want to disable some of the in-line error handling because of its runtime overhead. Second, we might want to bail out of the parser upon the first syntax error. For example, when parsing a command line for a shell like `bash`, there's no point in trying to recover from errors. We can't risk executing that command anyway, so the parser can bail out at the first sign of trouble.

To explore the error handling strategy, take a look at interface `ANTLRErrorStrategy` and its concrete implementation class `DefaultErrorStrategy`. That class holds everything associated with the default error handling behavior. ANTLR parsers signal that object to report errors and recover. For example, here's the `catch` block inside of each ANTLR-generated rule function:

```
_errHandler.reportError(this, re);
_errHandler.recover(this, re);
```

_errHandler is a variable holding a reference to an instance of DefaultErrorStrategy. Methods reportError() and recover() embody the error reporting and sync-and-return functionality. reportError() delegates error reporting to one of three methods, according to the type of exception thrown.

Turning to our first atypical situation, let's decrease the runtime burden that error handling places on the parser. Take a look at this code that ANTLR generates for the member+ subrule in grammar Simple:

```
_errHandler.sync(this);
_la = _input.LA(1);
do {
    setState(22); member();
    setState(26);
    _errHandler.sync(this);
    _la = _input.LA(1);
} while ( _la==6 );
```

For applications where it's safe to assume the input is syntactically correct, such as network protocols, we might as well avoid the overhead of detecting and recovering from errors in subrules. We can do that by subclassing DefaultErrorStrategy and overriding sync() with an empty method. The Java compiler would likely then inline and eliminate the _errHandler.sync(this) calls. We'll see how to notify the parser to use a different error strategy through the next example.

The other atypical situation is bailing out of the parser upon the first syntax error. To make this work, we have to override three key recovery methods, as shown in the following code:

errors/BailErrorStrategy.java

```
import org.antlr.v4.runtime.*;

public class BailErrorStrategy extends DefaultErrorStrategy {
    /** Instead of recovering from exception e, rethrow it wrapped
     *  in a generic RuntimeException so it is not caught by the
     *  rule function catches.  Exception e is the "cause" of the
     *  RuntimeException.
     */

    @Override
    public void recover(Parser recognizer, RecognitionException e) {
        throw new RuntimeException(e);
    }
```

```
    /** Make sure we don't attempt to recover inline; if the parser
     *  successfully recovers, it won't throw an exception.
     */
    @Override
    public Token recoverInline(Parser recognizer)
        throws RecognitionException
    {
        throw new RuntimeException(new InputMismatchException(recognizer));
    }

    /** Make sure we don't attempt to recover from problems in subrules. */
    @Override
    public void sync(Parser recognizer) { }
}
```

For a test rig, we can reuse our typical boilerplate code. In addition to creating and launching the parser, we need to create a new BailErrorStrategy instance and tell the parser to use it instead of the default strategy.

errors/TestBail.java

```
parser.setErrorHandler(new BailErrorStrategy());
```

While we're at it, we should also bail out at the first lexical error. To do that, we have to override method recover() in Lexer.

errors/TestBail.java

```
public static class BailSimpleLexer extends SimpleLexer {
    public BailSimpleLexer(CharStream input) { super(input); }
    public void recover(LexerNoViableAltException e) {
        throw new RuntimeException(e); // Bail out
    }
}
```

Let's try a lexical error first by inserting a wacky # character at the beginning of the input. The lexer throws an exception that blasts control flow all the way out to the main program.

```
⇒ $ antlr4 Simple.g4
⇒ $ javac Simple*.java TestBail.java
⇒ $ java TestBail
⇒ # class T { int i; }
⇒ EOF
❮ line 1:1 token recognition error at: '#'
  Exception in thread "main"
  java.lang.RuntimeException: LexerNoViableAltException('#')
  at TestBail$BailSimpleLexer.recover(TestBail.java:9)
  at org.antlr.v4.runtime.Lexer.nextToken(Lexer.java:165)
  at org.antlr.v4.runtime.BufferedTokenStream.fetch(BufferedT...Stream.java:139)
  at org.antlr.v4.runtime.BufferedTokenStream.sync(BufferedT...Stream.java:133)
  at org.antlr.v4.runtime.CommonTokenStream.setup(CommonTokenStream.java:129)
  at org.antlr.v4.runtime.CommonTokenStream.LT(CommonTokenStream.java:111)
```

```
at org.antlr.v4.runtime.Parser.enterRule(Parser.java:424)
at SimpleParser.prog(SimpleParser.java:68)
at TestBail.main(TestBail.java:23)
...
```

The parser also bails out at the first syntax error (a missing class name, in this case).

```
⇛ $ java TestBail
⇛ class { }
⇛ EOF
❮ Exception in thread "main" java.lang.RuntimeException:
    org.antlr.v4.runtime.InputMismatchException
...
```

To demonstrate the flexibility of the ANTLRErrorStrategy interface, let's finish up by altering how the parser reports errors. To alter the standard message, "no viable alternative at input X," we can override reportNoViableAlternative() and change the message to something different.

errors/MyErrorStrategy.java

```
import org.antlr.v4.runtime.*;
public class MyErrorStrategy extends DefaultErrorStrategy {
    @Override
    public void reportNoViableAlternative(Parser parser,
                                          NoViableAltException e)
        throws RecognitionException
    {
        // ANTLR generates Parser subclasses from grammars and
        // Parser extends Recognizer. Parameter parser is a
        // pointer to the parser that detected the error
        String msg = "can't choose between alternatives"; // nonstandard msg
        parser.notifyErrorListeners(e.getOffendingToken(), msg, e);
    }
}
```

Remember, though, that if all we want to do is change *where* error messages go, we can specify an ANTLRErrorListener as we did in Section 9.2, *Altering and Redirecting ANTLR Error Messages*, on page 153. To learn how to completely override how ANTLR generates code for catching exceptions, see *Catching Exceptions*, on page 266.

We've covered all of the important error reporting and recovery facilities within ANTLR. Because of ANTLRErrorListener and ANTLRErrorStategy interfaces, we have great flexibility over where error messages go, what those messages are, and how the parser recovers from errors.

In the next chapter, we're going to learn how to embed code snippets called *actions* directly within the grammar.

CHAPTER 10

Attributes and Actions

So far, we've isolated our application-specific code to parse-tree walkers, which means that our code has always executed after parsing is complete. As we'll see in the next few chapters, some language applications require executing application-specific code while parsing. To do that, we need the ability to inject code snippets, called *actions*, directly into the code ANTLR generates. Our first goal, then, is to learn how to embed actions in parsers and lexers and to figure out what we can put in those actions.

Keep in mind that, in general, it's a good idea to avoid entangling grammars and application-specific code. Grammars without actions are easier to read, aren't tied to a particular target language, and aren't tied to a specific application. Still, embedded actions can be useful for three reasons.

- *Simplicity*: Sometimes it's easier just to stick in a few actions and avoid creating a tree listener or visitor.
- *Efficiency*: In resource-critical applications, we might not want to waste the time or memory needed to build a parse tree.
- *Predicated parsing*: In rare cases, we can't parse properly without referencing data collected previously in the input stream. Some grammars need to build up a symbol table and then recognize future input differently, depending on whether an identifier is, say, a type or a method. We'll explore this in Chapter 11, *Altering the Parse with Semantic Predicates*, on page 189.

Actions are arbitrary chunks of code written in the target language (the language in which ANTLR generates code) enclosed in {...}. We can do whatever we want in these actions as long as they are valid target language statements. Typically, actions operate on the attributes of tokens and rule references. For example, we can ask for the text of a token or the text matched by an entire

rule invocation. Using data derived from token and rule references, we can print things out and perform arbitrary computations. Rules also allow parameters and return values so we can pass data around between rules.

We're going to learn about grammar actions by exploring three examples. First, we're going to build a calculator with the same functionality as that in Section 7.4, *Labeling Rule Alternatives for Precise Event Methods*, on page 117. Second, we'll add some actions to the CSV grammar (from Section 6.1, *Parsing Comma-Separated Values*, on page 84) to explore rule and token attributes. In the third example, we'll learn about actions in lexer rules by building a grammar for a language whose keywords aren't known until runtime.

It's time to get our hands dirty, starting with an action-based calculator implementation.

10.1 Building a Calculator with Grammar Actions

Let's revisit the expression grammar from Section 4.2, *Building a Calculator Using a Visitor*, on page 38 to learn about actions. In that section, we built a calculator using a tree visitor that evaluated expressions such as the following:

actions/t.expr
```
x = 1
x
x+2*3
```

Our goal here is to reproduce that same functionality, but without using a visitor and without even building a parse tree. Moreover, we'll employ a little trick to make it interactive, meaning we get results when we hit `Return`, not at the end of the input. Our examples so far have scarfed up the entire input and then processed the resulting parse trees.

As we go through this section, we're going to learn how to put generated parsers into packages, define parser fields and methods, insert actions within rule alternatives, label grammar elements for use within actions, and define rule return values.

Using Actions Outside of Grammar Rules

Outside of grammar rules, there are two kinds of things we want to inject into generated parsers and lexers: package/import statements and class members like fields and methods.

Here is an idealized code generation template that illustrates where we want to inject code snippets for, say, the parser:

```
<header>
public class <grammarName>Parser extends Parser {
        <members>
        ...
}
```

To specify a header action, we use `@header {...}` in our grammar. To inject fields or methods into the generated code, we use `@members {...}`. In a combined parser/lexer grammar, these named actions apply to both the parser and the lexer. (ANTLR option `-package` lets us set the package without a `header` action.) To restrict an action to the generated parser or lexer, we use `@parser::`*name* or `@lexer::`*name*.

Let's see what these look like for our calculator. The expression grammar starts with a grammar declaration like before, but now we're going to declare that the generated code lives in a Java package. We'll also need to import some standard Java utility classes.

actions/tools/Expr.g4
```
grammar Expr;

@header {
package tools;
import java.util.*;
}
```

The previous calculator's `EvalVisitor` class had a `memory` field that stored name-value pairs to implement variable assignments and references. We'll put that in our `members` action. To reduce clutter in the grammar, let's also define a convenience method called `eval()` that performs an operation on two operands. Here's what the complete `members` action looks like:

actions/tools/Expr.g4
```
@parser::members {
    /** "memory" for our calculator; variable/value pairs go here */
    Map<String, Integer> memory = new HashMap<String, Integer>();

    int eval(int left, int op, int right) {
        switch ( op ) {
            case MUL : return left * right;
            case DIV : return left / right;
            case ADD : return left + right;
            case SUB : return left - right;
        }
        return 0;
    }
}
```

With that infrastructure in place, let's see how to use these parser class members inside actions among the rule elements.

Embedding Actions Within Rules

In this section, we're going to learn how to embed actions in the grammar in order to generate some output, update data structures, and set rule return values. We'll also look at how ANTLR wraps up rule parameters, return values, and other attributes of a rule invocation into instances of ParserRuleContext subclasses.

The Basics

Rule stat recognizes expressions, variable assignments, and blank lines. Because we do nothing upon a blank line, stat needs only two actions.

actions/tools/Expr.g4

```
stat:   e NEWLINE           {System.out.println($e.v);}
    |   ID '=' e NEWLINE    {memory.put($ID.text, $e.v);}
    |   NEWLINE
    ;
```

Actions execute after the preceding grammar element and before the next one. In this case, the actions appear at the end of the alternatives, so they execute after the parser matches the entire statement. When stat sees an expression followed by NEWLINE, it should print the value of the expression. When stat sees a variable assignment, it should store the name-value pair in field memory.

The only unfamiliar syntax in those actions are $e.v and $ID.text. In general, $x.y refers to attribute y of element x, where x is either a token reference or a rule reference. Here, $e.v refers to the return value from calling rule e. (We'll see why it's called v in a second.) $ID.text refers to the text matched by the ID reference.

If ANTLR doesn't recognize the y component, it doesn't translate it. In this case, text is a known attribute of a token, and ANTLR translates it to getText(). We could also use $ID.getText() to get the same thing. For a complete list of the attributes for rules and tokens, see Section 15.4, *Actions and Attributes*, on page 271.

Turning to rule e now, let's see what it looks like with embedded actions. The basic idea is to mimic the EvalVisitor functionality by inserting code snippets directly into the grammar as actions.

actions/tools/Expr.g4
```
e returns [int v]
    : a=e op=('*'|'/') b=e  {$v = eval($a.v, $op.type, $b.v);}
    | a=e op=('+'|'-') b=e  {$v = eval($a.v, $op.type, $b.v);}
    | INT                   {$v = $INT.int;}
    | ID
      {
      String id = $ID.text;
      $v = memory.containsKey(id) ? memory.get(id) : 0;
      }
    | '(' e ')'             {$v = $e.v;}
    ;
```

A number of interesting things are going on in this example. The first thing we see is the return value specification for an integer, v. That's why stat's actions refer to $e.v. ANTLR return values differ from Java return values in that we get to name them and we can have more than one.

Next, we see labels on e rule references and on the operator subrules such as op=('*'|'/'). Labels refer to Token or ParserRuleContext objects derived from matching a token or invoking a rule.

Before turning to the action contents, it's worth looking at where ANTLR stores things like return values and labels. It'll make following the ANTLR-generated code easier when source-level debugging.

One Rule Context Object to Bind Them All

In Section 2.4, *Building Language Applications Using Parse Trees*, on page 16, we learned that ANTLR implements parse-tree nodes with rule context objects. Each rule invocation creates and returns a rule context object, which holds all of the important information about the recognition of a rule at a specific location in the input stream. For example, rule e creates and returns EContext objects.

```
public final EContext e(...) throws RecognitionException {...}
```

Naturally, a rule context object is a very handy place to put rule-specific entities. The first part of EContext looks like this:

```
public static class EContext extends ParserRuleContext {
        public int v;       // rule e return value from "returns [int v]"
        public EContext a;  // label a on (recursive) rule reference to e
        public Token op;    // label on operator sub rules like ('*'|'/')
        public EContext b;  // label b on (recursive) rule reference to e
        public Token INT;   // reference to INT matched by 3rd alternative
        public Token ID;    // reference to ID matched by 4th alternative
        public EContext e;  // reference to context object from e invocation
        ...
}
```

Labels always become fields in the rule context object, but ANTLR doesn't always generate fields for alternative elements such as ID, INT, and e. ANTLR generates fields for them only if they're referenced by actions in the grammar (as e's actions do). ANTLR tries to reduce the number of context object fields.

Now we have all the pieces in place, so let's analyze the contents of the actions among the alternatives of rule e.

Computing Return Values

All of the actions in e set the return value with assignment $v = ...;. This sets the return value but does not perform a return from the rule function. (Don't use a return statement in your actions because it will make the parser go insane.) Here is the action used by the first two alternatives:

```
$v = eval($a.v, $op.type, $b.v);
```

This action computes the value of the subexpression and sets the return value for e. The arguments to eval() are the return values from the two references to e, $a.v and $b.v, and the token type of the operator matched by the alternative, $op.type. $op.type will always be the token type for one of the arithmetic operators. Notice that we can reuse labels (as long as they refer to the same kind of thing). The second alternative reuses labels a, b, and op.

The third alternative's action uses $INT.int to access the integer value of the text matched by the INT token. This is just shorthand for Integer.valueOf($INT.text). The embedded action is much simpler than the equivalent visitor visitInt() method (but at the cost of entangling application-specific code with the grammar).

tour/EvalVisitor.java

```
/** INT */
@Override
public Integer visitInt(LabeledExprParser.IntContext ctx) {
    return Integer.valueOf(ctx.INT().getText());
}
```

The fourth alternative recognizes a variable reference and sets e's return value to the value stored in memory, if we've stored a value for that name. This action uses the Java ?: operator, but we could've just as easily used an if-then-else Java statement. We can put anything into an action that would work as a body of a Java method.

Finally, the $v = $e.v; action in the last alternative sets the return value to the result of the expression matched in parentheses. We're just passing the return value through. The value of (3) is 3.

That's it for the grammar and action code. Now, let's figure out how to build an interactive driver for our calculator.

Building an Interactive Calculator

Before exploring the details of building an interactive tool, let's give it a whirl by building and testing the grammar and Calc.java test rig. Because we put statement package tools; in the header action, we need to put the generated Java code in a directory called tools. (This reflects the standard Java relationship between package and directory structure.) That means we need to either run ANTLR from tools or run it from the directory above tools with path tools/Expr.g4 instead of just Expr.g4.

```
$ antlr4 -no-listener tools/Expr.g4  # gen parser w/o listener into tools
$ javac -d . tools/*.java            # compile, put .class files in tools
```

To try it, we run Calc using its fully qualified name.

```
⇛ $ java tools.Calc
⇛ x = 1
⇛ x
❮ 1
⇛ x+2*3
❮ 7
⇛ EOF
```

You'll notice that the calculator immediately responds with an answer when you hit Return. Because ANTLR reads the entire input (usually into a big buffer) by default, we have to pass input line by line to the parser to make it interactive. Each line represents a complete expression. (If you need to handle expressions that can span multiple lines, see *Fun with Python Newlines*, on page 214.) In the main() method, here's how we get the first expression:

actions/tools/Calc.java

```
BufferedReader br = new BufferedReader(new InputStreamReader(is));
String expr = br.readLine();              // get first expression
int line = 1;                             // track input expr line numbers
```

To maintain memory field values across expressions, we need a single shared parser for all input lines.

actions/tools/Calc.java

```
ExprParser parser = new ExprParser(null); // share single parser instance
parser.setBuildParseTree(false);          // don't need trees
```

As we read in a line, we'll create a new token stream and pass it to the shared parser.

actions/tools/Calc.java
```
while ( expr!=null ) {              // while we have more expressions
    // create new lexer and token stream for each line (expression)
    ANTLRInputStream input = new ANTLRInputStream(expr+"\n");
    ExprLexer lexer = new ExprLexer(input);
    lexer.setLine(line);            // notify lexer of input position
    lexer.setCharPositionInLine(0);
    CommonTokenStream tokens = new CommonTokenStream(lexer);
    parser.setInputStream(tokens); // notify parser of new token stream
    parser.stat();                  // start the parser
    expr = br.readLine();           // see if there's another line
    line++;
}
```

So, now we know how to build an interactive tool, and we have a pretty good idea about how to place and use embedded actions. Our calculator used a header action to specify a package and a members action to define two parser class members. We used actions within the rules to compute and return subexpression values as a function of token and rule attributes. In the next section, we're going to see some more attributes and identify a few more action locations.

10.2 Accessing Token and Rule Attributes

Let's use the CSV grammar from Section 6.1, *Parsing Comma-Separated Values*, on page 84 as a foundation for exploring some more action-related features. We're going to build an application that creates a map from column name to field for each row and prints out information gained from parsing the data. Our goal here is really just to learn more about rule-related actions and attributes.

First, let's take a look at how to define local variables using the locals section. As with parameters and return values, the declarations in a locals section become fields in the rule context object. Because we get a new rule context object for every rule invocation, we get a new copy of the locals as we'd expect. The following augmented version of rule file does a number of interesting things, but let's start by focusing on what it does with locals.

actions/CSV.g4
```
/** Derived from rule "file : hdr row+ ;" */
file
locals [int i=0]
     : hdr ( rows+=row[$hdr.text.split(",")] {$i++;} )+
       {
       System.out.println($i+" rows");
       for (RowContext r : $rows) {
           System.out.println("row token interval: "+r.getSourceInterval());
       }
       }
     ;
```

Rule file defines a local variable, i, and uses it to count how many rows there are in the input using action $i++;. To reference local variables, don't forget the $ character prefix, or the compiler will complain about undefined variables. ANTLR translates $i to _localctx.i; there is no i local variable in the rule function generated for file.

Next, let's take a look at the call to rule row. Rule invocation row[$hdr.text.split(",")] illustrates that we use square brackets instead of parentheses to pass parameters to rules (ANTLR uses parentheses for subrule syntax). Argument expression $hdr.text.split(",") splits the text matched by the hdr rule invocation to get an array of strings needed by row.

Let's break that apart. $hdr is a reference to the sole hdr invocation and evaluates to its HdrContext object. We don't need to label the hdr reference in this case (like h=hdr) because $hdr is unique. $hdr.text, then, is the text matched for the header row. We split the comma-separated header columns using the standard Java String.split() method, which returns an array of strings. We'll see shortly that rule row takes an array-of-string parameter.

The call to row also introduces a new kind of label that uses += instead of the = label operator. Rather than tracking a single value, label rows is a list of all RowContext objects returned from all row invocations. After printing out the number of rows, the final action in file has a loop that iterates through the RowContext objects. Each time through the loop, it prints out the range of token indexes matched by the row invocation (using getSourceInterval()).

That loop uses r, not $r, because r is a local variable created within a Java action. ANTLR can see only those local variables defined with the locals keyword, not in arbitrary user-defined embedded actions. The difference is that the parse tree node for rule file would define field i but not r.

Turning to rule hdr now, let's just print out the header row. We could do that by referencing $row.text, which is the text matched by the row rule reference. Alternatively, we can ask for the text of the surrounding rule with $text.

actions/CSV.g4

```
hdr : row[null] {System.out.println("header: '"+$text.trim()+"'");} ;
```

In this case, it will also be the text matched by row because that is all there is in that rule.

Now let's figure out how to convert every row of data into a map from column name to value with actions in rule row. To begin with, row takes the array of column names as a parameter and returns the map. Next, to move through the column names array, we'll need a local variable, col. Before parsing the

row, we need to initialize the return value map, and, for fun, let's print out the map after row finishes. All of that goes into the header for the rule.

actions/CSV.g4

```
/** Derived from rule "row : field (',' field)* '\r'? '\n' ;" */
row[String[] columns] returns [Map<String,String> values]
locals [int col=0]
@init {
    $values = new HashMap<String,String>();
}
@after {
    if ($values!=null && $values.size()>0) {
        System.out.println("values = "+$values);
    }
}
```

The init action happens before anything is matched in the rule, regardless of how many alternatives there are. Similarly, the after action happens after the rule matches one of the alternatives. In this case, we could express the after action functionality by putting the print statement in an action at the end of row's outer alternative.

With everything set up, we can collect the data and fill the map.

actions/CSV.g4

```
// rule row cont'd...
    :   field
        {
        if ($columns!=null) {
            $values.put($columns[$col++].trim(), $field.text.trim());
        }
        }
        (   ',' field
            {
            if ($columns!=null) {
                $values.put($columns[$col++].trim(), $field.text.trim());
            }
            }
        )* '\r'? '\n'
    ;
```

The meaty parts of the actions store the field value at the column name in the values map using $values.put(...). The first parameter gets the column name, increments the column count, and trims away the whitespace from the name: $columns[$col++].trim(). The second parameter trims the text of the most recently matched field: $field.text.trim(). (Both actions in row are identical, so it might be a good idea to factor that out into a method in a members action.)

Everything else in CSV.g4 is familiar to us, so let's move on to building this thing and giving it a try. We don't need to write a special test rig because of grun, so we can just generate the parser and compile it.

```
$ antlr4 -no-listener CSV.g4   # again, we won't use a listener
$ javac CSV*.java
```

Here's some CSV data we can use:

actions/users.csv

```
User,  Name,    Dept
parrt, Terence, 101
tombu, Tom,     020
bke, Kevin,     008
```

And, here is the output:

```
$ grun CSV file users.csv
header: 'User,  Name,    Dept'
values = {Name=Terence, User=parrt, Dept=101}
values = {Name=Tom, User=tombu, Dept=020}
values = {Name=Kevin, User=bke, Dept=008}
3 rows
row token interval: 6..11
row token interval: 12..17
row token interval: 18..23
```

Rule hdr prints out the first line of output, and the three calls to row print out the values = ... lines. Back in rule file, the action prints out the number of rows and the token intervals associated with each row of data.

At this point, we have a very good handle on the use of embedded actions, both inside and outside of rules. We also know quite a bit about rule attributes. On the other hand, both the calculator and CSV example use actions exclusively in the parser rules. It turns out actions can be very useful in lexer rules as well. We'll explore that next by seeing how to handle a large or dynamic set of keywords.

10.3 Recognizing Languages Whose Keywords Aren't Fixed

To explore actions embedded in lexer rules, let's build a grammar for a contrived programming language whose keywords can change dynamically (from run to run). This is not as unusual as it sounds. For example, in version 5, Java added the keyword enum, so the same compiler must be able to enable and disable a keyword depending on the -version option.

Perhaps a more common use would be dealing with languages that have huge keyword sets. Rather than making the lexer match all of the keywords individually (as separate rules), we can make a catchall ID rule and then look up the identifier

in a keywords table. If the lexer finds a keyword, we can specifically set the token type from a generic ID to the token type for that keyword.

Before we get to the ID rule and the keyword lookup mechanism, let's take a look at a statement rule that references keywords.

actions/Keywords.g4
```
stat:   BEGIN stat* END
    |   IF expr THEN stat
    |   WHILE expr stat
    |   ID '=' expr ';'
        ;

expr:   INT | CHAR ;
```

ANTLR will implicitly define token types for the keywords BEGIN, END, and so on. But, ANTLR will warn us that there is no corresponding lexical definition for these token types.

```
$ antlr4 Keywords.g4
warning(125): Keywords.g4:31:8: implicit definition of token BEGIN in parser
...
```

To hush this warning, let's be explicit.

actions/Keywords.g4
```
// explicitly define keyword token types to avoid implicit def warnings
tokens { BEGIN, END, IF, THEN, WHILE }
```

In the generated KeywordsParser, ANTLR defines token types like this:

```
public static final int ID=3, BEGIN=4, END=5, IF=6, ... ;
```

Now that we've defined our token types, let's look at the grammar declaration and a header action, which imports Map and HashMap.

actions/Keywords.g4
```
grammar Keywords;
@lexer::header {    // place this header action only in lexer, not the parser
import java.util.*;
}
```

We'll use a Map from keyword name to integer token type for the keywords table, and we can define the mappings inline using a Java instance initializer (the inner set of curly braces).

actions/Keywords.g4
```
@lexer::members {   // place this class member only in lexer
Map<String,Integer> keywords = new HashMap<String,Integer>() {{
    put("begin", KeywordsParser.BEGIN);
    put("end",   KeywordsParser.END);
    put("if",    KeywordsParser.IF);
```

```
    put("then",  KeywordsParser.THEN);
    put("while", KeywordsParser.WHILE);
}};
}
```

With all of our infrastructure in place, let's match identifiers as we've done many times before, but with an action that flips the token type appropriately.

actions/Keywords.g4

```
ID  :   [a-zA-Z]+
        {
        if ( keywords.containsKey(getText()) ) {
            setType(keywords.get(getText())); // reset token type
        }
        }
    ;
```

Here we use Lexer's getText() method to get the text of the current token. We use it to see whether the identifier exists within keywords. If it does, then we reset the token type from ID to the keyword's token type value.

While we're having fun in the lexer, let's figure out how to change the text of a token. This is useful for stripping single and double quotes from character and string literals. Usually, a language application would want just the text inside the quotes. Here's how to override the text of a token using setText():

actions/Keywords.g4

```
/** Convert 3-char 'x' input sequence to string x */
CHAR:   '\'' . '\'' {setText( String.valueOf(getText().charAt(1)) );} ;
```

If we wanted to get really crazy, we could even specify the Token object to return from the lexer using setToken(). This is a way to return custom token implementations. Another way is to override Lexer's emit() method.

We're ready to try our little language. The behavior we expect is to have keywords differentiated from regular identifiers. In other words, x = 34; should work, but if = 34; should not because if is a keyword. Let's run ANTLR, compile the generated code, and try it on the valid assignment.

```
⇛ $ antlr4 -no-listener Keywords.g4
⇛ $ javac Keywords*.java
⇛ $ grun Keywords stat
⇛ x = 34;
⇛ EOF
```

No problem; there are no errors. However, the parser gives a syntax error for the assignment that tries to use if as an identifier. It also accepts a valid if statement without error.

```
⇛ $ grun Keywords stat
⇛ if = 34;
⇛ EOF
❮ line 1:3 extraneous input '=' expecting {CHAR, INT}
  line 2:0 mismatched input '<EOF>' expecting THEN
⇛ $ grun Keywords stat
⇛ if 1 then i = 4;
⇛ EOF
```

If you are unlucky enough to be building a parser for a language that, in some contexts, allows keywords as identifiers, see *Treating Keywords As Identifiers*, on page 209.

Actions are less commonly needed in the lexer than the parser, but they are still useful in situations like this where we need to alter token types or the token text. We could also alter tokens by looking at the token stream after the fact instead of with actions while tokenizing the input.

In this chapter, we learned how to embed application code within grammars using actions among the rule elements and outside rules using named actions such as `header` and `members`. We also saw how to define and reference rule parameters and return values. Along the way, we also used token attributes such as `text` and `type`. Taken together, these action-related features let us customize the code ANTLR generates.

Again, try to avoid grammar actions when you can because actions tie a grammar to a particular programming language target. Actions also tie a grammar to a specific application. That said, you might not care about these issues because your company always programs in a single language and your grammar is specific to a particular application anyway. In that situation, it could make sense to embed actions directly in the grammar for simplicity or efficiency reasons (no parse tree construction). Most importantly, some parsing problems require runtime tests to recognize the input properly. In the next chapter, we're going to explore arbitrary Boolean expressions called *semantic predicates* that can dynamically turn alternatives on and off.

CHAPTER 11

Altering the Parse with Semantic Predicates

In the previous chapter, we learned how to embed actions within a grammar in order to execute application-specific code on-the-fly during the parse. Those actions did not affect the operation of the parser in any way, much as log statements do not affect the surrounding program. Our embedded actions just computed values or printed things out. In rare cases, however, altering the parse with embedded code is the only way to reasonably recognize the input sentences of a language.

In this chapter, we're going to learn about special actions, `{...}?`, called *semantic predicates* that let us selectively deactivate portions of a grammar at runtime. Predicates are Boolean expressions that have the effect of reducing the number of choices that the parser sees. Believe it or not, selectively reducing choice actually increases the power of the parser!

There are two common use cases for semantic predicates. First, we might need a parser to handle multiple, slightly different versions (dialects) of the same language. For example, the syntax of a database vendor's SQL evolves over time. To build a database front end for that vendor, we'd need to support different versions of the same SQL. Similarly, the Gnu C compiler, `gcc`, has to deal with ANSI C as well as its own dialect that adds things like the awesome computed goto. Moreover, semantic predicates let us choose between dialects at runtime with a command-line switch or other dynamic mechanism.

The second use case involves resolving grammar ambiguities (discussed in Section 2.2, *Implementing Parsers*, on page 11). In some languages, the same syntactic construct can mean different things, and predicates give us a way to choose between multiple interpretations of the same input phrase. For example, in good ol' Fortran, `f(i)` could be an array reference or a function call, depending on what `f` was defined to be—the syntax was the same. A compiler had to look up the identifier in a symbol table to properly interpret the input.

Semantic predicates give us a way to turn off improper interpretations based upon what we find in a symbol table. This leaves the parser with only a single choice, the proper interpretation.

We're going to learn about semantic predicates by working through examples taken from Java and C++. Along the way, we'll pick up most of the details, but you can check Section 15.7, *Semantic Predicates*, on page 286 in the reference chapter for a discussion of the fine print. Armed with embedded actions and predicates, we'll be sufficiently prepared to tackle some formidable language problems in the next chapter.

11.1 Recognizing Multiple Language Dialects

For our first lesson, we're going to learn how to use semantic predicates to deactivate parts of a Java grammar. The effect will be to recognize different dialects, according to results of evaluating Boolean expressions on-the-fly. In particular, we're going to see how the same parser can switch between allowing and disallowing enumerated types.

The Java language has been extended over the years to include new constructs. For example, prior to Java 5, the following declaration was invalid:

predicates/Temp.java
```
enum Temp { HOT, COLD }
```

Rather than building separate compilers for the slightly different dialects, the Java compiler `javac` has a `-source` option. Here's what happens when we try to compile that `enum` with Java version 1.4:

```
$ javac -source 1.4 Temp.java
Temp.java:1: enums are not supported in -source 1.4
(use -source 5 or higher to enable enums)
enum Temp { HOT, COLD }
^
1 error
$ javac Temp.java  # javac assumes the latest dialect; compiles fine.
```

Introducing enumerated types flipped `enum` from an identifier to a keyword, causing a backward-compatibility issue. Lots of legacy code uses `enum` as a variable like this: `int enum;`. With an option on the compiler to recognize the earlier dialect, we don't have to alter ancient code just to compile it again.

To get a taste for how `javac` handles multiple dialects, let's build a grammar that recognizes a tiny piece of Java: just `enum` declarations and assignment statements. The goal is to create a grammar that properly recognizes pre–and post–Java 5 languages, but *not both* at the same time. For example, using `enum` as both a keyword and an identifier should be invalid.

```
enum enum { HOT, COLD }    // syntax error in any Java version
```

Let's start our solution by looking at the core of a grammar that recognizes our minimal Java subset and then figure out how to handle the enum keyword.

predicates/Enum.g4

```
grammar Enum;
@parser::members {public static boolean java5;}

prog:   (   stat
        |   enumDecl
        )+
    ;

stat:   id '=' expr ';' {System.out.println($id.text+"="+$expr.text);} ;

expr
    :   id
    |   INT
    ;
```

We're already familiar with these grammatical constructs and actions, so let's move on to enum declarations.

```
enumDecl
    :   'enum' name=id '{' id (',' id)* '}'
        {System.out.println("enum "+$name.text);}
    ;
```

That rule recognizes the (simplified) syntax of an enumerated type, but there's nothing to suggest that enums are sometimes illegal. And that brings us to the heart of the matter: turning alternatives on and off with semantic predicates.

predicates/Enum.g4

```
enumDecl
    :   {java5}? 'enum' name=id '{' id (',' id)* '}'
        {System.out.println("enum "+$name.text);}
    ;
```

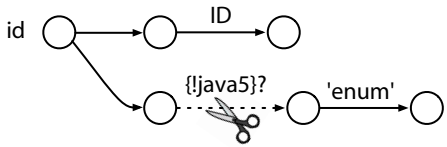
Predicate {java5}? evaluates to true or false at runtime, deactivating that alternative when java5 is false.

You'll notice that we're using rule id instead of just token ID like we usually do. That's because the notion of an identifier includes enum when recognizing pre–Java 5. (Our lexer returns enum as a keyword, not an identifier.) To express that choice, we need a rule with a semantic predicate.

predicates/Enum.g4

```
id  :   ID
    |   {!java5}? 'enum'
    ;
```

The {!java5}? predicate allows enum to act as an identifier only when not in Java 5 mode. It literally deactivates the second alternative when java5 is true. Internally, ANTLR parsers view rule id using a graph data structure, sort of like this:



The scissors icon indicates that the parser snips that branch from the graph when !java5 evaluates to false (when java5 is true). Notice that because the predicates are mutually exclusive, enum declarations and enum-as-identifiers are mutually exclusive constructs.

We could use grun to test the grammar, but we need a test rig that can flip between Java dialects. Here are the relevant bits from TestEnum that support a -java5 option for turning on Java 5 mode:

predicates/TestEnum.java

```
int i = 0;
EnumParser.java5 = false; // assume non-Java5 mode by default
if ( args.length>0 && args[i].equals("-java5") ) {
        EnumParser.java5 = true;
        i++;
}
```

Now let's build and compile everything.

```
$ antlr4 -no-listener Enum.g4
$ javac Enum*.java TestEnum.java
```

Let's start with pre–Java 5 mode and make sure it allows enum as an identifier and that it doesn't allow enumerated types.

```
⇛ $ java TestEnum
⇛ enum = 0;
⇛ EOF
❮ enum=0

⇛ $ java TestEnum
⇛ enum Temp { HOT, COLD }
⇛ EOF
❮ line 1:0 no viable alternative at input 'enum'
```

Java 5 mode, in contrast, shouldn't consider enum to be an identifier, but it should allow enumerated types.

```
⇛ $ java TestEnum -java5
⇛ enum = 0;
⇛ EOF
❮ line 1:0 no viable alternative at input 'enum'

⇛ $ java TestEnum -java5
⇛ enum Temp { HOT, COLD }
⇛ EOF
❮ enum Temp
```

Everything checks out, but let's take a look at predicate placement before moving on. Predicates work by activating or deactivating everything that could be matched after passing through the predicate. That means we don't technically need to put the {java5}? predicate in enumDecl proper. We could drag it out and put it in front of the call to that rule instead.

```
prog:   (   {java5}? enumDecl
        |   stat
        )+
    ;
```

They are functionally equivalent, and their placement in this case is a matter of style. The key is that the parser must encounter a predicate somewhere in the first alternative of the (...)+ subrule before reaching the 'enum' token reference in enumDecl.

And that's how to build a grammar that supports multiple dialects using a runtime Boolean switch. To build a real Java grammar, you could incorporate these predicates into the appropriate rules of a grammar, which we've called enumDecl and id here.

As with embedded actions, semantic predicates are also occasionally useful in the lexer.

11.2 Deactivating Tokens

In this section, we're going to solve the same problem again, but this time using predicates in the lexer instead of the parser. The idea is that predicates in the lexer activate and deactivate *tokens* rather than *phrases* in the language. Our approach will be to deactivate enum as a keyword and match it as a regular identifier in pre–Java 5 mode. In Java 5 mode, we want to separate enum out as its own keyword token. This simplifies the parser considerably because it can match an identifier just by referencing the usual ID token, rather than an id rule.

predicates/Enum2.g4
```
stat:   ID '=' expr ';' {System.out.println($ID.text+"="+$expr.text);} ;

expr:   ID
    |   INT
    ;
```

The lexer sends an ID only when it's appropriate for the current dialect. To pull this off, we need just one predicate in a lexical rule that matches enum.

predicates/Enum2.g4
```
ENUM:   'enum' {java5}? ; // must be before ID
ID  :   [a-zA-Z]+ ;
```

Notice that the predicate appears on the right edge of the lexical rule instead of on the left, like we did for parser alternatives. That's because parsers predict what's coming down the road and need to test the predicates before matching alternatives.

Lexers, on the other hand, don't predict alternatives. They just look for the longest match and make decisions after they've seen the entire token. (We'll learn more about this in the reference chapter, specifically, in Section 15.7, *Semantic Predicates*, on page 286.)

When java5 is false, the predicate deactivates rule ENUM. When it's true, however, both ENUM and ID match character sequence e-n-u-m. Those two rules are ambiguous for that input. ANTLR always resolves lexical ambiguities in favor of the rule specified first, in this case ENUM. If we had reversed the rules, the lexer would always match e-n-u-m as an ID. It wouldn't matter whether ENUM was activated or deactivated.

The beauty of this predicated lexer solution is that we don't need a predicate in the parser for deactivating the enum construct when not in Java 5 mode.

predicates/Enum2.g4
```
// No predicate needed here because 'enum' token undefined if !java5
enumDecl
    :   'enum' name=ID '{' ID (',' ID)* '}'
        {System.out.println("enum "+$name.text);}
    ;
```

Token 'enum', referenced at the start of the alternative, is looking for a specific keyword token. The lexer can present that to the parser only in Java 5 mode, so enumDecl will never match unless java5 is true.

Let's verify now that our lexer-based solution correctly recognizes constructs in the two dialects. In non-Java-5 mode, enum is an identifier.

```
⇛ $ antlr4 -no-listener Enum2.g4
⇛ $ javac Enum2*.java TestEnum2.java
⇛ $ java TestEnum2
⇛ enum = 0;
⇛ EOF
❮ enum=0
```

Because enum is an identifier, not a keyword token, the parser will never attempt to match enumDecl. It has no choice but to treat enum Temp { HOT, COLD } as an assignment, leading to syntax errors.

```
⇛ $ java TestEnum2
⇛ enum Temp { HOT, COLD }
⇛ EOF
❮ line 1:5 missing '=' at 'Temp'
  line 1:15 mismatched input ',' expecting '='
  line 1:22 mismatched input '}' expecting '='
```

In this case, ANTLR's error recovery realizes it doesn't have a valid assignment and scarfs tokens until it finds something that can start an assignment.

In Java 5 mode, an assignment to enum is invalid, but an enumerated type is valid.

```
⇛ $ java TestEnum2 -java5
⇛ enum = 0;
⇛ EOF
❮ line 1:5 mismatched input '=' expecting ID
⇛ $ java TestEnum2 -java5
⇛ enum Temp { HOT, COLD }
⇛ EOF
❮ enum Temp
```

If we wanted to avoid predicates, which can slow down the lexer, we could do away with the ENUM rule altogether and match enum always as an identifier. Then we'd flip the token type appropriately like we did in Section 10.3, *Recognizing Languages Whose Keywords Aren't Fixed*, on page 185.

```
ID  :   [a-zA-Z]+
        {if (java5 && getText().equals("enum")) setType(Enum2Parser.ENUM);}
    ;
```

We would also need a token definition for ENUM.

```
tokens { ENUM }
```

It's a good idea to avoid embedding predicates in the parser when possible for efficiency and clarity reasons. Instead, I recommend choosing one of the lexer-based solutions in this section to support the Java dialects related to

enum. Note that predicates slow the lexer down as well, so try to do without them entirely.

That's it for the basic syntax and usage of semantic predicates in the parser and the lexer. Predicates offer a straightforward way to selectively deactivate parts of a grammar, which lets us recognize dialects of the same language using the same grammar. Moreover, we can switch between dialects dynamically by flipping the value of a Boolean expression. Now let's investigate the second major use case: using predicates to resolve ambiguous input phrases in the parser.

11.3 Recognizing Ambiguous Phrases

We've just seen how to strip away parts of a grammar based upon a simple Boolean variable. It wasn't that the grammar matched the same input in multiple ways; we simply wanted to turn off certain language constructs. Now our goal is to force the parser to deactivate all but one interpretation of an ambiguous input phrase. Using our maze analogy, a maze and passphrase are ambiguous when we can follow multiple paths through the maze to the exit using that single passphrase. Predicates are like doors on path forks that we can open and close to direct movement through the maze.

Language Ambiguities Are Bad...Umkay?

Wise language designers deliberately avoid ambiguous constructs because they make it hard to read code. For example, f[0] in Ruby is either a reference to the first element of array f or a function call to f() that returns an array, which we then index. To make things even more fun, f [0] with a space before [0] passes an array with 0 in it to function f() as an argument. This all happens because parentheses are optional in Ruby for function calls. Ruby aficionados currently recommend using parentheses because of these very ambiguities.

Before we begin, let me point out that having more than one way to match an input phrase in a grammar is almost always a grammar bug. In most languages, the syntax alone uniquely dictates how to interpret all valid sentences. (See the sidebar *Language Ambiguities Are Bad...Umkay?*, on page 196.) That means our grammars should match each input stream in just one way. If we find multiple interpretations, we should rewrite the grammar to strip out the invalid interpretation(s).

That said, there are phrases in some languages where syntax alone just isn't enough to identify the meaning. Grammars for these languages will necessarily be ambiguous, but the meaning of syntactically ambiguous phrases will

be clear given sufficient context, such as how identifiers are defined (for example, as types or methods). We'll need predicates to properly select an interpretation for each ambiguous phrase by asking context questions. If a predicate successfully resolves a grammar ambiguity for an input phrase, then we say that the phrase is *context-sensitive*.

In this section, we're going to explore some ambiguities in the nooks and crannies of C++. As far as I can tell, C++ is the most difficult programming language to parse accurately and precisely. We'll start with function calls vs. constructor-style type casts and then look at declarations vs. expressions.

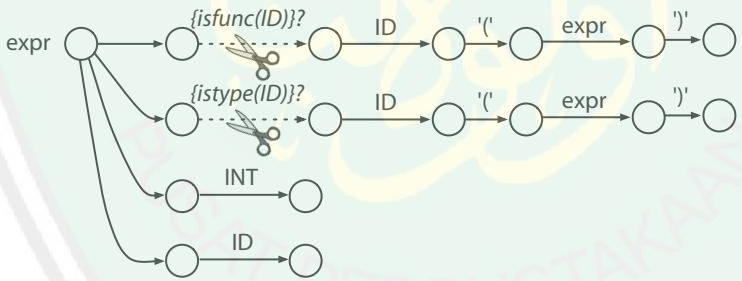
Properly Recognizing T(0) in C++

In C++, expression `T(0)` is either a function call or a constructor-style typecast depending on whether `T` is a function or type name. The expression is ambiguous because the same phrase syntax applies to both interpretations. To get the right interpretation, the parser needs to deactivate one of the alternatives according to how `T` is defined in the program. The following grossly simplified C++ expression rule has two predicates that check the `ID` to see whether it's a function or type name:

predicates/CppExpr.g4

```
/** Distinguish between alts 1, 2 using idealized predicates as demo */
expr:   {«isfunc(ID)»}? ID '(' expr ')' // func call with 1 arg
    |   {«istype(ID)»}? ID '(' expr ')' // ctor-style type cast of expr
    |   INT                             // integer literal
    |   ID                              // identifier
    ;
```

Visually, rule `expr` looks like the following graph with cut points in front of the first two alternatives:



You might be wondering why we simply don't collapse those two alternatives into a single one that handles both cases (function calls and type casts). One reason is that it complicates the job of the parse-tree walker. Instead of two specific methods, one for each case, there is a single `enterCallOrTypecast()`. Inside

that method we'd have to split the two cases manually. That's not the end of the world, though.

The bigger problem is that the ambiguous alternatives are rarely identical like they are here. For example, the function call alternative would also have to handle the case where there are no arguments, such as T(). That would not be a valid typecast, so collapsing the two alternatives wouldn't work in practice. It's also the case that the ambiguous alternatives can be in widely separated rules, which we'll consider in the next example.

Properly Recognizing T(i) in C++

Consider a slight variation of our expression: T(i). To keep things simple, let's assume that there are no constructor-style type casts in our C++ subset. As an expression, then, T(i) must be a function call. Unfortunately, it is also syntactically a valid declaration. It's the same as phrase T i, which defines variable i of type T. The only way to tell the difference is again with context. If T is a type name, then T(i) is a declaration of variable i. Else, it's a function call with i as an argument.

We can demonstrate ambiguous alternatives in separate rules with a small grammar that matches a few bits of C++. Let's say C++ statements can be just declarations or expressions.

predicates/CppStat.g4

```
stat:   decl ';'  {System.out.println("decl "+$decl.text);}
    |   expr ';'  {System.out.println("expr "+$expr.text);}
    ;
```

Syntactically, a declaration can be either T i or T(i).

predicates/CppStat.g4

```
decl:   ID ID           // E.g., "Point p"
    |   ID '(' ID ')'   // E.g., "Point (p)", same as ID ID
    ;
```

And let's say that an expression can be an integer literal, a simple identifier, or a function call with one argument.

predicates/CppStat.g4

```
expr:   INT             // integer literal
    |   ID              // identifier
    |   ID '(' expr ')' // function call
    ;
```

If we build and test the grammar on, say, f(i);, we get an ambiguity warning from the parser (when using the -diagnostics option).

```
⇒ $ antlr4 CppStat.g4
⇒ $ javac CppStat*.java
⇒ $ grun CppStat stat -diagnostics
⇒ f(i);
⇒ EOF
❮ line 1:4 reportAttemptingFullContext d=0, input='f(i);'
  line 1:4 reportAmbiguity d=0: ambigAlts={1, 2}, input='f(i);'
  decl f(i)
```

The parser starts out by notifying us that it detected a problem trying to parse the input with the simple *SLL(*)* parsing strategy. Since that strategy failed, the parser retried with the more powerful *ALL(*)* mechanism. See Section 13.7, *Maximizing Parser Speed*, on page 243. With the full grammar analysis algorithm engaged, the parser again found a problem. At that point, it knew that the input was truly ambiguous. If the parser had not found the problem, it would have printed a reportContextSensitivity message; we'll learn more about that after we add predicates.

That input matches both the second alternative of decl and the third alternative of expr. The parser must choose between them in rule stat. Given two viable alternatives, the parser resolves the ambiguity by choosing the alternative specified first (decl). That's why the parser interprets f(i); as a declaration instead of an expression.

If we had an "oracle" that could tell us whether an identifier was a type name, we could resolve the ambiguity with predicates in front of the ambiguous alternatives.

predicates/PredCppStat.g4

```
decl:   ID ID                          // E.g., "Point p"
    |   {istype()}? ID '(' ID ')'      // E.g., "Point (p)", same as ID ID
    ;

expr:   INT                            // integer literal
    |   ID                             // identifier
    |   {!istype()}? ID '(' expr ')'   // function call
    ;
```

The istype() helper method in the predicates asks the parser for the current token, gets its text, and looks it up in our predefined types table.

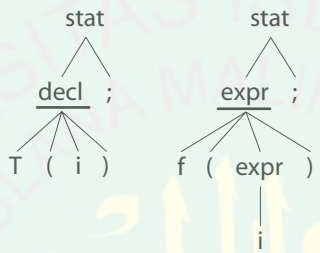
predicates/PredCppStat.g4

```
@parser::members {
Set<String> types = new HashSet<String>() {{add("T");}};
boolean istype() { return types.contains(getCurrentToken().getText()); }
}
```

When we test this predicated version of our grammar, input f(i); is interpreted properly as a function call expression, not a declaration. Input T(i); is unambiguously interpreted as a declaration.

```
⇛ $ antlr4 PredCppStat.g4
⇛ $ javac PredCppStat*.java
⇛ $ grun PredCppStat stat -diagnostics
⇛ f(i);
⇛ EOF
❮ expr f(i)
⇛ $ grun PredCppStat stat -diagnostics
⇛ T(i);
⇛ EOF
❮ decl T(i)
```

The following parse trees (created using the grun -ps file.ps option) illustrate clearly that the parser properly interprets the input phrases:



The key nodes in the parse tree are the underlined parents of T and f. Those internal nodes tell us what kind of thing the parser matched. Remember that the idea behind recognition is that we can distinguish one phrase from the other and can identify the constituent components. We could match any possible input file with grammar fragment .+ (match one or more of any symbol), but it wouldn't tell us anything about the input. Getting the correct structure from the input is crucial to building a language application.

The ambiguities in these C++ examples disappear because the predicates cut out the improper interpretation. Unfortunately, there are some ambiguities for which no predicate exists to resolve them. Let's tackle one more C++ example to see how this can happen.

Properly Recognizing T(i)[5] in C++

C++ is exciting because some phrases have two valid meanings. Consider the C++ phrase T(i)[5]. Syntactically this looks like both a declaration and an expression *even if we know that T is a type name*. That means we can't test identifier T and switch interpretations because there are two interpretations when T is a type name.

The declaration interpretation is as an array of five T elements: T i[5]. The expression interpretation is a typecast of i to T and then an index operation on the resulting array.

The C++ language specification document resolves this ambiguity by always choosing declarations over expressions. The language specification unambiguously tells us humans how to interpret T(i)[5], but it's impossible to build a conventional grammar that is unambiguous even if we add semantic predicates.

Fortunately, the parser resolves this ambiguity automatically so that it behaves naturally. Parsers resolve ambiguities by choosing the alternative specified first. So, we just have to make sure we put the decl alternative before the expr alternative in stat.

There's one final complication to consider when parsing C++.

Resolving Forward References

To compute our types table from the previous section or some other table to distinguish functions from type names, a real C++ parser would have to track names as it encountered them during the parse. Tracking symbols in C++ is a bit tricky but conceptually not a problem. We learned how to track name-value pairs for our calculator in the previous chapter. The problem is that C++ sometimes allows forward references to symbols like method and variable names. That means we might not know that T is a function name until after the parser has seen expression T(i). Gulp.

This should give you some idea of why C++ is so hard to parse. The only solution is to make multiple passes over the input or over an internal representation of the input such as a parse tree.

Using ANTLR, the simplest approach would probably be to tokenize the input, scan it quickly to find and record all of the symbol definitions, and then parse those tokens again "with feeling" to get the proper parse tree.

While most languages don't have such diabolical ambiguity-related issues, just about every language is ambiguous simply because it contains arithmetic expressions. For example, in Section 5.4, *Dealing with Precedence, Left Recursion, and Associativity*, on page 69, we saw that 1+2*3 is ambiguous because we can interpret it as (1+2)*3 or 1+(2*3).

The behavior of semantic predicates is more or less straightforward if we think of them as simple Boolean expressions that turn alternatives on and off. Unfortunately, things can get fairly complicated in grammars with multiple

predicates and embedded actions. The reference chapter goes over the details of when and how ANTLR uses predicates. If you don't plan on using lots of predicates mixed with actions in your grammar, you can probably skip Section 15.7, *Semantic Predicates*, on page 286. Later these details might help explain some perplexing grammar behavior.

Now that we know how to customize generated parsers using actions and semantic predicates, we have some fearsome skills. In the next chapter, we're going to solve some very difficult recognition problems using what we've learned so far in Part III.

CHAPTER 12

Wielding Lexical Black Magic

In this part of the book, we've learned some advanced skills. We know how to execute arbitrary code while parsing, and we can alter syntax recognition with semantic predicates. Now it's time to put those skills to work solving some challenging language recognition problems but in the lexer this time, not the parser.

In my experience, if a language problem is hard to solve, most of the head-scratching occurs in the lexer (well, with the exception of C++, which is hard all over). That's counterintuitive because the lexer rules we've seen so far have been pretty simple, such as identifiers, integers, and arithmetic expression operators. But, consider the harmless-looking two-character Java sequence: >>. A Java lexer could match it either as the right shift operator or as two > operators, which the parser could use to close a nested generic type like List<List<String>>.

The fundamental problem is that the lexer does the tokenizing, but sometimes only the parser has the context information needed to make tokenizing decisions. We'll explore this issue in Section 12.2, *Context-Sensitive Lexical Problems*, on page 208. During that discussion, we'll also look at the "keywords can be identifiers" problem and build a lexer to deal with Python's context-sensitive newline handling.

The next problem we'll look at involves *island languages* whose sentences have islands of interesting bits surrounded by a sea of stuff we don't care about. Examples include XML and template languages like StringTemplate. To parse these, we need *island grammars* and *lexical modes*, which we'll explore in Section 12.3, *Islands in the Stream*, on page 219.

Finally, we'll build an ANTLR XML parser and lexer from the XML specification. It's a great example of how to deal with input streams containing different

contexts (regions), how to draw a line between the parser and lexer, and how to accept non-ASCII input characters.

To get warmed up, let's learn how to ignore but not throw out special input regions such as comments and whitespace. The technique can be used to solve lots of language translation problems, and we'll demonstrate the most common use case here.

12.1 Broadcasting Tokens on Different Channels

Most programming languages ignore whitespace and comments in between tokens, which means they can appear anywhere. That presents a problem for a parser since it has to constantly check for optional whitespace and comment tokens. The common solution is to simply have the lexer match those tokens but throw them out, which is what we've done so far in this book. For example, our Cymbol grammar from Section 6.4, *Parsing Cymbol*, on page 98 threw out whitespace and comments using the skip lexer command.

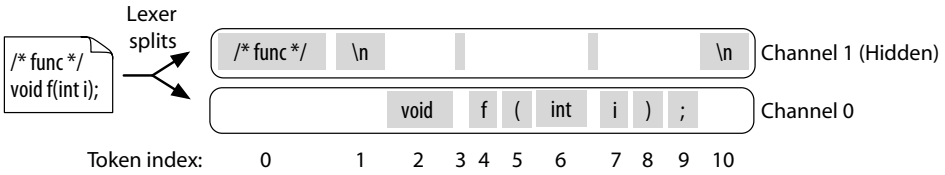
examples/Cymbol.g4

```
WS  :   [ \t\n\r]+ -> skip ;

SL_COMMENT
    :   '//' .*? '\n' -> skip
    ;
```

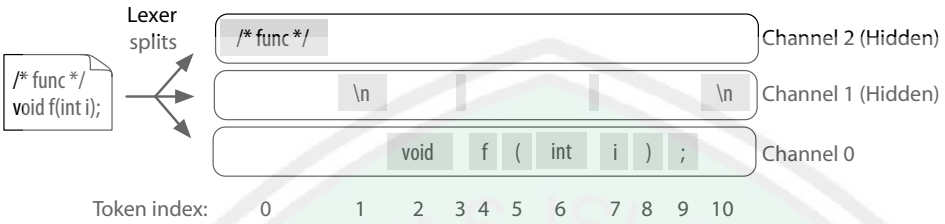
That works great for many applications, such as compilers, because the comments don't affect code generation. If, on the other hand, we're trying to build a translator to convert legacy code into a modern language, we really should keep the comments around because they're part of the program. This presents a conundrum: we want to keep the comments and whitespace, but we don't want to burden the parser with constant checks for them in between tokens.

Filling Token Channels

ANTLR's solution is to send the actual language tokens like identifiers to the parser on one *channel* and everything else on a different channel. Channels are like different radio frequencies. The parser tunes to exactly one channel and ignores tokens on the other channels. Lexer rules are responsible for putting tokens on different channels, and class CommonTokenStream is responsible for presenting only one channel to the parser. CommonTokenStream does this while preserving the original relative token order so we can request the comments before or after a particular language token. The following image represents CommonTokenStream's view of the tokens emitted by a C lexer that puts comments and whitespace on a hidden channel:



We can just as easily isolate the comments on one channel and the whitespace on another, leaving the real tokens on the default channel 0.



That way, we can ask for the comments and whitespace separately.

To transmit tokens on a different channel, we use lexer command channel(...) in the appropriate lexer rule. Let's demonstrate this technique by altering our Cymbol grammar to put comments on hidden channel 2 and whitespace on hidden channel 1, like the last image.

lexmagic/Cymbol.g4

```
WS  :   [ \t\n\r]+ -> channel(WHITESPACE) ;  // channel(1)

SL_COMMENT
    :   '//' .*? '\n' -> channel(COMMENTS)   // channel(2)
    ;
```

Constants WHITESPACE and COMMENTS come from a declaration in our grammar.

lexmagic/Cymbol.g4

```
@lexer::members {
    public static final int WHITESPACE = 1;
    public static final int COMMENTS = 2;
}
```

ANTLR translates channel(HIDDEN) to Java as _channel = HIDDEN, which sets class Lexer's _channel field to constant HIDDEN. We can use any valid Java qualified identifier as an argument to command channel().

Testing the grammar with grun shows that the comments appear on channel 2, whitespace appears on channel 1, and the other tokens appear on the default channel.

```
⇛ $ antlr4 Cymbol.g4
⇛ $ javac Cymbol*.java
⇛ $ grun Cymbol file -tokens -tree
⇛ int i = 3; // testing
⇛ EOF
❮ [@0,0:2='int',<10>,1:0]
  [@1,3:3=' ',<24>,channel=1,1:3]                 <-- HIDDEN channel 1
  [@2,4:4='i',<22>,1:4]
  [@3,5:5=' ',<24>,channel=1,1:5]                 <-- HIDDEN channel 1
  [@4,6:6='=',<11>,1:6]
  [@5,7:7=' ',<24>,channel=1,1:7]                 <-- HIDDEN channel 1
  [@6,8:8='3',<23>,1:8]
  [@7,9:9=';',<13>,1:9]
  [@8,10:10=' ',<24>,channel=1,1:10]              <-- HIDDEN channel 1
  [@9,11:21='// testing\n',<25>,channel=2,1:11]   <-- HIDDEN channel 2
  [@10,22:21='<EOF>',<-1>,2:22]
  (file (varDecl (type int) i = (expr 3) ;))      <-- parse tree
```

The parse tree also looks right, which means that the parser correctly interpreted the input. The lack of syntax error indicates that the parser didn't plow into a comment token. Now let's figure out how to access hidden comments from a language application.

Accessing Hidden Channels

To illustrate how to access the hidden channels from a language application, let's build a parse-tree listener that shifts comments following declarations to precede the declarations, tweaking them to use /*...*/-style comments. For example, given the following input:

lexmagic/t.cym

```
int n = 0; // define a counter
int i = 9;
```

we want to generate the following output:

```
/* define a counter */
int n = 0;
int i = 9;
```

Our basic strategy will be to rewrite the token stream using a TokenStreamRewriter, as we did in *Rewriting the Input Stream*, on page 52. Upon seeing a variable declaration, our application will grab the comment, if any, to the right of the semicolon and insert it before the first token of the declaration. Here's a Cymbol parse-tree listener called CommentShifter that sits inside a test rig class called ShiftVarComments:

lexmagic/ShiftVarComments.java

```
Line 1 public static class CommentShifter extends CymbolBaseListener {
   -     BufferedTokenStream tokens;
   -     TokenStreamRewriter rewriter;
   -             /** Create TokenStreamRewriter attached to token stream
   5              *  sitting between the Cymbol lexer and parser.
   -              */
   -     public CommentShifter(BufferedTokenStream tokens) {
   -         this.tokens = tokens;
   -         rewriter = new TokenStreamRewriter(tokens);
  10     }
   -
   -     @Override
   -     public void exitVarDecl(CymbolParser.VarDeclContext ctx) {
   -         Token semi = ctx.getStop();
  15         int i = semi.getTokenIndex();
   -         List<Token> cmtChannel =
   -             tokens.getHiddenTokensToRight(i, CymbolLexer.COMMENTS);
   -         if ( cmtChannel!=null ) {
   -             Token cmt = cmtChannel.get(0);
  20             if ( cmt!=null ) {
   -                 String txt = cmt.getText().substring(2);
   -                 String newCmt = "/* " + txt.trim() + " */\n";
   -                 rewriter.insertBefore(ctx.start, newCmt);
   -                 rewriter.replace(cmt, "\n");
  25             }
   -         }
   -     }
   - }
```

All of the work happens in exitVarDecl(). First we get the token index of the declaration semicolon (line 14) because we're looking for comments after that token. Line 17 asks the token stream if there are any hidden tokens on channel COMMENTS to the right of the semicolon. For simplicity, the code assumes there is only one, so line 19 grabs the first comment from the list. Then we derive the new style comment from the old comment and inject it using TokenStreamRewriter before the start of the variable declaration (line 23). Finally, we replace the existing following comment with a newline (line 24), effectively erasing it.

The test rig itself is the same old story, but at the end, we ask the TokenStreamRewriter class to give us the rewritten input with getText().

lexmagic/ShiftVarComments.java

```
CymbolLexer lexer = new CymbolLexer(input);
CommonTokenStream tokens = new CommonTokenStream(lexer);
CymbolParser parser = new CymbolParser(tokens);
RuleContext tree = parser.file();
```

```
ParseTreeWalker walker = new ParseTreeWalker();
CommentShifter shifter = new CommentShifter(tokens);
walker.walk(shifter, tree);
System.out.print(shifter.rewriter.getText());
```

Here's the build and test sequence:

```
$ antlr4 Cymbol.g4
$ javac Cymbol*.java ShiftVarComments.java
$ java ShiftVarComments t.cym
/* define a counter */
int n = 0;
int i = 9;
```

Note that if we had thrown out whitespace, rather than sending it on a hidden channel, that output would be all bunched together like `intn=0;`.

Token channels solve a tricky language translation problem by categorizing input tokens. Now, we're going to focus on problems related to the construction of the tokens themselves.

12.2 Context-Sensitive Lexical Problems

Consider the phrase "Brown leaves in the fall." It's ambiguous because there are two interpretations. If we're talking about trees, the phrase refers to nature's photosynthesis engine. If, on the other hand, we're discussing a certain Ms. Jane Brown, the context totally changes the function of those words. "Leaves" shifts from a noun to a verb.

This situation resembles the problems we solved in Section 11.3, *Recognizing Ambiguous Phrases*, on page 196 where context-sensitive C++ phrases like `T(0)` could be function calls or type casts depending on how `T` was defined elsewhere in the program. Such syntactic ambiguities arose because our C++ lexer sent vague generic `ID` tokens to the parser. We needed semantic predicates in the parser rules to choose between alternative interpretations.

To get rid of the predicates in the parser rules, we could have the lexer send more precise tokens to the parser such as `FUNCTION_NAME` vs. `TYPE_NAME`, depending on context. (For input "Brown leaves," we'd have the lexer send token sequence `ADJECTIVE NOUN` vs. `PROPER_NAME VERB`.) Unfortunately, that just shifts the context-sensitivity problem to the lexer, and the lexer has nowhere near as much context information as the parser. That's why we predicated parser rules in the previous chapter instead of trying to use lexer context to send more precise tokens to the parser.

Hack Alert: Parser Feedback

A common practice that's been around forever involves sending feedback from the parser to the lexer so that the lexer can send precise tokens to the parser. The parser can then be simpler because of the reduction in predicates. Unfortunately, this is not possible with ANTLR grammars because ANTLR-generated parsers often look very far ahead in the token stream to make parsing decisions. That means the lexer could be asked to tokenize input characters long before the parser has had a chance to execute actions that provide context information to the lexer.

We can't always escape context-sensitivity issues related to tokenizing input characters. In this section, we're going to look at three lexical problems that fit into the context-sensitivity bucket.

- *The same token character sequence can mean different things to the parser.* We'll tackle the well-known "keywords can also be identifiers" problem.
- *The same character sequence can be one token or multiple.* We'll see how to treat Java character sequence >> either as two close-type-parameter tokens or as a single right shift operator token.
- *The same token must sometimes be ignored and sometimes be recognized by the parser.* We'll learn how to distinguish between Python's physical and logical input lines. The solution requires both lexical actions and semantic predicates, techniques we learned in the previous two chapters.

Treating Keywords As Identifiers

Lots of languages, both old and new, allow keywords as identifiers, depending on the context. In Fortran, we could say things like `end = goto + if/while`. C# supports SQL queries with its Language-Integrated Query (LINQ) feature. Queries begin with keyword `from`, but we can also use `from` as a variable: `x = from + where;`. That's unambiguously an expression, not a query syntactically, so the lexer should treat `from` as an identifier, not a keyword. The problem is that the lexer doesn't parse the input. It doesn't know whether to send, say, `KEYWORD_FROM` or `ID` to the parser.

There are two approaches to allowing keywords to act as identifiers in some syntactic contexts. The first approach has the lexer pass all keywords to the parser as keyword token types, and then we create a parser `id` rule that matches `ID` and any of the keywords. The second approach has the lexer pass keywords as identifiers, and then we use predicates to test identifier names in the parser like this:

When in Paris...

I worked in France in the late 1980s and quickly found a problem when calling people on the phone. When the receiving end asked who was calling, I'd reply *Monsieur Parr*, but Parr sounds just like *part*, the third-person singular of the verb "to leave." It sounds like I'm saying that I'm going to hang up. Hilarious.

Here's a fun tongue twister in French where each word requires heavy context to decipher: "Si six cent scies scient six cent saucisses, six cent six scies scieront six cent six saucissons." The written form is repetitive but clear. When spoken, however, the pronunciation in English is something like this: "See see saw, see see see saw, sawcease, see saw see see seeron see saw see sawcease." The translation is "If six hundred saws saw six hundred sausages, six hundred and six saws will saw six hundred six sausages." Don't make fun of the French for the fact that *si*, *six*, *scies*, and *scient* all sound exactly the same. English has words that are written the same but pronounced differently, like read (present tense) and read (past tense)!

```
keyIF : {_input.LT(1).getText().equals("if")}? ID ;
```

That's pretty ugly, and likely slow, so we'll stick with the first approach. (For completeness, I've left a small but working predicated keyword example in PredKeyword.g4.)

To illustrate the recommended approach, here's a simple grammar that matches nutty statements like if if then call call;:

lexmagic/IDKeyword.g4

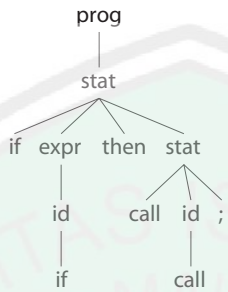
```
grammar IDKeyword;

prog: stat+ ;

stat: 'if' expr 'then' stat
    | 'call' id ';'
    | ';'
    ;

expr: id ;

id  :   'if' | 'call' | 'then' | ID ;

ID : [a-z]+ ;
WS : [ \r\n]+ -> skip ;
```

In a nutshell, the approach replaces all references to token ID with references to rule id. If you're faced with a language that allows different sets of keywords to be identifiers in different contexts, you'll need more than one id rule (one per context).

Here's the build and test sequence for grammar IDKeyword:

```
⇛ $ antlr4 IDKeyword.g4
⇛ $ javac IDKeyword*.java
⇛ $ grun IDKeyword prog
⇛ if if then call call;
⇛ EOF
```

The parse tree shows that the grammar treats the second if and second call symbols as identifiers.



In this problem, the lexer has to decide whether to return keyword or identifier tokens, but it doesn't have to worry about which characters constitute the tokens. Now we're going to work on a problem where the lexer doesn't know how much input to consume for each token.

Avoiding the Maximal Munch Ambiguity

There is a general assumption made by lexer-generator tools that lexers should match the longest possible token at each input position. That assumption usually gives lexers the most natural behavior. For example, given C input +=, a C lexer should match the single token +=, not two separate tokens, as in + and =. Unfortunately, there are a few cases we need to handle differently.

In C++, we can't close nested parameterized types with a double angle bracket like this: `A<B<C>>`. We have to use a space in between the final angle brackets, `A<B<C> >`, so the lexer doesn't confuse the double angle brackets with the right shift operator, >>.[1] It was considered a hard enough problem that the designers of C++ altered the language to overcome a nasty lexical implementation problem.

A number of suitable solutions have since popped up, but the simplest is to never have the lexer match the >> character sequence as a right shift operator. Instead, the lexer sends two > tokens to the parser, which can use context

1. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2005/n1757.html

to pack the tokens together appropriately. For example, a C++ parser expression rule would match two right angle brackets in a row instead of a single token. If you look back at the Java of grammar from Section 4.3, *Building a Translator with a Listener*, on page 42, you'll see an example implementation of this approach. Here are two `expr` rule alternatives that combine single-character tokens into multicharacter operators:

tour/Java.g4

```
|   expression ('<' '<' | '>' '>' '>' | '>' '>') expression
|   expression ('<' '=' | '>' '=' | '>' | '<') expression
```

Let's look at the tokens passed by the lexer to the parser for the shift operator.

```
⇛ $ antlr4 Java.g4
⇛ $ javac Java*.java
⇛ $ grun Java tokens -tokens
⇛ i = 1 >> 5;
⇛ EOF
❮ [@0,0:0='i',<98>,1:0]
  [@1,1:1=' ',<100>,channel=1,1:1]
  [@2,2:2='=',<25>,1:2]
  [@3,3:3=' ',<100>,channel=1,1:3]
  [@4,4:4='1',<91>,1:4]
  [@5,5:5=' ',<100>,channel=1,1:5]
  [@6,6:6='>',<81>,1:6]                <-- two '>' tokens not one '>>'
  [@7,7:7='>',<81>,1:7]
  [@8,8:8=' ',<100>,channel=1,1:8]
  [@9,9:9='5',<91>,1:9]
  [@10,10:10=';',<77>,1:10]
  [@11,11:11='\n',<100>,channel=1,1:11]
  [@12,12:11='<EOF>',<-1>,2:12]
```

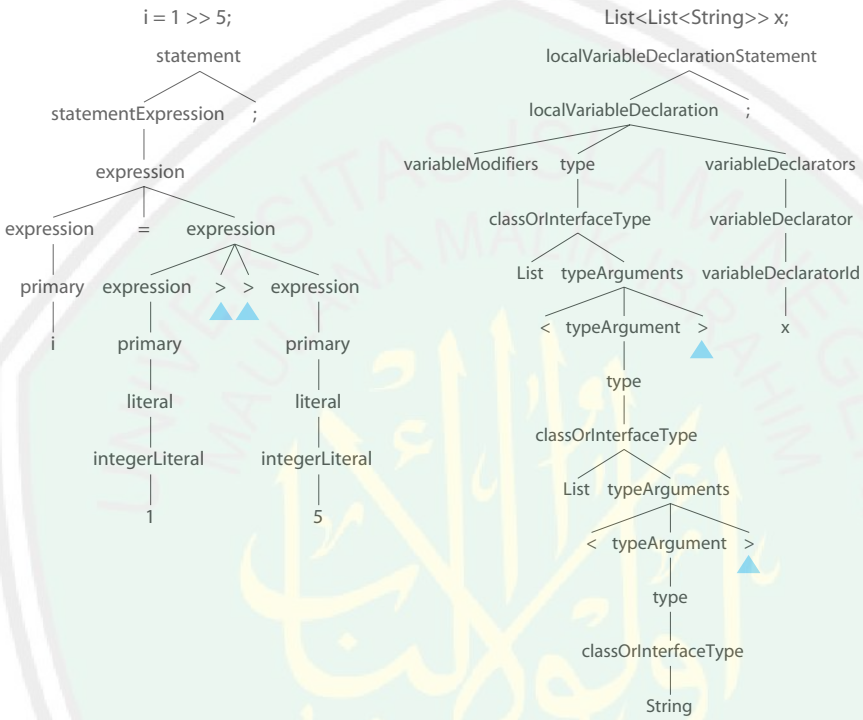
And here's what the token stream looks like for a nested generic type reference:

```
⇛ $ grun Java tokens -tokens
⇛ List<List<String>> x;
⇛ EOF
❮ [@0,0:3='List',<98>,1:0]
  [@1,4:4='<',<5>,1:4]
  [@2,5:8='List',<98>,1:5]
  [@3,9:9='<',<5>,1:9]
  [@4,10:15='String',<98>,1:10]
  [@5,16:16='>',<81>,1:16]
  [@6,17:17='>',<81>,1:17]
  [@7,18:18=' ',<100>,channel=1,1:18]
  [@8,19:19='x',<98>,1:19]
  [@9,20:20=';',<77>,1:20]
  [@10,21:21='\n',<100>,channel=1,1:21]
  [@11,22:21='<EOF>',<-1>,2:22]
```

Now let's generate the parse trees for those phrases because they make it clear how the grammar uses the > tokens.

```
⇒ $ grun Java statement -gui
⇒ i = 1 >> 5;
⇒ EOF
⇒ $ grun Java localVariableDeclarationStatement -gui
⇒ List<List<String>> x;
⇒ EOF
```

Here are the parse trees side by side, rooted at rules statement and localVariableDeclarationStatement with the angle brackets highlighted:



The only problem with splitting the two right angle brackets of the shift operator is that the parser will also accept angle brackets separated by a space character, > >. To address this, either we can add semantic predicates to the grammar or we can check the parse tree afterward using a listener or visitor to ensure that the > token column numbers are adjacent for shift operators. It'd be inefficient to use predicates during the parse, so it's better to check the right shift operators after the parse. Most language applications need to walk the parse tree anyway. (Predicates in an expression rule would also break ANTLR's left-recursive rule pattern that it knows how to convert to a non-left-recursive version. See Chapter 14, *Removing Direct Left Recursion*, on page 247.)

At this point, we've seen how to put tokens on different channels and how to split context-sensitive tokens into their smallest valid token components. Now we're going to figure out how to treat the same character sequence as two different token types, depending on the context.

Fun with Python Newlines

Python's newline handling is very natural for the programmer. Rather than using a semicolon, newlines terminate statements. Most of us put one statement per line anyway, so typing the semicolon constantly is a nuisance. At the same time, we don't want to put really long expressions on the same physical line, so Python ignores newlines in certain contexts. For example, Python lets us split method calls over multiple lines like this:

```
f(1,
2,
3)
```

To figure out when to ignore newlines, let's put together all of the bits of documentation concerning newlines from the Python reference manual.[2] The most important rule is as follows:

> Expressions in parentheses, square brackets, or curly braces can be split over more than one physical line [...].

So, if we try to split expression `1+2` after the `+` with a newline, Python emits an error. We can, however, split `(1+2)` across lines. The manual also says that "Implicitly continued lines can carry comments" and "Blank continuation lines are allowed," like this:

```
f(1,  # first arg

2,    # second arg
      # blank line with a comment
3)    # third arg
```

We can also explicitly join physical lines into one logical line using a backslash.

> Two or more physical lines may be joined into logical lines using backslash characters (\), as follows: when a physical line ends in a backslash that is not part of a string literal or comment, it is joined with the following, forming a single logical line, deleting the backslash and the following end-of-line character.

That means we can split lines even outside of grouping symbols like this:

```
1+\
2
```

2. http://docs.python.org/reference/lexical_analysis.html

The manual does not explicitly say so, but the "line ends in a backslash" clause implies that there can't be a comment between the \ and the newline character.

The upshot is that either the parser or the lexer needs to toss out some newlines but not others. As we saw earlier with token channels, having the parser check all the time for optional whitespace isn't a good solution. That means that a Python lexer needs to handle the optional newlines. This then is another case of syntactic context dictating lexer behavior.

With all of these rules in mind, let's build a grammar for a trivial version of Python that matches assignments and simple expressions. We'll ignore strings in order to focus solely on proper comments and newline handling. Here are the syntax rules:

lexmagic/SimplePy.g4
```
file:   stat+ EOF ;

stat:   assign NEWLINE
    |   expr NEWLINE
    |   NEWLINE         // ignore blank lines
    ;

assign: ID '=' expr ;

expr:   expr '+' expr
    |   '(' expr ')'
    |   call
    |   list
    |   ID
    |   INT
    ;

call:   ID '(' ( expr (',' expr)* )? ')' ;

list:   '[' expr (',' expr)* ']' ;
```

To build the lexer, let's get the familiar rules out of the way first. The INT rule for integers is the usual one, and, according to the reference, identifiers look like this:

```
identifier ::=  (letter|"_") (letter | digit | "_")*
letter     ::=  lowercase | uppercase
```

In ANTLR notation, that's as follows:

lexmagic/SimplePy.g4
```
ID  :   [a-zA-Z_] [a-zA-Z_0-9]* ;
```

Then, we need the usual whitespace rule and a rule to match newlines, which sends NEWLINE tokens to the parser.

lexmagic/SimplePy.g4
```
/** A logical newline that ends a statement */
NEWLINE
    :   '\r'? '\n'
    ;

/** Warning: doesn't handle INDENT/DEDENT Python rules */
WS  :   [ \t]+ -> skip
    ;
```

To handle Python's line comments, we need a rule that strips out the comment part but doesn't touch the newline.

lexmagic/SimplePy.g4
```
/** Match comments. Don't match \n here; we'll send NEWLINE to the parser. */
COMMENT
    :    '#' ~[\r\n]* -> skip
    ;
```

We want NEWLINE to handle all newlines so that the following:

```
i = 3 # assignment
```

looks like an assignment followed by NEWLINE.

Now it's time to handle the special newline stuff. Let's start with explicit line joining. We add a rule to match \ immediately followed by a newline, tossing it out.

lexmagic/SimplePy.g4
```
/** Ignore backslash newline sequences. This disallows comments
 *  after the backslash because newline must occur next.
 */
LINE_ESCAPE
    :   '\\' '\r'? '\n' -> skip
    ;
```

That means the parser won't see either the \ or the newline character(s).

Now we have to make the lexer ignore newlines inside grouping symbols like parentheses and brackets. That means we need a lexer rule called IGNORE_NEWLINE that matches newlines like NEWLINE but skips the token if it's within grouping symbols. Because those two rules match the same character sequence, they're ambiguous, and we need a semantic predicate to differentiate them. If we imagine for the moment that there's a magic nesting variable that is greater than zero when the lexer has seen an open grouping symbol but not the closing symbol, we can write IGNORE_NEWLINE like this:

lexmagic/SimplePy.g4
```
/** Nested newline within a (..) or [..] are ignored. */
IGNORE_NEWLINE
    :   '\r'? '\n' {nesting>0}? -> skip
    ;
```

That rule must appear before rule NEWLINE so that when the predicate is true, the lexer resolves the ambiguity by choosing rule IGNORE_NEWLINE. We could also put a {nesting==0}? predicate in NEWLINE to resolve the order dependency.

Now let's wiggle this variable appropriately as we see opening and closing parentheses and brackets. (Our syntax does not allow curly braces.) First, let's define the magic nesting variable.

lexmagic/SimplePy.g4
```
@lexer::members {
    int nesting = 0;
}
```

Then, we need to execute actions that bump nesting up and down as we see the grouping symbols. The following rules do the trick:

lexmagic/SimplePy.g4
```
LPAREN    : '(' {nesting++;} ;

RPAREN    : ')' {nesting--;} ;

LBRACK    : '[' {nesting++;} ;

RBRACK    : ']' {nesting--;} ;
```

To be strictly correct, we should use a different variable for parentheses and for brackets so that we can make sure they balance. But we don't really have to worry about imbalances like [1,2) because the parser will detect an error. Any inexact behavior with ignored newlines is not important in the presence of such a syntax error.

To test our SimplePy grammar, the following test file exercises the key elements of Python newline and comment processing: blanks are ignored, newlines are ignored within grouping symbols, backslashes hide the next newline, and comments don't affect newline processing inside grouping symbols.

lexmagic/f.py
```
# a test
f(1, # first arg

  2, # second arg
     # blank line with a comment
  3) # third arg
```

```
g() # on end

1+\
2+\
3
```

Here's the build and test sequence showing the token stream sent to the parser with highlighted NEWLINE tokens:

```
$ antlr4 SimplePy.g4
$ javac SimplePy*.java
$ grun SimplePy file -tokens f.py
[@0,8:8='\n',<11>,1:8]
[@1,9:9='f',<4>,2:0]
[@2,10:10='(',<6>,2:1]
[@3,11:11='1',<5>,2:2]
[@4,12:12=',',<1>,2:3]
[@5,29:29='2',<5>,4:2]
[@6,30:30=',',<1>,4:3]
[@7,80:80='3',<5>,6:2]
[@8,81:81=')',<7>,6:3]
[@9,94:94='\n',<11>,6:16]
[@10,95:95='\n',<11>,7:0]
[@11,96:96='g',<4>,8:0]
[@12,97:97='(',<6>,8:1]
[@13,98:98=')',<7>,8:2]
[@14,108:108='\n',<11>,8:12]
[@15,109:109='\n',<11>,9:0]
[@16,110:110='1',<5>,10:0]
[@17,111:111='+',<2>,10:1]
[@18,114:114='2',<5>,11:0]
[@19,115:115='+',<2>,11:1]
[@20,118:118='3',<5>,12:0]
[@21,119:119='\n',<11>,12:1]
[@22,120:119='<EOF>',<-1>,13:2]
```

The key thing to notice is that there are six NEWLINE tokens in the stream but twelve newlines in file f.py. Our lexer successfully chucks out six newlines. The parse tree with highlighted newline tokens looks like Figure 11, *Parse tree with highlighted newline tokens*, on page 219.

The first newline is a blank line and matched as an empty statement (rule stat) by the parser. The third and fifth newlines are also empty statements. The three other newlines terminate expression statements. Running f.py into a Python interpreter (with suitable f() and g() definitions) confirms that f.py is valid Python.

We've just worked through three kinds of context-sensitivity problems associated with tokens. The contexts we considered were defined by the syntax, not

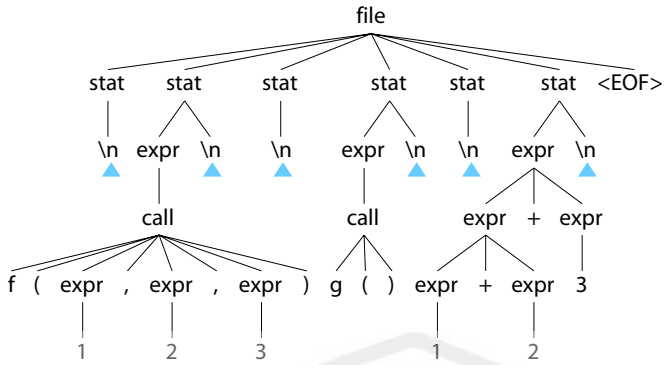


Figure 11—Parse tree with highlighted newline tokens

a region of the input file. Next, we're going to look at input files that have isolated regions of interest surrounded by regions we don't care about.

12.3 Islands in the Stream

The input files we've discussed so far all contain a single language. For example, DOT, CSV, Python, and Java files contain nothing but text conforming to those languages. But, there are file formats that contain random text surrounding structured regions or *islands*. We call such formats *island languages* and describe them with *island grammars*. Examples include template engine languages such as StringTemplate and the LaTeX document preparation language, but XML is the quintessential island language. XML files contain structured tags and & entities surrounded by a sea of stuff we don't care about. (Because there is some structure between the tags themselves, we might call XML an *archipelago language*.)

Classifying something as an island language often depends on our perspective. If we're building a C preprocessor, the preprocessor commands form an island language where the C code is the sea. On the other hand, if we're building a C parser suitable for an IDE, the parser must ignore the sea of preprocessor commands.

Our goal in this section is to learn how to ignore the sea and tokenize the islands so the parser can verify syntax within those islands. We'll need both of those techniques to build a real XML parser in the next section. Let's start by learning how to distinguish XML islands from the sea.

Separating XML Islands from a Sea of Text

To separate XML tags from text, our first thought might be to build an input character stream filter that strips everything between tags. This might make it easy for the lexer to identify the islands, but the filter would throw out all of the text data, which is not what we want. For example, given input `<name>John</name>`, we don't want to throw out `John`.

Instead, let's build a baby XML grammar that lumps the text inside of tags together as one token and the text outside of tags as another token. Since we're focusing on the lexer here, we'll use a single syntactic rule that matches a bunch of tags, `&` entities, `CDATA` sections, and text (the sea).

lexmagic/Tags.g4

```
grammar Tags;
file : (TAG|ENTITY|TEXT|CDATA)* ;
```

Rule `file` makes no attempt to ensure the document is well formed—it just indicates the kinds of tokens found in an XML file.

To split up an XML file with lexer rules, we can just give rules for the islands and then a catchall rule called `TEXT` at the end to match everything else.

lexmagic/Tags.g4

```
COMMENT : '<!--' .*? '-->' -> skip ;
CDATA : '<![CDATA[' .*? ']]>' ;
TAG : '<' .*? '>' ; // must come after other tag-like structures
ENTITY : '&' .*? ';' ;
TEXT : ~[<&]+ ;     // any sequence of chars except < and & chars
```

Those rules make heavy use of the nongreedy `.*?` operator (see *Matching String Literals*, on page 75) that scans until it sees what follows that operation in the rule.

Rule `TEXT` matches one or more characters, as long as the character isn't the start of a tag or entity. It's tempting to put `.+` instead of `~[<&]+`, but that would consume until the end of the input once it got into the loop. There's no string to match following `.+` in `TEXT` that would tell the loop when to stop.

An important but subtle ambiguity-resolving mechanism is in play here. In Section 2.3, *You Can't Put Too Much Water into a Nuclear Reactor*, on page 13, we learned that ANTLR lexers resolve ambiguities in favor of the rule specified first in the grammar file. For example, rule `TAG` matches anything in angle brackets, which includes comments and `CDATA` sections. Because we specified `COMMENT` and `CDATA` first, rule `TAG` matches only those tags that failed to match the other tag rules.

As a side note, XML technically doesn't allow comments that end with ---> or comments that contain --. Using what we learned in Section 9.4, *Error Alternatives*, on page 170, we could add lexical rules to look for bad comments and give specific and informative error messages.

```
BAD_COMMENT1:   '<!--' .*? '--->'
                {System.err.println("Can't have ---> end comment");} -> skip ;
BAD_COMMENT2:   '<!--' ('--'|.)*? '-->'
                {System.err.println("Can't have -- in comment");}    -> skip ;
```

I've left them out of grammar `Tags` for simplicity.

Now let's see what our baby XML grammar does with the following input:

lexmagic/XML-inputs/cat.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<?do not care?>
<CATALOG>
<PLANT id="45">Orchid</PLANT>
</CATALOG>
```

Here's the build and test sequence, using `grun` to print out the tokens:

```
$ antlr4 Tags.g4
$ javac Tags*.java
$ grun Tags file -tokens XML-inputs/cat.xml
[@0,0:37='<?xml version="1.0" encoding="UTF-8"?>',<3>,1:0]
[@1,38:38='\n',<5>,1:38]
[@2,39:53='<?do not care?>',<3>,2:0]
[@3,54:54='\n',<5>,2:15]
[@4,55:63='<CATALOG>',<3>,3:0]
[@5,64:64='\n',<5>,3:9]
[@6,65:79='<PLANT id="45">',<3>,4:0]
[@7,80:85='Orchid',<5>,4:15]
[@8,86:93='</PLANT>',<3>,4:21]
[@9,94:94='\n',<5>,4:29]
[@10,95:104='</CATALOG>',<3>,5:0]
[@11,105:105='\n',<5>,5:10]
[@12,106:105='<EOF>',<-1>,6:11]
```

This baby XML grammar properly reads in XML files and matches a sequence of the various islands and text. What it doesn't do is pull apart the tags and pass the pieces to a parser so it can check the syntax.

Issuing Context-Sensitive Tokens with Lexical Modes

The text inside and outside of tags conform to different languages. For example, `id="45"` is just a lump of text outside of a tag, but it's three tokens inside of a tag. In a sense, we want an XML lexer to match different sets of rules depending on the context. ANTLR provides *lexical modes* that let lexers

switch between contexts (modes). In this section, we'll learn to use lexical modes by improving the baby XML grammar from the previous section so that it passes tag components to the parser.

Lexical modes allow us to split a single lexer grammar into multiple sublexers. The lexer can return only those tokens matched by entering a rule in the current mode. One of the most important requirements for mode switching is that the language have clear lexical sentinels that can trigger switching back and forth, such as left and right angle brackets. To be clear, modes rely on the fact that the lexer doesn't need syntactic context to distinguish between different regions in the input.

To keep things simple, let's build a grammar for an XML subset where tags contain an identifier but no attributes. We'll use the default mode to match the sea outside of tags and another mode to match the inside of tags. When the lexer matches < in default mode, it should switch to island mode (inside tag mode) and return a tag start token to the parser. When the inside mode sees >, it should switch back to default mode and return a tag stop token. The inside mode also needs rules to match identifiers and /. The following lexer encodes that strategy:

lexmagic/ModeTagsLexer.g4

```
lexer grammar ModeTagsLexer;

// Default mode rules (the SEA)
OPEN  : '<'     -> mode(ISLAND) ;       // switch to ISLAND mode
TEXT  : ~'<'+ ;                         // clump all text together

mode ISLAND;
CLOSE : '>'     -> mode(DEFAULT_MODE) ; // back to SEA mode
SLASH : '/' ;
ID    : [a-zA-Z]+ ;                     // match/send ID in tag to parser
```

Rules OPEN and TEXT are in the default mode. OPEN matches a single < and uses lexer command mode(ISLAND) to switch modes. Upon the next token request from the parser, the lexer will consider only those rules in ISLAND mode. TEXT matches any sequence of characters that doesn't start a tag. Because none of the lexical rules in this grammar uses lexical command skip, all of them return a token to the parser when they match.

In ISLAND mode, the lexer matches closing >, /, and ID tokens. When the lexer sees >, it will execute the lexer command to switch back to the default mode, identified by constant DEFAULT_MODE in class Lexer. This is how the lexer ping-pongs back and forth between modes.

The parser for our slightly augmented XML subset matches tags and text chunks as in grammar Tags, but now we're using rule tag to match the individual tag elements instead of a single lumped token.

lexmagic/ModeTagsParser.g4

```
parser grammar ModeTagsParser;

options { tokenVocab=ModeTagsLexer; } // use tokens from ModeTagsLexer.g4

file: (tag | TEXT)* ;

tag : '<' ID '>'
    | '<' '/' ID '>'
    ;
```

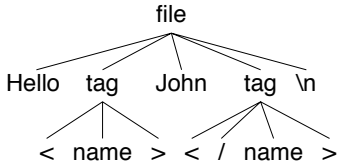
The only unfamiliar syntax in the parser is the tokenVocab option. When we have the parser and lexer in separate files, we need to make sure that the token types and token names from the two files are synchronized. For example, lexer token OPEN must have the same token type in the parser as it does in the lexer.

Let's build the grammar and try it on some simple XML input.

```
⇛ $ antlr4 ModeTagsLexer.g4   # must be done first to get ModeTagsLexer.tokens
⇛ $ antlr4 ModeTagsParser.g4
⇛ $ javac ModeTags*.java
⇛ $ grun ModeTags file -tokens
⇛ Hello <name>John</name>
⇛ EOF
❮ [@0,0:5='Hello ',<2>,1:0]
  [@1,6:6='<',<1>,1:6]
  [@2,7:10='name',<5>,1:7]
  [@3,11:11='>',<3>,1:11]
  [@4,12:15='John',<2>,1:12]
  [@5,16:16='<',<1>,1:16]
  [@6,17:17='/',<4>,1:17]
  [@7,18:21='name',<5>,1:18]
  [@8,22:22='>',<3>,1:22]
  [@9,23:23='\n',<2>,1:23]
  [@10,24:23='<EOF>',<-1>,2:24]
```

The lexer sends <name> to the parser as the three tokens at indexes 1, 2, and 3. Also notice that Hello, which lives in the sea, would match rule ID but only in ISLAND mode. Since the lexer starts out in default mode, Hello matches as token TEXT. You can see the difference in the token types between tokens at index 0 and 2 where name matches as token ID (token type 5).

Another reason that we want to match tag syntax in the parser instead of the lexer is that the parser has much more flexibility to execute actions. Furthermore, the parser automatically builds a parse tree for us.



To use our grammar for an application, we could either use the usual listener or visitor mechanism or add actions to the grammar. For example, to implement an XML SAX event mechanism, we could shut off the automatic tree construction and embed grammar actions to trigger SAX method calls.

Now that we know how to separate the XML islands from the sea and how to send tag components to a parser, let's build a real XML parser.

12.4 Parsing and Lexing XML

Because XML is a well-defined language, it's a good idea to start our XML project by reviewing the W3C XML language definition.[3] Unfortunately, the XML specification (henceforth *the spec*) is huge, and it's very easy to get lost in all of the details. To make our lives easier, let's get rid of stuff we don't need in order to parse XML files: <!DOCTYPE..> document type definitions (DTDs), <!ENTITY..> entity declarations, and <!NOTATION..> notation declarations. Besides, handling those tags wouldn't teach us anything beyond what we need to handle the other constructs.

We're going to start out by building the syntactic rules for XML. The good news is that we can reuse the informal grammar rules from the spec almost verbatim by changing them to ANTLR notation.

XML Spec to ANTLR Parser Grammar

Using our experience with XML, we could probably come up with a reasonably complete and accurate XML grammar. To make sure we don't forget anything, however, let's filter and condense the spec to its key grammatical rules.

```
document     ::=  prolog element Misc*
prolog       ::=  XMLDecl? Misc*
content      ::=  CharData?
                  ((element | Reference | CDSect | PI | Comment) CharData?)*
element      ::=  EmptyElemTag
              |   STag content ETag
```

3. http://www.w3.org/TR/REC-xml/

```
EmptyElemTag ::=  '<' Name (S Attribute)* S? '/>'
STag         ::=  '<' Name (S Attribute)* S? '>'
ETag         ::=  '</' Name S? '>'
XMLDecl      ::=  '<?xml' VersionInfo EncodingDecl? SDDecl? S? '?>'
Attribute    ::=  Name Eq AttValue
Reference    ::=  EntityRef | CharRef
Misc         ::=  Comment | PI | S
```

There are lots of other rules we'll need, but they'll go into our lexer. This is a good example of where to draw the line, per our discussion in Section 5.6, *Drawing the Line Between Lexer and Parser*, on page 79. The key criterion to follow is whether we need to see inside the element's structure. For example, we don't care about the inside of comments or processing instructions (PI), so we can have the lexer match them as lumps.

Let's compare these informal spec rules with the following complete ANTLR parser grammar. Relative to the grammars we've built for languages like JSON and Cymbol, the XML parser rules are pretty simple.

lexmagic/XMLParser.g4

```
parser grammar XMLParser;
options { tokenVocab=XMLLexer; }

document    :   prolog? misc* element misc*;

prolog      :   XMLDeclOpen attribute* SPECIAL_CLOSE ;

content     :   chardata?
                ((element | reference | CDATA | PI | COMMENT) chardata?)* ;

element     :   '<' Name attribute* '>' content '<' '/' Name '>'
            |   '<' Name attribute* '/>'
            ;

reference   :   EntityRef | CharRef ;

attribute   :   Name '=' STRING ; // Our STRING is AttValue in spec
/** ``All text that is not markup constitutes the character data of
 *  the document.''
 */
chardata    :   TEXT | SEA_WS ;

misc        :   COMMENT | PI | SEA_WS ;
```

There are a number of important differences between the spec's rules and ours. First, the spec rule XMLDecl can match three specific attributes (version, encoding, and standalone), whereas ours matches any set of attributes inside <?xml ...?>. Later, a semantic phase would have to check that the attribute names were correct.

Alternatively, we could put predicates inside the grammar, but it makes the grammar hard to read and would slow down the generated parser.

```
prolog       : XMLDecl versionInfo encodingDecl? standalone? SPECIAL_CLOSE ;
versionInfo  : {_input.LT(1).getText().equals("version")}? Name '=' STRING ;
encodingDecl : {_input.LT(1).getText().equals("encoding")}? Name '=' STRING ;
standalone   : {_input.LT(1).getText().equals("standalone")}? Name '=' STRING ;
```

The next difference is that our lexer will match and discard whitespace inside of tags between the attributes, so we don't need to check for whitespace inside of our element rule. (element is an expanded version of rule tag from the previous section.) Our lexer also differentiates between whitespace (SEA_WS) and non-whitespace text (TEXT) outside of tags but returns both to the parser as tokens. (The previous two sections lumped all text outside of tags into a single TEXT token.) That's because the spec allows whitespace but not text in certain locations such as before the root element. Therefore, chardata is a parser rule, not a token in our grammar.

The XML parser is not too bad, but we're going to earn hazardous-duty pay building the lexer.

Tokenizing XML

Let's start our XML lexer by extracting the relevant rules from the spec to see what we're dealing with.

```
Comment   ::=  '<!--' ((Char - '-') | ('-' (Char - '-')))* '-->'
CDSect    ::=  '<![CDATA[' CData ']]>'
CData     ::=  (Char* - (Char* ']]>' Char*)) // anything but ']]>'
PI        ::=  '<?' PITarget (S (Char* - (Char* '?>' Char*)))? '?>'
/** Any name except 'xml' */
PITarget  ::=  Name - (('X' | 'x') ('M' | 'm') ('L' | 'l'))
/** Spec: ``CharData is any string of characters which does not contain the
 *  start-delimiter of any markup and does not include the
 *  CDATA-section-close delimiter, "]]>".''
 */
CharData  ::=  [^<&]* - ([^<&]* ']]>' [^<&]*)
EntityRef ::=  '&' Name ';'
CharRef   ::=  '&#' [0-9]+ ';'
          |    '&#x' [0-9a-fA-F]+ ';'
Name      ::=  NameStartChar (NameChar)*
NameChar  ::=  NameStartChar | "-" | "." | [0-9] | #xB7
          |    [#x0300-#x036F] | [#x203F-#x2040]
NameStartChar
          ::=  ":" | [A-Z] | "_" | [a-z] | [#xC0-#xD6] | [#xD8-#xF6]
          |    [#xF8-#x2FF] | [#x370-#x37D] | [#x37F-#x1FFF]
          |    [#x200C-#x200D] | [#x2070-#x218F] | [#x2C00-#x2FEF]
          |    [#x3001-#xD7FF] | [#xF900-#xFDCF] | [#xFDF0-#xFFFD]
          |    [#x10000-#xEFFFF]
```

```
AttValue  ::=  '"' ([^<&"] | Reference)* '"'
          |    "'" ([^<&'] | Reference)* "'"
S         ::=  (#x20 | #x9 | #xD | #xA)+
```

Blech! That looks kind of complicated, but we'll break it down into three different modes and build them one by one. We'll need modes to handle outside tags, inside tags, and inside of the special <?...?> tags, very much like we did in *Issuing Context-Sensitive Tokens with Lexical Modes*, on page 221.

When you compare the spec rules to our ANTLR lexer rules, you'll see that we can reuse most of the same rule names. The specs notation is quite different from ANTLR's, but we can reuse the spirit of most rule right sides. Let's start with the default mode that matches the sea outside of tags. Here's that piece of our lexer grammar:

lexmagic/XMLLexer.g4

```
lexer grammar XMLLexer;

// Default "mode": Everything OUTSIDE of a tag
COMMENT     :   '<!--' .*? '-->' ;
CDATA       :   '<![CDATA[' .*? ']]>' ;
/** Scarf all DTD stuff, Entity Declarations like <!ENTITY ...>,
 *  and Notation Declarations <!NOTATION ...>
 */
DTD         :   '<!' .*? '>'            -> skip ;
EntityRef   :   '&' Name ';' ;
CharRef     :   '&#' DIGIT+ ';'
            |   '&#x' HEXDIGIT+ ';'
            ;
SEA_WS      :   (' '|'\t'|'\r'? '\n') ;

OPEN        :   '<'                     -> pushMode(INSIDE) ;
XMLDeclOpen :   '<?xml' S               -> pushMode(INSIDE) ;
SPECIAL_OPEN:   '<?' Name               -> more, pushMode(PROC_INSTR) ;

TEXT        :   ~[<&]+ ;        // match any 16 bit char other than < and &
```

The lexer grammar starts by dealing with all of the lexical structures that we can treat as complete tokens. First, we give rules for COMMENT and CDATA tokens. Next, we match and discard anything related to document, entity, and notation declarations of the following form: <!...>. We don't care about that stuff for this project. We then need rules to match the various entities and the whitespace token. Jumping ahead for a second, rule TEXT matches anything else in the input until the start of a tag or entity reference. This is sort of the "else clause."

And now for the fun stuff. When the lexer sees the start of the tag, it needs to switch contexts so that the next lexer token match will find a token that's valid

within a tag. That's what rule OPEN does. Unlike the ModeTagsLexer grammar that used just the mode command, we're using pushMode (and popMode in a moment). By pushing the mode, the lexer can pop the mode to return to the "invoking" mode. This is useful for nested mode switches, though we're not doing that here.

The next two rules distinguish between the special <?xml...?> tag and the regular <?...?> processing instruction. Because we want the parser prolog rule to match the attributes inside the <?xml...?> tag, we need the lexer to return an XMLDeclOpen token and then switch to the INSIDE tag mode, which will match the attribute tokens. Rule SPECIAL_OPEN matches the start of any other <?...?> tag and then switches to the PROC_INSTR mode (which we'll see shortly). It also uses an unfamiliar lexer command called more that instructs the lexer to look for another token and keep the text of the just-matched token.

Once inside mode PROC_INSTR, we want the lexer to keep consuming and piling up characters via rule IGNORE until it sees the end of the processing instruction, ?>.

lexmagic/XMLLexer.g4

```
mode PROC_INSTR;
PI          :   '?>'                    -> popMode ; // close <?...?>
IGNORE      :   .                       -> more ;
```

This is all a fancy way to match '<?' .*? '?>' for every processing instruction except for the <?xml...?> tag. The SPECIAL_OPEN rule matches <?xml also, but the lexer gives precedence to rule XMLDeclOpen since it's listed first per our discussion in Section 2.3, *You Can't Put Too Much Water into a Nuclear Reactor*, on page 13 . Unfortunately, we can't just have a simple rule like '<?' .*? '?>' and do away with the PROC_INSTR mode. Since '<?' .*? '?>' matches a longer character sequence than '<?xml' S, the lexer would never match XMLDeclOpen. This is similar to the situation in *Avoiding the Maximal Munch Ambiguity*, on page 211 , where the lexer favored one >> token over two > tokens.

Notice that SPECIAL_OPEN references rule Name, which doesn't appear in either mode we've looked at. It appears in the INSIDE mode we'll look at next. Modes just tell the lexer which set of rules it should consider matching when asked for a token. It's OK for one rule to call another in a different mode as a helper. But, keep in mind that the lexer can return token types to the parser only from those defined within the current lexer mode.

Our final mode is the INSIDE mode, which recognizes all of the elements within tags like this:

```
title id="chap2", center="true"
```

The lexical structure within tags reinforces the idea from Section 5.3, *Recognizing Common Language Patterns with ANTLR Grammars*, on page 61, that

many languages look the same from a lexical perspective. For example, a lexer for C would have no problem tokenizing that tag content.

Here's our (final) mode that handles the structures within tags:

lexmagic/XMLLexer.g4

```
mode INSIDE;

CLOSE       :   '>'                     -> popMode ;
SPECIAL_CLOSE:  '?>'                    -> popMode ; // close <?xml...?>
SLASH_CLOSE :   '/>'                    -> popMode ;
SLASH       :   '/' ;
EQUALS      :   '=' ;
STRING      :   '"' ~[<"]* '"'
            |   '\'' ~[<']* '\''
            ;
Name        :   NameStartChar NameChar* ;
S           :   [ \t\r\n]               -> skip ;

fragment
HEXDIGIT    :   [a-fA-F0-9] ;

fragment
DIGIT       :   [0-9] ;

fragment
NameChar    :   NameStartChar
            |   '-' | '.' | DIGIT
            |   '\u00B7'
            |   '\u0300'..'\u036F'
            |   '\u203F'..'\u2040'
            ;

fragment
NameStartChar
            :   [:a-zA-Z]
            |   '\u2070'..'\u218F'
            |   '\u2C00'..'\u2FEF'
            |   '\u3001'..'\uD7FF'
            |   '\uF900'..'\uFDCF'
            |   '\uFDF0'..'\uFFFD'
            ;
```

The first three rules match the end tag sequences. Here is where the popMode lexical command comes in handy. We don't have to specify the mode to switch to; the rules can just say "pop." The previous mode is on a mode stack.

Rule STRING matches rule AttValue from the spec and differs only in that STRING does not specifically match entities inside strings. We don't care about the inside of strings, so there's no point in carefully matching those characters.

We just have to make sure that we don't allow < or quotes inside of strings, per the spec.

Now that we have a parser and lexer grammar, let's build and test them.

Testing Our XML Grammar

As usual, we need to run ANTLR on the two grammars, being careful to process the lexer first because the parser depends on the token types generated by ANTLR for XMLLexer.g4.

```
$ antlr4 XMLLexer.g4
$ antlr4 XMLParser.g4
$ javac XML*.java
```

Here's a sample XML input file:

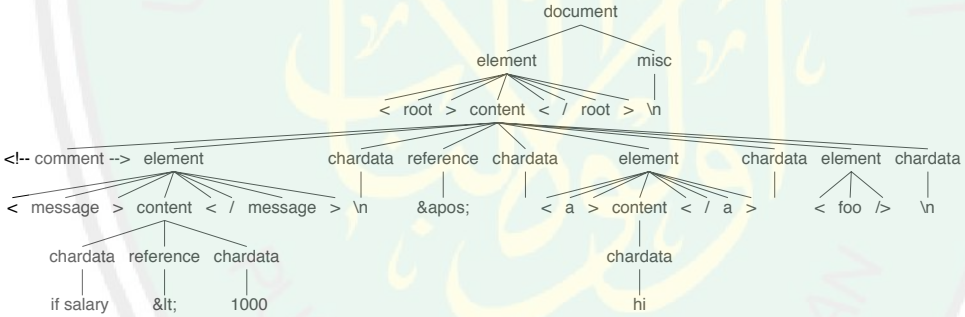
lexmagic/XML-inputs/entity.xml

```
<!-- a comment
-->
<root><!-- comment --><message>if salary &lt; 1000</message>
&apos; <a>hi</a>  <foo/>
</root>
```

Let's use grun to generate a parse tree.

```
$ grun XML document -gui XML-inputs/entity.xml
```

The parse tree indicates that our parser correctly handles comments, entities, tags, and text.



Now, let's check to make sure that our parser correctly handles <?xml...?> vs. other processing instruction tags. Here's a sample input file:

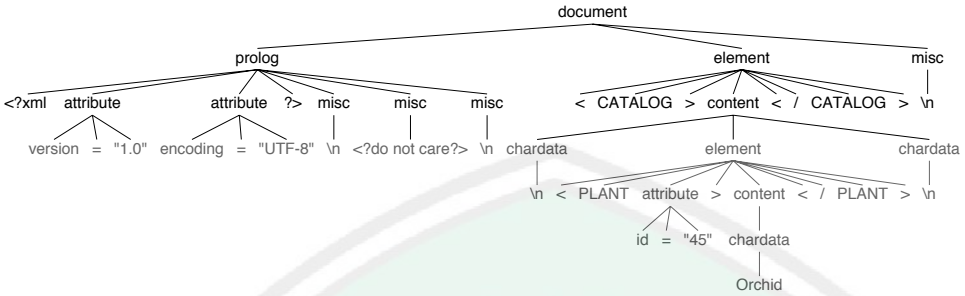
lexmagic/XML-inputs/cat.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<?do not care?>
<CATALOG>
<PLANT id="45">Orchid</PLANT>
</CATALOG>
```

We can generate a parse tree with this:

```
$ grun XML document -ps /tmp/t.ps XML-inputs/cat.xml
```

The following tree shows that the XML declaration tag is properly sent in pieces to the parser, whereas <?do not care?> comes as one PI chunk.



Most of the lexer rules in mode INSIDE deal with properly matching tag names, using all of the valid Unicode code points. That allows us to recognize, for example, XML files with Japanese tag names.[4] Running sample file weekly-euc-jp.xml into our parser requires the proper Japanese encoding option for grun.

```
$ grun XML document -gui -encoding euc-jp XML-inputs/weekly-euc-jp.xml
```

Figure 12, *A window into a dialog box*, on page 232 shows a window into the much larger dialog box.

This XML grammar is a great example of how complexity often resides in the lexer. Parsers can often be big, but they're usually not that difficult. When a language is hard to recognize, it's usually because it's hard to group the characters into tokens. This is either because the lexer really needs syntactic context to make decisions or because there are different regions in the file with different lexical rules.

This chapter is big, and we went through a lot of material, but it will serve as a good resource when you run into challenging recognition problems. We started out learning how to send different tokens on different channels so that we can ignore but not throw out key tokens such as comments and whitespace. Next, we looked at how to solve some context-sensitive lexical problems such as the pesky keywords-as-identifiers problem. Then, we learned how to tokenize multiple regions of the input differently using lexical modes, separating the islands from the sea. In our grand finale, we used lexical modes to build a precise XML lexer.

4. http://people.apache.org/~edwingo/jaxp-ri-1.2.0-fcs/samples/data/weekly-euc-jp.xml

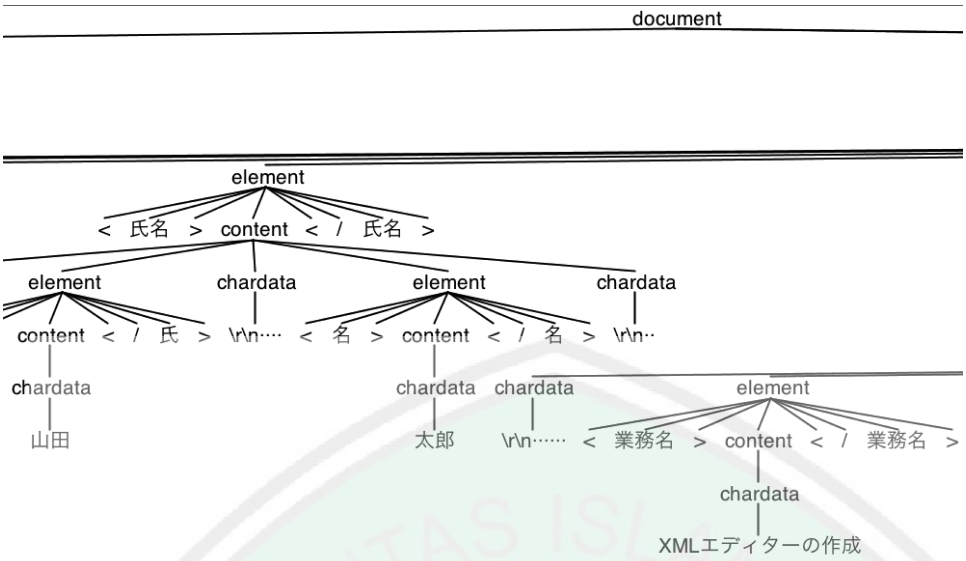


Figure 12—A window into a dialog box

At this point, we've learned a great deal about how to use ANTLR. The next part of the book is a reference section that fills in a lot of details we avoided for clarity earlier in the book.

Part IV

ANTLR Reference

The first three parts of this book were a guide to using ANTLR, whereas this final part is primarily reference material. We'll start by summarizing the runtime API and looking at how ANTLR handles left-recursive rules. And, finally, we'll see the giant reference chapter.

CHAPTER 13

Exploring the Runtime API

This chapter gives an overview of the ANTLR runtime API and is meant to kick-start your exploration of the runtime library. It describes the programmer-facing classes but does not reproduce the details from the Javadoc.[1] Please see the comments on the classes and individual methods for detailed information on their usage.

13.1 Library Package Overview

ANTLR's runtime consists of six packages, with most of the application-facing classes in the main org.antlr.v4.runtime package. By far the most common classes are the ones used to launch a parser on some input. Here is the typical code snippet for a grammar file called X.g and a parse-tree listener called MyListener that implements XListener:

```
XLexer lexer = new XLexer(input);
CommonTokenStream tokens = new CommonTokenStream(lexer);
XParser parser = new XParser(tokens);
ParseTree tree = parser.XstartRule();

ParseTreeWalker walker = new ParseTreeWalker();
MyListener listener = new MyListener(parser);
walker.walk(listener, tree);
```

We first encountered this in Section 3.3, *Integrating a Generated Parser into a Java Program*, on page 26.

Here is a summary of the packages:

org.antlr.v4.runtime This package contains the most commonly used classes and interfaces, such as the hierarchies for input streams, character and token buffers, error handling, token construction, lexing, and parsing.

1. http://www.antlr.org/api

org.antlr.v4.runtime.atn This is used internally for ANTLR's Adaptive *LL(*)* lexing and parsing strategy. The *atn* term means *augmented transition network*[2] and is a state machine that can represent a grammar where edges represent grammar elements. ANTLR walks the ATN during lexing and parsing to make predictions based upon lookahead symbols.

org.antlr.v4.runtime.dfa Using the ATN to make predictions is expensive, so the runtime caches prediction results in *deterministic finite automata* (DFA).[3] This package holds all of the DFA implementation classes.

org.antlr.v4.runtime.misc This package holds miscellaneous data structures but also the commonly used TestRig class that we've used throughout this book via command-line alias grun.

org.antlr.v4.runtime.tree ANTLR-generated parsers build parse trees by default, and this package holds all of the classes and interfaces needed to implement them. It also contains the basic parse-tree listener, walker, and visitor mechanisms.

org.antlr.v4.runtime.tree.gui ANTLR ships with a basic parse tree viewer accessible via tree method inspect(). You can also save trees in PostScript form via save(). TestRig's -gui option launches this viewer.

The remaining sections describe the runtime API grouped by functionality.

13.2 Recognizers

ANTLR generates lexers and parsers that are subclasses of Lexer and Parser. Superclass Recognizer abstracts the notion of recognizing structure within a sequence of characters or tokens. Recognizers feed off of IntStreams, which we'll look at later. Here is the inheritance and interface implementation relationships (interfaces are in italics).

Lexer implements interface TokenSource, which specifies the core lexer functionality: nextToken(), getLine(), and getCharPositionInLine(). Rolling our own lexer to use with an ANTLR parser grammar is not too much work. Let's build a lexer that tokenizes simple identifiers and integers like the following input file:

2. http://en.wikipedia.org/wiki/Augmented_transition_network
3. http://en.wikipedia.org/wiki/Deterministic_finite_automaton

api/Simple-input

```
a 343x
abc 9 ;
```

Here's the core of a handbuilt lexer:

api/SimpleLexer.java

```
@Override
public Token nextToken() {
    while (true) {
        if ( c==(char)CharStream.EOF ) return createToken(Token.EOF);
        while ( Character.isWhitespace(c) ) consume(); // toss out whitespace
        startCharIndex = input.index();
        startLine = getLine();
        startCharPositionInLine = getCharPositionInLine();
        if ( c==';' ) {
            consume();
            return createToken(SEMI);
        }
        else if ( c>='0' && c<='9' ) {
            while ( c>='0' && c<='9' ) consume();
            return createToken(INT);
        }
        else if ( c>='a' && c<='z' ) { // VERY simple ID
            while ( c>='a' && c<='z' ) consume();
            return createToken(ID);
        }
        // error; consume and try again
        consume();
    }
}

protected Token createToken(int ttype) {
    String text = null; // we use start..stop indexes in input
            Pair<TokenSource, CharStream> source =
                    new Pair<TokenSource, CharStream>(this, input);
    return factory.create(source, ttype, text, Token.DEFAULT_CHANNEL,
                          startCharIndex, input.index()-1,
                          startLine, startCharPositionInLine);
}

protected void consume() {
    if ( c=='\n' ) {
        line++;  // \r comes back as a char, but \n means line++
        charPositionInLine = 0;
    }
    if ( c!=(char)CharStream.EOF ) input.consume();
    c = (char)input.LA(1);
    charPositionInLine++;
}
```

With a handbuilt lexer, we need a way to share the same token names in the ANTLR parser grammar. For parser code generation, we also need to inform ANTLR of the token type integer values established in the lexer source. This is the role of the .tokens file.

api/SimpleLexer.tokens
```
ID=1
INT=2
SEMI=3
```

Here's a simple grammar that feeds off of those token definitions:

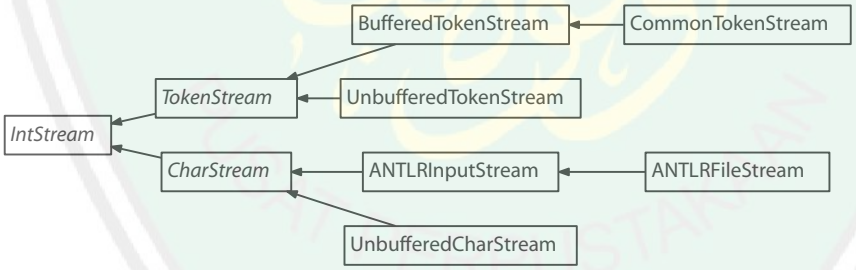
api/SimpleParser.g4
```
parser grammar SimpleParser;
options {
  // get token types from SimpleLexer.tokens; don't name it
  // SimpleParser.tokens as ANTLR will overwrite!
  tokenVocab=SimpleLexer;
}

s : ( ID | INT )* SEMI ;
```

And here's the build and test sequence:

```
$ antlr4 SimpleParser.g4
$ javac Simple*.java TestSimple.java
$ java TestSimple Simple-input
(s a 343 x abc 9 ;)
```

13.3 Input Streams of Characters and Tokens

At the most abstract level, both lexers and parsers check the syntax of integer streams. Lexers process characters (short integers), and parsers process token types (integers). That is why the root of the ANTLR input stream class hierarchy is called IntStream.



Interface IntStream defines most of the key operations for a stream, including methods to consume symbols and fetch lookahead symbols, namely, consume() and LA(). Because ANTLR recognizers need to scan ahead and rewind the input, IntStream defines the mark() and seek() methods.

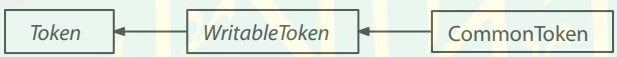
The CharStream and TokenStream subinterfaces add methods to extract text from the streams. The classes implementing those interfaces typically read all of the input in one go and buffer it. That's because it is easier to build those classes, it provides ready access to the input, and it suits the common case. If your input is too big to buffer or is infinite (for example, via a socket), you can use UnbufferedCharStream and UnbufferedTokenStream.

The usual code sequence to perform a parse is to create an input stream, attach a lexer to it, create a token stream attached to the lexer, and then create a parser attached to the token stream.

```
ANTLRInputStream input = new ANTLRFileStream("an-input-file");
//ANTLRInputStream input = new ANTLRInputStream(System.in); // or read stdin
SimpleLexer lexer = new SimpleLexer(input);
CommonTokenStream tokens = new CommonTokenStream(lexer);
SimpleParser parser = new SimpleParser(tokens);
ParseTree t = parser.s();
```

13.4 Tokens and Token Factories

Lexers break up character streams into Token objects, and parsers apply grammatical structure to the stream of tokens. Generally, we think of tokens as immutable after construction in the lexer, but sometimes we need to alter token fields after we've created them. For example, the token streams like to set the token index of the tokens as they go by. To support this, ANTLR uses interface WritableToken, which is a kind of Token with "setter" methods. Finally, we have CommonToken that implements a full-featured token.



We usually don't need to define our own kind of tokens, but it's a useful capability. Here's a sample specialized Token implementation that adds a field to every token object:

api/MyToken.java

```
import org.antlr.v4.runtime.CharStream;
import org.antlr.v4.runtime.CommonToken;
import org.antlr.v4.runtime.TokenSource;
import org.antlr.v4.runtime.misc.Pair;

/** A Token that tracks the TokenSource name in each token. */
public class MyToken extends CommonToken {
    public String srcName;

    public MyToken(int type, String text) {
        super(type, text);
    }
```

```
    public MyToken(Pair<TokenSource, CharStream> source, int type,
                   int channel, int start, int stop)
    {
        super(source, type, channel, start, stop);
    }

    @Override
    public String toString() {
        String t = super.toString();
        return srcName +":"+t;
    }
}
```

To get the lexer to create these special tokens, we need to create a factory and pass it to the lexer. We also tell the parser so that its error handler can conjure up the right kind of tokens if necessary.

Here's a token factory that creates MyToken objects:

api/MyTokenFactory.java

```
import org.antlr.v4.runtime.CharStream;
import org.antlr.v4.runtime.TokenFactory;
import org.antlr.v4.runtime.TokenSource;
import org.antlr.v4.runtime.misc.Interval;
import org.antlr.v4.runtime.misc.Pair;

/** A TokenFactory that creates MyToken objects */
public class MyTokenFactory implements TokenFactory<MyToken> {
    CharStream input;

    public MyTokenFactory(CharStream input) { this.input = input; }
    @Override
    public MyToken create(int type, String text) {
        return new MyToken(type, text);
    }
    @Override
    public MyToken create(Pair<TokenSource, CharStream> source, int type,
                          String text,
                          int channel, int start, int stop, int line,
                          int charPositionInLine)
    {
        MyToken t = new MyToken(source, type, channel, start, stop);
        t.setLine(line);
        t.setCharPositionInLine(charPositionInLine);
        t.srcName = input.getSourceName();
        return t;
    }
}
```

And here's some sample code that notifies the lexer and parser of the factory:

api/TestSimpleMyToken.java

```
ANTLRInputStream input = new ANTLRFileStream(args[0]);
SimpleLexer lexer = new SimpleLexer(input);
MyTokenFactory factory = new MyTokenFactory(input);
lexer.setTokenFactory(factory);
CommonTokenStream tokens = new CommonTokenStream(lexer);

// now, print all tokens
tokens.fill();
List<Token> alltokens = tokens.getTokens();
for (Token t : alltokens) System.out.println(t.toString());

// now parse
SimpleParser parser = new SimpleParser(tokens);
parser.setTokenFactory(factory);
ParseTree t = parser.s();
System.out.println(t.toStringTree(parser));
```

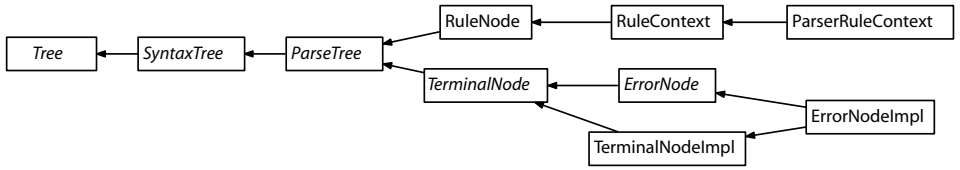
It reuses the SimpleParser.g4 grammar from before. Here's the build and test sequence:

```
$ antlr4 SimpleParser.g4
$ javac Simple*.java MyToken*.java TestSimpleMyToken.java
$ java TestSimpleMyToken Simple-input
Simple-input:[@0,0:0='a',<1>,1:0]
Simple-input:[@1,2:4='343',<2>,1:2]
Simple-input:[@2,5:5='x',<1>,1:5]
Simple-input:[@3,7:9='abc',<1>,2:1]
Simple-input:[@4,11:11='9',<2>,2:5]
Simple-input:[@5,13:13=';',<3>,2:7]
Simple-input:[@6,15:14='<EOF>',<-1>,3:1]
(s a 343 x abc 9 ;)
```

The toString() method in MyToken adds the Simple-input: prefix to the normal token string representation.

13.5 Parse Trees

Interface Tree defines the basic notion of a tree that has a payload and children. A SyntaxTree is a tree that knows how to associate tree nodes with tokens in a TokenStream. Getting more specific, interface ParseTree represents a node in a parse tree. It knows how to return the text associated with all leaf nodes below it in the tree. We saw sample parse trees in Section 2.4, *Building Language Applications Using Parse Trees*, on page 16, and how the nodes correspond to the types in this class hierarchy. ParseTree also provides the usual visitor pattern double-dispatch accept() method for ParseTreeVisitor, which we looked at in Section 2.5, *Parse-Tree Listeners and Visitors*, on page 17.

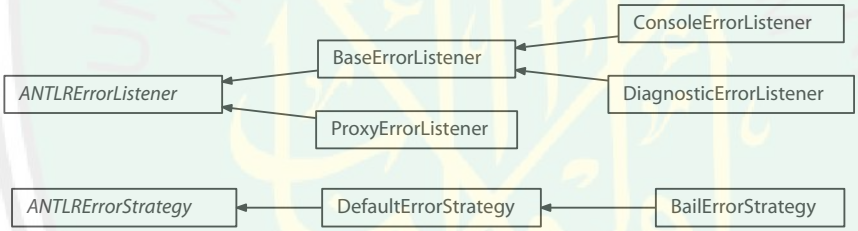


Classes RuleNode and TerminalNode correspond to subtree roots and leaf nodes. ANTLR creates ErrorNodeImpl nodes during single-token-insertion recovery (see *Recovering from Mismatched Tokens*, on page 162).

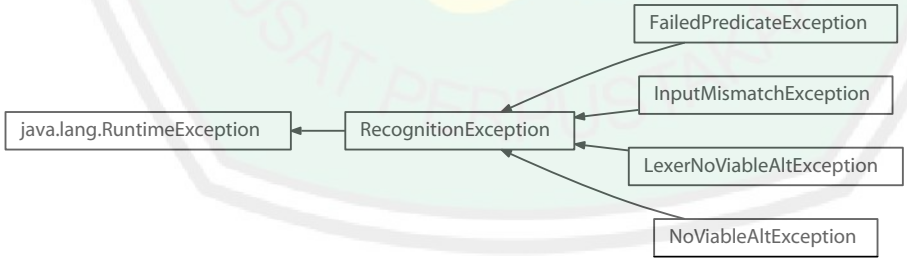
A RuleContext is a record of a single rule invocation and knows which context invoked it, if any, by walking up the getParent() chain. ParserRuleContext has a field to track parse-tree children, if the parser is creating trees. These are primarily implementation classes so you can focus on the specialized subclasses that ANTLR generates for each rule in your grammar.

13.6 Error Listeners and Strategies

There are two key interfaces associated with ANTLR's syntax error mechanism: ANTLRErrorListener and ANTLRErrorStrategy. We learned about the former in Section 9.2, *Altering and Redirecting ANTLR Error Messages*, on page 153, and the latter in Section 9.5, *Altering ANTLR's Error Handling Strategy*, on page 171. Listeners let us alter error messages and where they go. Strategy implementations alter how parsers react to errors.



ANTLR throws specific RecognitionExceptions according to the error. Note that they are unchecked runtime exceptions, so you don't have to specify throws clauses all the time in your methods.



13.7 Maximizing Parser Speed

ANTLR v4's adaptive parsing strategy is more powerful than v3's, but it comes at the cost of a little bit of speed. If you need the most speed and the smallest memory footprint possible, you can do a two-step parsing strategy. The first step uses a slightly weaker parsing strategy, *SLL(*)*, that almost always works. (It's very similar to v3's strategy, except it doesn't need to backtrack.) If the first parsing step fails, you have to try the full *LL(*)* parse. After failing the first step, we don't know whether it's a true syntax error or whether it's because the *SLL(*)* strategy wasn't strong enough. Input that passes the *SLL(*)* step is guaranteed to pass the full *LL(*)*, so there's no point in trying out that more expensive strategy.

```
parser.getInterpreter().setSLL(true); // try with simpler/faster SLL(*)
// we don't want error messages or recovery during first try
parser.removeErrorListeners();
parser.setErrorHandler(new BailErrorStrategy());
try {
    parser.startRule();
    // if we get here, there was no syntax error and SLL(*) was enough;
    // there is no need to try full LL(*)
}
catch (RuntimeException ex) {
    if (ex.getClass() == RuntimeException.class &&
        ex.getCause() instanceof RecognitionException)
    {
        // The BailErrorStrategy wraps the RecognitionExceptions in
        // RuntimeExceptions so we have to make sure we're detecting
        // a true RecognitionException not some other kind
        tokens.reset(); // rewind input stream
        // back to standard listeners/handlers
        parser.addErrorListener(ConsoleErrorListener.INSTANCE);
        parser.setErrorHandler(new DefaultErrorStrategy());
        parser.getInterpreter().setSLL(false); // try full LL(*)
        parser.startRule();
    }
}
```

Input that fails the second step is truly syntactically invalid.

13.8 Unbuffered Character and Token Streams

Because ANTLR recognizers buffer up the entire input character stream and all input tokens by default, they can't handle input files that are bigger than a computer's memory and can't handle infinite streams like socket connections. To overcome this, you can use unbuffered versions of the character

and token streams, which keep just a small sliding window into the streams: UnbufferedCharStream and UnbufferedTokenStream.

As a demonstration, here's a modification of the comma-separated-value grammar from Section 6.1, *Parsing Comma-Separated Values*, on page 84 that sums the floating-point values in a two-column file:

api/CSV.g4

```
/** Rows are two real numbers:
    0.9962269825793676, 0.9224608616182103
    0.91673278673353,  -0.6374985722530822
    0.9841464019977713, 0.03539546030010776
    ...
 */
grammar CSV;

@members {
double x, y; // keep column sums in these fields
}

file: row+ {System.out.printf("%f, %f\n", x, y);} ;

row : a=field ',' b=field '\r'? '\n'
      {
      x += Double.valueOf($a.start.getText());
      y += Double.valueOf($b.start.getText());
      }
    ;

field
    : TEXT
    ;

TEXT : ~[,\n\r]+ ;
```

If all you care about are the sums of the columns, you need to keep only one or two tokens in memory at once. To prevent complete buffering, there are three things to do. First, use the unbuffered streams instead of the usual ANTLFileStream and CommonTokenStream. Second, pass the lexer a token factory that copies characters from the input stream into the text of the tokens. Otherwise, the getText() method for tokens would try to access the input character stream, which probably would no longer be available. (See the diagram in Section 2.4, *Building Language Applications Using Parse Trees*, on page 16 that shows the relationship between tokens and the character stream.) Finally, ask the parser not to create parse trees. The following test rig has the key lines highlighted:

api/TestCSV.java

```
import org.antlr.v4.runtime.CharStream;
import org.antlr.v4.runtime.CommonToken;
import org.antlr.v4.runtime.CommonTokenFactory;
import org.antlr.v4.runtime.Token;
import org.antlr.v4.runtime.TokenStream;
import org.antlr.v4.runtime.UnbufferedCharStream;
import org.antlr.v4.runtime.UnbufferedTokenStream;

import java.io.FileInputStream;
import java.io.InputStream;
public class TestCSV {
    public static void main(String[] args) throws Exception {
        String inputFile = null;
        if ( args.length>0 ) inputFile = args[0];
        InputStream is = System.in;
        if ( inputFile!=null ) {
            is = new FileInputStream(inputFile);
        }
        CharStream input = new UnbufferedCharStream(is);
        CSVLexer lex = new CSVLexer(input);
        // copy text out of sliding buffer and store in tokens
        lex.setTokenFactory(new CommonTokenFactory(true));
        TokenStream tokens = new UnbufferedTokenStream<CommonToken>(lex);
        CSVParser parser = new CSVParser(tokens);
        parser.setBuildParseTree(false);
        parser.file();
    }
}
```

Here's a sample build and test sequence using a 1,000-line sample file:

```
$ antlr4 CSV.g4
$ javac TestCSV.java CSV*.java
$ wc sample.csv
    1000    2000   39933 sample.csv # 1000 lines, 2000 words, 39933 char
$ java TestCSV sample.csv
1000.542053, 1005.587845
```

To verify that the recognizer is not buffering up everything, I ran a 310M CSV input file with 7.8M value pairs into the test rig while restricting the Java VM to just 10M RAM.

```
$ wc big.csv
 7800000 15600000 310959090 big.csv # 7800000 lines, ...
$ time java -Xmx10M TestCSV big.csv
11695395.953785, 7747174.349207

real    0m43.415s # wall clock duration to compute the sums
user    0m51.186s
sys     0m6.195s
```

These unbuffered streams are useful when efficiency is the top concern. (You can even combine them with the technique from the previous section.) Their disadvantage is that you are forced to buffer things up manually. For example, you can't use $text in an action embedded within a rule because it goes to the input stream and asks for the text (and the text isn't being buffered).

13.9 Altering ANTLR's Code Generation

ANTLR uses two things to generate code: a StringTemplate[4] group file (containing templates) and a Target subclass called *Language*Target where *Language* is the grammar language option. The StringTemplate group file is org/antlr/v4/tool/templates/codegen/*Language*.stg. If you would like to tweak the code generation templates for, say, Java, all you have to do is copy and modify org/antlr/v4/tool/templates/codegen/Java.stg. Then, put it in the CLASSPATH *before* ANTLR's jar. ANTLR uses a resource loader to get those templates so it'll see your modified version first.

The templates just generate code specific to a grammar. Most of the common functionality has been factored out into the runtime library. So, Lexer, Parser and so on are all part of the runtime library, not generated by ANTLR.

To add a new target for language *L*, you might need to create class *L*Target. If so, place it in package org.antlr.v4.codegen and put it before ANTLR's jar in the CLASSPATH. You need this class only if your target needs to alter some of the default functionality in Target. If no *L*Target class is found, ANTLR uses the Target base class. (This is what it does for the Java language target.)

4. http://www.stringtemplate.org

CHAPTER 14

Removing Direct Left Recursion

In Section 5.4, *Dealing with Precedence, Left Recursion, and Associativity*, on page 69, we saw that the natural way to specify arithmetic expressions grammatically is ambiguous. For example, the following expr can interpret 1+2*3 as (1+2)*3 or 1+(2*3). By giving precedence to the alternatives specified first, however, ANTLR neatly sidesteps the ambiguity.

left-recursion-removal/Expr.g4

```
stat: expr ';' ;

expr:   expr '*' expr       // precedence 4
    |   expr '+' expr       // precedence 3
    |   INT                 // primary (precedence 2)
    |   ID                  // primary (precedence 1)
    ;
```

Rule expr is still left-recursive, though, which traditional top-down grammars (for example, ANTLR v3) cannot handle. In this chapter, we're going to explore how ANTLR deals with left recursion and how it handles operator precedence. In a nutshell, ANTLR replaces left recursion with a (...)* that compares the precedence of the previous and next operators.

It's important to get familiar with the rule transformation because the generated code reflects the transformed rule, not the original. More importantly, when a grammar doesn't give the expected grouping or associativity for operators, we need to know why. Most users can stop reading after the next section that shows the valid recursive alternative patterns; advanced users interested in the implementation details can continue to the second section.

Let's start by looking at the transformations that ANTLR performs and then walk through an example to see the *precedence climbing* in action.[1]

1. Theodore Norvell coined the term (http://www.engr.mun.ca/~theo/Misc/exp_parsing.htm), but the original work was done by Keith Clarke (http://antlr.org/papers/Clarke-expr-parsing-1986.pdf).

14.1 Direct Left-Recursive Alternative Patterns

ANTLR examines any left-recursive rule looking for one of four subexpression operator patterns.

binary Any alternative of the form expr *op* expr or expr (*op1*|*op2*|...|*opN*) expr. *op* can be a single-token or multitoken operator. For example, a Java grammar might treat angle brackets individually instead of treating operators <=> and >= as single tokens. Here's an alternative that handles comparison operators at the same precedence level:

```
expr: ...
    | expr ('<' '=' | '>' '=' | '>' | '<') expr
   ...
    ;
```

op can also be a rule reference. For example, we can factor out those tokens into another rule.

```
expr: ...
    | expr compareOps expr
   ...
    ;
compareOps : ('<' '=' | '>' '=' | '>' | '<') ;
```

ternary Any alternative of the form expr *op1* expr *op2* expr. *op1* and *op2* must be single-token references. This pattern handles the ?: operator in C-derived languages:

```
expr: ...
    | expr '?' expr ':' expr
   ...
    ;
```

unary prefix Any alternative of the form *elements* expr. ANTLR recognizes any sequence of elements followed by a tail-recursive rule reference as a unary prefix operation, as long as the alternative does not fit the binary or ternary pattern. Here are two alternatives with prefix operators:

```
expr: ...
    | '(' type ')' expr
   ...
    | ('+'|'-'|'++'|'--') expr
   ...
    ;
```

unary suffix Any alternative of the form expr *elements*. As with the prefix pattern, ANTLR recognizes alternatives with a direct left-recursive rule reference followed by any sequence of elements, as long as it doesn't fit

the binary or ternary pattern. Here are two alternatives with suffix operators:

```
expr: ...
    | expr '.' Identifier
   ...
    | expr '.' 'super' '(' exprList? ')'
   ...
    ;
```

Any other alternative pattern is considered a *primary expression* element like an identifier or an integer but includes things like '(' expr ')' because it doesn't fit an operator pattern. This makes sense because the whole point of parentheses is to treat the enclosed expression as a single atomic element. These "other" alternatives can actually appear in any order. ANTLR collects and deals with them properly. The order of all other alternatives matters. Here are a few sample primary expression alternatives:

```
expr: ...
    |   literal
    |   Identifier
    |   type '.' 'class'
   ...
    ;
```

Unless otherwise specified, ANTLR assumes that all operators are left associative. In other words, `1+2+3` groups like this: `(1+2)+3`. Some operators, however, are right associative, such as assignment and exponentiation, as we saw in Section 5.4, *Dealing with Precedence, Left Recursion, and Associativity*, on page 69. To specify right associativity, use the `assoc` token option.

```
expr: expr '^'<assoc=right> expr
   ...
    | expr '='<assoc=right> expr
   ...
    ;
```

In the next section, we'll take a look at how ANTLR translates these patterns.

14.2 Left-Recursive Rule Transformations

If you turn on the `-Xlog` ANTLR command-line option, you can find the transformed left-recursive rules in the log file. Here's what happens to rules `stat` and `expr` from `Expr.g4` shown earlier:

```
// use "antlr4 -Xlog Expr.g4" to see transformed rules
stat:   expr[0] ';' ;  // match an expr whose operators have any precedence

expr[int _p]           // _p is expected minimum precedence level
```

```
:   ( INT          // match primaries (non-operators)
    | ID
    )
    // match operators as long as their precedence is at or higher than
    // expected minimum
    ( {4 >= $_p}? '*' expr[5]  // * has precedence 4
    | {3 >= $_p}? '+' expr[4]  // + has precedence 3
    )*
;
```

Whoa! That's quite a transformation. Don't worry about how ANTLR conjures up all of those parameters to expr. We're mainly interested in learning a bit about how those predicates test operator precedence to direct the parse and get the right groupings.

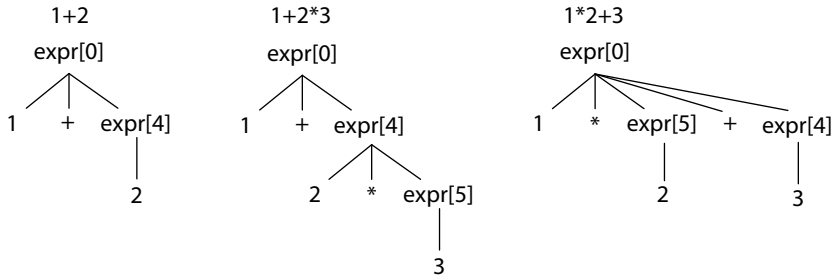
The key is deciding whether to match the next operator in the current invocation of expr or to let the calling invocation of expr match the next operator. The (...)* loop matches operator and right operand pairs. For input 1+2*3, the loop would match +2 and *3. The purpose of the predicates in the loop alternatives is to decide whether the parser should match the operator/operand pair immediately or fall out of the current invocation of expr. For example, predicate {3 >= $_p}? deactivates the addition alternative if the addition operator's precedence, 3, is below the expected minimum precedence, _p, for the current subexpression.

This Ain't Operator Precedence Parsing

Don't confuse this mechanism with *operator precedence parsing*, despite what you read on Wikipedia.[a] Operator precedence parsing can't handle things like the minus sign that has two different precedences, one for unary negation and one for binary subtraction operators. It also can't handle alternatives that have two adjacent rule references like expr ops expr. See *Compilers: Principles, Techniques, and Tools [ALSU06]* to get the real definition.

a. http://en.wikipedia.org/wiki/Operator-precedence_parser

Parameter _p's value is always derived from the precedence of the previous operator. _p starts at 0, since nonrecursive calls to expr pass 0, like stat does: expr[0]. To see _p in action, let's look at some parse trees derived from the transformed rule (showing the value of parameter _p in square brackets). Note that these parse trees are not what ANTLR would build for us from the original left-recursive rule. These are the parse trees for the transformed rule, not the original. Here are some sample inputs and associated parse trees:



In the first tree, the initial call to expr has _p of 0, and expr immediately matches the 1 to the (INT|ID) subrule. Now expr has to decide whether it will match the + or skip the loop entirely and return. The predicate evaluates as {3>=0}? and so we enter the loop to match + and then call expr recursively with an argument of 4. That invocation matches 2 and immediately returns because there is no more input. expr[0] then returns to the original call to expr in stat.

The second tree illustrates how expr[0] matches 1, and again {3>=0}? allows us to match the + operator followed by the second operand, expr[4]. The recursive call to expr[4] matches the 2 and then evaluates {4 >= 4}?, which lets the parser proceed to match the * operator followed by the last operand, 3, via a call to expr[5].

The third parse tree is the most interesting. The initial invocation, expr[0], matches 1 and then decides to match the * operation because {4>=0}? is true. That loop then recursively calls expr[5], which immediately matches 2. Now, inside the call to expr[5], the parser should not match the + because otherwise the 2+3 would evaluate before the multiply. (In the parse tree, we would see expr[5] with 2+3 as children instead of just the 2.) Predicate {3 >= 5}? deactivates that alternative and so expr[5] returns without matching the +. After returning, expr[0] matches +3 since {3>=0}? is true.

I hope this gives you a good feel for the precedence climbing mechanism. If you'd like to learn more, Norvell's description[2] goes into a lot of detail.

2. http://www.engr.mun.ca/~theo/Misc/exp_parsing.htm

CHAPTER 15

Grammar Reference

Most of this book is a guide to using ANTLR. This chapter is a reference and summarizes grammar syntax and the key semantics of ANTLR grammars. It is not meant as an isolated and complete description of how to use ANTLR. The source code for all examples in this book is a good resource and is available at the website.[1]

15.1 Grammar Lexicon

The lexicon of ANTLR is familiar to most programmers because it follows the syntax of C and its derivatives with some extensions for grammatical descriptions.

Comments

There are single-line, multiline, and Javadoc-style comments.

```
/** This grammar is an example illustrating the three kinds
 *  of comments.
 */
grammar T;

/* a multi-line
   comment
 */

/** This rule matches a declarator for my language */
decl : ID ; // match a variable name
```

The Javadoc comments are sent to the parser and are not ignored. These are allowed only at the start of the grammar and any rule.

1. http://pragprog.com/book/tpantlr2/the-definitive-antlr-4-reference

Identifiers

Token names always start with a capital letter and so do lexer rules as defined by Java's Character.isUpperCase() method. Parser rule names always start with a lowercase letter (those that fail Character.isUpperCase()). The initial character can be followed by uppercase and lowercase letters, digits, and underscores. Here are some sample names:

```
ID, LPAREN, RIGHT_CURLY // token names/rules
expr, simpleDeclarator, d2, header_file // rule names
```

Like Java, ANTLR accepts Unicode characters in ANTLR names.

```
grammar 外;
a:'外';
```

To support Unicode parser and lexer rule names, ANTLR uses the following rule:

```
ID  :   a=NameStartChar NameChar*
        {
        if ( Character.isUpperCase(getText().charAt(0)) ) setType(TOKEN_REF);
        else setType(RULE_REF);
        }
    ;
```

NameChar identifies the valid identifier characters.

```
fragment
NameChar
    :   NameStartChar
    |   '0'..'9'
    |   '_'
    |   '\u00B7'
    |   '\u0300'..'\u036F'
    |   '\u203F'..'\u2040'
    ;
```

NameStartChar is the list of characters that can start an identifier (rule, token, or label name).

```
fragment
NameStartChar
    :   'A'..'Z' | 'a'..'z'
    |   '\u00C0'..'\u00D6'
    |   '\u00D8'..'\u00F6'
    |   '\u00F8'..'\u02FF'
    |   '\u0370'..'\u037D'
    |   '\u037F'..'\u1FFF'
    |   '\u200C'..'\u200D'
```

```
|   '\u2070'..'\u218F'
|   '\u2C00'..'\u2FEF'
|   '\u3001'..'\uD7FF'
|   '\uF900'..'\uFDCF'
|   '\uFDF0'..'\uFFFD'
;
```

These more or less correspond to isJavaIdentifierPart() and isJavaIdentifierStart() in Java's Character class. Make sure to use the -encoding option on the ANTLR tool if your grammar file is not in UTF-8 format so that ANTLR reads characters properly.

Literals

ANTLR does not distinguish between character and string literals like most languages do. All literal strings that are one or more characters in length are enclosed in single quotes such as ';', 'if', '>=', and '\'' (refers to the one-character string containing the single quote character). Literals never contain regular expressions.

Literals can contain Unicode escape sequences of the form \uXXXX, where XXXX is the hexadecimal Unicode character value. For example, '\u00E8' is the French letter *e* with a grave accent: 'è'. ANTLR also understands the usual special escape sequences: '\n' (newline), '\r' (carriage return), '\t' (tab), '\b' (backspace), and '\f' (form feed). You can use Unicode characters directly within literals or use the Unicode escape sequences. See code/reference/Foreign.g4.

```
grammar Foreign;
a : '外' ;
```

The recognizers that ANTLR generates assume a character vocabulary containing all Unicode characters. The input file encoding assumed by the runtime library depends on the target language. For the Java target, the runtime library assumes files are in UTF-8. Using the constructors, you can specify a different encoding. See, for example, ANTLR's ANTLRFileStream.

Actions

Actions are code blocks written in the target language. You can use actions in a number of places within a grammar, but the syntax is always the same: arbitrary text surrounded by curly braces. You don't need to escape a closing curly character if it's in a string or comment: {"}"} or {/*}*/;}. If the curlies are balanced, you also don't need to escape }: {{...}}. Otherwise, escape extra curlies with a backslash: {\{} or {\}}. The action text should conform to the target language as specified with the language option.

Embedded code can appear in @header and @members named actions, parser and lexer rules, exception catching specifications, attribute sections for parser rules (return values, arguments, and locals), and some rule element options (currently predicates).

The only interpretation ANTLR does inside actions relates to grammar attributes; see *Token Attributes*, on page 271, as well as Chapter 10, *Attributes and Actions*, on page 175. Actions embedded within lexer rules are emitted without any interpretation or translation into generated lexers.

Keywords

Here's a list of the reserved words in ANTLR grammars: import, fragment, lexer, parser, grammar, returns, locals, throws, catch, finally, mode, options, tokens. Also, although it is not a keyword, do not use the word rule as a rule or alternative label name since it results in RuleContext as a context object; RuleContext clashes with the built-in class. Further, do not use any keyword of the target language as a token, label, or rule name. For example, rule if would result in a generated function called if().

15.2 Grammar Structure

A grammar is essentially a grammar declaration followed by a list of rules but has the following general form:

```
/** Optional Javadoc-style comment */
grammar Name;                                    (1)
options {...}
import ... ;
tokens {...}
@actionName {...}

«rule1» // parser and lexer rules, possibly intermingled
...
«ruleN»
```

The filename containing grammar *X* must be called *X*.g4. You can specify options, imports, token specifications, and actions in any order. There can be at most one each of options, imports, and token specifications. All of those elements are optional except for the header (1) and at least one rule. Rules take the following basic form:

```
ruleName : «alternative1» | ... | «alternativeN» ;
```

Parser rule names must start with a lowercase letter, and lexer rules must start with a capital letter.

Grammars defined without a prefix on the grammar header are *combined* grammars that can contain both lexical and parser rules. To make a parser grammar that allows only parser rules, use the following header:

```
parser grammar Name;
...
```

And, naturally, a pure lexer grammar looks like this:

```
lexer grammar Name;
...
```

Only lexer grammars can contain mode specifications.

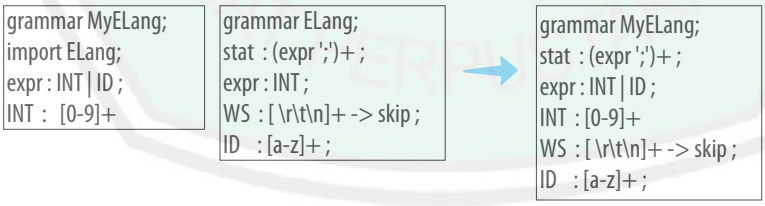
Section 15.5, *Lexer Rules*, on page 277 and Section 15.3, *Parser Rules*, on page 261 contain details on rule syntax. Section 15.8, *Options*, on page 292 describes grammar options, and Section 15.4, *Actions and Attributes*, on page 271 has information on grammar-level actions. We'll look at grammar imports, token specifications, and named actions next.

Grammar imports

Grammar imports let you break up a grammar into logical and reusable chunks, as we saw in *Importing Grammars*, on page 36. ANTLR treats imported grammars very much like object-oriented programming languages treat superclasses. A grammar inherits all of the rules, tokens specifications, and named actions from the imported grammar. Rules in the "main grammar" override rules from imported grammars to implement inheritance.

Think of import as more like a smart include statement (which does not include rules that are already defined). The result of all imports is a single combined grammar; the ANTLR code generator sees a complete grammar and has no idea there were imported grammars.

To process a main grammar, the ANTLR tool loads all of the imported grammars into subordinate grammar objects. It then merges the rules, token types, and named actions from the imported grammars into the main grammar. In the following diagram, the grammar on the right illustrates the effect of grammar MyELang importing grammar ELang:

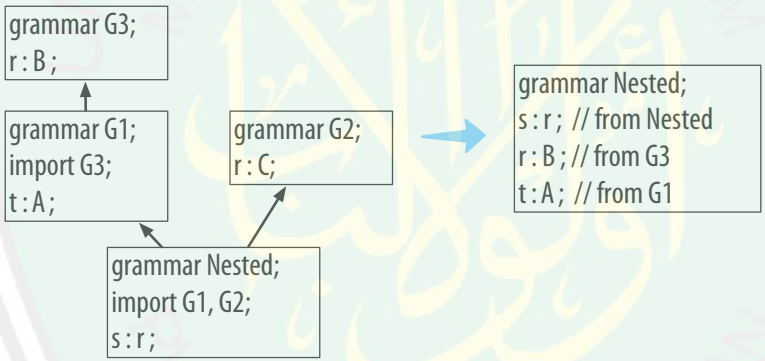


MyELang inherits rules stat, WS, and ID, but it overrides rule expr and adds INT. Here's a sample build and test run that shows MyELang can recognize integer expressions whereas the original ELang can't. The third, erroneous input statement triggers an error message that also demonstrates the parser was looking for MyELang's expr, not ELang's.

```
⇛ $ antlr4 MyELang.g4
⇛ $ javac MyELang*.java
⇛ $ grun MyELang stat
⇛ 34;
⇛ a;
⇛ ;
⇛ EOF
❮ line 3:0 extraneous input ';' expecting {<EOF>, INT, ID}
```

If there were any tokens specifications, the main grammar would merge the token sets. Any named actions such as @members would be merged. In general, you should avoid named actions and actions within rules in imported grammars since that limits their reuse. ANTLR also ignores any options in imported grammars.

Imported grammars can also import other grammars. ANTLR pursues all imported grammars in a depth-first fashion. If two or more imported grammars define rule r, ANTLR chooses the first version of r it finds. In the following diagram, ANTLR examines grammars in the following order: Nested, G1, G3, G2:



Nested includes the r rule from G3 because it sees that version before the r in G2.

Not every kind of grammar can import every other kind of grammar.

- Lexer grammars can import lexer grammars.
- Parser grammars can import parser grammars.
- Combined grammars can import lexer or parser grammars.

ANTLR adds imported rules to the end of the rule list in a main lexer grammar. That means lexer rules in the main grammar get precedence over imported rules. For example, if a main grammar defines rule IF : 'if' ; and an imported grammar defines rule ID : [a-z]+ ; (which also recognizes if), the imported ID won't hide the main grammar's IF token definition.

tokens Section

The purpose of the tokens section is to define token types needed by a grammar for which there is no associated lexical rule. The basic syntax is as follows:

```
tokens { «Token1», ..., «TokenN» }
```

Most of the time, the tokens section is used to define token types needed by actions in the grammar (as we did in Section 10.3, *Recognizing Languages Whose Keywords Aren't Fixed*, on page 185).

```
// explicitly define keyword token types to avoid implicit definition warnings
tokens { BEGIN, END, IF, THEN, WHILE }
@lexer::members {   // keywords map used in lexer to assign token types
Map<String,Integer> keywords = new HashMap<String,Integer>() {{
    put("begin", KeywordsParser.BEGIN);
    put("end",   KeywordsParser.END);
    ...
}};
}
```

The tokens section really just defines a set of tokens to add to the overall set.

```
$ cat Tok.g4
grammar Tok;
tokens { A, B, C }
a : X ;
$ antlr4 Tok.g4
warning(125): Tok.g4:3:4: implicit definition of token X in parser
$ cat Tok.tokens
A=1
B=2
C=3
X=4
```

Actions at the Grammar Level

Using Actions Outside of Grammar Rules, on page 176, illustrates the use of named actions at the top level of the grammar file. Currently there are only two defined actions (for the Java target): header and members. The former injects code into the generated recognizer class file, before the recognizer class definition, and the latter injects code into the recognizer class definition, as fields and methods.

For combined grammars, ANTLR injects the actions into both the parser and the lexer. To restrict an action to the generated parser or lexer, use @parser::*name* or @lexer::*name*.

Here's an example where the grammar specifies a package for the generated code:

reference/foo/Count.g4

```
grammar Count;

@header {
package foo;
}

@members {
int count = 0;
}

list
@after {System.out.println(count+" ints");}
    : INT {count++;} (',' INT {count++;} )*
    ;

INT : [0-9]+ ;
WS : [ \r\t\n]+ -> skip ;
```

The grammar itself should be in directory foo so that ANTLR generates code in that same foo directory (at least when not using the -o ANTLR tool option).

```
⇛ $ cd foo
⇛ $ antlr4 Count.g4   # generates code in the current directory (foo)
⇛ $ ls
❮ Count.g4               CountLexer.java        CountParser.java
  Count.tokens           CountLexer.tokens
  CountBaseListener.java CountListener.java
⇛ $ javac *.java
⇛ $ cd ..
⇛ $ grun foo.Count list
⇛ 9, 10, 11
⇛ EOF
❮ 3 ints
```

The Java compiler expects classes in package foo to be in directory foo.

Now that we've seen the overall structure of a grammar, let's dig into the parser and lexer rules.

15.3 Parser Rules

Parsers consist of a set of parser rules in either a parser or a combined grammar. A Java application launches a parser by invoking the rule function, generated by ANTLR, associated with the desired start rule. The most basic rule is just a rule name followed by a single alternative terminated with a semicolon.

```
/** Javadoc comment can precede rule */
retstat : 'return' expr ';' ;
```

Rules can also have alternatives separated by the | operator.

```
stat:   retstat
    |   'break' ';'
    |   'continue' ';'
    ;
```

Alternatives are either a list of rule elements or empty. For example, here's a rule with an empty alternative that makes the entire rule optional:

```
superClass
    :   'extends' ID
    |                   // empty means other alternative(s) are optional
    ;
```

Alternative Labels

As we saw in Section 7.4, *Labeling Rule Alternatives for Precise Event Methods*, on page 117, we can get more precise parse-tree listener events by labeling the outermost alternatives of a rule using the # operator. All alternatives within a rule must be labeled, or none of them should be. Here are two rules with labeled alternatives:

reference/AltLabels.g4

```
grammar AltLabels;
stat: 'return' e ';' # Return
    | 'break'    ';' # Break
    ;
e   : e '*' e        # Mult
    | e '+' e        # Add
    | INT            # Int
    ;
```

Alternative labels do not have to be at the end of the line, and there does not have to be a space after the # symbol.

ANTLR generates a rule context class definition for each label. For example, here is the listener that ANTLR generates:

```
public interface AltLabelsListener extends ParseTreeListener {
    void enterMult(AltLabelsParser.MultContext ctx);
    void exitMult(AltLabelsParser.MultContext ctx);
    void enterBreak(AltLabelsParser.BreakContext ctx);
    void exitBreak(AltLabelsParser.BreakContext ctx);
    void enterReturn(AltLabelsParser.ReturnContext ctx);
    void exitReturn(AltLabelsParser.ReturnContext ctx);
    void enterAdd(AltLabelsParser.AddContext ctx);
    void exitAdd(AltLabelsParser.AddContext ctx);
    void enterInt(AltLabelsParser.IntContext ctx);
    void exitInt(AltLabelsParser.IntContext ctx);
}
```

There are enter and exit methods associated with each labeled alternative. The parameters to those methods are specific to alternatives.

You can reuse the same label on multiple alternatives to indicate that the parse-tree walker should trigger the same event for those alternatives. For example, here's a variation on rule e that reuses label BinaryOp:

```
e   : e '*' e        # BinaryOp
    | e '+' e        # BinaryOp
    | INT            # Int
    ;
```

ANTLR would generate the following listener methods for e:

```
void enterBinaryOp(AltLabelsParser.BinaryOpContext ctx);
void exitBinaryOp(AltLabelsParser.BinaryOpContext ctx);
void enterInt(AltLabelsParser.IntContext ctx);
void exitInt(AltLabelsParser.IntContext ctx);
```

ANTLR gives errors if an alternative name conflicts with a rule name. Here's another rewrite of rule e where two alternative labels conflict with rule names:

reference/Conflict.g4

```
e   : e '*' e        # e
    | e '+' e        # Stat
    | INT            # Int
    ;
```

The context objects generated from rule names and labels get capitalized, so label Stat conflicts with rule stat.

```
$ antlr4 Conflict.g4
error(124): Conflict.g4:6:23: rule alt label e conflicts with rule e
error(124): Conflict.g4:7:23: rule alt label Stat conflicts with rule stat
warning(125): Conflict.g4:2:13: implicit definition of token INT in parser
```

Rule Context Objects

ANTLR generates methods to access the rule context objects (parse-tree nodes) associated with each rule reference. For rules with a single rule reference, ANTLR generates a method with no arguments. Consider the following rule:

```
inc : e '++' ;
```

ANTLR generates this context class:

```
public static class IncContext extends ParserRuleContext {
    public EContext e() { ... } // return context object associated with e
    ...
}
```

ANTLR also provides support to access context objects when there is more than a single reference to a rule.

```
field : e '.' e ;
```

ANTLR generates a method with an index to access the *i*th element as well as a method to get context for all references to that rule.

```
public static class FieldContext extends ParserRuleContext {
    public EContext e(int i) { ... }    // get ith e context
    public List<EContext> e() { ... }   // return ALL e contexts
    ...
}
```

If we had another rule, s, that references field, an embedded action could access the list of e rule matches performed by field.

```
s : field
    {
    List<EContext> x = $field.ctx.e();
     ...
    }
  ;
```

A listener or visitor could do the same thing. Given a pointer to a FieldContext object, f, f.e() would return List<EContext>.

Rule Element Labels

You can label rule elements using the = operator to add fields to the rule context objects.

```
stat: 'return' value=e ';' # Return
    | 'break'          ';' # Break
    ;
```

Here value is the label for the return value of rule e, which is defined elsewhere.

Labels become fields in the appropriate parse-tree node class. In this case, label value becomes a field in ReturnContext because of the Return alternative label.

```
public static class ReturnContext extends StatContext {
    public EContext value;
    ...
}
```

It's often handy to track a number of tokens, which you can do with the += "list label" operator. For example, the following rule creates a list of the Token objects matched for a simple array construct:

```
array : '{' el+=INT (',' el+=INT)* '}' ;
```

ANTLR generates a List field in the appropriate rule context class.

```
public static class ArrayContext extends ParserRuleContext {
    public List<Token> el = new ArrayList<Token>();
    ...
}
```

These list labels also work for rule references.

```
elist : exprs+=e (',' exprs+=e)* ;
```

ANTLR generates a field holding the list of context objects.

```
public static class ElistContext extends ParserRuleContext {
    public List<EContext> exprs = new ArrayList<EContext>();
    ...
}
```

Rule Elements

Rule elements specify what the parser should do at a given moment just like statements in a programming language. The elements can be a rule, a token, or a string literal like expression, ID, and 'return'. Here's a complete list of the rule elements (we'll look at actions and predicates in more detail later):

Syntax	Description
T	Match token *T* at the current input position. Tokens always begin with a capital letter.
'*literal*'	Match the string literal at the current input position. A string literal is simply a token with a fixed string.
r	Match rule *r* at the current input position, which amounts to invoking the rule just like a function call. Parser rule names always begin with a lowercase letter.

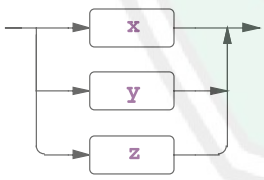
Syntax	Description
r[«*args*»]	Match rule *r* at the current input position, passing in a list of arguments just like a function call. The arguments inside the square brackets are in the syntax of the target language and are usually a comma-separated list of expressions.
{«*action*»}	Execute an action immediately after the preceding alternative element and immediately before the following alternative element. The action conforms to the syntax of the target language. ANTLR copies the action code to the generated class verbatim, except for substituting attribute and token references such as `$x` and `$x.y`.
{«*p*»}?	Evaluate semantic predicate «*p*». Do not continue parsing past a predicate if «*p*» evaluates to false at runtime. Predicates encountered during prediction, when ANTLR distinguishes between alternatives, enable or disable the alternative(s) surrounding the predicate(s).
.	Match any single token except for the end-of-file token. The "dot" operator is called the *wildcard*.

When you want to match everything but a particular token or set of tokens, use the ~ "not" operator. This operator is rarely used in the parser but is available. `~INT` matches any token except the `INT` token. `~','` matches any token except the comma. `~(INT|ID)` matches any token except an `INT` or an `ID`.

Token, string literal, and semantic predicate rule elements can take options. See *Rule Element Options*, on page 293.

Subrules

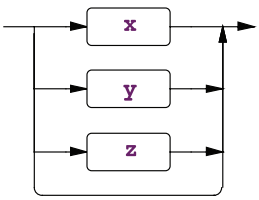
A rule can contain alternative blocks called *subrules* (as allowed in Extended BNF Notation [EBNF]). A subrule is like a rule that lacks a name and is enclosed in parentheses. Subrules can have one or more alternatives inside the parentheses. Subrules cannot define attributes with `locals` and `returns` like rules can. There are four kinds of subrules (x, y, and z represent grammar fragments).



`(x|y|z)`

Match any alternative within the subrule exactly once. Here's an example:

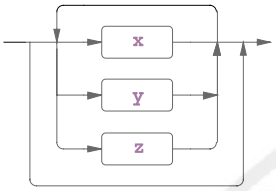
```
returnType : (type | 'void') ;
```



(x|y|z)?

Match nothing or any alternative within the subrule. Here's an example:

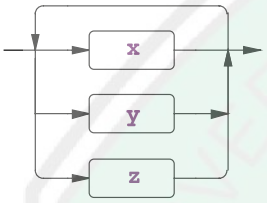
```
classDeclaration
    :   'class' ID (typeParameters)? ('extends' type)?
        ('implements' typeList)?
        classBody
    ;
```



(x|y|z)*

Match an alternative within the subrule zero or more times. Here's an example:

```
annotationName : ID ('.' ID)* ;
```



(x|y|z)+

Match an alternative within the subrule one or more times. Here's an example:

```
annotations : (annotation)+ ;
```

You can suffix the ?, *, and + subrule operators with the nongreedy operator, which is also a question mark: ??, *?, and +?. See Section 15.6, *Wildcard Operator and Nongreedy Subrules*, on page 283.

As a shorthand, you can omit the parentheses for subrules composed of a single alternative with a single-rule element reference. For example, annotation+ is the same as (annotation)+, and ID+ is the same as (ID)+. Labels also work with the shorthand. ids+=INT+ make a list of INT token objects.

Catching Exceptions

When a syntax error occurs within a rule, ANTLR catches the exception, reports the error, attempts to recover (possibly by consuming more tokens), and then returns from the rule. Every rule is wrapped in a try/catch/finally statement.

```
void r() throws RecognitionException {
    try {
        «rule-body»
    }
    catch (RecognitionException re) {
        _errHandler.reportError(this, re);
```

```
        _errHandler.recover(this, re);
    }
    finally {
        exitRule();
    }
}
```

In Section 9.5, *Altering ANTLR's Error Handling Strategy*, on page 171, we saw how to use a strategy object to alter ANTLR's error handling. Replacing the strategy changes the strategy for all rules, however. To alter the exception handling for a single rule, specify an exception after the rule definition.

```
r   :   ...
    ;
    catch[RecognitionException e] { throw e; }
```

That example demonstrates how to avoid default error reporting and recovery. `r` rethrows the exception, which is useful when it makes more sense for a higher-level rule to report the error. Specifying any exception clause prevents ANTLR from generating a clause to handle `RecognitionException`.

You can specify other exceptions as well.

```
r   :   ...
    ;
    catch[FailedPredicateException fpe] { ... }
    catch[RecognitionException e] { ... }
```

The code snippets inside curly braces and the exception "argument" actions must be written in the target language, Java, in this case.

When you need to execute an action even if an exception occurs, put it into the `finally` clause.

```
r   :   ...
    ;
    // catch blocks go first
    finally { System.out.println("exit rule r"); }
```

The `finally` clause executes right before the rule triggers `exitRule()` before returning. If you want to execute an action after the rule finishes matching the alternatives but before it does its cleanup work, use an `after` action.

Here's a complete list of exceptions:

Exception name	Description
`RecognitionException`	The superclass of all exceptions thrown by an ANTLR-generated recognizer. It's a subclass of `RuntimeException` to avoid the hassles of checked exceptions. This

Exception name	Description
	exception records where the recognizer (lexer or parser) was in the input, where it was in the ATN (internal graph data structure representing the grammar), the rule invocation stack, and what kind of problem occurred.
NoViableAltException	Indicates that the parser could not decide which of two or more paths to take by looking at the remaining input. This exception tracks the starting token of the offending input and also knows where the parser was in the various paths when the error occurred.
LexerNoViableAltException	The equivalent of NoViableAltException but for lexers only.
InputMismatchException	The current input Token does not match what the parser expected.
FailedPredicateException	A semantic predicate that evaluates to false during prediction renders the surrounding alternative nonviable. Prediction occurs when a rule is predicting which alternative to take. If all viable paths disappear, the parser will throw NoViableAltException. This exception gets thrown by the parser when a semantic predicate evaluates to false outside of prediction, during the normal parsing process of matching tokens and calling rules.

Rule Attribute Definitions

There are a number of action-related syntax elements associated with rules to be aware of. Rules can have arguments, return values, and local variables just like functions in a programming language. (Rules can have actions embedded among the rule elements, as we'll see in Section 15.4, *Actions and Attributes*, on page 271.) ANTLR collects all of the variables you define and stores them in the rule context object. These variables are usually called *attributes*. Here's the general syntax showing all possible attribute definition locations:

```
rulename[«args»] returns [«retvals»] locals [«localvars»] : ... ;
```

The attributes defined within those [...] can be used like any other variable. Here is a sample rule that copies parameters to return values:

```
// Return the argument plus the integer value of the INT token
add[int x] returns [int result] : '+=' INT {$result = $x + $INT.int;} ;
```

As with the grammar level, you can specify rule-level named actions. For rules, the valid names are init and after. As the names imply, parsers execute init actions immediately before trying to match the associated rule and execute after actions immediately after matching the rule. ANTLR after actions do not execute as part of the finally code block of the generated rule function. Use the ANTLR finally action to place code in the generated rule function finally code block.

The actions come after any argument, return value, or local attribute definition actions. The row rule preamble from Section 10.2, *Accessing Token and Rule Attributes*, on page 182 illustrates the syntax nicely.

actions/CSV.g4

```
/** Derived from rule "row : field (',' field)* '\r'? '\n' ;" */
row[String[] columns] returns [Map<String,String> values]
locals [int col=0]
@init {
    $values = new HashMap<String,String>();
}
@after {
    if ($values!=null && $values.size()>0) {
        System.out.println("values = "+$values);
    }
}
```

Rule row takes argument columns, returns values, and defines local variable col. The "actions" in square brackets are copied directly into the generated code.

```
public class CSVParser extends Parser {
    ...
    public static class RowContext extends ParserRuleContext {
        public String[] columns;
        public Map<String,String> values;
        public int col=0;
        ...
    }
    ...
}
```

The generated rule functions also specify the rule arguments as function arguments, but they are quickly copied into the local RowContext object.

```
public class CSVParser extends Parser {
    ...
    public final RowContext row(String[] columns) throws RecognitionException {
        RowContext _localctx = new RowContext(_ctx, 4, columns);
        enterRule(_localctx, RULE_row);
        ...
    }
    ...
}
```

ANTLR tracks nested [...] within the action so that String[] columns is parsed properly. It also tracks angle brackets so that commas within generic type parameters do not signify the start of another attribute. Map<String,String> values is one attribute definition.

There can be multiple attributes in each action, even for return values. Use a comma to separate attributes within the same action.

```
a[Map<String,String> x, int y] : ... ;
```

ANTLR interprets that action to define two arguments, x and y.

```
public final AContext a(Map<String,String> x, int y)
    throws RecognitionException
{
    AContext _localctx = new AContext(_ctx, 0, x, y);
    enterRule(_localctx, RULE_a);
    ...
}
```

Start Rules and EOF

A *start rule* is the rule engaged first by the parser; it's the rule function called by the language application. For example, a language application that parses Java code might call parser.compilationUnit() on a JavaParser object called parser. Any rule in the grammar can act as a start rule.

Start rules don't necessarily consume all of the input. They consume only as much input as needed to match an alternative of the rule. For example, consider the following rule that matches one, two, or three tokens, depending on the input:

```
s : ID
  | ID '+'
  | ID '+' INT
  ;
```

Upon a+3, rule s matches the third alternative. Upon a+b, it matches the second alternative and ignores the final b token. Upon a b, it matches the first alternative, ignoring the b token. The parser does not consume the complete input in the latter two cases because rule s doesn't explicitly say that the end of file must occur after matching an alternative of the rule.

This default functionality is very useful for building things such as IDEs. Imagine the IDE wanting to parse a method somewhere in the middle of a big Java file. Calling rule methodDeclaration should try to match just a method and ignore whatever comes next.

On the other hand, rules that describe entire input files should reference special predefined-token EOF. If they don't, you might scratch your head for a while wondering why the start rule doesn't report errors for any input no matter what you give it. Here's a rule that's part of a grammar for reading configuration files:

```
config : element*; // can "match" even with invalid input.
```

Invalid input would cause config to return immediately without matching any input and without reporting an error. Here's the proper specification:

```
file : element* EOF; // don't stop early. must match all input
```

15.4 Actions and Attributes

In Chapter 10, *Attributes and Actions*, on page 175, we learned how to embed actions within grammars and looked at the most common token and rule attributes. This section summarizes the important syntax and semantics from that chapter and provides a complete list of all available attributes.

Actions are blocks of text written in the target language and enclosed in curly braces. The recognizer triggers them according to their locations within the grammar. For example, the following rule emits found a decl after the parser has seen a valid declaration:

```
decl: type ID ';' {System.out.println("found a decl");} ;
type: 'int' | 'float' ;
```

Most often, actions access the attributes of tokens and rule references.

```
decl: type ID ';'
      {System.out.println("var "+$ID.text+":"+$type.text+";");}
    | t=ID id=ID ';'
      {System.out.println("var "+$id.text+":"+$t.text+";");}
    ;
```

Token Attributes

All tokens have a collection of predefined, read-only attributes. The attributes include useful token properties such as the token type and text matched for a token. Actions can access these attributes via $*label*.*attribute* where *label* labels a particular instance of a token reference (a and b in the following example are used in the action code as $a and $b). Often, a particular token is referenced only once in the rule, in which case the token name itself can be used unambiguously in the action code (token INT can be used as $INT in the action). The following example illustrates token attribute expression syntax:

```
r   :   INT {int x = $INT.line;}
        ( ID {if ($INT.line == $ID.line) ...;} )?
        a=FLOAT b=FLOAT {if ($a.line == $b.line) ...;}
    ;
```

The action within the (...)? subrule can see the INT token matched before it in the outer level.

Because there are two references to the FLOAT token, a reference to $FLOAT in an action is not unique; you must use labels to specify which token reference you're interested in.

Token references within different alternatives are unique because only one of them can be matched for any invocation of the rule. For example, in the following rule, actions in both alternatives can reference $ID directly without using a label.

```
r   :   ... ID {System.out.println($ID.text);}
    |   ... ID {System.out.println($ID.text);}
    ;
```

To access the tokens matched for literals, you must use a label.

```
stat:   r='return' expr ';' {System.out.println("line="+$r.line);} ;
```

Most of the time you access the attributes of the token, but sometimes it is useful to access the Token object itself because it aggregates all the attributes. Further, you can use it to test whether an optional subrule matched a token.

```
stat: 'if' expr 'then' stat (el='else' stat)?
      {if ( $el!=null ) System.out.println("found an else");}
    | ...
    ;
```

$*T* and $*l* evaluate to Token objects for token name *T* and token label *l*. $*ll* evaluates to List<Token> for list label *ll*. $*T.attr* evaluates to the type and value specified in the following table for attribute *attr*:

Attribute	Type	Description
text	String	The text matched for the token; translates to a call to getText(). Example: $ID.text.
type	int	The token type (nonzero positive integer) of the token such as INT; translates to a call to getType(). Example: $ID.type.
line	int	The line number on which the token occurs, counting from 1; translates to a call to getLine(). Example: $ID.line.

Attribute	Type	Description
pos	int	The character position within the line at which the token's first character occurs counting from zero; translates to a call to getCharPositionInLine(). Example: $ID.pos.
index	int	The overall index of this token in the token stream, counting from zero; translates to a call to getTokenIndex(). Example: $ID.index.
channel	int	The token's channel number. The parser tunes to only one channel, effectively ignoring off-channel tokens. The default channel is 0 (Token.DEFAULT_CHANNEL), and the default hidden channel is Token.HIDDEN_CHANNEL. Translates to a call to getChannel(). Example: $ID.channel.
int	int	The integer value of the text held by this token; it assumes that the text is a valid numeric string. Handy for building calculators and so on. Translates to Integer.valueOf(text-of-token). Example: $INT.int.

Parser Rule Attributes

ANTLR predefines a number of read-only attributes associated with parser rule references that are available to actions. Actions can access rule attributes only for references that precede the action. The syntax is $*r*.*attr* for rule name *r* or a label assigned to a rule reference. For example, $expr.text returns the complete text matched by a preceding invocation of rule expr.

```
returnStat : 'return' expr {System.out.println("matched "+$expr.text);} ;
```

Using a rule label looks like this:

```
returnStat : 'return' e=expr {System.out.println("matched "+$e.text);} ;
```

You can also use $ followed by the name of the attribute to access the value associated with the currently executing rule. For example, $start is the starting token of the current rule.

```
returnStat : 'return' expr {System.out.println("first token "+$start.getText());} ;
```

$*r* and $*rl* evaluate to ParserRuleContext objects of type *R*Context for rule name *r* and rule label *rl*. $*rll* evaluates to List<*R*Context> for rule list label *rll*. $*r*.*attr* evaluates to the type and value specified in the following table for attribute *attr*:

Attribute	Type	Description
text	String	The text matched for a rule or the text matched from the start of the rule up until the point of the $text expression evaluation. Note that this includes the text for all tokens including those on hidden channels, which is what you want because usually that has all the whitespace and comments. When referring to the current rule, this attribute is available in any action including any exception actions.
start	Token	The first token to be potentially matched by the rule that is on the main token channel; in other words, this attribute is never a hidden token. For rules that end up matching no tokens, this attribute points at the first token that could have been matched by this rule. When referring to the current rule, this attribute is available to any action within the rule.
stop	Token	The last nonhidden channel token to be matched by the rule. When referring to the current rule, this attribute is available only to the after and finally actions.
ctx	ParserRuleContext	The rule context object associated with a rule invocation. All of the other attributes are available through this attribute. For example, $ctx.start accesses the start field within the current rules context object. It's the same as $start.

Dynamically Scoped Attributes

You can pass information to and from rules using parameters and return values, just like functions in a general-purpose programming language. Programming languages don't allow functions to access the local variables or parameters of invoking functions, however.

For example, the following reference to local variable x from a nested method call is illegal in Java:

```
void f() {
    int x = 0;
    g();
}
```

```
void g() {
    h();
}
void h() {
    int y = x; // INVALID reference to f's local variable x
}
```

Variable x is available only within the scope of f(), which is the text lexically delimited by curly brackets. For this reason, Java is said to use *lexical scoping*. Lexical scoping is the norm for most programming languages.[2] Languages that allow methods further down in the call chain to access local variables defined earlier are said to use *dynamic scoping*. The term *dynamic* refers to the fact that a compiler cannot statically determine the set of visible variables. This is because the set of variables visible to a method changes depending on who calls that method.

It turns out that, in the grammar realm, distant rules sometimes need to communicate with each other, mostly to provide context information to rules matched below in the rule invocation chain. (Naturally, this assumes you are using actions directly in the grammar instead of the parse-tree listener event mechanism.) ANTLR allows dynamic scoping in that actions can access attributes from invoking rules using syntax $r::x where r is a rule name and x is an attribute within that rule. It is up to the programmer to ensure that r is in fact an invoking rule of the current rule. A runtime exception occurs if r is not in the current call chain when you access $r::x.

To illustrate the use of dynamic scoping, consider the real problem of defining variables and ensuring that variables in expressions are defined. The following grammar defines the symbols attribute where it belongs in the block rule but adds variable names to it in rule decl. Rule stat then consults the list to see whether variables have been defined.

reference/DynScope.g4

```
grammar DynScope;

prog:   block
    ;

block
/* List of symbols defined within this block */
locals [
    List<String> symbols = new ArrayList<String>()
]
```

2. See http://en.wikipedia.org/wiki/Scope_(programming)#Static_scoping.

```
    :   '{' decl* stat+ '}'
        // print out all symbols found in block
        // $block::symbols evaluates to a List as defined in scope
        {System.out.println("symbols="+$symbols);}
    ;

/** Match a declaration and add identifier name to list of symbols */
decl:   'int' ID {$block::symbols.add($ID.text);} ';'
    ;

/** Match an assignment then test list of symbols to verify
 *  that it contains the variable on the left side of the assignment.
 *  Method contains() is List.contains() because $block::symbols
 *  is a List.
 */
stat:   ID '=' INT ';'
        {
        if ( !$block::symbols.contains($ID.text) ) {
            System.err.println("undefined variable: "+$ID.text);
        }
        }
    |   block
    ;

ID  :   [a-z]+ ;
INT :   [0-9]+ ;
WS  :   [ \t\r\n]+ -> skip ;
```

Here's a simple build and test sequence:

```
⇛ $ antlr4 DynScope.g4
⇛ $ javac DynScope*.java
⇛ $ grun DynScope prog
⇛ {
⇛   int i;
⇛   i = 0;
⇛   j = 3;
⇛ }
⇛ EOF
❮ undefined variable: j
  symbols=[i]
```

There's an important difference between a simple field declaration in an @members action and dynamic scoping. symbols is a local variable, so there is a copy for each invocation of rule block. That's exactly what we want for nested blocks so that we can reuse the same input variable name in an inner block. For example, the following nested code block redefines i in the inner scope. This new definition must hide the definition in the outer scope.

reference/nested-input

```
{
  int i;
  int j;
  i = 0;
  {
    int i;
    int x;
    x = 5;
  }
  x = 3;
}
```

Here's the output generated for that input by DynScope:

```
$ grun DynScope prog nested-input
symbols=[i, x]
undefined variable: x
symbols=[i, j]
```

Referencing $block::symbols accesses the symbols field of the most recently invoked block's rule context object. If you need access to a symbols instance from a rule invocation further up the call chain, you can walk backward starting at the current context, $ctx. Use getParent() to walk up the chain.

15.5 Lexer Rules

A lexer grammar is composed of lexer rules, optionally broken into multiple modes, as we saw in *Issuing Context-Sensitive Tokens with Lexical Modes*, on page 221. Lexical modes allow us to split a single lexer grammar into multiple sublexers. The lexer can return only those tokens matched by rules from the current mode.

Lexer rules specify token definitions and more or less follow the syntax of parser rules except that lexer rules cannot have arguments, return values, or local variables. Lexer rule names must begin with an uppercase letter, which distinguishes them from parser rule names.

```
/** Optional document comment */
TokenName : «alternative1» | ... | «alternativeN» ;
```

You can also define rules that are not tokens but rather aid in the recognition of tokens. These fragment rules do not result in tokens visible to the parser.

```
fragment HelperTokenRule : «alternative1» | ... | «alternativeN» ;
```

For example, DIGIT is a pretty common fragment rule.

```
INT : DIGIT+ ;          // references the DIGIT helper rule
fragment DIGIT : [0-9] ; // not a token by itself
```

Lexical Modes

Modes allow you to group lexical rules by context, such as inside and outside of XML tags. It's like having multiple sublexers, with one for context. The lexer can return only those tokens matched by entering a rule in the current mode. Lexers start out in the so-called default mode. All rules are considered to be within the default mode unless you specify a `mode` command. Modes are not allowed within combined grammars, just `lexer` grammars. (See grammar `XMLLexer` from *Tokenizing XML*, on page 226.)

```
«rules in default mode»
...
mode MODE1;
«rules in MODE1»
...
mode MODEN;
«rules in MODEN»
...
```

Lexer Rule Elements

Lexer rules allow two constructs that are unavailable to parser rules: the `..` range operator and the character set notation enclosed in square brackets, [*characters*]. Don't confuse character sets with arguments to parser rules. [*characters*] only means character set in a lexer. Here's a summary of all lexer rule elements:

Syntax	Description
'*literal*'	Match that character or sequence of characters. Here's an example: `'while'` or `'='`.

[*char set*]	Match one of the characters specified in the character set. Interpret *x*-*y* as a set of characters between range *x* and *y*, inclusively. The following escaped characters are interpreted as single special characters: `\n`, `\r`, `\b`, `\t`, and `\f`. To get `]`, `\`, or `-`, you must escape them with `\`. You can also use Unicode character specifications: `\uXXXX`. Here are a few examples:
	`WS : [ \n\u000D] -> skip ;  // same as [ \n\r]`
	`ID : [a-zA-Z] [a-zA-Z0-9]* ; // match usual identifier spec`
	`DASHBRACK : [\-\]]+ ;  // match - or ] one or more times`

'*x*'..'*y*'	Match any single character between range *x* and *y*, inclusively. Here's an example: `'a'..'z'`. `'a'..'z'` is identical to `[a-z]`.
T	Invoke lexer rule *T*; recursion is allowed in general but not left recursion. *T* can be a regular token or `fragment` rule.

Syntax	Description

```
ID          :   LETTER (LETTER|'0'..'9')* ;
fragment
LETTER      :   [a-zA-Z\u0080-\u00FF_] ;
```

`.` The dot is a single-character wildcard that matches any single character. Here's an example:

```
ESC : '\\' . ; // match any escaped \x character
```

{«*action*»} Lexer actions must appear at the end of the outermost alternative. If a lexer rule has more than one alternative, enclose them in parentheses and put the action afterward.

```
END : ('endif'|'end') {System.out.println("found an end");} ;
```

The action conforms to the syntax of the target language. ANTLR copies the action's contents into the generated code verbatim; there is no translation of expressions such as `$x.y` like there is in parser actions.

{«*p*»}? Evaluate semantic predicate «*p*». If «*p*» evaluates to false at runtime, the surrounding rule becomes "invisible" (nonviable). Expression «*p*» conforms to the target language syntax. While semantic predicates can appear anywhere within a lexer rule, it is most efficient to have them at the end of the rule. The one caveat is that semantic predicates must precede lexer actions. See *Predicates in Lexer Rules*, on page 290.

~*x* Match any single character not in the set described by *x*. Set *x* can be a single character literal, a range, or a subrule set like ~('x'|'y'|'z') or ~[xyz]. Here is a rule that uses ~ to match any character other than characters using ~[\r\n]*:

```
COMMENT :   '#' ~[\r\n]* '\r'? '\n' -> skip ;
```

Just as with parser rules, lexer rules allow subrules in parentheses and EBNF operators: ?, *, +. The `COMMENT` rule illustrates the * and ? operators. A common use of + is `[0-9]+` to match integers. Lexer subrules can also use the nongreedy ? suffix on those EBNF operators.

Recursive Lexer Rules

ANTLR lexer rules can be recursive, unlike most lexical grammar tools. This comes in handy when you want to match nested tokens like nested action blocks: {...{...}...}.

reference/Recur.g4
```
lexer grammar Recur;

ACTION : '{' ( ACTION | ~[{}] )* '}' ;

WS     : [ \r\t\n]+ -> skip ;
```

Redundant String Literals

Be careful that you don't specify the same string literal on the right side of multiple lexer rules. Such literals are ambiguous and could match multiple token types. ANTLR makes this literal unavailable to the parser. The same is true for rules across modes. For example, the following lexer grammar defines two tokens with the same character sequence:

reference/L.g4
```
lexer grammar L;
AND : '&' ;
mode STR;
MASK : '&' ;
```

A parser grammar cannot reference literal '&', but it can reference the name of the tokens.

reference/P.g4
```
parser grammar P;
options { tokenVocab=L; }
a : '&' // results in a tool error: no such token
    AND // no problem
    MASK // no problem
  ;
```

Here's a build and test sequence:

```
⇛ $ antlr4 L.g4   # yields L.tokens file needed by tokenVocab option in P.g4
⇛ $ antlr4 P.g4
❮ error(126): P.g4:3:4: cannot create implicit token for string literal '&'
                        in non-combined grammar
```

Lexer Rule Actions

An ANTLR lexer creates a Token object after matching a lexical rule. Each request for a token starts in Lexer.nextToken(), which calls emit() once it has identified a token. emit() collects information from the current state of the lexer to build the token. It accesses fields _type, _text, _channel, _tokenStartCharIndex, _tokenStartLine, and _tokenStartCharPositionInLine. You can set the state of these with the various setter methods such as setType(). For example, the following rule turns enum into an identifier if enumIsKeyword is false:

```
ENUM : 'enum' {if (!enumIsKeyword) setType(Identifier);} ;
```

ANTLR does no special $x attribute translations in lexer actions (unlike v3).

There can be at most a single action for a lexical rule, regardless of how many alternatives there are in that rule.

Lexer Commands

To avoid tying a grammar to a particular target language, ANTLR supports *lexer commands*. Unlike arbitrary embedded actions, these commands follow specific syntax and are limited to a few common commands. Lexer commands appear at the end of the outermost alternative of a lexer rule definition. Like arbitrary actions, there can be only one per token rule. A lexer command consists of the -> operator followed by one or more command names that can optionally take parameters.

TokenName : *«alternative» -> command-name*

TokenName : *«alternative» -> command-name*(*«identifier or integer»*)

An alternative can have more than one command separated by commas. Here are the valid command names:

skip Do not return a token to the parser for this rule. This is typically used for whitespace:

```
WS : [ \r\t\n]+ -> skip ;
```

more Match this rule but continue looking for a token. The token rule that matches next will include the text matched for this rule. This is typically used with modes. Here's an example that matches string literals with a mode:

reference/Strings.g4

```
lexer grammar Strings;
LQUOTE : '"' -> more, mode(STR) ;
WS     : [ \r\t\n]+ -> skip ;

mode STR;

STRING : '"' -> mode(DEFAULT_MODE) ; // token we want parser to see
TEXT   : .   -> more ;               // collect more text for string
```

Here's a sample run:

```
⇒ $ antlr4 Strings.g4
⇒ $ javac Strings.java
⇒ $ grun Strings tokens -tokens
⇒ "hi"
⇒ "mom"
⇒ EOF
```

```
❮ [@0,0:3='"hi"',<2>,1:0]
  [@1,5:9='"mom"',<2>,2:0]
  [@2,11:10='<EOF>',<-1>,3:0]
```

type(*T*) Set the token type for the current token. Here's an example that forces two different tokens to use the same token type:

reference/SetType.g4

```
lexer grammar SetType;

tokens { STRING }

DOUBLE : '"' .*? '"'   -> type(STRING) ;
SINGLE : '\'' .*? '\'' -> type(STRING) ;
WS     : [ \r\t\n]+ -> skip ;
```

Here's a sample run. You can see that both tokens come back as token type 1.

```
⇒ $ antlr4 SetType.g4
⇒ $ javac SetType.java
⇒ $ grun SetType tokens -tokens
⇒ "double"
⇒ 'single'
⇒ EOF
❮ [@0,0:7='"double"',<1>,1:0]
  [@1,9:16='''single''',<1>,2:0]
  [@2,18:17='<EOF>',<-1>,3:0]
```

channel(*C*) Set the channel for the current token. The default is Token.DEFAULT_CHANNEL. You can define constants and then use it or an integer literal above Token.DEFAULT_CHANNEL in value (0). There's a generic hidden channel called Token.HIDDEN_CHANNEL with value 1.

```
@lexer::members { public static final int WHITESPACE = 1; }
...
WS  :   [ \t\n\r]+ -> channel(WHITESPACE) ;
```

mode(*M*) After matching this token, switch the lexer to mode *M*. The next time the lexer tries to match a token, it will look only at rules in mode *M*. *M* can be a mode name from the same grammar or an integer literal. See grammar Strings earlier.

pushMode(*M*) This is the same as mode except that it pushes the current mode onto a stack as well as setting the mode *M*. It should be used in conjunction with popMode.

popMode Pop a mode from the top of the mode stack and set the current mode of the lexer to that. This is used in conjunction with pushMode.

15.6 Wildcard Operator and Nongreedy Subrules

EBNF subrules like (...)?, (...)*, and (...)+ are *greedy*—they consume as much input as possible, but sometimes that's not what's needed. Constructs like .* consume until the end of the input in the lexer and sometimes in the parser. We want that loop to be *nongreedy*, so we need to use different syntax: .*? borrowed from regular expression notation. We can make any subrule that has a ?, *, or + suffix nongreedy by adding another ? suffix. Such nongreedy subrules are allowed in both the parser and the lexer, but they are used much more frequently in the lexer.

Nongreedy Lexer Subrules

Here's the very common C-style comment lexer rule that consumes any characters until it sees the trailing '*/':

```
COMMENT : '/*' .*? '*/' -> skip ; // .*? matches anything until the first */
```

Here's another example that matches strings that allow \" as an escaped quote character:

reference/Nongreedy.g4

```
grammar Nongreedy;
s : STRING+ ;
STRING : '"' ( '\\"' | . )*? '"' ; // match "foo", "\"", "x\"\"y", ...
WS     : [ \r\t\n]+ -> skip ;
```

```
⇛ $ antlr4 Nongreedy.g4
⇛ $ javac Nongreedy*.java
⇛ $ grun Nongreedy s -tokens
⇛ "quote:\""
⇛ EOF
❮ [@0,0:9='"quote:\""',<1>,1:0]
  [@1,11:10='<EOF>',<-1>,2:0]
```

Nongreedy subrules should be used sparingly because they complicate the recognition problem and sometimes make it tricky to decipher how the lexer will match text. Here is how the lexer chooses token rules:

- The primary goal is to match the lexer rule that recognizes the most input characters.

  ```
  INT   : [0-9]+ ;
  DOT   : '.' ;         // match period
  FLOAT : [0-9]+ '.' ;  // match FLOAT upon '34.' not INT then DOT
  ```

- If more than one lexer rule matches the same input sequence, the priority goes to the rule occurring first in the grammar file.

```
DOC : '/**' .*? '*/' ; // both rules match /** foo */,  resolve to DOC
CMT : '/*' .*? '*/' ;
```

- Nongreedy subrules match the fewest characters that still allow the surrounding lexical rule to match.

```
/** Match anything except \n inside of double angle brackets */
STRING : '<<' ~'\n'*? '>>' ; // Input '<<foo>>>>' matches STRING then END
END    : '>>' ;
```

- After crossing through a nongreedy subrule *within* a lexical rule, all decision making from then on is "first match wins."

  For example, alternative 'ab' in the rule right-side .*? ('a'|'ab') is dead code and can never be matched. If the input is ab, the first alternative, 'a', matches the first character and therefore succeeds. ('a'|'ab') by itself on the right side of a rule properly matches the second alternative for input ab. This quirk arises from a nongreedy design decision that's too complicated to go into here.

To illustrate the different ways to use loops within lexer rules, consider the following grammar, which has three different action-like tokens (using different delimiters so that they all fit within one example grammar):

reference/Actions.g4

```
ACTION1 : '{' ( STRING | . )*? '}' ;    // Allows {"foo}
ACTION2 : '[' ( STRING | ~'"' )*? ']' ; // Doesn't allow ["foo]; nongreedy *?
ACTION3 : '<' ( STRING | ~[">] )* '>' ; // Doesn't allow <"foo>; greedy *
STRING : '"' ( '\\"' | . )*? '"' ;
```

Rule ACTION1 allows unterminated strings, such as {"foo}, because input "foo matches to the wildcard part of the loop. It doesn't have to go into rule STRING to match a quote. To fix that, rule ACTION2 uses ~'"' to match any character but the quote. Expression ~'"' is still ambiguous with the ']' that ends the rule, but the fact that the subrule is nongreedy means that the lexer will exit the loop upon a right square bracket. To avoid a nongreedy subrule, make the alternatives explicit. Expression ~[">] matches anything but the quote and right angle bracket. Here's a sample run:

```
⇛ $ antlr4 Actions.g4
⇛ $ javac Actions*.java
⇛ $ grun Actions tokens -tokens
⇛ {"foo}
⇛ EOF
❮ [@0,0:5='{"foo}',<1>,1:0]
  [@1,7:6='<EOF>',<-1>,2:0]
⇛ $ grun Actions tokens -tokens
⇛ ["foo]
```

```
⇒ EOF
❮ line 1:0 token recognition error at: '["foo]\n'
  [@0,7:6='<EOF>',<-1>,2:0]
⇒ $ grun Actions tokens -tokens
⇒ <"foo>
⇒ EOF
❮ line 1:0 token recognition error at: '<"foo>\n'
  [@0,7:6='<EOF>',<-1>,2:0]
```

Nongreedy Parser Subrules

Nongreedy subrules and wildcards are also useful within parsers to do "fuzzy parsing" where the goal is to extract information from an input file without having to specify the full grammar. In contrast to nongreedy lexer decision making, parsers always make globally correct decisions. A parser never makes a decision that will ultimately cause valid input to fail later during the parse. Here is the central idea: *nongreedy parser subrules match the shortest sequence of tokens that preserves a successful parse for a valid input sentence.*

For example, here are the key rules that demonstrate how to pull integer constants out of an arbitrary Java file:

reference/FuzzyJava.g4

```
grammar FuzzyJava;
/** Match anything in between constant rule matches */
file : .*? (constant .*?)+ ;

/** Faster alternate version (Gets an ANTLR tool warning about
 *  a subrule like .* in parser that you can ignore.)
 */
altfile : (constant | .)* ; // match a constant or any token, 0-or-more times

/** Match things like "public static final SIZE" followed by anything */
constant
     : 'public' 'static' 'final' 'int' Identifier
       {System.out.println("constant: "+$Identifier.text);}
     ;
Identifier : [a-zA-Z_$] [a-zA-Z_$0-9]* ; // simplified
```

The grammar contains a greatly simplified set of lexer rules from a real Java lexer; the whole file about 60 lines. The recognizer still needs to handle string and character constants as well as comments so it doesn't get out of sync, trying to match a constant inside of the string, for example. The only unusual lexer rule performs the "match any character not matched by another lexer rule" functionality.

reference/FuzzyJava.g4

```
OTHER : . -> skip ;
```

This catchall lexer rule and the .*? subrule in the parser are the critical ingredients for fuzzy parsing.

Here's a sample file that we can run into the fuzzy parser:

reference/C.java

```
import java.util.*;
public class C {
    public static final int A = 1;
    public static final int B = 1;
    public void foo() { }
    public static final int C = 1;
}
```

And here's the build and test sequence:

```
$ antlr4 FuzzyJava.g4
$ javac FuzzyJava*.java
$ grun FuzzyJava file C.java
constant: A
constant: B
constant: C
```

Notice that it totally ignores everything except for the public static final int declarations. This all happens with only two parser rules.

15.7 Semantic Predicates

Semantic predicates, {...}?, are Boolean expressions written in the target language that indicate the validity of continuing the parse along the path "guarded" by the predicate. Predicates can appear anywhere within a parser rule just like actions can, but only those appearing on the left edge of alternatives can affect prediction (choosing between alternatives). We discussed predicates in detail in Chapter 11, *Altering the Parse with Semantic Predicates*, on page 189. This section provides all of the fine print regarding the use of semantic predicates in parser and lexer rules. Let's start by digging deeper into how the parser incorporates predicates into parsing decisions.

Making Predicated Parsing Decisions

ANTLR's general decision-making strategy is to find all *viable alternatives* and then ignore the alternatives guarded with predicates that currently evaluate to false. (A viable alternative is one that matches the current input.) If more than one viable alternative remains, the parser chooses the alternative specified first in the decision.

Consider a variant of C++ where array references also use parentheses instead of square brackets. If we predicate only one of the alternatives, we still have an ambiguous decision in expr.

```
expr:               ID '(' expr ')' // array reference (ANTLR picks this one)
    |   {istype()}? ID '(' expr ')' // ctor-style typecast
    |               ID '(' expr ')' // function call
    ;
```

In this case, all three alternatives are viable for input x(i). When x is not a type name, the predicate evaluates to false, leaving only the first and third alternatives as possible matches for expr. ANTLR automatically chooses the first alternative matching the array reference to resolve the ambiguity. Leaving ANTLR with more than one viable alternative because of too few predicates is probably not a good idea. It's best to cover n viable alternatives with at least n-1 predicates. In other words, don't build rules like expr with too few predicates.

Sometimes, the parser finds *multiple* visible predicates associated with a single choice. No worries. ANTLR just combines the predicates with appropriate logical operators to conjure up a single meta-predicate on-the-fly.

For example, the decision in rule stat joins the predicates from both alternatives of expr with the || operator to guard the second stat alternative.

```
stat:   decl | expr ;
decl:   ID ID ;
expr:   {istype()}? ID '(' expr ')'  // ctor-style typecast
    |   {isfunc()}? ID '(' expr ')'  // function call
    ;
```

The parser will predict an expr from stat only when istype()||isfunc() evaluates to true. This makes sense because the parser should choose to match an expression only if the upcoming ID is a type name or function name. It wouldn't make sense to test just one of the predicates in this case. Note that when the parser gets to expr itself, the parsing decision tests the predicates individually, one for each alternative.

If multiple predicates occur in a sequence, the parser joins them with the && operator. For example, consider changing stat to include a predicate before the call to expr.

```
stat:   decl | {java5}? expr ;
```

In this case, the parser would predict the second alternative only if java5&&(istype()||isfunc()) evaluated to true.

Turning to the code inside the predicates themselves now, keep in mind the following guidelines:

Use meaningful predicates.
ANTLR assumes that your predicates actually resolve the ambiguity. For example, ANTLR has no idea that the following predicates don't meaningfully resolve the ambiguity between the two alternatives:

```
expr:   {isTuesday()}?  ID '(' expr ')'  // ctor-style typecast
    |   {isHotOutside}? ID '(' expr ')'  // function call
    ;
```

Predicates must be free from side effects. ANTLR assumes that it can evaluate your predicates out of order or even multiple times, so don't use predicates like {$i++ < 10}?. It's almost certain that such predicates won't behave like you want.

Even when the parser isn't making decisions, predicates can deactivate alternatives, causing rules to fail. This happens when a rule has only a single alternative. There is no choice to make, but ANTLR evaluates the predicate as part of the normal parsing process, just like it does for actions. That means the following rule always fails to match:

```
prog:   {false}? 'return' INT ; // throws FailedPredicateException
```

ANTLR converts {false}? in the grammar to a conditional in the generated parser.

```
if ( !false ) throw new FailedPredicateException(...);
```

So far, all of the predicates we've seen have been visible and available to the prediction process, but that's not always the case.

Finding Visible Predicates

The parser will not evaluate predicates during prediction that occur after an action or token reference. Let's think about the relationship between actions and predicates first.

ANTLR has no idea what's inside the raw code of an action, so it must assume any predicate could depend on side effects of that action. Imagine an action that computed value x and a predicate that tested x. Evaluating that predicate before the action executed to create x would violate the implied order of operations within the grammar.

More importantly, the parser can't execute actions until it has decided which alternative to match. That's because actions have side effects and we can't undo things like print statements. For example, in the following rule, the parser can't execute the action in front of the {java5}? predicate before committing to that alternative:

```
@members {boolean allowgoto=false;}
stat: {System.out.println("goto"); allowgoto=true;} {java5}? 'goto' ID ';'
    | ...
    ;
```

If we can't execute the action during prediction, we shouldn't evaluate the {java5}? predicate because it depends on that action.

The prediction process also can't see through token references. Token references have the side effect of advancing the input one symbol. A predicate that tested the current input symbol would find itself out of sync if the parser shifted it over the token reference. For example, in the following grammar, the predicates expect getCurrentToken() to return an ID token:

```
stat: '{' decl '}'
    | '{' stat '}'
    ;
decl: {istype(getCurrentToken().getText())}? ID ID ';' ;
expr: {isvar(getCurrentToken().getText())}?  ID ;
```

The decision in stat can't test those predicates because, at the start of stat, the current token is a left curly. To preserve the semantics, ANTLR won't test the predicates in that decision.

Visible predicates are those that prediction encounters before encountering an action or token. The prediction process ignores nonvisible predicates, treating them as if they don't exist.

In rare cases, the parser won't be able to use a predicate, even if it's visible to a particular decision. That brings us to our next fine print topic.

Using Context-Dependent Predicates

A predicate that depends on a parameter or local variable of the surrounding rule is considered a *context-dependent predicate*. Clearly, we can evaluate such predicates only within the rules in which they're defined. For example, it makes no sense for the decision in the following prog to test the context-dependent predicate {$i<=5}?. That $i local variable is not even defined in prog.

```
prog:   vec5
    |   ...
    ;
vec5
locals [int i=1]
    :   ( {$i<=5}? INT {$i++;} )* // match 5 INTs
    ;
```

ANTLR ignores context-dependent predicates that it can't evaluate in the proper context. Normally the proper context is simply the rule defining the predicate, but sometimes the parser can't even evaluate a context-dependent predicate from within the same rule! Detecting these cases is done on-the-fly at runtime during adaptive *LL(*)* prediction.

For example, prediction for the optional branch of the else subrule in stat here "falls off" the end of stat and continues looking for symbols in the invoking prog rule:

```
prog:   stat+ ; // stat can follow stat
stat
locals [int i=0]
    :   {$i==0}? 'if' expr 'then' stat {$i=5;} ('else' stat)?
    |   'break' ';'
    ;
```

The prediction process is trying to figure out what can follow an if statement other than an else clause. Since the input can have multiple stats in a row, the prediction for the optional branch of the else subrule reenters stat. This time, of course, it gets a new copy of $i with a value of 0, not 5. ANTLR ignores context-dependent predicate {$i==0}? because it knows that the parser isn't in the original stat call. The predicate would test a different version of $i, so the parser can't evaluate it.

The fine print for predicates in the lexer more or less follows these same guidelines, except of course lexer rules can't have parameters and local variables. Let's look at all of the lexer-specific guidelines in the next section.

Predicates in Lexer Rules

In parser rules, predicates must appear on the left edge of alternatives to aid in alternative prediction. Lexers, on the other hand, prefer predicates on the right edge of lexer rules because they choose rules after seeing a token's entire text. Predicates in lexer rules can technically be anywhere within the rule. Some positions might be more or less efficient than others; ANTLR makes no guarantees about the optimal spot. A predicate in a lexer rule might be executed multiple times even during a single token match. You can embed multiple predicates per lexer rule, and they are evaluated as the lexer reaches them during matching.

Loosely speaking, the lexer's goal is to choose the rule that matches the most input characters. At each character, the lexer decides which rules are still viable. Eventually, only a single rule will be still viable. At that point, the lexer creates a token object according the rule's token type and matched text.

Sometimes the lexer is faced with more than a single viable matching rule. For example, input enum would match an ENUM rule and an ID rule. If the next character after enum is a space, neither rule can continue. The lexer resolves the ambiguity by choosing the viable rule specified first in the grammar. That's why we have to place keyword rules before an identifier rule like this:

```
ENUM : 'enum' ;
ID   : [a-z]+ ;
```

If, on the other hand, the next character after input enum is a letter, then only ID is viable.

Predicates come into play by pruning the set of viable lexer rules. When the lexer encounters a false predicate, it deactivates that rule just like parsers deactivate alternatives with false predicates.

Like parser predicates, lexer predicates can't depend on side effects from lexer actions. That's because actions can execute only after the lexer positively identifies the rule to match. Since predicates are part of the rule selection process, they can't rely on action side effects. Lexer actions must appear after predicates in lexer rules. As an example, here's another way to match enum as a keyword in the lexer:

reference/Enum3.g4

```
ENUM:   [a-z]+ {getText().equals("enum")}?
        {System.out.println("enum!");}
    ;
ID  :   [a-z]+ {System.out.println("ID "+getText());} ;
```

The print action in ENUM appears last and executes only if the current input matches [a-z]+ and the predicate is true. Let's build and test Enum3 to see whether it distinguishes between enum and an identifier.

```
⇛ $ antlr4 Enum3.g4
⇛ $ javac Enum3.java
⇛ $ grun Enum3 tokens
⇛ enum abc
⇛ EOF
❮ enum!
  ID abc
```

That works great, but it's really just for instructional purposes. It's easier to understand and more efficient to match enum keywords with a simple rule like this:

```
ENUM : 'enum' ;
```

15.8 Options

You can specify a number of options at the grammar and rule element levels. (There are currently no rule options.) These change how ANTLR generates code from your grammar. The general syntax is as follows:

```
options { name1=value1; ... nameN=valueN; } // ANTLR not target language syntax
```

where a value can be an identifier, a qualified identifier (for example, a.b.c), a string, a multiline string in curly braces {...}, and an integer.

Grammar Options

All grammars can use the following options. In combined grammars, all options except language pertain only to the generated parser. Options may be set either within the grammar file using the options syntax (described earlier) or when invoking ANTLR on the command line, using the -D option (see Section 15.9, *ANTLR Tool Command-Line Options*, on page 294.) The following examples demonstrate both mechanisms; note that -D overrides options within the grammar:

superClass Set the superclass of the generated parser or lexer. For combined grammars, it sets the superclass of the parser.

```
$ cat Hi.g4
grammar Hi;
a : 'hi' ;
$ antlr4 -DsuperClass=XX Hi.g4
$ grep 'public class' HiParser.java
public class HiParser extends XX {
$ grep 'public class' HiLexer.java
public class HiLexer extends Lexer {
```

language Generate code in the indicated language, if ANTLR is able to do so. Otherwise, you will see an error message like this:

```
$ antlr4 -Dlanguage=C MyGrammar.g4
error(31):  ANTLR cannot generate C code as of version 4.0
```

tokenVocab ANTLR assigns token type numbers to the tokens as it encounters them in a file. To use different token type values, such as with a separate lexer, use this option to have ANTLR pull in the .tokens file. ANTLR generates a .tokens file from each grammar.

```
$ cat SomeLexer.g4
lexer grammar SomeLexer;
ID : [a-z]+ ;
$ cat R.g4
parser grammar R;
```

```
options {tokenVocab=SomeLexer;}
tokens {A,B,C} // normally, these would be token types 1, 2, 3
a : ID ;
$ antlr4 SomeLexer.g4
$ cat SomeLexer.tokens
ID=1
$ antlr4 R.g4
$ cat R.tokens
A=2
B=3
C=4
ID=1
```

TokenLabelType ANTLR normally uses type Token when it generates variables referencing tokens. If you have passed a TokenFactory to your parser and lexer so that they create custom tokens, you should set this option to your specific type. This ensures that the context objects know your type for fields and method return values.

```
$ cat T2.g4
grammar T2;
options {TokenLabelType=MyToken;}
a : x=ID ;
$ antlr4 T2.g4
$ grep MyToken T2Parser.java
    public MyToken x;
```

Rule Options

There are currently no valid rule-level options, but the tool still supports the following syntax for future use:

```
rulename
options {...}
    :   ...
    ;
```

Rule Element Options

Token options have the form *T<name=value>*, as we saw in Section 5.4, *Dealing with Precedence, Left Recursion, and Associativity*, on page 69. The only token option is assoc, and it accepts values left and right. Figure 13, *A sample grammar*, on page 294 shows a sample grammar with a left-recursive expression rule that specifies a token option on the '^' exponent operator token.

Semantic predicates also accept an option, per *Catching Failed Semantic Predicates*, on page 166. The only valid option is the fail option, which takes either a string literal in double quotes or an action that evaluates to a string.

reference/ExprLR.g4
```
grammar ExprLR;

expr : expr '^'<assoc=right> expr
     | expr '*' expr  // match subexpressions joined with '*' operator
     | expr '+' expr  // match subexpressions joined with '+' operator
     | INT            // matches simple integer atom
     ;

INT : '0'..'9'+ ;
WS  : [ \n]+ -> skip ;
```

Figure 13—A sample grammar

The string literal or string result from the action should be the message to emit upon predicate failure.

errors/VecMsg.g4
```
ints[int max]
locals [int i=1]
    :   INT ( ',' {$i++;} {$i<=$max}?<fail={"exceeded max "+$max}> INT )*
    ;
```

The action can execute a function as well as compute a string when a predicate fails: {...}?<fail={doSomethingAndReturnAString()}>.

15.9 ANTLR Tool Command-Line Options

If you invoke the ANTLR tool without command-line arguments, you'll get a help message.

```
$ antlr4
ANTLR Parser Generator  Version 4.0
 -o ___              specify output directory where all output is generated
 -lib ___            specify location of grammars, tokens files
 -atn                generate rule augmented transition network diagrams
 -encoding ___       specify grammar file encoding; e.g., euc-jp
 -message-format ___ specify output style for messages in antlr, gnu, vs2005
 -listener           generate parse tree listener (default)
 -no-listener        don't generate parse tree listener
 -visitor            generate parse tree visitor
 -no-visitor         don't generate parse tree visitor (default)
 -package ___        specify a package/namespace for the generated code
 -depend             generate file dependencies
 -D<option>=value    set/override a grammar-level option
 -Werror             treat warnings as errors
 -XdbgST             launch StringTemplate visualizer on generated code
 -Xforce-atn         use the ATN simulator for all predictions
 -Xlog               dump lots of logging info to antlr-timestamp.log
```

Here are more details on the options:

-o *outdir*

ANTLR generates output files in the current directory by default. This option specifies the output directory where ANTLR should generate parsers, listeners, visitors, and .tokens files.

```
$ antlr4 -o /tmp T.g4
$ ls /tmp/T*
/tmp/T.tokens             /tmp/TListener.java
/tmp/TBaseListener.java   /tmp/TParser.java
```

-lib *libdir*

When looking for .tokens files and imported grammars, ANTLR normally looks in the current directory. This option specifies which directory to look in instead.

```
$ cat /tmp/B.g4
parser grammar B;
x : ID ;
$ cat A.g4
grammar A;
import B;
s : x ;
ID : [a-z]+ ;
$ antlr4 -lib /tmp A.g4
```

-atn

This generates DOT graph files that represent the internal augmented transition network (ATN) data structures that ANTLR uses to represent grammars. The files come out as *Grammar.rule*.dot. If the grammar is a combined grammar, the lexer rules are named *Grammar*Lexer.*rule*.dot.

```
$ cat A.g4
grammar A;
s : b ;
b : ID ;
ID : [a-z]+ ;
$ antlr4 -atn A.g4
$ ls *.dot
A.b.dot         A.s.dot         ALexer.ID.dot
```

-encoding *encodingname*

By default ANTLR loads grammar files using the UTF-8 encoding, which is a very common character file encoding that degenerates to ASCII for characters that fit in one byte. There are many character file encodings from around the world. If that grammar file is not the default encoding for your locale, you need this option so that ANTLR can properly interpret

grammar files (such as the one below). This does not affect the input to the generated parsers, just the encoding of the grammars themselves.

```
# my locale is en_US on a Mac OS X box
# I saved this file with a UTF-8 encoding to handle grammar name 外 (\uCDE2)
# inside the grammar file
$ cat 外.g4
grammar 外;
a : 'foreign' ;
$ antlr4 -encoding UTF-8 外.g4
$ ls 外*.java
外BaseListener.java 外Listener.java
外Lexer.java 外Parser.java
$ javac -encoding UTF-8 外*.java
```

-message-format *format*

ANTLR generates warning and error messages using templates from directory tool/resources/org/antlr/v4/tool/templates/messages/formats. By default, ANTLR uses the antlr.stg (StringTemplate group) file. You can change this to gnu or vs2005 to have ANTLR generate messages appropriate for Emacs or Visual Studio. To make your own called *X*, create resource org/antlr/v4/tool/templates/messages/formats/*X* and place it in the CLASSPATH.

-listener

This option tells ANTLR to generate a parse-tree listener and is the default.

-no-listener

This option tells ANTLR not to generate a parse-tree listener.

-visitor

ANTLR does not generate parse-tree visitors by default. This option turns that feature on. ANTLR can generate both parse-tree listeners and visitors; this option and -listener aren't mutually exclusive.

-no-visitor

Tell ANTLR not to generate a parse-tree visitor; this is the default.

-package

Use this option to specify a package or namespace for ANTLR-generated files. Alternatively, you can add an @header {...} action, but that ties the grammar to a specific language. If you use this option and @header, make sure that the header action does not contain a package specification; otherwise, the generated code will have two of them.

-depend

Instead of generating a parser and/or lexer, generate a list of file dependencies, one per line. The output shows what each grammar depends on and what it generates. This is useful for build tools that need to know ANTLR grammar dependencies. Here's an example:

```
$ antlr4 -depend T.g
T.g: A.tokens
TParser.java : T.g
T.tokens : T.g
TLexer.java : T.g
TListener.java : T.g
TBaseListener.java : T.g
```

If you use -lib libdir with -depend and grammar option tokenVocab=A, then the dependencies include the library path as well: T.g: libdir/A.tokens. The output is also sensitive to the -o outdir option: outdir/TParser.java : T.g.

-D<option>=value

Use this option to override or set a grammar-level option in the specified grammar or grammars. This option is useful for generating parsers in different languages without altering the grammar. (I expect to have other targets in the near future.)

```
$ antlr4 -Dlanguage=Java T.g4  # default
$ antlr4 -Dlanguage=C T.g4
error(31):  ANTLR cannot generate C code as of version 4.0
```

-Werror

As part of a large build, ANTLR warning messages could go unnoticed. Turn on this option to have warnings treated as errors, causing the ANTLR tool to report failure back to the invoking command-line shell.

There are also some extended options that are useful mainly for debugging ANTLR itself.

-XdbgST

For those building a code generation target, this option brings up a window showing the generated code and the templates used to generate that code. It invokes the StringTemplate inspector window.

-Xforce-atn

ANTLR normally builds traditional "switch on token type" decisions where possible (one token of lookahead is sufficient to distinguish between all alternatives in a decision). To force even these simple decisions into the adaptive *LL(*)* mechanism, use this option.

-Xlog

This option creates a log file containing lots of information messages from ANTLR as it processes your grammar. If you would like to see how ANTLR translates your left-recursive rules, turn on this option and look in the resulting log file.

```
$ antlr4 -Xlog T.g4
wrote ./antlr-2012-09-06-17.56.19.log
```

APPENDIX 1

Bibliography

[ALSU06] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools* . Addison-Wesley Longman, Reading, MA, Second, 2006.

[Gro90] Josef Grosch. Efficient and Comfortable Error Recovery in Recursive Descent Parsers. *Structured Programming*. 11[3]:129–140, 1990.

[Par09] Terence Parr. *Language Implementation Patterns*. The Pragmatic Bookshelf, Raleigh, NC and Dallas, TX, 2009.

[Top82] Rodney W. Topor. A note on error recovery in recursive descent parsers. *SIGPLAN Notices*. 17[2]:37–40, 1982.

[Wir78] Niklaus Wirth. *Algorithms + Data Structures = Programs*. Prentice Hall, Englewood Cliffs, NJ, 1978.

Index

U

V

W

X

Long live the command line!

Use tmux for incredible mouse-free productivity, and learn how to create professional command-line apps.

Your mouse is slowing you down. The time you spend context switching between your editor and your consoles eats away at your productivity. Take control of your environment with tmux, a terminal multiplexer that you can tailor to your workflow. Learn how to customize, script, and leverage tmux's unique abilities and keep your fingers on your keyboard's home row.

Brian P. Hogan
(88 pages) ISBN: 9781934356968. $11.00
http://pragprog.com/book/bhtmux

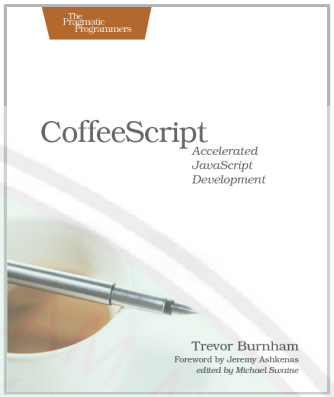
Speak directly to your system. With its simple commands, flags, and parameters, a well-formed command-line application is the quickest way to automate a backup, a build, or a deployment and simplify your life.

David Bryant Copeland
(200 pages) ISBN: 9781934356913. $33
http://pragprog.com/book/dccar

Welcome to the New Web

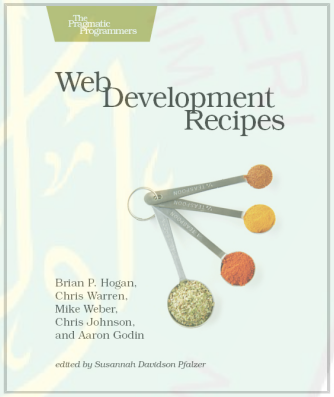
You need a better JavaScript and better recipes that professional web developers use every day. Start here.

CoffeeScript is JavaScript done right. It provides all of JavaScript's functionality wrapped in a cleaner, more succinct syntax. In the first book on this exciting new language, CoffeeScript guru Trevor Burnham shows you how to hold onto all the power and flexibility of JavaScript while writing clearer, cleaner, and safer code.



Trevor Burnham
(160 pages) ISBN: 9781934356784. $29
http://pragprog.com/book/tbcoffee

Modern web development takes more than just HTML and CSS with a little JavaScript mixed in. Clients want more responsive sites with faster interfaces that work on multiple devices, and you need the latest tools and techniques to make that happen. This book gives you more than 40 concise, tried-and-true solutions to today's web development problems, and introduces new workflows that will expand your skillset.



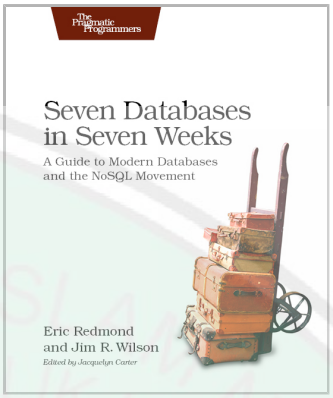
Brian P. Hogan, Chris Warren, Mike Weber, Chris Johnson, Aaron Godin
(344 pages) ISBN: 9781934356838. $35
http://pragprog.com/book/wbdev

Seven Databases, Seven Languages

There's so much new to learn with the latest crop of NoSQL databases. And instead of learning a language a year, how about seven?

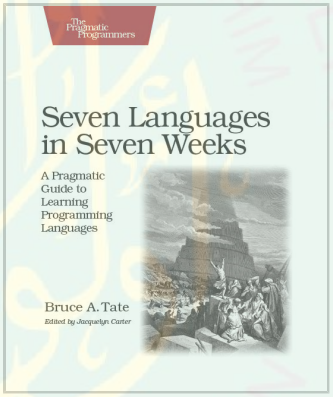
Data is getting bigger and more complex by the day, and so are your choices in handling it. From traditional RDBMS to newer NoSQL approaches, *Seven Databases in Seven Weeks* takes you on a tour of some of the hottest open source databases today. In the tradition of Bruce A. Tate's *Seven Languages in Seven Weeks*, this book goes beyond a basic tutorial to explore the essential concepts at the core of each technology.

Eric Redmond and Jim Wilson
(330 pages) ISBN: 9781934356920. $35
http://pragprog.com/book/rwdata



You should learn a programming language every year, as recommended by *The Pragmatic Programmer*. But if one per year is good, how about *Seven Languages in Seven Weeks*? In this book you'll get a hands-on tour of Clojure, Haskell, Io, Prolog, Scala, Erlang, and Ruby. Whether or not your favorite language is on that list, you'll broaden your perspective of programming by examining these languages side-by-side. You'll learn something new from each, and best of all, you'll learn how to learn a language quickly.

Bruce A. Tate
(328 pages) ISBN: 9781934356593. $34.95
http://pragprog.com/book/btlang

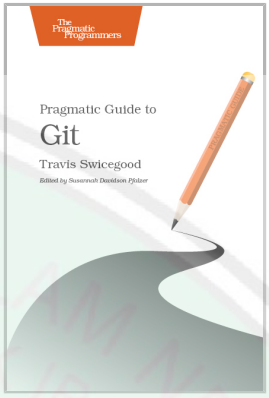


Pragmatic Guide Series

Get started quickly, with a minimum of fuss and hand-holding. The Pragmatic Guide Series features convenient, task-oriented two-page spreads. You'll find what you need fast, and get on with your work.

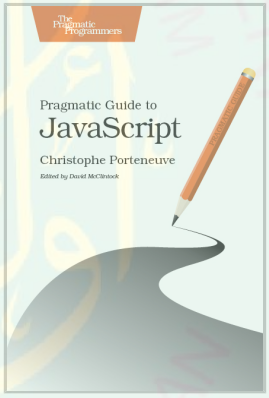
Need to learn how to wrap your head around Git, but don't need a lot of hand holding? Grab this book if you're new to Git, not to the world of programming. Git tasks displayed on two-page spreads provide all the context you need, without the extra fluff.

Travis Swicegood
(160 pages) ISBN: 9781934356722. $25
http://pragprog.com/book/pg_git



JavaScript is everywhere. It's a key component of today's Web—a powerful, dynamic language with a rich ecosystem of professional-grade development tools, infrastructures, frameworks, and toolkits. This book will get you up to speed quickly and painlessly with the 35 key JavaScript tasks you need to know.

Christophe Porteneuve
(160 pages) ISBN: 9781934356678. $25
http://pragprog.com/book/pg_js

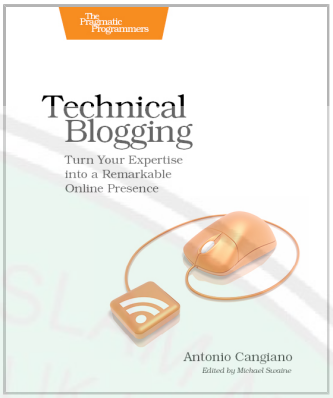


Career++

Ready to kick your career up to the next level? Start by growing a significant online presence, and then reinvigorate your job itself.

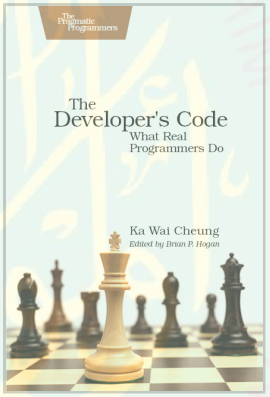
Technical Blogging is the first book to specifically teach programmers, technical people, and technically-oriented entrepreneurs how to become successful bloggers. There is no magic to successful blogging; with this book you'll learn the techniques to attract and keep a large audience of loyal, regular readers and leverage this popularity to achieve your goals.

Antonio Cangiano
(304 pages) ISBN: 9781934356883. $33
http://pragprog.com/book/actb



You're already a great coder, but awesome coding chops aren't always enough to get you through your toughest projects. You need these 50+ nuggets of wisdom. Veteran programmers: reinvigorate your passion for developing web applications. New programmers: here's the guidance you need to get started. With this book, you'll think about your job in new and enlightened ways.

Ka Wai Cheung
(250 pages) ISBN: 9781934356791. $29
http://pragprog.com/book/kcdc

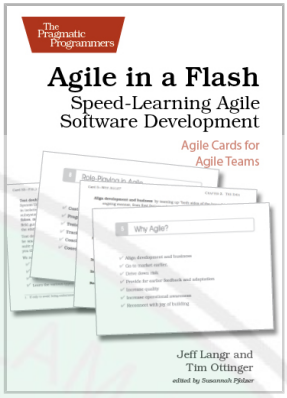


Be Agile

Don't just "do" agile; you want to *be* agile. We'll show you how.

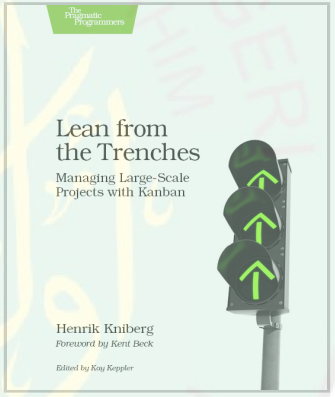
The best agile book isn't a book: *Agile in a Flash* is a unique deck of index cards that fit neatly in your pocket. You can tape them to the wall. Spread them out on your project table. Get stains on them over lunch. These cards are meant to be used, not just read.

Jeff Langr and Tim Ottinger
(110 pages) ISBN: 9781934356715. $15
http://pragprog.com/book/olag



You know the Agile and Lean development buzzwords, you've read the books. But when systems need a serious overhaul, you need to see how it works in real life, with real situations and people. *Lean from the Trenches* is all about actual practice. Every key point is illustrated with a photo or diagram, and anecdotes bring you inside the project as you discover why and how one organization modernized its workplace in record time.

Henrik Kniberg
(176 pages) ISBN: 9781934356852. $30
http://pragprog.com/book/hklean



The Pragmatic Bookshelf

The Pragmatic Bookshelf features books written by developers for developers. The titles continue the well-known Pragmatic Programmer style and continue to garner awards and rave reviews. As development gets more and more difficult, the Pragmatic Programmers will be there with more titles and products to help you stay on top of your game.

Visit Us Online

This Book's Home Page
http://pragprog.com/book/tpantlr2
Source code from this book, errata, and other resources. Come give us feedback, too!

Register for Updates
http://pragprog.com/updates
Be notified when updates and new books become available.

Join the Community
http://pragprog.com/community
Read our weblogs, join our online discussions, participate in our mailing list, interact with our wiki, and benefit from the experience of other Pragmatic Programmers.

New and Noteworthy
http://pragprog.com/news
Check out the latest pragmatic developments, new titles and other offerings.

Buy the Book

If you liked this eBook, perhaps you'd like to have a paper copy of the book. It's available for purchase at our store: *http://pragprog.com/titles/tpantlr2*

Contact Us

Online Orders:	*http://pragprog.com/catalog*
Customer Service:	*support@pragprog.com*
International Rights:	*translations@pragprog.com*
Academic Use:	*academic@pragprog.com*
Write for Us:	*http://pragprog.com/write-for-us*
Or Call:	+1 800-699-7764

The Pragmatic Programmers

The Definitive ANTLR Reference

Building Domain-Specific Languages

Terence Parr

What readers are saying about
The Definitive ANTLR Reference

Over the past few years ANTLR has proven itself as a solid parser generator. This book is a fine guide to making the best use of it.

► **Martin Fowler**
Chief Scientist, ThoughtWorks

The Definitive ANTLR Reference deserves a place in the bookshelf of anyone who ever has to parse or translate text. ANTLR is not just for language designers anymore.

► **Bob McWhirter**
Founder of the JBoss Rules Project (a.k.a. Drools), JBoss.org

Over the course of a career, developers move through a few stages of sophistication: becoming effective with a single programming language, learning which of several programming languages to use, and finally learning to tailor the language to the task at hand. This approach was previously reserved for those with an education in compiler development. Now, *The Definitive ANTLR Reference* reveals that it doesn't take a PhD to develop your own domain-specific languages, and you would be surprised how often it is worth doing. Take the next step in your career, and buy this book.

► **Neal Gafter**
Java Evangelist and Compiler Guru, Google (formerly at Sun Microsystems)

This book, especially the first section, really gave me a much better understanding of the principles of language recognition as a whole. I recommend this book to anyone without a background in language recognition looking to start using ANTLR or trying to understand the concept of language recognition.

► **Steve Ebersole**
Hibernate Lead Developer, Hibernate.org

Eclipse IDE users have become accustomed to cool features such as single-click navigation between symbol references and declarations, not to mention intelligent content assist. ANTLR v3 with its *LL(*)* parsing algorithm will help you immensely in building highly complex parsers to support these features. This book is a critical resource for Eclipse developers and others who want to take full advantage of the power of the new features in ANTLR.

► **Doug Schaefer**
Eclipse CDT Project Lead, Tools PMC Member, QNX Software Systems

Terence's new book is an excellent guide to ANTLR v3. It is very well written, with both the student and the developer in mind. The book does not assume compiler design experience. It provides the necessary background, from a pragmatic rather than a theoretical perspective, and it then eases the new user, whether someone with previous compiler design experience or not, into the use of the ANTLR tools. I recommend this book highly for anyone who needs to incorporate language capabilities into their software design.

► **Jesse Grodnik**
Software Development Manager, Sun Microsystems, Inc.

ANTLR v3 and *The Definitive ANTLR Reference* present a compelling package: an intuitive tool that handles complex recognition and translation tasks with ease and a clear book detailing how to get the most from it. The book provides an in-depth account of language translation utilizing the new powerful *LL(*)* parsing strategy. If you're developing translators, you can't afford to ignore this book!

► **Dermot O'Neill**
Senior Developer, Oracle Corporation

Whether you are a compiler newbie itching to write your own language or a jaded YACC veteran tired of shift-reduce conflicts, keep this book by your side. It is at once a tutorial, a reference, and an insider's viewpoint.

► **Sriram Srinivasan**
Formerly Principal Engineer, BEA/WebLogic

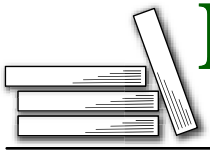
UNIVERSITAS ISLAM NEGERI
MAULANA MALIK IBRAHIM
PUSAT PERPUSTAKAAN

The Definitive ANTLR Reference

Building Domain-Specific Languages

Terence Parr

The Pragmatic Bookshelf
Raleigh, North Carolina Dallas, Texas



Our Pragmatic courses, workshops, and other products can help you and your team create better software and have more fun. For more information, as well as the latest Pragmatic titles, please visit us at

http://www.pragmaticprogrammer.com

Printed in the United States of America.

ISBN-10: 0-9787392-5-6

ISBN-13: 978-09787392-4-9

Printed on acid-free paper with 85% recycled, 30% post-consumer content.

First printing, May 2007

Version: 2007-5-17

This is Tom's fault.

UNIVERSITAS ISLAM NEGERI
MAULANA MALIK IBRAHIM
PUSAT PERPUSTAKAAN

Contents

Acknowledgments

A researcher once told me after a talk I had given that "It was clear there was a single mind behind these tools." In reality, there are many minds behind the ideas in my language tools and research, though I'm a benevolent dictator with specific opinions about how ANTLR should work. At the least, dozens of people let me bounce ideas off them, and I get a lot of great ideas from the people on the ANTLR interest list.[1]

Concerning the ANTLR v3 tool, I want to acknowledge the following contributors for helping with the design and functional requirements: Sriram Srinivasan (Sriram had a knack for finding holes in my *LL(*)* algorithm), Loring Craymer, Monty Zukowski, John Mitchell, Ric Klaren, Jean Bovet, and Kay Roepke. Matt Benson converted all my unit tests to use JUnit and is a big help with Ant files and other goodies. Juergen Pfundt contributed the ANTLR v3 task for Ant. I sing Jean Bovet's praises every day for his wonderful ANTLRWorks grammar development environment. Next comes the troop of hardworking ANTLR language target authors, most of whom contribute ideas regularly to ANTLR:[2] Jim Idle, Michael Jordan (no not that one), Ric Klaren, Benjamin Niemann, Kunle Odutola, Kay Roepke, and Martin Traverso.

I also want to thank (then Purdue) professors Hank Dietz and Russell Quong for their support early in my career. Russell also played a key role in designing the semantic and syntactic predicates mechanism.

The following humans provided technical reviews: Mark Bednarczyk, John Mitchell, Dermot O'Neill, Karl Pfalzer, Kay Roepke, Sriram Srinivasan, Bill Venners, and Oliver Ziegermann. John Snyders, Jeff Wilcox, and Kevin Ruland deserve special attention for their amazingly detailed feedback. Finally, I want to mention my excellent development editor Susannah Davidson Pfalzer. She made this a much better book.

1. See http://www.antlr.org:8080/pipermail/antlr-interest/.
2. See http://www.antlr.org/wiki/display/ANTLR3/Code+Generation+Targets.

Preface

In August 1993, I finished school and drove my overloaded moving van to Minnesota to start working. My office mate was a curmudgeonly astrophysicist named Kevin, who has since become a good friend. Kevin has told me on multiple occasions that only physicists do real work and that programmers merely support physicists. Because all I do is build language tools to support programmers, I am at least two levels of indirection away from doing anything useful.[3] Now, Kevin also claims that Fortran 77 is a good enough language for anybody and, for that matter, that Fortran 66 is probably sufficient, so one might question his judgment. But, concerning my usefulness, he was right—I am fundamentally lazy and would much rather work on something that made other people productive than actually do anything useful myself. This attitude has led to my guiding principle:[4]

Why program by hand in five days what you can spend five years of your life automating?

Here's the point: The first time you encounter a problem, writing a formal, general, and automatic mechanism is expensive and is usually overkill. From then on, though, you are much faster and better at solving similar problems because of your automated tool. Building tools can also be much more fun than your real job. Now that I'm a professor, I have the luxury of avoiding real work for a living.

---

3. The irony is that, as Kevin will proudly tell you, he actually played solitaire for at least a decade instead of doing research for his boss—well, when he wasn't scowling at the other researchers, at least. He claimed to have a winning streak stretching into the many thousands, but one day Kevin was caught overwriting the game log file to erase a loss (apparently per his usual habit). A holiday was called, and much revelry ensued.

4. Even as a young boy, I was fascinated with automation. I can remember endlessly building model ships and then trying to motorize them so that they would move around automatically. Naturally, I proceeded to blow them out of the water with firecrackers and rockets, but that's a separate issue.

My passion for the last two decades has been ANTLR, ANother Tool for Language Recognition. ANTLR is a parser generator that automates the construction of language recognizers. It is a program that writes other programs.

From a formal language description, ANTLR generates a program that determines whether sentences conform to that language. By adding code snippets to the grammar, the recognizer becomes a translator. The code snippets compute output phrases based upon computations on input phrases. ANTLR is suitable for the simplest and the most complicated language recognition and translation problems. With each new release, ANTLR becomes more sophisticated and easier to use. ANTLR is extremely popular with 5,000 downloads a month and is included on all Linux and OS X distributions. It is widely used because it:

- Generates human-readable code that is easy to fold into other applications
- Generates powerful recursive-descent recognizers using *LL(*)*, an extension to *LL(k)* that uses arbitrary lookahead to make decisions
- Tightly integrates StringTemplate,[5] a template engine specifically designed to generate structured text such as source code
- Has a graphical grammar development environment called ANTLRWorks[6] that can debug parsers generated in any ANTLR target language
- Is actively supported with a good project website and a high-traffic mailing list[7]
- Comes with complete source under the BSD license
- Is extremely flexible and automates or formalizes many common tasks
- Supports multiple target languages such as Java, C#, Python, Ruby, Objective-C, C, and C++

Perhaps most importantly, ANTLR is much easier to understand and use than many other parser generators. It generates essentially what you would write by hand when building a recognizer and uses technology that mimics how your brain generates and recognizes language (see Chapter 2, *The Nature of Computer Languages*, on page 34).

5. See http://www.stringtemplate.org.
6. See http://www.antlr.org/works.
7. See http://www.antlr.org:8080/pipermail/antlr-interest/.

You generate and recognize sentences by walking their implicit tree structure, from the most abstract concept at the root to the vocabulary symbols at the leaves. Each subtree represents a phrase of a sentence and maps directly to a rule in your grammar. ANTLR's grammars and resulting top-down recursive-descent recognizers thus feel very natural. ANTLR's fundamental approach dovetails your innate language process.

Why a Completely New Version of ANTLR?

For the past four years, I have been working feverishly to design and build ANTLR v3, the subject of this book. ANTLR v3 is a completely rewritten version and represents the culmination of twenty years of language research. Most ANTLR users will instantly find it familiar, but many of the details are different. ANTLR retains its strong mojo in this new version while correcting a number of deficiencies, quirks, and weaknesses of ANTLR v2 (I felt free to break backward compatibility in order to achieve this). Specifically, I didn't like the following about v2:[8]

- The v2 lexers were very slow albeit powerful.
- There were no unit tests for v2.
- The v2 code base was impenetrable. The code was never refactored to clean it up, partially for fear of breaking it without unit tests.
- The linear approximate *LL(k)* parsing strategy was a bit weak.
- Building a new language target duplicated vast swaths of logic and print statements.
- The AST construction mechanism was too informal.
- A number of common tasks were not easy (such as obtaining the text matched by a parser rule).
- It lacked the semantic predicates hoisting of ANTLR v1 (PCCTS).
- The v2 license/contributor trail was loose and made big companies afraid to use it.

ANTLR v3 is my answer to the issues in v2. ANTLR v3 has a very clean and well-organized code base with lots of unit tests. ANTLR generates extremely powerful *LL(*)* recognizers that are fast and easy to read.

8. See http://www.antlr.org/blog/antlr3/antlr2.bashing.tml for notes on what people did not like about v2. ANTLR v2 also suffered because it was designed and built while I was under the workload and stress of a new start-up (jGuru.com).

Many common tasks are now easy by default. For example, reading in some input, tweaking it, and writing it back out while preserving whitespace is easy. ANTLR v3 also reintroduces semantic predicates hoisting. ANTLR's license is now BSD, and all contributors must sign a "certificate of origin."[9] ANTLR v3 provides significant functionality beyond v2 as well:

- Powerful *LL(*)* parsing strategy that supports more natural grammars and makes it easier to build them
- Auto-backtracking mode that shuts off all grammar analysis warnings, forcing the generated parser to simply figure things out at runtime
- Partial parsing result memoization to guarantee linear time complexity during backtracking at the cost of some memory
- Jean Bovet's ANTLRWorks GUI grammar development environment
- StringTemplate template engine integration that makes generating structured text such as source code easy
- Formal AST construction rules that map input grammar alternatives to tree grammar fragments, making actions that manually construct ASTs no longer necessary
- Dynamically scoped attributes that allow distant rules to communicate
- Improved error reporting and recovery for generated recognizers
- Truly retargetable code generator; building a new target is a matter of defining StringTemplate templates that tell ANTLR how to generate grammar elements such as rule and token references

This book also provides a serious advantage to v3 over v2. Professionally edited and complete documentation is a big deal to developers. You can find more information about the history of ANTLR and its contributions to parsing theory on the ANTLR website.[10,11]

Look for *Improved in v3* and *New in v3* notes in the margin that highlight improvements or additions to v2.

9. See http://www.antlr.org/license.html.
10. See http://www.antlr.org/history.html.
11. See http://www.antlr.org/contributions.html.

Who Is This Book For?

The primary audience for this book is the practicing software developer, though it is suitable for junior and senior computer science undergraduates. This book is specifically targeted at any programmer interested in learning to use ANTLR to build interpreters and translators for domain-specific languages. Beginners and experts alike will need this book to use ANTLR v3 effectively. For the most part, the level of discussion is accessible to the average programmer. Portions of Part III, however, require some language experience to fully appreciate. Although the examples in this book are written in Java, their substance applies equally well to the other language targets such as C, C++, Objective-C, Python, C#, and so on. Readers should know Java to get the most out of the book.

What's in This Book?

This book is the best, most complete source of information on ANTLR v3 that you'll find anywhere. The free, online documentation provides enough to learn the basic grammar syntax and semantics but doesn't explain ANTLR concepts in detail. This book helps you get the most out of ANTLR and is required reading to become an advanced user. In particular, Part III provides the only thorough explanation available anywhere of ANTLR's *LL(*)* parsing strategy.

This book is organized as follows. Part I introduces ANTLR, describes how the nature of computer languages dictates the nature of language recognizers, and provides a complete calculator example. Part II is the main reference section and provides all the details you'll need to build large and complex grammars and translators. Part III treks through ANTLR's predicated-*LL(*)* parsing strategy and explains the grammar analysis errors you might encounter. Predicated-*LL(*)* is a totally new parsing strategy, and Part III is essentially the only written documentation you'll find for it. You'll need to be familiar with the contents in order to build complicated translators.

Readers who are totally new to grammars and language tools should follow the chapter sequence in Part I as is. Chapter 1, *Getting Started with ANTLR*, on page 21 will familiarize you with ANTLR's basic idea; Chapter 2, *The Nature of Computer Languages*, on page 34 gets you ready to study grammars more formally in Part II; and Chapter 3, *A Quick Tour for the Impatient*, on page 59 gives your brain something

concrete to consider. Familiarize yourself with the ANTLR details in Part II, but I suggest trying to modify an existing grammar as soon as you can. After you become comfortable with ANTLR's functionality, you can attempt your own translator from scratch. When you get grammar analysis errors from ANTLR that you don't understand, then you need to dive into Part III to learn more about *LL(*)*.

Those readers familiar with ANTLR v2 should probably skip directly to Chapter 3, *A Quick Tour for the Impatient*, on page 59 to figure out how v3 differs. Chapter 4, *ANTLR Grammars*, on page 86 is also a good place to look for features that v3 changes or improves on.

If you are familiar with an older tool, such as YACC [Joh79], I recommend starting from the beginning of the book as if you were totally new to grammars and language tools. If you're used to JavaCC[12] or another top-down parser generator, you can probably skip Chapter 2, *The Nature of Computer Languages*, on page 34, though it is one of my favorite chapters.

I hope you enjoy this book and ANTLR v3 as much as I have enjoyed writing them!

Terence Parr
March 2007
University of San Francisco

12. See https://javacc.dev.java.net.

Part I

Introducing ANTLR and Computer Language Translation

Chapter 1

Getting Started with ANTLR

This is a reference guide for ANTLR: a sophisticated parser generator you can use to implement language interpreters, compilers, and other translators. This is not a compiler book, and it is not a language theory textbook. Although you can find many good books about compilers and their theoretical foundations, the vast majority of language applications are not compilers. This book is more directly useful and practical for building common, everyday language applications. It is densely packed with examples, explanations, and reference material focused on a single language tool and methodology.

Programmers most often use ANTLR to build translators and interpreters for *domain-specific languages* (DSLs). DSLs are generally very high-level languages tailored to specific tasks. They are designed to make their users particularly effective in a specific domain. DSLs include a wide range of applications, many of which you might not consider languages. DSLs include data formats, configuration file formats, network protocols, text-processing languages, protein patterns, gene sequences, space probe control languages, and domain-specific programming languages.

DSLs are particularly important to software development because they represent a more natural, high-fidelity, robust, and maintainable means of encoding a problem than simply writing software in a general-purpose language. For example, NASA uses domain-specific command languages for space missions to improve reliability, reduce risk, reduce cost, and increase the speed of development. Even the first Apollo guidance control computer from the 1960s used a DSL that supported vector computations.[1]

1. See http://www.ibiblio.org/apollo/assembly_language_manual.html.

This chapter introduces the main ANTLR components and explains how they all fit together. You'll see how the overall DSL translation problem easily factors into multiple, smaller problems. These smaller problems map to well-defined translation phases (lexing, parsing, and tree parsing) that communicate using well-defined data types and structures (characters, tokens, trees, and ancillary structures such as symbol tables). After this chapter, you'll be broadly familiar with all translator components and will be ready to tackle the detailed discussions in subsequent chapters. Let's start with the big picture.

1.1 The Big Picture

A translator maps each input sentence of a language to an output sentence. To perform the mapping, the translator executes some code you provide that operates on the input symbols and emits some output. A translator must perform different actions for different sentences, which means it must be able to recognize the various sentences.

Recognition is much easier if you break it into two similar but distinct tasks or phases. The separate phases mirror how your brain reads English text. You don't read a sentence character by character. Instead, you perceive a sentence as a stream of words. The human brain subconsciously groups character sequences into words and looks them up in a dictionary before recognizing grammatical structure. The first translation phase is called *lexical analysis* and operates on the incoming character stream. The second phase is called *parsing* and operates on a stream of vocabulary symbols, called *tokens*, emanating from the lexical analyzer. ANTLR automatically generates the lexical analyzer and parser for you by analyzing the grammar you provide.

Performing a translation often means just embedding *actions* (code) within the grammar. ANTLR executes an action according to its position within the grammar. In this way, you can execute different code for different phrases (sentence fragments). For example, an action within, say, an expression rule is executed only when the parser is recognizing an expression.

Some translations should be broken down into even more phases. Often the translation requires multiple passes, and in other cases, the translation is just a heck of a lot easier to code in multiple phases. Rather than reparse the input characters for each phase, it is more convenient to construct an intermediate form to pass between phases.

Language Translation Can Help You Avoid Work

In 1988, I worked in Paris for a robotics company. At the time, the company had a fairly demanding coding standard that required very formal and structured comments on each C function and file.

After finishing my compiler project, I was ready to head back to the United States and continue with my graduate studies. Unfortunately, the company was withholding my bonus until I followed its coding standard. The standard required all sorts of tedious information such as which functions were called in each function, the list of parameters, list of local variables, which functions existed in this file, and so on. As the company dangled the bonus check in front me, I blurted out, "All of that can be automatically generated!" Something clicked in my mind. Of course. Build a quick C parser that is capable of reading all my source code and generating the appropriate comments. I would have to go back and enter the written descriptions, but my translator would do the rest.

I built a parser by hand (this was right before I started working on ANTLR) and created template files for the various documentation standards. There were holes that my parser could fill in with parameters, variable lists, and so on. It took me two days to build the translator. I started it up, went to lunch, and came back to commented source code. I quickly entered the necessary descriptions, collected my bonus, and flew back to Purdue University with a smirk on my face.

The point is that knowing about computer languages and language technology such as ANTLR will make your coding life much easier. Don't be afraid to build a human-readable configuration file (I implore everyone to please stop using XML as a human interface!) or to build domain-specific languages to make yourself more efficient. Designing new languages and building translators for existing languages, when appropriate, is the hallmark of a sophisticated developer.

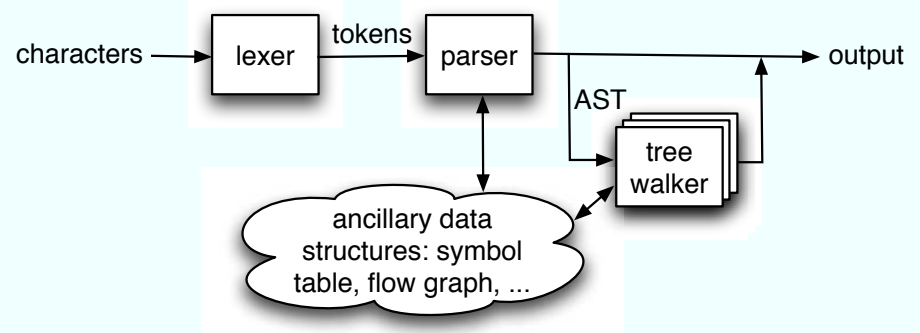


Figure 1.1: Overall translation data flow; edges represent data structure flow, and squares represent translation phases

This intermediate form is usually a tree data structure, called an *abstract syntax tree* (AST), and is a highly processed, condensed version of the input. Each phase collects more information or performs more computations. A final phase, called the *emitter*, ultimately emits output using all the data structures and computations from previous phases.

Figure 1.1 illustrates the basic data flow of a translator that accepts characters and emits output. The lexical analyzer, or *lexer*, breaks up the input stream into tokens. The parser feeds off this token stream and tries to recognize the sentence structure. The simplest translators execute actions that immediately emit output, bypassing any further phases.

Another kind of simple translator just constructs an internal data structure—it doesn't actually emit output. A configuration file reader is the best example of this kind of translator. More complicated translators use the parser only to construct ASTs. Multiple *tree parsers* (depth-first tree walkers) then scramble over the ASTs, computing other data structures and information needed by future phases. Although it is not shown in this figure, the final emitter phase can use templates to generate structured text output.

A template is just a text document with holes in it that an emitter can fill with values. These holes can also be expressions that operate on the incoming data values. ANTLR formally integrates the StringTemplate engine to make it easier for you to build emitters (see Chapter 9, *Generating Structured Text with Templates and Grammars*, on page 206).

StringTemplate is a domain-specific language for generating structured text from internal data structures that has the flavor of an output grammar. Features include template group inheritance, template polymorphism, lazy evaluation, recursion, output autoindentation, and the new notions of group interfaces and template regions.[2] StringTemplate's feature set is driven by solving real problems encountered in complicated systems. Indeed, ANTLR makes heavy use of StringTemplate to translate grammars to executable recognizers. Each ANTLR language target is purely a set of templates and fed by ANTLR's internal retargetable code generator.

Now, let's take a closer look at the data objects passed between the various phases in Figure 1.1, on the previous page. Figure 1.2, on the following page, illustrates the relationship between characters, tokens, and ASTs. Lexers feed off characters provided by a CharStream such as ANTLRStringStream or ANTLRFileStream. These predefined streams assume that the entire input will fit into memory and, consequently, buffer up all characters. Rather than creating a separate string object per token, tokens can more efficiently track indexes into the character buffer.

Similarly, rather than copying data from tokens into tree nodes, ANTLR AST nodes can simply point at the token from which they were created. CommonTree, for example, is a predefined node containing a Token *payload*. The type of an ANTLR AST node is treated as an Object so that there are no restrictions whatsoever on your tree data types. In fact, you can even make your Token objects double as AST nodes to avoid extra object instantiations. The relationship between the data types described in Figure 1.2, on the next page, is very efficient and flexible.

The tokens in the figure with checkboxes reside on a hidden *channel* that the parser does not see. The parser *tunes* to a single channel and, hence, ignores tokens on any other channel. With a simple action in the lexer, you can send different tokens to the parser on different channels. For example, you might want whitespace and regular comments on one channel and Javadoc comments on another when parsing Java. The token buffer preserves the relative token order regardless of the token channel numbers. The token channel mechanism is an elegant solution to the problem of ignoring but not throwing away whitespace and comments (some translators need to preserve formatting and comments).

2. Please see http://www.stringtemplate.org for more details. I mention these terms to entice readers to learn more about StringTemplate.

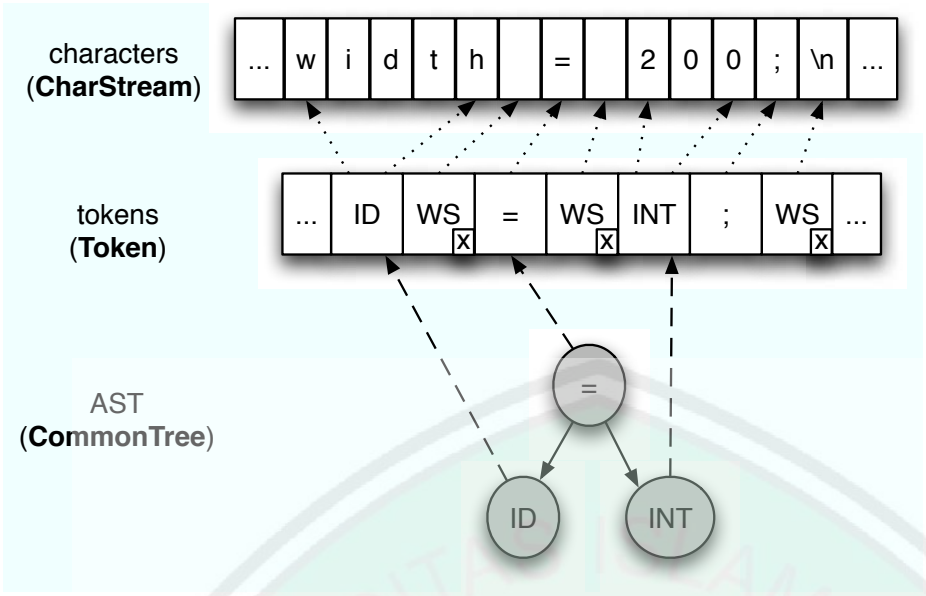


Figure 1.2: Relationship between characters, tokens, and ASTs; CharStream, Token, and CommonTree are ANTLR runtime types

As you work through the examples and discussions later in this book, it may help to keep in mind the analogy described in the next section.

1.2 An A-mazing Analogy

This book focuses primarily on two topics: the discovery of the implicit tree structure behind input sentences and the generation of structured text. At first glance, some of the language terminology and technology in this book will be unfamiliar. Don't worry. I'll define and explain everything, but it helps to keep in mind a simple analogy as you read.

Imagine a maze with a single entrance and single exit that has words written on the floor. Every path from entrance to exit generates a sentence by "saying" the words in sequence. In a sense, the maze is analogous to a grammar that defines a language.

You can also think of a maze as a sentence recognizer. Given a sentence, you can match its words in sequence with the words along the floor. Any sentence that successfully guides you to the exit is a valid sentence (a *passphrase*) in the language defined by the maze.

Language recognizers must discover a sentence's implicit tree structure piecemeal, one word at a time. At almost every word, the recognizer must make a decision about the interpretation of a phrase or subphrase. Sometimes these decisions are very complicated. For example, some decisions require information about previous decision choices or even future choices. Most of the time, however, decisions need just a little bit of *lookahead* information. Lookahead information is analogous to the first word or words down each path that you can see from a given fork in the maze. At a fork, the next words in your input sentence will tell you which path to take because the words along each path are different. Chapter 2, *The Nature of Computer Languages*, on page 34 describes the nature of computer languages in more detail using this analogy. You can either read that chapter first or move immediately to the quick ANTLR tour in Chapter 3, *A Quick Tour for the Impatient*, on page 59.

In the next two sections, you'll see how to map the big picture diagram in Figure 1.1, on page 24, into Java code and also learn how to execute ANTLR.

1.3 Installing ANTLR

ANTLR is written in Java, so you must have Java installed on your machine even if you are going to use ANTLR with, say, Python. ANTLR requires a Java version of 1.4 or higher. Before you can run ANTLR on your grammar, you must install ANTLR by downloading it[3] and extracting it into an appropriate directory. You do not need to run a configuration script or alter an ANTLR configuration file to properly install ANTLR. If you want to install ANTLR in /usr/local/antlr-3.0, do the following:

```
$ cd /usr/local
$ tar xvfz antlr-3.0.tar.gz
antlr-3.0/
antlr-3.0/build/
antlr-3.0/build.properties
antlr-3.0/build.xml
antlr-3.0/lib/
antlr-3.0/lib/antlr-3.0.jar
...
$
```

3. See http://www.antlr.org/download.html.

As of 3.0, ANTLR v3 is still written in the previous version of ANTLR, 2.7.7, and with StringTemplate 3.0. This means you need both of those libraries to run the ANTLR v3 tool. You do not need the ANTLR 2.7.7 JAR to run your generated parser, and you do not need the StringTemplate JAR to run your parser unless you use template construction rules. (See Chapter 9, *Generating Structured Text with Templates and Grammars*, on page 206.) Java scans the CLASSPATH environment variable looking for JAR files and directories containing Java .class files. You must update your CLASSPATH to include the antlr-2.7.7.jar, stringtemplate-3.0.jar, and antlr-3.0.jar libraries.

Just about the only thing that can go wrong with installation is setting your CLASSPATH improperly or having another version of ANTLR in the CLASSPATH. Note that some of your other Java libraries might use ANTLR (such as BEA's WebLogic) without your knowledge.

To set the CLASSPATH on Mac OS X or any other Unix-flavored box with the **bash** shell, you can do the following:

```
$ export CLASSPATH="$CLASSPATH:/usr/local/antlr-3.0/lib/antlr-3.0.jar:\
/usr/local/antlr-3.0/lib/stringtemplate-3.0.jar:\
/usr/local/antlr-3.0/lib/antlr-2.7.7.jar"
$
```

Don't forget the export. Without this, subprocesses you launch such as Java will not see the environment variable.

To set the CLASSPATH on Microsoft Windows XP, you'll have to set the environment variable using the System control panel in the Advanced subpanel. Click Environment Variables, and then click New in the top variable list. Also note that the path separator is a semicolon (;), not a colon (:), for Windows.

At this point, ANTLR should be ready to run. The next section provides a simple grammar you can use to check whether you have installed ANTLR properly.

1.4 Executing ANTLR and Invoking Recognizers

Once you have installed ANTLR, you can use it to translate grammars to executable Java code. Here is a sample grammar:

Download Introduction/T.g

```
grammar T;
/** Match things like "call foo;" */
r : 'call' ID ';' {System.out.println("invoke "+$ID.text);} ;
ID: 'a'..'z'+ ;
WS: (' '|'\n'|'\r')+   {$channel=HIDDEN;} ; // ignore whitespace
```

Java class Tool in package org.antlr contains the main program, so you execute ANTLR on grammar file T.g as follows:

```
$ java org.antlr.Tool T.g
ANTLR Parser Generator  Version 3.0  1989-2007
$ ls
T.g             TLexer.java     T__.g
T.tokens        TParser.java
$
```

As you can see, ANTLR generates a number of support files as well as the lexer, TLexer.java, and the parser, TParser.java, in the current directory.

To test the grammar, you'll need a main program that invokes start rule r from the grammar and reads from standard input. Here is program Test.java that embodies part of the data flow shown in Figure 1.1, on page 24:

Download Introduction/Test.java

```
import org.antlr.runtime.*;

public class Test {
    public static void main(String[] args) throws Exception {
        // create a CharStream that reads from standard input
        ANTLRInputStream input = new ANTLRInputStream(System.in);

        // create a lexer that feeds off of input CharStream
        TLexer lexer = new TLexer(input);

        // create a buffer of tokens pulled from the lexer
        CommonTokenStream tokens = new CommonTokenStream(lexer);

        // create a parser that feeds off the tokens buffer
        TParser parser = new TParser(tokens);
        // begin parsing at rule r
        parser.r();
    }
}
```

What's Available at the ANTLR Website?

At the http://www.antlr.org website, you will find a great deal of information and support for ANTLR. The site contains the ANTLR download, the ANTLRWorks graphical user interface (GUI) development environment, the ANTLR documentation, prebuilt grammars, examples, articles, a file-sharing area, the tech support mailing list, the wiki, and much more.

To compile everything and run the test rig, do the following (don't type the $ symbol—that's the command prompt):

```
⇐  $ javac TLexer.java TParser.java Test.java
⇐  $ java Test
⇐  call foo;
⇐  EOF
⇒  invoke foo
   $
```

In response to input call foo; followed by the newline, the translator emits invoke foo followed by the newline. Note that you must type the end-of-file character to terminate reading from standard input; otherwise, the program will stare at you for eternity.

This simple example does not include any ancillary data structures or intermediate-form trees. The embedded grammar action directly emits output invoke foo. See Chapter 7, *Tree Construction*, on page 162 and Chapter 8, *Tree Grammars*, on page 191 for a number of test rig examples that instantiate and launch tree walkers.

Before you begin developing a grammar, you should become familiar with ANTLRWorks, the subject of the next section. This ANTLR GUI will make your life much easier when building or debugging grammars.

1.5 ANTLRWorks Grammar Development Environment

ANTLRWorks is a GUI development environment written by Jean Bovet[4] that sits on top of ANTLR and helps you edit, navigate, and debug

4. See http://www.antlr.org/works. Bovet is the developer of ANTLRWorks, with some functional requirements from me. He began development during his master's degree at the University of San Francisco but is continuing to develop the tool.

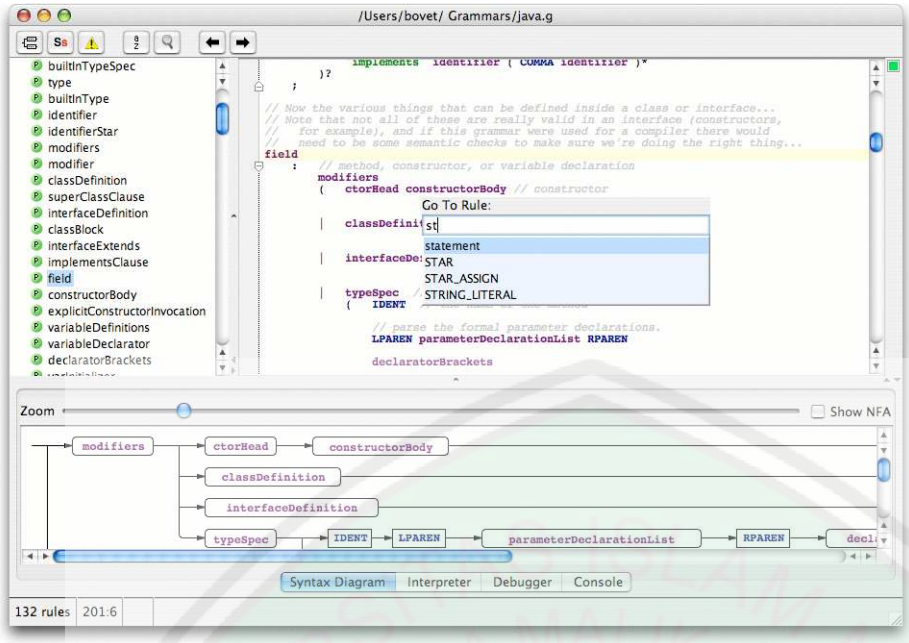


Figure 1.3: ANTLRWorks grammar development environment; grammar editor view

grammars. Perhaps most important, ANTLRWorks helps you resolve grammar analysis errors, which can be tricky to figure out manually. ANTLRWorks currently has the following main features:

- Grammar-aware editor
- Syntax diagram grammar view
- Interpreter for rapid prototyping
- Language-agnostic debugger for isolating grammar errors
- Nondeterministic path highlighter for the syntax diagram view
- Decision lookahead (DFA) visualization
- Refactoring patterns for many common operations such as "remove left-recursion" and "in-line rule"
- Dynamic parse tree view
- Dynamic AST view

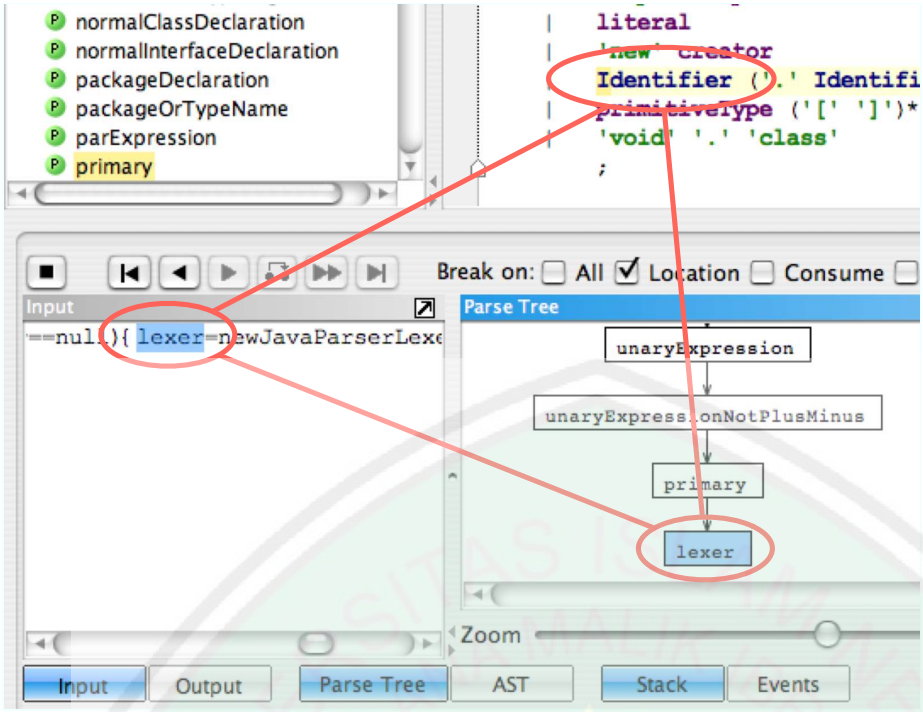


Figure 1.4: ANTLRWorks debugger while parsing Java code; the input, parse tree, and grammar are synched at all times

ANTLRWorks is written entirely in highly portable Java (using Swing) and is available as open source under the BSD license. Because ANTLRWorks communicates with running parsers via sockets, the ANTLRWorks debugger works with any ANTLR language target (assuming that the target runtime library has the necessary support code). At this point, ANTLRWorks has a prototype plug-in for IntelliJ[5] but nothing yet for Eclipse.

Figure 1.3, on the previous page, shows ANTLRWorks' editor in action with the Go To Rule pop-up dialog box. As you would expect, ANTLRWorks has the usual rule and token name autocompletion as well as syntax highlighting. The lower pane shows the syntax diagram for rule **field** from a Java grammar. When you have ambiguities in other non-

5. See http://plugins.intellij.net/plugin/?id=953.

determinisms in your grammar, the syntax diagram shows the multiple paths that can recognize the same input. From this visualization, you will find it straightforward to resolve the nondeterminisms. Part III of this book discusses ANTLR's *LL(*)* parsing strategy in detail and makes extensive use of the ambiguous path displays provided by ANTLRWorks.

Figure 1.4, on the preceding page, illustrates ANTLRWorks' debugger. The debugger provides a wealth of information and, as you can see, always keeps the various views in sync. In this case, the grammar matches input identifier lexer with grammar element **Identifier**; the parse tree pane shows the implicit tree structure of the input. For more information about ANTLRWorks, please see the user guide.[6]

This introduction gave you an overall view of what ANTLR does and how to use it. The next chapter illustrates how the nature of language leads to the use of grammars for language specification. The final chapter in Part I—Chapter 3, *A Quick Tour for the Impatient*, on page 59—demonstrates more of ANTLR's features by showing you how to build a calculator.

6. See http://www.antlr.org/works/doc/antlrworks.pdf.

Chapter 2

The Nature of Computer Languages

This book is about building translators with ANTLR rather than resorting to informal, arbitrary code. Building translators with ANTLR requires you to use a formal language specification called a *grammar*. To understand grammars and to understand their capabilities and limitations, you need to learn about the nature of computer languages. As you might expect, the nature of computer languages dictates the way you specify languages with grammars.

The whole point of writing a grammar is so ANTLR can automatically build a program for you that recognizes sentences in that language. Unfortunately, starting the learning process with grammars and language recognition is difficult (from my own experience and from the questions I get from ANTLR users). The purpose of this chapter is to teach you first about language generation and then, at the very end, to describe language recognition. Your brain understands language generation very well, and recognition is the dual of generation. Once you understand language generation, learning about grammars and language recognition is straightforward.

Here is the central question you must address concerning generation: how can you write a stream of words that transmits information beyond a simple list of items? In English, for example, how can a stream of words convey ideas about time, geometry, and why people don't use turn signals? It all boils down to the fact that sentences are not just clever sequences of words, as Steven Pinker points out in *The Language Instinct* [Pin94]. The implicit structure of the sentence, not just

Example Demonstrating That Structure Imparts Meaning

Humans are hardwired to recognized the implicit structure within a sentence (a linear sequence of words). Consider this English sentence:

"Terence says Sriram likes chicken tikka."

The sentence's subject is "Terence," and the verb is "says." Now, interpret the sentence differently using "likes" as the verb:

"Terence, says Sriram, likes chicken tikka."

The commas alter the sentence structure in the same way that parentheses alter operator precedence in expressions. The key observation is that the same sequence of words means two different things depending on the structure you assume.

the words and the sequence, imparts the meaning. What exactly is sentence structure? Unfortunately, the answer requires some background to answer properly. On the bright side, the search for a precise definition unveils some important concepts, terminology, and language technology along the way. In this chapter, we'll cover the following topics:

- State machines (DFAs)
- Sentence word order and dependencies that govern complex language generation
- Sentence tree structure
- Pushdown machines (syntax diagrams)
- Language ambiguities
- Lexical phrase structure
- What we mean by "recognizing a sentence"

Let's begin by demonstrating that generating sentences is not as simple as picking appropriate words in a sequence.

2.1 Generating Sentences with State Machines

When I was a suffering undergraduate student at Purdue University (back before GUIs), I ran across a sophisticated documentation generator that automatically produced verbose, formal-sounding manuals. You could read about half a paragraph before your mind said, "Whoa!

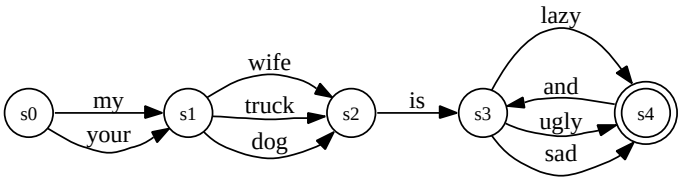


Figure 2.1: A state machine that generates blues lyrics

That doesn't make sense." Still, it was amazing that a program could produce a document that, at first glance, was human-generated. How could that program generate English sentences? Believe it or not, even a simple "machine" can generate a large number of proper sentences. Consider the blues lyrics machine in Figure 2.1 that generates such valid sentences as "My wife is sad" and "My dog is ugly and lazy."[1,2]

The *state machine* has states (circles) and transitions (arrows) labeled with vocabulary symbols. The transitions are directed (one-way) connections that govern navigation among the states. Machine execution begins in state s0, the *start state*, and stops in s4, the *accept state*. Transitioning from one state to another emits the label on the transition. At each state, pick a transition, "say" the label, and move to the target state. The full name for this machine is *deterministic finite automaton* (DFA). You'll see the acronym DFA used extensively in Chapter 11, *LL(*) Parsing*, on page 262.

DFAs are relatively easy to understand and seem to generate some sophisticated sentences, but they aren't powerful enough to generate all programming language constructs. The next section points out why DFAs are underpowered.

1. Pinker's book has greatly influenced my thinking about languages. This state machine and related discussion were inspired by the machines in *The Language Instinct*.
2. What happens if you run the blues machine backward? As the old joke goes, "You get your dog back, your wife back. . . ."

The Maze as a Language Generator

A state machine is analogous to a maze with words written on the floor. The words along each path through the maze from the entrance to the exit represent a sentence. The set of all paths through the maze represents the set of all sentences and, hence, defines the language.

Imagine that at least one loopback exists along some path in the maze. You could walk around forever, generating an infinitely long sentence. The maze can, therefore, simulate a finite or infinite language generator just like a state machine.

Finite State Machines

The blues lyrics state machine is called a *finite state automaton*. An *automaton* is another word for machine, and *finite* implies the machine has a fixed number of states. Note that even though there are only five states, the machine can generate an infinite number of sentences because of the "and" loop transition from s4 to s3. Because of that transition, the machine is considered *cyclic*. All cyclic machines generate an infinite number of sentences, and all acyclic machines generate a finite set of sentences. ANTLR's *LL(*)* parsing strategy, described in detail in Part III, is stronger than traditional *LL(k)* because *LL(*)* uses cyclic prediction machines whereas *LL(k)* uses acyclic machines.

One of the most common acronyms you'll see in Part III of this book is DFA, which stands for *deterministic finite automaton*. A deterministic automaton (state machine) is an automaton where all transition labels emanating from any single state are unique. In other words, every state transitions to exactly one other state for a given label.

A final note about state machines. They do not have a memory. States do not know which states, if any, the machine has visited previously. This weakness is central to why state machines generate some invalid sentences. Analogously, state machines are too weak to recognize many common language constructs.

2.2 The Requirements for Generating Complex Language

Is the lyrics state machine correct in the sense it generates valid blues sentences and only valid sentences? Unfortunately, no. The machine can also generate invalid sentences, such as "Your truck is sad and sad." Rather than choose words (transitions) at random in each state, you could use known probabilities for how often words follow one another. That would help, but no matter how good your statistics were, the machine could still generate an invalid sentence. Apparently, human brains do something more sophisticated than this simple state machine approach to generate sentences.

State machines generate invalid sentences for the following reasons:[3]

- Grammatical does not imply sensible. For example, "Dogs revert vacuum bags" is grammatically OK but doesn't make any sense. In English, this is self-evident. In a computer program, you also know that a syntactically valid assignment such as `employeeName=milesPerGallon;` might make no sense. The variable types and meaning could be a problem. The meaning of a sentence is referred to as the *semantics*. The next two characteristics are related to syntax.
- There are dependencies between the words of a sentence. When confronted with a], every programmer in the world has an involuntary response to look for the opening [.
- There are order requirements between the words of a sentence. You immediately see "(a[i+3)]" as invalid because you expect the] and) to be in a particular order (I even found it hard to type).

So, walking the states of a state machine is too simple an approach for the generation of complex language. There are word dependencies and order requirements among the output words that it cannot satisfy. Formally, we say that state machines can generate only the class of *regular languages*. As this section points out, programming languages fall into a more complicated, demanding class, the *context-free languages*. The difference between the regular and context-free languages is the difference between a state machine and the more sophisticated machines in the next section. The essential weakness of a state machine is that it has no memory of what it generated in the past. What do we need to remember in order to generate complex language?

3. These are Pinker's reasons from pp. 93–97 in *The Language Instinct* but rephrased in a computer language context.

2.3 The Tree Structure of Sentences

To reveal the memory system necessary to generate complex language, consider how you would write a book. You don't start by typing "the" or whatever the first word of the book is. You start with the concept of a book and then write an outline, which becomes the chapter list. Then you work on the sections within each chapter and finally start writing the sentences of your paragraphs. The phrase that best describes the organization of a book is not "sequence of words." Yes, you can read a book one word at a time, but the book is *structured*: chapters nested within the book, sections nested with the chapters, and paragraphs nested within the sections. Moreover, the substructures are ordered: chapter i must appear before chapter $i+1$. "Nested and ordered" screams tree structure. The components of a book are tree structured with "book" at the root, chapters at the second level, and so on.

Interestingly, even individual sentences are tree structured. To demonstrate this, think about the way you write software. You start with a concept and then work your way down to words, albeit very quickly and unconsciously using a top-down approach. For example, how do you get your fingers to type statement `x=0;` into an editor? Your first thought is not to type `x`. You think "I need to reset `x` to 0" and then decide you need an assignment with `x` on the left and `0` on the right. You finally add the `;` because you know all statements in Java end with `;`. The image in Figure 2.2, on the following page, represents the implicit tree structure of the assignment statement. Such trees are called *derivation trees* when generating sentences and *parse trees* when recognizing sentences. So, instead of directly emitting `x=0;`, your brain does something akin to the following Java code:

```
void statement() {
  assignment();
  System.out.println(";");
}

void assignment() {
  System.out.println("x");
  System.out.println("=");
  expr();
}

void expr() {
  System.out.println("0");
}
```

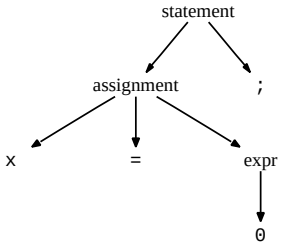


Figure 2.2: “x=0;” assignment statement tree structure

Each method represents a level in the sentence tree structure, and the print statements represent leaf nodes. The leaves are the vocabulary symbols of the sentence.

Each subtree in a sentence tree represents a *phrase* of a sentence. In other words, sentences decompose into phrases, subphrases, subsubphrases, and so on. For example, the statements in a Java method are phrases of the method, which is itself a phrase of the overall class definition sentence.

This section exposed the tree-structured nature of sentences. The next section shows how a simple addition to a state machine creates a much more powerful machine. This more powerful machine is able to generate complex valid sentences and only valid sentences.

2.4 Enforcing Sentence Tree Structure with Pushdown Machines

The method call chain for the code fragment in Section 2.3, *The Tree Structure of Sentences*, on the previous page gives a big hint about the memory system we need to enforce sentence structure. Compare the tree structure in Figure 2.2 with the method call graph in Figure 2.3, on the next page, for this code snippet. The trees match up perfectly. Yep, adding a method call and return mechanism to a state machine turns it into a sophisticated language generator.

It turns out that the humble stack is the perfect memory structure to solve both word dependency and order problems.[4] Adding a stack to a

4. Method call mechanisms use a stack to save and restore return addresses.

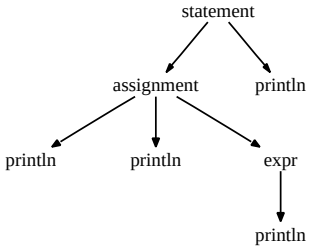


Figure 2.3: Method Call Graph for "x=0;" assignment statement generation

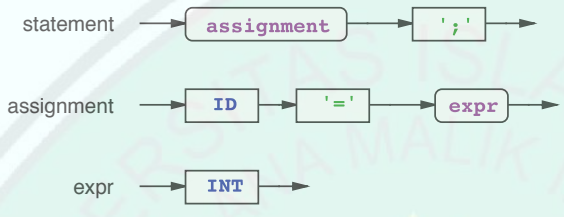


Figure 2.4: Syntax diagram for assignment statement sentence structure

state machine turns it into a *pushdown machine* (pushdown automaton). A state machine is analogous to a stream of instructions trapped within a single method, unable to make method calls. A pushdown machine, on the other hand, is free to invoke other parts of the machine and return just like a method call. The stack allows you to partition a machine into submachines. These submachines map directly to the rules in a grammar.

Representing pushdown machines requires a different visualization called a *syntax diagram*, which looks like a flowchart. There is a flowchart-like submachine per phrase (tree structure subtree). Figure 2.4, illustrates the syntax diagram for the assignment statement sentence structure. The rectangular elements generate vocabulary symbols, and the rounded elements invoke the indicated submachine. Like a method call, the pushdown machine returns from a submachine invocation upon reaching the end of that submachine.

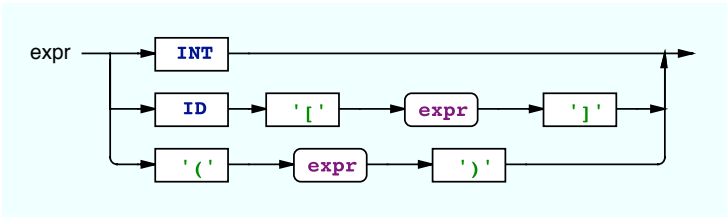


Figure 2.5: Syntax diagram for recursive expression generation

Let's take a look at how a syntax diagram enforces word dependencies and word order. Consider the problem of pairing up square brackets and parentheses with the proper nesting. For example, you must close a bracketed subexpression and do so before the closing outer parenthesized expression. Figure 2.5, shows the syntax diagram for an expression generation pushdown machine. The pushdown machine can generate expressions like 29342, a[12], (89), a[(1)], and (a[a[1]]).

The pushdown machine satisfies bracket symbol dependencies because for every [the machine has no choice but to generate a] later. The same is true for parentheses. But what about enforcing the proper word order for nested expressions?

Look at the second alternative of the machine. The machine must generate the] after the index expression. Any structure that the nested index expression generates must terminate before the]. Similarly, the third alternative guarantees that the) occurs after the structure generated by the enclosed expression. That is, the pushdown machine ensures that grouping symbols are properly nested.

Nested phrases can be *recursive*—they can refer to themselves as the expression syntax diagram does. For example, the pushdown machine generates the nested (1) phrase within a[(1)] using recursion because it invokes itself. Figure 2.6, on the following page shows the derivation tree. The nested **expr** invocations represent recursive submachine invocations. Do not let recursion bother you. Languages are highly recursive by their nature—you cannot generate arbitrarily nested code blocks, for example, without recursion. Besides, as L. Peter Deutsch says, "To iterate is human, to recurse divine."[5]

5. I've also seen another play on the same phrase, "To err is human, to moo bovine."

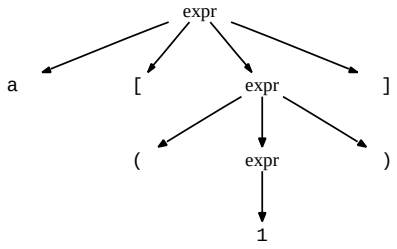


Figure 2.6: Recursive tree structure for expression a[(1)]

This section demonstrated that pushdown machines generate syntactically valid sentences. In other words, each sentence that the pushdown machine generates has a valid interpretation. The next section demonstrates that, unfortunately, some valid sentences have more than one interpretation.

2.5 Ambiguous Languages

As we all know, English and other natural languages can be delightfully *ambiguous*. Any language with an ambiguous sentence is considered ambiguous, and any sentence with more than a single meaning is ambiguous. Sentences are ambiguous if at least one of its phrases is ambiguous. Here is an ambiguous faux newspaper headline: "Bush appeals to democrats." In this case, the verb *appeals* has two meanings: "is attractive to" and "requests help from." This is analogous to operator overloading in computer languages, which makes programs hard to understand just like overloaded words do in English.[6]

Ambiguity is a source of humor in English but the bane of computing. Computers must always know exactly how to interpret every phrase. At the lowest level, computers must always make decisions *deterministically*—they must know exactly which path to take. A classic example of an ambiguous computer phrase relates to arithmetic expressions.

6. A friend put the following dedication into his PhD thesis referring to the advisor he disliked: "To my advisor, for whom no thanks is too much." *The Language Instinct* cites a marvelously ambiguous statement by Groucho Marx: "I once shot an elephant in my pajamas. How he got into my pajamas I'll never know."

The expression 3+4*5 is not ambiguous to an adult human. It means multiply 4 by 5 and add 3, yielding 23. An elementary school student doing the operations from left to right might ask, "Why is the result not 35?" Indeed, why not? Because mathematicians have decreed it so. In fact, German mathematician Leopold Kronecker went so far as to say, "God made the natural numbers; all else is the work of man." So, the language is ambiguous, but syntax and some precedence rules make it unambiguous.

Within a syntax diagram, an ambiguous sentence or phase is one that the diagram can generate following more than one path. For example, a syntax diagram for C can generate statement `i*j;` following the path for both a multiplicative expression and a variable definition (in other words, `j` is a pointer to type `i`). To learn more about the relationship of ambiguous languages to ANTLR grammars, see Section 11.5, *Ambiguities and Nondeterminisms*, on page 273. For example, Section 11.5, *Arithmetic Expression Grammars*, on page 275 has an in-depth discussion of the arithmetic expression ambiguity.

Although syntax is sometimes insufficient to interpret sentences, the informal language definition usually has some extra rules such as precedence that disambiguate the sentences. Chapter 13, *Semantic Predicates*, on page 317 illustrates how to use semantic predicates to enforce these nonsyntactic rules. Semantic predicates are boolean expressions, evaluated at runtime, that guide recognition.

Before turning to the last subject of this chapter, sentence recognition, let's examine the structure of vocabulary symbols themselves. Recognizing sentences is actually a two-level problem: breaking up the input character stream into vocabulary symbols and then applying syntactic structure to the vocabulary symbol sequence.

2.6 Vocabulary Symbols Are Structured Too

Just as sentences consist of phrases, vocabulary symbols have structure. In English, for example, a linguist sees the word *destabilize* as "de.stabil.ize."[7] Similarly, the real number 92.5 is two integers separated by a dot. Humans unconsciously scan sentences with these characters and group them into words.

7. See http://en.wikipedia.org/wiki/Stem_(linguistics).

Sentences are actually sequences of characters, but your brain sees them as sequences of words as if they were complete symbols like Chinese characters. This happens no matter how long the words are. For example, the word for the Hawaiian state fish (Humuhumunukunukuapua'a) is pretty long, but you read it as one symbol (pronouncing it is a lot harder). Your brain somehow implicitly forms words from the characters and looks them up in a dictionary of sorts.

Reading Morse code makes this process even more obvious because it is clear you are combining dots and dashes, representing letters, into words before reading the sentence. For example, what does the following say (using international Morse code)?

```
.-  -.  -  .-..  .-.  ..  ...  -.-.  ---  ---  .-..
```

If you guessed "ANTLR is cool," you'd be right (and are either an obsequious ANTLR fan or a ham radio operator). Here is the character-by-character translation:[8]

```
.-  -.  -  .-..  .-.  ..  ...  -.-.  ---  ---  .-..
A   N T    L    R    I   S     C    O    O    L
```

To mimic the technique your brain uses to recognize sentences, you need to separate the language-level processing from the vocabulary-level processing into two complete recognizers. The language-level recognizer is usually called the *parser*, and the vocabulary recognizer is usually called the *scanner*, *lexical analyzer*, or *lexer*. Lexers create *tokens*[9] (vocabulary symbols) and pass them to the parser. The only difference between the two recognizers is that the parser recognizes grammatical structure in a stream of tokens while the lexer recognizes structure in a stream of characters. Both perform essentially the same task, and ANTLR implements both using the same strategy.

As an example, consider the following simple Java statement:

```
width=200;
```

The lexer scans the individual characters, grouping the letters into an **ID** token type and grouping the digits into an **INT** token type. The punctua-

8. Amazingly, experienced operators can receive and decode Morse code without the help of pen and paper and can generate twenty or thirty words per minute!

9. In practice, tokens consist of at least two pieces of information: the token type (which lexical structure) and the text matched by the lexer. You can put anything you want into the token structure. ANTLR's default CommonToken objects include start/stop indexes into the character buffer, the line number, and the token's character position within that line, among other things.

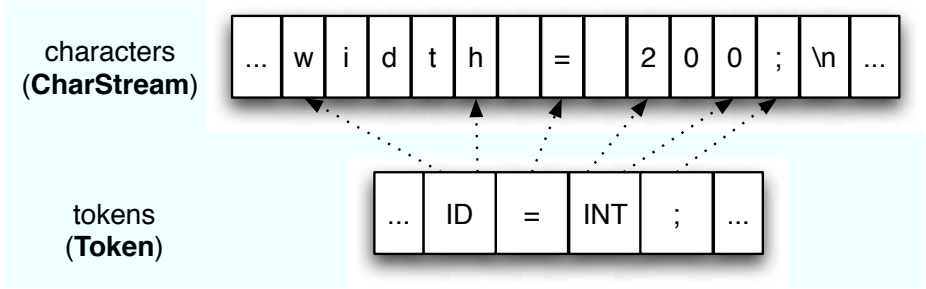


Figure 2.7: Tokens and character buffer. Tokens use character indexes to mark start/stop of tokens in buffer. The lexer does not create tokens for whitespace characters. CharStream and Token are ANTLR runtime types

tion symbols exist as their own token types (that is, no other characters map to that token type). The lexer sees the newline character following the semicolon but throws it out instead of passing it to the parser. The parser therefore sees a sequence of four tokens: **ID**, '=', **INT**, and ';', as illustrated in Figure 2.7. The dotted lines represent the character indexes stored in the tokens that refer to the start and stop character positions (single-character tokens are represented with a single dotted line for clarity).

Separating the parser and lexer might seem like an unnecessary complication if you're used to building recognizers by hand; however, the separation reduces what you have to worry about at each language level. You also get several implementation advantages:

- The parser can treat arbitrarily long character sequences as single tokens. Further, the lexer can group related tokens into token "classes" or *token types* such as **INT** (integers), **ID** (identifiers), **FLOAT** (floating-point numbers), and so on. The lexer groups vocabulary symbols into types when the parser doesn't care about the individual symbols, just the type. For example, the parser doesn't care which integer is approaching on the input stream, just that it is an integer. Also, the parser does not have to wonder whether the next vocabulary symbol is an integer or floating-point number. The lexer can figure this out beforehand and send token type **INT** or **FLOAT** accordingly, allowing the parser to be much simpler.

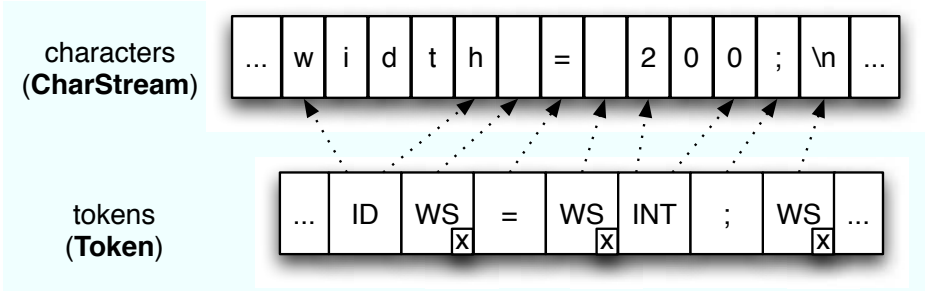


Figure 2.8: Token buffer with hidden tokens. The lexer creates tokens even for whitespace, but puts them on a hidden channel (tokens shown with checkbox)

- The parser sees a pipeline of tokens, which isolates it from the token source. The source of the tokens is irrelevant. For efficiency reasons, you want to save the results of lexing a character stream in a number of situations. For example, interpreters walk the same program statements multiple times during a loop. Once your lexer has tokenized the input, the parser can walk the same token buffer over and over. Some compilers (C and C++ come to mind) can even save tokenized header files to avoid repeatedly tokenizing them.
- The lexer can filter the input, sending only tokens of interest to the parser. This feature makes it easy to handle whitespace, comments, and other lexical structures that you want to discard. For example, if comments were passed to the parser, the parser would have to constantly check for comment tokens and filter them out. Instead, the lexer can simply throw them out, as shown in Figure 2.7, on the previous page, or pass them to the parser on a hidden channel, as shown in Figure 2.8. The tokens on the hidden channel are marked with an "x." Note that the channel is an integer and you can put tokens on any channel you want, but the parser listens to only one channel. For more information about token channels, see Section 4.3, *Lexical Rules*, on page 107 and classes Token, CommonToken, and CommonTokenStream in the org.antlr.runtime package.

At this point, you have the entire language generation picture. Sentences are not ingenious word sequences. Complex language genera-

tion enforces word dependencies and order requirements. Your brain enforces these constraints by subconsciously creating a tree structure. It does not generate sentences by thinking about the first word, the second word, and so on, like a simple state machine. It starts with the overall sentence concept, the root of the tree structure. From there the brain creates phrases and subphrases until it reaches the leaves of the tree structure. From a computer scientist's point of view, generating a sentence is a matter of performing a depth-first tree walk and "saying" the words represented by the leaves. The implicit tree structure conveys the meaning.

Sentence recognition occurs in reverse. Your eyes see a simple list of words, but your brain subconsciously conjures up the implicit tree structure used by the person who generated the sentence. Now you see why language recognition is the dual of language generation. ANTLR builds recognizers that mimic how your brain recognizes sentences. The next section gives you an intuitive feel for what sentence recognition by computer means. Afterward, you will be in good shape for Chapter 4, *ANTLR Grammars*, on page 86.

2.7 Recognizing Computer Language Sentences

Recognizing a sentence means identifying its implicit tree structure, but how do you do that with a program? Many possible solutions exist, but ANTLR generates recognizers with a method for every grammar rule. The methods match symbols and decide which other methods to invoke based upon what they see on the input stream. This is similar to making decisions in the maze based upon the words in the passphrase. The beauty of this implementation is in its simplicity—the method call graph of the recognizer mirrors the parse tree structure. Recall how the call graph in Figure 2.3, on page 41, overlays the tree structure in Figure 2.2, on page 40, perfectly.

To get the most out of Part II of this book, you'll need a basic understanding of language recognition technology and the definition of a few important terms. This section answers three fundamental questions:

- What is the difference between loading a data file into memory and actually recognizing what's in the file?
- From which grammars can ANTLR generate a valid recognizer?
- Can you define the syntax of every language using grammar rules, and if not, what can you use?

The Difference between Reading and Recognizing Input

To characterize what it means for a computer to recognize input, this section compares two small programs. The first program reads a file full of random characters, and the second reads a file of letters and digits. We'll see that the first program doesn't recognize anything, but the second one does. From there, we'll look at the grammar equivalent of the second program and then morph it into something that ANTLR would generate.

To read the file of random characters into memory, we can use the following Java code:

```
BufferedReader f = new BufferedReader(new FileReader("random.txt"));
int c = f.read(); // get the first char
StringBuffer s = new StringBuffer();
while ( c != -1 ) {
  s.append((char)c);
  c = f.read();
}
```

You could say that the file has no structure or that the structure is simply "one or more characters." Now, consider an input file that contains a series of letters followed by a series of digits such as this:

```
acefbetqd392293
```

We can read in this structured input with something like this:

```
BufferedReader f =
  new BufferedReader(new FileReader("lettersAndDigits.txt"));
int c = f.read(); // get the first char
StringBuffer letters = new StringBuffer();
StringBuffer digits = new StringBuffer();
// read the letters
while ( Character.isLetter((char)c) ) {
  letters.append((char)c);
  c = f.read();
}
// read the digits
while ( Character.isDigit((char)c) ) {
  digits.append((char)c);
  c = f.read();
}
```

On the other hand, the previous simple loop would also read in the letters and digits. The difference lies in the fact that the previous loop would not recognize the structure of the input. Recognizing structure involves comparing the input against a series of constraints dictated by the structure.

In this example, recognition implies that the program verifies that the input consists of letters followed by digits. The previous example verifies no constraints. Another way to think about recognition is that, after recognizing some input, the program can provide the groups of characters associated with the various elements of the structure. In English, this is analogous to being able to identify the subject, verb, and object after reading a sentence. In this case, the second program could provide the letters and digits whereas the first program could not. Simply munching up all the input does not identify the input substructures in any way.

In grammar form, the difference between the two recognizer programs stands out more clearly. The first program, which simply consumes all characters, is equivalent to the following grammar:

```
/** Read this as "a file is defined to be one-or-more characters."  The
 *  dot is the wildcard character and the plus means one or more.
 */
file : .+ ; // consume until EOF
```

The second program that matches a bunch of letters followed by digits is equivalent to the following grammar:

```
file: LETTERS DIGITS ;
LETTERS: 'a'..'z'* ; // zero or more lowercase letters
DIGITS : '0'..'9'* ; // zero or more digits
```

Clearly, the first grammar tells us nothing about the input format, whereas the second grammar explicitly defines the input file language structure.

Now, let's look at the kind of code that ANTLR generates. If we break up the stream of instructions into multiple methods, the second program parallels the second grammar:

```
void file() {
    LETTERS();  // go match the letters
    DIGITS();   // go match the digits
}
void LETTERS() {
    while ( Character.isLetter((char)c) ) {
      c = f.read();
    }
}
void DIGITS() {
    while ( Character.isDigit((char)c) ) {
      c = f.read();
    }
}
```

This is not exactly what ANTLR would generate, but it illustrates the type of recognizer ANTLR generates. Once you get used to seeing grammars, you will find them easier to write and understand than unstructured, hand-built recognizers. Can ANTLR always generate a recognizer from a grammar, though? Unfortunately, the answer is no. The next section describes the different kinds of recognizers, discusses the relationship between grammars and recognizers, and illustrates the kinds of recognizers ANTLR can build.

Categorizing Recognizers

Recognizers that begin the recognition process at the most abstract language level are called *top-down* recognizers. The formal term for a top-down recognizer is *LL*.[10] Within the top-down category, the most common implementation is called a *recursive-descent* recognizer. These recognizers have one (possibly recursive) method per rule and are what programmers build by hand. The method call graph traces out the implicit sentence tree structure (the parse tree). This is how top-down recognizers conjure up tree structure without actually building a tree. You can think of top-down recognizers as walking sentence tree structures in a depth-first manner. The root of the tree for the sentence in the previous section is the **file** level. The root has two children: **LETTERS** and **DIGITS**.

Unfortunately, ANTLR cannot generate a top-down recognizer for every grammar—*LL* recognizers restrict the class of acceptable grammars somewhat. For example, ANTLR cannot accept left-recursive grammars such as the following (see Section 11.5, *Left-Recursive Grammars*, on page 274):

```
/** An expression is defined to be an expression followed by '++' */
expr : expr '++'
     ;
```

ANTLR translates this grammar to a recursive method called expr() that immediately invokes itself:

```
void expr() {
    expr();
    match("++");
}
```

10. See http://en.wikipedia.org/wiki/LL_parser. *LL* means "recognize input from left to right using a leftmost derivation." You can interpret "leftmost derivation" as attempting rule references within alternative from left to right. For our purposes, simply consider *LL* a synonym for a top-down recognizer.

Comparing Bottom-Up and Top-Down Recognizers

The other major class of recognizers is called *LR* because these recognizers perform rightmost derivations rather than leftmost (YACC builds *LR*-based recognizers). *LR* recognizers are called *bottom-up* recognizers because they try to match the leaves of the parse tree and then work their way up toward the starting rule at the root. Loosely speaking, *LR* recognizers consume input symbols until they find a matching complete alternative. In contrast, *LL* recognizers are goal-oriented. They start with a rule in mind and then try to match the alternatives. For this reason, *LL* is easier for humans to understand because it mirrors our own innate language recognition mechanism.

Another grammar restriction stems from the fact that the recognition strategy might be too weak to handle a particular grammar. The strength of an *LL*-based recognizer depends on the amount of *lookahead* it has.

Lookahead refers to scanning ahead in the input stream one or more symbols in order to make decisions. In the maze, you match the word under your feet with the next word of lookahead in your passphrase. If you reach a fork where the same word begins both paths, you must use more lookahead to figure out which path to take. Most top-down recognizers use a fixed amount of lookahead, k, and are called *LL(k)* recognizers. *LL(k)* recognizers look up to k words down each path hoping to find a distinguishing word or sequence of words. Here is an example of an *LL(3)* grammar (a grammar for which you can build an *LL(3)* recognizer):

```
/** A decl is 'int' followed by an identifier followed by
 *  an initializer or ';'.
 */
decl : 'int' ID '=' INT ';'  // E.g., "int x = 3;"
     | 'int' ID ';'          // E.g., "int x;"
     ;
```

With less than three lookahead symbols, the recognizer cannot see past the type name and ID token to the assignment operator or the semicolon beyond. Depth k=3 distinguishes the two alternatives. This grammar is not *LL(1)* or *LL(2)*. Although this grammar needs three symbols of lookahead, that does not mean you can't alter the grammar so that it is *LL(1)*.

To be precise, the syntax is *LL(1)*, but that particular grammar for it is *LL(3)*. Refactoring the grammar is more work and yields a less natural-looking grammar:

```
/** LL(1) version of decl; less lookahead, but less natural */
decl : 'int' ID ('=' INT)? ';' // optionally match initializer
     ;
```

Increasing the lookahead depth from 1 to k significantly increases the decision-making power of a recognizer. With more power comes a larger class of grammars, which means you'll find it easier to write grammars acceptable to ANTLR. Still, some natural grammars are not *LL(k)* for even large values of k.

Extending the previous grammar to allow a sequence of modifiers before the type renders it non-*LL(k)* for any fixed k:

```
decl : // E.g., "int x = 3;", "static int x = 3;"
       modifier* 'int' ID '=' INT ';'

     | // E.g., "int x;", "static int x;", "static register int x;",
       // "static static register int x;" (weird but grammar says legal)
       modifier* 'int' ID ';'
     ;
modifier // match a single 'static' or 'register' keyword
     : 'static'
     | 'register'
     ;
```

Because the grammar allows a prefix of zero or more **modifier** symbols, no fixed amount of lookahead will be able to see past the modifiers. One of ANTLR v3's key features is its powerful extension to *LL(k)* called *LL(*)* that allows lookahead to roam arbitrarily far ahead (see Section 11.2, *Why You Need LL(*)*, on page 264). For rule **decl**, ANTLR generates something similar to the following (assume lookahead() is a method that returns the lookahead at the indicated depth):

```
void decl() {
    // PREDICT EITHER ALT 1 or 2
    int alt;
    int k = 1; // start with k=1
    // scan past all the modifiers; LL(k) for fixed k cannot do this!
    while ( lookahead(k) is a modifier ) { k++; }
    k++; // scan past 'int'
    k++; // scan past ID
    if ( lookahead(k) is '=' ) alt = 1; // predict alternative 1
    else alt = 2;                       // else predict alternative 2
```

```
    // MATCH ONE OF THE ALTS
    switch (alt) {
        case 1 :
            // match modifier* 'int' ID '=' INT ';'
            break;
        case 2 :
            // match modifier* 'int' ID ';'
            break;
    }
}
```

LL(*)'s arbitrary lookahead is like bringing a trained monkey along in the maze. The monkey can race ahead of you down the various paths emanating from a fork. It looks for some simple word sequences from your passphrase that distinguish the paths. *LL(*)* represents a significant step forward in recognizer technology because it dramatically increases the number of acceptable grammars without incurring a large runtime speed penalty. Nonetheless, even *LL(*)* is not enough to handle some useful grammars. *LL(*)* cannot see past nested structures because it uses a DFA, not a pushdown machine, to scan ahead. This means it cannot handle some decisions whose alternatives have recursive rule references. In the following grammar, rule **decl** allows C-like declarators instead of simple identifiers:

```
decl : 'int' declarator '=' INT ';' // E.g., "int **x=3;"
     | 'int' declarator ';'         // E.g., "int *x;"
     ;
declarator // E.g., "x", "*x", "**x", "***x"
     : ID
     | '*' declarator
     ;
```

Rule **decl** is not *LL(*)*, but don't worry. ANTLR has an even more powerful strategy that can deal with just about any grammar for a slight reduction in recognition speed.

When ANTLR cannot generate a valid *LL(*)* recognizer from a grammar, you can tell ANTLR to simply try the alternatives in the order specified (see Section 5.3, *backtrack Option*, on page 121). If the first alternative fails, ANTLR rewinds the input stream and tries the second alternative, and so on, until it finds a match. In the maze, this is analogous to trying the alternative paths until you find one that leads to the exit. Such a mechanism is called *backtracking* and is very powerful, but it comes at an exponential speed complexity in the worst case. The speed penalty arises from having to repeatedly evaluate a rule for the same

input position. A decision can have multiple alternatives that begin with the same rule reference, say, **expression**. Backtracking over several of these alternatives means repeatedly invoking rule **expression** at the left edge. Because the input position will always be the same, evaluating **expression** again and again is a waste of time.

Surprisingly, using a technique called *memoization*,[11] ANTLR can completely eliminate such redundant computation at the cost of some memory (see Section 14.5, *Memoization*, on page 343). By recording the result of invoking **expression** while attempting the first alternative, the remaining alternatives can reuse that result. Memoization plus backtracking provides all the power you need with fast recognition speed.

So, ANTLR can usually build a recognizer from a grammar, but can you always define a grammar for a given language? The answer is no. Well, you can't always define one purely with the grammar rules themselves. The next section describes language constructs that are easy to define in English language descriptions, but difficult to express in a straightforward grammar.

Encoding Phrase Context and Precedence

Some languages have phrases that only make sense in the context of other phrases, which is sometimes difficult to encode properly with a grammar. For example, some phrase *x* of sentence *s* might make sense only if preceded (or succeeded) by the phrase *p*; i.e., *s*="...*p*...*x*...". This case abstracts some common programming language recognition problems. Take arithmetic expressions that reference variables. Expressions must have corresponding definitions for those variables, such as *s*="...int i;...i+1...". In English, this is analogous to *x*="open it", which only makes sense in the context of another phrase *p* that defines "it." Is "it" a window or a bottle of German dunkel beer? Here, the meaning of the phrase is clear; you just need to know what to open according to *p*.

Computer recognizers typically record variable definitions such as i in dictionaries called *symbol tables* that map variable names to their type and scope. Later, user-defined actions embedded within the expression rules can look up i in the symbol table to see whether i has been defined.

11. See http://en.wikipedia.org/wiki/Memoization. Bryan Ford applied memoization to parsing, calling it *packrat parsing*.

Some Sentences Force Humans to Backtrack

Sometimes you have to see an entire sentence to get the proper meaning for the initial phrase(s). In some English sentences, you find a word halfway through that is inconsistent with your current understanding. You have to restart from the beginning of the sentence. Trueswell et al in (TTG94) performed experiments that tracked people's eye movements as they read the following sentence:

"The defendant examined by the lawyer turned out to be unreliable."

People paused at *by* because their initial interpretation was that the defendant examined something. The researchers found that people have a strong preference to interpret *examined* as the main verb upon first reading of the sentence. After seeing *by*, people usually backtracked to the beginning of the sentence to begin a new interpretation. Adding commas allows your brain to interpret the sentence properly in one pass:

"The defendant, examined by the lawyer, turned out to be unreliable."

Just like adding commas in an English sentence, adding new symbols to computer language phrases can often reduce the need for backtracking. If you are in control of the language definition, try to make statements as clear as possible. Use extra vocabulary symbols if necessary.

See Section 6.5, *Rule Scopes*, on page 150 for a grammar that looks up variable definitions in a symbol table. For an in-depth discussion, see *Symbol Tables and Scopes*.[12]

What if you can't interpret a phrase at all without examining a previous or future phrase? For example, in C++, expression `T(i)` is either a function call or a constructor-style typecast (as in `(T)i`). If `T` is defined elsewhere as a function, then the expression is a function call. If `T` is defined as a class or other type, then the expression is a typecast. Such phrases are *context-sensitive*. Without the context information, the phrase is ambiguous. The problem is that grammars, as we've defined them, have

12. See http://www.cs.usfca.edu/~parrt/course/652/lectures/symtab.html.

no way to restrict the context in which rules can be applied. The grammars we've examined are called *context-free grammars* (CFGs). Each grammar rule has a name and list of alternatives, and that's it—no context constraints.

To recognize context-sensitive language constructs, ANTLR augments CFGs with *semantic predicates* (see Chapter 13, *Semantic Predicates*, on page 317). Semantic predicates are boolean expressions, evaluated at parse time, that basically turn alternatives on and off. If a semantic predicate evaluates to false, then the associated alternative disappears from the set of viable alternatives. You can use semantic predicates to ask questions about context in order to encode context-sensitivity. In the following rule, both alternatives can match expression `T(i)`, but the predicates turn off the alternative that makes no sense in the current context:

```
expr : {«lookahead(1) is function»}? functionCall
     | {«lookahead(1) is type»}?     ctorTypecast
     ;
```

The predicates are the actions with question marks on the left edge of the alternatives. If `T`, the first symbol of lookahead, is a function, the recognizer will attempt the first alternative; otherwise, it will attempt the second. This easily resolves the ambiguity.

Even with context information, some phrases are still ambiguous. For example, even if you know that `T` is a type, you can interpret statement `T(i);` in two ways: as a constructor-style typecast expression statement and as a variable declaration statement (as in `T i;`). Multiple alternatives within the statement rule are able to match the same input phrase. The C++ language specification resolves the ambiguity by stating that if a statement can be both a declaration and an expression, interpret it as a declaration. This is not an ambiguity arising from lack of context—it is an ambiguity in the language syntax itself. Consequently, semantic predicates won't help in this situation.

To encode the C++ language specification's resolution of this ambiguity, we need a way to encode the precedence of the alternatives. ANTLR provides *syntactic predicates* that order a rule's alternatives, effectively letting us specify their precedence (see Chapter 14, *Syntactic Predicates*, on page 331). Syntactic predicates are grammar fragments enclosed in parentheses followed by `=>`.

In the following rule, the first alternative has a syntactic predicate that means, "If the current statement looks like a declaration, it is." Even if both alternatives can match the same input phrase, the first alternative will always take precedence.

```
stat : (declaration)=> declaration
     | expression
     ;
```

By adding semantic and syntactic predicates to an *LL(*)* grammar, you can usually convince ANTLR to build a valid recognizer for even the nastiest computer language.

This chapter demonstrated that a sentence's implicit tree structure conveys much of its meaning—it's not just the words and word sequence. You learned that your brain generates word sequences using an implicit tree structure and that computers can mimic that behavior with a pushdown machine. Pushdown machines use a stack to recreate sentence tree structure—the submachine invocation trace mirrors the sentence tree structure.

Sentence recognition is the dual of sentence generation. Recognition means identifying the phrases and subphrases within a sentence. To do this, a computer must find the implicit tree structure appropriate for the input sentence. Finding the implicit tree structure is a matter of finding subtrees that match the various phrases.

Rather than writing arbitrary code to recognize implicit sentence tree structure, we'll use formal grammars to define languages. Grammars conform to a domain-specific language that is particularly good at specifying sentence structure. The first chapter of Part II will explain why we use grammars and describes ANTLR's specific grammar notation. Before diving into the reference section of this book, we'll look at a complete example to give your brain something concrete to consider. The next chapter illustrates ANTLR's main components by showing how to build a simple calculator.

Chapter 3

A Quick Tour for the Impatient

The best way to learn about ANTLR is to walk through a simple but useful example. In this chapter, we'll build an arithmetic expression evaluator that supports a few operators and variable assignments. In fact, we'll implement the evaluator in two different ways. First, we'll build a parser grammar to recognize the expression language and then add actions to actually evaluate and print the result. Second, we'll modify the parser grammar to build an intermediate-form tree data structure instead of immediately computing the result. We'll then build a tree grammar to walk those trees, adding actions to evaluate and print the result. When you're through with this chapter, you'll have a good overall view of how to build translators with ANTLR. You'll learn about parser grammars, tokens, actions, ASTs, and tree grammars by example, which will make the ensuing chapters easier to understand.

This chapter is not the ideal starting point for programmers who are completely new to languages and language tools. Those programmers should begin by studying this book's introduction.

To keep it simple, we'll restrict the expression language to support the following constructs:

- Operators plus, minus, and multiply with the usual order of operator evaluation, allowing expressions such as this one:
  ```
  3+4*5-1
  ```
- Parenthesized expressions to alter the order of operator evaluation, allowing expressions such as this one:
  ```
  (3+4)*5
  ```
- Variable assignments and references, allowing expressions such as these:

  Download tour/basic/input

  ```
  a=3
  b=4
  2+a*b
  ```

Here's what we want the translator to do: when it sees 3+4, it should emit 7. When it sees dogs=21, it should map dogs to value 21. If the translator ever sees dogs again, it should pretend we typed 21 instead of dogs. How do we even start to solve this problem? Well, there are two overall tasks:

A parser triggers an embedded action after seeing the element to its left.

1. Build a grammar that describes the overall syntactic structure of expressions and assignments. The result of that effort is a recognizer that answers yes or no as to whether the input was a valid expression or assignment.

2. Embed code among the grammar elements at appropriate positions to evaluate pieces of the expression. For example, given input 3, the translator must execute an action that converts the character to its integer value. For input 3+4, the translator must execute an action that adds the results from two previous action executions, namely, the actions that converted characters 3 and 4 to their integer equivalents.

After completing those two large tasks, we'll have a translator that translates expressions to the usual arithmetic value. In the following sections, we'll walk through those tasks in detail. We'll follow this process:

1. Build the expression grammar.
2. Examine the files generated by ANTLR.
3. Build a test rig and test the recognizer.
4. Add actions to the grammar to evaluate expressions and emit results.
5. Augment the test rig and test the translator.

Now that we've defined the language, let's build an ANTLR grammar that recognizes sentences in that language and computes results.

3.1 Recognizing Language Syntax

We need to build a grammar that completely describes the syntax of our expression language, including the form of identifiers and integers. From the grammar, ANTLR will generate a program that recognizes valid expressions, automatically issuing errors for invalid expressions. Begin by thinking about the overall structure of the input, and then

break that structure down into substructures, and so on, until you reach a structure that you can't break down any further. In this case, the overall input consists of a series of expressions and assignments, which we'll break down into more detail as we proceed.

Ok, let's begin your first ANTLR grammar. The most common ANTLR grammar is a combined grammar that specifies both the parser and lexer rules. These rules specify an expression's grammatical structure as well as its lexical structure (the so-called tokens). For example, an assignment is an identifier, followed by an equals sign, followed by an expression, and terminated with a newline; an identifier is a sequence of letters. Define a combined grammar by naming it using the **grammar** keyword:

```
grammar Expr;
«rules»
```

Put this grammar in file Expr.g because the filename must match the grammar name.

A *program* in this language looks like a series of statements followed by the newline character (newline by itself is an empty statement and ignored). More formally, these English *rules* look like the following when written in ANTLR notation where : starts a rule definition and | separates rule alternatives:

Download tour/basic/Expr.g

```
prog:   stat+ ;

stat:   expr NEWLINE
    |   ID '=' expr NEWLINE
    |   NEWLINE
    ;
```

A grammar rule is a named list of one or more alternatives such as **prog** and **stat**. Read **prog** as follows: a **prog** is a list of **stat** rules. Read rule **stat** as follows: a **stat** is one of the three alternatives:

- An **expr** followed by a newline (token **NEWLINE**)
- The sequence **ID** (an identifier), '=', **expr**, **NEWLINE**
- A **NEWLINE** token

Now we have to define what an expression, rule **expr**, looks like. It turns out that there is a grammar design pattern for arithmetic expressions (see Section 11.5, *Arithmetic Expression Grammars*, on page 275). The pattern prescribes a series of rules, one for each operator precedence

level and one for the lowest level describing expression atoms such as integers. Start with an overall rule called **expr** that represents a complete expression. Rule **expr** will match operators with the weakest precedence, plus and minus, and will refer to a rule that matches subexpressions for operators with the next highest precedence. In this case, that next operator is multiply. We can call the rule **multExpr**. Rules **expr**, **multExpr**, and **atom** look like this:

Download tour/basic/Expr.g

```
expr:   multExpr (('+'|'-') multExpr)*
    ;

multExpr
    :   atom ('*' atom)*
    ;

atom:   INT
    |   ID
    |   '(' expr ')'
    ;
```

Turning to the lexical level, let's define the vocabulary symbols (tokens): identifiers, integers, and the newline character. Any other whitespace is ignored. Lexical rules all begin with an uppercase letter in ANTLR and typically refer to character and string literals, not tokens, as parser rules do. Here are all the lexical rules we'll need:

Download tour/basic/Expr.g

```
ID  :   ('a'..'z'|'A'..'Z')+ ;
INT :   '0'..'9'+ ;
NEWLINE:'\r'? '\n' ;
WS  :   (' '|'\t'|'\n'|'\r')+ {skip();} ;
```

Rule **WS** (whitespace) is the only one with an action (skip();) that tells ANTLR to throw out what it just matched and look for another token.

The easiest way to work with ANTLR grammars is to use ANTLRWorks,[1] which provides a sophisticated development environment (see also Section 1.5, *ANTLRWorks Grammar Development Environment*, on page 30). Figure 3.1, on the following page, shows what grammar **Expr** looks like inside ANTLRWorks. Notice that the syntax diagram view of a rule makes it easy to understand exactly what the rule matches.

At this point, we have no Java code to execute. All we have is an ANTLR grammar. To convert the ANTLR grammar to Java, invoke ANTLR from

1. See http://www.antlr.org/works.

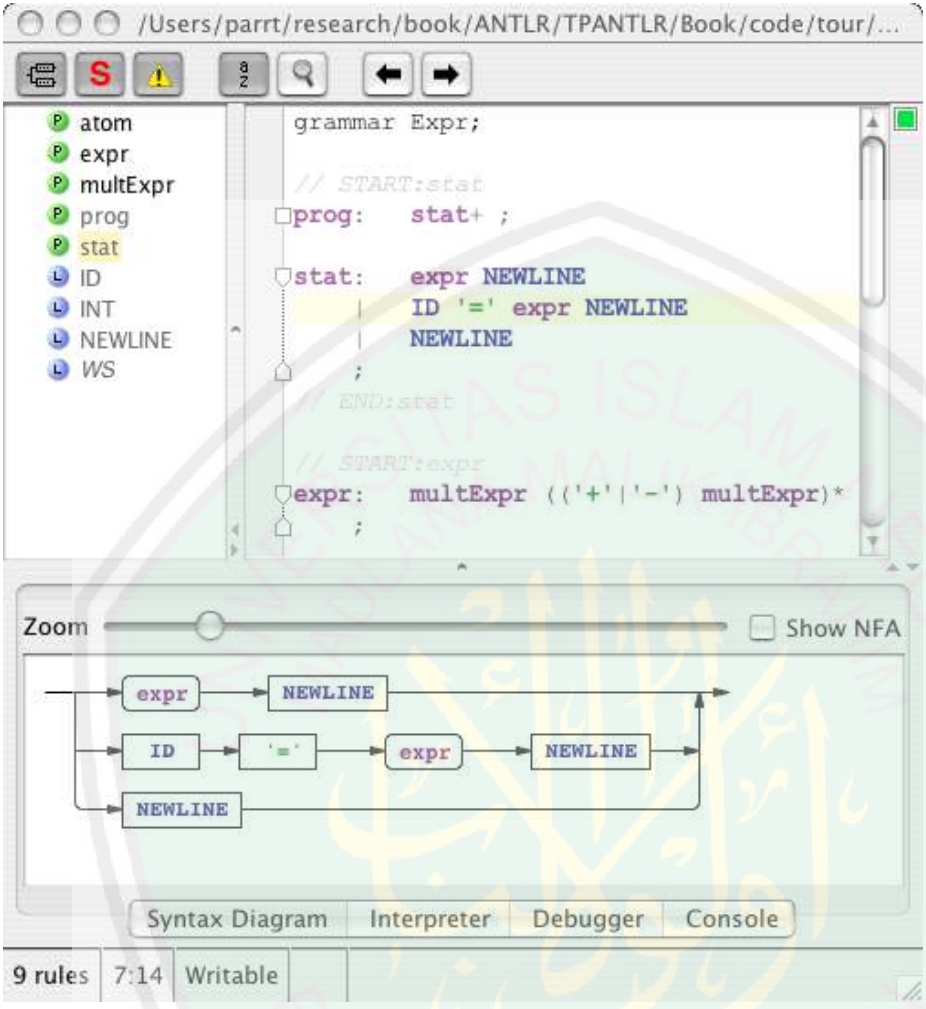


Figure 3.1: ANTLRWorks GUI grammar development tool showing Expr.g and syntax diagram for rule **stat**

the command line (make sure antlr-3.0.jar, antlr-2.7.7, and stringtemplate-3.0.jar are in your CLASSPATH):

```
$ java org.antlr.Tool Expr.g
ANTLR Parser Generator  Version 3.0  1989-2007
$
```

You can also use ANTLRWorks to generate code using the Generate menu's Generate Code option, which will generate code in the same directory as your grammar file. In the next section, we will see what ANTLR generates from the grammar.

What Does ANTLR Generate?

From a combined grammar, ANTLR generates a parser and lexer (written in Java, in this case) that you can compile. Better yet, the code ANTLR generates is human-readable. I strongly suggest you look at the generated code because it will really help demystify ANTLR. ANTLR will generate the following files:

ANTLR generates recognizers that mimic the recursive-descent parsers that you would build by hand; most other parser generators, on the other hand, generate tables full of integers because they simulate state machines.

Generated File	Description
ExprParser.java	The recursive-descent parser generated from the grammar. From grammar **Expr**, ANTLR generates ExprParser and ExprLexer.
Expr.tokens	The list of token-name, token-type assignments such as INT=6.
Expr__.g	The automatically generated **lexer** grammar that ANTLR derived from the combined grammar. The generated file begins with a header: lexer grammar Expr;.
ExprLexer.java	The recursive-descent lexer generated from Expr__.g.

If you look inside ExprParser.java, for example, you will see a method for every rule defined in the grammar. The code for rule **multExpr** looks like the following pseudocode:

```
void multExpr() {
    try {
        atom();
        while ( «next input symbol is *» ) {
            match('*');
            atom();
        }
    }
    catch (RecognitionException re) {
        reportError(re);  // automatic error reporting and recovery
        recover(input,re);
    }
```

The pseudocode for rule **atom** looks like this:

```
void atom() {
    try {
        // predict which alternative will succeed
        // by looking at next (lookahead) symbol: input.LA(1)
        int alt=3;
        switch ( «next input symbol» ) {
            case INT: alt=1; break;
            case ID:  alt=2; break;
            case '(':  alt=3; break;
            default: «throw NoViableAltException»
        }
        // now we know which alt will succeed, jump to it
        switch (alt) {
            case 1 : match(INT); break;
            case 2 : match(ID); break;
            case 3 :
                match('(');
                expr();  // invoke rule expr
                match(')');
                break;
        }
    }

    catch (RecognitionException re) {
        reportError(re);  // automatic error reporting and recovery
        recover(input,re);
    }
}
```

Notice that rule references are translated to method calls, and token references are translated to match(TOKEN) calls.

*The generated code here is general and more complicated than necessary for this simple parser. A future version of ANTLR will optimize these common situations down to simpler code. For example, clearly, the two **switch** statements could be collapsed into a single one.*

All the FOLLOW_multExpr_in_expr160 variable and pushFollow() method references are part of the error recovery strategy, which always wants to know what tokens could come next. In case of a missing or extra token, the recognizer will resynchronize by skipping tokens until it sees a token in the proper "following" set. See Chapter 10, *Error Reporting and Recovery*, on page 241 for more information.

ANTLR generates a file containing the token types just in case another grammar wants to use the same token type definitions, as you will do in Section 3.3, *Evaluating Expressions Encoded in ASTs*, on page 79 when building a tree parser.

Token types are integers that represent the "kind" of token, just like ASCII values represent characters:

Download tour/basic/Expr.tokens

```
INT=6
WS=7
NEWLINE=4
ID=5
'('=12
')'=13
'*'=11
'='=8
'-'=10
'+'=9
```

Testing the Recognizer

Running ANTLR on the grammar just generates the lexer and parser, ExprParser and ExprLexer. To actually try the grammar on some input, we need a test rig with a main() method such as this one:

Download tour/basic/Test.java

```
import org.antlr.runtime.*;

public class Test {
    public static void main(String[] args) throws Exception {
        // Create an input character stream from standard in
        ANTLRInputStream input = new ANTLRInputStream(System.in);
        // Create an ExprLexer that feeds from that stream
        ExprLexer lexer = new ExprLexer(input);
        // Create a stream of tokens fed by the lexer
        CommonTokenStream tokens = new CommonTokenStream(lexer);
        // Create a parser that feeds off the token stream
        ExprParser parser = new ExprParser(tokens);
        // Begin parsing at rule prog
        parser.prog();
    }
}
```

Classes ANTLRInputStream and CommonTokenStream are standard ANTLR classes in the org.antlr.runtime package. ANTLR generated all the other classes instantiated in the test rig.

Once you have compiled the generated code and the test rig, Test.java, run Test, and type a simple expression followed by newline and then the end-of-file character appropriate for your platform:[2]

2. The end-of-file character is Ctrl+D on Unix and Ctrl+Z on Windows.

```
⇐ $ javac Test.java ExprLexer.java ExprParser.java
⇐ $ java Test
⇐ (3+4)*5
⇐ EOF
⇒ $
```

ANTLR doesn't emit any output because there are no actions and grammar. If, however, you type an invalid expression—one that does not follow the grammar—the ANTLR-generated parser will emit an error. The parser will also try to recover and continue matching expressions. The same is true of the generated lexer. For example, upon seeing invalid character @, the lexer reports the following:

```
⇐ $ java Test
⇐ 3+@
⇐ EOF
⇒ line 1:2 no viable alternative at character '@'
  line 1:3 no viable alternative at input '\n'
  $
```

The recognizer found two errors in this case. The first error is a lexical error: an invalid character, @. The second error is a parser error: a missing **atom** (the parser saw 3+\n, not a valid atom such as an integer). A lexer or parser emits the phrase "no viable alternative" when it can't figure out what to do when confronted with a list of alternatives. This means that the next input symbols don't seem to fit any of the alternatives.

For mismatched tokens, recognizers indicate the incorrect token found on the input stream and the expected token:

```
⇐ $ java Test
⇐ (3
⇐ EOF
⇒ line 1:2 mismatched input '\n' expecting ')'
  $
```

The error message is saying that the newline token was unexpected and that the parser expected) instead. The 1:2 error prefix indicates that the error occurred on line 1 and in character position 2 within that line. Because the character position starts from 0, position 2 means the third character.

At this point we have a program that will accept valid input or complain if we give it invalid input. In the next section, you will learn how to add actions to the grammar so you can actually evaluate the expressions.

3.2 Using Syntax to Drive Action Execution

You've made significant progress at this point because you've got a parser and lexer, all without writing any Java code! As you can see, writing a grammar is much easier than writing your own parsing code. You can think of ANTLR's notation as a domain-specific language specifically designed to make recognizers and translators easy to build.

To move from a recognizer to a translator or interpreter, we need to add actions to the grammar, but which actions and where? For our purposes here, we'll need to perform the following actions:

1. Define a hashtable called memory to store a variable-to-value map.
2. Upon expression, print the result of evaluating it.
3. Upon assignment, evaluate the right-side expression, and map the variable on the left side to the result. Store the results in memory.
4. Upon INT, return its integer value as a result.
5. Upon ID, return the value stored in memory for the variable. If the variable has not been defined, emit an error message.
6. Upon parenthesized expression, return the result of the nested expression as a result.
7. Upon multiplication of two atoms, return the multiplication of the two atoms' results.
8. Upon addition of two multiplicative subexpressions, return the addition of the two subexpressions' results.
9. Upon subtraction of two multiplicative subexpressions, return the subtraction of the two subexpressions' results.

We now just have to implement these actions in Java and place them in the grammar according to the location implied by the "Upon. . . " phrase. Begin by defining the memory used to map variables to their values (action 1):

Download tour/eval/Expr.g

```
@header {
import java.util.HashMap;
}

@members {
/** Map variable name to Integer object holding value */
HashMap memory = new HashMap();
}
```

We don't need an action for rule **prog** because it just tells the parser to look for one or more **stat** constructs. Actions 2 and 3 from the previous itemized list need to go in **stat** to print and store expression results:

Download tour/eval/Expr.g

```
prog:   stat+ ;

stat:   // evaluate expr and emit result
        // $expr.value is return attribute 'value' from expr call
        expr NEWLINE {System.out.println($expr.value);}

        // match assignment and stored value
        // $ID.text is text property of token matched for ID reference
    |   ID '=' expr NEWLINE
        {memory.put($ID.text, new Integer($expr.value));}

        // do nothing: empty statement
    |   NEWLINE
    ;
```

For the rules involved in evaluating expressions, it's really convenient to have them return the value of the subexpression they match. So, each rule will match and evaluate a piece of the expression, returning the result as a method return value. Look at **atom**, the simplest subexpression first:

Download tour/eval/Expr.g

```
atom returns [int value]
    :   // value of an INT is the int computed from char sequence
        INT {$value = Integer.parseInt($INT.text);}

    |   ID // variable reference
        {
        // look up value of variable
        Integer v = (Integer)memory.get($ID.text);
        // if found, set return value else error
        if ( v!=null ) $value = v.intValue();
        else System.err.println("undefined variable "+$ID.text);
        }

        // value of parenthesized expression is just the expr value
    |   '(' expr ')' {$value = $expr.value;}
    ;
```

Per the fourth action from the itemized list earlier, the result of an **INT atom** is just the integer value of the **INT** token's text. An **INT** token with text 91 results in the value 91. Action 5 tells us to look up the **ID** token's text in the memory map to see whether it has a value.

If so, just return the integer value stored in the map, or else print an error. Rule **atom**'s third alternative recursively invokes rule **expr**. There is nothing to compute, so the result of this **atom** evaluation is just the result of calling expr(); this satisfies action 6. As you can see, $value is the result variable as defined in the **returns** clause; $expr.value is the result computed by a call to **expr**. Moving on to multiplicative subexpressions, here is rule **multExpr**:

Download tour/eval/Expr.g

```
/** return the value of an atom or, if '*' present, return
 *  multiplication of results from both atom references.
 *  $value is the return value of this method, $e.value
 *  is the return value of the rule labeled with e.
 */
multExpr returns [int value]
    :   e=atom {$value = $e.value;} ('*' e=atom {$value *= $e.value;})*
    ;
```

Rule **multExpr** matches an **atom** optionally followed by a sequence of * operators and **atom** operands. If there is no * operator following the first atom, then the result of **multExpr** is just the **atom**'s result. For any multiplications that follow the first atom, all we have to do is keep updating the **multExpr** result, $value, per action 7. Every time we see a * and an **atom**, we multiply the **multExpr** result by the **atom** result.

The actions in rule **expr**, the outermost expression rule, mirror the actions in **multExpr** except that we are adding and subtracting instead of multiplying:

Download tour/eval/Expr.g

```
/** return value of multExpr or, if '+'|'-' present, return
 *  multiplication of results from both multExpr references.
 */
expr returns [int value]
    :   e=multExpr {$value = $e.value;}
        (   '+' e=multExpr {$value += $e.value;}
        |   '-' e=multExpr {$value -= $e.value;}
        )*
    ;
```

These actions satisfy the last actions, 8 and 9, from the itemized list earlier in this section. One of the big lessons to learn here is that syntax drives the evaluation of actions in the parser. The structure of an input sequence indicates what kind of thing it is. Therefore, to execute actions only for a particular construct, all we have to do is place actions in the grammar alternative that matches that construct.

What does ANTLR do with grammar actions? ANTLR simply inserts actions right after it generates code for the preceding element—parsers must execute embedded actions after matching the preceding grammar element. ANTLR spits them out verbatim except for the special attribute and template reference translations (see Section 6.6, *References to Attributes within Actions*, on page 159 and Section 9.9, *References to Template Expressions within Actions*, on page 238).

ANTLR's handling of actions is straightforward. For example, ANTLR translates rule return specifications such as this:

```
multExpr returns [int value]
  : ...
  ;
```

to the following Java code:

```
public int expr() throws RecognitionException {
    int value = 0; // rule return value, $value
    ...
    return value;
}
```

ANTLR translates labels on rule references, such as e=multExpr, to method call assignments, such as e=multExpr(). References to rule return values, such as $e.value, become e when there is only one return value and e.value when there are multiple return values.

Take a look at the pseudocode for rule **expr**. The highlighted lines in the output derive from embedded actions:

```
public int expr() {
►   int value = 0; // our return value, automatically initialized
    int e = 0;
    try {
        e=multExpr();
►       value = e; // if no + or -, set value to result of multExpr
        // Expr.g:27:9: ( '+' e= multExpr | '-' e= multExpr )*
loop3:
        while ( true ) {
            int alt=3;
            if ( «next input symbol is +» ) { alt=1; }
            else if ( «next input symbol is -» ) { alt=2; }
            switch ( alt ) {
            case 1 :
                match('+');
                e=multExpr();
►               value += e; // add in result of multExpr
                break;
```

```
                case 2 :
                    match('-');
                    e=multExpr();
►                   value -= e; // subtract out result of multExpr
                    break;
                default :
                    break loop3;
                }
            }
        }
        catch (RecognitionException re) {
            reportError(re);
            recover(input,re);
        }
        return value;
    }
```

OK, we're ready to test the grammar. We have added actions only to the grammar, so the main() program in Test can stay the same. It still just invokes the **prog** start rule. Note that if we change the grammar, we have to recompile the generated files, such as ExprParser.java and ExprLexer.java. Running expressions into the program now returns the expected computations:

```
⇐  $ java Test
⇐  3+4*5
⇐  EOF
⇒  23
⇐  $ java Test
⇐  (3+4)*5
⇐  EOF
⇒  35
   $
```

Variable assignments will store expression results, and then we can pull the results back later by referencing the variable, as shown in the following input file:

Download tour/eval/input

```
a=3
b=4
2+a*b
```

Running those expressions into the test rig with IO redirection gives the proper result:

```
$ java Test < input
14
$
```

Upon invalid input, ANTLR reports an error and attempts to recover. Recovering from an error means resynchronizing the parser and pretending that nothing happened. This means the parser should still execute actions after recovering from an error. For example, if you type 3++4, the parser fails at the second + because it can't match it against **atom**:

```
⇐  $ java Test
⇐  3++4
⇐  E0F
⇒  line 1:2 no viable alternative at input '+'
   7
   $
```

The parser recovers by throwing out tokens until it sees a valid **atom**, which is 4 in this case. It can also recover from missing symbols by pretending to insert them. For example, if you leave off a right parenthesis, the parser reports an error but then continues as if you had typed the):

```
⇐  $ java Test
⇐  (3
⇐  E0F
⇒  line 1:2 mismatched input '\n' expecting ')'
   3
   $
```

This concludes your first complete ANTLR example. The program uses a grammar to match expressions and uses embedded actions to evaluate expressions. Now let's look at a more sophisticated solution to the same problem. The solution is more complicated but worth the trouble because it demonstrates how to build a tree data structure and walk it with another grammar. This multipass strategy is useful for complicated translations because they are much easier to handle if you break them down into multiple, simple pieces.

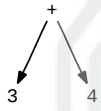
3.3 Evaluating Expressions Using an AST Intermediate Form

Now that you've seen how to build a grammar and add actions to implement a translation, this section will guide you through building the same functionality but using an extra step involving trees. We'll use that same parser grammar to build an intermediate data structure, replacing the embedded actions with tree construction rules. Once we have that tree, we'll use a tree parser to walk the tree and execute embedded actions.

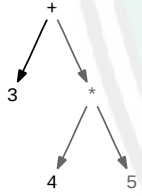
ANTLR will generate a tree parser from a tree grammar automatically for us. The parser grammar converts a token stream into a tree that the tree grammar parses and evaluates.

Although the previous section's approach was more straightforward, it does not scale well to a full programming language. Adding constructs such as function calls or **while** loops to the language means that the interpreter must execute the same bits of code multiple times. Every time the input program invoked a method, the interpreter would have to reparse that method. That approach works but is not as flexible as building an intermediate representation such as an abstract syntax tree (AST) and then walking that data structure to interpret the expressions and assignments. Repeatedly walking an intermediate-form tree is much faster than reparsing an input program. See Section 1.1, *The Big Picture*, on page 22 for more about ASTs.

An intermediate representation is usually a tree of some flavor and records not only the input symbols but also the relationship between those symbols as dictated by the grammatical structure. For example, the following AST represents expression 3+4:



In many cases, you'll see trees represented in text form. For example, the text representation of 3+4 is (+ 3 4). The first symbol after the (is the root and the subsequent symbols or its children. The AST for expression 3+4*5 has the text form (+ 3 (* 4 5)) and looks like this:



As you can see, the structure of the tree implicitly encodes the precedence of the operators. Here, the multiplication must be done first because the addition operation needs the multiplication result as its right operand.

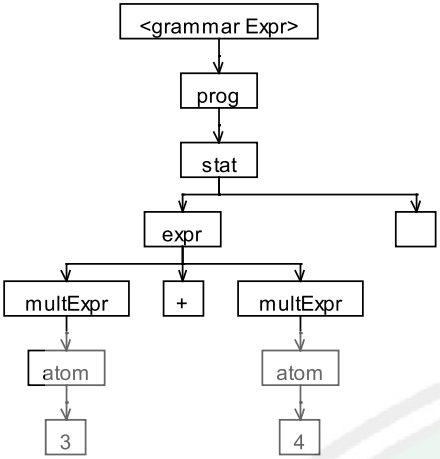


Figure 3.2: Parse tree for 3+4

An AST is to be distinguished from a *parse tree*, which represents the sequence of rule invocations used to match an input stream. Figure 3.2 shows the parse tree for 3+4 (created by ANTLRWorks).

The leaves of a parse tree are input symbols, and the nonleaves are rule names (the very top node, <grammarExpr>, is something ANTLRWorks adds to show you what grammar the parse tree comes from). The top rule node, **prog**, indicates that 3+4 is a **prog** overall. More specifically, it is a **stat**, which in turn is an **expr** followed by a newline, and so on. So the parse tree records how the recognizer navigates the rules of the grammar to match the input. Compare this to the much smaller and simpler AST for 3+4 where all nodes in the AST are input symbol nodes. The structure of the tree encodes the meaning that '+' is an operator with two children. This is much easier to see without all of the "noise" introduced by the grammar rule nodes.

In practice, it is also useful to decouple the grammar from the trees that it yields; hence, ASTs are superior to parse trees. A tweak in a grammar usually alters the structure of a parse tree while leaving an AST unaffected, which can make a big difference to the code that walks your trees.

Once you have the tree, you can walk it in multiple ways in order to evaluate the tree that the expression represents. In general, I recommend using a grammar to describe the tree structure just as you use a parser grammar to describe a one-dimensional input language. From the tree grammar, ANTLR can generate a tree walker using the same top-down recursive-descent parsing strategy used for lexers and parsers.

In the following sections, you'll learn how to build ASTs, how to walk them with a tree grammar, and how to embed actions within a tree grammar to emit a translation. At the end, you'll have a translator that is functionally equivalent to the previous one.

Building ASTs with a Grammar

Building ASTs with ANTLR is straightforward. Just add AST construction rules to the parser grammar that indicate what tree shape you want to build. This declarative approach is much smaller and faster to read and write than the informal alternative of using arbitrary embedded actions. When you use the output=AST option, each of the grammar rules will implicitly return a node or subtree. The tree you get from invoking the starting rule is the complete AST.

Let's take the raw parser grammar without actions from Section 3.1, *Recognizing Language Syntax*, on page 60 and augment it to build a suitable AST. As you we along, we'll discuss the appropriate AST structure. We begin by telling ANTLR to build a tree node for every token matched on the input stream:

```
grammar Expr;
options {
    output=AST;
    // ANTLR can handle literally any tree node type.
    // For convenience, specify the Java type
    ASTLabelType=CommonTree; // type of $stat.tree ref etc...
}
...
```

For each token the recognizer matches, it will create a single AST node. Given no instructions to the contrary, the generated recognizer will build a flat tree (a linked list) of those nodes. To specify a tree structure, simply indicate which tokens should be considered operators (subtree roots) and which tokens should be excluded from the tree. Use the ^ and ! token reference suffixes, respectively.

Starting with the raw parser grammar without actions, modify the expression rules as follows:

Download tour/trees/Expr.g

```
expr:   multExpr (('+'^|'-'^) multExpr)*
    ;

multExpr
    :   atom ('*'^ atom)*
    ;

atom:   INT
    |   ID
    |   '('! expr ')'!
    ;
```

We only need to add AST operators to tokens +, -, (, and). The ! operator suffix on the parentheses tells ANTLR to avoid building nodes for those tokens. Parentheses alter the normal operator precedence by changing the order of rule method calls. The structure of the generated tree, therefore, encodes the arithmetic operator precedence, so we don't need parentheses in the tree.

For the **prog** and **stat** rule, let's use tree rewrite syntax (see Section 4.3, *Rewrite Rules*, on page 103) because it is clearer. For each alternative, add a -> AST construction rule as follows:

Download tour/trees/Expr.g

```
/** Match a series of stat rules and, for each one, print out
 *  the tree stat returns, $stat.tree.  toStringTree() prints
 *  the tree out in form: (root child1 ... childN)
 *  ANTLR's default tree construction mechanism will build a list
 *  (flat tree) of the stat result trees.  This tree will be the input
 *  to the tree parser.
 */
prog:   ( stat {System.out.println($stat.tree.toStringTree());} )+ ;

stat:   expr NEWLINE         -> expr
    |   ID '=' expr NEWLINE -> ^('=' ID expr)
    |   NEWLINE             ->
    ;
```

The grammar elements to the right of the -> operator are tree grammar fragments that indicate the structure of the tree you want to build. The first element within a ^(...) tree specification is the root of the tree. The remaining elements are children of that root. You can think of the rewrite rules as grammar-to-grammar transformations. We'll see in a moment that those exact tree construction rules become alternatives in

the tree grammar. The **prog** rule just prints the trees and, further, does not need any explicit tree construction. The default tree construction behavior for **prog** builds what you want: a list of statement trees to parse with the tree grammar.

The rewrite rule for the first alternative says that **stat**'s return value is the tree returned from calling **expr**. The second alternative's rewrite says to build a tree with '=' at the root and **ID** as the first child. The tree returned from calling **expr** is the second child. The empty rewrite for the third alternative simply means don't create a tree at all.

The lexical rules and the main program in Test do not need any changes. Let's see what the translator does with the previous file, input:

Download tour/trees/input

```
a=3
b=4
2+a*b
```

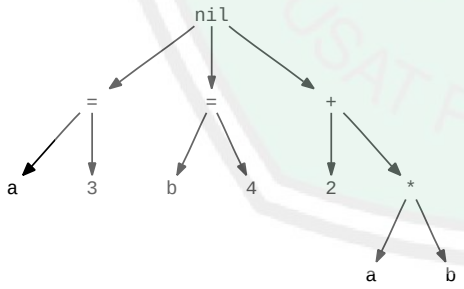
First ask ANTLR to translate Expr.g to Java code and compile as you did for the previous solution:

```
$ java org.antlr.Tool Expr.g
ANTLR Parser Generator  Version 3.0  1989-2007
$ javac Test.java ExprParser.java ExprLexer.java
$
```

Now, redirect file input into the test rig, and you will see three trees printed, one for each input assignment or expression. The test rig prints the tree returned from start rule **prog** in text form:

```
$ java Test < input
(= a 3)
(= b 4)
(+ 2 (* a b))
$
```

The complete AST built by the parser (and returned from **prog**) looks like the following in memory:



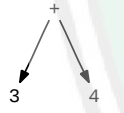
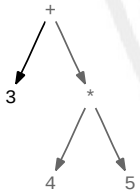
The nil node represents a list of subtrees. The children of the nil node track the elements in a list.

Now we have a parser that builds appropriate ASTs, and we need a way to walk those trees to evaluate the expressions they represent. The next section shows you how to build a tree grammar that describes the AST structure and how to embed actions, like you did in Section 3.2, *Using Syntax to Drive Action Execution*, on page 68.

Evaluating Expressions Encoded in ASTs

In this section, we'll write a tree grammar to describe the structure of the ASTs we built using a parser grammar in the previous section. Then, we'll add actions to compute subexpression results. Like the previous solution, each expression rule will return these partial results. From this augmented grammar, ANTLR will build a tree parser that executes your embedded actions.

Parsing a tree is a matter of walking it and verifying that it has not only the proper nodes but also the proper two-dimensional structure. Since it is harder to build parsers that directly recognize tree structures, ANTLR uses a one-dimensional stream of tree nodes computed by iterating over the nodes in a tree via a depth-first walk. ANTLR inserts special imaginary **UP** and **DOWN** nodes to indicate when the original tree dropped down to a child list or finished walking a child list. In this manner, ANTLR reduces tree parsing to conventional one-dimensional token stream parsing. For example, the following table summarizes how ANTLR serializes two sample input trees.

AST	Node Stream
	+ DOWN 3 4 UP
	+ DOWN 3 * DOWN 4 5 UP UP

The ANTLR notation for a tree grammar is identical to the notation for a regular grammar except for the introduction of a two-dimensional tree construct. The beauty of this is that we can make a tree grammar by cutting and pasting from the parser grammar. We just have to remove

recognition grammar elements to the left of the -> operator, leaving the AST rewrite fragments. These fragments build ASTs in the parser grammar and recognize that structure in the tree grammar.

Let's build your first tree grammar in a separate file, which we'll call Eval.g. Tree grammars begin very much like parser grammars with a grammar header and some options:

```
tree grammar Eval; // yields Eval.java

options {
    tokenVocab=Expr; // read token types from Expr.tokens file
    ASTLabelType=CommonTree; // what is Java type of nodes?
}
...
```

The **tokenVocab** option indicates that the tree grammar should preload the token names and associated token types defined in Expr.tokens. (ANTLR generates that file after processing Expr.g.) When we say **ID** in the tree grammar, we want the resulting recognizer to use the same token type that the parser used. **ID** in the tree parser must match the same token type it did in the parser.

Before writing the rules, define a memory hashtable to store variable values, like we did for the parser grammar solution:

Download tour/trees/Eval.g

```
@header {
import java.util.HashMap;
}

@members {
/** Map variable name to Integer object holding value */
HashMap memory = new HashMap();
}
```

As you'll learn in Section 7.1, *Proper AST Structure*, on page 163, ASTs should be simplified and normalized versions of the token stream that implicitly encode grammatical structure. Consequently, tree grammars are usually much simpler than the associated parser grammars that build their trees. In fact, in this case, all the expression rules from the parser grammar collapse to a single **expr** rule in the tree grammar (see Section 8.3, *Building a Tree Grammar for the C- Language*, on page 199 for more about the expression grammar design pattern). This parser grammar normalizes expression trees to have an operator at the root and its two operands as children. We need an **expr** rule that reflects this structure:

Download tour/trees/Eval.g

```
expr returns [int value]
    :   ^('+' a=expr b=expr) {$value = a+b;}
    |   ^('-' a=expr b=expr) {$value = a-b;}
    |   ^('*' a=expr b=expr) {$value = a*b;}
    |   ID
        {
        Integer v = (Integer)memory.get($ID.text);
        if ( v!=null ) $value = v.intValue();
        else System.err.println("undefined variable "+$ID.text);
        }
    |   INT                  {$value = Integer.parseInt($INT.text);}
    ;
```

Rule **expr** indicates that an expression tree is either a simple node created from an **ID** or an **INT** token or that an expression tree is an operator subtree. With the simplification of the grammar comes a simplification of the associated actions. The **expr** rule normalizes all computations to be of the form "result = a <operator> b." The actions for **ID** and **INT** nodes are identical to the actions we used in the parser grammar's **atom** rule.

The actions for rules **prog** and **stat** are identical to the previous solution. Rule **prog** doesn't have an action—it just matches a sequence of expression or assignment trees. Rule **stat** does one of two things:

1. It matches an expression and prints the result.
2. It matches an assignment and maps the result to the indicated variable.

Here is how to say that in ANTLR notation:

Download tour/trees/Eval.g

```
prog:   stat+ ;

stat:   expr
        {System.out.println($expr.value);}
    |   ^('=' ID expr)
        {memory.put($ID.text, new Integer($expr.value));}
    ;
```

Rule **stat** does not have a third alternative to match the **NEWLINE** (empty) expression like the previous solution. The parser strips out empty expressions by not building trees for them.

What about lexical rules? It turns out you don't need any because tree grammars feed off a stream of tree nodes, not tokens.

At this point, we have a parser grammar that builds an AST and a tree grammar that recognizes the tree structure, executing actions to evaluate expressions. Before we can test the grammars, we have to ask ANTLR to translate the Eval.g grammar to Java code. Execute the following command line:

```
$ java org.antlr.Tool Eval.g
ANTLR Parser Generator  Version 3.0  1989-2007
$
```

This results in the following two files:

Generated File	**Description**
Eval.java	The recursive-descent tree parser generated from the grammar.
Eval.tokens	The list of token-name, token-type assignments such as INT=6. Nobody will be using this file in this case, but ANTLR always generates a token vocabulary file.

We now need to modify the test rig so that it walks the tree built by the parser. At this point, all it does is launch the parser, so we must add code to extract the result tree from the parser, create a tree walker of type Eval, and start walking the tree with rule **prog**. Here is the complete test rig that does everything we need:

Download tour/trees/Test.java

```
import org.antlr.runtime.*;
import org.antlr.runtime.tree.*;

public class Test {
    public static void main(String[] args) throws Exception {
        // Create an input character stream from standard in
        ANTLRInputStream input = new ANTLRInputStream(System.in);
        // Create an ExprLexer that feeds from that stream
        ExprLexer lexer = new ExprLexer(input);
        // Create a stream of tokens fed by the lexer
        CommonTokenStream tokens = new CommonTokenStream(lexer);
        // Create a parser that feeds off the token stream
        ExprParser parser = new ExprParser(tokens);
        // Begin parsing at rule prog, get return value structure
        ExprParser.prog_return r = parser.prog();

        // WALK RESULTING TREE
        CommonTree t = (CommonTree)r.getTree(); // get tree from parser
        // Create a tree node stream from resulting tree
        CommonTreeNodeStream nodes = new CommonTreeNodeStream(t);
        Eval walker = new Eval(nodes); // create a tree parser
        walker.prog();                 // launch at start rule prog
    }
}
```

The test rig extracts the AST from the parser by getting it from the return value object **prog** returns. This object is of type prog_return, which ANTLR generates within ExprParser:

```
// from the parser that builds AST for the tree grammar
public static class prog_return extends ParserRuleReturnScope {
    CommonTree tree;
    public Object getTree() { return tree; }
};
```

In this case, rule **prog** does not have a user-defined return value, so the constructed tree is the sole return value.

We have a complete expression evaluator now, so we can try it. Enter some expressions via standard input:

```
⇐ $ java Test
⇐ 3+4
⇐ EOF
⇒ (+ 3 4)
  7
⇐ $ java Test
⇐ 3*(4+5)*10
⇐ EOF
⇒ (* (* 3 (+ 4 5)) 10)
  270
  $
```

You can also redirect file input into the test rig:

```
$ java Test < input
(= a 3)
(= b 4)
(+ 2 (* a b))
14
$
```

The output of the test rig first shows the tree structure (in serialized form) for each input statement. Serialized form (= a 3) represents the tree built for a=3;. The tree has = at the root and two children: a and 3. The rig then emits the expression value computed by the tree parser.

In this chapter, we built two equivalent expression evaluators. The first implementation evaluated expressions directly in a parser grammar, which works great for simpler translations and is the fastest way to build a translator. The second implementation separated parsing and evaluation into two phases. The first phase parsed as before but built ASTs instead of evaluating expressions immediately. The second phase walked the resulting trees to do the evaluation. You will need this second approach when building complicated translators that are easier to

In fact, some language tasks, such as programming language interpreters, need to repeatedly walk the input by their very nature. Building a simple and concise intermediate form is much faster than repeatedly parsing the token stream.

At this point, you have an overall sense of how to work with ANTLR and how to write simple grammars to recognize and translate input sentences. But, you have a lot more to learn, as you will see in the next few chapters. In particular, the first two chapters of Part II are important because they explain more about ANTLR grammars, embedded actions, and attributes. If you'd like to jump deeper into ANTLR before reading more of this reference guide, please see the tree-based interpreter tutorial on the wiki, which extends this expression evaluator to support function definitions and function calls.[3]

3. See http://www.antlr.org/wiki/display/ANTLR3/Simple+tree-based+interpeter.

Part II

ANTLR Reference

Chapter 4

ANTLR Grammars

ANTLR's highest-level construct is a grammar, which is essentially a list of rules describing the structure of a particular language. From this list of rules, ANTLR generates a recursive-descent parser that recognizes sentences in that language (recursive-descent parsers are the kind of parsers you write by hand). The language might be a programming language or simply a data format, but ANTLR does not intuitively know anything about the language described by the grammar. ANTLR is merely building a recognizer that can answer this question: is an input sequence a valid sentence according to the grammar? In other words, does the input follow the rules of the language described by the grammar? You must embed code within the grammar to extract information, perform a translation, or interpret the incoming symbols in some other way according to the intended application.

This chapter defines the syntax of ANTLR grammars and begins with an introduction to the specification of languages with formal grammars. This chapter covers the following:

- Overall ANTLR file structure
- Grammar lexical elements such as comments, rule names, token names, and so on
- Individual rule syntax including parameters and return values
- Rule elements, actions, alternatives, and EBNF subrules
- Rewrite rule syntax
- Lexical and tree matching rules
- Dynamically scoped attribute syntax
- Grammar-level actions

You'll need to learn the information in this chapter well because building grammar rules and embedding actions will be your core activities. Readers familiar with v2 can scan through looking for margin notes. These notes identify new, improved, and modified features for v3. The wiki also has a good migration page.[1] If you're unfamiliar with ANTLR altogether, you should do an initial quick scan of this chapter and the next to familiarize yourself with the general pieces. Then try modifying a few existing v3 grammar examples.[2]

4.1 Describing Languages with Formal Grammars

Before diving into the actual syntax of ANTLR, let's discuss the general idea of describing languages with grammars and define some common terms. We'll start by explicitly stating the behavior of a translator.

A translator is a program that reads some input and emits some output. By *input*, we mean a sequence of *vocabulary symbols*, not random characters. The vocabulary symbols are analogous to the words in the English language. An input sequence is formally called a *sentence*. Technically, each sentence represents a complete input sequence implicitly followed by the end-of-file symbol. For example, a complete Java class definition file is a sentence, as is a data file full of comma-separated values. A *language* then is simply a well-defined set of sentences. A *translator* is a program that maps each input sentence, *s*, in its input language to a specific output sentence, *t*.[3]

Translating complete sentences such as Java class definitions in a single step is generally impractical. Translators decompose sentences into multiple subsequences, or *phrases*, that are easier to translate. A phrase, *x*, exists somewhere in sentence *s*, *s*="...*x*..." (at the outermost level, *s*="*x*"). Translators further decompose phrases into subphrases, subsubphrases, and so on. For example, the statements in a Java method are phrases of the method, which is itself a phrase of the overall class definition sentence. Breaking sentences down into phrases and subphrases is analogous to breaking large methods into smaller, more manageable methods.

1. See http://www.antlr.org/wiki/display/ANTLR3/Migrating+from+ANTLR+2+to+ANTLR+3.
2. See http://www.antlr.org/v3.
3. This translator definition covers compilers, interpreters, and in some sense almost any program that generates output.

To map each phrase *x* to some output phrase *y*, a translator computes *y* as a function of *x*. The mapping function can be anything from a simple lookup to an involved computation. A lookup might map input symbol *x*="int" to *y*="integer" and a computation might map *x*="i+1-1" to *y*="i". In order to execute the correct mapping function, the translator needs to identify *x*. Identifying *x* within a sentence means *recognizing* it or distinguishing it from the other phrases within a linear sequence of symbols.

To recognize phrase *x*, a translator needs to know what *x* looks like. The best way to tell a computer what phrases and sentences look like is to use a formal, text-based description called a *grammar*. Grammars conform to a DSL that was specifically designed for describing other languages. Such a DSL is called a *metalanguage*. English is just too loose to describe other languages, and besides, computers can't yet grok English prose.

A grammar describe the syntax of a language. We say that a grammar "generates a language" because we use grammars to describe what languages look like. In practice, however, the goal of formally describing a language is to obtain a program that recognizes sentences in the language. ANTLR converts grammars to such recognizers.

A grammar is a set of rules where each rule describes some phrase (subsentence) of the language. The rule where parsing begins is called the *start rule* (in ANTLR grammars, any and all rules can be the starting rule). Each rule consists of one or more *alternatives*.

For example, a rule called **variableDef** might have two alternatives, one for a simple definition and another for a definition with an initialization expression. Often the rules in a grammar correspond to abstract language phrases such as **statement** and **expression**. There will also be a number of finer-grained helper rules, such as **multiplicativeExpression** and **declarator**. Rules reference other rules as well as tokens to match the phrase and subphrase structure of the various sentences.

The most common grammar notation is called *Backus-Naur Form* (BNF). ANTLR uses a grammar dialect derived from YACC [Joh79] where rules begin with a lowercase letter and token types begin with an uppercase letter.

Here is a sample rule with two alternatives:

```
/** Match either a simple declaration followed by ';' or match
 *  a declaration followed by an initialization expression.
 *  Rules: variableDef, declaration, expr.
 *  Tokens: SEMICOLON, EQUALS
 */
variableDef
    :   declaration SEMICOLON
    |   declaration EQUALS expr SEMICOLON
    ;
```

BNF notation is cumbersome for specifying things like repeated elements because you must use recursion. ANTLR supports *Extended BNF* (EBNF) notation that allows optional and repeated elements. EBNF also supports parenthesized groups of grammar elements called *subrules*. See Section 4.3, *Extended BNF Subrules*, on page 98 and, in particular, Figure 4.3, on page 99.

EBNF grammars are called *context-free grammars* (CFGs). They are called context-free because we can't restrict rules to certain contexts. For example, we can't constrain an expression rule to situations where the recognizer has matched or will match another rule. Such a grammar is called *context-sensitive grammar*, but no one uses them in practice because there is no efficient context-sensitive recognition algorithm. Instead, we'll use semantic and syntactic predicates to achieve the same effect (see Chapter 13, *Semantic Predicates*, on page 317 and Chapter 14, *Syntactic Predicates*, on page 331).

The remainder of this chapter describes ANTLR's EBNF grammar syntax using the concepts and terminology defined in this section.

4.2 Overall ANTLR Grammar File Structure

ANTLR generates recognizers that apply grammatical structure to a stream of input symbols, which can be characters, tokens, or tree nodes. ANTLR presents you with a single, consistent syntax for lexing, parsing, and tree parsing. In fact, ANTLR generates the same kind of recognizer for all three. Contrast this with having to use different syntax for lexer and parser as you do with tools lex [Les75] and YACC [Joh79].

This section describes ANTLR's consistent grammar syntax for the four kinds of ANTLR grammars (*grammarType*): **lexer**, **parser**, **tree**, and com-

bined lexer and parser (no modifier). All grammars have the same basic structure:

```
/** This is a document comment */
grammarType grammar name;
«optionsSpec»
«tokensSpec»
«attributeScopes»
«actions»

/** doc comment */
rule1 : ... | ... | ... ;
rule2 : ... | ... | ... ;
...
```

The order of the grammar sections must be as shown, with the rules appearing after all the other sections.

From a grammar **T**, ANTLR generates a recognizer with a name indicating its role. Using the Java language target, ANTLR generates TLexer.java, TParser.java, and T.java for **lexer**, **parser**, and **tree** grammars, respectively. For combined grammars, ANTLR generates both TLexer.java and TParser.java as well as an intermediate temporary file, T__.g, containing the lexer specification extracted from the combined grammar. ANTLR always generates a vocabulary file in addition, called T.tokens, that other grammars use to keep their token types in sync with **T**.

A target is one of the languages in which ANTLR knows how to generate code.

The following simple (combined) grammar illustrates some of the basic features of ANTLR grammars:

Download grammars/simple/T.g

```
grammar T;
options {
    language=Java;
}
@members {
String s;
}
r : ID '#' {s = $ID.text; System.out.println("found "+s);} ;
ID: 'a'..'z'+ ;
WS: (' '|'\n'|'\r')+ {skip();} ; // ignore whitespace
```

Grammar **T** matches an identifier followed by a pound symbol. The action sets an instance variable and prints the text matched for the identifier. There is another action, this one in the lexer. {skip();} tells the lexer to skip the whitespace token and look for another.

To have ANTLR translate grammar **T** to Java code (or any other target according to the grammar's **language** option in T.g), use the following command line:

```
$ java org.antlr.Tool T.g
ANTLR Parser Generator  Version 3.0  1989-2007
$
```

The following main program illustrates how to have grammar **T** parse data from the standard input stream.

Download grammars/simple/Test.java

```
import org.antlr.runtime.*;

public class Test {
    public static void main(String[] args) throws Exception {
        ANTLRInputStream input = new ANTLRInputStream(System.in);
        TLexer lexer = new TLexer(input);
        CommonTokenStream tokens = new CommonTokenStream(lexer);
        TParser parser = new TParser(tokens);
        parser.r();
    }
}
```

Support classes ANTLRInputStream and CommonTokenStream provide streams of characters and tokens to the lexer and parser. Parsers begin parsing when control programs, such as Test, invoke one of the methods generated from grammar rules. The first rule invoked is usually called the *start symbol*. By convention, the first rule is usually the start symbol, though you can invoke any rule first. Here, parser.r() invokes rule **r** as the start symbol.

To test some input against grammar **T**, launch Java class Test in the usual way, and then type an identifier followed by space and then # and then a newline. To close standard input, do not forget to type the end-of-file character appropriate for your operating system, such as Ctrl+D on Unix. Here is a sample session:

```
⇐ $ java Test
⇐ abc #
⇐ EOF
⇒ found abc
$
```

Grammar Lexicon

The lexicon of ANTLR is familiar to most programmers because it follows the syntax of C and its derivatives with some extensions for grammatical descriptions.

Comments

There are single-line, multiline, and Javadoc-style comments:

```
/** This grammar is an example illustrating the three kinds
 *  of comments.
 */
grammar T;

/* a multi-line
   comment
 */

/** This rule matches a declarator for my language */
decl : ID ; // match a variable name
```

Identifiers

Token names always start with a capital letter and so do lexer rules. Nonlexer rule names always start with a lowercase letter. The initial character can be followed by uppercase and lowercase letters, digits, and underscores. Only ASCII characters are allowed in ANTLR names. Here are some sample names:

```
ID, LPAREN, RIGHT_CURLY // token names/rules
expr, simpleDeclarator, d2, header_file // rule names
```

Literals

ANTLR does not distinguish between character and string literals as most languages do. All literal strings one or more characters in length are enclosed in single quotes such as ';', 'if', '>=', and '\'' (refers to the one-character string containing the single quote character). Literals never contain regular expressions.

This changed in v3. In v2, char literals used single quotes, and string literals used double quotes.

Literals can contain Unicode escape sequences of the form \uXXXX, where XXXX is the hexadecimal Unicode character value. For example, '\u00E8' is the French letter *e* with a grave accent: 'è'. ANTLR also understands the usual special escape sequences: '\n' (newline), '\r' (carriage return), '\b' (backspace), and '\f' (form feed). ANTLR itself does not currently allow non-ASCII input characters. If you need non-ASCII characters within a literal, you must use the Unicode escape sequences at the moment.

The recognizers that ANTLR generates, however, assume a character vocabulary containing all Unicode characters.

Actions

Actions are code blocks written in the target language. You can use actions in a number of places within a grammar, but the syntax is always the same: arbitrary text surrounded by curly braces. To get the close curly character, escape it with a backslash: {System.out.println("\}");}. The action text should conform to the target language as specified with the **language** option.

The only interpretation ANTLR does inside actions relates to symbols for grammar attribute and output template references. See Chapter 6, *Attributes and Actions*, on page 130.

Templates

To emit structured text, such as source code, from a translator, use StringTemplate templates (see Chapter 9, *Generating Structured Text with Templates and Grammars*, on page 206). Set the **output** option to **template**. Each parser or tree parser rule then implicitly returns a template, which you can set with template rewrites. These rewrite templates are most often references to template names defined in a StringTemplate group elsewhere, but you can also specify in-line template literals. Template literals are either single-line strings in double quotes, "...", or multiline strings in double angle brackets, <<...>>, as shown in the highlighted section in the following grammar: *New in v3.*

```
/** Insert implied "this." on front of method call */
methodCall
    :   ID '(' ')' -> template(m={$ID.text}) "this.<m>();"
    ;
methodBody
    :   '{' ACTION '}'
        -> template(body={$ACTION.text})
<<
{
    <body>
}
>>
    ;
```

We'll explore the full syntax for template construction rules that follow the -> operator in Chapter 9, *Generating Structured Text with Templates and Grammars*, on page 206. For now, just be aware that templates are enclosed in either double quotes or double angle brackets.

The next section describes how to define rules within a grammar.

4.3 Rules

The rules of a grammar collectively define the set of valid sentences in a language. An individual rule, then, describes a partial sentence (sometimes called a *phrase* or *substructure*). Each rule has one or more alternatives. The alternatives can, in turn, reference other rules just as one function can call another function in a programming language. If a rule directly or indirectly invokes itself, the rule is considered recursive. The syntax for rules in lexers, parsers, and tree parsers is the same except that tree grammar rules may use tree structure elements, as we'll see below.

Because of ANTLR's unified recognition strategy, lexer grammar rules can also use recursion. Recursive rules in the lexer are useful for matching things such as nested comments; contrast this with most lexer generators such as lex that are limited to nonrecursive regular expressions rather than full grammars.

Besides specifying syntactic phrase structure, ANTLR rules have a number of components for specifying options, attributes, exception handling, tree construction, and template construction. The general structure of a rule looks like the following:

```
/** comment */
access-modifier rule-name[«arguments»] returns [«return-values»]
    «throws-spec»
    «options-spec»
    «rule-attribute-scopes»
    «rule-actions»
    :   «alternative-1» -> «rewrite-rule-1»
    |   «alternative-2» -> «rewrite-rule-2»
    ...
    |   «alternative-n» -> «rewrite-rule-n»
    ;
    «exceptions-spec»
```

The simplest rules specify pure syntactic structure:

```
/** A decl is a type followed by ID followed by 0 or more
 *  ',' ID sequences.
 */
decl:   type ID (',' ID)* ;  // E.g., "int a", "int a,b"
type:   'int'   // match either an int or float keyword
    |   'float'
    ;
ID  :   'a'..'z'+ ;
```

As an optimization, ANTLR collapses rules and subrules, whose alternatives are single token references without actions, into token sets, as demonstrated in the following rule:

```
type:   'int' | 'float' | 'void' ;
```

ANTLR generates a bit set test or token type range check instead of a **switch** statement.

Elements within Alternatives

The elements within an alternative specify what to do at any moment just like statements in a programming language. Elements either match a construct on the input stream or execute an action. These actions perform a translation or aid in the recognition process, such as updating or testing a symbol table. You can suffix elements with operators that alter the recognition of that element or specify what kind of abstract syntax tree to create (see Chapter 7, *Tree Construction*, on page 162). Figure 4.1, on the following page, summarizes the core elements and their variations.

Element Sets

Token and character elements can be combined into sets using subrule notation, a range operator, or the "not" operator, as summarized in the following table.

Syntax	Description
'*x*'..'*y*'	*Lexer*. Match any single character between range *x* and *y*, inclusively.
(*A* | *B* | . . . | *C*)	*Parser or tree parser*. Match any token from the list of tokens within the subrule. These subrules are collapsed to a set only when there are no actions and each alternative is exactly one token reference.
('*x*'..'*y*' | '*a*' | . . . | '*b*')	*Lexer*. Match any single character from the list of characters and ranges within the subrule.
~*x*	*Any grammar*. Match any single character or token *not* in *x* where *x* can be a single element, a range, or a subrule set.

Element Labels

Actions that need to refer to rule, token, or character references within a grammar can do so by referencing labels attached to those elements. To label an element, assign the element to a label name: *x*=*T*. For example, the following rule defines labels on a string literal, a token reference, and a rule reference:

```
classDef
    :   c='class' name=ID '{' m=members '}'
    ;
```

Actions can reference these labels via $c, $name, and $m, where the first two labels have type Token and the last label has type members_return.

Syntax	Description
T	*Parser or tree parser.* Match token *T* at the current input position. Tokens always begin with a capital letter.
T	*Lexer.* Invoke lexer rule *T*.
T[«*args*»]	*Lexer.* Invoke **fragment** lexer rule *T*, passing in a list of arguments. It does not work for non-**fragment** rules.
'*literal*'	*Any grammar.* Match the string literal at the current input position. A string literal is simply a token with a fixed string.
r	*Parser or tree parser.* Match rule r at current input position, which amounts to invoking the rule just like a function call.
r[«*args*»]	*Parser or tree parser.* Match rule r at current input position, passing in a list of arguments just like a function call. The arguments inside the square brackets are in the syntax of the target language and are usually a comma-separated list of expressions.
.	*Any grammar.* The wildcard. Match a single token in a parser. Match any single character in a lexer. In a tree parser, match an entire subtree. For example, the following rule matches a function without parsing the body: `func : ^(FUNC ID args .) ;`. The wildcard skips the entire last child (the body subtree).
{«*action*»}	*Any grammar.* Execute an action immediately after the preceding alternative element and immediately before the following alternative element. The action conforms to the syntax of the target language. ANTLR ignores what is inside the action except for attribute and template references such as `$x.y`. Actions are not executed if the parser is backtracking; instead, the parser rewinds the input after backtracking and reparses in order to execute the actions once it knows which alternatives will match.
{«*p*»}?	*Any grammar.* Evaluate semantic predicate «*p*». Throw FailedPredicateException if «*p*» evaluates to false at parse time. Expression «*p*» conforms to the target language syntax. ANTLR might also hoist semantic predicates into parsing decisions to guide the parse. These predicates are invaluable when symbol table information is needed to disambiguate syntactically ambiguous alternatives. Cf. Chapter 13, *Semantic Predicates*, on page 317 for more information.

Figure 4.1: ANTLR rule elements

ANTLR generates members_return while generating code for rule **members**. See Chapter 6, *Attributes and Actions*, on page 130 for more information about the predefined properties of token and rule labels.

When an action must refer to a collection of tokens matched by an alternative, ANTLR provides a convenient labeling mechanism that automatically adds elements to a list in the order they are matched, for example, *x*+=*T*.[4] In the following example, all **ID** tokens are added to a single list called ids.

New in v3.

```
decl:   type ids+=ID (',' ids+=ID)* ';'
    ;
```

In an action, the type of ids is List and will contain all Token objects associated with **ID** tokens matched at the time of reference. ANTLR generates the following code for rule **decl** where the highlighted sections result from the ids label:

```
public void decl() throws RecognitionException {
    Token ids=null;
    List list_ids=null;
    try {
        // match type
        type();
        // match ids+=ID
        ids=(Token)input.LT(1);
        match(input,ID,FOLLOW_ID_in_decl13);
        if (list_ids==null) list_ids=new ArrayList();
        list_ids.add(ids);
        ...
    }
    catch (RecognitionException re) {
        «error-handling»
    }
}
```

You can also collect rule AST or template return values using the += operator:

```
options {output=AST;} // or output=template
// collect ASTs from expr into a list called $e
elist:  e+=expr (',' e+=expr)* ;
```

ANTLR will emit an error if you use a += label without the **output** option. In an action, the type of e will be List and will contain either tree nodes (for example, CommonTree) or StringTemplate instances.

4. This was suggested by John D. Mitchell, a longtime supporter of ANTLR and research collaborator.

Syntax	Description
T!	*Parser*. Match token *T*, but do not include a tree node for it in the tree created for this rule.
r!	*Parser*. Invoke rule *r*, but do not include its subtree in the tree created for the enclosing rule.
T^	*Parser*. Match token *T* and create an AST node using the parser's tree adapter. Make the node the root of the enclosing rule's tree regardless of whether *T* is in a subrule or at the outermost level.
r^	*Parser*. Invoke rule *r*, and make its return AST node the root of the enclosing rule's tree regardless of whether *r* is in a subrule or at the outermost level. Rule *r* should must a single node, not a subtree.

Figure 4.2: Tree construction operators for rule elements

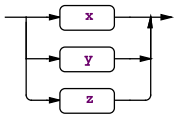
Tree Operators

If you are building trees, you can suffix token and rule reference elements in parsers with AST construction operators. These operators indicate the kind of node to create in the tree and its position relative to other nodes. These operators are extremely convenient in some kinds of rules such as expression specifications. But, in general, the rewrite rules provide a more readable tree construction specification. Figure 4.2 summarizes the tree operators and rewrite rules.

See Chapter 7, *Tree Construction*, on page 162 for more information about tree construction using these operators.

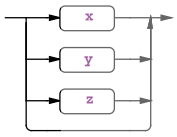
Extended BNF Subrules

ANTLR supports EBNF grammars: BNF grammars augmented with repetition and optional operators as well as parenthesized subrules to support terse grammatical descriptions. All parenthesized blocks of grammar elements are considered subrules. Because single elements suffixed with an EBNF operator have implied parentheses around them, they too are subrules. Subrules are like anonymous embedded rules and support an **options** section. The ANTLR EBNF syntax is summarized in Figure 4.3, on the next page.

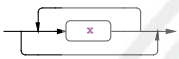


(«x»|«y»|«z»)
Match any alternative within the subrule exactly once.

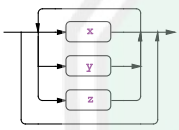
x?
Element *x* is optional.



(«x»|«y»|«z»)?
Match nothing or any alternative within subrule.

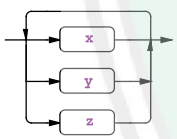


*x**
Match element *x* zero or more times.



(«x»|«y»|«z»)*
Match an alternative within subrule zero or more times.

x+
Match element *x* one or more times.



(«x»|«y»|«z»)+
Match an alternative within subrule one or more times.

Figure 4.3: EBNF grammar subrules where «...» represents a grammar fragment

Here is an example that matches an optional **else** clause for an **if** statement:

```
stat:   'if' '(' expr ')' ( 'else' stat )? // optional else clause
    |   ...
    ;
```

In the lexer, you will commonly use subrules to match repeated character groups. For example, here are two common rules for identifiers and integers:

```
/** Match identifiers that must start with '_' or a letter. The first
 *  characters are followed by zero or more letters, digits, or '_'.
 */
ID : ('a'..'z'|'A'..'Z'|'_') ('a'..'z'|'A'..'Z'|'_'|'0'..'9')* ;
INT: '0'..'9'+ ;
```

Subrules allow all the rule-level options listed in Figure 4.4, on page 113, and use this syntax:

```
( options {«option-assignments»} : «subrule-alternatives» )
```

The only addition is the handy **greedy** option. This option alters how ANTLR resolves nondeterminisms between subrule alternatives and the subrule exit branch. A *nondeterminism* is a situation in which the recognizer cannot decide which path to take because an input symbol predicts taking multiple paths (see Section 11.5, *Ambiguities and Nondeterminisms*, on page 273). Greedy decisions match as much input as possible in the current subrule even if the element following the subrule can match that same input. A nongreedy decision instead chooses to exit the subrule as soon as it sees input consistent with what follows the subrule. This option is useful only when ANTLR detects a nondeterminism; otherwise, ANTLR knows there is only one viable path. The default is to be greedy in all recognizer types.

You'll use nongreedy subrules in rules that can match any character until it sees some character sequence. For example, the following is a rule that matches multiline comments. The subrule matches any character until it finds the final '*/':

```
ML_COMMENT
    :   '/*' ( options {greedy=false;} : . )* '*/'
    ;
```

Because the subrule can also match '*/', ANTLR considers the subrule nondeterministic. By telling ANTLR to be nongreedy, the subrule matches characters until it sees the end of the comment.

What you really want to type, though, and what you will see in other systems, is the terse notation: '.*' and '.+'. Unfortunately, following the usual convention that all subrules are greedy makes this notation useless. Such greedy subrules would match all characters until the end of file. Instead, ANTLR considers them idioms for "Match any symbol until you see what lies beyond the subrule." ANTLR automatically makes these two subrules nongreedy. So, you can use '.*' instead of manually specifying the option.

Actions Embedded within Rules

To execute an action before anything else in the rule and to define local variables, use an **init** action. Similarly, to execute something after any alternative has been executed and right before the rule returns, use an **after** action. For example, here is a rule that initializes a return value to zero before matching an alternative. It then prints the return value after matching one of the alternatives:

*New in v3. The **after** action is new, and the **init** action was simply a code block in v2.*

```
r returns [int n]
@init {
    $n=0; // init return value
}
@after {
    System.out.println("returning value n="+$n);
}
    :   ... {$n=23;}
    |   ... {$n=9;}
    |   ... {$n=1;}
    |   // use initialized value of n
    ;
```

One of the most common uses for the **init** action is to initialize return values. See Chapter 6, *Attributes and Actions*, on page 130 for more information.

Rule Arguments and Return Values

Just like function calls, ANTLR parser and tree parser rules can have arguments and return values. ANTLR lexer rules cannot have return values, and only **fragment** lexer rules (see Section 4.3, *Lexical Rules*, on page 107) can have parameters. Those rules that can define return values can return multiple values, whereas functions in most languages can return only a single value. For example, consider the following rule with two arguments and two return values:

New in v3. In v2, you could have only one return value.

```
r[int a, String b] returns [int c, String d]
    : ... {$c=$a; $d=$b;}
    ;
```

To set the return value, use the relevant attribute on the left side of an assignment such as $c=...;. See Chapter 6, *Attributes and Actions*, on page 130 for more information about the use of attributes.

For rule **r**, ANTLR generates the following code where the highlighted executable lines derive from rule **r**'s parameters and return values:

```
public static class r_return extends ParserRuleReturnScope {
    public int c;
    public String d;
};
public r_return r(int a, String b) throws RecognitionException {
    r_return retval = new r_return();
    retval.start = input.LT(1);
    try {
        ...
        retval.c=a; retval.d=b;
    }
    catch (RecognitionException re) {
        reportError(re);
        recover(input,re);
    }
    finally {
        retval.stop = input.LT(-1);
    }
    return retval;
}
```

Rule references use syntax similar to function calls. The only difference is that, instead of parentheses, you use square brackets to pass rule parameters. For example, the following rule invokes rule **r** with two parameters and then, in an action, accesses the second return value, $v.d:

```
s   :   ... v=r[3,"test"] {System.out.println($v.d);}
    ;
```

Dynamic Rule Attribute Scopes

Besides the predefined rule attributes, rules can define scopes of attributes that are visible to all rules invoked by a rule. Here's a simple example that makes attribute name available to any rule invoked directly or indirectly by **method**:

New in v3.

```
method
@scope {
    String name;
}
    :   'void' ID {$method::name = $ID.text;} '(' args ')' body
    ;
```

Down in a deeply nested expression rule, for example, you can directly access the method's name without having to pass the method name all the way down to that rule from **method**:

```
atom:   ID {System.out.println("ref "+$ID.text+" in "+$method::name);}
    ;
```

If for some reason you wanted your language to allow nested method definitions, each method definition would automatically get its own name variable. Upon entry to **method**, the recognizer pushes the old name value onto a stack and makes space for a new name. Upon exit, **method** pops off the current name value. See Section 6.5, *Dynamic Attribute Scopes for Interrule Communication*, on page 148 for more information.

Rewrite Rules

ANTLR parsers can generate ASTs or StringTemplate templates by specifying an **output** option (**AST** and **template**, respectively). In either case, all rules have an implied return value that is set manually in an action or, more commonly, in a *rewrite rule* (*rewrite alternative* would be more accurate, but *rewrite rule* sounds better). Every alternative, whether it is in a subrule or the outermost rule level, can have a rewrite rule. Regardless of location, the rewrite rule always sets the return object for the entire rule. Symbol -> begins each rewrite rule. For example, the following rule matches the unary minus operator followed by an identifier. The rewrite rule makes a tree with the operator at the root and the identifier as the first and only child.

New in v3.

```
unaryID : '-' ID -> ^('-' ID) ;
```

As a more complicated example, consider the following rule that specifies how to match a class definition. It also provides a rewrite rule that describes the shape of the desired AST with 'class' at the root and the other elements as children.

```
classDefinition
    :   'class' ID ('extends' sup=typename)?
        ('implements' i+=typename (',' i+=typename)*)?
        '{'
        (   variableDefinition
        |   methodDefinition
        |   ctorDefinition
        )*
        '}'
        -> ^('class' ID ^('extends' $sup)? ^('implements' $i+)?
             variableDefinition* ctorDefinition* methodDefinition*
            )
```

Rewrite rules for AST construction are parser-grammar-to-tree-grammar mappings. When generating templates, on the other hand, rewrite rules specify the template to create and a set of argument assignments that set the attributes of the template. The following expression rule from a tree grammar illustrates how to create template instances:

```
expr:   ^(CALL c=expr args=exprList) -> call(m={$c.st},args={$args.st})
    |   ^(INDEX a=expr i=expr) -> index(list={$a.st},e={$i.st})
    |   primary -> {$primary.st} // return ST computed in primary
    ;
```

The rewrite rule in the first alternative of rule **expr** instantiates the template called call and sets that template's two attributes: m and args. The argument list represents the interface between ANTLR and StringTemplate. The assignments reference arbitrary actions in order to set the attribute values. In this case, m is set to the template returned from invoking rule **expr**. args is set to the template returned from invoking **exprList**. The third alternative's rewrite rule does not reference a template by name. Instead, the specified action evaluates to a StringTemplate instance. In this case, it evaluates to the template returned from invoking **primary**.

The template definitions for call and index are defined elsewhere in a StringTemplate group, but you can specify in-line templates if necessary. Here are some sample templates in StringTemplate group format:

```
call(m,args) ::=  "<m>(<args>)"
index(list,e) ::= "<list>[e]"
```

See Chapter 7, *Tree Construction*, on page 162 and Chapter 9, *Generating Structured Text with Templates and Grammars*, on page 206 for details about using rewrite rules.

Rule Exception Handling

When an error occurs within a rule, ANTLR catches the exception, reports the error, attempts to recover (possibly by consuming more tokens), and then returns from the rule. In other words, every rule is wrapped in a **try/catch** block:

```
void rule() throws RecognitionException {
    try {
        «rule-body»
    }
    catch (RecognitionException re) {
        reportError(re);
        recover(input,re);
    }
}
```

To replace that **catch** clause, specify an exception after the rule definition:

```
r   :   ...
    ;
    catch[RecognitionException e] { throw e; }
```

This example shows how to avoid reporting an error and avoid recovering. This rule rethrows the exception, which is useful when it makes more sense for a higher-level rule to report the error.

You can specify other exceptions as well:

```
r   :   ...
    ;
    catch[FailedPredicateException fpe] { ... }
    catch[RecognitionException e] { ...; }
```

When you need to execute an action even if an exception occurs (and even when the parser is backtracking), put it into the **finally** clause:

```
r   :   ...
    ;
    // catch blocks go first
    finally { System.out.println("exit rule r"); }
```

The **finally** clause is the last part a rule executes before returning. The clause executes after any dynamic scopes close and after memoization occurs; see Section 6.5, *Dynamic Attribute Scopes for Interrule Communication*, on page 148 and Section 14.5, *Memoization*, on page 343. If you want to execute an action after the rule finishes matching the alternatives but before it does its cleanup work, use an **after** action (Section 6.2, *Embedding Actions within Rules*, on page 137).

ANTLR also knows how to do single-token insertion and deletion in order to recover in the middle of an alternative without having to exit the surrounding rule. This all happens automatically.

See Chapter 10, *Error Reporting and Recovery*, on page 241 for a complete list of exceptions and for more general information about their reporting and recovery.

Syntactic Predicates

A syntactic predicate indicates the syntactic context that must be satisfied if an alternative is to match. It amounts to specifying the lookahead language for an alternative (see Section 2.7, *Categorizing Recognizers*, on page 51 for more information about lookahead). In general, a parser

will need to backtrack over the elements within a syntactic predicate to properly test the predicate against the input stream. Alternatives predicated with syntactic predicates are attempted in the order specified. The first alternative that matches wins. The syntax of a syntactic predicate looks like an ordinary subrule, but with a suffix operator of =>. Syntactic predicates must be on the extreme left edge of an alternative. For example, consider rule **stat** whose two alternatives are ambiguous:

```
stat:   (decl)=>decl ';'
    |   expr ';'
    ;
```

The syntactic predicate on the first alternative tells ANTLR that a successful match of **decl** predicts that alternative. The parser attempts rule **decl**. If it matches, the parser rewinds the input and begins parsing the first alternative. If the predicate fails, the parser assumes that the second alternative matches.

This example illustrates the resolution of the C++ ambiguity that statements such as `T(i);` can be either declarations or expressions syntactically. To resolve the issue, the language reference says to choose declaration over expression if both are valid. Because ANTLR chooses the first alternative whose predicate matches, the `decl ';'` alternative matches input `T(i);`.

The last alternative in a series of predicated alternatives does not need a predicate because it is assumed to be the default if nothing else before it matches. Alternatives that are not mutually ambiguous, even in the same block alternatives, do not need syntactic predicates. For example, adding a few more alternatives to rule **stat** does not confuse ANTLR:

```
stat:   (decl)=>decl ';'
    |   expr ';'
    |   'return' expr ';'
    |   'break' ';'
    ;
```

Even when the second ambiguous alternative is last, ANTLR still knows to choose it if the first alternative fails:

```
stat:   (decl)=>decl ';'
    |   'return' expr ';'
    |   'break' ';'
    |   expr ';'
    ;
```

As a convenience and to promote clean-looking grammars, ANTLR provides the **backtrack** option. This option tells ANTLR to automatically insert syntactic predicates where necessary to disambiguate decisions that are not *LL(*)* (cf. Section 2.7, *Categorizing Recognizers*, on page 51). Here is an alternative, functionally equivalent version of rule **stat**:

```
stat
options {
    backtrack=true;
}
    :   decl ';'
    |   'return' expr ';'
    |   'break' ';'
    |   expr ';'
    ;
```

See Chapter 14, *Syntactic Predicates*, on page 331 for more information about how syntactic predicates guide the parse.

Lexical Rules

Lexer rules differ from parser and tree parser rules in a number of important ways, though their syntax is almost identical. Most obviously, lexer rules are always token names and must begin with a capital letter. For example, the following rule matches ASCII identifiers:

```
ID  :   ('a'..'z'|'A'..'Z'|'_') ('a'..'z'|'A'..'Z'|'_'|'0'..'9')* ;
```

To distinguish methods generated from rules in the lexer from the token type definitions, ANTLR prefixes methods with m. Rule **ID**'s method is mID(), for example.

Unlike a parser grammar, there is no start symbol in a lexer grammar. The grammar is simply a list of token definitions that the lexer can match at any moment on the input stream. ANTLR generates a method called nextToken() (that satisfies the TokenSource interface). nextToken() amounts to a big **switch** statement that routes the lexer to one of the lexer rules depending on which token is approaching.

Just as with parser grammars, it is useful to break up large rules into smaller rules. This makes rules more readable and also reusable by other rules. The next section describes how to factor rules into helper rules.

Fragment Lexer Rules

Because ANTLR assumes that all lexer rules are valid tokens, you must prefix factored "helper rules" with the **fragment** keyword. This keyword

and that it should not yield a token to the parser. The following rule specifies the syntax of a Unicode character and uses a **fragment** rule to match the actual hex digits:

Changed in v3. v2 called these helper rules protected, a truly horrible name.

```
UNICODE_CHAR
    :   '\\' 'u' HEX_DIGIT HEX_DIGIT HEX_DIGIT HEX_DIGIT
    ;
fragment
HEX_DIGIT
    :   '0'..'9'|'a'..'f'|'A'..'F'
    ;
```

If **UNICODE_CHAR** were to be used only by another rule, such as **STRING**, then it too would be a **fragment** rule.

It makes no sense to allow parameters and return values on token definition rules because, in general, no one is invoking those rules explicitly. Method nextToken() is implicitly invoking those rules, and it would not know what parameters to pass and could not use the return value. **fragment** rules, on the other hand, are never implicitly invoked—other lexer rules must explicitly invoke them. Consequently, **fragment** rules can define parameters, though they can't define return values because lexer rules always return Token objects. For example, here is a rule that matches (possibly nested) code blocks in curly braces and takes a parameter dictating whether it should strip the curlies:

```
fragment
CODE[boolean stripCurlies]
    :   '{' ( CODE[stripCurlies] | ~('{'|'}') )* '}'
        {
        if ( stripCurlies ) {
           setText(getText().substring(1, getText().length()));
        }
        }
    ;
```

Another rule would invoke **CODE** via CODE[false] or CODE[true].

Also note that lexer rule **CODE** is recursive because it invokes itself. This is the only way to properly match nested code blocks. You could add embedded actions to count the number of open and close braces, but that is inelegant. ANTLR generates recursive-descent recognizers for lexers just as it does for parsers and tree parsers. Consequently, ANTLR supports recursive lexer rules, unlike other tools such as lex.

Lexer rules can call other lexer rules, but that doesn't change the token type of the invoking rule. When lexer rule *T* calls another lexer rule or fragment rule, the return type for *T* is still *T*, not the token type of the

Ignoring Whitespace and Comments

One of the most difficult lexing issues to deal with is the paradox that the parser must ignore whitespace and comments but at the same time provide access to those tokens for use during translation. To solve this problem, ANTLR allows each token object to exist on different *channels*, sort of like different radio frequencies. The parser can "tune" to any single channel; hence, it ignores any off-channel tokens. Objects implementing the TokenStream interface, such as CommonTokenStream, provide access to these off-channel tokens for use by actions. The following token definition matches whitespace and specifies with an action that the token should go on to the standard hidden channel (see class Token for the definitions of DEFAULT_CHANNEL and HIDDEN_CHANNEL):

Improved in v3.

```
WS  :  (' '|'\t'|'\r'|'\n')+ {$channel=HIDDEN;} ;
```

Character streams in the ANTLR runtime library automatically track newlines so that you do not have to manually increment a line counter. Further, the current character position within the current line is always available via getCharPositionInLine() in Lexer. Character positions are indexed from 0. Note that tab characters are not taken into consideration—the character index tracks tabs as one character.

Improved in v3.

For efficiency reasons, lexer rules can also indicate that the token should be matched but no actual Token object should be created. Method skip() in an embedded lexer rule action forces the lexer to throw out the token and look for another. Most language applications can ignore whitespace and comments, allowing you to take advantage of this efficiency. Here is an example whitespace rule:

```
WS  :  (' '|'\t'|'\r'|'\n')+ {skip();} ;
```

Rather than throwing out tokens, sometimes you'd like to emit more than a single token per lexer invocation.

Emitting More Than One Token per Lexer Rule

Lexer rules can force the lexer to emit more token per rule invocation by manually invoking method emit(). This feature solves some fairly difficult problems such as inserting *imaginary tokens* (tokens for which there is no input counterpart). The best example is lexing Python. Because Python uses indentation to indicate code blocks, there are no explicit begin and end tokens to group statements within a block. For example, in the following Python code, the if statement and the method call to g() are at the same outer level. The print statement and method call to f() are the same inner, nested level.

```
if foo:
    print "foo is true"
    f()
g()
```

Without begin and end tokens, parsing this input presents a problem for the parser when it tries to group statements. The lexer needs to emit imaginary **INDENT** and **DEDENT** tokens to indicate the begin and end of code blocks. The token sequence must be as follows:

```
IF ID : NL INDENT PRINT STRINGLITERAL NL ID ( ) NL DEDENT ID ( ) NL
```

The parser rule for matching code blocks would look like this:

```
block : INDENT statement+ DEDENT ;
```

The lexer can emit the **INDENT** token when it sees whitespace that is more deeply indented than the previous statement's whitespace; however, there is no input character to trigger the **DEDENT** token. In fact, the lexer must emit a **DEDENT** token when it sees less indentation than the previous statement. The lexer might even have to emit multiple **DEDENT** tokens depending on how far out the indentation has moved from the previous statement. The **INDENT** rule in the lexer might look something like this:

New in v3. In v2, your lexer could emit only one token at a time.

```
INDENT
    :   // turn on rule only if at left edge
        {getCharPositionInLine()==0}?=>
        (' '|'\t')+ // match whitespace
        {
        if ( «indentation-bigger-than-before» ) {
            // can only indent one level at a time
            emit(«INDENT-token»);
            «track increased indentation»
        }
        else if ( «indentation-smaller-than-before» ) {
            int d = «current-depth» - «previous-depth»;
            // back out of d code blocks
            for (int i=1; i<=d; i++) {
                emit(«DEDENT-token»);
            }
            «reduce indentation»
        }
        }
    ;
```

After matching a lexer rule, if you have not emitted a token manually in an action, nextToken() will emit a token for you. The token is based upon the text and token type for the rule. Note that, for efficiency reasons, the CommonTokenStream class does not support multiple token emissions

for the same invocation of nextToken(). Read the following from class Lexer when you try to implement multiple token emission:

```
/** The goal of all lexer rules/methods is to create a token object.
 *  This is an instance variable as multiple rules may collaborate to
 *  create a single token.  nextToken will return this object after
 *  matching lexer rule(s).  If you subclass to allow multiple token
 *  emissions, then set this to the last token to be matched or
 *  something nonnull so that the auto token emit mechanism will not
 *  emit another token.
 */
protected Token token;

/** Currently does not support multiple emits per nextToken invocation
 *  for efficiency reasons.  Subclass and override this method and
 *  nextToken (to push tokens into a list and pull from that list rather
 *  than a single variable as this implementation does).
 */
public void emit(Token token) {
    this.token = token;
}
```

Tree Matching Rules

For translation problems that require multiple passes over the input stream, you should create parsers that build ASTs. ASTs not only are a terse representation of the input stream but also encode the grammatical structure applied to that input string. Rather than building a tree walker by hand or using a simple visitor pattern that has no contextual information, use a tree grammar that specifies the two-dimensional structure of the tree created by the parser. Again, ANTLR uses essentially the same syntax with the addition of a new construct that specifies the two-dimensional structure of a subtree: a root and one or more children. The syntax uses parentheses with a caret symbol on the front to distinguish it from an EBNF subrule:

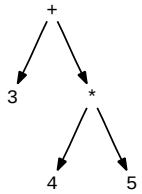
Changed in v3. Using ^ makes much more sense than the # symbol v2 used.

```
^( root child1 child2 ... childn )
```

For example, rule **expr**:

```
expr:   ^('+' expr expr)
    |   ^('*' expr expr)
    |   INT
    ;
```

matches ASTs consisting of expression subtrees with '+' and '*' operators as interior nodes (subtree roots) and **INT** nodes as leaves. Here is the tree for 3+4*5:



Trees have self-similar structures in that a small subtree and a big subtree are identical in structure. In this case, they both have operators at the root and two children as operands, which can also be subtrees. That self-similar structure is reflected in the recursive nature of the **expr** rule.

As a larger example, consider the following rules taken from a Java-like language grammar:

```
expression
    :   ^(unary_op expression)
    |   ^(CALL expression expressionList)
    |   ^(INDEX expression expression)
    |   primary
    ;

unary_op
    :   UNARY_MINUS|UNARY_PLUS|UNARY_NOT|UNARY_BNOT
    ;

primary
    :   ID
    |   INT
    |   FLOAT
    |   'null'
    ;
```

As you can see, tree grammars look like parser grammars with the addition of a few tree expressions. If the input to a tree grammar were a flat tree (a linked list), then the tree grammar would not have any tree constructs and would look identical to a parser grammar. In that sense, a tree grammar reduces to a parser grammar when the input looks like a one-dimensional token stream.

Tree grammar rules can have parameters and return values just like parser grammar rules, and actions can contain attribute references. The difference is that references to T for some token T yield pointers to the tree node matched for that reference. Further, ANTLR v3 does not allow tree grammars to create new trees. A future version will allow this in order to support tree transformations.

Option	Description
backtrack	When **true**, indicate that ANTLR should backtrack when static *LL(*)* grammar analysis fails to yield a deterministic decision for this rule or subrules within the rule. This is usually used in conjunction with the **memoize** option. The default is to use the grammar's **backtrack** option. See Section 5.3, *backtrack Option*, on page 121.
memoize	Record partial parsing results to guarantee that, while backtracking, the parser never parses a rule more than once for a given input position. The default is to use the grammar's **memoize** option. See Section 5.4, *memoize Option*, on page 122.
k	Limit the decision generated for this rule, and any contained subrules, to use a maximum, fixed-lookahead depth of *k*. This turns off *LL(*)* analysis in favor of classical *LL(k)*. If the overall grammar has fixed lookahead, a rule can override this by setting k=*. The default is to use the grammar's **k** option. See Section 5.11, *k Option*, on page 129.

Figure 4.4: Summary of ANTLR rule-level options

See Section 5.5, *tokenVocab Option*, on page 122 for a sample test program that invokes a tree parser.

Rule Options

Sometimes rules need options such as turning on memoization. The syntax mirrors the options specification for grammars:

```
decl
options {
    memoize=true;
}
    :   type ID (',' ID)*
    ;
```

The set of options available to rules are summarized in Figure 4.4.

The following sections return to the outer grammar level to discuss **tokens** and **scope** specifications.

4.4 Tokens Specification

Use the grammar **tokens** specification to introduce new tokens or to give better names to token literals (see Section 2.6, *Vocabulary Symbols Are Structured Too*, on page 44 for more about tokens). The **tokens** specification has the following form:

```
tokens {
    token-name1;
    token-name2 = 'string-literal';
    ...
}
```

It allows you to introduce *imaginary tokens*, which are token names that are not associated with any particular input character(s). Imaginary tokens usually become subtree root nodes that act as operators for a series of operands (children). For example, **VARDEF** is a convenient root node for the declaration int i;. It would have the type (int) and variable name (i) as children. Using ANTLR's tree grammar notation, the tree looks like this:

```
^(VARDEF int i)
```

and could be built with a parser grammar such as this:

```
grammar T;
tokens {
    VARDEF;
}
var : type ID ';' -> ^(VARDEF type ID) ;
```

The **tokens** specification also allows you to provide an alias for a string literal, which is useful when the alias is more descriptive than the literal. For example, token name **MOD** is more descriptive than the literal '%':

```
grammar T;
tokens {
    MOD='%'; // alias MOD to '%'
}
expr : INT (MOD INT)* ;
```

4.5 Global Dynamic Attribute Scopes

Usually the only way to pass information from one rule to another is via parameters and return values. Unfortunately, this can be particularly inconvenient when rules are far away in the call chain. One of the best examples is determining whether an identifier is defined in the current

scope. A rule deeply nested in the expression call chain such as **atom** needs to access the current scope. The scope is probably updated in multiple rules such as **block**, **classDefinition**, and **methodDefinition**. There could be twenty rule invocations between **classDefinition** and **atom**. Passing the current scope down through all the twenty rules is a hassle and makes the grammar harder to read.

In a manner similar to an instance variable, ANTLR allows you to define global attribute scopes. These scopes are visible to actions in all rules. The specification uses the following syntax:

New in v3.

```
scope name {
    type1 attribute-name1;
    type2 attribute-name2;
    ...
}
```

So, for example, you might define a global scope to handle lists of symbols:

```
scope SymbolScope {
    List symbols;
}
```

Multiple rules would access that scope:

```
classDefinition
scope SymbolScope;
    : ...
    ;

methodDefinition
scope SymbolScope;
    : ...
    ;

block
scope SymbolScope;
    : ...
    ;
```

The key difference between the symbols list in SymbolScope and an instance variable is that the recognizer pushes a new list onto a stack of scopes upon entry to each method that declares scope SymbolScope;. See Chapter 6, *Attributes and Actions*, on page 130 for information about the usage and purpose of global dynamic scopes.

4.6 Grammar Actions

ANTLR generates a method for each rule in a grammar. The methods are wrapped in a class definition for object-oriented target languages. ANTLR provides named actions so you can insert fields and instance methods into the generated class definition (or global variables in the case of languages such as C). The syntax is as follows:

Improved in v3. The new name and scope based scheme is much clearer and more flexible than the global action mechanism in v2.

```
@action-name { ... }
@action-scope-name::action-name { ... }
```

The following example defines a field and method for the Java target using the **members** action:

```
grammar T;
options {language=Java;}
@members {
    int n;
    public void foo() {...}
}
a : ID {n=34; foo();} ;
```

To place the generated code in a particular Java package, use action **header**:

```
grammar T;
options {language=Java;}
@header {
package org.antlr.test;
}
```

When building a combined grammar, containing lexer and parser, you need a way to set the **members** or **header** for both. To do this, prefix the action with the action scope, which is one of the following: **lexer**, **parser**, or **treeparser**. For example, @header {...} is shorthand for @parser::header {...} if it is in a parser or combined parser. Here is how to add members to the lexer from a combined grammar and also set the package for both:

```
grammar T;
@header {import org.antlr.test;} // not auto-copied to lexer
@lexer::header{import org.antlr.test;}
@lexer::members{int aLexerField;}
```

See Section 6.2, *Grammar Actions*, on page 134 for more details. This chapter described the general structure of an ANTLR grammar as well as the details of the various grammar-level specifications. It also provided details on the structure of a rule, the rule elements, and the rule operators. The next chapter describes the grammar-level options that alter the way ANTLR generates code.

Chapter 5

ANTLR Grammar-Level Options

In the **options** section of an ANTLR grammar, you can specify a series of key/value assignments that alter the way ANTLR generates code. These options globally affect all the elements contained within the grammar, unless you override them in a rule. This chapter describes all the available options.

The options section must come after the **grammar** header and must have the following form:

```
options {
name1 = value1;
name2 = value2;
...
}
```

Option names are always identifiers, but values can be identifiers, single-quoted string literals, integers, and the special literal star, * (currently usable only with option **k**). Values are all literals and, consequently, can't refer to option names. For single-word string literals such as 'Java', you can use the shorthand Java, as shown here:

```
options {
  language=Java;
}
```

The list following this paragraph summarizes ANTLR grammar-level options. The subsections that follow describe them all in more detail; they're ordered from most commonly used to least commonly used.

language

Specify the target language in which ANTLR should generate recognizers. ANTLR uses the **CLASSPATH** to find directory org/antlr/codegen/templates/Java, in this case, used to generate Java. The default is Java. See Section 5.1, *language Option*, on page 119.

output

Generate output templates, **template**, or trees, **AST**. This is available only for combined, **parser**, and **tree** grammars. Tree grammars cannot currently output trees, only templates. The default is to generate nothing. See Section 5.2, *output Option*, on page 120.

backtrack

When **true**, indicates that ANTLR should backtrack when static *LL(*)* grammar analysis fails to yield a deterministic decision. This is usually used in conjunction with the **memoize** option. The default is **false**. See Section 5.3, *backtrack Option*, on page 121.

memoize

Record partial parsing results to guarantee that while backtracking the parser never parses the same input with the same rule more than once. This guarantees linear parsing speed at the cost of nonlinear memory. The default is **false**. See Section 5.4, *memoize Option*, on page 122.

tokenVocab

Specify where ANTLR should get a set of predefined tokens and token types. This is needed to have one grammar use the token types of another. Typically a **tree** grammar will use the token types of the **parser** grammar that creates its trees. The default is to not import any token vocabulary. See Section 5.5, *tokenVocab Option*, on page 122.

rewrite

When the output of your translator looks very much like the input, the easiest solution involves modifying the input buffer in-place. Re-creating the entire input with actions just to change a small piece is too much work. **rewrite** works in conjunction with output=template. Template construction actions usually just set the return template for the surrounding rule (see Section 9.4, *The ANTLR StringTemplate Interface*, on page 214). When you use rewrite=true, the recognizer also replaces the input matched by the rule with the template. See Section 5.6, *rewrite Option*, on page 124. The default is **false**.

superClass

Specify the superclass of the generated recognizer. This is not the supergrammar—it affects only code generation. The default is Lexer, Parser, or TreeParser depending on the grammar type. See Section 5.7, *superClass Option*, on page 125.

filter

Lexer only. All lexer rules are tried in order specified, looking for a match. Upon finding a matching rule, nextToken() returns that rule's Token object. If no rule matches, the lexer consumes a single character and again looks for a matching rule. The default is not to filter, **false**. See Section 5.8, *filter Option*, on page 126.

ASTLabelType

Set the target language type for all tree labels and tree-valued expressions. The default is Object. See Section 5.9, *ASTLabelType Option*, on page 127.

TokenLabelType

Set the target language type for all token labels and token-valued expressions. The default is interface Token. Cf. Section 5.10, *TokenLabelType Option*, on page 128.

k

Limit the recognizer generated from this grammar to use a maximum, fixed-lookahead depth of *k*. This turns off *LL(*)* analysis in favor of classical *LL(k)*. The default is * to engage *LL(*)*. See Section 5.11, *k Option*, on page 129.

5.1 language Option

The **language** option specifies the target language in which you want ANTLR to generate code. By default, **language** is Java. You must write your embedded grammar actions in the language you specify with the **language** option. For example:

```
grammar T;
options {language=Java;}
a : ... {«action-in-Java-language»} ... ;
```

Because of ANTLR's unique StringTemplate-based code generator, new targets are relatively easy to build; hence, you can choose from numerous languages such as Java, C, C++, C#, Objective-C, Python, and Ruby.[1]

The **language** option value informs ANTLR that it should look for all code generation templates in a directory with the same name such as org/antlr/codegen/templates/Java or org/antlr/codegen/templates/C.

1. See http://www.antlr.org/wiki/display/ANTLR3/Code+Generation+Targets for the latest information about ANTLR language targets.

Inside this directory, you will find a set of StringTemplate group files that tell ANTLR how to generate code for that language. For example, file Java.stg contains all the templates needed to generate recognizers in Java. To make a variation on an existing target, copy the directory to a new directory, and tweak the template files within. If you move files to directory TerenceJava, then say language=TerenceJava in your grammar to use the altered templates. All target directories must have a prefix of org/antlr/codegen/templates and be visible via the CLASSPATH environment variable.

5.2 output Option

The **output** option controls the kind of data structure that your recognizer will generate. Currently the only possibilities are to build abstract syntax trees, **AST**, and StringTemplate templates, **template**. When this option is used, every rule yields an AST or template.

Using output=AST allows you to use tree construction operators and rewrite rules described in Chapter 7, *Tree Construction*, on page 162. For example, the following simple grammar builds a tree with an imaginary root node, **DECL**, and a child node created from input token **ID**:

```
grammar T;

options {
        output=AST;
}

decl : ID -> ^(DECL ID) ;
ID : 'a'..'z'+ ;
```

Rules without defined return values generally have **void** return types unless the output option is used. For example, here is a piece of the generated code for this grammar:

```
public static class decl_return extends ParserRuleReturnScope {
    Object tree;
    public Object getTree() { return tree; }
};
public decl_return decl() throws RecognitionException {...}
```

When using output=template, rule definitions yield templates instead of trees:

```
public static class decl_return extends ParserRuleReturnScope {
    public StringTemplate st;
    /** To avoid unnecessary dependence on StringTemplate library,
     *  superclass uses Object not StringTemplate as return type.
     */
    public Object getTemplate() { return st; }
};
public decl_return decl() throws RecognitionException {...}
```

For more about generating templates, see Chapter 9, *Generating Structured Text with Templates and Grammars*, on page 206.

This option is available only for combined, **parser**, or **tree** grammars.

5.3 backtrack Option

The **backtrack** option informs ANTLR that, should *LL(*)* analysis fail, it should try the alternatives within the decision in the order specified at parse time, choosing the first alternative that matches. Once the parser chooses an alternative, it rewinds the input and matches the alternative a second time, this time "with feeling" to execute any actions within that alternative. Actions are not executed during the "guessing" phase of backtracking because there is no general way to undo arbitrary actions written in the target language.

New in v3.

No nondeterminism warnings are reported by ANTLR during grammar analysis time because, by definition, there is no uncertainty in backtracking mode—ANTLR simply chooses the first alternative that matches at parse time. You can look upon this option as a rapid prototyping feature because ANTLR accepts just about any grammar you give it. Later you can optimize your grammar so that it more closely conforms to the needs of *LL(*)*.

The nice aspect of using backtrack=true is that the generated recognizer will backtrack only in decisions that are not *LL(*)*. Even within decisions that are not completely *LL(*)*, a recognizer will backtrack only on those input sequences that render the decision non-*LL(*)*. ANTLR implements this option by implicitly adding a syntactic predicate to the front of every alternative that does not have a user-specified predicate there already (see Chapter 14, *Syntactic Predicates*, on page 331 for more information). The grammar analysis converts syntactic predicates to semantic predicates.

The analysis uses semantic predicates only when syntax alone is insufficient to distinguish between alternatives. Therefore, the generated code uses backtracking (syntactic predicates) only when *LL(*)* fails.

You should use backtracking sparingly because it can turn the usual linear parsing complexity into an exponential algorithm. It can mean the difference between having a parser that terminates in your lifetime and one that does not. To use the strength of backtracking but with the speed of a linear parser at the cost of some memory utilization, use the **memoize** option discussed in the next section.

5.4 memoize Option

Backtracking is expensive because of repeated rule evaluations for the same input position. By recording the result of such evaluations, the recognizer can avoid recomputing them in the future. This recording process is a form of dynamic programming called *memoization* (see Section 14.5, *Memoization*, on page 343). When the **memoize** option is **true**, ANTLR generates code at the beginning of each parsing method to check for prior attempts:

New in v3.

```
if ( backtracking>0 && alreadyParsedRule(input, rule-number) ) {return;}
```

and inserts code at the end of the rule's method to memoize whether this rule completed successfully for the current input position:

```
if ( backtracking>0 ) {
   memoize(input, rule-number, rule-starting-input-position);
}
```

Using the **memoize** option at the grammar level turns on memoization for each rule in the grammar. This results in considerable parsing overhead to store partial results even when backtracking never invokes the same rule at the same input position. It is often more efficient to turn on the **memoize** option at the rule level rather than globally at the grammar level. Options specified at the rule level override the same options given at the grammar level. See Chapter 14, *Syntactic Predicates*, on page 331 for more information.

5.5 tokenVocab Option

For large language projects, the parser typically creates an intermediate representation such as an abstract syntax tree (AST). This AST is walked one or more times to perform analysis and ultimately generate

code. I highly recommend you use a tree grammar to walk ASTs over a simple depth-first tree walk or visitor pattern (see Section 4.3, *Tree Matching Rules*, on page 111). Because a tree grammar must live in a different file than the parser that feeds it ASTs, there must be a way to synchronize the token types. Referring to **ID** in the tree grammar must have the same meaning as it does in the parser and lexer.

A grammar can import the vocabulary of another grammar using the **tokenVocab** option. The value associated with this option is the name of another grammar, not the name of a file. Consider the following simple grammar:

Download grammars/vocab/P.g

```
grammar P;
options {
    output=AST;
}
expr:   INT ('+'^ INT)* ;
INT :   '0'..'9'+;
WS  :   ' ' | '\r' | '\n' ;
```

From grammar **P**, ANTLR generates the recognizer files and a .tokens file that contains a list of token name and token type pairs for all tokens and string literals used in grammar **P**. There are two tokens in this case:

Download grammars/vocab/P.tokens

```
INT=4
WS=5
'+'=6
```

To walk the trees generated by **P**, a tree grammar must import the token vocabulary using tokenVocab=P, as shown in the following simple tree grammar:

Download grammars/vocab/Dump.g

```
tree grammar Dump;
options {
    tokenVocab=P;
    ASTLabelType=CommonTree;
}
expr:   ^( '+' expr {System.out.print('+');} expr )
    |   INT {System.out.print($INT.text);}
    ;
```

ANTLR looks for .tokens files in the library directory specified by the -lib command-line option to ANTLR, which defaults to the current directory.

For example, to generate code for the tree grammar, use the following (with or without the -lib option):

```
java org.antlr.Tool -lib . Dump.g
```

For completeness, here is a test program that prints the tree generated by the parser and then invokes the tree parser to dump the expression back out to text:

Download grammars/vocab/Test.java

```
// Create an input character stream from standard input
ANTLRInputStream input = new ANTLRInputStream(System.in);
PLexer lexer = new PLexer(input);      // create lexer
// Create a buffer of tokens between the lexer and parser
CommonTokenStream tokens = new CommonTokenStream(lexer);
PParser parser = new PParser(tokens); // create parser
PParser.expr_return r = null;
r = parser.expr(); // parse rule expr and get return structure
CommonTree t = (CommonTree)r.getTree();// extract AST
System.out.println(t.toStringTree());  // print out
// Create a stream of nodes from a tree
CommonTreeNodeStream nodes = new CommonTreeNodeStream(t);
Dump dumper = new Dump(nodes);         // create tree parser
dumper.expr();                         // parse expr
System.out.println();
```

Here is a sample session:

```
⇐ $ java Test
⇐ 3+4+5
⇐ EOF
⇒ (+ (+ 3 4) 5)
  3+4+5
  $
```

5.6 rewrite Option

For many translations, the output looks very different from the input. The translator generates and then buffers up bits of translated input that it subsequently organizes into larger and larger chunks. This leads to the final chunk representing the complete output. In other cases, however, the output looks very much like the input. The easiest approach is to have the translator just tweak the input. For example, you might want to instrument source code for debugging, as shown in Section 9.7, *Rewriting the Token Buffer In-Place*, on page 228. What you want to do is rewrite a few pieces of the input buffer during translation and then print the modified buffer.

To get this functionality, set option **rewrite** to true. Recognizers in this mode automatically copy the input to the output except where you specify translations using template rewrite rules. This option works only with output=template and in parsers or tree parsers. The following grammar (minus the lexical rules) translates int tokens to Integer, leaving the rest of the input alone:

Download grammars/rewrite/T.g

```
grammar T;
options {output=template; rewrite=true;}

decl:   type ID ';' ;                  // no translation here

type:   'int' -> template() "Integer" // translate int to Integer
    |   ID                            // leave this alone
    ;
```

Input int i; becomes Integer i;, but String i; stays the same. Without using rewrite mode, rule **decl** returns nothing because rules return nothing by default. Rewrite mode, on the other hand, is altering the token buffer—the main program can print the altered buffer. Here is the core of a test rig:

Download grammars/rewrite/Test.java

```
ANTLRInputStream input = new ANTLRInputStream(System.in);
TLexer lexer = new TLexer(input);
// use TokenRewriteStream not CommonTokenStream!!
TokenRewriteStream tokens = new TokenRewriteStream(lexer);
TParser parser = new TParser(tokens);
parser.decl();
System.out.print(tokens.toString()); // emit rewritten source
```

Note that this mode works only with TokenRewriteStream, not CommonTokenStream.

5.7 superClass Option

Sometimes it is useful to have your recognizers derive from a class other than the standard ANTLR runtime superclasses. You might, for example, define a class that overrides some of the standard methods to alter the behavior of your recognizers.

The **superClass** option specifies the class name that ANTLR should use as the superclass of the generated recognizer. The superclass is usually Lexer, Parser, or TreeParser but is DebugParser or DebugTreeParser if you use

the -debug command-line option. Whatever superclass you use instead must derive from the appropriate class mentioned here in order for the recognizer to compile and work properly. Here is a sample partial grammar that defines a superclass:

```
grammar T;
options {
    superClass=MyBaseParser;
}
```

ANTLR generates TParser as follows:

```
public class TParser extends MyBaseParser {...}
```

Naturally this option makes sense only for targets that are object-oriented programming languages. Note that the superclass is not a supergrammar—it is any Java class name.

5.8 filter Option

In general, programmers use ANTLR to specify the entire grammatical structure of an input file they want to process, but this is often overkill. Even for complicated files, sometimes it is possible to extract a few items of interest without having to describe the entire grammatical structure (as long as these items are lexically easy to identify). Some people call this *fuzzy parsing* because the recognizer does not match the exact structure of the input according to the full language syntax—the recognizer matches only those constructs of interest.[2]

The idea is to provide a series of lexical rules as with a normal grammar but have the lexer ignore any text that does not match one of the rules. You can look at **filter=true** mode as a normal lexer that has an implicit rule to catch and discard characters that do not match one of the other rules. nextToken() keeps scanning until it finds a matching lexical rule at which point it returns a Token object. If more than one rule matches the input starting from the current input position, the lexer resolves the issue by accepting the rule specified first in the grammar file; in other words, specify rules in the order of priority.

The krugle.com code search engine uses ANTLR v3's filter option to extract variable, method, and class definitions from, for example, Java and Python code spidered from the Internet.

2. See http://www.antlr.org/download/examples-v3.tar.gz for a full fuzzy Java parser that extracts class, method, and variable definitions as well as method call sites.

The following **lexer** grammar illustrates how to extract Java method calls:

```
lexer grammar FuzzyJava;
options {filter=true;}

CALL
    :   name=QID WS? '('
        {System.out.println("found call "+$name.text);}
    ;

SL_COMMENT
    :   '//' .* '\n'
    ;

WS  :   (' '|'\t'|'\r'|'\n')+
    ;

fragment
QID :   ID ('.' ID)*
    ;

fragment
ID  :   ('a'..'z'|'A'..'Z'|'_') ('a'..'z'|'A'..'Z'|'_'|'0'..'9')*
    ;
```

The **SL_COMMENT** rule is necessary because the program should not track method calls within comments. The multiline comments and strings would also be checked to do this for real.

Lexical rules can use skip() to force nextToken() to throw out the current token and look for another. The lexer returns the first nonskipped token to the parser.

Lexical **filter** mode is generally not used with a parser because the lexer yields an incomplete stream of tokens.

5.9 ASTLabelType Option

ANTLR makes no assumption about the actual type of tree nodes built during tree AST construction—by default, all node variables pointers are of type Object. ANTLR relies on the TreeAdaptor interface to know how to create nodes and hook them together to form trees (the CommonTreeAdaptor is a predefined implementation). Although this makes ANTLR very flexible, it can make embedded grammar actions inconvenient because of constant typecasting.

For example, in the following grammar, the type of expression $ID.tree is Object by default:

```
grammar T;
options {output=AST;}

/** we are creating CommonTree nodes, but ANTLR can't be sure;
 *  it assumes Object.
 */
e : ID {CommonTree t = (CommonTree)$ID.tree;} ;
ID: 'a'..'z'+ ;
```

The generated code for matching the **ID** token and building the AST will look something like this:

```
ID1=(Token)input.LT(1);
match(input,ID,FOLLOW_ID_in_e17);
ID1_tree = (Object)adaptor.create(ID1);
```

where ID1_tree is defined as follows:

```
Object ID1_tree=null;
```

The embedded action is translated to this:

```
CommonTree t = (CommonTree)ID1_tree;
```

To avoid having to use typecasts everywhere in your grammar actions, specify the type of your tree nodes via the **ASTLabelType** option. For example, if you are building CommonTree nodes with the CommonTreeAdaptor class, use option ASTLabelType=CommonTree. ANTLR will define variables such as ID1_tree to be of type CommonTree. And then actions can refer to tree nodes with the proper type:

```
e : ID {CommonTree t = $ID.tree;} ;
```

5.10 TokenLabelType Option

By default, ANTLR generates lexers that create CommonToken objects. If you have overridden Lexer method emit() to create special tokens, then you will want to avoid lots of typecasts in your embedded actions by using the **TokenLabelType** option. Here is a simple example:

```
grammar T;
options {TokenLabelType=MyToken;}

e : ID {$ID.methodFromMyToken();} ;
ID: 'a'..'z'+ ;
```

In this case, $ID will evaluate to type MyToken.

5.11 k Option

Use the **k** option to limit ANTLR's grammar analysis to classical *LL(k)* parsing. This option is mostly useful for optimizing the speed of your parser by limiting the amount of lookahead available to the parser. In general, however, using this option will force alterations in the grammar to avoid nondeterminism warnings, which can lead to unnatural grammars.

The argument to the **k** option is either a star or an integer representing the desired fixed-lookahead depth. By default, ANTLR assumes this:

```
grammar T;
options {
    k=*;
}
...
```

This chapter described the various grammar-level options that alter how ANTLR perceives input grammars and how it generates code. The next chapter looks at grammar actions and where ANTLR inserts them in the generated code. Further, it describes how actions can access information about the rule elements matched on the input stream.

Chapter 6

Attributes and Actions

In the previous two chapters, we examined the general structure of ANTLR grammar files and how to properly build rules, but a grammar by itself is not particularly useful. The resulting recognizer can answer only yes or no as to whether an input sentence conforms to the language specified by the grammar. To build something useful such as an interpreter or translator, you must augment grammars with actions. Grammar actions perform computations on the input symbols or emit new output symbols.

This chapter describes the syntax of actions, the significance of their locations, and the manner in which they can access data elements derived from the recognition process. In addition, this chapter defines dynamic scopes that allow distant rules to read and write mutually accessible variables. Dynamic scopes allow rules to communicate without having to define and pass parameters through intermediate rules. In particular, this chapter answers all the following common questions:

- "How can we insert actions among the rule elements to perform a translation or build an interpreter?"
- "How can we access information about the input symbols matched by various rule elements?"
- "How can rules pass data back and forth?"
- "What information about rule references and token references does ANTLR automatically create for us?"

6.1 Introducing Actions, Attributes, and Scopes

You must execute some code to do anything beyond recognizing the syntax of a language. You must, therefore, embed code directly in your grammar as *actions*. Actions usually directly operate on the input symbols, but they can also trigger method calls to appropriate external code.

Actions are blocks of text written in the target language and enclosed in curly braces. The recognizer triggers them according to their locations within the grammar. For example, the following rule emits `found a decl` after the parser has seen a valid declaration:

```
decl: type ID ';' {System.out.println("found a decl");} ;
type: 'int' | 'float' ;
```

The action performs a translation, but it's an uninteresting translation because every declaration results in the same output. To perform a useful translation, actions must refer to the input symbols. Token and rule references both have predefined attributes associated with them that are useful during translation. For example, you can access the text matched for rule elements:

```
decl: type ID ';'
      {System.out.println("var "+$ID.text+":"+$type.text+";");}
    ;
type: 'int' | 'float' ;
```

where `text` is a predefined attribute. `$ID` is a `Token` object, and `$type` is a data aggregate that contains all the predefined properties for that particular reference to rule **type**. Given input `int x;`, the translator emits `var x:int;`.

When references to rule elements are unique, actions can use $*elementName* to access the associated attributes. When there is more than one reference to a rule element, $*elementName* is ambiguous, and you must label the elements to resolve the ambiguity. For example, to allow user-defined types in the language described by rule **decl**, you could add another rule that matches two identifiers in a row. To access both `ID` tokens, you must label them and then refer to their labels in the action:

```
decl: type ID ';'
      {System.out.println("var "+$ID.text+":"+$type.text+";");}
    | t=ID id=ID ';'
      {System.out.println("var "+$id.text+":"+$t.text+";");}
    ;
```

When a rule matches elements repeatedly, translators commonly need to build a list of these elements. As a convenience, ANTLR provides the += label operator that automatically adds all associated elements to an ArrayList, whereas the = label operator always refers to the last element matched. The following variation of rule **decl** captures all identifiers into a list called ids for use by actions:

New in v3.

```
decl: type ids+=ID (',' ids+=ID)* ';' ; // ids is list of ID tokens
```

Beyond these predefined attributes, you can also define your own rule attributes that behave like and are implemented as rule parameters and return values (or inherited and synthesized attributes, as academics refer to them). In the following example, rule **declarator** defines a parameter attribute called typeText that is available to actions as $typeText (see Section 4.3, *Rule Arguments and Return Values*, on page 101 for more about rule parameter and return value definition syntax):

```
decl: type declarator[$type.text] ';' ;
declarator[String typeText]
    : '*' ID {System.out.println("var "+$ID.text+":^"+$typeText+";");}
    | ID     {System.out.println("var "+$ID.text+":"+$typeText+";");}
    ;
```

Rule references use square brackets instead of parentheses to pass parameters. The text inside the square brackets is a comma-separated expression list written in the target language. ANTLR does no interpretation of the action except for the usual attribute reference translation such as $type.text.

Actions can access rule return values just like predefined attributes:

```
field
    : d=decl ';' {System.out.println("type "+$d.type+", vars="+$d.vars);}
    ;
decl returns [String type, List vars]
    : t=type ids+=ID (',' ids+=ID)* {$type = $t.text; $vars = $ids;}
    ;
```

Given input int a, b;, the translator emits type int, vars=[a, b] (the square brackets come from Java's standard List toString()).

All the attributes described thus far are visible only in the rule that defines them or as a return value. In many situations, however, you'll want to access tokens matched previously by other rules. For example, consider a **statement** rule that needs to access the name of the immediately enclosing method.

One solution involves defining an instance variable that is set by the **method** rule and accessed by the **statement** rule:

```
@members {
      String methodName;
}
method: type ID {methodName=$ID.text;} body
    ;
body:   '{' statement+ '}' ;
statement
    :   decl {...methodName...} ';' // ref value set in method
    |   ...
    ;
```

In this way, you can avoid having to define and pass parameters all the way down to rule **statement**. Because this sort of thing is so common, ANTLR formalizes such communication by allowing you to define rule attributes that any rule invoked can access (this notion is technically called *dynamic scoping*). Here is the functionally equivalent version of the earlier grammar using a dynamic rule scope:

```
method
scope {
      String name;
}
    :   type ID {$method::name=$ID.text;} body
    ;
body:   '{' statement+ '}' ;
statement
    :   decl {...$method::name...} ';' // ref value set in method
    |   ...
    ;
```

Note that the $method:: syntax clearly distinguishes a dynamically scoped attribute from a normal rule attribute.

A rule scope acts exactly like a list of local variables that just happens to be visible to rules further down in the call chain. As such, recursive indications of rule **method** each get their own copy of name, which distinguishes it from the first, simpler implementation that used just an instance variable. This feature turns out to be extremely useful because distant rules can communicate without having to define and pass arguments through the intermediate rules; see Section 6.5, *Dynamic Attribute Scopes for Interrule Communication*, on page 148.

The following sections describe all the concepts introduced here in more detail.

6.2 Grammar Actions

Actions are snippets of code that you write in the target language and embed in your grammar. ANTLR then inserts them into the generated recognizer according to their positions relative to the surrounding grammar elements. ANTLR inserts the actions verbatim except for some special expressions prefixed with $ or %.

Actions specified outside rules generally define global or class member program elements such as variables and methods. Most translators need at least a few helper methods and instance variables. Actions embedded within rules define executable statements, and the recognizer executes them as it recognizes input symbols.

Consider the following simple grammar that illustrates most of the locations within a grammar where you can put actions:

```
parser grammar T;

@header {
package p;
}

@members {
int i;
public TParser(TokenStream input, int foo) {
    this(input);
    i = foo;
}
}

a[int x] returns [int y]
@init {int z=0;}
@after {System.out.println("after matching rule; before finally");}
    :   {«action1»} A {«action2»}
    ;
    catch[RecognitionException re] {
        System.err.println("error");
    }
    finally { «do-this-no-matter-what» }
```

ANTLR generates TParser.java from grammar T; the code on the next page shows the surrounding infrastructure for class TParser where the highlighted lines indicate the actions copied from grammar T into the generated recognizer.

Download attributes/actions/TParser.java

```
package p;

import org.antlr.runtime.*;
import java.util.Stack;
import java.util.List;
import java.util.ArrayList;

public class TParser extends Parser {
    public static final String[] tokenNames = new String[] {
        "<invalid>", "<EOR>", "<DOWN>", "<UP>", "A"
    };
    public static final int A=4;
    public static final int EOF=-1;
        public TParser(TokenStream input) {
            super(input);
        }
    public String[] getTokenNames() { return tokenNames; }
    public String getGrammarFileName() { return "T.g"; }

    int i;
    public TParser(TokenStream input, int foo) {
        this(input);
        i = foo;
    }
```

For rule a, ANTLR generates the following method a() definition:

Download attributes/actions/TParser.java

```
    // T.g:15:1: a[int x] returns [int y] : A ;
    public int a(int x) throws RecognitionException {
        int y = 0; // auto-initialized by ANTLR
        int z=0;
        try {
            // T.g:18:4: A
            action1
            match(input,A,FOLLOW_A_in_a40);
            action2
            System.out.println("exiting");
        }
        catch (RecognitionException re) {
            System.err.println("error");
        }
        finally {
             do-this-no-matter-what
        }
        return y;
    }
```

Action placement is fairly straightforward if you view grammars as programs in a very high-level domain-specific language. Actions outside rules become member declarations, and actions with rules are executable statements inserted into the generated rule methods. The next two subsections describe actions and the significance of their locations in more detail.

Using Named Global Actions

ANTLR gives names to all the locations in the generated code that you can fill with user-defined code. The following list summarizes the action names and their purposes:

Improved in v3. Actions are named to indicate where in the output file ANTLR inserts the action.

header

Specify code that should appear before the class definition (this is usually where package definitions and imports go).

members

Specify instance variables and methods.

rulecatch

Replace default **catch** clauses generated for syntax errors with this action. Cf. Chapter 10, *Error Reporting and Recovery*, on page 241.

synpredgate

Replace the expression that gates actions on and off during syntactic predicate evaluation. This indicates when it is OK to execute embedded user actions. By default, embedded actions are executed by the recognizer only when the backtracking depth is 0:

```
if ( backtracking==0 ) { «embedded-user-action» }
```

To change the condition, set the **synpredgate** action:

```
@synpredgate { «ok-to-execute-action-expression» }
```

This replaces the backtracking==0 default expression.

In combined grammars, **header** and **members** actions refer to the parser component only, not the implicitly generated lexer. If you need to override this default, you can prefix these actions with an action scope name. The action scope is one of **lexer**, **parser**, or **treeparser**. It specifies to which recognizer you would like to associate the action. For example, @header is shorthand for @parser::header in a combined grammar. Use @lexer::header to specify packages and other header code needed by the lexer.

The various action names might differ depending on the ANTLR target language, but they all support at least the previously mentioned action names. Also note that some targets might not support the **treeparser** action scope.

Embedding Actions within Rules

Actions are treated like any other rule elements and, as such, can appear just about anywhere among the alternatives of a rule. Actions are executed immediately after the preceding rule element and immediately before the following rule element. For example, in the following rule, an action is executed after each reference to token **ID**:

```
@members {
Map symbols = new HashMap();
}
decl: type a=ID {symbols.put($a,$type.text);}
      (',' b=ID {symbols.put($b,$type.text);} )* ';'
    ;
```

The goal of this code is to map Token objects to the text name of their type, implementing a primitive symbol table. The first action, symbols.put($a,$type.text), is executed only once and immediately after the recognition of the first **ID** token. The second action,symbols.put($b,$type.text), is executed once for every iteration of the loop generated for the EBNF (...)* zero-or-more subrule. Every time through the loop $b points to the most recently matched identifier. Given input float a, b, c;, rule **decl** yields a hashtable where all three Token objects associated with a,b, and c point at the string float.

Because Java allows you to define variables in-line as opposed to the start of a code block, you can define variables inside any action. The variables are visible to the immediately surrounding alternative, but not other alternatives and not surrounding alternatives if the action is within a subrule. Targets such as the C target, however, are limited to defining variables with **init** actions.

Beyond the actions executed among the rule elements, some special actions execute just before the rule starts, just after the rule finishes, and upon syntax error within the rule.

The following example shows the utility of the **init** and **after** actions:

```
@members {
boolean inMethod = false;
}

methodDefinition returns [int numStats]
@init {
inMethod = true;
}
@after {
inMethod = false;
}
    :   ...
    ;
```

Any rule invoked from the **methodDefinition** rule can use the boolean instance variable inMethod to test its context (that is, whether the rule is being matched within the context of a method). This is a good way to distinguish between local variables and fields of a class definition, for example.

ANTLR inserts **init** actions after all definitions and initialization generated for the rule and right before the code generated for the rule body. ANTLR inserts **after** actions after all the rule cleanup code that sets return values and so on. The **after** action can, for example, access the tree computed by the rule. ANTLR inserts **after** actions before the code that cleans up the dynamic scopes used by the rule; see Section 6.5, *Rule Scopes*, on page 150 for an example that illustrates the **after** action's position relative to dynamic scope cleanup code. Subrules can't define **init** or **after** actions.

For completeness, note that rules can also specify actions as part of the exception handlers, but please see Chapter 10, *Error Reporting and Recovery*, on page 241 for more information.

6.3 Token Attributes

All tokens matched by parser and lexer rules have a collection of predefined, read-only attributes. The attributes include useful token properties such as the token type and text matched for a token. Actions can access these attributes via $*label.attribute* where *label* labels a token reference. As shorthand, actions can use $*T.attribute* where *T*

is a unique token reference visible to the action. The following example illustrates token attribute expression syntax:

Improved in v3. Token references now have a rich set of predefined attributes.

```
r   :   INT {int x = $INT.line;}
        ( ID {if ($INT.line == $ID.line) ...;} )?
        a=FLOAT b=FLOAT {if ($a.line == $b.line) ...;}
    ;
```

The action within the (...)? subrule can see the **INT** token matched before it in the outer level.

Because there are two references to the **FLOAT** token, a reference to $FLOAT in an action is not unique; you must use labels to specify which token reference you are interested in.

Token references within different alternatives are unique because only one of them can be matched for any invocation of the rule. For example, in the following rule, actions in both alternatives can reference $ID directly without using a label:

```
r   :   ... ID {System.out.println($ID.text);}
    |   ... ID {System.out.println($ID.text);}
    ;
```

To access the tokens matched for literals, you must use a label:

```
stat:   r='return' expr ';' {System.out.println("line="+$r.line);} ;
```

Most of the time you access the attributes of the token, but sometimes it is useful to access the Token object itself because it aggregates all the attributes. Further, you can use it to test whether an optional subrule matched a token:

```
stat: 'if' expr 'then' stat (el='else' stat)?
      {if ( $el!=null ) System.out.println("found an else");}
    | ...
    ;
```

Figure 6.1, on the next page, summarizes the attributes available for tokens; this includes lexer rule references in a lexer.

For lexer rules, note that labels on elements are sometimes characters, not tokens. Therefore, you can't reference token attributes on all labels. For example, the following rule defines three labels, of which $a and $c are character labels and evaluate to type int, not Token. If the literal is more than a single character like 'hi', then the label is a token reference, not a character reference:

```
lexer grammar T;
R : a='c' b='hi' c=. {$a, $b.text, $c} ;
```

Attribute	Type	Description
text	String	The text matched for the token; translates to a call to getText().
type	int	The token type (nonzero positive integer) of the token such as INT; translates to a call to getType().
line	int	The line number on which the token occurs, counting from 1; translates to a call to getLine().
pos	int	The character position within the line at which the token's first character occurs counting from zero; translates to a call to getCharPositionInLine().
index	int	The overall index of this token in the token stream, counting from zero; translates to a call to getTokenIndex().
channel	int	The token's channel number. The parser tunes to only one channel, effectively ignoring off-channel tokens. The default channel is 0 (Token.DEFAULT_CHANNEL), and the default hidden channel is Token.HIDDEN_CHANNEL.
tree	Object	When building trees, this attribute points to the tree node created for the token; translates to a local variable reference that points to the node, and therefore, this attribute does not live inside the Token object itself.

Figure 6.1: Predefined token and lexer rule attributes

ANTLR generates the following code for the body of the method:

```
// a='c'
int a = input.LA(1);
match('c');
// b='hi'
int bStart = getCharIndex();
match("hi");
Token b = new CommonToken(input, Token.INVALID_TOKEN_TYPE,
    Token.DEFAULT_CHANNEL, bStart, getCharIndex()-1);
// c=.
int c = input.LA(1);
matchAny();
// {$a, $b.text, $c}
a, b.getText(), c
```

6.4 Rule Attributes

When a recognizer matches a rule, it gathers a few useful attributes such as the text matched for the entire rule (**text**), the first symbol matched for the rule (**start**), and the last symbol matched by the rule (**stop**). The recognizer automatically sets the attributes for you. With a few exceptions, you cannot write to these attributes. You can, however, define more attributes in the form of rule parameters and return values. Actions within the rule use them to pass data between rules. User-defined attributes behave exactly like method parameters and return values except that rules can have multiple return values. The following subsections describe the predefined attributes and how to create rule parameters and return values.

Improved in v3. Rules now have a richer set of predefined attributes, and the general attribute mechanism is more consistent in v3.

Predefined Rule Attributes

For translation applications, actions often need to know about the complete input text matched by a rule. When generating trees or templates, actions also need to get the subtree or subtemplate created by a rule invocation. ANTLR predefines a number of read-only attributes associated with rule references that are available to actions. As you might expect, actions can access rule attributes only for references that precede the action. The syntax is $*r.attribute* for rule name *r* or a label assigned to a rule reference. For example, $expr.text returns the complete text matched by a preceding invocation of rule **expr**.

The predefined attributes of the currently executing rule are also available via shorthand: $*attribute* or $*enclosingRuleName.attribute*. Consider the following rule that illustrates where you can put actions:

```
r
@init {
Token myFirstToken = $start; // do me first
}
@after {
Token myLastToken = $r.stop; // do me after rule matches
}
    :   ID {String s = $r.text;} INT {String t = $text;}
    ;
```

In the generated code, you will see that these attribute references translate to field accesses of the retval aggregate. For example, the **init** action is translated as follows:

```
Token myFirstToken = ((Token)retval.start);
```

The recognizer automatically computes the predefined attributes, and your actions should not attempt to modify them. One exception is that your **after** action can set attributes **tree** and **st** when generating ASTs (see Chapter 7, *Tree Construction*, on page 162) or templates (see Chapter 9, *Generating Structured Text with Templates and Grammars*, on page 206). Setting them in any other action has no effect because they are set by the rule's bookkeeping code right before the **after** action.

Figure 6.2, on the following page, describes the predefined attributes that are available to actions.

Predefined Lexer Rule Attributes

Lexer rules always have an implicit return value of type Token that is sent back to the parser. However, lexer rules that refer to other lexer rules can access those portions of the overall token matched by the other rules and returned as implicit tokens. The following rule illustrates a composite lexer rule that reuses another token definition:

```
PREPROC_CMD
    :   '#' ID {System.out.println("cmd="+$ID.text);}
    ;
ID  :   ('a'..'z'|'A'..'Z')+
    ;
```

ANTLR translates rule **PREPROC_CMD**'s production to the following code that creates a temporary token for use by the embedded action (note that $ID.text is automatically translated to ID1.getText()):

```
match('#');
int ID1Start = getCharIndex();
mID();
Token ID1 = new CommonToken(
    input, ID, Token.DEFAULT_CHANNEL, ID1Start, getCharIndex()-1);
System.out.println("cmd="+ID1.getText());
```

The attributes of a lexer rule reference are the same as a token reference in a parser grammar (see Section 6.3, *Token Attributes*, on page 138). The only exception is that **index** is undefined. The lexer does not know where in the token stream the token will eventually appear. Figure 6.3, on page 144 summarizes the attributes available to attribute expressions referring to the surrounding rule.

Attribute	Type	Description
text	String	The text matched from the start of the rule up until the point of the $text expression evaluation. Note that this includes the text for all tokens including those on hidden channels, which is what you want because usually that has all the whitespace and comments. When referring to the current rule, this attribute is available in any action including any exception actions.
start	Token	The first token to be potentially matched by the rule that is on the main token channel; in other words, this attribute is never a hidden token. For rules that end up matching no tokens, this attribute points at the first token that could have been matched by this rule. When referring to the current rule, this attribute is available to any action within the rule.
stop	Token	The last nonhidden channel token to be matched by the rule. When referring to the current rule, this attribute is available only to the **after** action.
tree	Object	The AST computed for this rule, normally the result of a -> rewrite rule. When referring to the current rule, this attribute is available only to the **after** action.
st	StringTemplate	The template computed for this rule, usually the result of a -> rewrite rule. When referring to the current rule, this attribute is available only to the **after** action.

Figure 6.2: Predefined parser rule attributes

Attribute	Type	Description
text	String	The text matched thus far from the start of the token at the outermost rule nesting level; translated to getText().
type	int	The token type of the surrounding rule, even if this rule does not emit a token (because it is invoked from another rule).
line	int	The line number, counting from 1, of this rule's first character.
pos	int	The character position in the line, counting from 0, of this rule's first character.
channel	int	The default channel number, 0, unless you set it in an action in this rule.

Figure 6.3: Lexer rule attributes available to expressions

Predefined Tree Grammar Rule Attributes

Tree grammar rules have the same attributes as parser grammar rules except that the input symbols are tree nodes instead of tokens and **stop** is not defined. So, for example, $ID refers to the tree node matched for token **ID** rather than a Token object. Figure 6.4, on the next page, summarizes the predefined attributes for tree grammar rules.

Rule Parameters

Besides the predefined attributes, rules can define user-defined attributes in the form of parameters whose values are set by invoking rules. For example, the following rule has a single parameter that the embedded action uses to generate output:

```
declarator[String typeName]
    :   ID {System.out.println($ID.text+" has type "+$typeName);}
    ;
```

Rule parameters often contain information about the rule's context. The rule can use that information to guide the parse.

Attribute	Type	Description
text	String	The text derived from the first node matched by this rule. Each tree node knows the range of input tokens from which it was created. Parsers automatically set this range to the first and last token matched by the rule that created the tree (see Section 7.3, *Default AST Construction*, on page 170). This attribute includes the text for all tokens including those on hidden channels, which is what you want because usually that has all the whitespace and comments. When referring to the current rule, this attribute is available in any action including exception actions. Note that **text** is not well defined for rules like this: `slist : stat+ ;` because **stat** is not a single node or rooted with a single node. $slist.text gets only the first **stat** tree.
start	Object	The first tree node to be potentially matched by the rule. For rules that end up matching no nodes, this attribute points at the first node that could have been matched by this rule. When referring to the current rule, this attribute is available to any action within the rule.
st	StringTemplate	The template computed for this rule, usually the result of a -> rewrite rule. When referring to the current rule, this attribute is available only to the **after** action.

Figure 6.4: Predefined attributes for tree grammar rules

To illustrate this, consider the following rule with parameter needBody that indicates whether a body is expected after the method header:

```
methodDefinition[boolean needBody]
    :   modifiers typename ID '(' formalArgs ')'
        ( {$needBody}?=>  body
        | {!$needBody}?=> ';' // abstract method
        )
    ;
```

where {$needBody}?=> is a gated semantic predicates that turns on the associated alternative according to the value of parameter needBody.

ANTLR generates the following method structure for rule **methodDefinition**:

```
public void methodDefinition(boolean needBody)
    throws RecognitionException {
    ...
}
```

These parameter attributes are visible to the entire rule including exception, **init**, and **after** actions.

In lexers, only **fragment** rules can have parameters because they are the only rules you can explicitly invoke in the lexer. Non-**fragment** rules are implicitly invoked from the automatically generated **nextToken** rule.

Use square brackets to surround parameter lists. For example, here is a rule that invokes **methodDefinition**:

```
classDefinition
    :   methodDefinition[true]
    |   fieldDefinition
    ;
```

Although it is probably not very good form, you are able to set parameter values in actions. Naturally, actions can't access the parameters of rule references; here, $methodDefinition.needBody makes no sense.

Rule Return Values

Rules can define user-defined attributes in the form of rule return values. Actions access these return values via rule label properties.

For example, the following grammar defines rule **field** that invokes rule **decl** and accesses its return values:

Improved in v3. You can now return multiple return values from a rule.

```
field
    :   d=decl {System.out.println("type "+$d.type+", vars="+$d.vars);}
    ;
/** Compute and return a list of variables and their type. */
decl returns [String type, List vars]
    :   t=type ids+=ID (',' ids+=ID)* ';'
        {$type = $t.text; $vars = $ids;}
    ;
```

ANTLR generates the following (slightly cleaned up) code for rule **field**:

```
public void field() throws RecognitionException {
    decl_return d = null;
    try {
        d=decl();
        System.out.println("type "+d.type+", vars="+d.vars);
    }
    catch (RecognitionException re) {
        reportError(re);
        recover(input,re);
    }
}
```

and generates a return value structure to represent the multiple return values of rule **decl**:

```
public static class decl_return extends RuleReturnScope {
    public String type;
    public List vars;
};
```

Rule **decl** then creates a new instance of this aggregate to hold the return values temporarily:

```
public decl_return decl() throws RecognitionException {
    // create return value data aggregate
    decl_return retval = new decl_return();
    retval.start = input.LT(1);
    try {
        ...
        // set $stop to previous token (last symbol matched by rule)
        retval.stop = input.LT(-1);
        retval.type = input.toString(t.start,t.stop); // $type = $t.text
        retval.vars = list_ids; // $vars = $ids
    }
    catch (RecognitionException re) {
        reportError(re);
        recover(input,re);
    }
    return retval;
}
```

The highlighted lines are generated directly or indirectly from the embedded action and use of the += label operator.

In the future, expect ANTLR to optimize the object creation away if only one return value is used or set.

Actions within **decl** that access return values translate to field references of this data aggregate; they can read and write these values. Invoking rules can't set return values. Here, executing $d.type="int"; from within rule **field** makes no sense.

6.5 Dynamic Attribute Scopes for Interrule Communication

Rule attributes have very limited visibility—rule r can pass information (via parameters) only to rule s if r directly invokes s. Similarly, s can pass information back to only that rule that directly invokes s (via return values). This is analogous to the normal programming language functionality whereby methods communicate directly through parameters and return values. For example, the following reference to local variable x from a deeply nested method call is illegal in Java:

New in v3.

```
void f() {
    int x = 0;
    g();
}
void g() {
    h();
}
void h() {
    int y = x; // INVALID reference to f's local variable x
}
```

Variable x is available only within the scope of f(), which is the text lexically delimited by curly brackets. For this reason, Java is said to use *lexical scoping*. Lexical scoping is the norm for most programming languages.[1] Languages that allow methods further down in the call chain to access local variables defined earlier are said to use *dynamic scoping*. The term *dynamic* refers to the fact that a compiler cannot statically determine the set of visible variables. This is because the set of variables visible to a method changes depending on who calls that method. For general programming languages, I am opposed to dynamic scoping, as are most people, because it is often difficult to decide which variable you are actually accessing. Languages such as Lisp and Perl have been criticized for dynamic scoping.

New in v3.

1. See http://en.wikipedia.org/wiki/Scope_(programming)#Static_scoping.

In a general-purpose language like Java, to allow h() to access x, either you have to pass x all the way down as a parameter, which is very inconvenient, or you can define an instance variable that f() and h() can both access:

```
int x;
void f() {
    x = 0; // same as this.x = 0
    g();
}
void g() {
    h();
}
void h() {
    int y = x; // no problem, x is this.x
}
```

It turns out that, in the grammar realm, distant rules often need to communicate with each other, mostly to provide context information to rules matched below in the rule invocation chain (that is, below in the parse tree). For example, an expression might want to know whether an identifier was previously defined. An expression might want to know whether it is an assignment right-side or a loop conditional. For the first problem, a symbol table instance variable works nicely, but for the latter problem, it is bad "information hiding" programming practice to define a bunch of instance variables visible to all rules just to provide context to a specific few. Further, the definition of the context variable is often far from the rule that sets its value.

Recognizing the importance of context in language translation problems, ANTLR gives you both locality of definition and wide visibility through dynamic scoping.[2] ANTLR allows you to define rule attributes that look like local variables but that are visible to any rule invoked from that rule no matter how deeply nested. The following grammar is functionally equivalent to the earlier Java code:

New in v3.

```
f
scope {int x;}
    : {$f::x = 0;} g
    ;
g : h ;
h : {int y = $f::x;} ;
```

Here, $f::x accesses the dynamically scoped attribute x defined in rule **f**.

2. See http://en.wikipedia.org/wiki/Scope_(programming)#Dynamic_scoping.

In contrast to unrestricted dynamic scoping, all references to dynamically scoped attributes must include the scope containing the definition. This syntax neatly sidesteps the primary concern that it is difficult to decide which dynamically scoped variable is being accessed. The $f:: prefix on dynamically scoped references also highlights that it is a nonlocal reference.

Rule Scopes

To define a dynamic scope of attributes in a rule, specify a list of variable definitions without initialization values inside a **scope** action:

```
r
scope {
    «attribute1»;
    «attribute2»;
    ...
    «attributeN»;
}
    : ...
    ;
```

To illustrate the use of dynamic scopes, consider the real problem of defining variables and ensuring that variables in expressions are defined. The following grammar defines the symbols attribute where it belongs in the **block** rule but adds variable names to it in rule **decl**. Rule **expr** then consults the list to see whether variables have been defined.

Download attributes/rulescope/T.g

```
grammar T;

prog:   block
    ;

block
scope {
    /** List of symbols defined within this block */
    List symbols;
}
@init {
    // initialize symbol list
    $block::symbols = new ArrayList();
}
    :   '{' decl* stat+ '}'
        // print out all symbols found in block
        // $block::symbols evaluates to a List as defined in scope
        {System.out.println("symbols="+$block::symbols);}
    ;
```

```
/** Match a declaration and add identifier name to list of symbols */
decl:   'int' ID {$block::symbols.add($ID.text);} ';'
    ;

/** Match an assignment then test list of symbols to verify
 *  that it contains the variable on the left side of the assignment.
 *  Method contains() is List.contains() because $block::symbols
 *  is a List.
 */
stat:   ID '=' INT ';'
        {
        if ( !$block::symbols.contains($ID.text) ) {
            System.err.println("undefined variable: "+$ID.text);
        }
        }
    ;
ID  :   'a'..'z'+ ;
INT :   '0'..'9'+ ;
WS  :   (' '|'\n'|'\r')+ {$channel = HIDDEN;} ;
```

Here's the given input:

```
{
  int i;
  int j;
  i = 0;
}
```

The translator emits the following:

```
symbols=[i, j]
```

Given this input:

Download attributes/rulescope-recursive/input2

```
{
  int i;
  int j;
  i = 0;
  x = 4;
}
```

the translator gives an error message about the undefined variable reference:

```
undefined variable: x
symbols=[i, j]
```

In this example, defining symbols as an instance variable would also work, but it is not the same solution as using a dynamically scoped variable. Dynamically scoped variables act more like a local variable.

There is a copy of symbols for each invocation of rule **block**, whereas there is only one copy of an instance variable per grammar. This might not be a problem in general, but having only a single copy of symbols would not work if **block** were invoked recursively. For nested blocks, you want each block to have its own list of variable definitions so that you can reuse the same variable name in an inner block. For example, the following nested code block redefines i in the inner scope. This new definition must hide the definition in the outer scope.

Download attributes/rulescope-recursive/input

```
{
  int i;
  int j;
  i = 0;
  {
    int i;
    int x;
    x = 5;
  }
  x = 3;
}
```

The following modified grammar supports nested code blocks by allowing rule **stat** to recursively invoke **block**. The invocation of **block** automatically creates an entirely new copy of the symbols attribute. The invocation still pushes symbols on the same stack, though, so that it maintains a single stack of all scopes.

Download attributes/rulescope-recursive/T.g

```
grammar T;

@members {
/** Track the nesting level for better messages */
int level=0;
}

prog:   block
    ;
block
scope {
    List symbols;
}
@init {
    $block::symbols = new ArrayList();
    level++;
}
```

```
@after {
    System.out.println("symbols level "+level+" = "+$block::symbols);
    level--;
}
    :   '{' decl* stat+ '}'
    ;
decl:   'int' ID {$block::symbols.add($ID.text);} ';'
    ;
stat:   ID '=' INT ';'
        {
        if ( !$block::symbols.contains($ID.text) ) {
            System.err.println("undefined variable level "+level+
                               ": "+$ID.text);
        }
        }
    |   block
    ;
ID  :   'a'..'z'+ ;
INT :   '0'..'9'+ ;
WS  :   (' '|'\n'|'\r')+ {$channel = HIDDEN;} ;
```

Instance variable level tracks the recursion level of rule **block** so that the print action can identify the level in which the symbols are defined. The print action is also now in an **after** action for symmetry with the **init** action. For the earlier input, the translator emits the following:

```
symbols level 2 = [i, x]
undefined variable level 1: x
symbols level 1 = [i, j]
```

Note that the undefined variable message is still there because the list of symbols for the inner block disappears just like a local variable after the invocation of that recursive invocation of rule **block**.

This example is a nice illustration of dynamically scoped variables, but for completeness, the check for undefined variables should really look at all scopes on the stack rather than just the current scope. For example, a reference to j within the nested code block yields an undefined variable error. The action needs to walk backward up the stack of symbols attributes until it finds a scope containing variable reference. If the search reaches the bottom of the stack, then the variable is undefined in any scope.

To access elements other than the top of the dynamically scoped attribute stack, use syntax $x[i]::y where x is the scope name, y is the attribute name, and i is the absolute index into the stack with 0 being the bottom of the stack. Expression $x.size()-1 is the index of the top of the stack. The following method, defined in the **members** action, com-

putes whether a variable is defined in the current scope or any earlier scope:

```
boolean isDefined(String id) {
    for (int s=level-1; s>=0; s--) {
        if ( $block[s]::symbols.contains(id) ) {
            System.out.println(id+" found in nesting level "+(s+1));
            return true;
        }
    }
    return false;
}
```

Then rule **stat** should reference the method to properly identify undefined variables:

```
stat:   ID '=' INT ';'
        {
        if ( !isDefined($ID.text) ) {
            System.err.println("undefined variable level "+level+
                               ": "+$ID.text);
        }
        }
    |   block
    ;
```

Given the following input:

Download attributes/rulescope-resolve/input

```
{
  int i;
  int j;
  {
    int i;
    int x;
    x = 5;
    i = 9;
    j = 4;
  }
  x = 3;
}
```

the program emits this:

```
x found in nesting level 2
i found in nesting level 2
j found in nesting level 1
symbols level 2 = [i, x]
undefined variable level 1: x
symbols level 1 = [i, j]
```

The next section describes how to handle the case where multiple rules need to share dynamically scoped attributes.

Global Scopes

There is a separate attribute stack for each rule that defines a dynamic scope. In the previous section, only rule **block** defined a scope; therefore, there was only one stack of scopes. Each invocation of **block** pushed a new symbols list onto the stack. The isDefined() method walked up the single stack trying to resolve variable references. This simple technique works because there is only one kind of variable definition scope in the language described by the grammar: code blocks enclosed in curly braces.

Consider the scoping rules of the C programming language. Ignoring **struct** definitions and parameters for simplicity, there are a global scope, function scopes, and nested code blocks. The following extension to the previous grammar matches a language with the flavor of C:

```
prog:   decl* func*
    ;
func:   'void' ID '(' ')' '{' decl* stat+ '}'
    ;
...
```

This deceptively simple grammar matches input such as the following:

Download attributes/globalscope/input

```
int i;
void f()
{
  int i;
  {
    int i;
    i = 2;
  }
  i = 1;
}
void g() {
  i = 0;
}
```

Variable i is defined in three different nested scopes: at the global level, in function f(), and in a nested code block. When trying to resolve references to i, you need to look in the closest enclosing scope and then in outer scopes. For example, the reference to i in i=1; should be resolved in f()'s scope rather than the preceding nested code block. The reference to i in g() should resolve to the global definition.

Your first attempt at using dynamic scopes to solve this problem might involve defining a dynamic scope in each rule that matches declarations:

```
prog
scope {
    List symbols;
}
    :   decl* func*
    ;
func
scope {
    List symbols;
}
    :   'void' ID '(' ')' '{' decl* stat+ '}'
    ;
block
scope {
    List symbols;
}
    :   '{' decl* stat+ '}'
    ;
...
```

The problem with this solution is that there are three different stacks, one for each C language scope. To properly resolve variable references, however, you must maintain a single stack of variable scopes (as is clear by examining method isDefined()).

ANTLR provides a mechanism for multiple rules to share the same dynamic scope stack. Simply define a named scope outside any rule. In this case, define CScope with two attributes:

*In the future, expect ANTLR to allow initialization code within **scope** definitions such as constructors.*

Download attributes/globalscope/T.g

```
// rules prog, func, and block share the same global scope
// and, therefore, push their scopes onto the same stack
// of scopes as you would expect for C (a code block's
// scope hides the function scope, which in turn, hides
// the global scope).
scope CScope {
    String name;
    List symbols;
}
```

where attribute name records the name of the scope such as “global” or the associated function name.

Method isDefined() can then print a more specific name rather than just the level number:

Download attributes/globalscope/T.g

```
@members {
/** Is id defined in a CScope?  Walk from top of stack
 *  downwards looking for a symbols list containing id.
 */
boolean isDefined(String id) {
    for (int s=$CScope.size()-1; s>=0; s--) {
        if ( $CScope[s]::symbols.contains(id) ) {
            System.out.println(id+" found in "+$CScope[s]::name);
            return true;
        }
    }
    return false;
}
}
```

Instead of manually tracking the scope level, you can use $CScope.size()-1 as isDefined() does.

The rules that define variables within a C scope (**prog**, **func**, and **block**) must indicate that they want to share this scope by specifying scope CScope;. The following rules properly use a global dynamic scope to share a single stack of attributes:

Download attributes/globalscope/T.g

```
prog
scope CScope;
@init {
    // initialize a scope for overall C program
    $CScope::symbols = new ArrayList();
    $CScope::name = "global";
}
@after {
    // dump global symbols after matching entire program
    System.out.println("global symbols = "+$CScope::symbols);
}
    :   decl* func*
    ;

func
scope CScope;
@init {
    // initialize a scope for this function
    $CScope::symbols = new ArrayList();
}
```

```
@after {
    // dump variables defined within the function itself
    System.out.println("function "+$CScope::name+" symbols = "+
                       $CScope::symbols);
}
    :   'void' ID {$CScope::name=$ID.text;} '(' ')' '{' decl* stat+ '}'
    ;

block
scope CScope;
@init {
    // initialize a scope for this code block
    $CScope::symbols = new ArrayList();
    $CScope::name = "level "+$CScope.size();
}
@after {
    // dump variables defined within this code block
    System.out.println("code block level "+$CScope.size()+" = "+
                       $CScope::symbols);
}
    :   '{' decl* stat+ '}'
    ;
```

These rules push a new attribute scope onto the stack (identified by $CScope) upon invocation and pop the scope off upon returning. For example, here is the code generated for the global scope, the stack of scopes, and the **prog** rule:

```
/** Put all elements within a scope into a class */
protected static class CScope_scope {
    String name;
    List symbols;
}
/** The stack of scopes; each element is of type CScope_scope */
protected Stack CScope_stack = new Stack();

/** Code generated for rule prog */
public void prog() throws RecognitionException {
    CScope_stack.push(new CScope_scope());
    ((CScope_scope)CScope_stack.peek()).symbols = new ArrayList();
    ((CScope_scope)CScope_stack.peek()).name = "global";
    try {
        ...
        // from @after action
        System.out.println("global symbols = "+
            ((CScope_scope)CScope_stack.peek()).symbols);
        CScope_stack.pop();
    }
}
```

where the highlighted code is the **init** action from rule **prog**. **init** actions are executed after the new scope becomes available, and **after** actions are executed right before the scope disappears.

The global dynamic scope mechanism works well for many simple symbol table implementation such as this, but in general, a more sophisticated symbol table is required for real applications (for example, symbol table scopes generally must persist beyond parser completion).[3]

The following, final section summarizes all the special symbols related to attributes and scopes that you can reference within actions.

6.6 References to Attributes within Actions

ANTLR ignores everything inside user-defined actions except for expressions beginning with $ and %. The list that follows summarizes the special symbols and expressions that ANTLR recognizes and translates to code in the target language. % references are template expressions and are described in Section 9.9, *References to Template Expressions within Actions*, on page 238.

$*tokenRef*

An expression of type Token that points at the Token object matched by the indicated grammar token reference, which is identified either by a token label or a reference to a token name mentioned in the rule. This is useful to test whether a token was matched in an optional subrule also. Example: `ID {$ID} (ELSE stat)? {if ($ELSE!=null) ...}`

$*tokenRef.attr*

Refers to the predefined token attribute *attr* of the referenced token, identified either by a token label or by a reference to a token name mentioned in the rule. See Figure 6.1, on page 140 for the list of valid attributes. Example: `id=ID {$id.text} INT {$INT.line}`

$*listLabel*

An expression that evaluates to type List and is a list of all elements collected thus far by the *listLabel*. A list label is identified by labels using the += operator; this is valid only within a parser or tree grammar. Example: `ids+=ID (',' ids+=ID)* {$ids}`

3. See http://www.cs.usfca.edu/~parrt/course/652/lectures/symtab.html for more information about symbol tables.

$*ruleRef*

Isolated $*rulename* is not allowed in a parser or tree grammar unless the rule has a dynamic scope and there is no reference to *rulename* in the enclosing alternative, which would be ambiguous. The expression is of type Stack. Example (checks how deeply nested rule **block** is): $block.size()

$*ruleRef.attr*

Refers to the predefined or user-defined attribute of the referenced rule, which is identified either by a rule label or by a reference to a rule mentioned in the rule. See Figure 6.2, on page 143 for the list of available rule attributes. Example: e=expr {$e.value, $expr.tree}

$*lexerRuleRef*

Within a lexer, this is an expression of type Token that contains all the predefined properties of the token except the token stream index matched by invoking the lexer rule. The lexer rule reference can be either a rule label or a reference to a lexer rule mentioned within the rule. As with token references, you can refer to predefined attributes of the returns token. Example: (DIGIT {$DIGIT, $DIGIT.text})+

$*attr*

attr is a return value, parameter, or predefined rule property of the enclosing rule. Example: r[int x] returns [Token t]: {$t=$start; $x} ;

$*enclosingRule.attr*

The fully qualified name of a return value, parameter, or predefined property. Example: r[int x] returns [Token t]: {$r.t=$r.start; $r.x;} ;

$*globalScopeName*

An isolated global dynamic scope reference. This is useful when code needs to walk the stack of scopes or check its size. Example: $symbols.size()

$*x*::*y*

Refer to the *y* attribute within the dynamic scope identified by *x*, which can either be a rule scope or be a global scope. In all cases, the scope prefix is required when referencing a dynamic attribute. Example: $CScope::symbols

$*x*[-1]::*y*

Attribute *y* (just under top of stack) of the previous *x* scope. Example: $block[-1]::symbols

$x[-i]::y
: Attribute y of a previous scope of x. The previous scope is i down from the top of stack. The minus sign must be present; that is, i cannot simply be negative. You must use the minus sign! Example: $block[-level]::symbols

$x[i]::y
: Attribute y of a the scope of x up from the bottom of the stack. In other words, i is an absolute index in the range 0..size-1. Example: $block[2]::symbols

$x[0]::y
: Attribute y of a bottommost scope of x. Example: $block[0]::symbols

In general, most attributes are writable, but attributes of token and rule references are read-only. Further, predefined rule attributes other than **tree** and **st** are read-only. The following example illustrates the writable and readable attributes:

```
r   :   s[42]
        {$s.x}            // INVALID: cannot access parameter
        {$s.y=3;}         // INVALID: cannot set return value
        {$r.text="ick";}  // INVALID: cannot set predefined attribute
        ID
        {$ID.text="ick";} // INVALID: cannot set predefined attribute
    ;
s[int x] returns [int y, int z]
@init {$y=0; $z=0;}
    :   {$y = $x*2;} {Token t = $start;} {$tree = ...;}
        {$start = ...;}   // INVALID: cannot set predefined attribute
    ;
```

This chapter described embedded actions and attributes in detail. Together with Chapter 4, *ANTLR Grammars*, on page 86, you've now learned all the basic building blocks for constructing grammars. Most language applications need more infrastructure from ANTLR, however, than simple actions. Complicated translators typically involve multiple passes over the input. A parser grammar builds an intermediate-form tree, and then one or more tree grammars walk that tree. In a final stage, a tree grammar emits output via template construction rules. In the next three chapters, we'll examine how to construct a tree, how to walk a tree via tree grammars, and how to emit output via templates.

Chapter 7

Tree Construction

Complex problems are much easier to solve when you break them down into several, smaller problems. This is particularly true when building language translators. Translators usually have logically separate phases that execute one after the other like the stages in a processor's pipeline. It is just too difficult to translate one programming language to another in one step, for example. Each phase computes some information, fills a data structure, or emits output.

In fact, many language problems cannot even be solved with a single pass. Resolving references to programming language variables defined further ahead is the most obvious example. A translator must walk the input program once to get variable definitions and a second time to resolve references to those variables. Rather than repeatedly rescanning the characters and reparsing the token stream, it is much more efficient to construct and walk a condensed version of the input. This condensed version is called an *intermediate form* and is usually some kind of tree data structure. The tree not only records the input symbols, but it also records the structure used to match them.

Encoding structure in the intermediate-form tree makes walking it much easier and faster than scanning a linear list of symbols as a parser does. Figuring out that 3+4 is an expression from token stream INT + INT is much harder for a computer than looking at a tree node that explicitly says "Hi, I'm an addition expression with two operands." The most convenient way to encode input structure is with a special tree called an *abstract syntax tree* (AST). ASTs contain only those nodes associated with input symbols and are, therefore, not parse trees.

Parse trees also record input structure, but they have nodes for all rule references used to recognize the input. Parse trees are much bigger and highly sensitive to changes to the parser grammar.

Translators pass an AST between phases, and consequently, all of the phases following the AST-building parser are tree walkers. These phases can alter or simply extract information from the AST. For example, the first tree-walking phase might update a symbol table to record variable and method definitions. The next phase might alter the AST so that nodes created from variable and method references point to their symbol table entries. The final phase typically emits output using all the information collected during previous phases.

This chapter describes how to structure ASTs and then defines the tree node types ANTLR can deal with. Once you are familiar with those details, you need to learn how to use AST operators, AST rewrite rules, and actions within your parser grammar to build trees. In the next chapter, we'll build tree-walking translator phases using tree grammars.

7.1 Proper AST Structure

Before learning to build ASTs, let's consider what ASTs should look like for various input structures. Keep in mind the following primary goals as you read this section and when you design ASTs in general. ASTs should do the following:

- Record the meaningful input tokens (and only the meaningful tokens)
- Encode, in the two-dimensional structure of the tree, the grammatical structure used by the parser to match the associated tokens but not the superfluous rule names themselves
- Be easy for the computer to recognize and navigate

These goals give general guidance but do not directly dictate tree structure. For that, think about how computers deal with programs most naturally: as simple instruction streams. The next two sections describe how to break high-level programming language constructs into subtrees with simple instructions as root nodes.

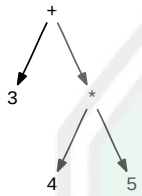
Encoding Arithmetic Expressions

Although humans prefer to think of arithmetic expressions using notation such as 3+4*5, computers prefer to think about the canonical operations needed to compute the result. The following pseudomachine instruction sequence multiplies 4 and 5 (storing the result in register r1) and then adds 3 to r1 (storing the result in register r2):

```
mul 4, 5, r1
add 3, r1, r2
```

Compilers do not generate such machine instructions directly from the input symbols—that would be way too complex to implement in one step. Compilers and other translators break down such difficult translations into multiple steps. The first step is to create a tree intermediate representation that is somewhere between source code and machine code in precision. Trees are much more convenient to examine and manipulate than low-level machine code. Here is what the typical AST looks like for 3+4*5:

Think of that machine instruction sequence as a "dismembered" tree with r1 as a symbolic reference to another subtree.



The structure of this tree dictates the order of operations because you cannot compute the addition without knowing both operands. The right operand is itself a multiply operation, and therefore, you must compute the multiply first.

Working backward from the machine instructions toward the AST, substitute the multiply instruction for r1 in the add (second) operation, and replace the instruction names with the operators:

```
+ 3, (* 4, 5), r2
```

If you consider the operators to be subtree roots, you get the following AST using ANTLR tree notation (the tree itself represents r2, so the r2 reference on the end is unnecessary):

```
^('+' 3 ^('*' 4 5))
```

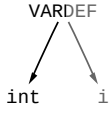
This notation is merely a text representation of the previous AST image and has the following form:

```
^(root child1 child2 ... childN)
```

Most people have no problem agreeing that this AST is an acceptable way to represent the expression in memory, but what about more abstract concepts such as variable definitions like `int i;`?

Encoding Abstract Instructions

To deal with language constructs that are more abstract than expressions, we need to invent some high-level pseudomachine instructions that represent the operation implied by the source language construct. Each subtree should be like an imperative command in English with a verb and object such as "define an integer variable `i`." Always think of subtrees as operations with operands that tell the computer precisely what to do, just like we saw with the expression AST above. The following AST is one possible representation of the variable definition:

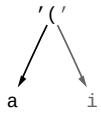


The VARDEF root node is an *imaginary node*, a node containing a token type for which there is no corresponding input symbol. `int i;` is implicitly a definition operation by its grammatical structure rather than explicitly because of an input symbol such as plus or multiply. For language constructs more abstract than expressions and statements, expect to invent pseudo-operations and the associated imaginary nodes.

Source code is meant for human consumption; hence, a single source-level construct often represents multiple lower-level operations. Consider a variation on the variable definition syntax that allows you to define multiple variables without having to repeat the type: `int i,j;`. You must unravel this into two operations when building an AST where the AST does not include the comma and semicolon.

The punctuation characters are not included because they only exist to make the syntax clear to the parser (that is, to make a programmer's intentions clear). Once the parser has identified the structure, though, the punctuation symbols are superfluous and should not be included in the tree.

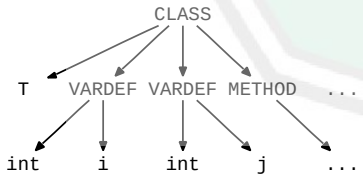
The VARDEF subtrees are in keeping with the third goal, that of creating trees that are easy to process. "Easy to process" usually means the node has an obvious operation and structure. To illustrate this, consider that Fortran syntax, a(i), is identical for both array indexes and function calls. Because the syntax alone does not dictate the operation, you will typically define semantic predicates in your parser grammar that query the symbol table to distinguish between the two based upon the type of the identifier (array or function name). The identifier type dictates the operation. Once you have discovered this information, do not throw it away. Use it to create different AST operation nodes. For example, the following AST does not satisfy the third goal because the operation is not obvious—the operation is still ambiguous.



The '(' token also just does not make much sense as an operator. Instead, build an AST that makes the operation clear such as an array index:

or a function call:

Besides nodes created from tokens and imaginary root nodes, you will also create lists of nodes and subtrees. Usually there is some obvious root node to which you should add the list elements as children. For example, the earlier variable definitions might be children of a class definition root node that also groups together method definitions:



Using ANTLR's tree description language, that tree is written like this:

```
^( CLASS ID ^(VARDEF int i) ^(VARDEF int j) ^(METHOD ...) ... )
```

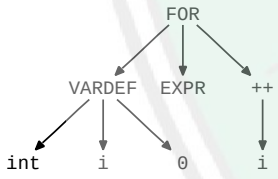
During construction, you will often have a rule that creates a list of subtrees. Until a valid root node can take on the list elements as children, you must use a special "nil operation" root node (trees must always have a single root node). Consider the variable definition subtrees created by, say, rule **definitions**. This rule would simulate a list using a tree with a nil root:

```
^( nil ^(VARDEF int i) ^(VARDEF int j) )
```

There is another situation in which you will want to create an imaginary node: to represent optional but missing subtrees. In general, you should not put something into the AST unless there is a corresponding input construct, but sometimes you must add extra nodes in order to remove ambiguity; remember, one of your goals is to make the subtrees easy to recognize during subsequent phases. Consider the **for** loop in Java, which has the following basic form:

```
forStat
    : 'for' '(' declaration? ';' expression? ';' expression? ')'
      slist
    ;
```

Because every element is optional, you might be tempted to leave missing expressions out of the AST. Unfortunately, this makes the AST ambiguous. If the tree has only one expression child, which is it? The conditional or the update expression? Instead, you need to leave a marker representing a missing expression (or variable definition). For the input `for (int i=0; ; i++) {...}`, create a tree that looks like the following where an imaginary node, EXPR, represents the missing conditional expression:



Similarly, if the initializer were missing from the variable definition, you could include an EXPR node on the end in its place. In this case, however, the optional initializer expression is last, and simply leaving it off would not be a problem.

You might also consider adding an EXPR node on top of all expressions because it might be a good place to squirrel away computation results or other information about the expression after it.

In summary, your goal is to identify and break apart the input constructs into simple operations and encode them in the AST. Your AST will contain nodes created from tokens, nodes with imaginary token types, subtrees with a root and children, and lists of these. Subtree roots represent operations, real or abstract, and subtree children represent operands. Do not include syntactic sugar tokens, such as semicolons, from the source language in the AST.

Now that you know something about what AST structures look like, let's examine the contents and types of the nodes themselves and how they are assembled into trees.

7.2 Implementing Abstract Syntax Trees

ANTLR assumes nothing about the actual Java type and implementation of your AST nodes and tree structure but requires that you specify or implement a TreeAdaptor. The adaptor plays the role of both factory and tree navigator. ANTLR-generated parsers can build trees, and tree parsers can walk them through the use of a single adapter object. Because ANTLR assumes tree nodes are of type Object, you could even make your Token objects double as AST nodes, thus avoiding a second allocation for the tree nodes associated with tokens.

Improved in v3. ANTLR v2 used cumbersome child-sibling trees for memory efficiency reasons, but v3 uses a simpler "list of children" approach.

ANTLR allows you to define your own tree nodes but provides a default tree node implementation, CommonTree, that is useful in most situations. Each node has a list of children and a payload consisting of the token from which the node was created.

For imaginary nodes, the parser creates a CommonToken object using the imaginary token type and uses that as the payload. CommonTreeAdaptor creates CommonTree objects, which satisfy interface Tree. Tree has, among a few other things, the ability to add and get children:

```
/** Get ith child indexed from 0..getChildCount()-1 */
Tree getChild(int i);

/** How many children does this node have? */
int getChildCount();
```

```
/** Add t as a child to this node.  If t is null, do nothing.  If t
 *  is nil, add all children of t to this node's children.
 */
void addChild(Tree t);

/** Indicates the node is a nil node but may still have children,
 *  meaning the tree is a flat list.
 */
boolean isNil();
```

The generic Tree functionality of CommonTree that has nothing to do with payload is contained in BaseTree, a Tree implementation with no user data. CommonTreeAdaptor works with any tree node that implements Tree. BaseTree is useful if you want the core tree functionality but with a payload other than the Token.

An example makes the use of a tree adapter clearer. The following simple program creates a list of identifiers via create() and prints it via toStringTree(). The root node of a list is a "nil" node, which the adapter creates using nil().

Download trees/XYZList.java

```
import org.antlr.runtime.tree.*;

public class XYZList {
    public static void main(String[] args) {
        int ID = 1; // define a fictional token type
        // create a tree adaptor to use for tree construction
        TreeAdaptor adaptor = new CommonTreeAdaptor();
        // a list has no root, creating a nil node
        CommonTree list = (CommonTree)adaptor.nil();
        // create a Token with type ID, text "x" then use as payload
        // in AST node; this variation on create does both.
        list.addChild((CommonTree)adaptor.create(ID,"x"));
        list.addChild((CommonTree)adaptor.create(ID,"y"));
        list.addChild((CommonTree)adaptor.create(ID,"z"));
        // recursively print the tree using ANTLR notation
        // ^(nil x y z) is shown as just x y z
        System.out.println(list.toStringTree());
    }
}
```

The unrestricted tree node data type comes at the cost of using the adapter to create and connect nodes, which is not the simplest means of building trees. Since you annotate grammars to build trees instead of manually coding the tree construction, this is not a burden.

When executed, the program emits the following:

```
$ java XYZList
x y z
$
```

If you'd like to use your own tree node type because you want to add some fields to each node, define your node as a subclass of CommonTree:

Download trees/MyNode.java

```
import org.antlr.runtime.tree.*;
import org.antlr.runtime.Token;

public class MyNode extends CommonTree {
    /** If this is an ID node, symbol points at the corresponding
     *  symbol table entry.
     */
    public Symbol symbol;

    public MyNode(Token t) {
        super(t);
    }
}
```

Then, subclass CommonTreeAdaptor, and override create() so that it builds your special nodes:

```
/** Custom adaptor to create MyNode nodes */
class MyNodeAdaptor extends CommonTreeAdaptor {
    public Object create(Token payload) {
        return new MyNode(payload);
    }
}
```

Finally, you must inform ANTLR that it should use your custom adaptor:

```
MyParser parser = new MParser(tokens,symtab); // create parser
MyNodeAdaptor adaptor = new MyNodeAdaptor();  // create adaptor
parser.setTreeAdaptor(adaptor);               // use my adaptor
parser.startRule();                           // launch!
```

If you'd like to build radically different trees or use an existing, say, XML DOM tree, then you will have to build a custom TreeAdaptor so ANTLR knows how to create and navigate those trees.

7.3 Default AST Construction

By default, ANTLR does not create ASTs, so you first need to set option **output** to AST. Without instructions to the contrary, ANTLR will simply

build a flat tree (a linked list) containing pointers to all the input token objects. Upon this basic default mechanism, you will add AST construction operators and AST rewrite rules as described in the following sections. Before going into those specifications, however, you need to learn a little bit about the AST implementation mechanism, which is fairly involved for even a small grammar but is not difficult to understand. The details are important for your overall understanding, and this section provides an in-depth look, but you can initially skim this section and refer to it later. The two sections that follow, on AST operators and rewrite rules, move back to the user level and show you how to annotate grammars in order to build trees.

To explore ANTLR's AST implementation mechanism, consider the following grammar that matches a list of identifiers and integers. Rule **r** yields a nil-rooted tree with the **ID** and **INT** nodes as children:

Download trees/List.g

```
grammar List;
options {output=AST;}
r : (ID|INT)+ ;
ID : 'a'..'z'+ ;
INT : '0'..'9'+;
WS : (' '|'\n'|'\r') {$channel=HIDDEN;} ;
```

The following code is a typical test harness that gets the return value data aggregate from rule **r**, extracts the tree created by **r**, and prints it:

Download trees/TestList.java

```
import org.antlr.runtime.*;
import org.antlr.runtime.tree.*;

public class TestList {
    public static void main(String[] args) throws Exception {
        // create the lexer attached to stdin
        ANTLRInputStream input = new ANTLRInputStream(System.in);
        ListLexer lexer = new ListLexer(input);
        // create the buffer of tokens between the lexer and parser
        CommonTokenStream tokens = new CommonTokenStream(lexer);
        // create the parser attached to the token buffer
        ListParser parser = new ListParser(tokens);
        // launch the parser starting at rule r, get return object
        ListParser.r_return result = parser.r();
        // pull out the tree and cast it
        Tree t = (Tree)result.getTree();
        System.out.println(t.toStringTree()); // print out the tree
    }
}
```

All rules return a tree when you specify output=AST. In this case, ANTLR generates the following return value structure for rule r:

```
public static class r_return extends ParserRuleReturnScope {
    Object tree;
    public Object getTree() { return tree; }
}
```

Here is a sample execution:

```
⇐  $ java TestList
⇐  abc 34 2 x
⇐  EOF
⇒  abc 34 2 x
   $
```

ANTLR's basic strategy is straightforward. In this case, the rules are as follows:

1. Create a root pointer for each rule.
2. For each token, create a tree node as a function of the token.
3. Add each tree node to the enclosing rule's current root.

The generated parser accordingly performs the following operations (in pseudocode):

```
define a root pointer for rule, root_0
root_0 = adaptor.nil(); // make a nil root for rule
create a node for abc (element of ID|INT set)
add node as child of root_0
create a node for 34
add node as child of root_0
create a node for 2
add node as child of root_0
create a node for x
add node as child of root_0
return root_0 as r's tree attribute
```

As usual, the best way to figure out exactly what ANTLR is doing is to examine the code it generates. Here is the general structure of the rule's corresponding implementation method:

```
// match rule r and return a r_return object
public r_return r() throws RecognitionException {
    r_return retval = new r_return(); // create return value struct
    retval.start = input.LT(1);       // compute $r.start
    Object root_0 = null;             // define r's root node
    try {
        root_0 = (Object)adaptor.nil(); // create nil root
        «r-prediction»
        «r-matching-and-tree-construction»
```

```
        // rule cleanup next
        retval.stop = input.LT(-1);   // compute $r.stop
        // set $r.tree to root_0 after postprocessing
        // by default, this just converts ^(nil x) to x
        retval.tree = (Object)adaptor.rulePostProcessing(root_0);
        adaptor.setTokenBoundaries(retval.tree, retval.start, retval.stop);
    }
    catch (RecognitionException re) {
        «error-recovery»
    }
    return retval;
}
```

The highlighted lines derive from the AST construction mechanism. The code to build tree nodes and add them to the tree is interspersed with the code that matches the set within the (...)+ subrule:

```
// This code happens every iteration of the subrule
set1=(Token)input.LT(1); // track the input token
if ( (input.LA(1)>=ID && input.LA(1)<=INT) ) {
    // we found an ID or INT; create node from set1
    // then ask the adapter to add it as a child of
    // rule's root node.
    adaptor.addChild(root_0, adaptor.create(set1));
    input.consume();     // move to next input symbol
    errorRecovery=false; // ignore this (just ANTLR error bookkeeping)
}
else
    «throw-exception»
// finished matching the alternative within (...)+
```

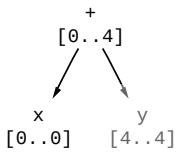
After all tree construction for a rule, the rule notifies the adapter that it should perform any necessary postprocessing on the tree before the rule returns it. This gives you a general hook to do whatever processing your trees might need on a per-rule basis. At minimum, the postprocessing needs to convert nil-rooted trees with a single child node to a simple node; that is, the postprocessing must convert ^(nil x) to x. Otherwise, the resulting tree will be filled with superfluous nil nodes. Remember that nil is used only to represent the dummy root node for lists.

Also as part of the final rule tree processing, the parser automatically computes and stores the range of tokens associated with the subtree created for that rule. This information is extremely useful later when generating code. Sometimes you want to replace sections of the input with a translation computed from the tree. To do that, you need to know the corresponding input tokens to replace. For example, consider the expression x + y (including the spaces). The root plus node will store

token boundaries 0..4, assuming that the expression is the only input and that it consists of five tokens:

```
0   1     2 3     4         Token index
INT SPACE + SPACE INT       Token sequence
```

The AST nodes will have start and stop token index boundaries, as shown in the following image:



The token range of the root node will include all hidden channel tokens (whitespace in this case) that happen to be between the first and last nonhidden tokens. Also note that, because each node has the token from which it was created as payload, each node knows its corresponding position in the original input stream. So, the + node has a token whose index is 2. These boundaries and indexes allow you to print or replace the associated construct in the original input stream. This assumes that you keep around a buffer of all tokens in input order, which CommonTokenStream does.

The details presented in this section give you a deeper understanding of how ANTLR builds ASTs, but initially you need only a basic understanding that ANTLR uses an adapter to create nodes, hooks nodes together to form trees, and uses nil-rooted trees to represent lists. The following sections move back to the user level and describe how you annotate grammars in order to build the trees you want.

7.4 Constructing ASTs Using Operators

The nice aspect of the automatic AST construction mechanism is that you can just turn it on and it builds a tree, albeit a flat one. With just a little work, however, you can add AST construction operators to have the parser incrementally build the trees you want. These AST operators force you to think about the emergent behavior of a set of operations, but they can be an extremely terse means of specifying AST structure. They work great for some common constructs such as expressions and statements. For other AST structures, though, the rewrite rules described in the next section are more effective.

The operators work like this. First turn on output=AST and assume, by the automatic mechanism, that the parser adds nodes for all tokens to the tree as siblings of the current rule's root. Then, if you do not want the parser to create a node for a particular token, suffix it with the ! operator. If you want certain tokens to become subtree roots (operators or pseudo-operators), suffix the token reference with ^. An example makes all this clear. Consider the following statement rule that has ^ and ! operators in order to build reasonable AST structures:

```
statement
    : // the result of compoundStatement is statement result
      // equivalent to -> compoundStatement
      compoundStatement

      // equivalent to -> ^('assert' $x $y?)
    | 'assert'^ x=expression (':'! y=expression)? ';'!

      // equivalent to -> ^('if' expression $s1 $s2?)
    | 'if'^ expression s1=statement ('else'! s2=statement)?

      // equivalent to -> ^('while' expression statement)
    | 'while'^ expression statement

      // equivalent to -> ^('return' expression?)
    | 'return'^ expression? ';'!
    ;
```

The comments provide the equivalent rewrite rules, which we'll examine in detail later. Generally, the rewrite rules are the clearest. For demonstrating the AST construction operators, though, statements are a good place to start.

The place where AST construction operators really shine is in expression rules. Consider the following rule that matches an integer or sum of integers:

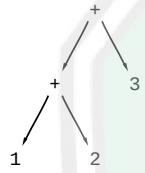
```
expr : INT ('+'^ INT)* ;
```

The rule says that the **INT** nodes are always children and that the '+' node is always a subtree root. The tree becomes one level higher for each iteration of the (...)* subrule. You should look at these as operators that alter the tree during recognition, not tree structure declarations that happen after recognition. As the parser recognizes **INT** tokens, it builds nodes for them and adds them as children of the current root node. The ^ operator takes the node created for the '+' token and makes it the new root, relegating the old root to be a child of the new root.

Operator	Description
!	Do not include node or subtree (if referencing a rule) in rule's tree. Without any suffix, all elements are added as children of current rule's root.
^	Make node root of subtree created for entire enclosing rule. Height of tree is increased by 1. Next nonsuffixed element's node or subtree becomes the first child of this root node. If the next element has a ^ suffix, then this node becomes the first child of that next element. If the suffixed element is a rule reference, that rule must return a single node, not a subtree. The result must become a root node.

Figure 7.1: AST construction operators

Given input 1, rule **expr** builds a single-node tree: INT. Input 1+2 yields ^('+' 1 2), and 1+2+3 yields ^(+ ^(+ 1 2) 3). Graphically, the latter tree looks like this:

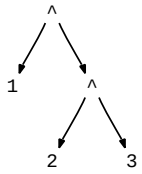


After matching 1+2, rule **expr**'s root pointer points to the three-node tree: ^('+' 1 2). After seeing the second plus operator, **expr**'s root pointer will point at a new plus node. The old ^('+' 1 2) tree will be the first child. Then, after seeing the final 3 token, the tree will be complete—the final token will be the second child of the topmost plus operator.

The AST construction operators work great for left-associative operators such as plus and multiply, but what about right-associative operators such as exponentiation? Per Section 11.5, *Arithmetic Expression Grammars*, on page 275, to handle right-associative arithmetic operators, use tail recursion to the enclosing rule. To make the exponential operator a subtree root, use the ^ suffix (do not confuse the input symbol '^', which means exponent in arithmetic expressions, with the ^ AST construction operator; they just happen to be the same symbol):

```
pow :   INT ('^'^ pow)? ;  // right-associative via tail recursion
```

Given input 1^2^3, the parser will build the following AST:



Compare this to the earlier AST built for left-associative operator plus, and input 1+2+3. The operations are effectively swapped as you would expect because the exponent operator is right-associative and should do the 2^3 operation first. The table in Figure 7.1, on the preceding page, summarizes the two AST operators.

The automatic mechanism, in combination with a few AST construction operators, provides a viable and extremely terse means of specifying trees. Except for building ASTs for expressions, however, the AST construction operators are not the best solution. The next section describes a declarative tree construction approach that is more powerful and is usually more obvious than using operators.

7.5 Constructing ASTs with Rewrite Rules

The recommended way to build ASTs is to add *rewrite rules* to your grammar. Rewrite rules are like output alternatives that specify the grammatical, two-dimensional structure of the tree you want to build from the input tokens. The notation is as follows:

New in v3.

```
rule: «alt1» -> «build-this-from-alt1»
    | «alt2» -> «build-this-from-alt2»
   ...
    | «altN» -> «build-this-from-altN»
    ;
```

For example, here is a rule that performs an identity transformation. From the **INT** input token, the parser builds a single-node tree with the **INT** token as payload:

```
e : INT -> INT ;
```

Each rule returns a single AST that you can set zero or more times using the -> operator; generally, you will use the rewrite operator once per rule invocation. The resulting tree is available to invoking rules as a predefined attribute.

For example, here is a rule that invokes **e** and prints the resulting node:

```
r : e {System.out.println($e.tree);} ;
```

While parser grammars specify how to recognize input tokens, rewrite rules are generational grammars that specify how to generate trees. ANTLR figures out how to map input to output grammars automatically. Rules queue up the elements on the left and then use them as input streams to the generational grammar on the right that specifies tree structure.

While designing this new rewrite mechanism,[1] I carefully studied many existing ANTLR v2 grammars to categorize and generalize the kind of input grammar to tree structure transformations programmers were doing. The following subsections describe the operations I found.

Omitting Input Elements

Languages use many input symbols, such as comma, semicolons, colons, curlies, parentheses, and so on, to indicate structure in the input. These symbols are not useful in the AST. The following (unrelated) rules delete superfluous tokens from the AST simply by omitting them from the rewrite specification:

```
stat: 'break' ';' -> 'break' ; // delete by omission

expr: '(' expr ')' -> expr     // omit parentheses
    | INT          -> INT
    ;

a : ID -> ; // return no AST
```

Reordering Input Elements

Sometimes the order of input that humans want to use is not the most convenient processing order for the AST. Reorder elements by specifying a new order in the rewrite rule:

```
/** flip order of ID, type; omit 'var', ':' */
decl : 'var' ID ':' type -> type ID ;
```

Making Input Elements the Root of Others

To specify two-dimensional structure, you must indicate which tokens should become subtree roots. Rewrite rules use ^(...) syntax where the

1. I am grateful to Loring Craymer, Monty Zukowski, and John Mitchell (longtime research collaborators) for their help in designing the tree rewrite mechanism.

first element is the root of the remaining (child) elements, as shown in the following rules:

```
/** Make 'return' the root of expr's result AST */
stat : 'return' expr ';' -> ^('return' expr) ;

/** Use 'var' as root of type, ID; flip over of type, ID */
decl : 'var' ID ':' type -> ^('var' type ID) ;
```

Adding Imaginary Nodes

As discussed in Section 7.1, *Proper AST Structure*, on page 163, you will create imaginary nodes to represent pseudo-operations such as "declare variable," "declare method," etc. To create an imaginary node in the AST, simply refer to its token type, and ANTLR will create a Token object with that token type and make it the payload of a new tree node:

```
/** Create a tree with imaginary node VARDEF as root and
 *  type, ID as children.
 */
decl : type ID ';' -> ^(VARDEF type ID) ;

/** Ensure that there is always an EXPR imaginary node for
 *  a loop conditional even when there is no expression.
 */
forLoopConditional
    : expression -> ^(EXPR expression)
    |            -> EXPR // return EXPR root w/o children
    ;
```

An imaginary token reference is a token reference for which there is no corresponding token reference on the left side of the -> operator. The imaginary token must be defined elsewhere in a grammar or in the **tokens** section.

Collecting Input Elements and Emitting Together

You can collect various input symbols and ASTs created by other rules to include them as a single list. You can also include a list of repeated symbols from the input grammar, as shown in the following rules:

```
/** Collect all IDs and create list of ID nodes; omit ',' */
list : ID (',' ID)* -> ID+ ; // create AST with 1 or more IDs

/** Collect the result trees from all formalArg invocations
 *  and put into an AST list.
 */
formalArgs
    :   formalArg (',' formalArg)* -> formalArg+
    |
    ;
```

```
/** Collect all IDs and create a tree with 'int' at the root
 *  and all ID nodes as children.
 */
decl : 'int' ID (',' ID)* -> ^('int' ID+) ;

/** Match a complete Java file and build a tree with a UNIT
 *  imaginary root node and package, import, and type definitions
 *  as children.  A package definition is optional in the input
 *  and, therefore, must be optional in the rewrite rule.  In
 *  general, the ''cardinality'' of the rewrite element must match
 *  how many input elements the rule can match.
 */
compilationUnit
    :   packageDef? importDef* typeDef+
        -> ^(UNIT packageDef? importDef* typeDef+)
    ;
```

Duplicating Nodes and Trees

When breaking a single source construct into multiple operations in your AST, you will often need to include pieces of the input in each of the operations. ANTLR will automatically duplicate nodes and trees as necessary, as demonstrated in the following examples:

```
/** Make a flat list consisting of two INT nodes pointing
 *  at the same INT token payload.  The AST nodes are duplicates,
 *  but they refer to the same token.
 */
dup : INT -> INT INT ;

/** Create multiple trees of form ^('int' ID), one for each input
 *  ID.  E.g., "int x,y" yields a list with two trees:
 *  ^(int x) ^(int y)
 *  The 'int' node is automatically duplicated.
 *  Note: to be distinguished from ^('int' ID+), which makes a
 *        single tree with all IDs as children.
 */
decl : 'int' ID (',' ID)* -> ^('int' ID)+ ;

/** Like previous but duplicate tree returned from type */
decl : type ID (',' ID)* -> ^(type ID)+ ;

/** Include a duplicate of modifier if present in input.
 *  Trees look like:
 *      ^(int public x) ^(int public y)
 *  or
 *      ^(int x) ^(int y)
 */
decl : modifier? type ID (',' ID)* -> ^(type modifier? ID)+ ;
```

The advantage of always creating duplicates for elements that you reference multiple times is that there is no possibility of creating cycles in the tree. Recall that a tree must not have children that point upward in the tree.

Otherwise, tree walkers (visitors, tree grammars, and so on) will get stuck in an infinite loop; for example, the AST resulting from t.addChild(t); will prevent a tree walker from terminating. Without automatic duplication, the following rule would be equivalent to t.addChild(t);:

```
a : ID -> ^(ID ID) ; // no cycle
```

ANTLR generates the following code for the rewrite rule:

```
// make dummy nil root for ^(...)
Object root_1 = adaptor.nil();

// make a node created from INT the root
root_1 = adaptor.becomeRoot((Token)list_INT.get(i_0), root_1);

// make another node created from INT as child
adaptor.addChild(root_1, (Token)list_INT.get(i_0));

// add rewrite ^(...) tree to rule result
adaptor.addChild(root_0, root_1);
```

In a rewrite rule, ANTLR duplicates any element with cardinality one (that is, one node or one tree) when referenced more than once or encountered more than once because of an EBNF * or + suffix operator. See Section 7.5, *Rewrite Rule Element Cardinality*, on page 184.

Choosing between Tree Structures at Runtime

Sometimes you do not know which AST structure to create for a particular alternative until runtime. Just list the multiple structures with a semantic predicate in front that indicates the runtime validity of applying the rewrite rule. The predicates are tested in the order specified. The rewrite rule associated with the first true predicate generates the rule's return tree:

```
/** A field or local variable.  At runtime, boolean inMethod
 *  determines which of the two rewrite rules the parser applies.
 */
variableDefinition
    :   modifiers type ID ('=' expression)? ';'
        -> {inMethod}? ^(VARIABLE ID modifier* type expression?)
        ->             ^(FIELD ID modifier* type expression?)
    ;
```

You may specify a default rewrite as the last unpredicated rewrite:

```
a[int which] // pass in parameter indicating which to build
  : ID INT -> {which==1}? ID
           -> {which==2}? INT
           ->                  // yield nothing as else-clause
  ;
```

Referring to Labels in Rewrite Rules

The previous sections enumerated the operations you will likely need when building ASTs and identified, by example, the kinds of rewrite rule elements you can use. The example rewrite rules used token and rule references.

The problem with token and rule references is that they grab all elements on the left side of the -> operator with the same name. What if you want only some of the **ID** references, for example? Using ID+ in the rewrite rule yields a list of all **ID** tokens matched on the left side.

Use labels for more precise control over which input symbols the parser adds to the AST. Imagine that a program in some language is a list of methods where the first method is considered the main method in which execution begins. The AST you create should clearly mark which method is the main method with an imaginary root node; otherwise, you will get a list of undifferentiated methods.

The following rule splits the usual method+ grammar construct into two pieces, method method*, identified with labels so that the rewrite rule can treat the first method separately:

```
prog: main=method others+=method* -> ^(MAIN $main) $others* ;
```

As another example, consider that you might want to encode the expressions in a **for** loop differently in the AST. The following rule illustrates how to refer to two different expressions via labels:

```
forStat
    : 'for' '(' decl? ';' cond=expr? ';' iter=expr? ')' slist
      -> ^( 'for' decl? ^(CONDITION $cond)? ^(ITERATE $iter)? )
    ;
```

You can use labels anywhere you would usually use a token or rule reference.

Creating Nodes with Arbitrary Actions

As a "safety blanket," ANTLR provides a way for you to specify tree nodes via an arbitrary action written in the target language, as shown in the following rule:

```
/** Convert INT into a FLOAT with text = $INT.text + ".0" */
a : INT -> {new CommonTree(new CommonToken(FLOAT,$INT.text+".0"))} ;
```

An action can appear anywhere that a token reference can appear, but you cannot suffix actions with cardinality operators such as + and *.

You can use arbitrary actions to access trees created elsewhere in a grammar. For example, when building class definitions for a Java-like language, the most natural grammar might match modifiers outside the rule that builds the AST for class definitions. The **typeDefinition** rule matches the modifiers and passes the resulting tree to the **classDefinition** rule:

```
typeDefinition
    :   modifiers! classDefinition[$modifiers.tree]
    |   modifiers! interfaceDefinition[$modifiers.tree]
    ;
```

The result AST of the **modifiers** rule is not included in the tree for **typeDefinition** because of the ! operator. The result of **typeDefinition** is, therefore, purely the result of either **classDefinition** or **interfaceDefinition**.

The rewrite rule in **classDefinition** illustrates a number of techniques described by the previous sections. The rule returns a tree rooted by the 'class' token. The children are the class name, the modifiers, the superclass, the potentially multiple interface implementations, the variables, the constructors, and the method definitions:

```
/** Match a class definition and pass in the tree of modifiers
 *  if any.
 */
classDefinition[CommonTree mod]
    :   'class' ID ('extends' sup=typename)?
        ('implements' i+=typename (',' i+=typename)*)?
        '{'
        (   variableDefinition
        |   methodDefinition
        |   ctorDefinition
        )*
        '}'
        -> ^('class' ID {$mod} ^('extends' $sup)? ^('implements' $i+)?
             variableDefinition* ctorDefinition* methodDefinition*
            )
    ;
```

Regardless of the input order of the member definitions, the tree orders them first by variable, then by constructor, and then by method. The action referencing $mod simply inserts that parameter as the second child of the resulting AST. Because you can reference rule elements only within a rewrite rule, you must enclose attribute references such as $mod in the curly braces of an action.

The third child, ^('extends' $sup)?, is a nested subtree whose child is the superclass. If the ('extends' typename)? input clause matches no input, the third child will evaluate to an empty tree. The fourth child similarly represents a nested tree with any interface implementations as children. The subtree evaluates to an empty tree if there were no implementations found on the input stream. The next section describes how EBNF operators such as ? can result in empty trees.

Rewrite Rule Element Cardinality

You can suffix rewrite rule elements with the EBNF subrule operators (?, *, or +) as you have seen, and, for the most part, their semantics are natural and clear. For example, ID+ and atom* obviously generate lists of one or more **ID** nodes and zero or more rule **atom** results, respectively. The parser throws a runtime exception if there is not at least one **ID** token from which to generate a node.

Also, if the parser sees more than one **ID** node during an ID? rewrite, it will also throw a runtime exception. Things get a little more complicated when the suffixed element is within a tree such as ^(VARDEF atom+). That subtree builds a single VARDEF-rooted tree with all **atom** results as children.

What about ('int' ID)+ where the + is now around a group of elements and not the individual **ID** element? This requires a more formal definition of what the EBNF operators do. The best way to think about rewrite rules is that they are the dual of parsing rules. Just imagine the rewrite rule matching input instead of generating output, and things usually make more sense. If you tried to match that closure operation, you would clearly need as many 'int' keywords as identifiers on the input stream.

Similarly, when generating output, ANTLR must create as many 'int' as **ID** nodes, even if it means replicating one or the other to match cardinalities. If the input grammar matched 'int' only once, for example, the AST construction mechanism would duplicate it once for every **ID** found on the input stream.

An Analogy to Explain Rewrite Rule Cardinality

Rewrite rule element cardinalities must match up just like stuffing letters into envelopes and stamping them. If you have one letter, you must have one envelope and one stamp. In general, you must match up the number of letters, envelopes, and stamps. The following rule captures that analogy:

```
/** Match a letter for each envelope and stamp, create trees with
 *  the letter as the root and the envelopes, stamps as children.
 */
pile : (letter envelope stamp)+ -> ^(letter envelope stamp)+ ;
```

Now, if you have only one letter but you want to send it to many people, you must duplicate the letter once for every addressed and stamped envelope:

```
/** Like duplicating one letter to put in multiple envelopes.
 *  The rule generates one tree for every stamped envelope with
 *  the letter node duplicated across them all.
 */
formLetter:letter (envelope stamp)+ -> ^(letter envelope stamp)+ ;
```

A closure operator such as + builds *n* nodes or trees according to the suffixed element where *n* is the maximum of the cardinalities of the element or elements within. Revisiting some previous examples, let's identify the cardinality of the elements:

Cardinality means how many of a particular element there are.

```
// cardinality of both type and ID is 1
decl : type ID ';' -> ^(VARDEF type ID) ;

// cardinality of ID is >= 1
list : ID (',' ID)* -> ID+ ;
```

Now consider the case where the + operator suffixes a tree:

```
// cardinality of ID is >= 1
decl : 'int' ID (',' ID)* ';' -> ^(VARDEF ID)+ ;
```

How many times does the + operator loop in the rewrite rule? That is, how many trees does the rule generate? The answer lies in the cardinality of elements referenced within. If all elements have cardinality of 1, then *n*=1, and the rule will generate one tree. If at least one element has cardinality greater than 1, then *n* is equal to that element's cardinality. All elements with cardinality greater than 1 must have exactly the same cardinality.

When there are some elements with cardinality one and others with cardinality greater than one, the elements with cardinality one are duplicated as the parser creates the tree. In the following rule, the 'int' token has cardinality one and is replicated for every **ID** token found on the input stream:

```
decl : 'int' ID (',' ID)* -> ^('int' ID)+ ;
```

Naturally, when the cardinality of the elements within a suffixed element is zero, the parser does not create a tree at all:

```
// if there is no "int ID" then no tree will be generated
decl : ('int' ID)? -> ^('int' ID)? ;
```

What about imaginary nodes, which always have cardinality one? Do they force the construction of trees even when the real elements within the tree have cardinality zero? No. Trees or subrules suffixed with EBNF operators yield trees only when at least one real element within the tree has cardinality greater than zero. For example, consider the following rule and AST rewrite rule:

```
initValue : expr? -> ^(EXPR expr)? ;
```

Rewrite ^(EXPR expr)? yields no tree when the **expr** on the left side returns no tree (the cardinality is zero). To be clear, ^(EXPR expr?) always yields at least an EXPR node. But, ^(EXPR expr)?, with the suffix on the entire tree, will not yield a tree at all if **expr** matched no input on the left side.

Most of the time the rewrite rules behave as you expect, given that they are the dual of recognition rules. Just keep in mind that the cardinality of the elements within suffixed subrules and trees must always be the same if their cardinality is greater than one. Also, if any element's cardinality is greater than one, the parser replicates any elements with cardinality one.

Rewrite Rules in Subrules

Even when a rewrite rule is not at the outermost level in a rule, it still sets the rule's result AST. For example, the following rule matches **if** statements and uses syntax to drive tree construction. The presence or absence of an **else** clause dictates which rewrite rule in the subrule to execute.

```
ifstat
    : 'if' '(' equalityExpression ')' s1=statement
      ('else' s2=statement -> ^('if' ^(EXPR equalityExpression) $s1 $s2)
      |                    -> ^('if' ^(EXPR equalityExpression) $s1)
      )
```

Here is another example where you might want to drive AST construction with syntax in a subrule:

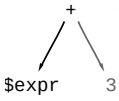
```
decl: type
      ( ID '=' INT -> ^(DECL_WITH_INIT type ID INT)
      | ID         -> ^(DECL type ID)
      )
    ;
```

Referencing Previous Rule ASTs in Rewrite Rules

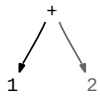
Sometimes you can't build the proper AST in a purely declarative manner. In other words, executing a single rewrite after the parser has matched everything in a rule is insufficient. Sometimes you need to iteratively build up the AST (which is the primary motivation for the automatic AST construction operators described in Section 7.4, *Constructing ASTs Using Operators*, on page 174). To iteratively build an AST, you need to be able to reference the previous value of the current rule's AST. You can reference the previous value by using $r within a rewrite rule where r is the enclosing rule. For example, the following rule matches either a single integer or a series of integers added together:

```
expr : (INT -> INT) ('+' i=INT -> ^('+' $expr $i) )* ;
```

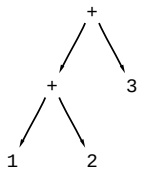
The (INT->INT) subrule looks odd but makes sense. It says to match **INT** and then make its AST node the result of **expr**. This sets a result AST in case the (. . .)* subrule that follows matches nothing. To add another integer to an existing AST, you need to make a new '+' root node that has the previous expression as the left child and the new integer as the right child. The following image portrays the AST that the rewrite rule in the subrule creates for an iteration matching +3. After each iteration, $expr has a new value, and the tree is one level taller.



Input 1 results in a single node tree with token type INT containing 1. Input 1+2 results in the following:



And, input 1+2+3 results in the following:



That grammar with embedded rewrite rules recognizes the same input and generates the same tree as the following version that uses the construction operators:

```
expr : INT ('+'^ INT)* ;
```

The version with operators is much easier to read, but sometimes you'll find embedded rewrite rules easier. Here is a larger example where a looping subrule must reference previous values of the rule's AST incrementally:

```
postfixExpression
    :   (primary->primary) // set return tree to just primary
        (   '(' args=expressionList ')'
            -> ^(CALL $postfixExpression $args)
        |   '[' ie=expression ']'
            -> ^(INDEX $postfixExpression $ie)
        |   '.' p=primary
            -> ^(FIELDACCESS $postfixExpression $p)
        )*
    ;
```

Again, the (primary->primary) subrule matches **primary** and then makes its result tree the result of **postfixExpression** in case nothing is matched in the (. . .)* subrule. Notice that the last rewrite rule must use a label to specifically target the proper **primary** reference (there are two references to **primary** in the rule).

Deriving Imaginary Nodes from Real Tokens

You will create a lot of imaginary nodes to represent pseudo-operations in your language. A problem with these imaginary nodes is that, because they are not created from real tokens, they have no line and column information or token index pointing into the input stream. This information is useful in a number of situations such as generating error messages from tree walkers. It's also generally nice to know from where in the input stream the parser derived an AST construct.

ANTLR allows you to create imaginary nodes with a constructor-like syntax so you can derive imaginary nodes from existing tokens. The following rule creates an SLIST imaginary node, copying information from the '{' real token:

```
compoundStatement
    : lc='{' statement* '}' -> ^(SLIST[$lc] statement*)
    ;
```

The SLIST node gets the line and column information from the left curly's information.

You can also set the text of an imaginary node to something more appropriate than a left curly by adding a second parameter to the imaginary node constructor: SLIST[$lc,"statements"]. The following table summarizes the possible imaginary node constructors and how they are implemented.

Imaginary Node Constructor	**Tree Adapter Invocation**
T	`adaptor.create(T, "T")`
T[]	`adaptor.create(T, "T")`
T[*token-ref*]	`adaptor.create(T, token-ref)`
T[*token-ref*, "*text*"]	`adaptor.create(T, token-ref, "text")`

Combining Rewrite Rules and Automatic AST Construction

By default, output=AST in your parser causes each rule to build up a list of the nodes and subtrees created by its elements. Rewrite rules, though, turn off this automatic tree construction mechanism. You are indicating that you want to specify the tree manually. In many cases, though, you might want to combine the automatic mechanism with rewrite rules in the same rule. Or, you might want to have some rules use the automatic mechanism and others use rewrite rules. For example, in the following rule, there is no point in specifying rewrite rules because the automatic mechanism correctly builds a node from the modifier tokens:

```
modifier
    :   'public'
    |   'static'
    |   'abstract'
    ;
```

The same is true for rules that reference other rules:

```
typename
    :   classname
    |   builtInType
    ;
```

The following rule illustrates when you might want to use a rewrite rule in one alternative and the automatic mechanism in another alternative:

```
primary
    :   INT
    |   FLOAT
    |   '(' expression ')' -> expression
    ;
```

In general, the automatic mechanism works well for alternatives with single elements or repeated constructs such as method+.

This chapter described the kinds of tree structures to build for various input constructs,[2] how ANTLR implements tree construction, and how to annotate grammars with rewrite rules and construction operators to build ASTs. In the next chapter, we'll construct a complete parser grammar that builds ASTs for a subset of C and a tree grammar that walks those trees.

2. Readers familiar with ANTLR v2 might be curious about building different Java node types depending on the token type or might want to build simple parse trees. Please see the wiki entries on heterogeneous trees at http://www.antlr.org/wiki/pages/viewpage.action?pageId=1760 and parse trees at http://www.antlr.org/wiki/pages/viewpage.action?pageId=1763.

Chapter 8

Tree Grammars

One of the most common questions programmers have when building a translator is, "What do I do with my AST now that I've built it?" Their first reaction is often to use a visitor pattern[1] that essentially does a depth-first walk of the tree, executing an action method at each node. Although easy to understand, this approach is useful only for the simplest of translators. It does not validate tree structure, and actions are isolated "event triggers" that do not have any context information. The actions know only about the current node and know nothing about the surrounding tree structure. For example, visitor actions do not know whether an **ID** node is in a variable definition or an expression.

A good way to think of tree grammars are as executable documentation, formally describing complete tree structure.

The next step is to write a tree walker that manually checks the structure. The walker is aware of context either implicitly by passing information down the tree during the walk or by setting globally visible variables such as instance variables. Rather than build a tree walker by hand, though, you should use a tree grammar just like you do when building a text parser. To execute actions for certain subtrees of interest, just embed actions in your grammar at the right location.

Improved in v3.

ANTLR implements tree parsers with the same mechanism used to parse token streams. When you look at the generated code for a tree parser, it looks almost identical to a token parser. The tree parser expects a one-dimensional stream of nodes with embedded **UP** and **DOWN** imaginary nodes to mark the beginning and end of child lists, as described in Section 3.3, *Evaluating Expressions Encoded in ASTs*, on page 79.

1. See http://en.wikipedia.org/wiki/Visitor_pattern.

The CommonTreeNodeStream class will serialize the tree for the tree parser, as shown in the following code template:

```
CommonTreeNodeStream nodes = new CommonTreeNodeStream(tree);
treeGrammarName walker = new treeGrammarName(nodes);
walker.start-symbol();
```

The three sections in this chapter describe the basic approach to writing a tree grammar (by copying and then altering the parser grammar) and then illustrate a complete parser grammar and tree grammar for a simple C-like language.

8.1 Moving from Parser Grammar to Tree Grammar

Once you have a grammar that builds ASTs, you need to build one or more tree grammars to extract information, compute ancillary data structures, or generate a translation. In general, your tree grammar will have the following preamble:

```
tree grammar treeGrammarName;

options {
    tokenVocab=parserGrammarName; // reuse token types
    ASTLabelType=CommonTree; // $label will have type CommonTree
}
...
```

Then it is a matter of describing the trees built by the parser grammar with a tree grammar. AST rewrite rules are actually generational grammars that describe the trees that the parser grammar rule should construct. To build the tree grammar, you can reuse these rewrite rules as tree matching rules. The easiest way to build your tree grammar then is simply to copy and paste your parser grammar into a tree grammar and to remove the original parser grammar components (leaving only the rewrite rules). For example, given the following parser grammar rule:

```
grammar T;
...
decl : 'int' ID (',' ID)* -> ^('int' ID+) ;
```

translate it to this tree grammar:

```
tree grammar T;
...
decl : ^('int' ID+) ;
```

Parser rules that have AST rewrite rules within subrules also translate easily because rewrite rules always set the rule's result tree. The following rule:

```
grammar T;
...
ifstat
    : 'if' '(' equalityExpression ')' s1=stat
      ( 'else' s2=stat -> ^('if' ^(EXPR equalityExpression) $s1 $s2)
      |                -> ^('if' ^(EXPR equalityExpression) $s1)
      )
    ;
```

translates to this:

```
tree grammar T;
...
ifstat
    : ^('if' ^(EXPR equalityExpression) stat stat?)
    ;
```

which merges the two possible tree structures for simplicity. Note that `$s1` in the parser grammar becomes the first `stat` reference in the tree grammar. `$s2` becomes the second `stat` reference if the parser rule matches an `else` clause.

If the parser rule decides at runtime which tree structure to build using a semantic predicate, there is no problem for the tree grammar. The tree will be either one of those alternatives—just list them. For example, the following grammar:

```
grammar T;
...
variableDefinition
    :   modifiers type ID ('=' expression)? ';'
        -> {inMethod}? ^(VARIABLE ID modifier* type expression?)
        ->             ^(FIELD ID modifier* type expression?)
    ;
```

translates to the following:

```
tree grammar T;
...
variableDefinition
    :   ^(VARIABLE ID modifier* type expression?)
    |   ^(FIELD ID modifier* type expression?)
    ;
```

The AST construction operators are a little trickier, which is why you should use the AST rewrite rules where possible.

The ! operator is usually straightforward to translate because you can simply leave that element out of the tree grammar. For example, in the following rule, the semicolons are not present in the tree grammar:

```
grammar T;
...
stat: forStat
    | expr ';'!
    | block
    | assignStat ';'!
    | ';'!
    ;
```

The tree grammar looks like this:

```
tree grammar T;
...
stat: forStat
    | expr
    | block
    | assignStat
    ;
```

where the final parser rule alternative is absent from the tree grammar because, after removing the semicolon, there are no more elements to match (see the next section for an example rule that contains the ^ operator).

Finally, you can usually copy rules verbatim to the tree grammar that neither have elements modified with AST construction operators nor have rewrite rules. Here are two such rules:

```
tree grammar T;
...
/** Match one or more declaration subtrees */
program
    :   declaration+
    ;

/** Match a single node representing the data type */
type:   'int'
    |   'char'
    |   ID
    ;
```

The best way to learn about tree grammars is by example. The next section illustrates a complete AST-building parser grammar for a simple programming language.

The section following the next section builds a tree grammar to walk those trees and print the variable and function names defined in the input.

8.2 Building a Parser Grammar for the C- Language

This section presents a complete parser grammar with annotations to build ASTs for a small subset of the C programming language that we can call "C-" in honor of the grades I received as an undergraduate.[2]

The next section presents a tree grammar that describes all the possible AST structures emitted from the parser. The tree grammar has a few actions to print variable and function definitions just to be sure that the test rig actually invokes the tree walker.

Where appropriate, this section describes the transformation from parser grammar to tree grammar because that is the best way to build tree grammars for ASTs created by parser grammars.

First, define the language itself. Here is some sample input:

Download trees/CMinus/t.cm

```
char c;
int x;
int foo(int y, char d) {
  int i;
  for (i=0; i!=3; i=i+1) {
    x=3;
    y=5;
  }
}
```

C- has variable and function definitions and types int and char. Statements are limited to **for**, function calls, assignments, nested code blocks, and the empty statement signified by a semicolon. Expressions are restricted to conditionals, addition, multiplication, and parenthesized expressions.

The combined parser and lexer grammar for C- begins with the **grammar** header, an **options** section to turn on AST construction, and a **tokens**

2. See also http://www.codegeneration.net/tiki-read_article.php?articleId=77, which has an ANTLR v2 version of the same parser grammar. This might be useful for those familiar with v2 wanting to upgrade to v3.

section to define the list of imaginary tokens that represent pseudo-operations in C-:

Download trees/CMinus/CMinus.g

```
/**  Recognize and build trees for C-
 *   Results in CMinusParser.java, CMinusLexer.java,
 *   and the token definition file CMinus.tokens used by
 *   the tree grammar to ensure token types are the same.
 */
grammar CMinus;

options {output=AST;} // build trees

tokens {
  VAR;   // variable definition
  FUNC;  // function definition
  ARG;   // formal argument
  SLIST; // statement list
}
```

A C- program is a list of declarations. Declarations can be either variables or functions, as shown in the next chunk of the grammar:

Download trees/CMinus/CMinus.g

```
program
    :   declaration+
    ;

declaration
    :   variable
    |   function
    ;

variable
    :   type ID ';' -> ^(VAR type ID)
    ;

type:   'int'
    |   'char'
    ;
```

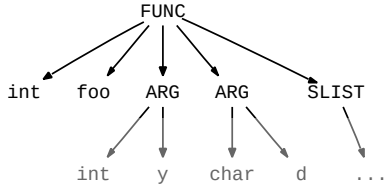
Functions look like this:

Download trees/CMinus/CMinus.g

```
// E.g., int f(int x, char y) { ... }
function
    :   type ID
        '(' ( formalParameter (',' formalParameter)* )?  ')'
        block
        -> ^(FUNC type ID formalParameter* block)
    ;
```

```
formalParameter
    :   type ID -> ^(ARG type ID)
    ;
```

The parser uses the VAR imaginary node to represent variable declarations and FUNC to represent function declarations. Arguments use ARG nodes. Rule **function** builds trees that look like this:



Notice rules **program**, **declaration**, and **type** do not need either AST construction operators or rewrite rules because the default action to build a list of nodes works in each case.

Here are the rules related to statements in C-:

Download trees/CMinus/CMinus.g

```
block
    :   lc='{' variable* stat* '}'
        -> ^(SLIST[$lc,"SLIST"] variable* stat*)
    ;

stat: forStat
    | expr ';'!
    | block
    | assignStat ';'!
    | ';'!
    ;

forStat
    :   'for' '(' first=assignStat ';' expr ';' inc=assignStat ')' block
        -> ^('for' $first expr $inc block)
    ;

assignStat
    :   ID '=' expr -> ^('=' ID expr)
    ;
```

Statement blocks in curly braces result in an AST rooted with the SLIST imaginary node. The SLIST node is derived from the left curly so that line and column information is copied into the imaginary node for error messages and debugging. The node constructor also changes the text from { to SLIST. Rule **stat** uses the ! AST construction operator to quickly and easily prevent the parser from creating nodes for

semicolons. Semicolons make it easier for parsers to recognize statements but are not needed in the tree.

The expression rules use AST construction operators almost exclusively because they are much more terse than rewrite rules:

Download trees/CMinus/CMinus.g

```
expr:   condExpr ;

condExpr
    :   aexpr ( ('=='^|'!='^) aexpr )?
    ;

aexpr
    :   mexpr ('+'^ mexpr)*
    ;

mexpr
    :   atom ('*'^ atom)*
    ;

atom:   ID
    |   INT
    |   '(' expr ')' -> expr
    ;
```

All nonsuffixed tokens are subtree leaf nodes (operands), and the elements suffixed with ^ are subtree root nodes (operators). The only rule that uses a rewrite is **atom**. The parenthesized expression alternative is much clearer when you say explicitly -> expr rather than putting ! on the left parenthesis and right parenthesis tokens.

There are only three lexical rules, which match identifiers, integers, and whitespace. The whitespace is sent to the parser on a hidden channel and is, therefore, available to the translator. Using skip() instead would throw out the whitespace tokens rather than just hiding them:

Download trees/CMinus/CMinus.g

```
ID  :   ('a'..'z'|'A'..'Z'|'_') ('a'..'z'|'A'..'Z'|'0'..'9'|'_')* ;

INT :   ('0'..'9')+ ;

WS  :   ( ' ' | '\t' | '\r' | '\n' )+ { $channel = HIDDEN; } ;
```

The following key elements of the test rig create the lexer attached to standard input, create the parser attached to a token stream from the lexer, invoke the start symbol **program**, and print the resulting AST (the full test rig appears in the next section):

```
ANTLRInputStream input = new ANTLRInputStream(System.in);
CMinusLexer lexer = new CMinusLexer(input);
CommonTokenStream tokens = new CommonTokenStream(lexer);
CMinusParser parser = new CMinusParser(tokens);
CMinusParser.program_return r = parser.program();
CommonTree t = (CommonTree)r.getTree();
System.out.println(t.toStringTree());
```

Given the previous input, this test rig emits the following textual version of the AST (formatted and annotated with the corresponding input symbols to be more readable):

```
(VAR char c)                        // char c;
(VAR int x)                         // int x;
(FUNC int foo                       // int foo(...)
  (ARG int y) (ARG char d)          // int y, char d
  (SLIST
    (VAR int i) // int i;
    (for (= i 0) (!= i 3) (= i 0) // for (int i=0; i!=3; i=i+1)
      (SLIST (= x 3) (= y 5))       // x=3; y=5;
    )
  )
)
```

At this point, we have a complete parser grammar that builds ASTs. Now imagine that we want to find and print all the variable and function definitions with their associated types. One approach would be to simply build a visitor that looks for the VAR and FUNC nodes so conveniently identified by the parser. Then, the visitor action code could pull apart the operands (children) to extract the type and identifier, but that would be manually building a recognizer for ^(VAR type ID) and ^(FUNC type ID ...). A better solution is to describe the AST structure formally with a grammar. Not only does this create nice documentation, but it lets you use a domain-specific language to describe your tree structures rather than arbitrary code. The next section describes how you build a tree grammar by copying and transforming your parser grammar.

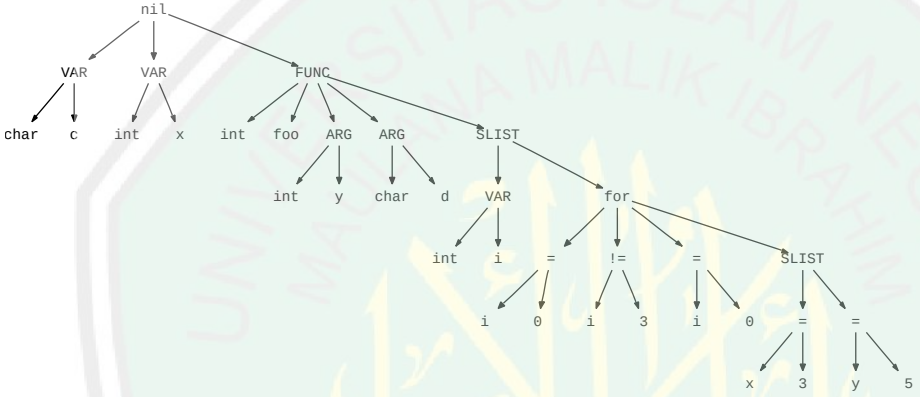
8.3 Building a Tree Grammar for the C- Language

Before building the tree grammar, take a step back and think about the relationship between your parser that builds ASTs and the tree parser that walks the ASTs. Your parser grammar describes the one-dimensional structure of the input token stream and defines the (usually infinite) set of valid sentences. You should design your ASTs so that they are a condensed and highly processed representation of the input stream.

Where there can be some tricky constructs in the input language and constructs that represent multiple operations, all subtrees in the AST should represent single operations. These operations should have obvious root node types that clearly differentiate the various subtrees.

For example, both variable and function declarations in the C- language begin with the same sequence, type ID, but in the tree grammar they have unique root nodes, VAR and FUNC, that make them easy to distinguish. So, your tree grammar should match a simpler and two-dimensional version of the same input stream as the parser grammar. Consequently, your tree grammar should look a lot like your parser grammar, as you will see in this section.

Again, begin by examining the input language for your grammar. The following image represents a sample input AST generated for the t.cm file shown in the previous section:



The goal is to generate output such as the following to identify the variables and functions defined in the input.

```
$ java TestCMinus < t.cm
define char c
define int x
define int i
define int foo()
$
```

Because the tree grammar should look very much like the parser grammar, begin by copying the parser grammar from CMinus.g to CMinusWalker.g and then altering the header per Section 8.1, *Moving from Parser Grammar to Tree Grammar*, on page 192:

Download trees/CMinus/CMinusWalker.g

```
tree grammar CMinusWalker;

options {
  tokenVocab=CMinus; // import tokens from CMinus.g
  ASTLabelType=CommonTree;
}
```

You do not need the imaginary token definitions in the tree grammar because the tokenVocab=CMinus option imports the token type definitions from the parser grammar, CMinus.g, via the CMinus.tokens file.

In the following declarations area of the grammar, the **program**, **declaration**, and **type** rules are identical to their counterparts in the parser grammar because they did not use rewrite rules or AST construction operators to create a two-dimensional structure. These rules merely created nodes or lists of nodes, and therefore, the tree grammar matches the same one-dimensional structure. The remaining rules match the trees built by the rules in the parser grammar with the same name:

Download trees/CMinus/CMinusWalker.g

```
program
    :   declaration+
    ;

declaration
    :   variable
    |   function
    ;

variable
    :   ^(VAR type ID)
        {System.out.println("define "+$type.text+" "+$ID.text);}
    ;

type:   'int'
    |   'char'
    ;

function
    :   ^(FUNC type ID formalParameter* block)
        {System.out.println("define "+$type.text+" "+$ID.text+"()");}
    ;

formalParameter
    :   ^(ARG type ID)
    ;
```

For the statements area of the grammar, transform the rules from the parser grammar simply by removing the elements to the left of the -> rewrite operator:

Download trees/CMinus/CMinusWalker.g

```
block
    :   ^(SLIST variable* stat*)
    ;

stat: forStat
    | expr
    | block
    | assignStat
    ;

forStat
    :   ^('for' assignStat expr assignStat block)
    ;

assignStat
    :   ^('=' ID expr)
    ;
```

Rule **stat** has three ! operators, and to transform that parser grammar to a tree grammar, remove the suffixed elements, as shown previously. The empty statement alternative containing just a semicolon disappears from the tree grammar. Once you remove the semicolon reference, nothing is left in that alternative. You do not want to create an empty alternative, which would make statements optional in the tree. They are not optional in the input and should not be optional in the tree grammar. Also of note is that SLIST[$ic,"SLIST"] becomes purely a token type reference, SLIST, in a tree grammar because you are matching, not creating, that node in the tree grammar.

The expressions area of the tree grammar is more interesting. All of the expression rules from the parser grammar collapse into a single recursive rule in the tree grammar. The parser constructs ASTs for expressions that encode the order of operations by their very structure. The tree grammar does not need to repeat the multilevel expression grammar pattern used by parser grammars to deal with different levels of precedence. A tree grammar can just list the possibilities, yielding a much simpler description for expressions than a parser grammar can use. Also notice that the parentheses used for nested expressions in the parser grammar are absent from the tree grammar because those exist only to override precedence in the parser grammar.

Parentheses alter the generated AST to change the order of operations but do not change the kinds of subtree structures found for expressions. Hence, the tree grammar is unaffected:

Collapsing all expression parser rules into a single recursive tree grammar rule is a general grammar design pattern.

Download trees/CMinus/CMinusWalker.g

```
expr:   ^('==' expr expr)
    |   ^('!=' expr expr)
    |   ^('+' expr expr)
    |   ^('*' expr expr)
    |   ID
    |   INT
    ;
```

Finally, there are no lexer rules in the tree grammar because you are parsing tree nodes, not tokens or characters.

Here is a complete test rig that invokes the parser to create an AST and then creates an instance of the tree parser, CMinusWalker, and invokes its **program** start symbol. The program spits out the text presentation of the AST unless you use the -dot option, which generates DOT[3] format files (that option was used to generate the AST images shown in this chapter).

Download trees/CMinus/TestCMinus.java

```
// Create input stream from standard input
ANTLRInputStream input = new ANTLRInputStream(System.in);
// Create a lexer attached to that input stream
CMinusLexer lexer = new CMinusLexer(input);
// Create a stream of tokens pulled from the lexer
CommonTokenStream tokens = new CommonTokenStream(lexer);

// Create a parser attached to the token stream
CMinusParser parser = new CMinusParser(tokens);
// Invoke the program rule in get return value
CMinusParser.program_return r = parser.program();
CommonTree t = (CommonTree)r.getTree();

// If -dot option then generate DOT diagram for AST
if ( args.length>0 && args[0].equals("-dot") ) {
    DOTTreeGenerator gen = new DOTTreeGenerator();
    StringTemplate st = gen.toDOT(t);
    System.out.println(st);
}
else {
    System.out.println(t.toStringTree());
}
```

3. See http://www.graphviz.org.

```
// Walk resulting tree; create treenode stream first
CommonTreeNodeStream nodes = new CommonTreeNodeStream(t);
// AST nodes have payloads that point into token stream
nodes.setTokenStream(tokens);
// Create a tree Walker attached to the nodes stream
CMinusWalker walker = new CMinusWalker(nodes);
// Invoke the start symbol, rule program
walker.program();
```

To build recognizers from these two grammar files, invoke ANTLR on the two grammar files and then compile:

```
$ java org.antlr.Tool CMinus.g CMinusWalker.g
$ javac TestCMinus.java # compiles the lexer, parser, tree parser too
$
```

The following session shows the output from running the test rig on the t.cm file:

```
$ java TestCMinus < t.cm
(VAR char c) (VAR int x) (FUNC int foo (ARG int y) (ARG char d) ... )
define char c
define int x
define int i
define int foo()
$
```

If your tree grammar is wrong or, for some reason, the input AST is improperly structured, the ANTLR-generated recognizer will emit an error message. For example, if you forgot the second **expr** reference on the != alternative of the **atom**:

```
expr:   ...
    |   ^('!=' expr) // should be ^('!=' expr expr)
   ...
    ;
```

and ran the input into the test rig, you would see the following error message:

```
(VAR char c) (VAR int x) (FUNC int foo (ARG int y) (ARG char d) ... )
CMinusWalker.g:line 5:15 mismatched tree node: 3; expecting type <UP>
```

Improved in v3. The tree parser error messages in v2 were useless.

The ^(!= i 3) subtree does not match the alternative because the alternative is looking for the UP token signifying the end of the child list. The input, on the other hand, correctly has the **INT** node containing 3 before the UP token.

The first part of the error message shows the stack of rules the tree parser had entered (starting from the start symbol) at the time of the error message. This is mainly for programmers to figure out where the problem is, and you can alter these messages to be more user-friendly (that is, have less information); see Chapter 10, *Error Reporting and Recovery*, on page 241.

This chapter provided a sample parser grammar and tree grammar for a small subset of C that prints the list of declarations as a simple translation.[4] It demonstrated that creating a tree grammar is just a matter of copying and transforming the parser grammar. A real application might have multiple copies of the grammar[5] for multiple passes such as symbol definition, symbol resolution, semantic analysis, and finally code generation. In the next chapter, we'll discover how to emit text source code via StringTemplate templates in order to build code generators.

4. For a tree grammar that executes some real actions, see the interpreter tutorial at http://www.antlr.org/wiki/display/ANTLR3/Simple+tree-based+interpeter on the wiki.

5. Managing multiple copies of the same grammar but with different actions is an onerous task currently, and discussing the solutions is beyond the scope of this reference guide. I anticipate building tools similar to revision control systems to help programmers deal with multiple tree grammars.

Chapter 9

Generating Structured Text with Templates and Grammars

At the most abstract level, translators map input sentences to output sentences. Translators can be as simple as data extraction programs that count the number of input lines or as complicated as language compilers. The complexity of the translation dictates the architecture of the translator, but no translator can escape the final phase: generating structured text from an internal data structures.[1] The translator component that emits structured text is called the *emitter*.

This chapter shows how to build emitters using ANTLR and the StringTemplate template engine. In particular, we'll see the following:

- Collectively, a group of templates represents a formal emitter specification.
- Templates are easier to build and understand than a collection of print statements in embedded grammar actions.
- Isolating the emitter from the code generation logic allows us to build retargetable code generators and follows good software engineering practice.
- There are two general classes of translators: rewriters and generators.

1. Compilers generate text assembly code and then use an assembler to translate to binary. Some translators actually do generate binary output, but they are in the minority.

- Template construction rules let us specify the text to emit for any given grammar rule; these templates rules parallel the AST construction rules described in Section 7.5, *Constructing ASTs with Rewrite Rules*, on page 177.
- StringTemplate has a simple but powerful syntax with a functional language flavor.
- ANTLR and StringTemplate complement each other; translators use ANTLR to recognize input and then use StringTemplate to emit output.

Without a template engine such as StringTemplate, translators must resort to emitting code with print statements. As we'll see in this chapter, using arbitrary code to generate output is not very satisfying. StringTemplate is a separate library, and naturally, we could embed template construction actions in a grammar just like any other action. To make using StringTemplate easier, ANTLR provides special notation that lets us specify output templates among the grammar rules.

To illustrate ANTLR's integration of StringTemplate, this chapter shows how to build a generator and a rewriter. The generator is a Java bytecode generator for the simple calculator language from Chapter 3, *A Quick Tour for the Impatient*, on page 59. The rewriter is a code instrumentor for the subset of C defined in Chapter 7, *Tree Construction*, on page 162. We'll build the code instrumentor first with a simple parser grammar and template construction rules and then solve it with a tree parser grammar that does the same thing. Before diving into the implementations, let's look at why templates provide a good solution, and see how ANTLR integrates StringTemplate.

9.1 Why Templates Are Better Than Print Statements

Emitters built with print statements scattered all over the grammar are more difficult to write, to read, and to retarget. Emitters built with templates, on the other hand, are easier to write and read because you can directly specify the output structure. You do not have to imagine the emergent behavior of each rule to conjure up the output structure. Template-based emitters are easier to maintain because the templates act as executable documentation describing the output structure. Further, by specifying the output structure separately, you can more easily retarget the translator. You can provide a separate group of templates

for each language target without having to alter your code generation logic. Indeed, you don't even need to recompile the translator to incorporate a new language target. This section argues for the use of formal template-based emitter specifications by appealing to your software engineering sense.

Consider the abstract behavior of a translator. Each output sentence is a function of input sentence data and computations on that data. For example, a translator's emitter might reference a variable's name and ask, "Is this variable defined and assigned to within the current scope?" An understanding of this behavior does not directly suggest a translator architecture, though. Programmers typically fall back on what they know—arbitrary embedded grammar actions that emit output as a function of the input symbols.

Most emitters are arbitrary, unstructured blobs of code that contain print statements interspersed with generation logic and computations. These emitters violate the important software engineering principle of "separation of concerns." In the terminology of design patterns, such informal emitters violate model-view-controller separation.[2] The collection of emitter output phrases comprises the view. Unfortunately, most emitters entangle the view with the controller (parser or visitor pattern) and often the model (internal data structures). The only exceptions are programming language compilers.

Modern compilers use a specialized emitter called an *instruction generator* such as BURG [FHP92], a tree-walking pattern matcher similar in concept to ANTLR's tree parsers. Instruction generators force a separate, formal output specification that maps intermediate-form subtrees to appropriate instruction sequences. In this way, a compiler can isolate the view from the code generation logic. Retargeting a compiler to a new processor is usually a matter of providing a new set of instruction templates.

Following the lead of compilers, other translators should use formal specifications to describe their output languages. Generated text is structured, not random, and therefore, output sentences conform to a language. It seems reasonable to think about specifying the output structure with a grammar or group of templates.

2. For an in-depth discussion, see "Enforcing Strict Model-View Separation in Template Engines" at http://www.cs.usfca.edu/~parrt/papers/mvc.templates.pdf.

To process this high-level specification, you need an "unparser generator" or template engine with the flavor of a generational grammar such as StringTemplate. To encourage the use of separate output specifications, ANTLR integrates StringTemplate by providing template construction rules. These rules let you specify the text to emit for any given grammar rule in a manner that parallels the AST construction rules described in Section 7.5, *Constructing ASTs with Rewrite Rules*, on page 177. Collectively, the templates represent a formal emitter specification similar to the formal specification compilers used to emit machine instructions. By separating the templates from the translation logic, you can trivially swap out one group of templates for another. This means you can usually retarget the translators you build with ANTLR and StringTemplate; that is, the same translator can emit output in more than one language.

New in v3.

ANTLR uses templates to generate the various back ends (such as Java, C, Objective-C, Python, and so on). Each back end has a separate template group. There is not a single print statement in the ANTLR code generator—everything is done with templates. The language target option tells ANTLR which group of templates to load. Building a new ANTLR language target is purely a matter of defining templates and building the runtime library.

Isolating your translator's emitter from the code generation logic and computations make sense from a software engineering point of view. Moreover, ANTLR v3's retargetable emitter proves that you can make it work well in practice. The next section drives the point home by showing you the difference between embedded actions and template construction rules.

9.2 Comparing Embedded Actions to Template Construction Rules

Translating the input matched by a rule without a template engine is painful and error prone. You must embed actions in the rule to translate and then emit or buffer up each rule element. A few example grammars comparing embedded actions with template construction rules makes this abundantly clear.

The Dream Is Alive!

During the development of the ANTLR v1 (PCCTS) and ANTLR v2 code generators, I harbored an extremely uncomfortable feeling. I knew that the undignified blobs of code generation logic and print statements were the wrong approach, but I had no idea how to solve the problem.

Oddly enough, the key idea germinated while building the second incarnation of a big nasty web server called jGuru.com. Tom Burns (jGuru CEO) and I designed StringTemplate in response to the entangled JSP pages used in the first server version. We wanted to physically prevent programmers from embedding logic and computations in HTML pages. In the back of my mind, I mused about turning StringTemplate into a sophisticated code generator.

When designing ANTLR v3, I dreamt of a code generator where each language target was purely a group of templates. Target developers would not have to know anything about the internals of the grammar analysis or code generation logic. StringTemplate evolved to satisfy the requirements of ANTLR v3's retargetable code generator. The dream is now reality. Every single character of output comes from a template, not a print statement. Contrast this with v2 where building a new target amounts to copying an entire Java file (thereby duplicating the generator logic code) and tweaking the print statements. The v2 generators represent 39% of the total lines (roughly 4,000 lines for each language target). In v3, the code generation logic is now only 4,000 lines (8% of the total). Each new language target is about 2,000 lines, a 50% reduction over v2. More important, v3 targets are pure templates, not code, making it much easier and faster to build a robust target.

Consider the following two rules from a Java tree grammar that contain embedded actions to spit Java code back out based upon the input symbols where emit() is essentially a print statement that emits output sensitive to some indentation level:

```
methodHead
    :   IDENT {emit(" "+$IDENT.text+"(");}
        ^( PARAMETERS
            (   p=parameterDef
                {if (there-is-another-parameter) emit(",");}
            )*
        )
        {emit(") ");}
        throwsClause?
    ;

throwsClause
    :   ^( "throws" {emit("throws ");}
            (   identifier
                {if (there-is-another-id) emit(", ");}
            )*
        )
    ;
```

To figure out what these rules generate, you must imagine what the output looks like by "executing" the arbitrary embedded actions in your mind. Embedding all those actions in a grammar is tedious and often makes it hard to read the grammar.

A better approach is to simply specify what the output looks like in the form of templates, which are akin to output grammar rules. The following version of the same rules uses the -> template construction operator to specify templates in double quotes:

```
methodHead
    :   IDENT ^( PARAMETERS ( p+=parameterDef )* ) throwsClause?
        -> template(name={$IDENT.text},
                    args={$p},
                    throws={$throwsClause.st})
           "<name>(<args; separator=\", \">) <throws>"
    ;

throwsClause
    :   ^( "throws" ( ids+=identifier )* )
        -> template(exceptions={$ids})
           "throws <exceptions; separator=\", \">"
    ;
```

Everything inside a template is pure output text except for the expressions enclosed in angle brackets. For most people, seeing a template

instead of print statements makes a rule's output much clearer. The template shows the overall structure of the output and has "holes" for computations or data from the input.

Aside from allowing you to specify output constructs in a more declarative and formal fashion, StringTemplate provides a number of great text generation features such as autoindentation. Here is a rule from a Java tree grammar that uses print statements to emit array initializers with the elements indented relative to the curlies that surround them:

```
arrayInitializer
    :   ^(  ARRAY_INIT
            {emit("{"); nl(); indent();}
            (   init:initializer
                {if (there-is-another-value) emit(", ");}
            )*
            {undent(); nl(); emit("}");}
        )
    ;
```

where indent() and undent() are methods that increase and decrease the indentation level used by emit(). nl() emits a newline.

Notice that the actions must take care of indentation manually. StringTemplate, on the other hand, automatically tracks and generates indentation. The following version of **arrayInitializer** generates the same output with the array initialization values indented two spaces (relative to the curlies) automatically:

```
arrayInitializer
    :   ^( ARRAY_INIT (v+=initializer)* )
        -> template(values={$v})
        <<
        {
          <values; separator=", ">
        }
        >>
    ;
```

This section illustrates the convenience of using templates to specify output. The next section describes the StringTemplate template engine in more detail. You'll learn more about what goes inside the templates.

9.3 A Brief Introduction to StringTemplate

StringTemplate[3] is a sophisticated template engine, and you should familiarize yourself with it by referring to its documentation[4] before you build a translator, but this section explains its basic operation. You will learn enough to understand the templates used in the large examples in Section 9.6, *A Java Bytecode Generator Using a Tree Grammar and Templates*, on page 219, Section 9.7, *Rewriting the Token Buffer In-Place*, on page 228, and Section 9.8, *Rewriting the Token Buffer with Tree Grammars*, on page 234. You should at least skim this section before proceeding to the examples.

Templates are strings or "documents" with holes that you can fill in with template expressions that are a function of *attributes*. You can think of attributes as the parameters passed to a template (but don't confuse template attributes with the grammar attributes described in Chapter 6, *Attributes and Actions*, on page 130). StringTemplate breaks up your template into chunks of text and attribute expressions, which are by default enclosed in angle brackets: <«*attribute-expression*»>. StringTemplate ignores everything outside attribute expressions, treating them as just text to spit out.

StringTemplate is not a system or server—it is just a library with two primary classes of interest: StringTemplate and StringTemplateGroup. You can directly create a template in code, you can load a template from a file, and you can load a single file with many templates (a template group file). Here is the core of a "Hello World" example that defines a template, sets its sole attribute (name), and prints it:

```
import org.antlr.stringtemplate.*;
...
StringTemplate hello = new StringTemplate("Hello <name>");
hello.setAttribute("name", "World");
System.out.println(hello.toString());
```

The output is Hello World. StringTemplate calls toString() on each object to render it to text. In this case, the name attribute holds a String already.

If an attribute is multivalued, such as an instance of a list, or if you set attribute name multiple times, the expression emits the elements one after the other.

3. See http://www.stringtemplate.org.
4. See http://www.antlr.org/wiki/display/ST/StringTemplate+3.0+Documentation.

For example, the following change to the earlier code sample sets the name attribute to an array:

```
...
String[] names = {"Jim", "Kay", "Sriram"};
hello.setAttribute("name", names);
System.out.println(hello.toString());
```

The output is Hello JimKaySriram, but if you change the template definition to include a separator string, you can emit better-looking output:

```
StringTemplate hello =
    new StringTemplate("Hello <name; separator=\", \">");
```

Now, the output is Hello Jim, Kay, Sriram.

One final operation that you will see is *template application*. Applying a template to an attribute or multivalued attribute using the : operator is essentially a map operation. A map operation passes the elements in the attribute one at a time to the template as if it were a method. The following example applies an anonymous template to a list of numbers (anonymous templates are enclosed in curly brackets). The n variable defined between the left curly and the | operator is a parameter for the anonymous template. n is the iterated value moving through the numbers.

```
<numbers:{ n | sum += <n>; }>
```

Assuming the numbers attribute held the values 11, 29, and 5 (in a list, array, or anything else multivalued), then the template would emit the following when evaluated:

```
sum += 11; sum += 29; sum += 5;
```

Rather than having to create instances of templates in actions manually, grammars can use template construction rules, as demonstrated in the next section.

9.4 The ANTLR StringTemplate Interface

Grammars that set option output=template can use the -> operator to specify the enclosing rule's template return value. The basic syntax mirrors AST construction rules:

```
rule: «alt1» -> «alt1-template»
    | «alt2» -> «alt2-template»
   ...
    | «altN» -> «altN-template»
    ;
```

Each rule's template specification creates a template using the following syntax (for the common case):

```
... -> template-name(«attribute-assignment-list»)
```

The resulting recognizer looks up *template-name* in the StringTemplateGroup that you pass in from your invoking program via setTemplateLib(). A StringTemplateGroup is a group of templates and acts like a dictionary that maps template names to template definitions. The attribute assignment list is the interface between the parsing element values and the attributes used by the template. ANTLR translates the assignment list to a series of setAttribute() calls. For example, attribute assignment a={«*expr*»} translates to the following where retval.st is the template return value for the rule:

```
retval.st.setAttribute("a", «expr»);
```

As a more concrete example, consider the following grammar that matches simple assignments and generates an equivalent assignment using the Pascal := assignment operator:

Download templates/T.g

```
grammar T;
options {output=template;}
s : ID '=' INT ';' -> assign(x={$ID.text},y={$INT.text}) ;
ID: 'a'..'z'+ ;
INT:'0'..'9'+ ;
WS :(' '|'\t'|'\n'|'\r') {skip();} ;
```

Rule **s** matches a simple assignment statement and then creates an instance of template **assign**, setting its x and y attributes to the text of the identifier and integer, respectively. Here is a group file that defines template **assign**:

Download templates/T.stg

```
group T;
assign(x,y) ::= "<x> := <y>;"
```

The following code is a simple test rig that loads the StringTemplate group from a file (T.stg), instantiates the parser, invokes rule **s**, gets the result template, and prints it:

Download templates/Test.java

```
import org.antlr.runtime.*;
import org.antlr.stringtemplate.*;
import java.io.*;
```

```
public class Test {
    public static void main(String[] args) throws Exception {
        // load in T.stg template group, put in templates variable
        FileReader groupFileR = new FileReader("T.stg");
        StringTemplateGroup templates =
            new StringTemplateGroup(groupFileR);
        groupFileR.close();

        // PARSE INPUT AND COMPUTE TEMPLATE
        ANTLRInputStream input = new ANTLRInputStream(System.in);
        TLexer lexer = new TLexer(input);      // create lexer
        CommonTokenStream tokens = new CommonTokenStream(lexer);
        TParser parser = new TParser(tokens); // create parser
        parser.setTemplateLib(templates); // give parser templates
        TParser.s_return r = parser.s();      // parse rule s
        StringTemplate output = r.getTemplate();
        System.out.println(output.toString());// emit translation
    }
}
```

The test program accepts an assignment from standard input and emits the translation to standard output:

```
⇐ $ java Test
⇐ x=101;
⇐ EOF
⇒ x := 101;
$
```

Most of the time you will place your templates in another file (a StringTemplateGroup file[5]) to separate your parser from your output rules. This separation makes it easy to retarget your translator because you can simply swap out one group of templates for another to generate different output. For simple translators, however, you might want to specify templates in-line among the rules instead. In-line templates use notation like this:

```
... -> template(«attribute-assignment-list») "in-line-template"
```

or like this:

```
... -> template(«attribute-assignment-list»)
<<
«in-line-template-spanning-multiple-lines»
>>
```

5. See http://www.antlr.org/wiki/display/ST/Group+Files.

You can also use an arbitrary action that evaluates to a template as the template specification:

```
... -> {«arbitrary-template-expression»}
```

Now that you have some understanding of what templates look like and how to use them in a grammar, you're about ready to tackle some real examples. The first example generates Java bytecodes for simple arithmetic expressions. The second example instruments C code to track function calls and variable assignments. On the surface, the two applications appear to be similar because they both use templates to specify the output. What they do with the templates, however, is very different. The Java bytecode generator combines template results computed from input phrases into an overall template. In contrast, the C instrumentor gets away with just tweaking the input. For example, when it sees an assignment statement, the translator can simply append an instrumentation call. Rebuilding the whole C file complete with whitespace and comments is much more difficult. Before diving into the examples, let's examine the two fundamental translator categories they represent.

9.5 Rewriters vs. Generators

A translator's input to output relationship classifies it into one of two overall categories in the translation taxonomy: either the output looks very much like the input, or it does not, which says a lot about the most natural implementation, as we'll see in a moment. Examples abound for each category.

In the first category, the output looks very much like the input, and the translator is usually just tweaking the input or, when reading source code, instrumenting it for debugging, code coverage, or profiling purposes. I would include in this category translators that keep the same output structure and data elements but that do a fair bit of augmentation. A wiki-to-HTML translator is a good example of this. Let's call this the *rewriter* category.

Generally speaking, the more different the output is from the input, the more difficult it is to build the translator.

In the second category, the output looks very different from the input. The translator might condense the input into a report such as a set of source code metrics or might generate Javadoc comments from Java source code. Compilers and other tools in this category such as ANTLR generate an equivalent version of the input but a version that looks totally different because it has been highly processed and reorganized.

In other cases, the output is similar to the input, but the relative order of the elements is different in the output. For example, translating from a language where declaration order doesn't matter to a language where it does means that the translator must topologically sort the declarations according to their referential dependencies (declarations that refer to type *T* must occur after the declaration for *T*). Because the majority of the text emitted by translators in this category is not directly present in the input or has been significantly rebuilt, let's call this category the *generator* category.

The translation category often dictates which general approach to use:

- *Rewriters*. I recommend modifying input constructs in-place, meaning rewriting sections of the input buffer during translation. At the end of translation, you can simply print the modified buffer to get the desired output. ANTLR provides an excellent token buffer called TokenRewriteStream that is specifically designed to efficiently handle multiple in-place insertions, deletions, and replacements.
- *Generators*. This category generates and then buffers up bits of translated input that it subsequently organizes into larger and larger chunks, leading to the final chunk representing the complete output.

For both rewriters and generators, you'll use grammars to direct the translation. You'll embed actions and template construction rules (using operator ->) to perform computations and create pieces of the output. Rewriters will directly replace portions of the original input buffer with these output pieces, whereas generators will combine output pieces to form the overall output without touching the input buffer. Your translator might even have multiple grammars because, for complicated translations, you might need to break the problem down into multiple phases to make the translator easier to build and maintain. This means creating an AST intermediate form and making multiple passes over the AST to gather information and possibly to alter the AST.

How do you know whether you need multiple grammars (one parser grammar and multiple tree grammars) or whether you can get away with just a single parser grammar? Answering this question before attempting to build the actual translator requires a good deal of experience.

Generally speaking, though, you can usually get away with a single parser grammar if your translator can generate all output pieces without needing information from further ahead in the input stream and without needing information from another input file. For example, you can generate Javadoc comments just by looking at the method itself, and you can compute word and line counts by looking at the current and previous lines. On the other hand, compilers for object-oriented languages typically allow forward references and references to classes and methods in other files. The possibility of forward references alone is sufficient to require multiple passes, necessitating an intermediate form such as an AST and tree grammars.

The remainder of this chapter provides a Java bytecode generator as an example in the generator category and a code instrumentor as an example in the rewriter category. Let's begin with the generator category because the rewriter category is just a special case of a generator.

9.6 Building a Java Bytecode Generator Using a Tree Grammar and Templates

In Chapter 3, *A Quick Tour for the Impatient*, on page 59, we saw how to build an interpreter for some simple arithmetic expressions. This section illustrates how to build a Java bytecode generator for those same expressions.

As before, the input to the translator is a series of expressions and assignments. The output is a Java bytecode sequence in bytecode assembler format. Then Jasmin[6] can translate the text-based bytecode assembly program to a binary Java .class file. Once you have the .class file, you can directly execute expressions via the Java virtual machine. In a sense, the generator is a Java compiler for an extremely limited subset of Java. Naturally, a real compiler would be vastly more complicated, and I am not suggesting that you build compilers this way—generating Java bytecodes is merely an interesting and educational demonstration of template construction.

Before exploring how to build the generator, you need to know something about Java bytecodes and the underlying stack machine.

6. See http://jasmin.sourceforge.net.

Java Bytecodes Needed for Expressions

The executable instructions you will use for the translator are limited to the following:

Bytecode Instruction	Description
ldc *integer-constant*	Push constant onto stack.
imul	Multiply top two integers on stack and leave result on the stack. Stack depth is one less than before the instruction. Executes push(pop*pop).
iadd	Add top two integers on stack and leave result on the stack. Stack depth is one less than before the instruction. Executes push(pop+pop).
isub	Subtract top two integers on stack and leave result on the stack. Stack depth is one less than before the instruction. Executes b=pop; a=pop; push(a-b).
istore *local-var-num*	Store top of stack in local variable and pop that element off the stack. Stack depth is one less than before the instruction.
iload *local-var-num*	Push local variable onto stack. Stack depth is one more than before the instruction.

The Java bytecode generator accepts input:

Download templates/generator/1pass/input

```
3+4*5
```

and generates the following bytecodes:

```
ldc 3   ; push integer 3
ldc 4   ; push 4
ldc 5   ; push 5
imul    ; pop 2 elements, multiply; leave 4*5 on stack
iadd    ; pop 2 elements, 3 and (4*5), add, leave on stack
```

What about storing and retrieving local variables? This input:

Download templates/generator/1pass/input2

```
a=3+4
a
```

results in the following bytecodes:

Download templates/generator/1pass/input2.j

```
; code translated from input stream
; compute 3+4
ldc 3
ldc 4
iadd
istore 1 ; a=3+4
; compute a
iload 1 ; a
```

The Java virtual machine places local variables and method parameters on the stack. The bytecodes must allocate space for both at the start of each method. Also, the method must have space for the operands pushed onto the stack by the instructions. The proper way to compute stack space needed by a method is too much to go into here. The generator simply estimates how much operand stack space it will need according to the number of operator instructions.

All Java code must exist within a method, and all methods must exist within a class. The generator must emit the bytecode equivalent of this:

```
public class Calc {
    public static void main(String[]) {
        System.out.println(«executable-bytecodes-from-expression»);
    }
}
```

Unfortunately, the overall bytecode infrastructure just to wrap the bytecode in a main() method is fairly heavy. For example, given input 3+4*5, the generator must emit the following complete assembly file:

Download templates/generator/1pass/input.j

```
; public class Calc extends Object { ...}
.class public Calc
.super java/lang/Object

; public Calc() { super(); } // calls java.lang.Object()
.method public ()V
   aload_0
   invokenonvirtual java/lang/Object/()V
   return
.end method

; main(): Expr.g will generate bytecode in this method
.method public static main([Ljava/lang/String;)V
    .limit stack 4 ; how much stack space do we need?
    .limit locals 1 ; how many locals do we need?
```

```
    getstatic java/lang/System/out Ljava/io/PrintStream;
    ; code translated from input stream
    ; compute 3+4*5
    ldc 3
    ldc 4
    ldc 5
    imul
    iadd
    ; print result on top of stack
    invokevirtual java/io/PrintStream/println(I)V
    return
.end method
```

The generator modifies or emits the highlighted lines. The getstatic and invokevirtual bytecodes implement the System.out.println(*expr*) statement:

```
; get System.out on stack
getstatic java/lang/System/out Ljava/io/PrintStream;
«compute-expr-leaving-on-stack»
; invoke println on System.out (object 1 below top of stack)
invokevirtual java/io/PrintStream/println(I)V
```

To execute the generated code, you must first run the Jasmin assembler, which converts bytecode file input.j into Calc.class. Calc is the class definition surrounding the expression bytecodes. Installing Jasmin is easy. Just download and unzip the ZIP file. The key file is jasmin-2.3/jasmin.jar, which you can add to your CLASSPATH or just reference directly on the command line. Here is a sample session that "executes" the earlier input.j file using Jasmin and then the Java interpreter:

```
$ java -jar /usr/local/jasmin-2.3/jasmin.jar input.j
Generated: Calc.class
$ java Calc
23
$
```

That big file of bytecodes looks complicated to generate, but if you break it down into the individual pieces, the overall generator is straightforward. Each rule maps an input sentence phrase to an output template. Combining the templates results in the output assembly file. The following section shows how to implement the translator.

Generating Bytecode Templates with a Tree Grammar

This section shows you how to build a Java bytecode generator for a simple calculator language. The implementation uses the parser and tree grammars given in Chapter 3, *A Quick Tour for the Impatient*, on page 59. The parser, **Expr**, creates ASTs and fills a symbol table with variable definitions. A second pass, specified with tree grammar **Gen**,

constructs templates for the various subtrees. There is a template construction for just about every rule in a generator category translator. Generators must literally generate output for even the simplest input constructs such as comma-separated lists.

Generating templates behaves just like generating trees using rewrite rules.Each rule implicitly returns a template or tree, and the -> rewrite operator declares the output structure.

Grammar **Expr** is the same as the tree construction grammar given in Chapter 3, *A Quick Tour for the Impatient*, on page 59, except for the actions tracking the number of operations and computing the locals symbol table. Here is the grammar header that contains the infrastructure instance variables:

Download templates/generator/2pass/Expr.g

```
/** Create AST and compute ID -> local variable number map */
grammar Expr;
options {
    output=AST;
    ASTLabelType=CommonTree; // type of $stat.tree ref etc...
}

@header {
import java.util.HashMap;
}

@members {
int numOps = 0; // track operations for stack size purposes
HashMap locals = new HashMap(); // map ID to local var number
/* Count local variables, but don't use 0, which in this case
 * is the String[] args parameter of the main method.
 */
int localVarNum = 1;
}
```

The **prog** and **stat** rules create trees with AST rewrite rules as before:

Download templates/generator/2pass/Expr.g

```
prog:   stat+ ;

stat:   expr NEWLINE        -> expr
    |   ID '=' expr NEWLINE
        {
        if ( locals.get($ID.text)==null ) {
            locals.put($ID.text, new Integer(localVarNum++));
        }
        }
        -> ^('=' ID expr)
    |   NEWLINE             ->
    ;
```

Do You Ever Need to Return a List of Templates?

Some of you will see rules such as this:

```
prog : stat+ ;
```

and question why **slist** should not return a list of templates. Remember that for translators in the generator category, you must literally specify what to emit for every input construct even if it is just a list of input elements. The proper generated output for **prog** is in fact a template that represents a list of statements. For example, you might use something akin to this:

```
prog : (s+=stat)+ -> template(stats={$s}) "<stats>"
```

Upon an assignment statement, though, the recognizer must track implicit local variable definitions by using the locals HashMap. Each variable in a list of input expressions receives a unique local variable number.

The expression rules are as before except for the addition of code to track the number of operations (to estimate stack size):

Download templates/generator/2pass/Expr.g

```
expr:   multExpr (('+'^|'-'^) multExpr {numOps++;})*
    ;

multExpr
    :   atom ('*'^ atom {numOps++;})*
    ;

atom:   INT
    |   ID
    |   '('! expr ')'!
    ;
```

Once you have a parser grammar that builds the appropriate trees and computes the number of operations and locals map, you can pass that information to the tree grammar. The tree grammar will create a template for each subtree in order to emit bytecodes. Start rule **prog**'s template return value represents the template for the entire assembly file. The grammar itself is identical to the **Eval** tree grammar from Chapter 3, *A Quick Tour for the Impatient*, on page 59, but of course the actions are different.

Here is the start of the **Gen** tree grammar:

Download templates/generator/2pass/Gen.g

```
tree grammar Gen;

options {
    tokenVocab=Expr; // use the vocabulary from the parser
    ASTLabelType=CommonTree; // what kind of trees are we walking?
    output=template; // generate templates
}

@header {
import java.util.HashMap;
}

@members {
/** Points at locals table built by the parser */
HashMap locals;
}
```

The test rig pulls necessary data out of the parser after parsing is complete and passes it to the tree grammar via **prog** rule parameters:

Download templates/generator/2pass/Gen.g

```
/** Match entire tree representing the arithmetic expressions. Pass in
 *  the number of operations and the locals table that the parser computed.
 *  Number of elements on stack is roughly number of operations + 1 and
 *  add one for the address of the System.out object. Number of locals =
 *  number of locals + parameters plus 'this' if non-static method.
 */
prog[int numOps, HashMap locals]
@init {
this.locals = locals; // point at map created in parser
}
    :   (s+=stat)+ -> jasminFile(instructions={$s},
                                 maxStackDepth={numOps+1+1},
                                 maxLocals={locals.size()+1})
    ;
```

The **stat** rule creates an instance of template **exprStat** or **assign**, depending on which alternative matches:

Download templates/generator/2pass/Gen.g

```
stat:   expr -> exprStat(v={$expr.st}, descr={$expr.text})
    |   ^('=' ID expr)
        -> assign(id={$ID.text},
                  descr={$text},
                  varNum={locals.get($ID.text)},
                  v={$expr.st})
    ;
```

The template specifications compute template attributes from grammar attributes and members such as locals.

Here are the templates used by rule **stat**:

Download templates/generator/2pass/ByteCode.stg

```
assign(varNum,v,descr,id) ::= <<
; compute <descr>
<v>
istore <varNum> ; <id>
>>

exprStat(v, descr) ::= <<
; compute <descr>
<v>
>>
```

All the expression-related rules in the parser grammar collapse into a single **expr** rule in the tree grammar:

Download templates/generator/2pass/Gen.g

```
expr returns [int value]
    :   ^('+' a=expr b=expr) -> add(a={$a.st},b={$b.st})
    |   ^('-' a=expr b=expr) -> sub(a={$a.st},b={$b.st})
    |   ^('*' a=expr b=expr) -> mult(a={$a.st},b={$b.st})
    |   INT -> int(v={$INT.text})
    |   ID  -> var(id={$ID.text}, varNum={locals.get($ID.text)})
    ;
```

Each subtree results in a different template instance; here are the template definitions used by rule **expr**:

Download templates/generator/2pass/ByteCode.stg

```
add(a,b) ::= <<
<a>
<b>
iadd
>>

sub(a,b) ::= <<
<a>
<b>
isub
>>

mult(a,b) ::= <<
<a>
<b>
imul
>>

int(v) ::= "ldc <v>"

var(id, varNum) ::= "iload <varNum> ; <id>"
```

The operation templates do the obvious: push both operands and then do the operation.

Finally, here is a test rig with in-line comments to explain the various operations:

Download templates/generator/2pass/Test.java

```
import org.antlr.runtime.*;
import org.antlr.runtime.tree.*;
import org.antlr.stringtemplate.*;
import java.io.*;

public class Test {
    public static void main(String[] args) throws Exception {
        // load the group file ByteCode.stg, put in templates var
        FileReader groupFileR = new FileReader("ByteCode.stg");
        StringTemplateGroup templates =
            new StringTemplateGroup(groupFileR);
        groupFileR.close();

        // PARSE INPUT AND BUILD AST
        ANTLRInputStream input = new ANTLRInputStream(System.in);
        ExprLexer lexer = new ExprLexer(input);     // create lexer
        // create a buffer of tokens pulled from the lexer
        CommonTokenStream tokens = new CommonTokenStream(lexer);
        ExprParser parser = new ExprParser(tokens); // create parser
        ExprParser.prog_return r = parser.prog();   // parse rule prog

        // WALK TREE
        // get the tree from the return structure for rule prog
        CommonTree t = (CommonTree)r.getTree();
        // create a stream of tree nodes from AST built by parser
        CommonTreeNodeStream nodes = new CommonTreeNodeStream(t);
        // tell it where it can find the token objects
        nodes.setTokenStream(tokens);
        Gen walker = new Gen(nodes); // create the tree Walker
        walker.setTemplateLib(templates); // where to find templates
        // invoke rule prog, passing in information from parser
        Gen.prog_return r2 = walker.prog(parser.numOps, parser.locals);

        // EMIT BYTE CODES
        // get template from return values struct
        StringTemplate output = r2.getTemplate();
        System.out.println(output.toString()); // render full template
    }
}
```

The one key piece that is different from a usual parser and tree parser test rig is that, in order to access the text for a tree, you must tell the tree node stream where it can find the token stream:

```
CommonTokenStream tokens = new CommonTokenStream(lexer);
...
CommonTreeNodeStream nodes = new CommonTreeNodeStream(t);
nodes.setTokenStream(tokens);
```

References such as $expr.text in rule **stat** in the tree parser make no sense unless it knows where to find the stream of token objects. Recall that $expr.text provides the text matched by the parser rule that created this subtree.

In this section, you learned how to generate output by adding template construction rules to a tree grammar. This generator category translator emits an output phrase for every input phrase. In the following two sections, you'll learn how to rewrite pieces of the input instead of generating an entirely new output file. You'll again use template construction rules, but this time the templates will replace input phrases.

9.7 Rewriting the Token Buffer In-Place

Traditionally, one of the most difficult tasks in language translation has been to read in some source code and write out a slightly altered version, all while preserving whitespace and comments. This is hard because translators typically throw out whitespace and comments since the parser has to ignore them. Somehow you must get the parser to ignore those tokens without throwing them away and without losing their original place in the input stream.

A further hassle involves the mechanism used to emit the tweaked source code. As you saw earlier, adding a bunch of print statements to your grammar is not a very good solution. Even using templates and the generator strategy, you have to specify a lot of unnecessary template construction rules. The generated recognizer performs a lot of unnecessary work. If the output is identical to the input most of the time, adding template construction actions to each rule is highly inefficient and error prone. You need a specification whose size is commensurate with the amount of work done by the translator.

In this section and the next, you'll see two different solutions to a source code rewriting problem, one using a parser and the next using a parser and tree parser combination. The task is to instrument simplified C source code with code snippets that track function calls and variable assignments. The translator must pass through whitespace and comments unmolested.

For example, it must rewrite this input:

Download templates/rewriter/1pass/input

```
int x;
/* A simple function foo */
void foo() {
    int y;
    y = 1;
    /* start */ g(34,y); /* end of line comment */
    x = h(/* inside */);
}
```

to be the following:

Download templates/rewriter/1pass/output

```
int x;
/* A simple function foo */
void foo() {
    int y;
    y = 1; write_to("y",y);
    /* start */ g(34,y); call("g"); /* end of line comment */
    x = eval("h",h(/* inside */)); write_to("x",x);
}
```

where call() tracks "procedure" calls, eval tracks function calls, and write_to() tracks variable assignments. The rule that translates function calls will have to use context to know whether the invocation is part of an assignment or is an isolated "procedure" (function without a return value) statement.

The way you'll solve this code instrumentation problem is with ANTLR's *rewrite mode* (see Section 5.6, *rewrite Option*, on page 124). This mode automatically copies the input to the output except where you have specified translations using template rewrite rules. In other words, no matter how big your grammar is, if you need to translate only one input phrase, you will probably need to specify only one template rewrite rule.

The "magic" behind ANTLR's rewrite mode is a fiendishly clever little token buffer called TokenRewriteStream that supports insert, delete, and replace operations. As recognition progresses, actions in your grammar or template rewrite rules can issue instructions to the token stream that are queued up as commands. The stream does not actually execute the commands until you invoke toString(), which avoids any physical modification of the token buffer. As toString() walks the buffer emitting each token's text, it looks for commands to be executed at that particular token index.

How ANTLR Differs from Perl and awk for Rewriting

Those familiar with common text-processing tools such as Perl or the Unix utility awk (or even sed) might disagree with me that tweaking source code is difficult while preserving whitespace and comments. Those tools are, in fact, specifically designed for translation, but lexical, not grammatical, translation. They are good at recognizing and translating structures that you can identify with regular expressions but not so good at recognizing grammatical structure, while ANTLR is. You can look at it like ANTLR is designed to handle "heavy lifting" translation tasks, but Perl and sed are better for simple tasks.

That is not to say that you can't amaze your family and friends with awk. For example, here's a Unix one-liner that generates Java class hierarchy diagrams in DOT (graphviz) format using only the Unix utilities cat, grep, and awk (try it—it's amazing!):

```
# pulls out superclass and class as $5 and $3:
# public class A extends B . . .
# only works for public classes and usual formatting
# Run the output into dot tool
cat *.java | grep 'public class' $1 | \
    awk 'BEGIN {print "digraph foo {";} \
        {print $5 "->" $3;} \
        END {print "}"}'
```

Although this is impressive, it is not perfect because it cannot handle class definitions on multiple lines and so on. More important, the strategy is not scalable because these tools do not have grammatical context like ANTLR does. Consider how you would alter the command so that it ignored inner classes. You would have to add a context variable that says "I am in a class definition already," which would require knowing when class definitions begin and end. Computing that requires that you track open and close curlies, which means ignoring them inside multiline comments and strings; eventually you will conclude that using a formal grammar is your best solution. Complicated translators built without the benefit of a grammar are usually hard to write, read, and maintain.

If the command is a replace operation, for example, toString() simply emits the replacement text instead of the original token (or token range). Because the operations are done lazily at toString() execution time, commands do not scramble the token indexes. An insert operation at token index *i* does not change the index values for tokens *i*+1..*n-1*. Also, because operations never actually alter the buffer, you can always get back the original token stream, and you can easily simulate transactions to roll back any changes. You can also have multiple command queues to get multiple rewrites from a single pass over the input such as generating both a C file and its header file (see the TokenRewriteStream Javadoc for an example).

To recognize the small subset of C, you will use a variation of the **CMinus** grammar from Section 8.2, *Building a Parser Grammar for the C-Language*, on page 195 (this variation adds function calls and removes a few statements). The grammar starts with the options to generate templates and turn on rewrite mode:

Download templates/rewriter/1pass/CMinus.g

```
grammar CMinus;
options {output=template; rewrite=true;}
```

The declarations are the same:

Download templates/rewriter/1pass/CMinus.g

```
program
    :   declaration+
    ;

declaration
    :   variable
    |   function
    ;

variable
    :   type ID ';'
    ;

function
    :   type ID '(' ( formalParameter (',' formalParameter)* )? ')' block
    ;

formalParameter
    :   type ID
    ;

type:   'int'
    |   'void'
```

The translator must rewrite assignments. The alternative that matches assignments in rule **stat** must, therefore, have a template rewrite rule:

Download templates/rewriter/1pass/CMinus.g

```
stat
scope {
boolean isAssign;
}
    :   expr ';'
    |   block
    |   ID '=' {$stat::isAssign=true;} expr ';'
        -> template(id={$ID.text},assign={$text})
           "<assign> write_to(\"<id>\",<id>);"
    |   ';'
    ;

block
    :   '{' variable* stat* '}'
    ;
```

Because this application is going from C to C and the number of templates is very small, it is OK to in-line the templates rather than referencing templates stored in a group file as the bytecode generator example did. The assignment rewrite just emits the original assignment (available as $text, the text matched for the entire rule) and then makes a call to write_to() with the name of the variable written to. Rule **stat** tracks whether it is in the process of recognizing an assignment. Rule **stat**'s dynamic scope makes that context information available to the rules it invokes directly or indirectly.

The **expr** rule must translate function calls in two different ways, depending on whether the expression is the right side of an assignment or an isolated procedure call in an expression statement:

Download templates/rewriter/1pass/CMinus.g

```
expr:   ID
    |   INT
    |   ID '(' ( expr (',' expr)* )? ')'
        -> {$stat::isAssign}? template(id={$ID.text},e={$text})
           "eval(\"<id>\",<e>)"  // rewrite ...=f(3) as eval("f",f(3))
        -> template(id={$ID.text},e={$text})
           "<e>; call(\"<id>\")" // rewrite ...f(3) as f(3); call("f")
    |   '(' expr ')'
    ;
```

As with AST rewrite rules, you can prefix rewrites with semantic predicates that indicate which of the rewrite rules to apply. Both templates spit the expression back out verbatim but with surrounding instrumentation code. Note that the text for the function call, available as

$text, will include any whitespace and comments matched in between the first and last real (nonhidden) tokens encountered by rule **expr**. The translator can use the usual token definitions:

Download templates/rewriter/1pass/CMinus.g

```
ID  :   ('a'..'z'|'A'..'Z'|'_') ('a'..'z'|'A'..'Z'|'0'..'9'|'_')*
    ;

INT :   ('0'..'9')+
    ;

CMT :   '/*' (options {greedy=false;} : .)* '*/' {$channel = HIDDEN;} ;

WS  :   ( ' ' | '\t' | '\r' | '\n' )+ {$channel = HIDDEN;}
    ;
```

The only thing to highlight in the token rules is that they must pass the whitespace and comments tokens to the parser on a hidden channel rather than throwing them out (with skip()). This is an excellent solution because, in order to tweak two constructs, only two rules of the grammar need template specifications. Perl aficionados might ask why you need all the other grammar rules. Can't you just look for function call patterns that match and translate constructs like g(34,y)? The unfortunate answer is no. The expressions within the argument list can be arbitrarily complex, and the regular expressions of Perl cannot match nested parentheses, brackets, and so on, that might appear inside the argument list. Avoiding code in comments and strings further complicates the issue. Just to drive the point home, if you change the problem slightly so that the translator should track only global variable access, not local variables and parameters, you will need a symbol table and context to properly solve the problem. The best you can do in general is to use a complete grammar with a rewrite mode that minimizes programmer effort and maximizes translator efficiency. For completeness, here is the core of a test rig:

Download templates/rewriter/1pass/Test.java

```
ANTLRInputStream input = new ANTLRInputStream(System.in);
CMinusLexer lexer = new CMinusLexer(input);
// rewrite=true only works with TokenRewriteStream
// not CommonTokenStream!
TokenRewriteStream tokens = new TokenRewriteStream(lexer);
CMinusParser parser = new CMinusParser(tokens);
parser.program();
System.out.println(tokens.toString()); // emit rewritten source
```

The next section solves the same problem using a two-pass parser and tree parser combination.

9.8 Rewriting the Token Buffer with Tree Grammars

Complicated translators need to use a multipass approach. In this situation, the parser grammar merely builds ASTs that one or more tree grammars examine. A final tree grammar pass must perform the rewriting instead of the parser grammar. Fortunately, tree grammars can generate templates just as easily as parser grammars because tree grammars also have access to the TokenRewriteStream object (via the TreeNodeStream).

This section solves the same source code instrumentation problem as the previous section but uses the parser grammar to build ASTs. We'll then walk the trees with a tree parser to perform the rewrites using the TokenRewriteStream. Let's start by building the AST for your simple subset of C. Here is the parser grammar again from Section 8.2, *Building a Parser Grammar for the C- Language*, on page 195 for your convenience:

Download templates/rewriter/2pass/CMinus.g

```
grammar CMinus;
options {output=AST;}

tokens {
  VAR;   // variable definition
  FUNC;  // function definition
  ARG;   // formal argument
  SLIST; // statement list
  CALL;  // function call
}
```

Download templates/rewriter/2pass/CMinus.g

```
program
    :   declaration+
    ;

declaration
    :   variable
    |   function
    ;

variable
    :   type ID ';' -> ^(VAR type ID)
    ;

function
    :   type ID '(' ( formalParameter (',' formalParameter)* )? ')' block
        -> ^(FUNC type ID formalParameter* block)
    ;
```

```
formalParameter
    :   type ID -> ^(ARG type ID)
    ;

type:   'int'
    |   'void'
    ;
```

This variation of the grammar has a simplified **stat** rule—the **for** statement is gone:

Download templates/rewriter/2pass/CMinus.g

```
block
    :   lc='{' variable* stat* '}'
        -> ^(SLIST[$lc,"SLIST"] variable* stat*)
    ;

stat
    :   expr ';'!
    |   block
    |   ID '=' expr ';' -> ^('=' ID expr)
    |   ';'!
    ;
```

Rule **expr** has an additional alternative to match function calls and generate ASTs with the CALL imaginary node:

Download templates/rewriter/2pass/CMinus.g

```
expr:   ID
    |   INT
    |   ID '(' ( expr (',' expr)* )? ')' -> ^(CALL ID expr*)
    |   '(' expr ')' -> expr
    ;
```

Here is the relevant piece of the test rig that launches the parser to build an AST:

Download templates/rewriter/2pass/Test.java

```
// PARSE INPUT AND BUILD AST
ANTLRInputStream input = new ANTLRInputStream(System.in);
CMinusLexer lexer = new CMinusLexer(input);       // create lexer
// create a buffer of tokens pulled from the lexer
// Must use TokenRewriteStream not CommonTokenStream!
TokenRewriteStream tokens = new TokenRewriteStream(lexer);
CMinusParser parser = new CMinusParser(tokens);    // create parser
CMinusParser.program_return r = parser.program(); // parse program
```

Once you have a parser that builds valid ASTs, you can build the tree grammar that alters the token buffer. The template rewrite rules to tweak assignments and function calls are identical to the rewrite rules in the parser grammar from the last section. You can simply move them

from the parser to the tree parser at the equivalent grammar locations. The tree grammar starts out very much like the tree grammar above in Section 9.6, *Generating Bytecode Templates with a Tree Grammar*, on page 222 with the addition of the rewrite=true option:

Download templates/rewriter/2pass/Gen.g

```
tree grammar Gen;

options {
  tokenVocab=CMinus; // import tokens from CMinus.g
  ASTLabelType=CommonTree;
  output=template;
  rewrite=true;
}
```

The tree matching rules for declarations do not have any template rewrite rules because the translator does not need to tweak those phrases:[7]

Download templates/rewriter/2pass/Gen.g

```
program
    :   declaration+
    ;

declaration
    :   variable
    |   function
    ;

variable
    :   ^(VAR type ID)
    ;

function
    :   ^(FUNC type ID formalParameter* block)
    ;

formalParameter
    :   ^(ARG type ID)
    ;

type:   'int'
    |   'void'
    ;
```

7. Unlike the parser grammar solution, you might be able to get away with just the tree grammar rules that specify rewrite templates. The AST is a highly processed version of the token stream, so it is much easier to identify constructs of interest. Future research should yield a way to avoid having to specify superfluous tree grammar rules for rewriting applications.

Rule **stat** defines the dynamic scope with isAssign like the parser grammar did. **stat** also specifies a template that rewrites the text from which the parser created the AST. In other words, parser grammar rule **stat** builds an assignment AST with token '=' at the root. **stat** records the token range it matches in its result AST's root node. For input x=3;, the root of the AST would contain token index range *i*..*i*+3 where *i* is the index of the x. Rule **stat** in the tree grammar then also knows this token range just by asking the root of the AST it matches. This is how template construction rewrite rules in a tree grammar know which tokens to replace in the input stream.

Download templates/rewriter/2pass/Gen.g

```
block
    :   ^(SLIST variable* stat*)
    ;

stat
scope {
boolean isAssign;
}
    :   expr
    |   block
    |   ^('=' ID {$stat::isAssign=true;} expr)
        -> template(id={$ID.text},assign={$text})
           "<assign> write_to(\"<id>\",<id>);"
    ;
```

A word of caution: if an alternative matches a list instead of a single subtree, ANTLR will not give you the right result for $text. That attribute expression is defined as the text for the first subtree a rule matches. This might seem a little odd, but it works well in practice.

Rule **expr** performs the same translation based upon context as did the solution in the previous section:

Download templates/rewriter/2pass/Gen.g

```
expr:   ID
    |   INT
    |   ^(CALL ID expr*)
        -> {$stat::isAssign}? template(id={$ID.text},e={$text})
           "eval(\"<id>\",<e>)"
        -> template(id={$ID.text},e={$text})
           "<e> call(\"<id>\")"
    ;
```

Here is the final piece of the test rig that walks the tree, rewrites the token buffer, and finally emits the altered token buffer:

Download templates/rewriter/2pass/Test.java

```
// WALK TREE AND REWRITE TOKEN BUFFER
// get the tree from the return structure for rule prog
CommonTree t = (CommonTree)r.getTree();
// create a stream of tree nodes from AST built by parser
CommonTreeNodeStream nodes = new CommonTreeNodeStream(t);
// tell it where it can find the token objects
nodes.setTokenStream(tokens);
Gen gen = new Gen(nodes);
gen.program(); // invoke rule program
System.out.println(tokens.toString()); // emit tweaked token buffer
```

At this point, you've seen one generator and two rewriter category examples. You know how to use template construction rules in both a parser grammar and a tree grammar. The type of grammar to use depends on the complexity of a translator. More complicated translators need multiple passes and, consequently, create templates in tree grammars.

For the most part, the examples given in this chapter avoid constructing templates with actions and do not set template attributes manually. Sometimes, however, the translation is complicated enough that you want to use arbitrary actions to create or modify templates. The next section describes the special symbols related to templates that you can use in actions as shorthand notation.

9.9 References to Template Expressions within Actions

In some rare cases, you might need to build templates and set their attributes in actions rather than using the template rewrite rules. For your convenience in this situation, ANTLR provides some template shorthand notation for use in embedded actions. All template-related special symbols start with % to distinguish them from normal attribute notation that uses $.

Consider the following grammar that uses a template rewrite rule:

```
s : ID '=' INT ';' -> assign(x={$ID.text},y={$INT.text})
  ;
```

Syntax	Description
%*foo*(*a*={},*b*={},. . .)	Template construction syntax. Create instance of template **foo**, setting attribute arguments *a*, *b*, ...
%({«*nameExpr*»})(*a*={},. . .)	Indirect template constructor reference. *nameExpr* evaluates to a String name that indicates which template to instantiate. Otherwise, it is identical to the other template construction syntax.
%*x.y* = «*z*»;	Set template attribute *y* of template *x* to *z*. Languages such as Python without semicolon statement terminators must still use them here. The code generator is free to remove them during code generation.
%{«*expr*»}.*y* = «*z*»;	Set template attribute *y* of StringTemplate typed expression *expr* to expression *z*.
%{«*stringExpr*»}	Create an anonymous template from String *stringExpr*.

Figure 9.1: Template shorthand notation available in grammar actions

You can get the same functionality using an action almost as tersely:

```
s : ID '=' INT ';' {$st = %assign(x={$ID.text},y={$INT.text});}
  ;
```

Or, you can create the template and then manually set the attributes using the attribute assignment notation:

```
s : ID '=' INT ';'
    {
    $st = %assign();    // set $st to instance of assign
    %{$st}.x=$ID.text;  // set attribute x of $st
    %{$st}.y=$INT.text;
    }
  ;
```

ANTLR translates that action to the following:

```
retval.st = templateLib.getInstanceOf("assign");
(retval.st).setAttribute("x",ID3.getText());
(retval.st).setAttribute("y",INT4.getText());
```

The table in Figure 9.1 summarizes the template shorthand notation available to you in embedded actions.

This chapter demonstrated that using StringTemplate templates in conjunction with ANTLR grammars is a great way to build both generators and rewriters. ANTLR v3's new rewrite mode is especially effective for those applications that need to tweak the input slightly but otherwise pass most of it through to the output.

The next chapter discusses an important part of any professional translator: error reporting and recovery.

Chapter 10

Error Reporting and Recovery

The quality of a language application's error messages and recovery strategy often makes the difference between a professional application and an amateurish application. Error recovery is the process of recovering from a syntax error by altering the input stream or consuming symbols until the parser can restart in a known state. Many hand-built and many non-*LL*-based recognizers emit less than optimal error messages, whereas ANTLR-generated recognizers automatically emit very good error messages and recover intelligently, as shown in this chapter. ANTLR's error handling facility is even useful during development.

During the development cycle, you will have a lot of mistakes in your grammar. The resulting parser will not recognize all valid sentences until you finish and debug your grammar. In the meantime, informative error messages help you track down grammar problems. Once you have a correct grammar, you then have to deal with ungrammatical sentences entered by users or even ungrammatical sentences generated by other programs gone awry.

In both situations, the manner in which your parser responds to ungrammatical input is an important productivity consideration. In other words, a parser that responds with "Eh?" and bails out upon the first syntax error is not very useful during development or for the people who have to use the resulting parser. For example, some SQL engines can only tell you the general vicinity where an error occurred rather than exactly what is wrong and where, making query development a trial-and-error process.

Developers using ANTLR get a good error reporting facility and a sophisticated error recovery strategy for free. ANTLR automatically generates recognizers that emit rich error messages upon syntax error and successfully resynchronize much of the time. The recognizers even avoid generating more than a single error message for each syntax error. *New in v3.*

This chapter describes the automatic error reporting and recovery strategy used by ANTLR-generated recognizers and shows how to alter the default mechanism to suit your particular needs.

10.1 A Parade of Errors

The best way to describe ANTLR's error recovery strategy is to show you how ANTLR-generated recognizers respond to the most common syntax errors: mismatched token, no viable alternative, and early exit from an EBNF (...)+ subrule loop. Consider the following grammar for simple statements and expressions, which we'll use as the core for the examples in this section and the remainder of the chapter:

Download errors/E.g

```
grammar E;

prog:   stat+ ;

stat:   expr ';'
        {System.out.println("found expr: "+$stat.text);}
    |   ID '=' expr ';'
        {System.out.println("found assign: "+$stat.text);}
    ;

expr:   multExpr (('+'|'-') multExpr)*
    ;

multExpr
    :   atom ('*' atom)*
    ;

atom:   INT
    |   '(' expr ')'
    ;

ID  :   ('a'..'z'|'A'..'Z')+ ;
INT :   '0'..'9'+ ;
WS  :   (' '|'\t'|'\n'|'\r')+ {skip();} ;
```

And here is the usual test rig that invokes rule **prog**:

Download errors/TestE.java

```
import org.antlr.runtime.*;

public class TestE {
    public static void main(String[] args) throws Exception {
        ANTLRInputStream input = new ANTLRInputStream(System.in);
        ELexer lexer = new ELexer(input);
        CommonTokenStream tokens = new CommonTokenStream(lexer);
        EParser parser = new EParser(tokens);
        parser.prog();
    }
}
```

First run some valid input into the parser to figure out what the normal behavior is:

```
⇐ $ java TestE
⇐ (3);
⇐ EOF
⇒ found expr: (3);
$
```

Upon either expression statement or assignment statement, the translator prints a message indicating the text matched for rule **stat**. In this case, (3); is an expression, not an assignment, as shown in the output.

Now, leaving off the final), the parser detects a mismatched token because rule **atom** was looking for the right parenthesis to match the left parenthesis:

```
⇐ $ java TestE
⇐ (3;
⇐ EOF
⇒ line 1:2 mismatched input ';' expecting ')'
found expr: (3;
$
```

The line 1:2 component of the error message indicates that the error occurred on the first line and at the third character position in that line (indexed from zero, hence, index 2).

Generating that error message is straightforward, but how does the parser successfully match the ; and execute the print action in the first alternative of rule **stat** after getting a syntax error all the way down in **atom**? How did the parser successfully recover from that mismatched token to continue as if nothing happened? This error recovery feature is called *single token insertion* because the parser pretends to insert the

missing) and keep going. We'll examine the mechanism in Section 10.7, *Recovery by Single Symbol Insertion*, on page 258. Notice that with multiple expressions, the parser successfully continues and matches the second alternative without error:

```
⇐ $ java TestE
⇐ (3;
⇐ 1+2;
⇐ EOF
⇒ line 1:2 mismatched input ';' expecting ')'
found expr: (3;
found expr: 1+2;
$
```

ANTLR also avoids generating cascading error messages if possible. That is, recognizers try to emit a single error message for each syntax error. In the following sample run, the first expression has two errors: the missing) and the missing ;. The parser normally emits only the first error message, suppressing the second message that has the [SPURIOUS] prefix:

```
⇐ $ java TestE
⇐ (3
⇐ a=1;
⇐ EOF
⇒ line 2:0 mismatched input 'a' expecting ')'
[SPURIOUS] line 2:0 mismatched input 'a' expecting ';'
found expr: (3
found assign: a=1;
$
```

Another common syntax error occurs when the parser is at a decision point and the current lookahead is not consistent with any of the alternatives of that rule or subrule. For example, the decision in rule **atom** must choose between an integer and the start of a parenthesized expression. Input `1+;` is missing the second operand, and rule **atom** would see `;` instead, causing a "no viable alternative exception:"

```
⇐ $ java TestE
⇐ 1+;
⇐ EOF
⇒ line 1:2 no viable alternative at input ';'
found expr: 1+;
$
```

The parser successfully recovers by scanning ahead to look for a symbol that can follow a reference to **atom** or a rule that has invoked **atom**. In this case, the `;` is a viable symbol following a reference to **atom** (and therefore **expr**). The parser consumes no tokens and simply exits from

Back When You Could Almost Parse C++

In the early 1990s, I was consulting at NeXT and was helping Sumana Srinivasan build a C++ code browser using ANTLR v1 (ANTLR is still used in NeXTStep, er, Mac OS X today). The manager, Steve Naroff, insisted that the ANTLR-generated parser provide the same high-quality error messages as his hand-built C parser did. Because of this, I introduced the notion of parser exception handling (the analog of the familiar programming exception handling) and created a simple single-token deletion mechanism. Ironically, the ANTLR-generated C++ recognizer emitted better messages in some circumstances than the hand-built parser because ANTLR never got tired of computing token sets and generating error recovery code—humans, on the other hand, often get sick of this tedious job.

You will also run into early subrule exit exceptions where a one-or-more (...)+ subrule matched no input. For example, if you send in an empty input stream, the parser has nothing to match in the stat+ loop:

```
⇐ $ java TestE
⇐ E0F
⇒ line 0:-1 required (..)+ loop did not match anything at input '<EOF>'
$
```

The line and character position information for EOF is meaningless; hence, you see the odd 0:-1 position information.

This section gave you a taste of ANTLR's default error reporting and recovery capabilities (see Section 10.7, *Automatic Error Recovery Strategy*, on page 256 for details about the automatic error recovery mechanism). The next few sections describe how you can alter the standard error messages to help with grammar debugging and to provide better messages for your users.

10.2 Enriching Error Messages during Debugging

By default, recognizers emit error messages that are most useful to users of your software. The messages include information only about what was found and what was expected such as in the following:

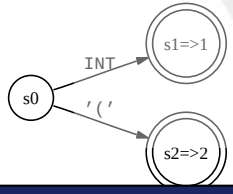
```
line 10:22 mismatched input INT expecting ID
```

Unfortunately, your grammar has probably 200 references to token **ID**. Where in the grammar was the parser when it found the **INT** instead of the **ID**? You can use the debugger in ANTLRWorks to set a breakpoint upon exception and then just look to see where in the grammar the parser is. Sometimes, though, sending text error messages to the console can be more convenient because you do not have to start the debugger.

With a little bit of work, you can override the standard error reporting mechanism to include information about the rule invocation stack. The invocation stack is the nested list of rules entered by the parser at any given moment, that is, the stack trace. You can also add more information about the mismatched token. For no viable alternative errors, you can do even more. For example, the following run illustrates a rich, no viable alternative error message that is much more useful for debugging grammars than the default:

```
⇐  $ java TestE2
⇐  1+;
⇐  EOF
⇒  line 1:2 [prog, stat, expr, multExpr, atom] no viable alternative,
   token=[@2,2:2=';',<7>,1:2] (decision=5 state 0)
   decision=<<35:1: atom : ( INT | '(' expr ')' );>>
   found expr: 1+;
   $
```

The message includes a rule invocation stack trace where the last rule mentioned is the rule the parser was in when it encountered the syntax error. The error includes a detailed report on the token itself that includes the token index, a character index range into the input stream, the token type, and the line and character position within the line. Finally, for no viable alternative exceptions such as this, the message includes information about the decision in a grammar: the decision number, the state within the decision's lookahead DFA, and a chunk of the grammar that describes the decision. To use the decision and state information, turn on ANTLR option -dfa, which will generate DOT (graphviz) descriptions you can display. Filenames are encoded with the grammar name and the decision number, so, for example, the DOT file for decision 5 of grammar **E** is E_dec-5.dot and looks like the following:



The state 0 mentioned in the error message is s0 in the diagram. In this case, the parser had a lookahead of ; that clearly does not match either alternative emanating from s0; hence, you have the no viable alternative exception.

To get these rich error messages, override two methods from BaseRecognizer, displayRecognitionError() and getTokenErrorDisplay(), where the grammar itself stays the same:

Download errors/E2.g

```
grammar E2;

@members {
public String getErrorMessage(RecognitionException e,
                               String[] tokenNames)
{
    List stack = getRuleInvocationStack(e, this.getClass().getName());
    String msg = null;
    if ( e instanceof NoViableAltException ) {
       NoViableAltException nvae = (NoViableAltException)e;
       msg = " no viable alt; token="+e.token+
          " (decision="+nvae.decisionNumber+
          " state "+nvae.stateNumber+")"+
          " decision=<<"+nvae.grammarDecisionDescription+">>";
    }
    else {
       msg = super.getErrorMessage(e, tokenNames);
    }
    return stack+" "+msg;
}
public String getTokenErrorDisplay(Token t) {
    return t.toString();
}
}
```

The next section describes how to improve error messages for your users rather than for yourself during debugging.

10.3 Altering Recognizer Error Messages

This section describes the information available to you when generating error messages and provides an example that illustrates how to enrich error messages with context information from the grammar.For each problem that can occur during sentence recognition, the recognizer creates an exception object derived from RecognitionException.

RecognitionException
: The superclass of all exceptions thrown by an ANTLR-generated recognizer. It tracks the input stream; the index of the symbol (character, token, or tree node) the recognizer was looking at when the error occurred; the erroneous symbol pointer (int, Token, or Object); the line; and the character position within that line.

MismatchedTokenException
: Indicates that the parser was looking for a particular symbol that it did not find at the current input position. In addition to the usual fields, this object tracks the expected token type (or character code).

MismatchedTreeNodeException
: Indicates that the tree parser was looking for a node with a particular token type and did not find it. This is the analog of a mismatched token exception for a token stream parser. It tracks the expected token type.

NoViableAltException
: The recognizer came to a decision point, but the lookahead was not consistent with any of the alternatives. It tracks the decision number and state number within the lookahead DFA where the problem occurred and also stores a chunk of the grammar from which ANTLR generated the decision.

EarlyExitException
: The recognizer came to a (..)+ EBNF subrule that must match an alternative at least once, but the subrule did not match anything. It tracks the decision number but not the state number because it is obviously not in the middle of the lookahead DFA; the whole thing was skipped.

FailedPredicateException
: A validating semantic predicates evaluated to false. It tracks the name of the rule in which the predicate failed as well as the text of the predicate itself from your grammar.

MismatchedRangeException
: The recognizer tried to match a range of symbols, usually characters, but could not. It tracks the minimum and maximum element in the range.

MismatchedSetException
: The recognizer attempted to match a set of symbols but could not. It tracks the set of elements in which the recognizer was interested.

MismatchedNotSetException
: The recognizer attempted to match the inverse of a set (using the ~ operator) but could not.

Figure 10.1: ANTLR recognition exceptions

These exception objects contain information about what was found on the input stream, what was expected, and sometimes information about the location in the grammar associated with the erroneous parser state.

To avoid forcing English-only error messages and to generally make things as flexible as possible, the recognizer does not create exception objects with string messages. Instead, it tracks the information necessary to generate an error.

Then the various reporting methods in BaseRecognizer generate a localized error message, or you can override them. Do not expect the exception getMessage() methods to return anything. The table in Figure 10.1, on the preceding page, summarizes the exception classes and the information they contain.

Improved in v3.

Beyond the information in these exception objects, you can collect any useful information you want via actions in the grammar and then use it to provide better error messages for your users.

One of the most useful enhancements to error messages is to include information about the kind of abstract construct the parser was recognizing when it encountered an error. For example, instead of just saying "missing **ID**," it is better to say "missing **ID** in expression." You could use the literal rule name such as "multExpr," but that is usually meaningless to users.

You can think of this as a *paraphrase* mechanism because you are representing a collection of grammar rules with a short description. What you want is a map from all rules associated with a particular abstract language construct (that is, declarations, statements, and expressions) to a user-friendly string such as "expression."

*In v2, there was a **paraphrase** option that automated this.*

The easiest way to implement a paraphrase mechanism is to push a string onto a stack when you enter a rule that represents an abstract construct in a language and then pop the value off when leaving the rule. Do not push a paraphrase string for all rules. Just push a paraphrase for the top-level rule such as **expr**, but not **multExpr** or **atom**.

First, define a stack of paraphrases and override getErrorMessage() to include the paraphrase at the end:

Download errors/P.g

```
/** This grammar demonstrates creating a "paraphrase" error reporting option. */
grammar P;

@members {
Stack paraphrases = new Stack();
public String getErrorMessage(RecognitionException e,
                              String[] tokenNames)
{
    String msg = super.getErrorMessage(e, tokenNames);
    if ( paraphrases.size()>0 ) {
        String paraphrase = (String)paraphrases.peek();
        msg = msg+" "+paraphrase;
    }
    return msg;
}
}
```

Then, at the start of each rule that relates to an abstract concept in your language, push and pop a paraphrased grammar location that will be used in the error reporting:

Download errors/P.g

```
prog:   stat+ ;

stat
@init  { paraphrases.push("in statement"); }
@after { paraphrases.pop(); }
    :   expr ';'
        {System.out.println("found expr: "+$stat.text);}
    |   ID '=' expr ';'
        {System.out.println("found assign: "+$stat.text);}
    ;

expr
@init  { paraphrases.push("in expression"); }
@after { paraphrases.pop(); }
    :   multExpr (('+'|'-') multExpr)*
    ;

multExpr
    :   atom ('*' atom)*
    ;

atom:   INT
    |   '(' expr ')'
    ;
```

Here are three sample runs to illustrate the improved error messages for the same invalid input used in previous sections:

```
⇐  $ java TestP
⇐  (3;
⇐  EOF
⇒  line 1:2 mismatched input ';' expecting ')' in expression
   found expr: (3;
⇐  $ java TestP
⇐  1+;
⇐  EOF
⇒  line 1:2 no viable alternative at input ';' in expression
   found expr: 1+;
⇐  $ java TestP
⇐  a;
⇐  EOF
⇒  line 1:1 mismatched input ';' expecting '=' in statement
   $
```

Recall that the parser detects the first error in **atom** but emits "in expression" rather than the less useful "in atom" (the user does not know what an atom is). All rules invoked from **expr** will use that paraphrase. The last run did not execute the embedded action because the parser could not recover using single symbol deletion or insertion and it therefore had to bail out of rule **stat**.

What if you want the parser to bail out upon the first syntax error without ever trying to recover? The next section describes how to make the parser exit immediately upon the first syntax error.

10.4 Exiting the Recognizer upon First Error

Sometimes for invalid input you simply want to report an error and exit without trying to recover and continue parsing. Examples include unrecognized network protocol messages and expressions in spreadsheets.

Most likely, your embedded actions will not be valid given the erroneous input, so you might as well just report an error and ask the user to fix the problem.

To make your recognizer (parser or tree parser) exit immediately upon recognition error, you must override two methods, mismatch() and recoverFromMismatchSet(), and alter how the parser responds to thrown exceptions.

Download errors/Bail.g

```
grammar Bail;

@members {
protected void mismatch(IntStream input, int ttype, BitSet follow)
    throws RecognitionException
{
    throw new MismatchedTokenException(ttype, input);
}
public void recoverFromMismatchedSet(IntStream input,
                                     RecognitionException e,
                                     BitSet follow)
    throws RecognitionException
{
    throw e;
}
}

// Alter code generation so catch-clauses get replace with
// this action.
@rulecatch {
catch (RecognitionException e) {
    throw e;
}
}
```

As usual, the remainder of the grammar is the same. Here is a sample run using the typical test rig:

```
⇐ $ java TestBail
⇐ (3;
⇐ 1+2;
⇐ EOF
⇒ Exception in thread "main" MismatchedTokenException(7!=13)
        at BailParser.mismatch(BailParser.java:29)
        at org.antlr.runtime.BaseRecognizer.match(BaseRecognizer.java:92)
        at BailParser.atom(BailParser.java:307)
        at BailParser.multExpr(BailParser.java:226)
        at BailParser.expr(BailParser.java:163)
        at BailParser.stat(BailParser.java:117)
        at BailParser.prog(BailParser.java:57)
        at TestBail.main(TestBail.java:9)
```

The parser throws an exception rather than recovering, and Java displays the uncaught exception emanating from the main program.

The **rulecatch** action changes the default code ANTLR generates at the end of each rule's method. In this case, it will change this code:

```
catch (RecognitionException re) {
    reportError(re);
    recover(input,re);
}
```

to the code in the **rulecatch** action:

```
catch (RecognitionException re) {
    throw e;
}
```

Note that because ANTLR traps exceptions only under RecognitionException, your parser will exit if you throw an exception that is not under this hierarchy such as a RuntimeException.

You have control over whether a recognizer throws an exception and what to do if it does. This even allows you to do context-sensitive error recovery by putting a conditional in the **rulecatch** action. In this way, some rules in your grammar could do automatic recovery where it makes sense, leaving the other rules to bail out of the parser.

The **rulecatch** alters the default code generation for all rules, but sometimes you want to alter exception handling for just one rule; the following section describes how to use individual exception handlers.

10.5 Manually Specifying Exception Handlers

You can trap any exception you want in any rule of your grammar by specifying manual exception handlers after the semicolon at the end of the rule. You can trap any Java exception. The basic idea is to force all errors to throw an exception and then catch the exception where you think it is appropriate. Given the same **rulecatch** action and method overrides from Bail.g in the previous section, here is how to catch all recognition exceptions in rule **stat**:

Download errors/Exc.g

```
stat:   expr ';'
        {System.out.println("found expr: "+$stat.text);}
    |   ID '=' expr ';'
        {System.out.println("found assign: "+$stat.text);}
    ;
    catch [RecognitionException re] {
        reportError(re);
        consumeUntil(input, SEMI); // throw away all until ';'
        input.consume(); // eat the ';'
```

The exception action reports the errors as usual but manually recovers by just killing everything until the semicolon. Also, the grammar needs a **tokens** specification so that the exception action can refer to the **SEMI** token type:

```
tokens { SEMI=';' }
```

Here are some sample runs:

```
⇐  $ java TestExc
⇐  (3;
⇐  a=1;
⇐  EOF
⇒  line 1:2 mismatched input ';' expecting ')'
   found assign: a=1;
⇐  $ java TestExc
⇐  3+;
⇐  a=1;
⇐  EOF
⇒  line 1:2 no viable alternative at input ';'
   found assign: a=1;
   $
```

This strategy does not do as well as ANTLR's default because it gobbles input more aggressively. In the following run, the mismatched token for the expression 3 forces the exception handler to consume until it sees a semicolon, effectively tossing out the second line of input:

```
⇐  $ java TestExc
⇐  3
⇐  a=1;
⇐  x=2;
⇐  EOF
⇒  line 2:0 mismatched input 'a' expecting SEMI
   found assign: x=2;
   $
```

The default mechanism would recover properly and match the first assignment correctly, but this example illustrates how to specify your own exceptions.

10.6 Errors in Lexers and Tree Parsers

So far this chapter has focused primarily on parsers, but errors can occur in lexers and tree parsers as well, of course. For example, returning to the original **E** grammar from the beginning of this chapter, entering an invalid character such as & results in a no viable alternative exception:

```
⇐  $ java TestE
⇐  &3;
⇐  EOF
⇒  line 1:0 no viable alternative at character '&'
   found expr: 3;
   $
```

The only difference is that the error message says "character" instead of "input." In between tokens, lexers recover by blindly killing the offending character and looking for another token. Once the lexer has recovered, it is free to send the next token to the parser. You can see that the parser finds a valid expression after the offending character and prints the expression, 3;, as usual.

Within a token definition rule, however, mismatched character errors can also occur, which are analogous to mismatched token errors. For example, if you forget to terminate a string, the lexer might consume until the end of file and would emit something like the following error:

```
line 129:0 mismatched character '<EOF>' expecting '"'
```

Turning to tree parsers now, recall that they are really just parsers that read streams of nodes instead of streams of tokens. Recall the expression evaluator from Chapter 3, *A Quick Tour for the Impatient*, on page 59. Imagine that, while developing a grammar, you forget to "delete" parentheses in rule **atom** (that is, you forgot to add the ! operators):

Download errors/Expr.g

```
atom:   INT
    |   ID
    |   '(' expr ')' // should have ! ops or use "-> expr"
    ;
```

Now if you rebuild and run with sample input a=(3), the parser will no longer build a proper tree. It will leave in the parenthesis tokens, which will cause syntax errors in the tree parser. The following session illustrates the error message emitted by the tree parser:

```
⇐  $ java org.antlr.Tool Expr.g Eval.g
⇐  $ javac TestEval.java # compiles all files
⇐  $ java TestEval
⇐  a=(3)
⇐  EOF
⇒  (= a ( 3 ))  // note the extra ( and ) around '3'
   Eval.g: node from line 1:2 no viable alternative at input '('
   3
   $
```

Tree parsers have slightly different error headers (the "*tree-grammar-name*: node from" prefix) to indicate that a tree parser emitted the error rather than a normal parser. Tree parsers emit error messages with the grammar name because the message is always intended for the programmer—a malformed tree is the programmer's fault, not the user's.

Naturally for development, you can augment these messages to indicate where in the grammar the tree parser had a problem just like with parser grammars. You can even print the offending subtree because the RecognitionException object contains the appropriate node pointer.

10.7 Automatic Error Recovery Strategy

ANTLR's error recovery mechanism is based upon Niklaus Wirth's early ideas in *Algorithms + Data Structures = Programs* [Wir78] (as well as Rodney Topor's *A Note on Error Recovery in Recursive Descent Parsers* [Top82]) but also includes Josef Grosch's good ideas from his CoCo parser generator (*Efficient and Comfortable Error Recovery in Recursive Descent Parsers* [Gro90]). Essentially, recognizers perform single-symbol insertion and deletion upon mismatched symbol errors (as described in a moment) if possible. If not, recognizers gobble up symbols until the lookahead is a member of the *resynchronization set* and then exit the rule. The resynchronization set is the set of input symbols that can legally follow references to the current rule and references to any invoking rules up the call chain. Similarly, if the recognizer cannot choose any of the alternatives from the start of a rule, the recognizer again uses the gobble-and-exit strategy.

This "consume until symbol in resynchronization set" strategy makes sense because the recognizer knows there is something wrong with the input that should match for the current rule. It decides to throw out tokens until the lookahead is consistent with something that should match after the recognizer exits from the rule. For example, if there is a syntax error within an assignment statement, it makes a great deal of sense to throw out tokens until the parser sees a semicolon or other statement terminator.

Another idea from Grosch that ANTLR implements is to emit only a single message per syntax error to prevent spurious, cascading errors. Through the use of a simple boolean variable, set upon syntax error, the recognizer can avoid emitting further errors until the recognizer

Language Theory Humor

Apparently, the great Niklaus Wirth* had an excellent sense of humor. He used to joke that in Europe people called him by "reference" (properly pronouncing his name "Ni-klaus Virt") and that in America people called him by "value" (pronouncing his name "Nickle-less Worth").

At the Compiler Construction 1994 conference, Kristen Nygaard† (inventor of Simula) told a story about how, while teaching a language theory course, he commented that "Strong typing is fascism," referring to his preference for languages that are loose with types. A student came up to him afterward and asked why typing hard on the keyboard was fascism.

*. See http://en.wikipedia.org/wiki/Niklaus_Wirth.
†. See http://en.wikipedia.org/wiki/Kristen_Nygaard.

successfully matches a symbol and resets the variable. See field errorRecovery in BaseRecognizer. The following three sections describe ANTLR's automatic error recovery system in more detail.

Recovery by Single Symbol Deletion

Consider erroneous expression (3));. The parser responds with this:

```
⇐  $ java TestE
⇐  (3));
⇐  EOF
⇒  line 1:3 mismatched input ')' expecting ';'
   found expr: (3));
   $
```

Even though there is an extra right parenthesis, the parser is able to continue, implicitly deleting the extra symbol. Instead of giving up on the rule, the parser can examine the next symbol of lookahead to see whether that symbol is the one it wanted to find. If the next symbol is exactly what it was looking for, the parser assumes the current symbol is a junk token, executes a consume() call, and continues. If the parser fails to resynchronize by deleting a symbol, it attempts to insert a symbol instead.

Recovery by Single Symbol Insertion

Consider the erroneous input (3; discussed at the start of this chapter. The parser continued parsing and executed the embedded action even though it complained that it was expecting ')' but it found ';':

```
⇐  $ java TestE
⇐  (3;
⇐  EOF
⇒  line 1:2 mismatched input ';' expecting ')'
   found expr: (3;
   $
```

The parser first tries single-token deletion but finds that the next symbol of lookahead, EOF, is not what it was looking for—deleting it and continuing would be wrong. Instead, it tries the opposite: is the current token consistent with what could come after the expected token? In this case, the parser expects to see a semicolon next, which is in fact the current token. In this case, the parser can assume that the user simply forgot the expected token and can continue, implicitly inserting the missing token.

If single-symbol deletion and insertion both fail, the parser has no choice but to attempt to resynchronize using the aggressive strategy described in the next section.

Recovery by Scanning for Following Symbols

When within-rule error recovery fails or upon a no viable alternative situation, the best recovery strategy is to consume tokens until the parser sees a token in the set of tokens that can legally follow the current rule. These following symbols are those that appear after a reference to the current rule or any rule invocation that led to the current rule. That set of tokens is called the *resynchronization set*.

It is worthwhile going through an example to understand how ANTLR recovers via resynchronization sets. Consider the following grammar and imagine that, at each rule invocation, the parser pushes the set of tokens that could follow that rule invocation onto a stack:

```
s : '[' b ']'
  | '(' b ')'
  ;
b : c '^' INT ;
c : ID
  | INT
  ;
```

Here are the following sets: ']' follows the first reference to **b** in **s**, ')' follows the reference to **b** in the second alternative, and '^' follows the reference to **c** in **b**.

With the erroneous input [], the call chain is as follows: **s** calls **b** calls **c**. In **c**, the parser discovers that the lookahead, ']', is not consistent with either alternative of **c**. The following table summarizes the call stack and associated resynchronization set context for each rule invocation.

Call Depth	Resynchronization Set	In Rule
0	EOF	main()
1	EOF	**s**
2	']'	**b**
3	'^'	**c**

The complete resynchronization set for the current rule is the union of all resynchronization sets walking up the call chain. Rule **c**'s resynchronization set is therefore { '^', ']', EOF }. To resynchronize, the parser consumes tokens until it finds that the lookahead is consistent with the resynchronization set. In this case, lookahead ']' starts out as a member of the resynchronization set, and the parser won't actually consume any symbols.

After resynchronization, the parser exits rule **c** and returns to rule **b** but immediately discovers that it does not have the '^' symbol. The process repeats itself, and the parser consumes tokens until it finds something in the resynchronization set for rule **b**, which is { ']', EOF }. Again, the parser does not consume anything and exits **b**, returning to the first alternative of rule **s**. Now, the parser finds exactly what it is looking for, successfully matching the ']'. The parser is now properly resynchronized.

Those familiar with language theory will wonder whether the resynchronization set for rule **c** is just *FOLLOW*(c), the set of all viable symbols that can follow references to **c** in some context. It is not that simple, unfortunately, and the resynchronization sets must be computed dynamically to get the set of symbols that can follow the rule in a particular context rather than in all contexts.[1] *FOLLOW*(b) is { ')', ']' }, which includes all symbols that can follow references to **b** in both contexts

1. In the generated source code, the partial resynchronization sets look like FOLLOW_b_in_s7. Every rule invocation is surrounded by something like pushFollow(FOLLOW_expr_in_expr334); expr(); _fsp--;. This code pushes the exact set of symbols that can follow the rule reference.

(alternatives one and two of **s**). Clearly, though at runtime, the parser can call **b** from only one location at a time. Note that *FOLLOW*(c) is '^', and if the parser resynchronized to that token instead of the resynchronization set, it would consume until the end of file because there is no '^' on the input stream.

ANTLR provides good error reporting for both the developer and the users of the deployed language application. Further, ANTLR's automatic error recovery mechanism resynchronizes recognizers very well, again yielding an excellent experience for your users.

This chapter concludes Part II of the book, which focused on explaining the various syntax and semantics of ANTLR grammars. Part III of this book explains and illustrates ANTLR's sophisticated new *LL(*)* parsing strategy and the use of semantic and syntactic predicates.

Part III

Understanding Predicated-*LL(*)* Grammars

Chapter 11

LL(*) Parsing

The chapters in the third part of this book represent a thorough study of ANTLR's predicated-*LL(*)* parsing mechanism. Unfortunately, this topic is not easy and typically requires an advanced parsing background to fully understand. The discussion is as clear and easy to follow as possible, but you'll still have to read some sections multiple times, which is typical of any dense technical material. This reference book includes such an advanced discussion because it is simply not available anywhere else, and ultimately, you'll want a deep understanding of exactly how ANTLR recognizers work. You'll be able to resolve grammar analysis warnings more easily, build simpler grammars, and tackle more challenging language problems. You don't need these chapters to get started using ANTLR, but you should read them when you run into grammar analysis errors or have trouble designing a grammar for a tricky language construct.

This chapter defines ANTLR's *LL(*)* parsing, describes how it works, and characterizes the kinds of grammars that are *LL(*)* conformant. More important, this chapter emphasizes grammars that are not *LL(*)* and tells you how to deal with them. When ANTLR accepts your grammar, you will not notice ANTLR's underlying parsing strategy—it is only when you get a warning that you need to know more about how *LL(*)* works. Although *LL(*)* increases the fundamental power of ANTLR's parsing strategy beyond *LL(k)*'s fixed lookahead parsing, sometimes you will need more power than even *LL(*)* can provide. The three chapters on predicated-*LL(*)* parsing following this chapter discuss how to handle non-*LL(*)* grammars for context-sensitive, ambiguous, and other problematic languages.

Let's begin the discussion of *LL(*)* by covering the difference between a grammar and the program that recognizes sentences in the language described by that grammar.

11.1 The Relationship between Grammars and Recognizers

Building a grammar means creating a specification that not only conforms to a particular parser generator's grammar metalanguage (per Chapter 4, *ANTLR Grammars*, on page 86) but that also conforms to its underlying parsing strategy. The stronger the parsing strategy, the more grammars that the parser generator will accept, thus making it easier to describe your language with a natural, easy-to-read grammar.

Ideally a parser generator would accept any grammar, but there are two reasons why such parser generators are not commonly used. First, parsing strategies that accept any grammar are usually less efficient and more difficult to understand.[1] Second, some syntactically valid grammars are ambiguous in that the grammar can match the same input following multiple paths through the grammar, which makes it difficult for actions to interpret or translate the input. Should the parser execute actions found along all paths or just one? If along just one path, which one? Computers only deal well with deterministic languages: languages that have exactly one meaning for each statement. It has been the focus of my research for more than fifteen years to make parsing as powerful as possible without allowing ambiguous grammars or sacrificing accessibility, simplicity, and efficiency—ANTLR is constrained by what most programmers can and will use.

ANTLR v3 introduces a new parsing strategy called *LL(*) parsing*[2] that is much more powerful than traditional *LL(k)*-based parsers, which are limited to a finite amount of lookahead, *k*. *LL(*)*, in contrast, allows the lookahead to roam arbitrarily far ahead, relieving you of the responsibility of specifying *k*. *LL(*)* does not alter the recursive-descent parsing strategy itself at all—it just enhances an *LL* decision's predictive capabilities, which we'll explore in a moment. You will find that building grammars for ANTLR v3 is much easier than for ANTLR v2 or any other

1. Generalized *LR* (*GLR*) parsing [Tom87] is the latest parsing technology that handles any context-free grammar.
2. The *LL(*)* term was coined by Sriram Srinivasan, a friend who helped me think through this new parsing strategy. See http://www.antlr.org/blog/antlr3/lookahead.tml for more information about the *LL(*)* algorithm.

LL-based parser generator. For example, *LL(*)* lets you build grammars the way you want and then automatically does left-factoring to generate efficient decisions. *LL(*)* is also much more flexible in terms of attributes and actions than bottom-up *LR*-based tools, such as YACC and its derivatives,[3] yet has as much or more parsing power.

Another great aspect of ANTLR is that it unifies the notions of lexing, parsing, and tree parsing. It doesn't matter whether you are parsing a stream of characters, tokens, or tree nodes: ANTLR uses the same recognition strategy. The generated recognizers even derive from the same base class, BaseRecognizer. This implies that lexers have the power of context-free grammars rather than simple regular expressions[4]—you can match recursive structures such as nested comments inside the lexer. Discussions of *LL(*)* apply equally well to any ANTLR grammar. Before detailing how *LL(*)* works, let's zero in on the weaknesses of *LL(k)* that provided the impetus for the development of *LL(*)*.

11.2 Why You Need *LL(*)*

Natural grammars are sometimes not *LL(k)*. For example, the following easy-to-read grammar specifies the syntax of both abstract and concrete methods:

```
method
    : type ID '(' args ')' ';'          // E.g., "int f(int x,int y);"
    | type ID '(' args ')' '{' body '}' // E.g., "int f() {...}"
    ;
type: 'void' | 'int' ;
args: arg (',' arg)* ; // E.g., "int x, int y, int z, ..."
arg : 'int' ID ;
body: ... ;
```

The grammar is valid in the general sense because the rules follow the syntax of the ANTLR metalanguage and because the grammar is unambiguous (the grammar cannot match the same sentence in more than

3. *LR*-based parsers can use only synthesized attributes, analogous to return values, whereas *LL*-based parsers can pass inherited attributes (parameters) to rules and use synthesized attributes. Further, introducing an action into an *LR* grammar can cause a grammar nondeterminism, which cannot happen in an *LL* grammar.

4. Regular expressions are essentially grammars that cannot invoke other rules. These expressions are said to match the "regular" languages, which are a subset of the context-free languages matched by context-free grammars. See Section 2.2, *The Requirements for Generating Complex Language*, on page 38 and Section 4.1, *Describing Languages with Formal Grammars*, on page 87 for more about context-free grammars.

one way). According to the requirements of *LL(k)* parsing technology, however, the grammar is not *LL(k)* for any *fixed* value of *k*. From the left edge of **method**'s alternatives, the amount of lookahead necessary to see the distinguishing input symbol, ';' or '{', is unbounded because the incoming method definition can have an arbitrary number of arguments. At runtime, though, "arbitrary" does not imply "infinite," and the required lookahead is usually from five to ten symbols for this decision. You will see in a moment that *LL(*)* takes advantage of this practical limit to generate efficient parsing decisions that, in theory, could require infinite lookahead.

The traditional way to resolve this *LL(k)* conflict is to left-factor offending rule **method** into an equivalent *LL(k)*-conformant rule. *Left-factoring* means to combine two or more alternatives into a single alternative by merging their common left prefix:

```
method
    :   type ID '(' args ')' (';' | '{' body '}')
    ;
```

Unfortunately, this version is less readable. Worse, in the presence of embedded actions, it can be difficult to merge alternatives. You have to delay actions until after the recognizer sees ';' or '{'.

Consider another natural grammar that matches class and interface definitions:

```
def : modifier* classDef     // E.g., public class T {...}
    | modifier* interfaceDef // E.g., interface U {...}
    ;
```

Again, you could refactor the rule, but it is not always possible and leads to unnatural grammars:

```
def : modifiers* (classDef|interfaceDef) ;
```

When building grammars for really difficult languages such as C++, engineers often leave the grammar in a natural condition and then add semantic predicates (see Chapter 13, *Semantic Predicates*, on page 317) to manually scan ahead looking for the distinguishing symbol:

```
def : {findAhead(CLASS_TOKEN)}?     modifier* classDef
    | {findAhead(INTERFACE_TOKEN)}? modifier* interfaceDef
    ;
```

where findAhead() is a loop that scans ahead in the input stream looking for a particular token. This solution works but requires manual coding and is sometimes difficult to get right.

In this case, for example, the findAhead() method must stop when it sees a '{', lest it look past the current type declaration to the next declaration beyond. The next two sections describe how ANTLR automatically generates a similar lookahead mechanism that is correct and efficient.

11.3 Toward *LL(*)* from *LL(k)*

Building a parser generator is easy except for the static grammar analysis that computes the lookahead sets needed to make parsing decisions. For BNF grammars, grammars with just rule and token references, the code generation templates are straightforward. References to rule *r* become method calls, *r*();. References to token *T* become match(*T*);. match() checks that the current input symbol is *T* and moves to the next symbol. Rule definitions themselves are a little more complicated; an arbitrary rule *r* definition with multiple alternatives translates to this:

```
void r() {
  if ( «lookahead-consistent-with-alt1» ) { «code-for-alt-1»; }
  else if ( «lookahead-consistent-with-alt2» ) { «code-for-alt-2»; }
  ...
  else error;
}
```

The series of **if** expressions represents the parsing decision for **r**. Therefore, the power of these expressions dictates the strength of your parser. When building such recursive-descent parsers (see Section 2.7, *Recognizing Computer Language Sentences*, on page 48) by hand, you are free to use any expression you want, but a parser generator must divine these expressions from the grammar. The smarter the grammar analysis algorithm, the more powerful the expressions it can generate.

Until the ANTLR v1.0 release fifteen years ago, all practical parser generators were limited to one symbol of lookahead (k=1). Top-down parser generators were therefore limited to *LL(1)*, which is pretty weak. For example, the following simple grammar is not *LL(1)* because rule **stat**'s parsing decision cannot distinguish between its two alternatives looking only at the first symbol, **ID**:

```
stat : ID '=' expr
     | ID ':' stat
     ;
```

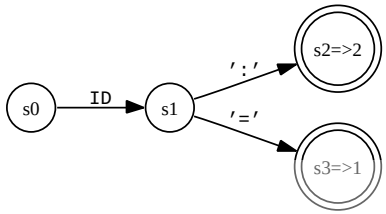
Rule **stat** is, however, *LL(2)*. By looking two symbols ahead, the parsing decision can see either the '=' or the ':'. ANTLR v1.0 could generate Java code like the following to implement rule **stat** using k=2:

```
void stat() {
  if ( LA(1)==ID&&LA(2)==EQUALS ) { // PREDICT
    match(ID);                      // MATCH
    match(EQUALS);
    expr();
  }
  else if ( LA(1)==ID&&LA(2)==COLON ) { // PREDICT
    match(ID);                          // MATCh
    match(COLON);
    stat();
  }
  else «error»;
}
```

Method LA() evaluates to the **int** token type of the token at the specified lookahead depth. As is often the case, it is not the sequence of lookahead symbols that distinguishes alternatives—it is a token (or tokens) at a particular lookahead depth that matters. Here, the first token **ID** does not help distinguish between the alternatives because it is common to both. Consider the following alternative implementation that focuses on the second symbol of lookahead:

```
void stat() {
  // PREDICTION CODE; yield an alternative number
  int alt=0;
  if ( LA(1)==ID ) {
    if ( LA(2)==EQUALS ) alt=1;      // predict alternative 1
    else if ( LA(2)==COLON ) alt=2;  // predict alternative 2
  }
  // MATCH PREDICTED ALTERNATIVE
  switch (alt) {
    case 1 : // match alternative 1
      match(ID);
      match(EQUALS);
      expr();
      break;
    case 2 : // match alternative 2
      match(ID);
      match(COLON);
      stat();
      break;
    default : «error»;
  }
}
```

This implementation style factors out the parsing decision to the front of the rule, which now yields an alternative number. The rest of the rule is just a **switch** on the predicted alternative number. In this style, the true form of the parsing decision becomes clear, as illustrated in the following DFA:



This DFA encodes that lookahead sequence ID '=' predicts alternative one by traversing states s0, s1, s3. s3 is an accept state that indicates the predicted alternative. The DFA encodes one other lookahead sequence, ID ':', that predicts alternative two by traversing states s0, s1, s2.

Lookahead decisions that use fixed lookahead, such as *LL(k)* decisions, always have *acyclic DFA*.[5] The next section describes the *cyclic DFA* *LL(*)* uses to support arbitrary lookahead.[6]

11.4 *LL(*)* and Automatic Arbitrary Regular Lookahead

LL(*) extends *LL(k)* by allowing cyclic DFA, DFA with loops, that can scan arbitrarily far ahead looking for input sequences that distinguish alternatives. Using the maze analogy, *LL(*)*'s arbitrary lookahead is like bringing a trained monkey along that can race ahead at each maze fork. If two paths emanating from a fork have the same initial words, you can send the monkey down both paths looking for a few of the future words in your passphrase. One of the most obvious benefits of *LL(*)* is that you do not have to specify the lookahead depth as you do with *LL(k)*—ANTLR simply figures out the minimum lookahead necessary to distinguish between alternatives. In the maze, *LL(k)* decision makers do not have trained monkeys and have limited information. They can see only the next *k* words along the paths emanating from a fork.

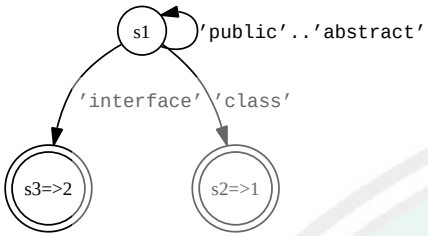
5. An acyclic DFA is one that matches a finite set of input sequences because there are no loops to match repeated, arbitrarily long sequences. While traversing an acyclic DFA, you can never revisit a state.
6. Cyclic DFA allows states to transition to previously visited states.

Reconsider the non-*LL(k)* **class** or **interface** definition grammar shown in the previous section:

```
def : modifier* classDef
    | modifier* interfaceDef
    ;
```

A cyclic DFA can easily skip ahead past the modifiers to the **class** or **interface** keyword beyond, as illustrated by the following DFA:

Using ANTLRWorks, you can look at the DFA created for a decision by right-clicking a rule or subrule and selecting Show Decision DFA.



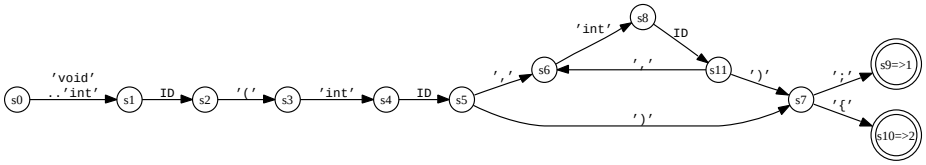
In this case, a simple **while** loop implements rule **def**'s prediction DFA:

```
void def() {
  int alt=0;
  while (LA(1) in modifier) consume(); // scan past modifiers
  if ( LA(1)==CLASS ) alt=1;           // 'class'?
  else if ( LA(1)==INTERFACE ) alt=2;  // 'interface'?
  switch (alt) {
    case 1 : ...
    case 2 : ...
    default : error;
  }
}
```

Just as with the earlier *LL(2)* solution, the initial **modifier** symbols do not help distinguish between the alternatives. The loop (trained monkey) simply scans past those tokens to the important token that follows. Notice that the decision DFA looks like a left-factored version of rule **def**.

It is important to point out that ANTLR is not approximating the entire grammar with a DFA. DFAs, which are equivalent to regular expressions, are not as powerful as context-free grammars (see Section 2.2, *The Requirements for Generating Complex Language*, on page 38). ANTLR is using the DFAs only to distinguish between alternatives. In the earlier example, ANTLR creates a DFA that stops matching at the **class** or **interface** keyword. ANTLR does not build a DFA that matches the entire **classDef** rule or **interfaceDef** rule. Once **def** predicts which alternative will succeed, it can begin parsing like any normal *LL* parser. As another example, here is the DFA for the grammar in Section 11.2,

Why You Need LL(*), on page 264 that predicts abstract vs. concrete methods:



This DFA matches the start of a method definition, splits for the optional arguments, and then pinches back to state s7. s7 then splits upon either the ';' or the '{' to distinguish between abstract and concrete methods. State s9 predicts alternative one ("s9=>1"), and state s10 predicts alternative two. Input void foo(int i); predicts alternative one by following this path: s0, s1, s2, s3, s4, s5, s7, s9. Input void foo(int i) {... predicts alternative two by following this path: s0, s1, s2, s3, s4, s5, s7, s10. Because the starting portion of these two inputs is identical, the state sequence is identical until the DFA reaches the critical s7 state.

Sometimes these DFAs become complicated, but ultimately they simply yield a predicted alternative number as in the previous example. DFAs scan past common left prefixes looking for distinguishing symbols. The following rule for a Java-like language has a subrule that matches the variables and method definitions within an **interface**:

```
interfaceDef
    :   'interface' ID ('extends' classname)?
        '{'
        (   variableDefinition
        |   methodDefinition
        )*
        '}'
    ;
```

The DFA for the embedded subrule, shown in Figure 11.1, on page 272, has some interesting characteristics. The details are not that important—the DFA merely illustrates that ANTLR sometimes needs to build a large DFA while looking for the few symbols that will differentiate alternatives. The accept states s18 and s19 are the most interesting states because the cloud of other states pinches back together into s16 and then splits on the single symbol that distinguishes between variable and method definitions. The complicated cyclic states before that just scan past tokens, as defined in the grammar, until the semicolon or left curly.

Also note that upon seeing the right curly the DFA immediately predicts the third alternative via states s0 and s1, which is the implied exit branch of the (. . .)* loop. State s1 is an accept state that predicts alternative 3. This DFA illustrates a case where a recognizer uses various lookahead depths even within the same decision. The DFA uses more-or-less lookahead for optimal efficiency, depending on what it finds on the input stream.

A decision is *LL(*)* if a DFA exists that recognizes the decision's exact lookahead language and has the following:

LL(*) degenerates to LL(k) for a fixed k if your grammar is LL(k). If it is not LL(k), ANTLR searches further ahead in a grammar to find tokens that will help it make choices.

- No unreachable states
- No dangling states, that is, states that cannot reach an accept state
- At least one accept state for each alternative

Each alternative has a lookahead language, and if the lookahead languages are disjoint for a decision, then the decision is *LL(*)*. It is like building a regular expression to describe what input predicts each alternative and then verifying that there is no overlap between the input matched by the regular expressions.

At this point, the reader might ask, "Isn't this just backtracking?" No. An *LL(5)* parser, for example, uses a maximum of five lookahead symbols and is considered to have linear complexity, albeit with a bigger constant in front than an *LL(1)* parser. Similarly, if an *LL(*)* parser can guarantee in practice that it will never look ahead more than five symbols, is it not effectively the same as *LL(5)*? Further, *LL(*)* is scanning ahead with a DFA, not backtracking with the full parser. It is the difference between having a trained monkey in the maze racing ahead looking for a few symbols and you having to laboriously backtrack through each path emanating from a fork. The lookahead DFAs are smaller and faster because they are not descending into deep rule invocation chains. Each transition of the DFA matches a symbol, whereas a backtracking parser might make twenty method calls without matching a single symbol. The DFAs are efficiently coded and automatically throttle down when less lookahead is needed. ANTLR is also not sucking actions into lookahead DFAs. DFAs automatically avoid action execution during *LL(*)* prediction. Backtracking parsers, on the other hand, must turn off actions or figure out how to unroll them.

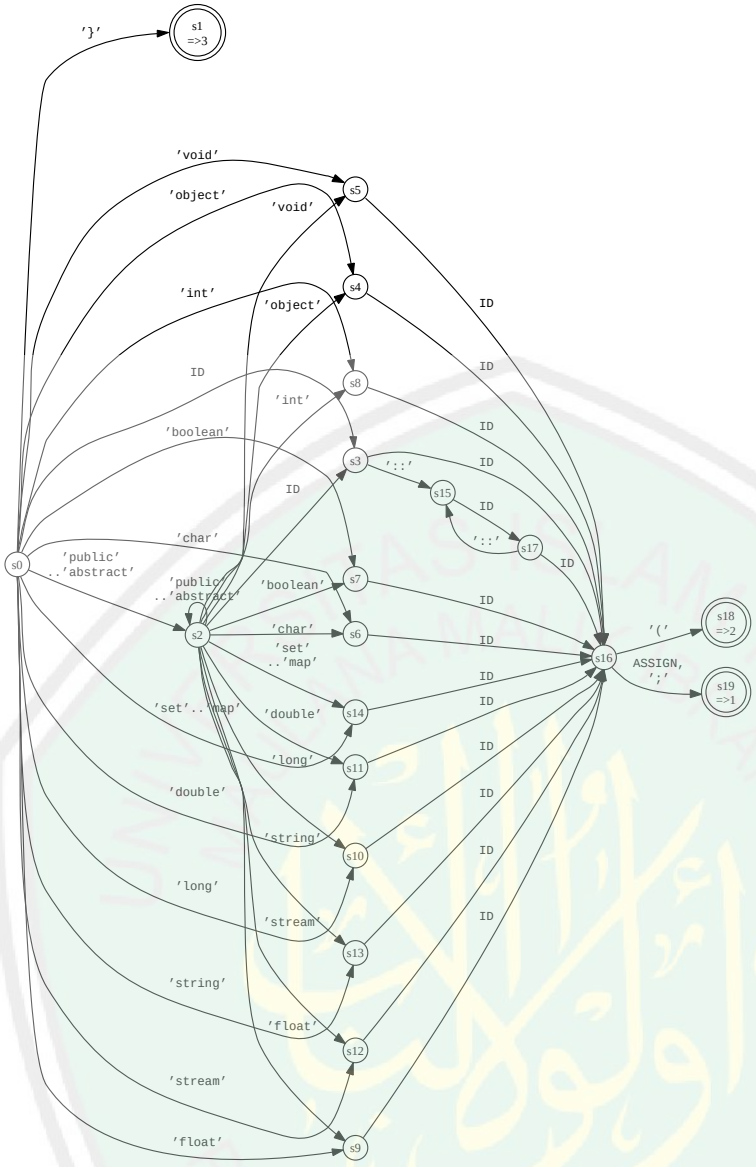


Figure 11.1: Variable vs. method prediction DFA that illustrates a complicated cloud of cyclic states that pinches back to a single state, s16, to distinguish between variable and method definitions

Now that you know a little bit about *LL(*)* parsing decisions, it is time to focus on non-*LL(*)* decisions because, in practice, that is when you care about the underlying parsing strategy. It is only when ANTLR fails to generate a deterministic *LL(*)* decision that you need to figure out why ANTLR could not handle the decision. The next section explains what can go wrong in detail.

11.5 Ambiguities and Nondeterminisms

When analyzing a grammar warning or error from ANTLR, your task is to impersonate ANTLR's grammar analysis algorithm and try to figure out why the decision has a problem. There are two general categories of issues. In the first category, you have specified a decision that either is fundamentally incompatible with top-down recursive-descent parsing or is non-*LL(*)* because the lookahead language is not regular (cannot be described with a regular expression). This category always involves recursive rules within the grammar. In the second category, ANTLR has no problem building a DFA, but at least one input sequence can be matched by more than one alternative within a decision. This category therefore deals with recognizer nondeterminism, an inability to decide which path to take. The following sections describe the issues in each category in detail.

LL-Incompatible Decisions

Some grammars just simply do not make sense regardless of the parsing tool. For example, here is a rule that can match only an infinite sequence, which is obviously incompatible with a finite computer:

```
a : A a ;
```

The sequence of rule invocations, the *derivation*, for this rule looks like the following:

```
a => A a
a => A A a
a => A A A a
...
```

where each occurrence of **a** on the right side is replaced with A a per the definition of **a**. Rule **a** matches a token **A** followed by another reference to **a**, which in turn can be a token **A** followed by another reference to **a**, *ad nauseam*.

This grammar is equivalent to the following Java code:

```
void a() {
    match(A);
    a();
}
```

Clearly, this will never terminate, and you will get a stack overflow exception. As a general rule, grammar rules with recursive alternatives must also include an alternative that is not recursive even if that alternative is an empty alternative. The following rewrite of the grammar is probably more what is intended anyway:

```
a : A a
  |
  ;
```

This grammar matches A*, and naturally, you should simply use this EBNF construct in your grammar rather than tail recursion:[7]

```
a : A* ;
```

The two grammars are equivalent, but the tail-recursive version is less efficient and less clear. A* clearly indicates repetition, whereas the programmer must imagine the tail recursion's emergent behavior to figure out what the grammar developer intends.

Left-Recursive Grammars

What if the recursion is on the left edge of an alternative or reachable from the left edge without consuming an input symbol? Such a rule is said to be *left recursive*. Left recursion is a perfectly acceptable grammar for some parser generators, but not for ANTLR. Consider the reverse of the previous tail recursive grammar that matches the same language:

```
a : a A
  |
  ;
```

The derivation for AAA looks like this where the final reference to a on the right side is replaced with the empty alternative:

```
a => a A
a => a A A
a => a A A A
a => A A A
```

7. A rule that uses tail recursion calls itself, or another rule that ultimately calls that rule, at the end of an alternative. A rule that invokes itself loops via recursion.

Unfortunately, although a valid grammar construct in general, an *LL*-based top-down parser cannot deal with left-recursion. ANTLR reports the following:

```
error(210): The following sets of rules are mutually left-recursive [a]
```

Be aware that left-recursion might not be direct and might involve a chain of multiple rules:

```
a : b A
  |
  ;
b : c ;
c : a ;
```

ANTLR reports the following:

```
error(210): The following sets of rules are mutually
            left-recursive [a, c, b]
```

Arithmetic Expression Grammars

In my view, left-recursion is pretty unnatural (ENBF looping constructs are easier to understand) except in one case: specifying the structure of arithmetic expressions. In this case, bottom-up parser generators allow a much more natural grammar than top-down parser generators, and you will encounter this if you are trying to convert a grammar from, say, YACC to ANTLR. The grammar usually looks something like this:

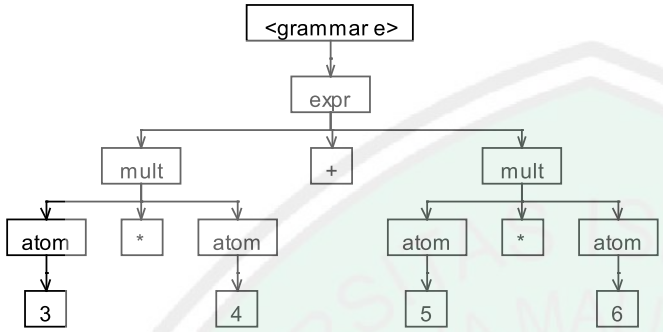
```
// non-LL yacc grammar
%left  '+'
%left  '*' // higher precedence than '+'
expr:   expr '+' expr
    |   expr '*' expr
   ...
    |   ID
    |   INT
    ;
```

This might seem natural if you ignore that the grammar is left-recursive. Without the precedence %left specifier, this grammar would be ambiguous, and bottom-up parser generators require that you specify the priority of those operators (as described in every compiler book ever written). In a top-down parser generator, you must explicitly encode the priorities of the operators by using different rules. The following *LL*-compatible grammar matches the same language, but with explicit priorities:

```
expr:   mult ('+' mult)* ;
mult:   atom ('*' atom)* ;
```

```
atom:   ID
    |   INT
    ;
```

The way to think about this grammar is from the highest level downward. In rule **expr**, think of the references to **mult** as simply metatokens separated by the '+' operator. View 3*4+5*6 as the addition of two metatokens as if the expression were (3*4)+(5*6). Here is the parse tree as generated by ANTLRWorks' interpreter:



The deeper the nesting level, the higher the precedence of the operator matched in that rule. The loop subrules, (...)*, match the addition of repeated multiplicative metatokens such as 3*4+5+7.

Top-down parsers naturally associate operators left to right so that the earlier operators are matched correctly using a natural grammar. But, what about operators that are right-associative such as the exponentiation operator? In this case, you must use tail recursion to get the associativity right:

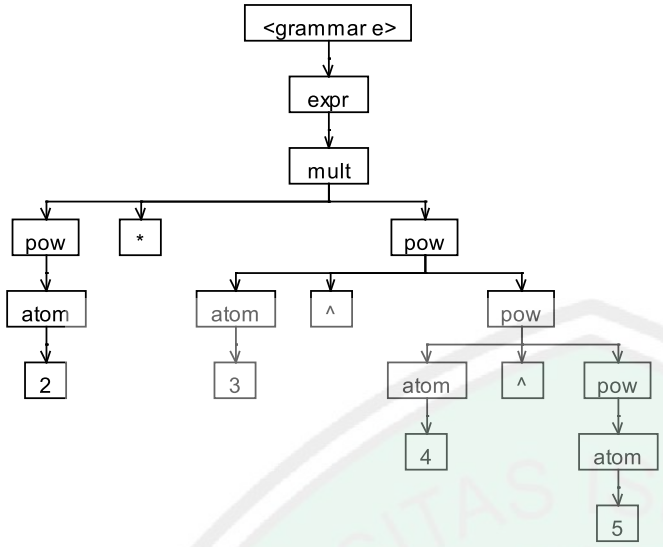
Even if you do not understand exactly how the precedence and associativity works, you can blindly accept these examples as grammar design patterns during the learning process.

```
expr:   mult ('+' mult)* ; // left-associative via (...)*
mult:   pow ('*' pow)* ;
pow :   atom ('^' pow)? ;  // right-associative via tail recursion
atom:   ID
    |   INT
    ;
```

The grammar derives input 2*3^4^5 as follows:

```
expr => mult
expr => pow * pow
expr => 2 * pow
expr => 2 * atom ^ pow
expr => 2 * 3 ^ pow
expr => 2 * 3 ^ atom ^ pow
expr => 2 * 3 ^ 4 ^ pow
expr => 2 * 3 ^ 4 ^ 5
```

The parse tree is as follows:



The first exponent operator has a complete subtree (also containing an exponent operator) as a right operand; consequently, the second exponent is evaluated first. Expression tree nodes must evaluate their children before they can perform the operation. This means the second exponent operator executes before the first, providing the necessary right associativity.

Non-*LL(*)* Decisions

Rule recursion can also cause trouble even when the rule references are not left-recursive. Although *LL(*)* DFA construction takes the parsing rule invocation stack into consideration, the resulting DFA will not have a stack. Instead, the DFA must use sequences of states. Consider the following grammar that allows zero or more labels on the front of each statement. Because the reference to rule **label** is common to both alternatives, ANTLR must try to see past it to the **ID** or 'return' token in order to distinguish between the alternatives.

```
grammar stat;

s : label ID '=' expr
  | label 'return' expr
  ;
label
  : ID ':' label // uses tail recursion to loop
  |
  ;
```

ANTLR reports two problems:

```
error(211): stat.g:3:5: [fatal] rule s has non-LL(*) decision due to
recursive rule invocations reachable from alts 1,2.  Resolve by
left-factoring or using syntactic predicates or using
backtrack=true option.
warning(200): stat.g:3:5: Decision can match input such as
"ID ':' ID ':'" using multiple alternatives: 1, 2
```

The first issue is that, without a stack, a DFA predictor cannot properly recognize the language as described by rule **label** because of the tail recursion. The second issue is derived from the fact that ANTLR tried to create a DFA anyway but had to give up after recursing a few times.[8] An easy fix for this grammar makes it trivially *LL(*)*:

```
grammar stat;

s : label ID '=' expr
  | label 'return' expr
  ;
label
  : (ID ':')*
  ;
```

The language is the same, but rule **label** expresses its repetitious nature with an EBNF construct rather than tail recursion. Figure 11.2, on the following page, shows the DFA for the decision in rule **s**. The EBNF looping construct maps to a cycle in the DFA between s1 and s4 and is clearer than tail recursion because you are explicitly expressing your intention to loop.

Generally, it is not possible to remove recursion because recursion is indispensable for describing self-similar language constructs such as nested parentheses (see Section 2.2, *The Requirements for Generating Complex Language*, on page 38). Imagine a simple language with expressions followed by '%' (modulo) or '!' (factorial):

```
se: e '%'
  | e '!'
  ;
e : '(' e ')'
  | ID
  ;
```

8. To be precise, ANTLR's analysis algorithm will follow recursive rule indications until it hits constant MAX_SAME_RULE_INVOCATIONS_PER_NFA_CONFIG_STACK in NFAContext. You can set this threshold using the -Xm option. After hitting the threshold, ANTLR will have created a DFA containing a state reachable upon ID:ID: that predicts both alternatives one and two of rule **s**. This nondeterminism results in the warning. Note that there might be

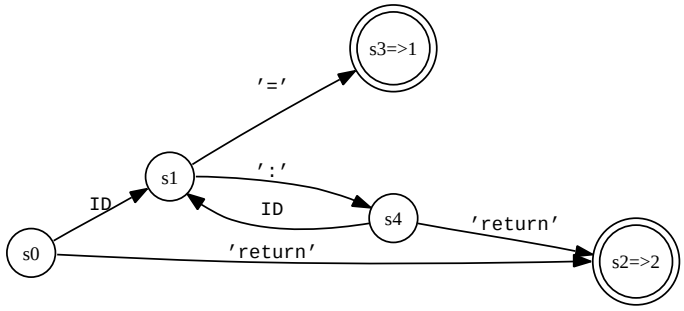


Figure 11.2: DFA predicting alternatives in rule **s** matching (ID ':')*

Not only can rule **e** match infinitely long sequences, the sequences must be properly nested, parenthesized expressions. There is no way to describe this construct without recursion (hence, a regular expression is insufficient). ANTLR will generate a DFA that favors alternative one in rule **se** unless it knows for sure that alternative two will succeed, such as when the input is ID!. ANTLR builds the DFA shown in Figure 11.3, on the following page, and warns this:

```
warning(200): e2.g:3:5: Decision can match input such as
"'(' '('" using multiple alternatives: 1, 2
```

The DFA does, however, correctly predict alternatives for input sequences not requiring deep recursive invocation of rule **e** such as v and (v).

Without left-factoring rule **se**, the only way to resolve this non-*LL(*)* decision is to allow ANTLR to backtrack over the reference to rule **e**, as discussed later in Chapter 14, *Syntactic Predicates*, on page 331. In this grammar, a parser could technically scan ahead looking for the '%' or '!', but in general this approach will not work. What if '%' were a valid binary operator as well as a suffix unary operator as shown? The only way to distinguish between the binary and unary suffix operators would be to properly match the expressions as part of the lookahead. Scanning ahead with a simple loop looking for the suffix operators amounts to a much weaker strategy than actually recognizing the expressions.

In some cases, however, ANTLR can deal with recursion as long as only one of the alternatives is recursive. The following grammar is *LL(*)* as long as the internal recursion overflow constant (specified by -Xm) is

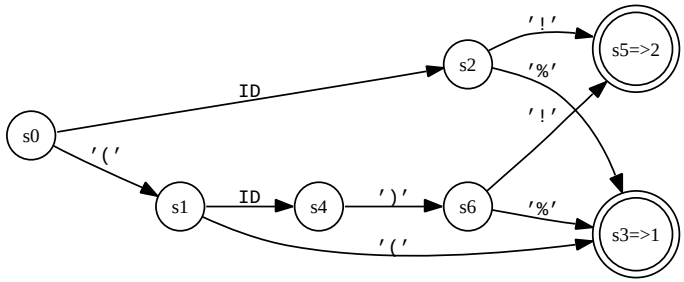
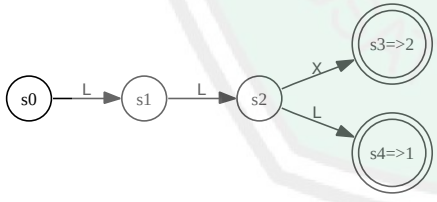


Figure 11.3: DFA predicting alternatives in rule **se**

sufficiently large; the default value is 4, meaning the analysis engine can recurse four times before hitting the threshold:

```
grammar t;
a : L a R
  | L L X
  ;
```

The analysis sees that **L** begins both alternatives and looks past it in both alternatives to see whether there is something that follows that will distinguish the two. In the first alternative, therefore, the analysis must enter rule **a** again. The first symbol that it can see upon reentry is **L**. Hence, the algorithm must continue again past that **L** recursively into rule **a** hoping for a lookahead symbol that will distinguish the two alternatives. Ultimately, the algorithm sees the **X** in the second alternative, which allows it to distinguish the two alternatives. Clearly, though, if the second alternative were recursive as well, this process would never terminate without a threshold. ANTLR generates the following DFA where state path s0, s1, s2, s3 predicts alternative two:



To illustrate what happens when you hit the recursion overflow threshold, consider the following invocation of ANTLR on the same grammar. The command-line option restricts the analysis engine so that it can recurse to **a** exactly once:

```
$ java org.antlr.Tool -Xm 1 t.g
ANTLR Parser Generator  Version 3.0  1989-2007
warning(206): t.g:2:5: Alternative 1: after matching input such as
L L decision cannot predict what comes next due to recursion
overflow to a from a
warning(201): t.g:2:5: The following alternatives are unreachable: 2
```

Because of the restriction that the analysis cannot recurse more than once, the analysis cannot enter **a** a second time when computing lookahead for the first alternative. It can't see past **L L** to another **L**. One more recursive examination of **a** would allow the analysis to distinguish the first alternative's lookahead from the **L L X** lookahead of the second. The recursion overflow threshold restricts only the maximum recursion depth, not the simple stack size of invoked rules, so it is not very restrictive.[9]

Nondeterministic Decisions

Once you get used to *LL* top-down parsers, you will not make many recursion mistakes. Most of your grammar problems will stem from *nondeterminisms*: the parser can go down two or more different paths given the same lookahead language.

Each lookahead sequence must uniquely predict an alternative. Just like in a maze with words written on the floor, if the next few words in your passphrase appear on the floor down both paths of a fork in the maze, you will not know which path to take to reach the exit. This section illustrates many of the common nondeterminisms you will encounter.

Most nondeterminisms arise because of ambiguities; all grammar ambiguities result in parser nondeterminisms, but some nondeterminisms are not related to ambiguities, as you will see in a moment.

9. The m=4 threshold makes sense because the Java grammar did not work with m=1 but did work with m=4. Recursion is sometimes needed to resolve some fixed lookahead decisions. Note: m=0 implies the algorithm cannot ever jump to another rule during analysis (stack size 0), m=1 implies you can make as many calls as you want as long as they are not recursive, and m=2 implies that you are able to recurse exactly once (that is, enter a rule twice from the same place).

Tracking Down Nondeterminisms

When tracking down nondeterminisms, the key is asking yourself how the grammar can match the indicated lookahead sequence in more than one way. ANTLRWorks was designed to be particularly good at helping you understand nondeterminisms, so this is easier than it used to be. With a little bit of experience, you will get good at figuring out what is wrong.

Once you discover exactly how the grammar can match a lookahead sequence in more than one way, you must generally alter the grammar so that ANTLR sees exactly one path. If you do not resolve a nondeterminism, ANTLR will always resolve it for you by simply choosing the alternative specified first in a decision.

Here is an obvious ambiguity:

```
r : ID {...}
  | ID {...}
  ;
```

to which ANTLR responds as follows:

```
warning(200): t.g:3:5: Decision can match input such as
"ID" using multiple alternatives: 1, 2
As a result, alternative(s) 2 were disabled for that input
warning(201): t.g:3:5: The following alternatives are unreachable: 2
```

Clearly, ANTLR could match **ID** by entering either alternative. To resolve the issue, ANTLR turns off alternative two for that input, which causes the unreachable alternative error. After removing **ID**, no tokens predict the second alternative; hence, it is unreachable. Figure 11.4, on the following page, illustrates how ANTLRWorks highlights the two paths (emphasized with thick lines here). You can trace the nondeterministic paths by using the cursor keys in ANTLRWorks.

Such an obvious syntactic ambiguity is not as crazy as you might think. In C++, for example, a typecast can look the same as a function call:

```
e : ID '(' exprList ')' // function-style typecast; E.g., "T(x)"
  | ID '(' exprList ')' // function call; E.g., "foo(32)"
  ;
```

The reality is that the syntax is inherently ambiguous—there is no amount of grammar shuffling that will overcome an ambiguity in the language syntax definition.

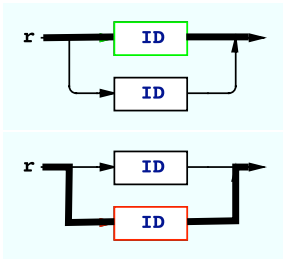


Figure 11.4: ANTLRWorks highlighting both paths predicted by **ID**

In the case of C++, however, knowledge about the **ID**'s type from the symbol table (that is, whether it is a type name or function name) neatly resolves the issue. Use a semantic predicate to consult the symbol table per Chapter 13, *Semantic Predicates*, on page 317.

Consider a more subtle ambiguity problem. Imagine you want to build a language with optional semicolons following statements (such as Python), but where semicolons can also be a statements:

```
grammar t;
slist
    :   ( stat ';'? )+
    ;
stat:   ID '=' expr
    |   ';'
    ;
```

The optional ';'? subrule in rule **slist** cannot decide whether to match ';' immediately or to bypass the subrule and reenter **stat** to match it as a proper, stand-alone statement. ANTLR reports this:

```
warning(200): t.g:3:11: Decision can match input such as
"';'" using multiple alternatives: 1, 2
As a result, alternative(s) 2 were disabled for that input
```

ANTLRWorks highlights both paths in the syntax diagram shown in Figure 11.5, on the following page.

The solution is to either make semicolons required or make them only statements. Semicolons should not be both statement terminators and statements as shown previously. Naturally, a good language designer would simply fix the language. With the grammar as is, though, ANTLR automatically resolves the nondeterminism *greedily* (see Section 4.3, *Extended BNF Subrules*, on page 98 for information about the **greedy**

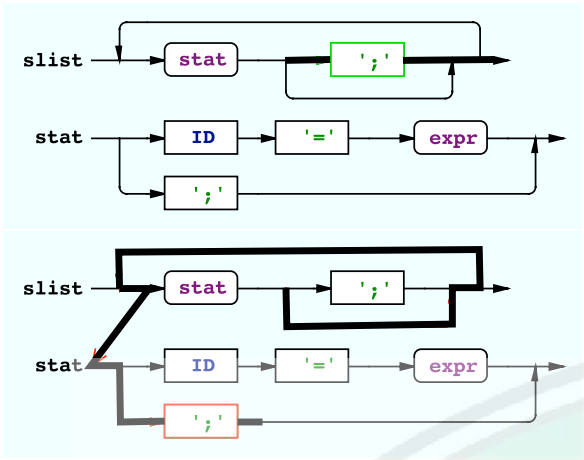


Figure 11.5: Syntax diagrams for **slist** and **stat**

semicolons immediately following a statement if one exists. A greedy decision is one that decides to match all possible input as soon as possible, rather than delegating the match to a future part of the grammar. Input `x=1;;` would match the first ';' in the optional ';'? subrule and the second as a statement in rule **stat**.

The most common form of this ambiguity, and one whose automatic resolution is handled naturally by ANTLR, is the **if-then-else** ambiguity:

```
grammar t;
stat:   'if' expr 'then' stat ('else' stat)?
    |   ID '=' expr ';'
    ;
```

```
warning(200): <t.g>:2:29: Decision can match input such as
"'else'" using multiple alternatives: 1, 2
As a result, alternative(s) 2 were disabled for that input
warning(201): <t.g>21:29: The following alternatives are unreachable: 2
```

The issue is similar to the optional semicolon in that ANTLR cannot decide whether to match 'else' immediately or bypass the subrule and match it to a previous 'if'. In other words, how should ANTLR interpret the following input?

```
if done then
if alreadySaved then x=2;
else x=3;
```

Should `else x=3;` bind to the second and most recent **if** or to the first one?

The grammar allows both. Language designers have decided that it's most natural to bind to the most recent **if** (greedily), which is fortunately the way ANTLR automatically resolves ambiguities. ANTLR generates a warning, but you can safely ignore it.[10]

Sometimes the explicit alternatives (nonexit branches of an EBNF construct) within a decision are ambiguous. Consider the following grammar for a Java-like language that can have nested code blocks in statements and also a code block at the class level that acts like a default constructor (Java uses these for default constructors for anonymous inner classes):

```
classDef
    :   'class' ID '{' decl* '}'
    ;
slist:  decl
    |   stat
    ;
decl:   field
    |   method
    |   block // default ctor code block
    ;
stat:   block // usual statement nested code block
    |   'return' expr ';'
    ;
block:  '{' slist '}'
    ;
```

Rule **slist** has a problem in that both alternatives eventually reach rule **block**, making the decision totally ambiguous for code blocks. Should ANTLR match a code block by entering rule **decl** or by entering **stat**? This matters because you are likely to have very different actions depending on the code block's context.

In this case, the grammar is loosely written because **decl** should recognize a code block only at the class level, not at the statement level. Any code block within a statement should be interpreted simply as a nested code block, not a constructor.

Tightening up the grammar to use context information makes it clearer and removes the ambiguity:

10. At some point ANTLR will let you silence warnings for decisions that ANTLR properly resolves.

```
classDef
    :   'class' ID '{' member* '}'
    ;
member
    :   decl
    |   block // default ctor code block
    ;
slist:  decl
    |   stat
    ;
decl:   field
    |   method
    ;
...
```

The addition of **member** makes it clear that the grammar should interpret a code block matched within the class definition differently than a code block matched via **slist**.

Sometimes a grammar is ambiguous but is the most natural and correct way to express the language. In the following grammar (pulled from the larger Java grammar at http://www.antlr.org), rule **castExpression** indicates that only typecasts based upon primitive type names such as **int** can prefix expressions that have '+' or '-'. Expressions such as (Book)+3 make no sense, and it is correct to make such cases illegal using the syntax of the language. Rule **castExpression** is a natural way to express the restriction, but it is ambiguous.

```
unaryExpression
    :   '+' unaryExpression
    |   '-' unaryExpression
    |   unaryExpressionNotPlusMinus
    ;
unaryExpressionNotPlusMinus
    :   '~' unaryExpression
    |   castExpression
    |   primary
    ;
castExpression
    :   '(' primitiveType ')' unaryExpression
    |   '(' type ')' unaryExpressionNotPlusMinus
    ;
```

primitiveType and **type** are defined as follows:

```
primitiveType
    :   'int'
    |   'float'
    ;
type:   (primitiveType|ID)  ('[' ']')*
    ;
```

The problem is that rule **type** is a superset of **primitiveType**, so both **castExpression** alternatives can match (int)34, for example. Without making a variant of **type**, there is no way to fix this ambiguity using pure *LL(*)*. A satisfying solution, however, involves syntactic predicates whereby you can simply tell ANTLR to try the two alternatives. ANTLR chooses the first alternative that succeeds (see Chapter 14, *Syntactic Predicates*, on page 331):

```
castExpression
// backtrack=true means to just try out the alternatives. If
// the first alternative fails, attempt the second alternative.
options {backtrack=true;}
    :   '(' primitiveType ')' unaryExpression
    |   '(' type ')' unaryExpressionNotPlusMinus
    ;
```

For completeness, it is worth mentioning one of the rare situations in which ANTLR reports a nondeterminism (as opposed to a recursion issue) that is not related to a grammar ambiguity. It turns out that when people say "*LL*," they actually mean "*SLL*" (*strongLL*). The *strong* term implies stronger constraints so that *SLL(k)* is weaker than *LL(k)* for k>1 (surprisingly, they are identical in strength for k=1). ANTLR and all other *LL*-based parser generators accept SLL grammars (grammars for which an *SLL* parser can be built). Using the proper terminology, you can say that the following grammar is *LL(2)*, but it is not *SLL(2)*:[11]

```
grammar t;
s : X r A B
  | Y r B
  ;
r : A
  |
  ;
```

Rule **r** cannot decide what to do upon lookahead sequence AB. The parser can match A in **r** and then return to the second alternative of **s**, matching B following the reference to **r**. Alternatively, the parser can choose the empty alternative in **r**, returning to the first alternative of **s** to match AB. The problem is that ANTLR has no idea which alternative of **s** will be invoking **r**. It must consider all possible rule invocation sites when building lookahead DFAs for rule **r**. ANTLR reports this:

```
warning(200): <t.g>:5:5: Decision can match input such as
"A A..B" using multiple alternatives: 1, 2
As a result, alternative(s) 2 were disabled for that input
```

11. Following convention, this book uses *LL* even though *SLL* is more proper.

This message is correct for S*LL(*)*, but it is a weakness in the parsing strategy, due to a lack of context, rather than a grammar ambiguity as described previously. Naturally, you could trivially rewrite the grammar, duplicating rule **r**, or ANTLR could generate different methods for **r** depending on the context. This transformation from *LL* to SLL is always possible but in the worst case results in exponentially large grammars and parsers.

ANTLR resolves nondeterminisms by predicting the first of multiple alternatives that can match the same lookahead sequence. It removes that lookahead sequence from the prediction set of the other alternatives. If ANTLR must remove all lookahead sequences that predict a particular alternative, then ANTLR warns you that the alternative is unreachable.

In previous versions, ANTLR presented you with all possible lookahead sequences for each nondeterminism. In v3, ANTLR displays a short lookahead sequence from the nondeterministic paths within the lookahead DFA to make it easier for you to find the troublesome paths. ANTLRWorks highlights these paths for you in the syntax diagram and is a great grammar debugging aid. *Improved in v3.*

In summary, all grammar ambiguities lead to parser nondeterminisms, but some nondeterminisms arise because of a weakness in the parsing algorithm. In these cases, you should consider altering the grammar rather than assuming that ANTLR will resolve things properly.

Lexer Grammar Ambiguities

Ambiguities result in nondeterministic lexers just like they do in parsers, but lexers have a special case not present in parsers. Recall that ANTLR builds an implicit **nextToken** rule that has all non-**fragment** tokens as alternatives. ANTLR builds a DFA that, at runtime, decides which token is coming down the input stream and then jumps to that token rule. Again, the **nextToken** prediction DFA examines only as much lookahead as necessary to decide which token rule to jump to.

As with the parser, the lexer sometimes has some fundamentally ambiguous constructs that ANTLR handles naturally. Consider the following grammar that defines a keyword and an identifier:

```
BEGIN : 'begin' ;
ID : 'a'..'z'+ ;
```

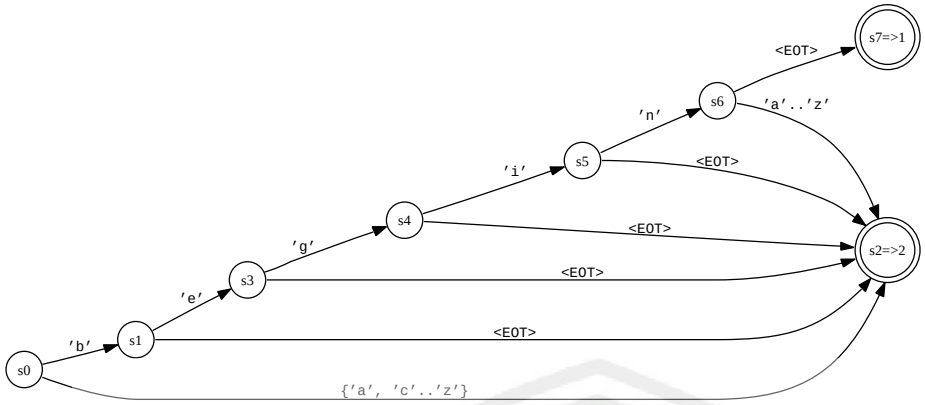


Figure 11.6: Lexer rule prediction DFA for keyword rules **BEGIN** vs. **ID**

The implicitly constructed **nextToken** rule is as follows:

```
nextToken
    :   BEGIN
    |   ID
    ;
```

Rule **nextToken**'s two alternatives are ambiguous upon 'begin' because rule **BEGIN** is a special case of rule **ID** (keywords are also lexically identifiers). This is such a common situation that ANTLR does not emit warnings about these ambiguities. Just like in the parser, ANTLR resolves the ambiguity by favoring the rule specified first in the grammar. Figure 11.6 shows the DFA that correctly predicts **BEGIN** vs. **ID**. The EOT (end of token) DFA edge label means "anything else," so input "beg␣" (with a space character afterward) predicts the second alternative (rule **ID**) via s0, s1, s3, s4, s2 since the space character is not i.

It is not always the case that specifying superset rules is OK. Using the following two rules is a common mistake:

```
INT : DIGIT+ ;
DIGIT : '0'..'9' ; // needs to be fragment
```

ANTLR reports this:

```
warning(208): t.g:2:1: The following token definitions are
                       unreachable: DIGIT
```

The prediction DFA must choose which rule to match upon saying a

single-digit such as 4. When one rule directly references another rule on the left edge, the referenced rule must usually be a **fragment** rule. In this case, clearly **DIGIT** is only a helper rule, and the parser is not expecting to see a **DIGIT** token. Make **DIGIT** a fragment rule:

```
INT : DIGIT+ ;
fragment
DIGIT : '0'..'9' ;
```

Another form of this same issue appears in the following grammar:

```
NUMBER : INT | FLOAT ;
INT : '0'..'9'+ ;
FLOAT : '0'..'9'+ ('.' '0'..'9'*)? ; // simplified
```

Both integers and floating-point numbers match, but they both match via token **NUMBER**. Rules **INT** and **FLOAT** are unreachable. In this case, however, the problem relates to the boundary between lexer and parser rules. You should restrict lexer rules to matching single lexical constructs whereas rule **NUMBER** is one of two different lexical constructs. That indicates you should draw the line between lexer and parser rules differently, as follows, where rule **number** is now a parser rule and correctly indicates a set of tokens representing numbers:

```
number : INT | FLOAT ; // a parser rule
INT : '0'..'9'+ ;
FLOAT : '0'..'9'+ ('.' '0'..'9'*)? ; // simplified
```

The wildcard . operator is often ambiguous with every other rule but can be very useful as an else clause rule if you use it last:

```
BEGIN : 'begin' ;
ID : 'a'..'z'+ ;
OTHER: . ; // match any other single character
```

Rule **OTHER** matches any single character that is not an **ID**. Be careful not to use .+ as a greedy subrule all by itself without anything following it. Such a subrule consumes all characters until the end of file. If you put a grammar element after the .+ loop, it will consume only until it finds that element.

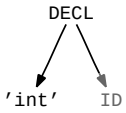
Tree Grammar Ambiguities

Tree grammar ambiguity warnings sometimes reference two implicitly defined tokens: **UP** and **DOWN**. ANTLR parses trees the same way it parses token streams by serializing trees into streams of nodes. The special imaginary tokens indicate the beginning and end of a child list. Consider the following tree grammar rule where both alternatives

can match the same input tree node sequence, assuming **type** matches matches 'int':

```
a : ^(DECL 'int' ID)
  | ^(DECL type ID)
  ;
```

Both alternatives can match a tree with the following structure and token types:



The associated tree node stream for that tree is as follows:

```
DECL DOWN 'int' ID UP
```

ANTLR reports this:

```
warning(200): u.g:3:5: Decision can match input such as
"DECL Token.DOWN 'int' ID Token.UP" using multiple alternatives: 1, 2
As a result, alternative(s) 2 were disabled for that input
```

After resolving grammar ambiguities, you can begin testing your grammar for correctness. Unfortunately, a lack of grammar nondeterminisms does not mean that the resulting parser will behave as you want or expect. Checking your grammar for correctness can highlight a number of other problems: Why is this grammar improperly matching a particular input? Why is there a syntax error given this input? Or even, why is there no syntax error given this ungrammatical input? ANTLRWorks' debugger is the best tool for answering these questions. ANTLRWorks has breakpoints and single-step facilities that allow you to stop the parser when it reaches an input construct of interest rather than merely breaking at a grammar location. ANTLRWorks' debugger can even move backward in the parse after a syntax error to examine the events leading up to it. ANTLRWorks' interpreter is also useful for figuring out how a particular input sequence matches.

This chapter described why you need *LL(*)* and how it works. It also explained grammar ambiguities and recognizer nondeterminisms by example. Used in conjunction with the semantic and syntactic predicates described in the next three chapters, *LL(*)* is close to the most powerful parsing algorithm that the average programmer will find accessible. The next chapter demonstrates how *LL(*)* augmented with predicates can resolve some difficult language recognition problems.

Chapter 12

Using Semantic and Syntactic Predicates

LL(*) is a powerful extension to *LL(k)* that makes it much easier to write natural grammars and build grammars for difficult languages. The previous chapter explained how *LL(*)* uses DFAs to scan arbitrarily far ahead looking for input symbols and sequences that distinguish alternatives. *LL(k)*, on the other hand, can see only fixed *k* symbols ahead. Even though *LL(*)* is much better than *LL(k)*, it still has its weaknesses, particularly when it comes to recursive rules. This chapter illustrates how to use two powerful constructs, *syntactic* and *semantic predicates*, that boost the power of *LL(*)* to the point where it is essentially indistinguishable from recognizers you could build by hand. Predicates alter the parse based upon runtime information. As we'll see in this chapter, predicates help do the following:

- Distinguish syntactically identical language constructs such as type names vs. variable or method names
- Resolve *LL(*)* recognizer nondeterminisms; that is, overcome weaknesses in *LL(*)*
- Resolve grammar ambiguities derived from true language ambiguities by prioritizing alternatives
- Formally encode semantic, syntactic, and other contextual constraints written loosely in English

Predicated-*LL(*)* recognizers readily match almost any context-free grammar and can even deal with context-sensitive language constructs.

ANTLR pioneered the use of predicates in practical parser generators, and you will find it easier to build natural grammars in ANTLR than in other parser generators—at least when it comes to embedding arbitrary actions, supporting context-sensitive parsing, and resolving ambiguous constructs.

Many real language problems require predicated parsers, but there is essentially nothing written about the practical use of predicated parsing. Worse, predicates can be a fairly complicated subject because they are most useful for difficult language implementation problems. For these reasons, it is worth devoting a significant portion of this book to predicated parsing. The discussion is broken into three chapters: how to use predicated parsing to solve real language recognition problems followed by more formal treatments of semantic and syntactic predicates in Chapter 13, *Semantic Predicates*, on page 317 and Chapter 14, *Syntactic Predicates*, on page 331. The last two chapters explain all the variations, hazards, and details concerning predicates, whereas this chapter emphasizes the application of predicates. You should read this chapter first to get the most out of the two subsequent chapters.

This chapter presents a number of situations in which pure *LL(*)* parsing provides an unsatisfactory solution or is even completely insufficient without semantic or syntactic predicates. Let's begin by examining the different kinds of semantic predicates and how they can resolve syntactic ambiguities.

12.1 Resolving Syntactic Ambiguities with Semantic Predicates

Some languages are just plain nasty to parse such as C++ and Ruby, because of context-sensitive constructs. Context-sensitive constructs are constructs that translators cannot interpret without relying on knowledge about surrounding statements. The unfortunate truth is that we cannot build a correct context-free grammar for many languages (see Section 4.1, *Describing Languages with Formal Grammars*, on page 87). We need the ability to drive recognition with runtime information such as symbol table information. Using semantic predicates to alter the parse is analogous to referring to a notebook (such as a symbol table) while navigating the maze. The notebook might contain information about where you've walked in the maze and how the maze matched previous words in the passphrase. The following sections illustrate solutions to a number of difficult language recognition problems that typically flummox everyone except language tool experts.

First let's demonstrate that some problems are hard to describe with a pure context-free grammar (that is, a grammar without semantic predicates).

The Three Semantic Predicate Variations

The semantics of a language refer, loosely, to everything beyond syntax. Another way to look at it is that you specify syntax with a grammar and semantics with embedded actions. Semantics can mean everything from the relationship between input symbols to the interpretation of statements. Although you can sometimes use the grammar to enforce certain semantic rules, most of the time you'll need *semantic predicates* to encode semantic constraints and other language "rules."

Consider the problem of matching an element at most four times. Surprisingly, this is difficult to specify with syntax rules. Using a pure context-free grammar (in other words, without semantic actions or predicates), you must delineate the possible combinations:

```
data:   BYTE BYTE BYTE BYTE
    |   BYTE BYTE BYTE
    |   BYTE BYTE
    |   BYTE
    ;
```

When four becomes a larger number, the delineation solution quickly breaks down. An easier solution is to match as many **BYTE** tokens as there are on the input stream and then, in an action, verify that there are not too many:

```
data:   ( b+=BYTE )+ {if ( $b.size()>4 ) «error»;}
    ;
```

Or, you can use the formal equivalent provided by ANTLR called a *validating semantic predicate*. A validating semantic predicate looks like an action followed by a question mark:

```
data:   ( b+=BYTE )+ {$b.size()<=4}?
    ;
```

Validating semantic predicates are boolean expressions that the recognizer evaluates at runtime. If the expression is false, the semantic predicate fails, and the recognizer throws a FailedPredicateException.

In other cases, no context-free grammar notation exists to specify what you want because an alternative must be gated in or out depending on runtime information. No amount of static grammar analysis will help.

For example, certain languages have extensions that must be turned on or off depending on a command-line switch. For example, Java has **enum** and **assert**; GCC has C extensions. Naturally, the recognizer could allow all extensions and then use an action to emit a syntax error if it sees a disallowed extension. Instead, ANTLR provides a more formal solution called a *gated semantic predicate* (see Section 13.2, *Gated Semantic Predicates Switching Rules Dynamically*, on page 325). Gated semantic predicates look like {...}?=> and enclose a boolean expression that is evaluated at runtime. The gated semantic predicates dictates whether the recognizer can choose that alternative. When false, the alternative is invisible to the recognizer. The following rule fragment is from the statement rule in a Java grammar. The gated semantic predicate uses the boolean variable allowAssert to turn the **assert** statement on and off dynamically.

New in v3.

```
stat:   ifStat
    |   {allowAssert}?=> assertStat
    ...
    ;
```

As another example, reconsider the earlier example matching four **BYTE**. If you want a syntax error rather than a FailedPredicateException, you can use a gated semantic predicate:

```
data
@init {int n=1;} // n becomes a local variable
    :   ( {n<=4}?=> BYTE {n++;} )+ // enter loop only if n<=4
    ;
```

The **BYTE** alternative becomes invisible after the parser has seen four **BYTE** tokens. ANTLR generates the following code for the (...)+ subrule:

```
do {
    int alt1=2;
    int LA1_0 = input.LA(1);
    // predict alternative one if lookahead is consistent with
    // first (and only) alternative of loop and if gated predicate
    // is true.
    if ( (LA1_0==BYTE) && (n<=4)) { // evaluate gated predicate
        alt1=1;
    }
    switch (alt1) {
        ...
    }
} while (true);
```

The gated semantic predicate is part of the decision expression that decides whether to enter an alternative one. There are plenty of real-world examples such as SQL and its vendor variations where you would like to dynamically turn on and off subsets of the language. In this way, you can create one large static grammar and then use gated semantic predicates to selectively turn on and off subsets.

The final variant of the semantic predicate is called a *disambiguating semantic predicate* and looks like {«*expression*»}?. Disambiguating semantic predicates are predicates that *LL(*)* recognizers include in prediction decisions just like gated semantic predicates. The difference is that decisions use disambiguating semantic predicates only when syntax alone is insufficient to distinguish between alternatives. In two general situations, disambiguating semantic predicates really help: when a property of a token must dictate how the parser interprets it and when a surrounding construct or some arbitrary boolean expression must alter how the parser matches the current construct. The following sections illustrate how to use disambiguating semantic predicates.

Keywords as Variables

Consider those twisted languages, written by social deviants, that allow keywords to be used as variables like this: `if if call call;` or `call if;`. The context dictates whether an identifier is a keyword or a variable. At the beginning of a statement, `if` is a keyword, but it is a variable in an expression. One possible solution is to treat all identifiers as variables except in the specific cases where you know an identifier must be a keyword. Because context-free grammars cannot test the attributes of a token, you must use semantic predicates to check that the text of an identifier matches a keyword. In the following grammar, the disambiguating semantic predicates indicate the semantic validity of matching an identifier as a keyword:

Download predicates/keywords/Pred.g

```
prog: stat+ ;

/** ANTLR pulls predicates from keyIF and keyCALL into
 *  decision for this rule.
 */
stat: keyIF expr stat
    | keyCALL ID ';'
    | ';'
    ;
```

```
expr: ID
    ;

/** An ID whose text is "if" */
keyIF : {input.LT(1).getText().equals("if")}? ID ;

/** An ID whose text is "call" */
keyCALL : {input.LT(1).getText().equals("call")}? ID ;
```

In a mechanism unique to ANTLR, the semantic predicates in rules **keyIF** and **keyCALL** are *hoisted* out of their native rules into the decision for rule **stat**. More specifically, the lookahead prediction DFA shown in Figure 12.1, on the following page incorporates semantic predicates when it finds that syntax alone is insufficient to distinguish between alternatives.

Notice that the DFA will evaluate the predicates only upon ambiguous sequence ID ID ;. Input ID ID ID, for example, can be an **if** statement only because it is too long to be a **call** statement. The DFA predicts alternative one without evaluating a predicate. For this grammar, the decision needs to know only its grammatical context ("start of statement") and the next token's text attribute.

Sometimes, however, a decision needs context information about how the parser interpreted previous statements. Typically these previous statements are variable, method, or type definitions. The next section shows how to resolve a syntactic ambiguity in the Ruby language with its optional method call parentheses.

Ruby Array Reference vs. Method Call

The Ruby language is nice, but it has some syntactic ambiguities. For example, a[i] can be either an array reference or a method call with an array return value. The proper interpretation depends on how the program previously defines a.[1] The following Ruby code fragment uses a as an array; hence, a[i] is an array reference:

```
a = [20,30]
puts a[1]
```

1. The situation is made worse by the lack of static typing in Ruby because you must look backward in the source code for a prior assignment to a even if it's inside a nested conditional. a [i] with a space after the variable name could even mean that [i] (a list with i in it) is a parameter if a is a method.

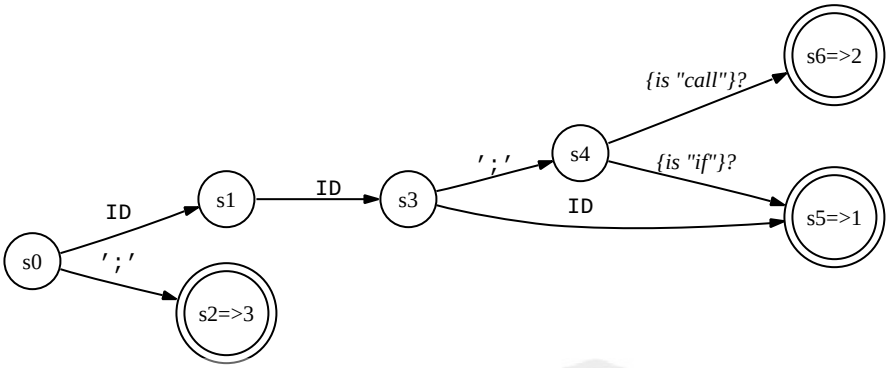


Figure 12.1: Prediction DFA for rule **stat** that distinguishes between **if** and **call** statements when keywords can be variables

If a is defined as a method, then a[1] represents a method call (call the method and then index into the array that it returns):

```
def a
  return [20,30]
end
puts a[1]
```

In both cases, Ruby prints "30" to the console. At runtime, it is not ambiguous because the [] (array reference) message is sent to whatever object a is. But, imagine you want to perform static analysis of Ruby source code and print all the method references. A simplified grammar demonstrating this notation is as follows:

Download predicates/ruby/Ruby.g

```
grammar Ruby;

expr:   atom ('+' atom)* // E.g., "a[i]+foo[i]"
    ;

atom:   arrayIndex
    |   methodCall ('[' INT ']')? // E.g., "foo[i]" or "foo(3,4)[i]"
    ;

arrayIndex
    :   ID '[' INT ']' // E.g., "a[i]"
    ;

methodCall
    :   ID ( '(' expr (',' expr)* ')' )? // E.g., "foo" or "foo(3,4)"
    ;
//...
```

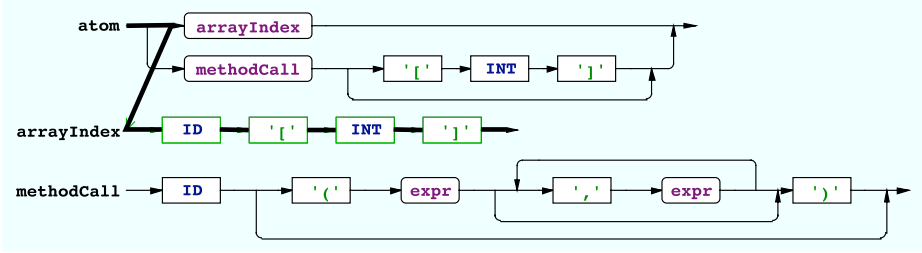


Figure 12.2: syntax diagram illustrating the path ANTLR chooses for ambiguous input ID [INT]

ANTLR reports this:

```
$ java org.antlr.Tool Ruby.g
ANTLR Parser Generator  Version 3.0  1989-2007
warning(200): Ruby.g:6:9: Decision can match input such as
"ID '[' INT ']'" using multiple alternatives: 1, 2
As a result, alternative(s) 2 were disabled for that input
```

The syntax for an array index and the syntax for the method invocation without parentheses are syntactically identical, so you cannot tell the difference just by looking at the syntax. Rule **atom** is nondeterministic because of the ambiguity. Figure 12.2 illustrates the path that the recognizer will take. Figure 12.3, on the following page illustrates the path that is syntactically ambiguous with the first alternative and that the recognizer will not take.

Adding semantic predicates resolves this ambiguity nicely where isArray() and isMethod() look up their token arguments in a symbol table that records how variables are used:

```
arrayIndex
    :   {isArray(input.LT(1))}? ID '[' INT ']'
    ;

methodCall
    :   {isMethod(input.LT(1))}? ID ('(' expr (',' expr)* ')')?
    ;
```

During a parse, the two semantic predicates test to see whether the next symbol is an array variable or a method.

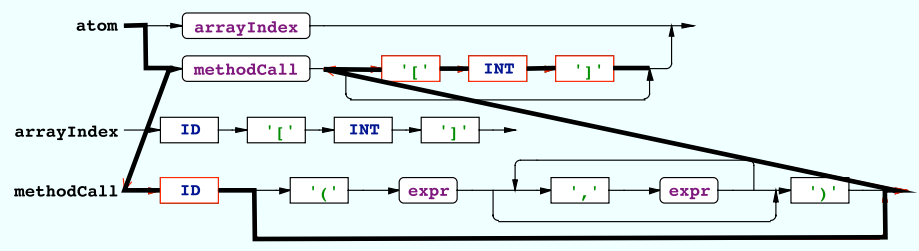
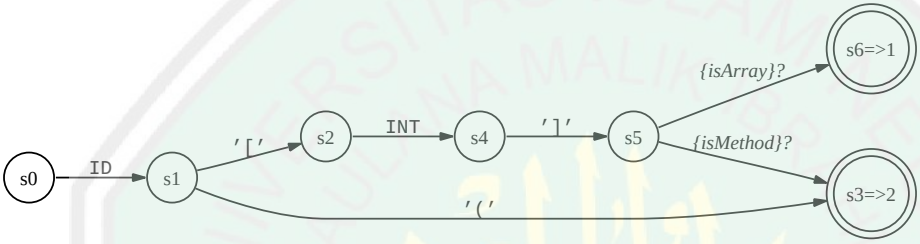


Figure 12.3: syntax diagram illustrating the path ANTLR does not choose for ambiguous input ID [INT]

Here is the lookahead prediction DFA for rule **atom** that illustrates how the parser incorporates the semantic predicates:



Notice that upon ID '(', the decision immediately predicts alternative two (via s0, s1, s3) because that input can begin only a method call.

This Ruby example illustrated how you can distinguish between two alternatives. The next section uses an example from C to show how semantic predicates can alter a looping subrule's exit test.

C Type Names vs. Variables

C is another language whose grammar needs a semantic predicate. The **typedef** keyword introduces new types that are available later in the program:

```
typedef int I;
I a; // define a as an int
```

C also allows some rather strange-looking declaration-modifier orderings such as the following:

```
int register unsigned g;
I register i;
```

The easiest way to encode this in a grammar is simply to loop around the various declaration modifiers and types even if some combinations are not semantically valid:

```
static register int i;
```

Using a pure context-free grammar, the only other way to deal with this would be to try to delineate all possible combinations. That is tedious and awkward because you would be trying to enforce semantics with syntax. The more appropriate way to draw the line between syntax and semantics for C is to allow the parser to match an arbitrary modifier list and have the compiler's semantic phase examine the list for nonsensical combinations. The simplified and partial grammar for a C declaration looks like this:

```
declaration
    :   declaration_specifiers declarator? ';' // E.g., "int x;"
    ;

declarator
    :   '*' declarator // E.g., "*p", "**p"
    |   ID
    ;

declaration_specifiers
    :   (   storage_class_specifier // E.g., "register"
        |   type_specifier
        |   type_qualifier          // E.g., "const", "volatile"
        )+
    ;

type_specifier
    : 'void'
    | 'int'
    | ...
    | 'unsigned'
    | struct_or_union_specifier // E.g., "struct {...}", "struct a"
    | type_id
    ;

type_id
    :   ID
    ;
```

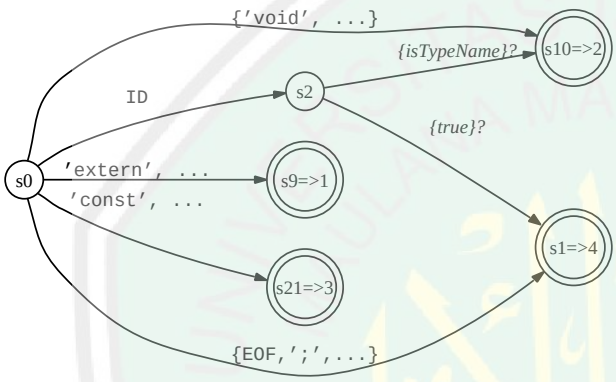
The problem is that the looping subrule in **declarator_specifiers** and rule **declarator** can both begin with an **ID** token. Upon **ID**, the subrule loop does not know whether to continue matching or to exit and match the identifier in **declarator**. Now, we know as humans that `x y;` syntacti-

cally must be a variable declaration and, hence, that x must be a user-defined type. Unfortunately, the (. . .)+ loop has no idea how many times it has gone around. The definition of the EBNF construct simply does not incorporate the notion of history. Syntax cannot resolve the recognizer nondeterminism.

Semantic predicates provide a much simpler solution. You can use the natural grammar with a small amount of semantic testing to tell ANTLR when an identifier is a type name:

```
type_id
    :   {isTypeName(input.LT(1).getText())}? ID
    ;
```

The predicate is hoisted into the prediction DFA for the (...)+ subrule in **declaration_specifiers**:



Upon **ID**, the DFA reaches s2 and predicts alternative two (**type_id**) if the identifier is defined as a type name. Otherwise, the DFA fixes alternative four, which is the loop exit branch. In other words, the loop exits when it sees an identifier that is not a type because this must be a variable or method name (the declarator).

To make this semantic predicate work, other parts of the grammar must add type names to a symbol table upon **typedef**. Unfortunately, tracking C symbols properly is a bit involved, particularly using ANTLR's fancy dynamic scopes, as shown next. You can skip to the next section on C++ if you don't care about those details. The point related to semantic predicates has already been made.

To make life easier when passing information between rules, your solution can use dynamic scopes, as described in Section 6.5, *Dynamic Attribute Scopes for Interrule Communication*, on page 148. You'll need a global dynamic scope to track type names because multiple rules will share the same stack of scopes (rules **translation_unit**, **function_definition**, **struct_or_union_specifier**, and **compound_statement**):

```
scope Symbols {
    Set types; // track types only for example
}
```

In this way, the rules that represent C scopes use ANTLR specification scope Symbols; to share the same stack of scopes. For example, here is **translation_unit**:

```
translation_unit
scope Symbols; // entire file is a scope; pushes new scope
@init {
  $Symbols::types = new HashSet(); // init new scope
}
    : external_declaration+
    ;
```

Because of the distance between the **typedef** keyword and the actual **ID** token recognition in rule **declarator**, you must pass information from the **declaration** rule all the way down to **declarator**. The easiest way to do that is to declare a rule-level dynamic scope with a boolean that indicates whether the current declaration is a **typedef**:

```
declaration
scope {
  boolean isTypedef;
}
@init {
  $declaration::isTypedef = false;
}
    : {$declaration::isTypedef=true;} // special case, look for typedef
      'typedef' declaration_specifiers declarator ';'
    | declaration_specifiers declarator? ';'
    ;
```

Any rule, such as **declarator** ultimately invoked from **declaration**, can access the boolean via $declaration::isTypedef:

```
declarator
    :   '*' declarator // E.g., "*p", "**p"
    |   ID
        {
        // if we're called from declaration and it's a typedef.
        // $declaration.size() is 0 if declaration is not currently
        // being evaluated.
```

```
        if ($declaration.size()>0&&$declaration::isTypedef) {
            // add ID to list of types for current scope
            $Symbols::types.add($ID.text);
            System.out.println("define type "+$ID.text);
        }
        }
    ;
```

This example illustrated how prior statements can affect future statements in C. The next section provides an example where a phrase in the future affects the interpretation of the current phrase.

C++ Typecast vs. Method Call

In C++, the proper interpretation of an expression might depend on *future* constructs. For example, `T(i)` can be either a constructor-style typecast or a method call depending on whether `T` is a type name or a method. Because `T` might be defined below in a class definition file, prior context is insufficient to properly interpret `T(i)`. The recognizer needs future context in a sense.

You can solve this dilemma in two ways. The first solution involves using a parser generator based upon GLR [Tom87] that allows all context-tree grammars including ambiguous grammars such as this one for the typecast vs. method call ambiguity. The parser returns a *parse forest*, rather than a single parse tree, that contains all possible interpretations of the input. You must make a pass over the trees to define methods and types and then make a second pass to choose which interpretation is appropriate for each ambiguous construct.

Using ANTLR, you can implement a similar strategy. Build a single tree for both constructs with a subtree root that represents both cases:

```
primary
    :   ID '(' exprList ')' // ctor-style typecast or method call
        -> ^(TYPECAST_OR_CALL ID exprList)
    |   ID
    |   INT
   ...
    ;
```

Then, similar to the GLR solution, walk the tree, and flip the type of the node once you have the complete symbol table information. The solution does not always work because some constructs need a completely different tree. You must parse the input twice no matter how you want to think about this problem because of forward references.

The second solution, using ANTLR, is not particularly satisfying either. You can parse a file twice, once to find the definitions and then a second time to distinguish between syntactically ambiguous constructs. You might even be able to do a quick initial "fuzzy" pass over the input file just looking for method and type definitions to fill up your symbol table and then parse the code again for real. Because lexing is expensive, tokenize the input only once—pass the token buffer to the second pass to avoid relexing the input characters.

No matter how you approach this problem, there is no escaping multiple passes. This example illustrates again that you can draw the line between syntax and semantics in different places.

Once you have complete symbol table information, the second pass can use semantic predicates to distinguish between typecasts and method calls where isType() looks up its token argument to see whether it is defined as a type in the symbol table:

```
primary
    :   {isType(input.LT(1))}? ID '(' expr ')' // ctor-style typecast
        -> ^(TYPECAST ID expr)
    |   ID '(' exprList ')'                    // method call
        -> ^(CALL ID exprList)
    |   ID
    |   INT
   ...
    ;
```

It is because of this ambiguity and many others that language implementers loath C++.

Semantic predicates resolve context-sensitivity problems in grammars but are sometimes used to examine the token stream ahead of the current position in order to make parsing decisions. In a sense, such semantic predicates are like manually specified lookahead DFA. Although ultimately powerful because semantic predicates are unrestricted actions in the target language, it is better to use a formal method to describe arbitrary lookahead.

In the next section, we'll look at formal solutions to a number of non-*LL(*)* problems from Java, C, and C++ that require arbitrary lookahead.

12.2 Resolving Ambiguities and Nondeterminisms with Syntactic Predicates

ANTLR supports arbitrary lookahead in the form of *syntactic predicates* that are similar to semantic predicates except that they specify the syntactic validity of applying an alternative rather than the semantic validity. Both kinds of predicates alter the parse based upon information available at runtime. The difference is that syntactic predicates automatically examine future input symbols, whereas semantic predicates test arbitrary programmer-specified expressions.

You can view syntactic predicates as a special case of semantic predicates. Indeed, ANTLR implements syntactic predicates as special semantic predicates that invoke parser backtracking methods. The backtracking methods compare the grammar fragments found in syntactic predicates against the input stream. If a fragment matches, the syntactic predicate is true, and the associated alternative is considered valid.

Syntactic predicates and normal *LL(*)* lookahead are similar in that both support arbitrary lookahead. The difference lies in how much syntactic, structural awareness the two methods have of the input. *LL(*)* uses a DFA to examine the future input symbols, whereas syntactic predicates use a pushdown machine, a full context-free language parser (see Section 2.4, *Enforcing Sentence Tree Structure*, on page 40). In Section 2.2, *The Requirements for Generating Complex Language*, on page 38, you learned that DFAs are equivalent to regular expressions, and therefore, DFAs are too weak to recognize many common language constructs such as matched parentheses. For that, you need a recognizer capable of matching nested, tree-structured constructs (see Section 2.3, *The Tree Structure of Sentences*, on page 39). Syntactic predicates contain context-free grammars, which were designed specifically to deal with the tree-structured nature of sentences. The result of all this is that syntactic predicates can recognize more complicated sentential structures in the lookahead than DFAs. Consequently, syntactic predicates dramatically increase the recognition strength of *LL(*)* parsers. Better yet, backtracking is a simple and well-understood mechanism.

Syntactic predicates are useful in two situations:

- When *LL(*)* cannot handle the grammar the way you would like to write it

- When you must specify the precedence between two ambiguous alternatives; ambiguous alternatives can both match the same input sequence

This section shows three examples whose natural grammar is non-*LL(*)* and then provides an example from C++ where a syntactic predicate resolves an ambiguity between declarations and expression statements by specifying the precedence.

How to Resolve Non-*LL(*)* Constructs with Syntactic Predicates

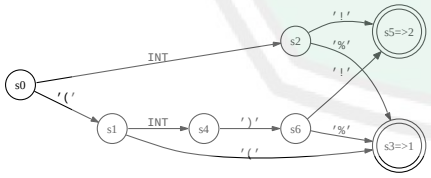
Let's start with a simple non-*LL(*)* example that we can easily resolve with a syntactic predicate. Consider the following natural grammar for matching expressions followed by two different operators: percent and factorial:

```
grammar x;

s : e '%'
  | e '!'
  ;

e : '(' e ')'
  | INT
  ;

INT : '0'..'9'+ ;
```

Rule **s** is non-*LL(*)* because the left prefix **e** is common to both alternatives. Rule **e** is recursive, rendering a decision in **s** non-*LL(*)*. Prediction DFAs do not have stacks and, therefore, cannot match recursive constructs such as nested parentheses. ANTLR reports this:

```
$ java org.antlr.Tool x.g
ANTLR Parser Generator  Version 3.0  1989-2007
x.g:3:5: [fatal] rule s has non-LL(*) decision due to recursive rule
invocations reachable from alts 1,2.  Resolve by left-factoring or
using syntactic predicates or using backtrack=true option.
warning(200): x.g:3:5: Decision can match input such as "'(' '('"
using multiple alternatives: 1, 2
As a result, alternative(s) 2 were disabled for that input
```

and builds a DFA:



DFAs vs. Backtracking in the Maze

The DFAs of *LL(*)* and the backtracking of syntactic predicates both provide arbitrary lookahead. In the maze, the difference lies in who is doing the lookahead. A DFA is analogous to a trained monkey who can race ahead of you, looking for a few symbols or simple sequences. When a trained monkey isn't smart enough, you must walk the alternative paths emanating from a fork yourself to figure out exactly what is down each path. You are smarter, but slower, than the agile trained monkey.

The recognizer can't match input such as ((x))!. ANTLR resolves nondeterministic input ((by choosing alternative one (via accepts state s3). Notice that the DFA knows how to handle one invocation of rule **e**, (INT), but cannot figure out what to do when **e** invokes itself.

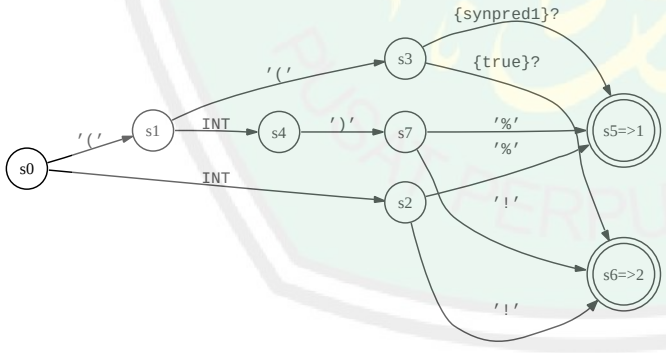
Rather than left-factor the grammar, making it less readable, like this:

```
s : e ('%'|'!')
  ;
```

we can use a syntactic predicate that explicitly tells ANTLR when to match alternative one:

```
s : (e '%')=> e '%'
  | e '!'
  ;
```

This says, "If e '%' matches next on the input stream, then alternative one will succeed; if not, try the next conflicting alternative." ANTLR generates the following prediction DFA:



Notice that now, upon ((, the DFA evaluates synpred1, which asks the parser to evaluate the syntactic predicate. You can think of the syntactic predicates as forking another parser that tries to match the grammar fragment in the predicate: e '%'.

We do not need to put a predicate on the second alternative because the parser always attempts the last conflicting alternative like an **else** clause. In the DFA, the **else** clause is represented by the {true}? predicate.

ANTLR also supports an auto-backtracking feature whereby ANTLR inserts syntactic predicates on the left edge of every alternative (see Section 14.4, *Auto-backtracking*, on page 340). The auto-backtracking does not cost anything extra because the *LL(*)* algorithm incorporates these predicates only when normal *LL(*)* grammar analysis fails to produce a deterministic prediction DFA. Using the auto-backtracking feature, we avoid cluttering the grammar with syntactic predicates:

```
grammar x;
options {backtrack=true;}
s : e '%'
  | e '!'
  ;
...
```

The example in this section epitomizes a common situation in real grammars where recursion renders a grammar non-*LL(*)*. The following sections provide more realistic examples from Java and C.

Java 1.5 For-Loop Specification

LL(*) cannot handle alternatives that reference recursive rules. For example, the following rules describe the enhanced "foreach" **for-loops** in Java 1.5:

```
// E.g., enhanced: "for (String n : names) {...}"
//       old style: "for (int i; i<10; i++) {...}"
stat: 'for' '(' forControl ')' statement
   ...
    ;

// E.g., "String n : names" or "int i; i<10; i++"
// non-LL(*) because both alternatives can start by matching rule type
forControl
    :   forVarControl
    |   forInit? ';' expression? ';' forUpdate?
    ;
```

```
forInit
    :   'final'? type variableDeclarators
    |   expressionList
    ;

forVarControl // new java 1.5 "foreach"
    :   'final'? annotation? type Identifier ':' expression
    ;

forUpdate
    :   expressionList
    ;
```

Rule **forControl** is non-*LL(*)* because rule **type** is reachable at the start of both **forVarControl** and **forInit**. This would not be a problem except that **type** is self-recursive because of generics, which can have nested type specifications such as List<List<int>>. For example, an *LL(*)* parser cannot decide which alternative to match after seeing this input:

```
for (List<List<int>> data = ...
```

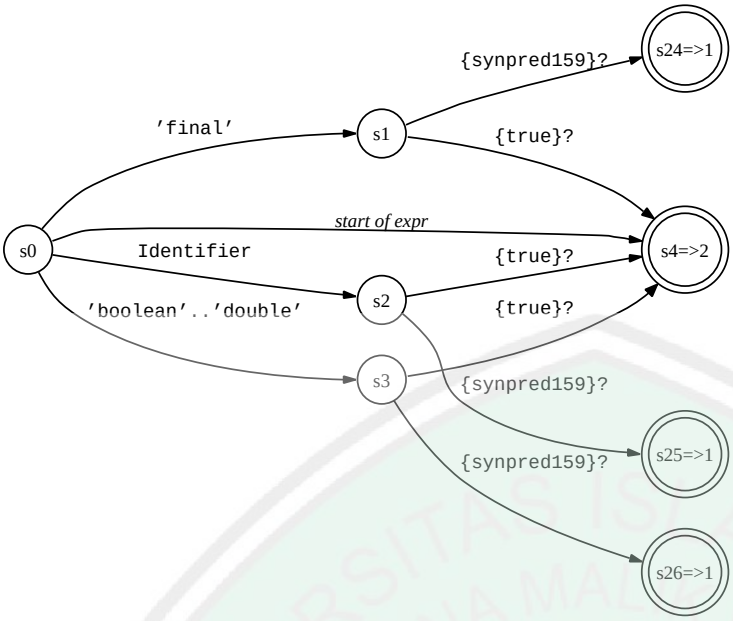
LL(*)'s DFAs cannot see past the recursive type structure.

Rewriting this grammar to be *LL(*)* might be possible, but it would mean extra work and a less readable grammar. Inserting a single syntactic predicate resolves the issue quickly and easily:

```
forControl
    :   (forVarControl)=> forVarControl
    |   forInit? ';' expression? ';' forUpdate?
    ;
```

To be clear, the alternatives have no valid sentence in common—it is just that *LL(*)* by itself is too weak to distinguish between the two alternatives as written. ANTLR scales back the analysis to *LL(1)* from *LL(*)* because it knows *LL(*)* will fail to yield a valid DFA. *LL(1)* plus a syntactic predicate is sufficient, however.

The DFA looks like this:



The DFA is not minimized, as you can see—a future version of ANTLR will optimize the generated DFA.

The DFA forces backtracking unless it is clear that the lookahead represents an expression. In that case, the parser knows that only the second alternative would match. Although pure *LL(*)* fails, you can turn on auto-backtracking mode and increase the fixed lookahead to *LL(3)* in order to help ANTLR optimize the decision:

```
forControl
options {k=3; backtracking=true;}
    :   forVarControl
    |   forInit? ';' expression? ';' forUpdate?
    ;
```

Now, the DFA will not backtrack upon input such as `Color c : colors` because it can see the ':' with three symbols of lookahead (the DFA is too large to effectively show here). The functionality is the same, but this particular decision is much faster for the common case. The increased fixed lookahead prevents backtracking. Manually specified syntactic predicates are always evaluated, but those implicitly added by auto-backtracking mode are not. Auto-backtracking syntactic predicates are evaluated only if *LL(*)* fails to predict an alternative.

The following section provides a similar situation from C where recursive constructs stymie *LL(*)*.

C Function Definition vs. Declaration

Section 11.2, *Why You Need LL(*)*, on page 264 motivated the need for *LL(*)* by showing a rule matching both abstract and concrete methods:

```
method
    : type ID '(' args ')' ';'          // E.g., "int f(int x,int y);"
    | type ID '(' args ')' '{' body '}' // E.g., "int f() {...}"
    ;
```

The prediction DFA easily saw past the left common prefix to the ';' or the '{' because neither **type** nor **args** is recursive. In C, however, arguments are recursive constructs and can be arbitrarily long because of nested parentheses. Here is a sample C function definition whose single argument, p, is a pointer to a function returning **int** that has a **float** argument:[2]

```
void f(int ((*p))(float)) { «body» }
```

The argument declarator can be arbitrarily nested, making it impossible for a DFA to recognize the argument list in order to see past it properly:

```
external_declaration
    : function_definition
    | declaration
    ;
```

Although you could simply turn on the auto-backtracking feature, that is unnecessarily inefficient because the parser will backtrack over the entire function body:

```
external_declaration
options {backtrack=true;}
    : function_definition // uses (function_definition)=> predicate
    | declaration
    ;
```

The implicitly created syntactic predicate for the first alternative references **function_definition**, which tells ANTLR to try to match the entire function. A more efficient approach is to use a manually specified syntactic predicate that provides the minimum necessary lookahead to distinguish the first alternative from the second:

```
external_declaration
    : ( declaration_specifiers? declarator declaration* '{' )=>
        function_definition
    | declaration
    ;
```

2. See http://www.cs.usfca.edu/~parrt/course/652/lectures/cdecls.html for a complete description of how to easily read any C declaration.

With this predicate, the backtracking mechanism stops after seeing the '{' instead of parsing the entire function body. Another way to make the decision efficient, without the manually specified syntactic predicate, is to simply reorder the alternatives:

```
external_declaration
options {backtrack=true;}
    : declaration
    | function_definition
    ;
```

Now, even with the auto-backtracking, the parser will stop backtracking much sooner. The backtracking stops at the ';' in **declaration** (resulting in success) or the '{' in **function_definition** (resulting in failure). This reordering works because the two alternatives do not have any valid sentences in common.

In practice, ANTLR generates a very large (nonoptimized) DFA for this decision even with the backtracking; turning on *k=1* is a good idea to reduce its size:

```
external_declaration
options {backtrack=true; k=1;}
    : declaration
    | function_definition
    ;
```

The previous three examples illustrated how to use syntactic predicates to resolve grammar nondeterminisms arising from weaknesses in *LL(*)*'s DFA-based lookahead. The next sections examine the second use of syntactic predicates: resolving ambiguous alternatives. The examples show constructs that result in grammars where two alternatives can match the same input. The solutions use syntactic predicates to order the ambiguous alternatives, giving precedence to the alternative with the proper interpretation.

Resolving the If-Then-Else Ambiguity

Recall from Section 11.5, *Nondeterministic Decisions*, on page 281 that all grammar ambiguities lead to parser nondeterminisms, meaning that ambiguous decisions are not *LL(*)* and result in ANTLR warnings. Sometimes, however, the syntax of the language makes a single sentence consistent with two different interpretations. The language reference manual specifies which interpretation to use, but pure context-free grammars have no way to encode the precedence. The most famous example is the **if-then-else** ambiguity:

```
stat:   'if' expr 'then' stat ('else' stat)?
    |   ID '=' expr ';'
    ;
```

ANTLR reports that the grammar has two paths that match the **else** clause. The parser can enter the ('else' stat)? subrule or bypassing the subrule to match the **else** clause to a previous **if** statement. ANTLR resolves the conflict correctly by choosing to match the **else** clause immediately, but you still get an analysis warning. To hush the warning, you can specify a syntactic predicate or turn on auto-backtracking. The warning goes away because syntactic predicates specify the precedence of the two alternatives. Simply put, the alternative that matches first wins. The following rewrite of rule **stat** is not ambiguous because it indicates that the parser should match the **else** clause immediately if present:

```
stat
options {backtrack=true;}
    :   'if' expr 'then' stat 'else' stat
    |   'if' expr 'then' stat
    |   ID '=' expr ';'
    ;
```

The only problem is that the decision is now much less efficient because of the backtracking. This example merely demonstrates how syntactic predicates resolve true ambiguities by imposing order on alternatives. For this situation, do not use syntactic predicates; let ANTLR resolve the nondeterminism because it does the right thing with the *k=1* lookahead. The next section provides an example from C++ whose solution absolutely requires a syntactic predicate.

Distinguishing C++ Declarations from Expressions

Some C++ expressions are valid statements such as `x;` or `f();`. Unfortunately, some expressions require arbitrary lookahead to distinguish from declarations. Quoting from Ellis and Stroustrup's *The Annotated C++ Reference Manual* [ES90], "There is an ambiguity in the grammar involving expression-statements and declarations. . . The general cases cannot be resolved without backtracking. . . In particular, the lookahead needed to disambiguate this case is not limited." The authors use the following examples to make their point, where `T` represents a type:

```
T(*a)->m=7; // expression statement with type cast to T
T(*a)(int); // a is a pointer to function returning T with int argument
```

These statements illustrate that expression statements are not distinguishable from declarations without seeing all or most of the statement. For example, in the previous expression statement, the '->' symbol is the first indication that it is a statement. Syntactic predicates resolve this nondeterminism by simply backtracking until the parser finds a match.

It turns out that the situation in C++ gets much worse. Some sentences can be both expressions and declarations, a true language syntactic ambiguity. For example, in the following C++ code, I(x) is both a declaration (x is an integer as in I x;) and an expression (cast x to type I as in (I)x):[3]

```
typedef int I;
char x = 'a';
void foo() {
    I(x); // read as "I x;" not "(I)x;" (hides global char x)
}
```

The C++ language definition resolves the ambiguity by saying you should choose declaration over expression when a sentence is consistent with both. To paraphrase Ellis and Stroustrup further, in a parser with backtracking, the disambiguating rule can be stated simply as follows:

1. If it looks like a declaration, it is.
2. Otherwise, if it looks like an expression, it is.
3. Otherwise, it is a syntax error.

There is no way to encode these rules in a context-free grammar because there is no notion of order between alternatives. Syntactic predicates, on the other hand, implicitly order alternatives. They provide an exact formal means of encoding the precedence dictated by the C++ language reference:

```
stat: (declaration)=> declaration // if looks like declaration, it is
    | expression                  // else its expression
    ;
```

The beauty of this solution is that a syntactic predicates handles both cases: when the parser needs arbitrary lookahead to distinguish declarations from expressions and when it needs to disambiguate sentences that are both declarations and expressions.

3. *T*(x) for type *T* can be a constructor-style typecast in C++.

In general, semantic and syntactic predicates overcome the weaknesses of pure context-free grammars. Predicates allow you to formally encode the constraints and rules described in English in a language reference manual using an ANTLR grammar. Parser generators without predicates force the use of *ad hoc* hacks, tweaks, and tricks. For example, a common trick is to insert a smart token filter between the lexer and parser that flips token types when necessary.

This chapter informally defined syntactic predicates, showed how to use them, and demonstrated their power. This information will get you started building predicated grammars, but ultimately you will need to understand how ANTLR implements these predicates more precisely. The next two chapters explain the important details about semantic and syntactic predicates.

Chapter 13

Semantic Predicates

Translators map input sentences to output sentences, which means that the translator must establish a unique interpretation for each input sentence. Some languages, unfortunately, have ambiguous phrases whose syntax allows more than a single interpretation (Section 2.5, *Ambiguous Languages*, on page 43). The proper interpretation depends on the phrase's context. In C++, for example, `T(i);` syntactically looks like a function call and a constructor-style typecast. The proper interpretation depends on what kind of thing `T` is, which in turn depends on how the input defines `T` elsewhere. Pure context-free grammars are unable to impose such conditions on rules in order to uniquely interpret that phrase. ANTLR gets around this problem by augmenting context-free grammars with *semantic predicates* that can alter the parse based upon context.

Semantic predicates are boolean expressions you can use to specify the semantic validity of an alternative. The term *predicate* simply means conditional, and the term *semantic* implies you are talking about arbitrary boolean expressions rather than a syntactic condition. In practice, the rule is pretty simple: if a predicate's expression is false, the associated alternative is invalid. Because predicates can ask questions about other input phrases, you can encode the context in which alternatives apply. For example, in Section 12.1, *C++ Typecast vs. Method Call*, on page 304, you saw how to use semantic predicates to distinguish between C++ method calls and constructor-style typecasts.

Semantic predicates have been in the literature since the 1970s, but Russell Quong and I extended the functionality to include hoisting, whereby you can incorporate a predicate in one rule into the prediction decision of another rule.

Semantic predicates are available to any ANTLR grammar and have three variations:

- Disambiguating semantic predicates, which disambiguate syntactically identical statements

- Gated semantic predicates, which dynamically turn on and off portions of a grammar
- Validating semantic predicates, which throw a recognition exception if the predicate fails

This chapter describes the functional details and limitations of semantic predicates, whereas the previous chapter illustrated how to use semantic predicates. Let's begin with the most important kind of semantic predicate: disambiguating semantic predicates that can resolve nondeterministic *LL(*)* decisions.

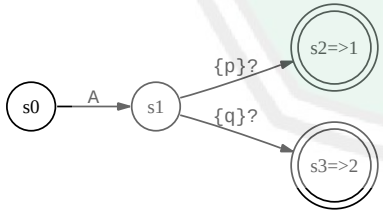
13.1 Resolving Non-*LL(*)* Conflicts with Disambiguating Semantic Predicates

Upon finding an *LL(*)* nondeterminism, ANTLR typically emits a grammar analysis warning. In the presence of semantic predicates, however, ANTLR tries to *hoist* them into the alternative prediction for that decision to resolve the conflict. More specifically, ANTLR hoists only those semantic predicates that are reachable from the left edge without consuming an input symbol. When a prediction DFA evaluates a semantic predicate, that predicate is called a *disambiguating semantic predicate*. For efficiency reasons, ANTLR hoists predicates into DFAs only when *LL(*)* lookahead alone is insufficient to distinguish alternatives.

For semantic predicates to fully resolve a nondeterminism, you must *cover* all alternatives that contribute to the nondeterminism, as shown in the following grammar:

```
a : {p}? A
  | {q}? A
  ;
```

Predicates implicitly specify the precedence of the conflicting alternatives. Those predicated alternatives specified earlier have precedence over predicated alternatives specified later. Generally speaking, parsers evaluate semantic predicates in the order specified among the alternatives. For this grammar, ANTLR generates the following DFA:



If **A** is the next symbol of lookahead and p evaluates to true, the DFA predicts alternative one; otherwise, if **A** and q, the DFA predicts alternative two. The following Java implementation clarifies the functionality:

```
int alt1=2;
int LA1_0 = input.LA(1);
if ( (LA1_0==A) ) {
    if ( (p) ) { alt1=1; }
    else if ( (q) ) { alt1=2; }
    else {
        NoViableAltException nvae =
            new NoViableAltException(
            "2:1: a : ({...}? A | {...}? A );", 1, 1, input);
        throw nvae;
    }
}
```

ANTLR can also hoist a predicate out of its original rule into the prediction decision for another rule. The following grammar is equivalent to the previous version in that ANTLR generates the same prediction DFA for rule **a**:

```
// ANTLR hoists {p}? and {q}? into a's prediction decision
a : b
  | c
  ;
b : {p}? A ;
c : {q}? A ;
```

This nonlocal hoisting allows you to specify the semantics and syntax for language constructs together in the same place. Here is a commonly used rule that indicates when an identifier is a type name:

```
// typename when lookahead token is ID and isType says text is a type
typename : {isType(input.LT(1))}? ID ;
```

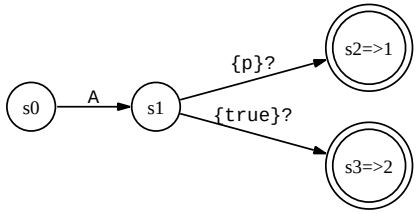
The next section explains what happens if you leave off one of the predicates.

Alternatives That ANTLR Implicitly Covers

As a convenience, you can cover just *n-1* alternatives with predicates for *n* conflicting alternatives. The following rule is perfectly fine even though it has two alternatives and just one predicate:

```
a : {p}? A
  | A
  ;
```

ANTLR implicitly covers this special case of an uncovered final conflicting alternative with {true}? as sort of an **else** clause. ANTLR computes a DFA with one real predicate, p, and one implied predicate, true:



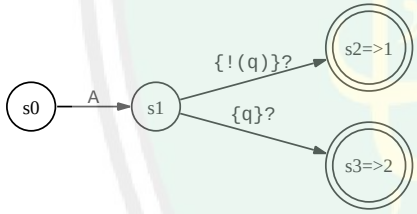
lThis feature also automatically covers the exit branch of a looping subrule, as shown in the following rule:

```
loop: ( {p1}? A )+ A
    ;
```

Upon lookahead **A** and p, the rule stays in the loop. Because of the implied order between conflicting alternatives, ANTLR cannot add {true}? unless the uncovered alternatives is last. In general, ANTLR must assume that the n^{th} predicate is the complement of the union of the other *n-1* predicates. Consider the same grammar as before, but with the predicate on the second alternative instead of the first:

```
a : A
  | {q}? A
  ;
```

ANTLR generates the following DFA:



If ANTLR assumed predicate true implicitly covered the first alternative, the second alternative covered with q would be unreachable because the first predicate would always win.

If you fail to cover all nondeterministic alternatives implicitly or explicitly, ANTLR will give you a nondeterminism warning. This rule:

```
a : {p}? A
  | A
  | A
  ;
```

results in the following warning:

```
warning(200): t.g:2:5: Decision can match input such as "A"
using multiple alternatives: 1, 2, 3
As a result, alternative(s) 3,2 were disabled for that input
warning(201): t.g:2:5: The following alternatives are unreachable: 2,3
```

Supplying a semantic predicate for all conflicting alternatives is not always sufficient to resolve nondeterminisms because ANTLR can hoist predicates out of their original rules. The next section illustrates a situation where a single predicate does not sufficiently cover a nondeterministic lookahead sequence.

Insufficiently Covered Nondeterministic Lookahead Sequences

ANTLR assumes you know what you're doing when you specify semantic predicates that disambiguate alternatives, but it does its best to identify situations where you have not given a complete solution. ANTLR will warn you if you have insufficiently covered an alternative rather than simply forgotten to add one.[1] In the following grammar, lookahead token **A** predicts both alternatives of rule **a**:

```
grammar T;
a : b
  | A        // implicitly covered with !predicates from first alternative
  ;
b : {q}? A
  | A        // alternative not covered
  ;
```

The pure *LL(*)* prediction DFA for **a** is, therefore, nondeterministic because **A** labels multiple transitions (predicts more than one alternative). ANTLR looks for semantic predicates to resolve the conflict. It hoists {q}? from **b** into the prediction DFA for **a**, but a problem still exists. ANTLR warns the following:

```
warning(203): t.g:3:5: The following alternatives are insufficiently
covered with predicates: 2
warning(200): t.g:3:5: Decision can match input such as "A"
using multiple alternatives: 1, 2
As a result, alternative(s) 2 were disabled for that input
warning(201): t.g:3:5: The following alternatives are unreachable: 2
```

1. Paul Lucas brought the issue of insufficiently covered alternatives to my attention at the ANTLR2004 workshop at the University of San Francisco.

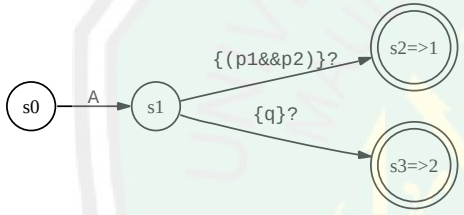
ANTLR notices that not every path to an **A** token reachable from the first alternative of **a** is covered by a predicate—only the first alternative of **b** has a predicate. As a result, ANTLR warns you about the uncovered nondeterministic lookahead sequence. Note that the lookahead decision for rule **b** is OK because it has one predicate for two **A** alternatives. Without this "flow analysis," subtle and difficult-to-find bugs would appear in your grammars. The next section shows how ANTLR combines predicates when it does find sufficient predicates to cover a lookahead sequence.

Combining Multiple Predicates

When ANTLR finds more than one predicate reachable from a decision left edge, it combines them with the && and || operators to preserve the semantics. For example, in the following grammar, ANTLR can see two predicates for the first alternative, `p1` and `p2`:

```
a : {p1}? {p2}? A
  | {q}? A
  ;
```

The grammar results in the following prediction DFA:

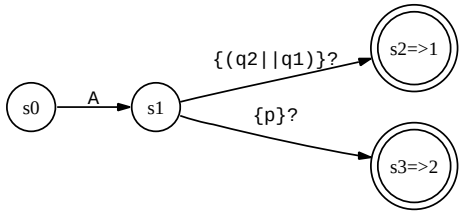


ANTLR combines sequences of predicates with the && operator. Looking at the grammar, it is clear that the semantic validity of applying the first alternative is `p1&&p2`. Both predicates must be true for the alternative to be semantically valid.

When combining multiple predicates taken from different alternatives, however, ANTLR combines the alternative predicates with the || operator. Consider the following grammar where rule **a**'s decision is nondeterministic upon token **A**:

```
a : b
  | {p}? A
  ;
b : {q1}? A
  | {q2}? A
  ;
```

ANTLR creates the following DFA:

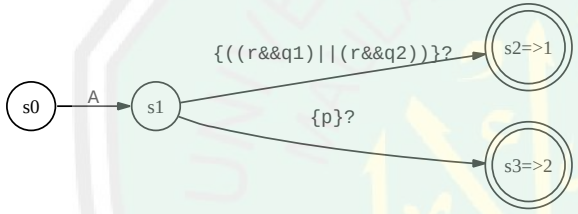


Rule **b** can match token **A** in either of two semantic contexts: q1 or q2; hence, those predicates must be ||'d together to specify the semantic applicability of the first alternative of rule **a**.

Naturally, combinations also work:

```
a : {r}? b
  | {p}? A
  ;
b : {q1}? A
  | {q2}? A
  ;
```

This results in the following DFA:



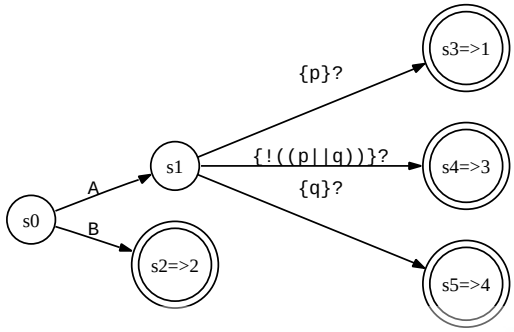
The DFA says that rule **a** can match an **A** via the first alternative if r and q1 are true or r and q2 are true. Otherwise, rule **a** can match **A** via the second alternative if p.

Decisions with Both Deterministic and Nondeterministic Alternatives

ANTLR properly handles the case where a subset of the alternatives are nondeterministic and when there are multiple conflicting alternatives. The following grammar has three alternatives that match **A**, but there are sufficient predicates to resolve the conflict:

```
a : {p}? A
  | B
  | A
  | {q}? A
  ;
```

ANTLR generates the following prediction DFA:



The important observation is that, upon input **B**, the DFA immediately predicts the second alternative without evaluating a semantic predicate. Upon token **A**, however, the DFA must evaluate the semantic predicates in the order specified in the grammar. The second alternative is semantically valid when the other **A** alternatives are invalid: `!p` and `!q` or equivalently `!(p||q)`.

The final issue related to disambiguating semantic predicates is that DFAs must evaluate predicates within their syntactic context.

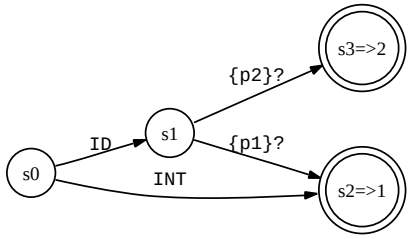
Evaluating Predicates in the Proper Syntactic Context

When ANTLR hoists semantic predicates into prediction DFAs, it must carry along the syntactic context in which it found the predicates. Consider the following grammar. ANTLR resolves rule **a**'s nondeterministic prediction DFA with the predicates `p1` and `p2`.

```
a : b
  | {p2}? ID
  ;
b : {p1}? ID // evaluate only upon ID not INT
  | INT
  ;
```

A predicate covers every path reaching an **ID** reference. But what about **INT**? Both **INT** and **ID** syntactically predict the first alternative of rule **a**. It turns out that it is important for the DFA to avoid evaluating `p1` upon **INT**. For example, `p1` might look up the token in the symbol table, which makes no sense for **INT**; worse, looking it up might cause an exception.

ANTLR generates the following DFA:



Token **INT** immediately predicts alternative one without using a predicate. It is only after seeing **ID** in s1 that the DFA evaluates the predicates.

As you have seen, ANTLR hoists disambiguating semantic predicates only when the *LL(*)* lookahead is insufficient to distinguish between alternatives. Sometimes, however, you always want ANTLR to hoist a predicate into the decision DFA, as explained in the next section.

13.2 Turning Rules On and Off Dynamically with Gated Semantic Predicates

Sometimes you want to distinguish between alternatives that are not syntactically ambiguous. For example, you might want to turn off some language features dynamically such as the Java **assert** keyword or GCC C extensions. This requires a semantic predicate to turn off an alternative even though the enclosing decision is deterministic. But, disambiguating semantic predicates are not hoisted into deterministic decisions. ANTLR introduces a new kind of predicate called a *gated semantic predicate* that is always hoisted into the decision. Gated semantic predicates use the syntax {*pred*}?=>, as the following grammar demonstrates:

```
stat: 'if' ...
    | {allowAssert}?=> 'assert' expr
   ...
    ;
```

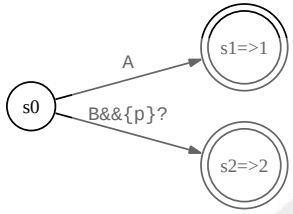
To see the difference between disambiguating semantic predicates and gated semantic predicates, contrast the following grammar:

```
a : A
  | {p}? B
  ;
```

with the following grammar:

```
a : A
  | {p}?=> B
  ;
```

In the first version, with the disambiguating semantic predicate, ANTLR ignores the predicate because syntax alone is sufficient to predict alternatives. In the second version, however, the gated semantic predicate is included in rule **a**'s decision:

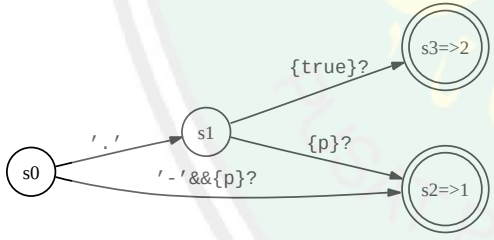


The DFA predicts alternative two only when p is true. The transition from s0 to s2 effectively disappears when p is false; the only viable alternative would be the first one.

In general, gated predicates appear along the transitions along all paths leading to the accept state predicting the associated gated alternative. For example, the following grammar can match input .. either by matching the first alternative of rule **a** once or by matching the second alternative twice:

```
a : {p}?=> ('.'|'-')+
  | '.'
  ;
```

The grammar distinguishes between the two cases using the p gated semantic predicate. The following DFA does the appropriate prediction:



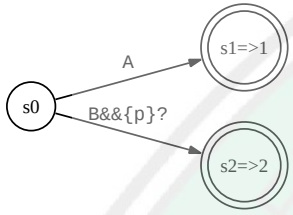
The DFA guards access to accept state s2, predicting alternative one, with gated predicate p. But, notice that the DFA doesn't test p on the

transition from states s0 to s1. That transition is common to both alternatives one and two. When p is false, the DFA must still be able to match . to alternative two.

Remember that ANTLR hoists predicates out of one rule to use in the decision for another rule. In this respect, gated predicates behave just like disambiguating semantic predicates. In the following grammar, the predicate is hoisted from rule **b** into the prediction decision for rule **a**:

```
a : A
  | b // as if you had used {p}?=> here too
  ;
b : {p}?=> B
  ;
```

ANTLR generates the following DFA:



Even though syntax alone yields a deterministic prediction decision, rule **a**'s DFA uses the gated semantic predicates hoisted from **b**.

The next section describes the third and final predicate variation.

13.3 Verifying Semantic Conditions with Validating Semantic Predicates

Although most semantic analysis occurs in a separate phase for complicated language applications, sometimes it is convenient to place semantic checks within a grammar that throw an exception upon failure like syntax errors do. For example, in the following grammar, the recognizer throws a FailedPredicateException if the input program references a variable without a prior definition where the highlighted region is the code generated for the validating semantic predicate:

```
grammar t;
expr:   INT
    |   ID {isDefined($ID.text)}?
    ;
```

Such a predicate is called a *validating semantic predicate*, and ANTLR generates the following code for rule **expr**:

```
public void expr() throws RecognitionException {
    Token ID1=null;
    try {
        «alternative-prediction-code»
    switch (alt1) {
        case 1 :
        // t.g:2:9: INT
        match(input,INT,FOLLOW_INT_in_expr10);
        break;
        case 2 :
        // t.g:3:7: ID {...}?
        ID1=(Token)input.LT(1);
        match(input,ID,FOLLOW_ID_in_expr18);
        if ( !(isDefined(ID1.getText())) ) {
            throw new FailedPredicateException(input,
                    "expr", "isDefined($ID.text)");
        }
        break;
    }
    }
    catch (RecognitionException re) {
    reportError(re);
    recover(input,re);
    }
}
```

All semantic predicates result in such predicate validation code regardless of how else ANTLR uses the predicates.

Validating semantic predicates do not alter the decision-making process—they throw exceptions after the recognizer sees an erroneous statement. The true power of semantic predicates, however, is their ability to alter the parse upon runtime information. The next section examines such disambiguating semantic predicates in detail.

At this point, you have all the details about how ANTLR uses semantic predicates, but you need to know about some limitations imposed on predicate expressions.

13.4 Limitations on Semantic Predicate Expressions

Semantic predicates must be free of side effects in that repeated evaluations must return the same result and not affect other predicates. Further, the order in which DFAs evaluate predicates must not matter

within the same decision. Alternatives specified first still have priority over subsequent alternatives, but the DFA must be able to evaluate predicates in any order. Here is an example grammar where the predicate is not free of side effects:

```
a
@init {int i=0;}
 : {i++==0}? A  // BAD! side effects in predicate
 | A
 ;
```

ANTLR generates the following code:

```
if ( (LA1_0==A) ) {
    if ( (i++==0) ) {  // increments i here
        alt1=1;
    }
    // i is now 1
    else if ( (true) ) { alt1=2; }
    else «error»;
}
else «error»;
switch (alt1) {
    case 1 :
        // t.g:5:4: {...}? A
        // i is not 0 anymore so exception will be thrown
        if ( !(i++==0) ) { // tests i (and increments) again here
            throw new FailedPredicateException(input, "a", "i++==0");
        }
        match(input,A,FOLLOW_A_in_a18);
        break;
    «alternative-two»
}
```

The highlighted lines derive from the predicate. The first time the DFA evaluates the predicate `i` is 0, so it predicts alternative one. Upon reaching the code for the first alternative, the recognizer evaluates the predicate again as if it were a simple validating predicate. But `i` is 1, and the validating predicate fails.

Another limitation on semantic predicates is that they should not reference local variables or parameters. In general, predicates should not reference anything not visible to all rules just in case they are hoisted out of one rule into another's prediction DFA. If you are positive that the predicate will not be hoisted out of the rule, you can use a parameter.

For example, here is a rule that alters prediction as a function of its parameter:

```
/** Do not allow concrete methods (methods with bodies) unless
 *  parameter allowConcrete is true.
 */
method[boolean allowConcrete]
    :   {allowConcrete}?=> methodHead body
    |   methodHead ';'
    ;
```

If ANTLR hoists that predicate out of rule **method**, however, the target compiler will complain about undefined references. Technically, this is a limitation of the target language, not a limitation of semantic predicates or ANTLR. For example, you can usually use attributes or dynamic scopes to overcome limitations related to using parameters and semantic predicates.

Semantic predicates are a powerful means of recognizing context-sensitive language structures by allowing runtime information to drive recognition. But, they are also useful as an implementation vehicle for syntactic predicates, the subject of the next chapter. As we'll see, the fact that ANTLR hoists semantic predicates into decisions only when *LL(*)* fails automatically minimizes how often the recognizer needs to backtrack.

Chapter 14

Syntactic Predicates

Syntactic predicates have been available since ANTLR v1.0 (since the early 1990s) and are certainly one of the big reasons why ANTLR became popular. Russell Quong and I invented the term and mechanism.

A *syntactic predicate* specifies the syntactic validity of applying an alternative just like a semantic predicate specifies the semantic validity of applying an alternative. Syntactic predicates, as we've seen, are parenthesized grammar fragments followed by the => operator. If a syntactic predicate matches, the associated alternative is valid. Syntactic predicates are a simple way to dramatically improve the recognition strength of any *LL*-based recognizer by providing arbitrary lookahead. In practice, this means syntactic predicates let you use write grammars that ANTLR would otherwise reject. For example, Section 12.2, *Resolving Ambiguities and Nondeterminisms*, on page 306 illustrated difficult-to-parse language constructs from Java, C, and C++ that resolve nicely with syntactic predicates.

Syntactic predicates also let you specify the precedence between two or more ambiguous alternatives. If two alternatives can match the same input, ANTLR ordinarily emits a grammar analysis warning. By adding a syntactic predicate, you force the generated recognizer to try the alternatives in order. ANTLR resolves the ambiguity in favor of the first alternative whose predicate matches. Such precedence resolves the ambiguity, and ANTLR does not emit a warning. For example, Section 12.2, *Resolving the If-Then-Else Ambiguity*, on page 313 illustrated how to hush the warning from ANTLR stemming from the classic **if-then-else** ambiguity.

Chapter 12, *Using Semantic and Syntactic Predicates*, on page 292 delineated a number of examples that illustrated how to use syntactic predicates and demonstrated their power. This chapter focuses on the details of their implementation and other information necessary to fully understand syntactic predicates. In particular, we'll see the following:

- ANTLR implements syntactic predicates using semantic predicates.
- Syntactic predicates force the parser to backtrack.
- ANTLRWorks has a number of visualizations that can help you understand backtracking parsers.
- Parsers do not execute actions during the evaluation of syntactic predicates to avoid having to undo them during backtracking.
- Auto-backtracking is a great rapid prototyping mode that automatically inserts a syntactic predicate on the left edge of every alternative.
- Memoization is a form of dynamic programming that squirrels away partial parsing results and guarantees linear parsing complexity.
- Syntactic predicates can hide true ambiguities in grammars.
- Backtracking does not generally affect the use of embedded grammar actions.

Let's begin with a discussion of how ANTLR incorporates syntactic predicates into the parsing decision-making process.

14.1 How ANTLR Implements Syntactic Predicates

For each syntactic predicate, ANTLR defines a special method[1] that returns true or false depending on whether the predicate's grammar fragment matches the next input symbols. In this way, ANTLR can implement syntactic predicates as semantic predicates that invoke special boolean recognition methods. Consider the following simple non-*LL(*)* grammar that requires backtracking because of the two ambiguous alternatives. As with semantic predicates, you do not have to predicate the final nondeterministic alternative.

```
a : (A)=>A {System.out.println("ok");}
  | A      {System.out.println("this can never be printed");}
  ;
```

ANTLR rephrases the problem in terms of gated semantic predicates by translating the grammar to the following equivalent grammar:

```
a : {input.LA(1)==A}?=> A {System.out.println("ok");}
  | A {System.out.println("this can never be printed");}
  ;
```

1. Unfortunately, because of a limitation in Java, the bookkeeping machinery cannot be generalized with any kind of efficiency, resulting in code bloat.

The "this can never be printed" Error

The "this can never be printed" error message (in the first grammar example of Section 14.1, *How ANTLR Implements Syntactic Predicates*, on the previous page) came out of some Unix program I was using 20 years ago, but I can't remember which one. It cracked me up.

My favorite compiler of all time was the Apple MPW C compiler. If I remember correctly, it gave error messages such as the following:

"Don't you want a ')' to go with that '('?"

When it got really confused, it would say this:

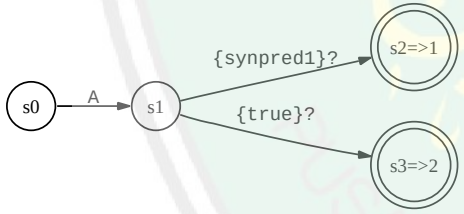
"That ';' came as a complete surprise to me at this point your program."

Pure genius.

To be precise, manually specified syntactic predicates become gated semantic predicates. These syntactic predicates are always evaluated just like gated semantic predicates.

On the other hand, syntactic predicates implicitly added by auto-backtracking mode become regular semantic predicates. Such predicates are evaluated only when *LL(*)* fails to predict an alternative.

The prediction DFA for rule **a** with the gated semantic predicate is as follows:



where `synpred1()` is the method that ANTLR generates to evaluate the syntactic predicate. The DFA indicates that, after seeing **A**, the syntactic predicate chooses between alternative one (s2) and two (s3).

The following straightforward Java code implements the DFA:

```
int alt1=2;
int LA1_0 = input.LA(1); // get first symbol of lookahead
if ( (LA1_0==A) ) {
    if ( (synpred1()) ) { alt1=1; } // test syntactic predicate
    else if ( (true) ) { alt1=2; }  // default to alternative 2
    else {
        if (backtracking>0) {failed=true; return ;}
        «throw-NoViableAltException» // throw only if not backtracking
    }
}
else {
    // no A found; error
    if (backtracking>0) {failed=true; return ;}
    «throw-NoViableAltException» // throw only if not backtracking
}
```

The highlighted sections specify what to do if the prediction code does not find a match. The code indicates that the recognizer should throw an exception unless it is backtracking. If backtracking, the recognizer sets a failed instance variable and returns, which is much faster than throwing an exception in all target languages.[2] ANTLR incorporates that prediction decision code into the method that it generates for rule `a`:

Improved in v3.

```
public void a() throws RecognitionException {
    «prediction-code»
    try {
        switch (alt1) {
            case 1 :
                match(input,A,FOLLOW_A_in_a14); if (failed) return ;
                if ( backtracking==0 ) {
                  System.out.println("ok");
                }
                break;
            case 2 :
                match(input,A,FOLLOW_A_in_a22); if (failed) return ;
                if ( backtracking==0 ) {
                  System.out.println("this can never be printed");
                }
                break;
        }
    }
    catch (RecognitionException re) {
        reportError(re);
        recover(input,re);
    }
}
```

2. According to the NetBeans C/C++ Development Pack team (http://www.netbeans.org), removing exceptions from backtracking in ANTLR v2 sped up recognition rates by about

The highlighted chunks illustrate how ANTLR generates code for the actions embedded in a grammar. Actions are not executed if the recognizer is backtracking because there is no general mechanism to undo actions (for example, how would you undo a print statement?).

If a syntactic predicate matches for an alternative, the recognizer rewinds the input and matches the alternative again "with feeling." The recognizer rewinds and reparses so that it can execute any actions embedded in the alternative. In this way, you do not really have to worry about how backtracking affects your actions at the cost of some time efficiency.

To avoid using exceptions during backtracking, ANTLR generates extra code to test for recognition failure. For example, you will see code such as the following after every call to a rule's method and after every token match:

```
if (failed) return;
```

In terms of the predicate, ANTLR generates a method that matches the predicate grammar fragment:

```
public void synpred1_fragment() throws RecognitionException {
    match(input,A,FOLLOW_A_in_synpred111); if (failed) return ;
}
```

It also generates method synpred1(), which the semantic predicate invokes. Method synpred1() increases the backtracking level, tests the predicate, rewinds the input, and decreases the backtracking level. backtracking is 0 upon entry to synpred1() unless the recognizer is already backtracking when it enters rule a.

```
public boolean synpred1() {
    backtracking++;
    int start = input.mark();
    try {
        synpred1_fragment(); // can never throw exception
    } catch (RecognitionException re) {
        System.err.println("impossible: "+re);
    }
    boolean success = !failed;
    input.rewind(start);
    backtracking--;
    failed=false;
    return success;
}
```

By examining the code ANTLR generates, you'll understand precisely how ANTLR implements syntactic predicates. In the next section, you'll see how ANTLRWorks makes syntactic predicates easier to understand by visualizing parse trees and prediction DFAs.

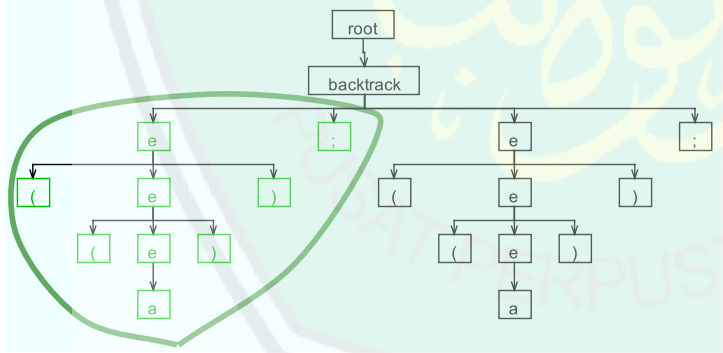
14.2 Using ANTLRWorks to Understand Syntactic Predicates

ANTLRWorks provides an excellent means to become familiar with the backtracking process. For example, in the following grammar, rule **backtrack** is non-*LL(*)*. It has three alternatives, each requiring arbitrary lookahead to see past the recursive structure in rules **e** and **cast** to the symbols beyond.

Download synpred/b.g

```
grammar b;
backtrack
    :   (cast ';')=> cast ';'
    |   (e ';')=>    e ';'
    |                e '.'
    ;

cast:   '(' ID ')' ;

e   :   '(' e ')'
    |   ID
        ;

ID  :   'a'..'z'+ ;
```

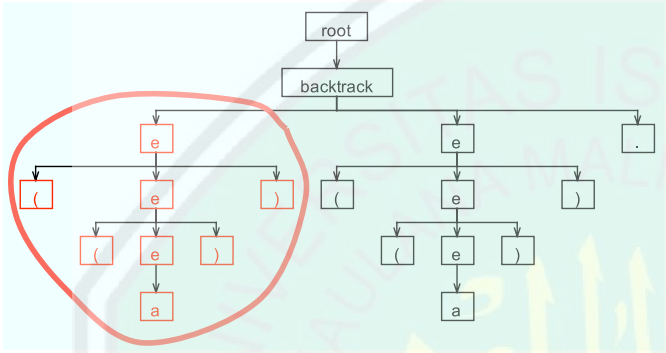
Rule **backtrack** matches input ((a)); with the second alternative; hence, ANTLRWorks shows the parse tree[3] for both the syntactic predicate and the actual match for the alternative:



3. The lassos look like a four-year-old drew them on purpose—mainly I like the way they look, but also their roundness makes it easy for you to distinguish them from the rigid,

The first two subtrees of rule **backtrack** (the lassoed region) represent the successful parse of ((a)); by the syntactic predicate (e ';')=> in the second alternative. Note that the syntactic predicate in the first alternative is avoided altogether because ((at the start of the input cannot begin a **cast**. The prediction DFA shown in Figure 14.1, on the following page, illustrates this. The only input that routes through the DFA to the syntactic predicate for alternative one (at s8) is (a);.

For input ((a))., however, the syntactic predicate in the second alternative fails. The trailing period does not match, and the recognizer matches the third alternative instead. The successful parse yields a parse tree with one (failed) predicate evaluation, followed by the actual parse tree built from matching alternative three of rule **backtrack**:



The parser does not need to backtrack in order to choose between alternatives two and three once the first predicate fails (input ((does not match predicate cast ';'. State s4 in the prediction DFA shown in Figure 14.1, on the next page illustrates that if the second syntactic predicates fails, the DFA can immediately predict alternative three by traversing s2 to s6.

Using ANTLRWorks to visualize decision DFAs and parse trees helps even more when your recognizer needs to start backtracking when it is already backtracking, the subject of the next section.

14.3 Nested Backtracking

While attempting one syntactic predicate, the recognizer might encounter another decision that requires it to backtrack. In this case, the recognizer enters a nested backtracking situation. There are no implementation problems, but nested backtracking can be the source of some confusion (and is the source of the worst-case exponential time

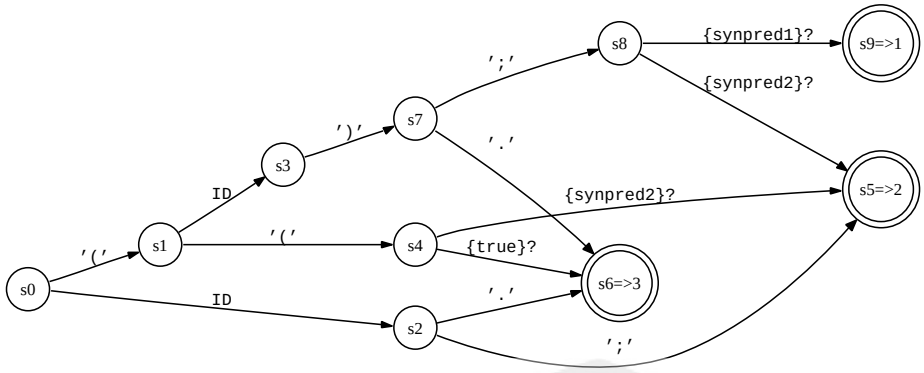


Figure 14.1: The prediction DFA for rule **backtrack** illustrating when the recognizer must evaluate syntactic predicates

complexity). The current nesting level is stored in instance variable backtracking. Consider the following grammar that matches an expression followed by either ';' or '.'. An expression, in turn, is either a typecast, a conventional nested expression, or a simple identifier.

Download synpred/nested.g

```
grammar nested;

a : (e ';')=> e ';'
  | e '.'
  ;

e : ('(' e ')' e)=> '(' e ')' e  // type cast
  | '(' e ')'            // nested expression
  | ID
  ;

ID      :       'a'..'z'+ ;
```

Input ((x))y; forces the recognizer to backtrack. Let's examine how ANTLRWorks visualizes the backtracking. When the recognizer completes the predicate in the first alternative of rule **a** and is about to match the first alternative for real, ANTLRWorks shows the partial parse tree, as shown in Figure 14.2, on the following page. The outer lasso shows all the nodes associated with matching the predicate in rule **a**, and the inner lasso shows the nodes associated with matching the predicate

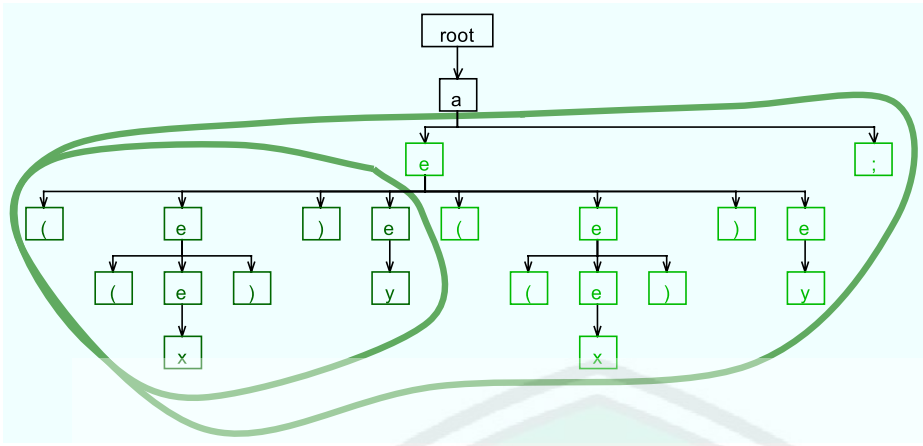


Figure 14.2: Partial parse tree for ((x))y; showing speculation of syntactic predicates in rules **a** and **e**

in the first alternative of rule **e**. At the outer level, rule **a**'s predicate matches rule **e** followed by ';', which is what you see as the two children of **a**'s parse tree node. The predicate in rule **e** matches '(' e ')' e, which is what you see as the first four children of the topmost **e** node. That topmost node shows the speculative match for the predicate and then the real match for the first alternative of rule **e**.

Because ANTLR-generated recognizers parse each alternative predicted by successful syntactic predicates twice, the full parse tree for input ((x))y; repeats the nodes in the outer lasso, making the tree twice as wide, as shown in Figure 14.3, on the next page. Figure 14.3, on the following page is the right half of the full parse tree, and Figure 14.2, is the left half.

The recognizer does a lot of extra work during nested backtracks. For example, the recognizer matches ((x))y against the predicate in the first alternative of rule **e** twice. It also matches the inside of the typecast, (x), multiple times. See Section 14.5, *Memoization*, on page 343 to learn how ANTLR can save partial parsing results to avoid repeated parsing at the cost of some memory.

Now consider input ((x)); that also matches alternative one in rule **a**. While evaluating the outermost **e** invocation, the second alternative of **e** matches input ((x)). This means the syntactic predicates in the first

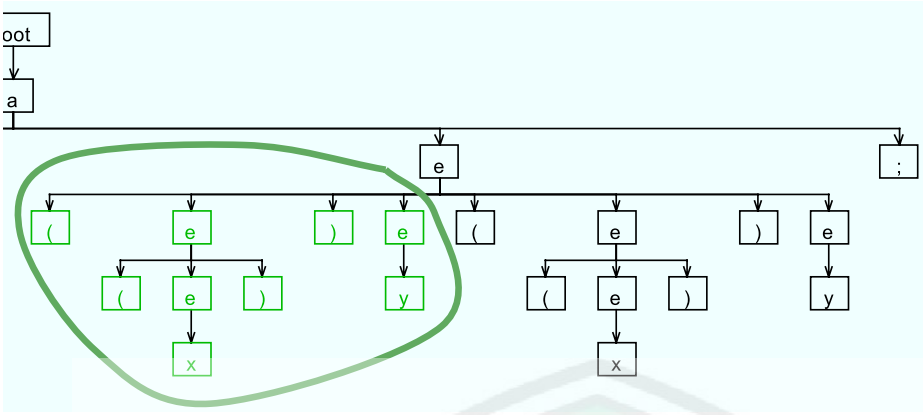


Figure 14.3: Partial parse tree for ((x))y; showing reevaluation of syntactic predicate in rule **e**

alternative of **e** fails. Figure 14.4, on the next page, shows the partial parse tree for this input. The inner lasso (in red in the PDF version) shows the nodes matched until the predicate failed. The outer lasso shows that, nonetheless, the syntactic predicate in the first alternative of rule **a** succeeded.

Specifying lots of predicates to resolve non-*LL(*)* decisions can be a hassle and can make your grammar less readable. The next section describes how you can simply turn on automatic backtracking, which automatically engages backtracking for non-*LL(*)* decisions.

14.4 Auto-backtracking

Altering your grammar to suit the needs of the underlying *LL(*)* parsing algorithm can be a hassle, but so can adding syntactic predicates all over the place. Instead, you have the option to leave the grammar as is and turn on the **backtrack** option:

```
grammar Cpp;
options {backtrack=true;}
...
```

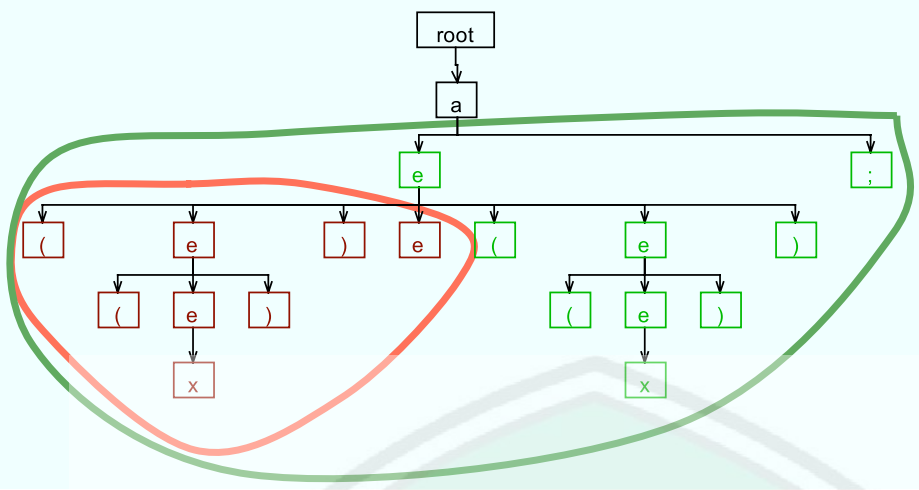


Figure 14.4: Partial parse tree for ((x)); showing speculation of syntactic predicates in rules **a** and **e** where the syntactic predicate in the first alternative of **e** fails

The idea is that, when *LL(*)* analysis fails, the recognizers should automatically backtrack at runtime to figure it out. This feature provides a "newbie," or rapid prototyping, mode that makes it fast to build a grammar. After you have created a correct and readable grammar, you can optimize the grammar a little to reduce the amount of backtracking by tweaking rules to make them *LL(*)*.

ANTLR implements the auto-backtracking mode by adding a syntactic predicate to the left edge of every alternative. These implicit syntactic predicates become regular (nongated) semantic predicates. *LL(*)* analysis uses such semantic predicates only in non-*LL(*)* decisions; that is, recognizers evaluate the predicates only when *LL(*)* fails.

In contrast, manually specified syntactic predicates become gated semantic predicates; they are always evaluated when predicting the associated alternative. By converting implicit syntactic predicates to nongated semantic predicates, you get clarity, power, and efficiency.

To illustrate this, here is a version of the **nested** grammar from earlier rewritten to use auto-backtracking mode; it is much easier to read:

```
grammar nested;
options {backtrack=true;}
a : e ';'
  | e '.'
  ;

e : '(' e ')' e  // type cast
  | '(' e ')'    // nested expression
  | ID
  ;

ID: 'a'..'z'+ ;
```

You still need manually specified syntactic predicates in some cases, however. To illustrate how manually specified predicates differ from auto-backtracking syntactic predicates, consider the following grammar. It reflects a situation where you want to treat identifiers `s` and `sx` differently depending on whether they followed by an integer. The syntactic predicates examine more of the input than the associated alternatives match (consume).

Download synpred/P.g

```
lexer grammar T;
/** For input s followed by INT, match only s; must exec action1.
 *  For sx followed by INT, match only sx; exec action2.
 */
ID : ('s' INT)=> 's'                {action1;}
   | ('sx' INT)=> 'sx'              {action2;}
   | 'a'..'z' ('a'..'z'|'0'..'9')*  {action3;}
   ;
INT : '0'..'9'+ ;
```

Input `s` must trigger `action3` unless followed by an integer. In that case, `s` must trigger `action1` but still consume only the `s`. The same is true for input `sx`. Without the syntactic predicates, ANTLR emits a warning about the same input predicting multiple alternatives. ANTLR resolves the ambiguity by forcing `s` to always predict alternative one. `sx` would always predict alternative two. Using auto-backtracking mode won't work in this case as a replacement for manually specified syntactic predicates. Auto-backtracking adds predicates to each alternative, but they would not have the **INT** reference in them. Clearly, you would not get the desired functionality.

When using auto-backtracking mode, ANTLR grammars behave like Bryan Ford's *parser expression grammars* (PEGs) [For04]. Several interesting PEG-based parser generators are available, including Ford's original implementation [For02] and Robert Grimm's Rats! [Gri06]. I also want to mention James Cordy's TXL [Cor06] that has ordered alternatives but without syntactic predicates and without partial result memoization, the topic of the next section.

Although auto-backtracking is useful, it can be very expensive in time. The next section describes how recognizers can record partial parsing results to guarantee linear time at the cost of memory.

14.5 Memoization

New in v3.

Backtracking is an exponentially complex algorithm in the worst case. The recognizer might have to try multiple alternatives in a decision, which, in turn, might invoke rules that also must try their alternatives, and so on. This nested backtracking yields a combinatorial explosion of speculative parsing. For any given input position, a backtracking parser might attempt the same rule many times, resulting in a lot of wasted effort. In contrast, a parser without backtracking examines the same input position at most once for a given rule, resulting in linear time complexity.

If, on the other hand, the recognizer remembers the result of attempting rules at the various input positions, it can avoid all the wasted effort. Saving partial results achieves linear complexity at the cost of potentially large amounts of memory.

This process of remembering partial recognition results is a form of dynamic programming called *memoization* or, more specifically in the parsing arena, as *packrat parsing* [For02]. Bryan Ford introduced the technology and coined the term.[4] Because ANTLR uses packrat parsing only when *LL(*)* fails, it often generates parsers that are substantially more efficient in time and space than pure packrat parsers such as Grimm's Rats!

The easiest way to demonstrate the benefit of memoization is to examine the vivid differences in the parse trees for the same input with and without memoization.

4. See http://pdos.csail.mit.edu/~baford/packrat for more information about packrat parsing.

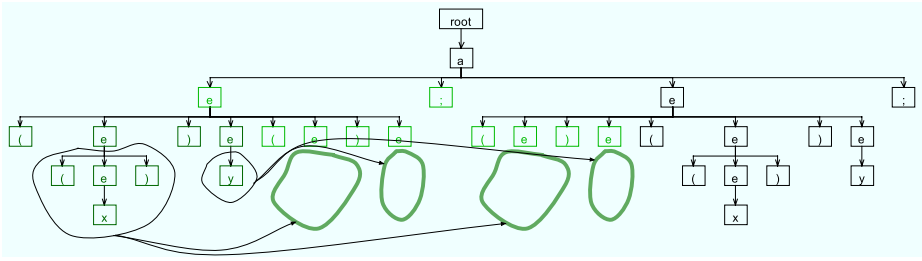


Figure 14.5: parse tree for ((x))y; showing redundant parse subtree elimination for two successful invocations of rule **e** during evaluation of **e**'s syntactic predicate

Let's revisit the nested syntactic predicate grammar from Section 14.3, *Nested Backtracking*, on page 337 but with the addition of the **memoize** option:

Download synpred/memoize.g

```
grammar memoize;
options {memoize=true;}

a : (e ';')=> e ';'
  | e '.'
  ;

e : ('(' e ')' e)=> '(' e ')' e  // type cast
  | '(' e ')'            // nested expression
  | ID
  ;

ID: 'a'..'z'+ ;
```

Reexamine Figure 14.2, on page 339, and Figure 14.3, on page 340. Again with input ((x))y;, compare those figures to Figure 14.5. Focus on the overall shape of the parse trees here rather than the individual nodes. Notice that the memoized version is missing redundant subtrees. The lassos indicate portions of the parse that the parser avoids because of memoization. The thin lassos and arrows indicate which partial results the parser reused. The nested syntactic predicate in rule **e** succeeds, as does the outer syntactic predicates in the first alternative of rule **a**. When that outer predicate succeeds, rule **a** reenters **e** during the actual parse of the first alternative of rule **a**. Rule **e** again enters the ('(' e ')' e)=> syntactic predicate. Thanks to memoization, the

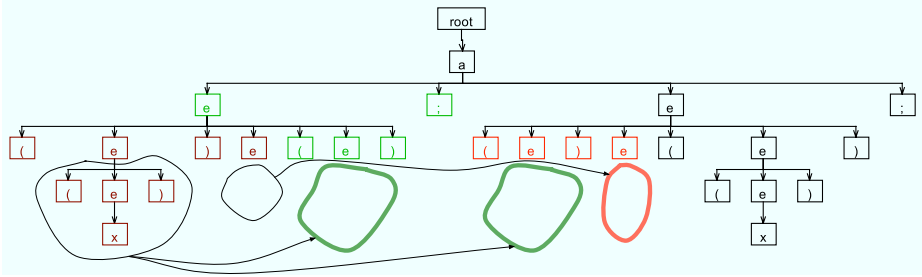


Figure 14.6: Parse tree for ((x)); showing redundant parse subtree elimination for one successful and one failed invocation of rule **e** during evaluation of **e**'s syntactic predicate

parser does not have to enter the two **e** references for that predicate. It already knows they will succeed.

Memoization also records failures to avoid reparsing rules that it knows won't succeed. Compare Figure 14.4, on page 341, to Figure 14.6, again with input ((x));. The syntactic predicate in rule **a**'s first alternative invokes rule **e**. Rule **e** then evaluates the syntactic predicate in its first alternative. That syntactic predicate recursively invokes rule **e** and records success for (x) (the '(' e ')' portion of the predicate). Because no identifier comes after the parentheses and before the ';', the parser records failure for the second invocation of rule **e** in the '(' e ')' e predicate. Failing the predicate, rule **e** matches its second alternative instead. This allows the outer syntactic predicate in the first alternative of rule **a** to succeed. During the subsequent real parse of **a**'s first alternative, the parser invokes **e** immediately and reencounters **e**'s syntactic predicate in its first alternative. Memoization tells the parser that the first **e** reference in the syntactic predicate will succeed, but the second will fail. The parser fails just as it did before while evaluating **a**'s predicate, but this time, the parser skips the redundant parse thanks to memoization. The lassos in Figure 14.6, show redundant parse subtree elimination and the original subtree computation.

To implement memoization, ANTLR inserts code at the beginning of each rule recognition method to check for prior attempts. Naturally, the parser can avoid parsing a rule only when backtracking:

```
if ( backtracking>0 && alreadyParsedRule(input,rule-number) ) {return;}
```

If the recognizer has already attempted this rule and at the same input position, the recognizer seeks ahead to where the rule finished parsing last time. Then the rule returns immediately, effectively skipping all the previously done parsing work. Moreover, the parser has effectively handled the rule in constant time. If the rule failed during the previous attempt, the rule also returns immediately but sets an error flag indicating failure.

To do the actual memoization, ANTLR inserts code at the end of a rule. This code records whether the rule completed successfully at this input position:

```
if ( backtracking>0 ) {
   memoize(input, rule-number, rule-starting-input-position);
}
```

In other words, the **memoize** option alters code generation from this:

```
public void r() throws RecognitionException {
    try {
        «r-prediction»
        «r-matching»
    }
    catch (RecognitionException re) {
        reportError(re);
        recover(input,re);
    }
}
```

to the following:

```
public void r() throws RecognitionException {
    int r_StartIndex = input.index();
    try {
        if ( backtracking>0 && alreadyParsedRule(input,r-index) ) {
            return;
        }
        «r-prediction»
        «r-matching»
    }
    catch (RecognitionException re) {
        reportError(re);
        recover(input,re);
    }
    finally {
        if ( backtracking>0 ) { memoize(input,r-index,r_StartIndex); }
    }
}
```

The «*r-prediction*» and «*r-matching*» code blocks are the same as is the usual **try/catch** surrounding them to handle error recovery. The difference is that, when backtracking, the rule immediately returns if it has already parsed the input starting from the current input position. If there is no memoized result for this rule, the method memoizes success or failure after it finishes parsing.

A word about memoization efficiency: thanks to *LL(*)* prediction DFA, recognizers often avoid backtracking even in the presence of multiple syntactic predicates. This has the side effect of requiring much less storage space than a pure packrat parser because ANTLR-generated parsers are not always backtracking. This does not mean, however, that ANTLR optimizes away all unnecessary memoization.

In the future, ANTLR will have a "redundant reevaluation hotspot analysis" feature that you can turn on at runtime to figure out which rules are invoked repeatedly at the same input position.

Recording the result of each rule invocation during backtracking is expensive in memory and time, often actually slowing your recognizer down instead of speeding it up. The general rule is that you should use memoization on a rule-by-rule basis. Determining which rules to memoize can be difficult just by looking at the grammar. In general, look for sequences of syntactic predicates within the same decision that invoke the same rule directly or indirectly, as shown in the following grammar:

```
/** This rule must backtrack because of the common left prefix
 *  that is not only arbitrarily long but has nested structure;
 *  hence non-LL(*).
 */
s
options {backtrack=true;}
 : e ';' // invoke e once
 | e '.' // invoke e twice at same position
 | e '!' // default, invoke e "with feeling"
 ;

/** This rule should memoize parsing results because it is invoked
 *  repeatedly by rule s at the same input position.
 */
e
options {memoize=true;}
 : '(' e ')'
 | ID
 ;
```

You want to turn on memoization for the repeatedly invoked rule **e**, not the invoking rule. In this grammar, input ((x))! causes the parser to enter rule **e** from the implicit syntactic predicate in the first alternative of rule **s**. At the end of that invocation, the recognizer records

the successful parse of **e**. The second invocation of **e** from the second alternative of **s** can use that result to avoid reparsing **e**. The implicit syntactic predicates for rule **s**'s first two alternatives fail because the input ends with '!', not ';' or ';'. After the syntactic predicate for alternative two of **s** fails, the parser begins nonspeculative matching of **s**'s third alternative. At this point, the recognizer is no longer speculating because the third alternative is clearly the final choice. Even though the recognizer knows that rule **e** will succeed, it must enter rule **e** to execute any potential embedded actions (none in this case).

In the previous sections, you saw that ANTLR can automatically backtrack to handle just about any grammar, and you saw how that can be done in linear time. Unfortunately, automatic backtracking comes at the cost of some potential pitfalls when you build grammars—automatic backtracking can hide grammar ambiguities.

14.6 Grammar Hazards with Syntactic Predicates

If you turn on the **backtrack** option at the grammar level, you will not get any static grammar analysis warnings because the generated recognizer can resolve any non-*LL(*)* decisions at runtime by backtracking. Although this is a good way to rapidly prototype a language, the lack of static analysis can hide a number of serious problems with your grammar. Ultimately, your best bet is to selectively turn on backtracking at the rule level to resolve non-*LL(*)* decisions (assuming you don't want to alter the grammar instead).

This section describes a number of situations where you will not get what you want despite the lack of warnings from ANTLR. These hazards are inherent to the backtracking strategy, not a limitation of ANTLR.

The simplest way to explain the problem is with the following example grammar where the **k** option artificially constrains grammar analysis to a single symbol of lookahead:

```
parser grammar t;
s
options {k=1;}  // can't see past ID
  : ID
  | ID ';'
  ;
```

ANTLR reports this:

```
warning(200): t.g:4:5: Decision can match input such as
"ID" using multiple alternatives: 1, 2
As a result, alternative(s) 2 were disabled for that input
warning(201): t.g:4:5: The following alternatives are
unreachable: 2
```

With only a single symbol of lookahead, the recognizer cannot see past the **ID** in order to properly predict an alternative. Now, turn on auto-backtracking to allow ANTLR to resolve the conflict at runtime with backtracking:

```
parser grammar t;
options {backtracking=true;}
s
options {k=1;}  // force backtracking
  : ID
  | ID ';'      // unreachable
  ;
```

ANTLR issues no warnings, but the grammar can't ever match input `x;` even though that is clearly the intention. Syntactic predicates remove ambiguity by ordering alternatives, but just because a decision is unambiguous doesn't mean you get what you expect. The second alternative of rule **s** is unreachable because the first alternative will always win when **ID** is the next input symbol. The implied predicate on the first alternative is `(ID)=>`, which says "match this alternative if the first symbol of lookahead is an ID." The way to fix this is simply to reorder the alternatives:

```
parser grammar t;
options {backtracking=true;}
s
options {k=1;}  // force backtracking
  : ID ';'
  | ID
  ;
```

These hazards typically emerge only after extensive unit testing. To illustrate a less obvious hazard, recall the Java grammar from Section 11.5, *Nondeterministic Decisions*, on page 281 that matched a code block in rule **decl** as well as rule **stat**. Rule **slist** is ambiguous because both alternatives can match the same input construct, `{...}`:

```
options {backtrack=true;}
...
slist:  decl // can match {...} as default ctor
    |   stat // code block {...} is UNREACHABLE!
    ;
```

Turning on auto-backtracking mode will hush any ANTLR grammar warnings because ANTLR can then silently choose **decl** over **stat** for input that matches both alternatives. Clearly, though, turning on auto-backtracking is not the solution. You should refactor the grammar, as shown in Section 11.5, *Nondeterministic Decisions*, on page 281. Most grammars are unambiguous, and therefore, you do not want to hide improperly resolved *LL(*)* nondeterminisms by choosing alternatives in the order presented.

The next kind of grammar hazard involves optional rules and alternatives, which are commonly used in grammars. Sometimes an optional alternative renders further alternatives in that decision unreachable, as illustrated in the following grammar that ANTLR processes without warning:

Download synpred/unreachable.g

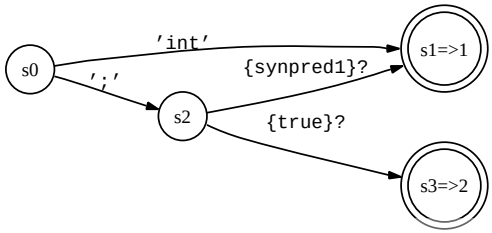
```
grammar unreachable;

slist
  : (var)=>var ';' {System.out.println("slist alt 1");}
  |            ';' {System.out.println("slist alt 2");}
  ;

var
  : 'int' ID       {System.out.println("match var");}
  |                {System.out.println("bypass var");}
  ;

ID: 'a'..'z'+ ;
WS: (' '|'\n'|'\r')+ {skip();} ;
```

Because the first alternative invokes an optional rule, **var**, the recognizer will not know what to do upon lookahead symbol *';'* because both alternatives match the simple *';'*. Because of the syntactic predicates, ANTLR assumes you know what you're doing and does not warn you about the nondeterminism. The syntactic predicates assumes that you mean to order the alternatives in terms of precedence. Without the syntactic predicate, however, ANTLR does warn you:

```
$ java org.antlr.Tool unreachable.g
ANTLR Parser Generator  Version 3.0  1989-2007
warning(200): unreachable.g:4:5: Decision can match input such as
"';'" using multiple alternatives: 1, 2
As a result, alternative(s) 2 were disabled for that input
warning(201): unreachable.g:4:5: The following alternatives are unreachable: 2
```

This highlights that you should use syntactic predicates as sparingly as possible to avoid hiding grammar conflicts.

Take a look at the prediction DFA for the decision in rule **slist**:



The DFA transitions from s0 to s2 upon input ; and then uses the syntactic predicate synpred1() to distinguish between alternatives one and two (predicted by s1 and s3, respectively). The syntactic predicate will always return true because even if the first alternative of rule **var** fails, the second alternative will always succeed because it does not have to match anything. Consequently, rule **slist**'s alternative two is unreachable. Input ; will always force the recognizer to match the second alternative of **var** and the first alternative of **slist**. You can use the following test harness to verify this behavior:

Download synpred/TestUnreachable.java

```
import java.io.*;
import org.antlr.runtime.*;

public class TestUnreachable {
    public static void main(String[] args) throws Exception {
        unreachableLexer lex =
            new unreachableLexer(new ANTLRInputStream(System.in));
        CommonTokenStream tokens = new CommonTokenStream(lex);
        unreachableParser p = new unreachableParser(tokens);
        p.slist();
    }
}
```

Sending in a valid variable declaration behaves as you would expect, but sending in ; does not execute the action in the second alternative of **slist**. Most likely, given that the grammar designer specifically provided an alternative matching purely a semicolon, this behavior would be unwelcome.

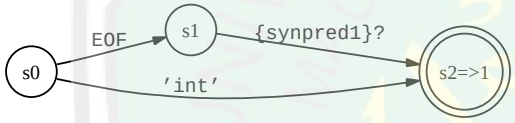
The worst-case scenario for the optional rule hazard results in an infinite loop. Imagine that an EBNF looping construct must backtrack to distinguish between an optional alternative and the implicit exit branch.

This loop will never terminate because you will always choose the optional alternative rather than the exit. The optional alternative matches any symbol; hence, the syntactic predicate's speculative parse will always succeed. Since syntactic predicates tell the recognizer to choose the first alternative that matches, the "stay in the loop" alternative always wins. The following grammar matches an optional list of variable definitions and illustrates exactly such an infinite loop situation:

Download synpred/infloop.g

```
grammar infloop;

slist
  : ( (var)=> var {System.out.println("in loop");} )+
  ;

var
  : 'int' ID ';' {System.out.println("match var");}
  |              {System.out.println("bypass var");}
  ;

ID: 'a'..'z'+ ;
WS: (' '|'\n'|'\r')+ {skip();} ;
```

It is worth looking at the prediction DFA:



Clearly, the DFA can predict only alternative one, but the syntactic predicate hushes the warning about alternative two being unreachable. Here are the warnings ANTLR emits if the subrule were var+ (without the syntactic predicates):

```
$ java org.antlr.Tool infloop.g
ANTLR Parser Generator  Version 3.0  1989-2007
warning(200): infloop.g:4:44: Decision can match input such as
"{EOF, 'int'}" using multiple alternatives: 1, 2
As a result, alternative(s) 2 were disabled for that input
warning(201): infloop.g:4:44: The following alternatives are
unreachable: 2
$
```

Backtracking and Action Execution in the Maze

While backtracking in the maze, you do not want to record anything in your notebook as you speculatively explore the various paths emanating from a fork. It is difficult to remember what you need to erase after each speculative walk. Worse, some actions are impossible to undo such as print statements or annoying your spouse from your cell phone as you walk the maze. It is easier to ignore actions until you are sure which alternative path you will take. Then, you can write in your notebook (execute actions) as you move down the correct path for the second time.

Using the following test harness, you can verify that this program will not terminate either on a valid variable definition or on an empty input stream.

Download synpred/TestInfLoop.java

```
import java.io.*;
import org.antlr.runtime.*;

public class TestInfLoop {
    public static void main(String[] args) throws Exception {
        infloopLexer lex =
            new infloopLexer(new ANTLRInputStream(System.in));
        CommonTokenStream tokens = new CommonTokenStream(lex);
        infloopParser p = new infloopParser(tokens);
        p.slist();
    }
}
```

Fortunately, these grammar hazards are not huge problems as long as you are aware of them. As a general rule, use auto-backtracking mode to resolve non-*LL(*)* conflicts and use manually specified syntactic predicates to explicitly resolve grammar ambiguities. The next section discusses one final potential difficulty when using syntactic predicates.

14.7 Issues with Actions and Syntactic Predicates

ANTLR allows you to embed arbitrary actions (written in the target language) in your grammar, which is a really nice feature unavailable in a lot of recent parser generators. The reason other tools do not support arbitrary actions is because actions often conflict with the parsing

strategy. As you saw earlier, ANTLR recognizers always match successful alternatives twice—once during backtracking and once "with feeling" to execute any actions.

To avoid executing actions while backtracking, ANTLR generates an **if** statement around all embedded actions that turns the action off unless the backtracking level is 0:

```
if (backtracking==0) { «action» }
```

This implies that you should not define local variables in actions other than the **init** action because the scope of a variable is limited to the **if** statement. ANTLR does not gate **init** actions in and out because they typically include variable declarations. If you need to, you can define your own special action gate expression with a global action:

```
@synpredgate { «my-expr-to-turn-on-actions» }
```

There is another important limitation that ANTLR must impose upon your grammar actions and rules because of backtracking. Consider the following grammar with non-*LL(*)* rule **a** that needs backtracking:

```
grammar t;
a
@init {int y=34;}
  : (b[y] '.')=> b[y] '.'
  |              b[y] ':'
  ;

b[int i]
  : '(' b[i] ')' {System.out.println(i);}
  | 'z'
  ;
```

For syntactic predicate (b[y] '.')=> in the first alternative, ANTLR generates this:

```
public void synpred1_fragment() throws RecognitionException {
    // t.g:5:5: b[y] '.'
    b(y); // TROUBLE! local variable y undefined!
    if (failed) return ;
    // match '.'
    match(input,4,FOLLOW_4_in_synpred144); if (failed) return ;
}
```

Unfortunately, this code will not compile because argument y passed to rule **b** does not exist in synpred1_fragment()'s scope. That argument is essentially an action, albeit a single expression, and ANTLR has taken it out of its original context. ANTLR cannot simply leave it out because the method for rule **b** requires a parameter. Without a parameter on b(),

Fortunately, this problem is not a limitation of the fundamental parsing strategy, and there is a way out using dynamic scoping (Section 6.5, *Dynamic Attribute Scopes for Interrule Communication*, on page 148). We need a solution that does not require the use of method parameters so that syntactic predicates, taken out of context, will compile. But, at the same time, we must still be able to pass information from one rule to another. ANTLR's dynamic scopes are a perfect solution. Here is the equivalent grammar using a dynamic scope in rule **a**:

```
grammar t;
a
scope {
  int i;
}
@init {$a::i=34;}
  : (b '.')=> b '.'
  |           b ':'
  ;

b : '(' b ')' {System.out.println($a::i);}
  | 'z'
  ;
```

This version is not as explicit and is less conventional, but it does yield a functionally equivalent recognizer that compiles.

Rule return values do not have the same problem because ANTLR can simply avoid generating code that stores a return value. In the regular code, ANTLR generates `x=b();`. For syntactic predicate fragments, however, ANTLR generates just `b();`.

In a nutshell, if you use backtracking and parameter passing a lot, you might run into a situation where method arguments result in noncompilable code. In such a situation, recode the arguments as dynamically scoped variables.

Syntactic predicates, introduced more than 15 years ago, have become a well-entrenched bit of parsing technology. Thanks to Bryan Ford, we now have a formal language treatment in the form of PEGs and, most important, the packrat parsing strategy that guarantees linear parse time at the cost of nonlinear memory. ANTLR's implementation of packrat parsing is particularly efficient because it avoids backtracking whenever classical *LL(k)* lookahead or the new *LL(*)* lookahead is capable of distinguishing between alternatives. Moreover, static *LL(*)* grammar analysis provides useful ambiguity warnings that help avoid grammar hazards; this analysis is not available to pure PEG parser generators.

This chapter concludes a long (four-chapter) sequence representing a complete course of study in predicated-*LL(*)* parsing. If you have read and grokked most of this, you are well on your way to becoming a practical parsing and grammar expert. The discussion is short on theory but long on annotated examples because those are more valuable to programmers trying to get their jobs done. At the same time, each chapter contains sufficient details for someone to reimplement the technology in another parser generator. In the interest of friendly competition, I hope that the authors of other parser generators will incorporate hoisted semantic predicates and something akin to *LL(*)* static grammar analysis.

Appendix A

Bibliography

[Cor06] James R. Cordy. The txl source transformation language. *Sci. Comput. Program.*, 61(3):190–210, 2006.

[ES90] Margaret A. Ellis and Bjarne Stroustrup. *The annotated C++ reference manual*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1990.

[FHP92] Christopher W. Fraser, Robert R. Henry, and Todd A. Proebsting. Burg: fast optimal instruction selection and tree parsing. *SIGPLAN Not.*, 27(4):68–76, 1992.

[For02] Bryan Ford. Packrat parsing:: simple, powerful, lazy, linear time, functional pearl. In *ICFP '02: Proceedings of the seventh ACM SIGPLAN international conference on Functional programming*, pages 36–47, New York, NY, USA, 2002. ACM Press.

[For04] Bryan Ford. Parsing expression grammars: a recognition-based syntactic foundation. In *POPL '04: Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 111–122, New York, NY, USA, 2004. ACM Press.

[Gri06] Robert Grimm. Better extensibility through modular syntax. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, pages 38–51, New York, NY, USA, 2006. ACM Press.

[Gro90] Josef Grosch. Efficient and comfortable error recovery in recursive descent parsers. *Structured Programming*, 11(3):129–140, 1990.

[Joh79] Steven C. Johnson. Yacc: Yet another compiler compiler. In *UNIX Programmer's Manual*, volume 2, pages 353–387. Holt, Rinehart, and Winston, New York, NY, USA, 1979.

[Les75] M.E. Lesk. Lex - a lexical analyzer generator. Technical Report Computing Science Technical Report No.39, Bell Telephone Laboratories, 1975.

[Pin94] S. Pinker. *The Language Instinct: How the Mind Creates Language*. HarperCollins, New York, 1994.

[Tom87] Masaru Tomita. An efficient augmented-context-free parsing algorithm. *Comput. Linguist.*, 13(1-2):31–46, 1987.

[Top82] Rodney W. Topor. A note on error recovery in recursive descent parsers. *SIGPLAN Not.*, 17(2):37–40, 1982.

[TTG94] J. C. Trueswell, M. K. Tanenhaus, and S. M. Garnsey. Semantic influences on parsing: Use of thematic role information in syntactic ambiguity resolution. *Journal of Memory and Language*, 33(3):285–318, June 1994.

[Wir78] Niklaus Wirth. *Algorithms + Data Structures = Programs*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1978.

Index

Symbols

A

D

E

F

G

H

I

J

K

L

M

N

O

P

S

T

U

V

W

Z

Web-Based Architecture of an Intelligent Tutoring System for Remote Students Learning to Program Java™

Edward R. Sykes
School of Computing and Information Management, Sheridan College
1430 Trafalgar Road, Oakville, Ont., Canada, L6H 2L1
+1 (905) 845-9430 Ext. 2490
ed.sykes@sheridanc.on.ca

Franya Franek
Department of Computing and Software, Faculty of Science, McMaster University
1280 Main Street W., Hamilton, Ont., Canada, L8S 4L8
+1 (905) 525-9140 Ext. 23233
franek@mcmaster.ca

Abstract

The "Java™ Intelligent Tutoring System" (JITS) research project involves the development of a programming tutor designed for students in their first programming course in Java™ at the College and University level. This paper presents an overview of the architectural design including state-of-the-art web-based distributed architecture, the AI techniques used, and the programmer-optimized user interface. This project is a prototype being constructed which will model the domain of a small subset of the Java™ programming language in a very specific context. Research is in progress and it is hypothesized that the completed prototype will be sufficient to prove the concept and that a fully developed Java™ Intelligent Tutoring System will provide an interactively-rich learning environment for students that will result in increased achievement. Based on the success of similar Intelligent Tutoring Systems, it is also hypothesized that these students will be able to learn programming skills and knowledge more quickly and effectively than students in traditional educational settings.

Keywords: Intelligent Tutoring Systems, Web-Based Education, Programming Tutors, AI in Education.

1. Introduction

Intelligent Tutoring Systems (ITS) are, in many respects, very similar to human tutors. Based on cognitive science and Artificial Intelligence (AI), ITS have proven their worth in multiple ways in multiple domains in Education [1, 5]. Currently, ITS can be found in core Mathematics, Physics, and Language courses in many schools across Canada, the United States, and various countries in Europe. ITS are growing in acceptance and popularity for reasons including: i) increased student performance, ii) deepened cognitive development, and iii) reduced time for the student to acquire skills and knowledge [1, 2, 5].

Intelligent Tutoring Systems that tutor and monitor programming have been developed and evaluated for many years in the field of Artificial Intelligence in Education. In many ways, programming has been a very productive domain in the evolution of most aspects of the field, including student modeling, knowledge representation, and the application of sound pedagogical principles. Effective programming requires a range of problem-solving and diagnostic strategies. The manner in which a student writes code provides rich insight into the reasoning processes of the student. As a result, programming provides an interesting domain for studying learning and cognitive processes.

The goal of the current research is to bring together recent developments in the fields of Intelligent Tutoring Systems, Cognitive Science, and AI to construct an effective intelligent tutor help students learn to program in Java™. In addition to contributing to understand the learning process in general, it is hoped that this research will have a positive impact on supporting instructors teaching Java™ programming in their institution. More than ever, this is an important area for institutions where there are more students wishing to learn to program, and where it is difficult to provide personalized instruction that they need [3]. Additionally, since there are a growing number of institutions investing in distance learning, this research will play a significant role to provide appropriate methods of teaching this key subject to students learning remotely.

2. Java ITS Model and Architecture

This section presents the model and architecture for the Java™ Intelligent Tutoring System. Four distinct components are presented that support JITS: the curriculum design, the AI module, the distributed web-based infrastructure, and the user interface design.

2.1. JITS Curriculum Design

This section describes the curriculum architectural model for JITS. Due to the complexity involved with semantic parsing, it is necessary to restrict the JITS to tutor a small subset of the Java programming language. The area of focus involves the following list of Java™ language basics:

a. variables (declaration, use, local vs. global),
b. operators, and
c. looping structures.

A database of records with the following structure will be constructed:

Given a Problem (P), there are n number of Solutions (S_1, S_2, …, S_n), with a number (m) of categories of classifiable incorrect responses (R_1, R_1,…, R_m). For each incorrect response category there is a finite number (t) of appropriate hints (H_1, H_2,,…,H_t). Table 1 illustrates an example of a problem with solutions, some incorrect responses.

Table 1. Java™ ITS Curriculum Architecture

Problem:
Write a program called "Summer" which adds all the integer numbers from 1 to a specified number (N). For example, if N were assigned the value 10, then the sum of the numbers from 1 to 10 is 55.

Program specifications:
This program requires the use of a for-loop structure. A skeleton structure of the solution is given. Fill in the code to complete this program.

```
OUTPUT>Sum = 55
```

Skeleton Program (located in Source Code area):

```
public class Summer {
 public static void main(String[] args)
 {
   int sum = 0;
   int i = 0;

   /* student writes code here */

   System.out.println("Sum = " + sum);
 }
}
```

Solution (one of n):

```
public class Summer {
   public static void main(String[] args) {
     int sum = 0;
     int i = 0;
     for (i = 1; i <= 10; i++) {
         sum += i;
     }
     System.out.println("Sum = " + sum);
  }
}
```

Incorrect response #1 (student response area): (redeclaration of variable 'i')

```
     for (int i = 1; i <= 10; i++) {
         sum += i;
     }
```

Incorrect response #2: (sum is 0, as the body of the loop is never executed)

```
     for (i = 1; i > 10; i++) {
         sum += i;
     }
```

Incorrect response #3: (adding 1 instead of variable 'i': results in sum being lower than expected)

```
     for (i = 1; i <= 10; i++) {
         sum += 1;
     }
```

etc.

The astute reader recognizes that there are limitless possibilities for student responses and the system cannot simply list incorrect responses coupled with feedback messages. For instance, the student could write:
`sum=(n+1)*(n/2);` or `sum=(n*n+n)/2;`

Both answers are completely correct and the system needs to recognize these types of responses and not merely respond back to the student indicating a failure. Testing the correctness of a program is not an easy task, and cannot be achieved just by giving a set of fixed responses. JITS is designed to be pedagogically sound. So, although the above formulas result in correct answers, this is not the final goal of the tutoring system. Rather, JITS focuses on the methodology by which a student attempts to solve a problem. Conventions, style, and professional programming techniques are modeled in JITS. In this fashion effective tutoring may take place. These pedagogical issues as they are designed in JITS are addressed in the following sections.

2.2. JITS AI Module

In order for JITS to provide intelligent feedback to the student the AI module relies on a collection of information: the problem statement, the problem specification, student's code, the established student model, the expert model, the Java™ Parser, the syntactic_decision_tree, the semantic_decision_tree, the

Java™ Parse Tree, the output from the Java™ compiler, and the result from the Java™ runtime engine. Obviously, based on the context, some of this information will not be available. The two decision trees (i.e., syntactic_decision_tree, and semantic_decision_tree) represent the strategic and judgmental knowledge for the specific programming problem currently being examined by the student [4].

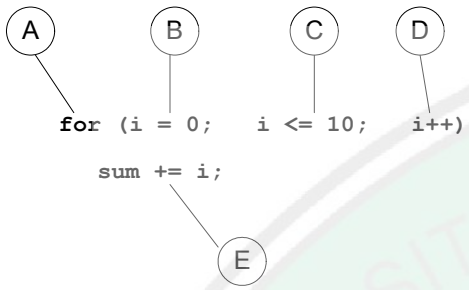


Figure 1. Sub-sections of expert model solution

Using the arithmetic sum problem described in Table 1, Figure 1 depicts how the expert model's solution may be divided into discrete sections. Based on this, Figure 2 represents the high-level flowchart showing the process from student to feedback generation.

2.2.1. JITS AI Module: Feedback

The information gathered by the AI module previously described is then carefully scrutinized so that appropriate hints can be generated for the student. For instance, if the Java™ Parser fails the "Fuzzy Scanner" (FS) module computes the edit distance between the student's code ($s1$) and the solution ($s2$), and constructs a transformation function string (i.e., $T : s1 \rightarrow s2$). T involves all insertions, deletions, transpositions, and character changes that are required to transform $s1$ into $s2$. The FS module uses approximate string matching techniques implemented by a recursive dynamic programming algorithm. The AI module then goes ahead of the student by constructing feasibly-sound variations of the student's code and proceeds to compile and run these. The information produced by the FS module is fed into the syntactic_decision_tree to determine appropriate feedback.

An expert system (supporting the decision trees), and two methods support the feedback mechanism: `general_hint(int context, String snippet)` and `specific_hint(int context, String snippet)`. Although the two decision trees isolate the specific area of error within the student's code, additional fine-grained analysis occurs within these methods. These methods are passed an integer representing the context in which the current programming issue has been identified.

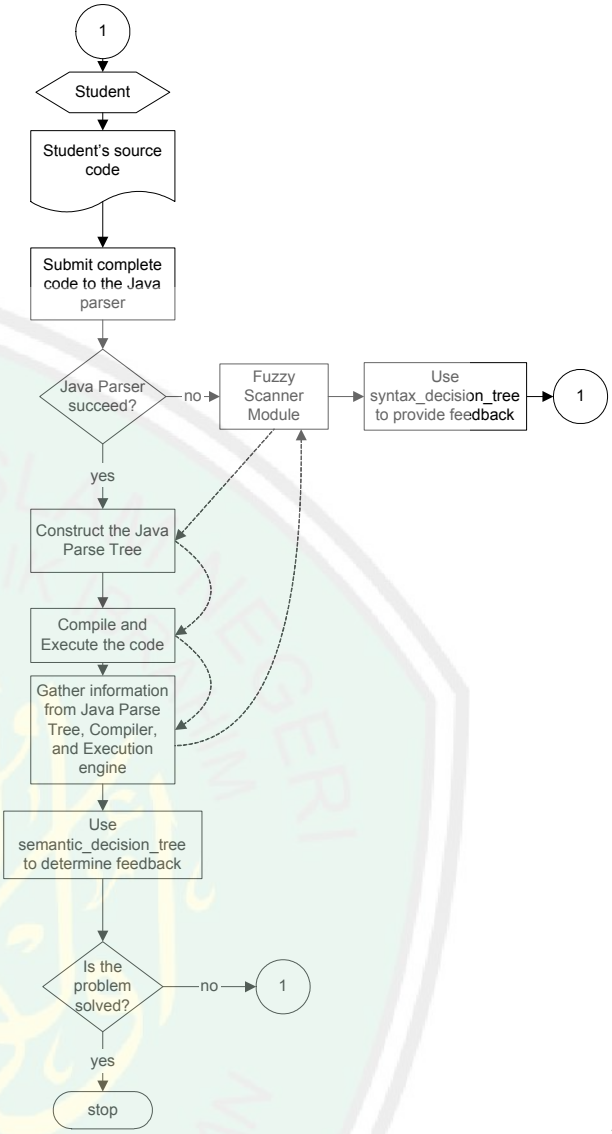


Figure 2. Flowchart of JITS AI Module

The second argument, `snippet`, represents the small portion of code associated with the given `context`. For example, if the student submitted: `sum = (n*n+n)/2;` only, then `general_hint()` would issue a message such as, *"Your answer is correct. However, the program specification is asking for a solution using a for-loop construct. Please revise your code."* Specific hints are code-specific generated in the same fashion as a human tutor would during troubleshooting to pinpoint the exact situation in which a syntactic or logic error has occurred. Figure 3 depicts a small section of the semantic_decision_tree (please refer to sections A and B from Figure 1).

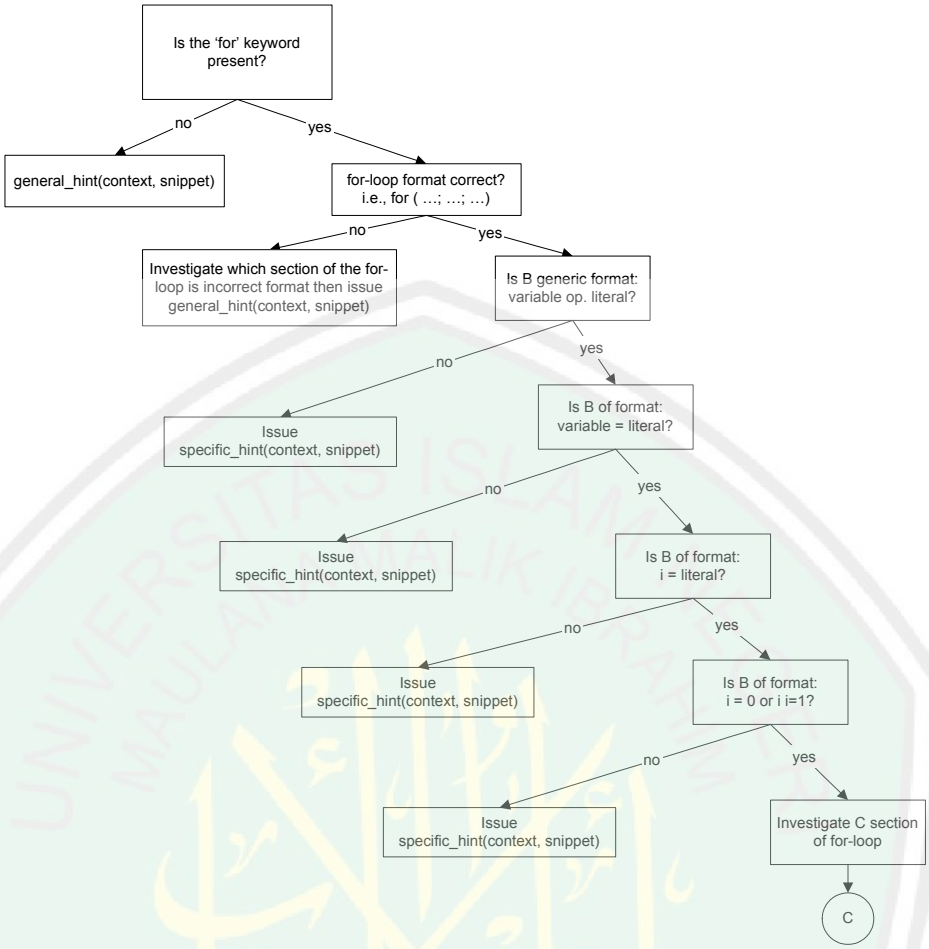


Figure 3. JITS semantic_decision_tree (sections A and B from Figure 1 only)

2.3. JITS Distributed Web-based Infrastructure

The JITS infrastructure supports the student via a browser accessing information from the tutor via an HTTP request/response process model. The processing is accomplished by Enterprise JavaBeans™ within a J2EE compliant server in combination with a web server supporting the presentation logic for the tutor. The presentation layer uses JavaServer Pages™ technology which communicates with the home interface of the bean for processing and returns a simple page back to the student's browser (e.g., html, xml, etc.). During processing the bean gathers all the information about the student's code and submits it to the AI module for processing. The infrastructure architecture uses a JDBC connection from the Enterprise JavaBeans™ to an external database which stores and retrieves specific information about the student including student history and performance statistics.

The proposed architecture has numerous benefits. It is scalable, platform-independent, and lightweight. The student will never need to install software on their machine and will not need a high-speed network connection to use JITS. Other benefits include fast execution as all processing is done on the J2EE server and the middle-tier web server which typically have much faster and more efficient hardware than typical PCs. The net result is a product that increases the accessibility for JITS to many students – a vital requirement for an equitable and successful educational product in today's Internet-ready community.

2.4. JITS User Interface

The interface for computer-based programming tutors is a significant factor that was given careful consideration during the design of JITS. The user interface is based on a presentation format implemented in many popular Integrated Development Environments used by professional programmers (e.g., Visual Café, JDeveloper). Upon connecting to JITS website, the student's browser displays the working environment for JITS. An appropriate skill-level problem is selected or the problem that last attempted is presented to the student.

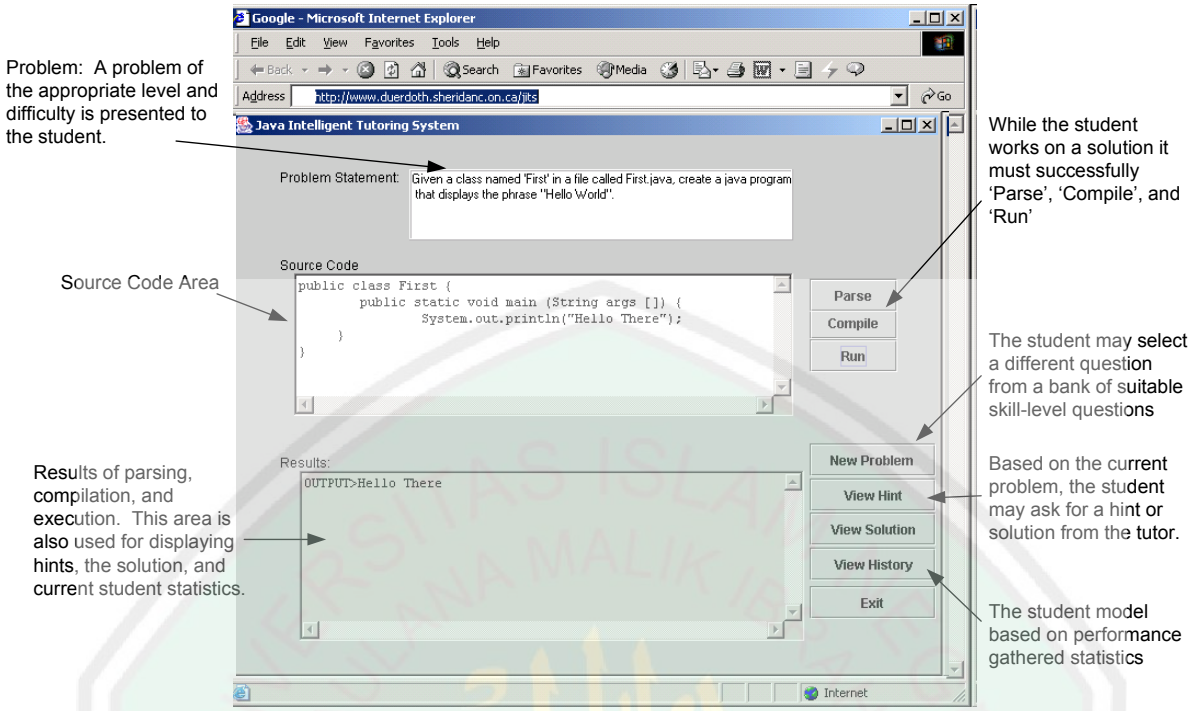


Figure 4. JITS User Interface

The student types in a solution in the Source Code Area and presses 'Parse'. This invokes a call to the corresponding Enterprise JavaBean™ representing the student. Information is gathered (i.e., student model, Java™ Parser, compiler, runtime engine, etc.) and submitted to the AI module. If the parser does not succeed then the AI module will determine the appropriate response based on the syntactic_decision_tree. JITS also goes beyond the student and attempts to compile and execute the code and variations on the submitted code. This yields additional information for the AI module to construct intelligent feedback. Specific feedback is generated and sent to the student's browser.

The student, at any time, may explicitly request a hint from JITS, view the solution, opt to quit the problem and select another, and view their performance history based on statistics including problems attempted, problems solved, number of attempts on a problem, and problem difficulty. The JITS user interface is shown in Figure 4.

3. Conclusions

In summary, the Java™ Intelligent Tutoring System prototype is designed using advanced cognitive science and AI techniques promoting the necessity for on-going research and development in the field of web-based educational tools. This research project is significant since it has the potential to be applied to numerous programming courses at the College and University level. Furthermore, it is important to the field of Education in both e-learning and traditional settings. The project in progress is based on sound theories and practices used in successful Intelligent Tutoring Systems and draws from the achievements ITS researchers have had in related projects.

4. References

[1] Anderson, J. R., Corbett, A. T., Koedinger, K. R., & Pelletier, R. (1995). Cognitive Tutors: Lessons learned. *The Journal of the Learning Sciences, 4,* 167-207.

[2] Graesser A. C., Person, N. K., & Harter, D. (2001). Teaching tactics and dialog in autotutor. *International Journal of Artificial Intelligence in Education, 12,* 12-23.

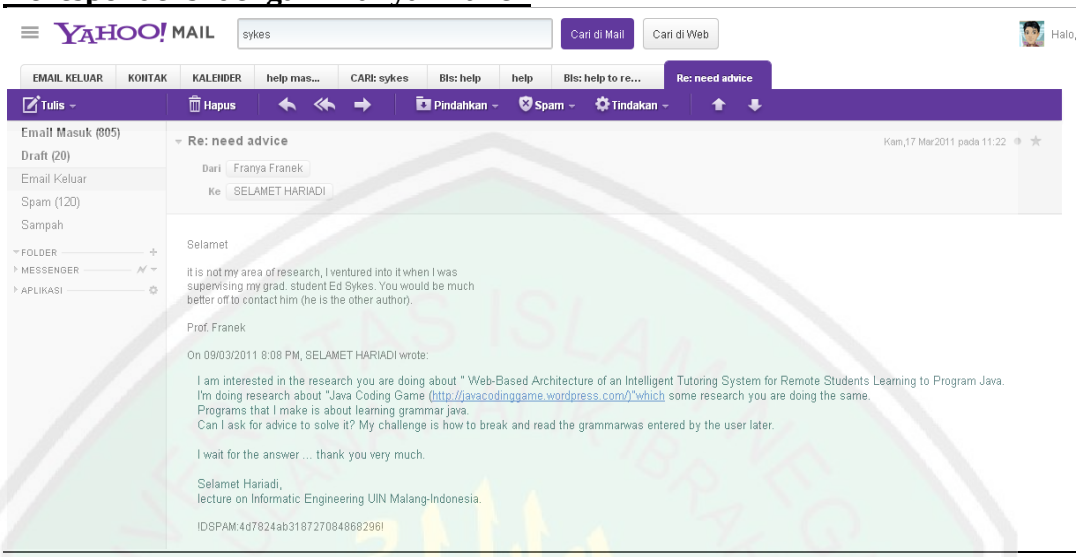
[3] Koedinger, K. R. (2001). Cognitive tutors. In K. D. Forbus & P. J. Feltovich (Eds.), *Smart machines in education* (pp. 145-167). Cambridge, MA: MIT Press.

[4] Scott, A. C., Clayton, J., E., & Gibson, E., L. (1991). *A Practical Guide to Knowledge Acquisition*. Menlo Park, CA: Addison-Wesley.

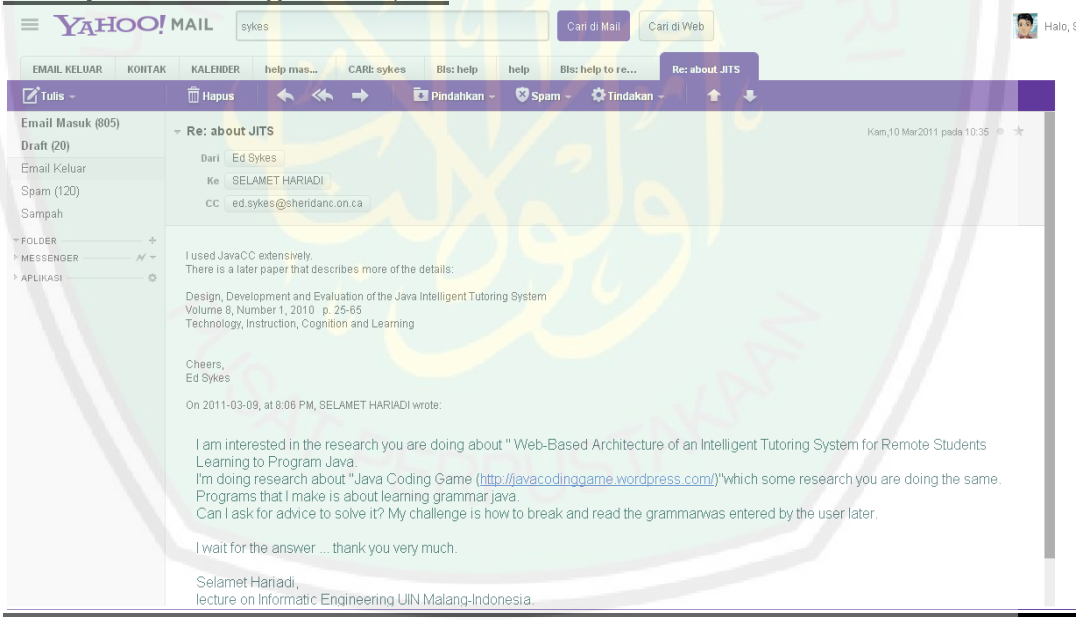
[5] Woolf, B., P., Beck, J., Eliot, C., & Stern, M. (2001). Growth and maturity of intelligent tutoring systems: A status report, In K. D. Forbus & P. J. Feltovich (Eds.), *Smart machines in education* (pp. 100-144). Cambridge, MA: MIT Press.

LAMPIRAN
KORESPONDENSI ONLINE

Korespondensi dengan Franya Franek

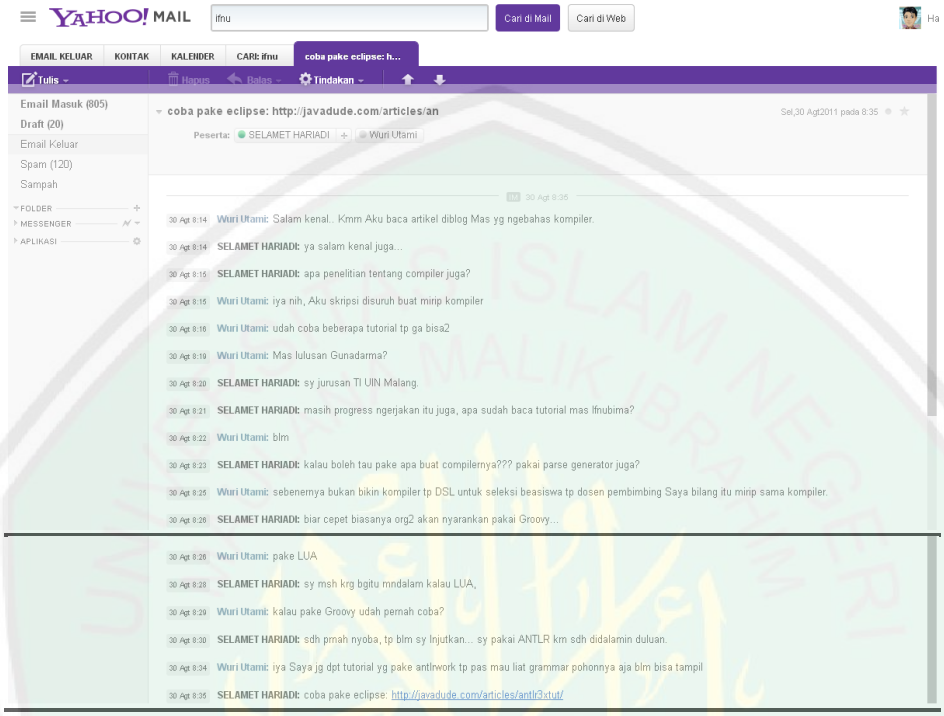


Korespondensi dengan Ed Sykes

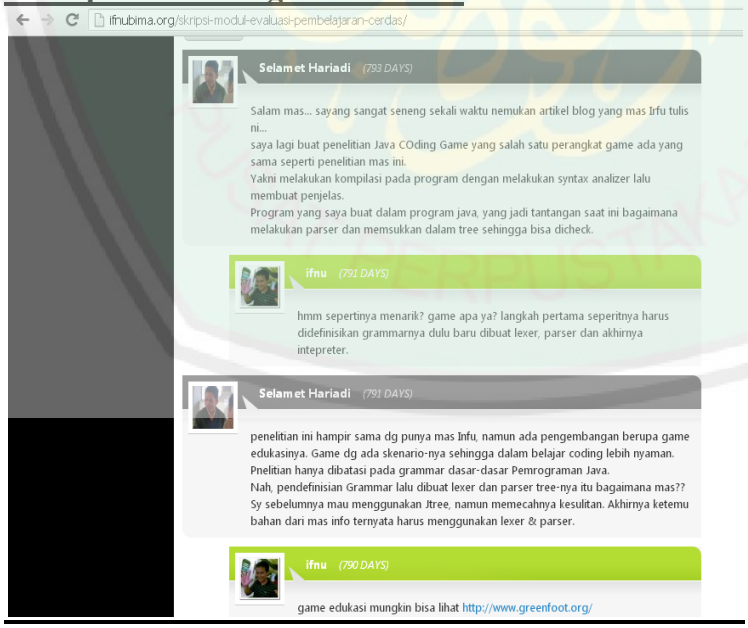


LAMPIRAN
KORESPONDENSI ONLINE

Korespondensi dengan Wuri Utami



Korespondensi dengan Ifnu Bima



LAMPIRAN
KORESPONDENSI ONLINE

Re: Re: [ASK] Error Handling Antlr

by ifnu Nov 01, 2012; 05:02pm :: Rate this Message: - Use ratings to moderate (?)

Reply | Print | View Threaded | Show Only this Message

Hmm, belum ketemu juga ya jawabanya? udah berapa bulan? :D #grin #troll

Pertama, perlu disclaimer dulu, Parser itu bukan perkara gampang. ANTLR ditujukan untuk menyederhanakan masalah dengan membuat bahasa yang ditujukan untuk parsing dokumen. Nah, kalau ada error biasanya parsernya ANTLR akan throw exception. Masalah kamu adalah gimana caranya km catch exception tersebut kemudian harus menjelaskan ke user apa yang salah. Nah sayangnya java.g yang ada di website ANTLR.org itu tidak mencantumkan sistem penjelas kesalahan yang bagus. Kamu sendiri yang harus menambahkanya ke dalam java.g

Saya dulu melakukan hack sedikit di grammar file-nya dengan menambahkan kode java di sana. Kode ini saya tulis satu per satu di bagian error handlingnya antlr, kalau parser / lexer menemukan kesalahan saya tambahkan kode di sana untuk mengecek.

Itu sekitar 5 tahun lalu, sudah lupa dan kodenya sudah gak ada :D

good luck buat skripsinya. Kalau sudah mentok lebih dari 1 semester sebaiknya segera ganti saja daripada gak lulus-lulus, buang-buang waktu kamu yang sangat berharga.

--
http://ifnubima.org/indo-java-podcast/
http://tanyajava.com/download/javadesktop
http://tanyajava.com/download/scrum
@ifnubima

regards

LAMPIRAN
KORESPONDENSI ONLINE

Korespondensi dengan Edwin Nataniel

Re: [ASK] Error Handling Antlr

by edwin.ugroups Oct 30, 2012; 10:03pm :: Rate this Message: - Use ratings to moderate (?)

Reply | Print | View Threaded | Show Only this Message

Hi SELAMET,

Cara kerja Antlr secara garis besar adalah:

1) Siapkan grammar file (biasanya dengan extension .g)
- Isinya : Rule rule untuk lexing, parsing, and tokenizing
- Tambahan: kadang bisa dimasukin Java code juga, khusus nya untuk handling exception/error messages
** NOTE: Feature terakhir untuk error messages di ANTLR agak limited, ga bisa terlalu user friendly (i.e.: for no-programmer) klo mau di customize

2) Di compile melalui antlr command-line
- Output: parser.java, lexer.java, token

3) Cara pakai
Pakai API/methods dari parser dan lexer java file yang di produced daripada antlr command-line

Kalau mau tahu Java nya benar atau salah, ada beberapa cara:

1) Does it throw an exception?
- Pas lagi parsing/tokenizing + running lexer rule, kalau ga salah ada 2 macam exception yang di thrown sama ANTLR (1 for each parsing and lexing)

2) Unit-test
- Habis di process oleh parser/lexer tanpa exception, biasanya bisa di simpan di dalam Tree structure,
- di Tree structure tersebut bapak bisa cek Node nya satu-persatu (i.e.: iterate the node) apakah isinya sesuai dengan yang bapak mau?

Ed
CHECK OUT MY App: http://nbaschedules.appspot.com
Web: http://www.edwinnathaniel.com
Twitter: http://www.twitter.com/enathaniel

From: SELAMET <met_hariadi@...>
To: jug-indonesia@...
Sent: Tuesday, October 30, 2012 1:46:51 AM

Re: Re: [ASK] Error Handling Antlr

by edwin.ugroups Nov 03, 2012; 04:06am :: Rate this Message: - Use ratings to moderate (?)

Reply | Print | View Threaded | Show Only this Message

Hi SELAMET,

Boleh di check disini cara kerja exception handling:

http://docs.oracle.com/javase/tutorial/essential/exceptions/

Ed
Web: http://www.edwinnathaniel.com
Twitter: http://www.twitter.com/enathaniel
App: http://nbaschedules.appspot.com

From: SELAMET <met_hariadi@...>
To: jug-indonesia@...
Sent: Friday, November 2, 2012 4:28:04 AM
Subject: [JUG-Indonesia] Re: [ASK] Error Handling Antlr

bisa tlg contohnya seperti apa?

sy pgn yg cepet aja untuk check-nya.

--- In jug-indonesia@..., Edwin Nathaniel <edwin.ugroups@...> wrote:

>
> Kan udah dijawab.... ada exception or not?
>
> Â
> Ed
> Web:Â http://www.edwinnathaniel.com

LAMPIRAN
KORESPONDENSI ONLINE

Korespondensi dengan Hendry Luk

Re: Re: [ASK] Error Handling Antlr

by Hendry Luk Nov 02, 2012; 12:52pm :: Rate this Message: - Use ratings to moderate (?)

Reply | Print | View Threaded | Show Only this Message

Di antlr, ada beberapa tempat buat detect errors.

Pertama, ada lexing error. Ini dah dihandle oleh antlr, dan bisa ditangkep dengan implement reportError method di java filenya, yang nge-catch RecognitionException. Di exception class ini ada semua informasi mengenai text yang salah beserta lokasinya (e.g. getInput, getCharPositionInLIne). Buat laporin errornya ke user ini simple. Biasanya gw tinggal menggunakan informasi dari exception ini buat nampilin line yang error dan kasih tanda panah di lokasi errornya.

Kedua, ada parser error. Soal detection, ini juga sama, serahin aja ke antlr dengan implement reportError. Tapi tergantung mo seberapa user-friendly error messagenya, terkadang lu mesti cast ke Exception turunan (e.g. MismatchedTokenException, MissingTokenException etc) supaya bisa ngasih suggestion yang lebih descriptive ke user buat memperbaiki error tersebut (e.g. syntax/keywords yg diharapkan).

Ketiga adalah logic error. Ini dah diluar antlr. Ada banyak pattern buat gimana menggunakan AST yg dihasilkan oleh antlr. Bisa pake tree-walkernya antlr, bisa juga bikin sendiri (e.g. pake visitor pattern). Either way, lu mesti bisa nulis error message yg tepat kalo AST yang dihasilkan antlr tidak sesuai dengan yg dikehendaki. Misalnya di contoh specific lu untuk memparse java code, maka mesti ngecek apakah nama method yg akan dipanggil sudah dideclarasikan pada class tersebut, atau apakah sebuah method dideclarasikan dua kali dengan signature yang sama. Ini adalah jenis2 error yang mesti diimplementasikan sendiri oleh tree-walker yang lu implement.

Mengenai pattern2 penggunaan antlr ini, ada banyak resources yang bisa lu cari di internet. Teknik semacem penggunaan visitor-pattern, gimana mengimplementasikan look-ahead (yang out-of-the box gak disupport oleh antlr), gimana menghandle context-sensitive, operator presedence, dan sebagainya. Pertama kali gw pake antlr, resource yg paling berguna adalah dari source-code Hibernate. Contek dari sana berbagai tricks dan patterns yg berguna untuk menggunakan antlr.

Mengenai error handling ini, patten yg digunakan di hibernate misalnya adalah dengan interface IErrorReporter. Jadi di parser/lexer lu, implementasi handleErrornya cuma sekedar nerusin doank ke ErrorReporter. ErrorReporter ini yg melakukan heavy-lifting gimana meng-intepretasikan error tersebut ke user.

Soal unit-test... Biasa ada 2 level.

Pertama unit-test buat parsernya. Ini cukup simple, tinggal ngasih string, dan evaluate AST yang dihasilkan apakah sesuai yang diharapkan. Kalo mo nge-test error handling, maka kasih string yg ngaco, dan ngetes keterangan error yang dihasilkan.
Kedua adalah unit-test buat walker nya. Mestinya sih lu input ASTnya. Kemudian ya sesuai dengan program apa yg lagi dibangun, tinggal diverify apakah behavior yg diharapkan sudah benar.

Rancang Bangun *Java Coding Game* menggunakan Metode *Generating Tree* pada *Context Free Grammar*

Selamet Hariadi (06550073) - Dosen Pembimbing: Fatchurrochman, M.Kom & Dr. Ahmad Barizi, M.A
Universitas Islam Negeri "Maulana Malik Ibrahim" Malang
Fakultas Sains & Teknologi Jurusan Teknik Informatika
Jalan Gajayana No. 50 Dinoyo – Malang
Email*: met_hariadi@yahoo.co.id* Blog*: selamethariadi.com*

Abstrak

Terinspirasi dari bahasa natural manusia, ilmuwan-ilmuwan ilmu komputer yang mengembangkan bahasa pemrograman, turut serta memberikan tata bahasa (pemrograman) secara formal. Tata bahasa ini diciptakan secara bebas konteks dan disebut CFG (Context Free Grammar). Hasilnya, dengan pendekatan formal ini, kompiler suatu bahasa pemrograman dapat dibuat lebih mudah dan menghindari ambiguitas ketika parsing bahasa tesebut. ANTLR membantu membuat grammar dengan menggunakan konsep Generating Tree menghasilkan konsep Domain Spesific Lenguage. GTGE merupakan Game Engine buatan Indonesia yang memudahkan bagi para pembuat game bersifat 2 dimensi. Dengan game yang sudah dihasilkan seperti Warlock yang di dalamnya ada Level Builder dapat membantu pembuatan halaman Level, mengatur tata letak, enemy, hingga waktu bermain dalam sebuah game lebih cepat.

Kata Kunci: context free grammar, ANTLR, grammar, GTGE, generating tree

1. Pendahuluan

Perkembangan aplikasi berbasis komputer mulai banyak diminati oleh khalayak ramai, hal ini dikarenakan lebih interaktif dan memudahkan dalam memakainya. Dibalik aplikasi program dan sistem yang kompleks terdapat banyak aturan program yang terangkai dalam kode-kode program. Kode-kode program inilah yang akan terlihat tampilan sistem dan fungsionalitas dari program dapat dipergunakan oleh pengguna program tersebut. Di dalam Al-Qur'an Allah berfirman dalam Surat Al-An'am [6] : 101 sebagaimana berikut:

بَدِيعُ ٱلسَّمَٰوَٰتِ وَٱلۡأَرۡضِۖ أَنَّىٰ يَكُونُ لَهُۥ وَلَدٞ وَلَمۡ تَكُن لَّهُۥ صَٰحِبَةٗۖ وَخَلَقَ كُلَّ شَيۡءٖۖ وَهُوَ بِكُلِّ شَيۡءٍ عَلِيمٞ ١٠١

Artinya: *Dia (Allah) Pencipta langit dan bumi. bagaimana Dia mempunyai anak padahal Dia tidak mempunyai isteri. Dia menciptakan segala sesuatu; dan Dia mengetahui segala sesuatu. (Al-An'am [6] : 101)*

Dalam Ayat diatas menunjukkan kuasa Allah yang menciptakan segala sesuatu, Allah adalah pemrogram kehidupan ini. Semua yang ada di manapun merupakan kekuasaan Allah serta hanya Allah yang mengetahui segala sesuatu. Dalam kaitannya dengan pemrograman yang dibutuhkan pemeran utama seorang pemrogram untuk pemrogramannya maka Allah adalah pemrogram kehidupan ini yang segala hal dijalankan atas kehendak Allah.

Banyak hal yang ditemui dalam kehidupan mahasiswa informatika, sistem informasi atau singkatnya yang berbasis di bidang komputer serta teknologi informasi adalah kurang kuatnya pemahaman akan *script coding* program. Padahal pemahaman yang mendalam terhadap *script coding* adalah yang paling tidak terpenting di dalam orang yang mendalami bidang berbasis IT. Hal ini dikarenakan dalam setiap sistem yang besar dalam sebuah program terdapat jaringan *script coding* yang saling berhubungan secara kompleks dalam menampilkan tampilan yang diinginkan pembuat program. Memang menjadi *programmer* adalah bukan pilihan yang harus dibuat pilihan wajib bagi yang memilih bidang IT sebagai pilihan hidupnya untuk berkontribusi untuk dunianya. Masih banyak pilihan konsentrasi bidang lainnya, seperti: desainer, analis sistem, manajer proyek, dan lain sebagainya jika dijelaskan lebih dalam akan terlihat lebih banyak. Namun, seperti kata salah seorang pakar IT di Indonesia yakni Romi Satria Wahono, kemampuan *script coding* patutlah menjadi hal yang terpenting dalam mereka yang bergelut di dunia berbasis IT, apalagi bagi mereka yang membuat dan merancang sistem berbasis IT.

Melihat media pembelajaran yang cukup sukses membantu dalam mengetik sepuluh jari seperti *typing master*, maka media pembelajaran berupa *Game* atau permainan dalam pengenalan hingga pemahaman aturan-aturan program dirasa perlu guna mempermudah *programmer* awam atau pun mereka yang bergelut di dunia IT. Media yang dirancang dan dibangun ini akan sangat berbeda dengan media pembelajaran untuk mengetik karena dengan sistem yang lebih kompleks.

2. Tinjauan Pustaka

2.1 Edukasi

Edukasi bisa diartikan sebagai rangkaian usaha yang bertujuan untuk mempengaruhi orang lain, mulai dari individu, kelompok, keluarga dan masyarakat luas. Saat ini perkembangan Edukasi mulai berkembang ke berbagai ranah bidang kehidupan sebagai sarana untuk menjadikan masyarakat lebih baik.

Umumnya orang mengartikan edukasi dari bahasa Inggris yakni *education*, dalam kamus besar bahasa Inggris *education* berarti pendidikan,

pendidikan berasal dari kata didik, atau mendidik yang berarti memelihara dan membentuk latihan.

Di Indonesia berdasarkan undang-undang (UU) nomor 2 tahun 1989 disebutkan bahwa pendidikan nasional bertujuan mencerdaskan kehidupan bangsa dan mengembangkan mannusia Indonesia seutuhnya, yaitu manusia yang beriman dan bertaqwa terhadap Tuhan Yang Maha Esa dan berbudi pekerti luhur, memiliki pengetahuan dan ketrampilan, kesehatan jasmani dan rohani, kepribadian yang mantap dan mandiri serta rasa tanggungjawab kemasyarakatan dan kebangsaan.

Dari landasan undang-undang ini edukasi tak hanya pada peningkatan pengetahuan namun juga keterampilan hingga kepribadian yang lebih baik yang mana menjadi tugas masyarakat bersama. Hal ini seperti usaha Pemerintah menjadikan masa anak-anak diasupi dengan edukasi untuk menjadikan masa remaja-nya dapat lebih berpengetahuan untuk menyongsong masa kehidupan di tengah pergaulan dunia internasional.

Proses mengajar juga penting dalam sebuah sistem pembelajaran. Menurut Novian Triwidia Jaya (2010 : 28) mengajat pada prinsipnya adalah mengkomunikasikan dan mengirimkan informasi dari pengajar ke pelajar. Penyaluran informasi berupa pengetahuan inilah yang menjadi hal penting dalam proses belajar dan mengajar.[2]

2.2 Game Engine

Permainan dalam kamus bahasa Inggris berarti *game* . Definisi *game* bisa diartikan sebagai permainan. Selain dampak positif dan negatif yang ditimbulkan, semua *game* elektronik yang dimainkan saat ini, baik itu versi *console*, HP atau PC masing-masing mempunyai elemen yang hampir sama. Sifat *game* edukasi, kadang ada pula yang memasukkan dalam gabungan *education* dan *entertainment*, yakni edukasi dan hiburan.

Game engine merupakan alat pendukung membuat *game*. Menurut Andi Taru (2010 : 15), GTGE (*Golden T game Engine*) merupakan game engine bebasis bahasa pemrograman java [5]. GTGE *Game* Engine dapat membantu kita mempermudah pengembangan *game* 2 Dimensi. Karena dengan *Game* Engine kita akan dengan mudah melakukan pembuatan *game.*

2.3 ANTLR

ANTLR digunakan dalam penelitian ini sebagai penyeleksi kebenaran tata bahasa pemrograman. ANTLR merupakan kepanjangan sebenarnya dari *ANother Tool for Language Recognition*. ANTLR adalah generator *Parser* yang dapat di gunakan untuk membaca, mengolah, melaksanakan, atau menerjemahkan teks terstruktur atau file biner [3].

2.4 Context Free Grammar

Menurut Bambang Hariyanto (2004 : 233) semula *Context Free Grammar* (CFG) ditemukan untuk membantu menspesifikasikan bahasa manusia dan ternyata sangat cocok untuk mendefinisikan bahasa komputer, memformulasikan pengertian parsing, menyederhanakan penerjemah bahasa komputer dan aplikasi-aplikasi pengolah string lainnya [1]. *Context Free Grammar* (CFG) adalah tata bahasa yang hampir sama tujuannya pada tata bahasa reguler seperti bahasa Inggris, bahasa Indonesia atau bahasa Arab yang menghasilkan untaian sebuah bahasa.

2.5 Generating Tree

Generation Tree juga disebut diagram *parsing*, Pohon Pembangkitan atau phrase marker (Hariyanto, 2004 : 240). Pada *Generation* atau *Generating Tree* ini berguna untuk memecah grammar dari bahasa pemrograman menjadi diturunkan lebih kecil, hal ini digunakan untuk mengecek kebenaran dengan yang diminta pada *Java Coding game* ini. *Generating Tree* digunakan dalam pembentukan *Grammar* di ANTLR dengan bahasa EBNF.

3. Analisis dan Perancangan Sistem

3.1 Skenario Game

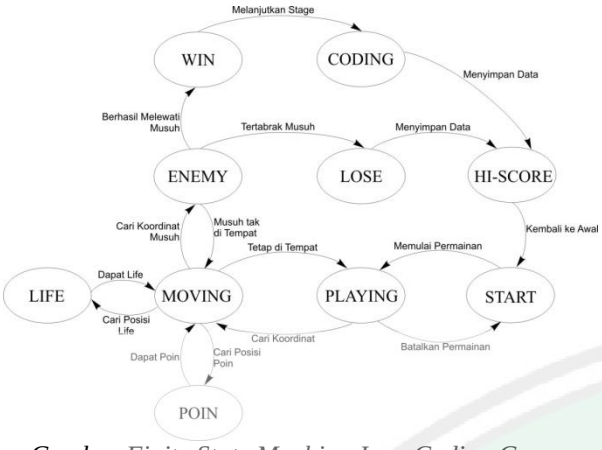
Musuh (*Enemy*) menyebar tantangan dan menyebar untuk menghalangi pemain dalam game mendapatkan poin, kunci-kunci, bonus hingga pintu *exit*. Pemain dalam game berusaha melewati musuh dengan menemukan kunci-kunci, tambahan poin, bunus untuk menemukan pintu *exit* setelah itu akan masuk pada halaman tantangan tentang kode-kode java untuk dipecahkan. Perjalanan game akan lebih cepat dengan menekan tombol *shift* untuk pilihan *Turbo Mode.*

Pemain dalam game akan mati jika menabrak musuh (*Enemy*) atau kehabisan waktu untuk melewati tantangan di tiap level. Permainan masih dapat dilanjutkan jika pemain dalam game masih memiliki nyawa, akan terjadi *game over* jika nyawa sudah habis yang kemudian akan masuk ke halaman *hi-score* game untuk mengisi nama jika *score* atau poin yang dikumpulkan masuk jajaran *top hi-score.*

Jika permainan disudahi karena adanya permintaan dari pemain menemekan tombol Q untuk keluar (*Quit*), maka pemain akan diarahkan ke halaman *hi-score* jika poin atau *score*-nya masuk jajaran top *hi-score*. Permainan juga dapat di-ulang kembali atau *restart* jika pemain menekan tombol R (*restart*).

3.2 *Finite State Machine* (FSM)

Menurut Iwan Setiawan (2006 : 1), *Finite State Machines* (FSM) adalah sebuah metodologi perancangan sistem kontrol yang menggambarkan tingkah laku atau prinsip kerja sistemdengan menggunakan tiga hal berikut: *State* (Keadaan), *Event* (kejadian) dan *Action* (aksi). FSM cukup banyak dipakai sebagai basis perancangan aplikasi yang mempunyai kontinuitas seperti *Game* [4].



Gambar Finite State Machine Java Coding Game

4. Hasil dan Pembahasan

4.1 Menu Game

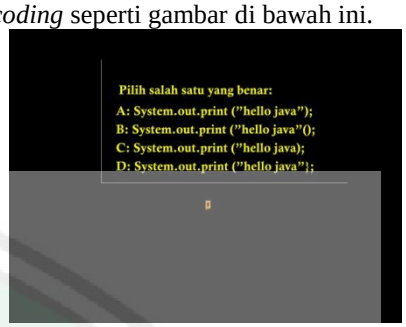
Game akan menampilkan beberapa pilihan tombol yang dapat dipilih oleh pemain dalam Game edukasi Java Coding Game.

Gambar Menu Game

Gambar di atas merupakan gambar untuk menu Game awal, dimana isi dari menu Game awal ini berbagai macam. Tombol-tombol tersebut didetailkan sebagai berikut.

a) Tombol *Start Game* : Sebagai pemicu untuk memasuki permainan. Pemain akan dibawa ke *level* satu setelah menekan tombol ini
b) Tombol *Instructions* :Berisikan instruksi *Game* dari mengenali pemain yang dijalankan, musuh hingga tombol cara menjalankan *Game*.
c) Tombol *Highscore*: Tombol ini untuk melihat hasil sepuluh skor tertinggi yang mampu diraih oleh pemain dalam *Game* edukasi *Java Coding Game*
d) Tombol *Sound* : Pemain dapat mematikan atau menghidupkan suara (*sound track*) yang sedang berjalan dalam *Java Coding Game*
e) Tombol *Level* : Pemain dapat memilih tombol level yang diinginkan sebagai tantangan kemampuan pemain. Hal ini karena tiap orang punya karakter kesukaan tersendiri dalam tantangan yang ditawarkan *Game*. Perlu diketahui untuk level *Easy* pemain yang dijalankan akan memiliki 4 nyawa.
f) Tombol *Credits* : Tombol ini digunakan sebagai halaman profil dan ucapan terima kasih untuk pihak yang membantu membuat *Game* ini
g) Tombol *Quit Game*: Tombol ini berada pada urutan terakhir dari list tombol yang ada. Tombol ini berfungsi sebagai pemicu untuk keluar dari program.

Dalam setiap level akan ada halaman *Game* yang hampir sama dengan gambar di atas, namun dengan rintangan yang berbeda. Setelah melewati stage di halaman *Game* tersebut dengan menemukan pintu *exit*, maka pemain dalam *Game* akan menuju ke stage tentang *coding* seperti gambar di bawah ini.



Gambar Coding Stage

Pada Gambar di atas yang merupaan *coding stage*, pemain dalam *Game* diminta memilih jawaban mana yang tepat dari pertanyaan yang diajukan.

4.2 Rule Grammar

Berikut ini contoh *coding Game* pada *Grammar* untuk penentukan "hello world" sebagai inputan dari pemain *Game*. Inputan tersebut pertama kali diinisialisasikan melalui *coding grammar* berikut ini.

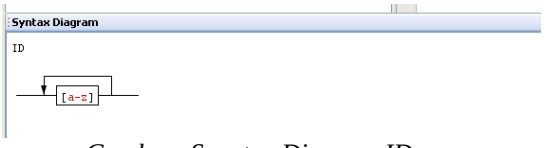
```
grammar Hello;
r  : 'hello' ID ;
ID : [a-z]+ ;
WS : [ \t\r\n]+ -> skip ;
```

Kode program: Grammar Hello

Dengan menggunakan plugin *ANTLRWorks* di Netbeans 7.3 akan ditemui tampilan *Syntax Diagram* untuk *grammar Hello* sebagaimana berikut ini.

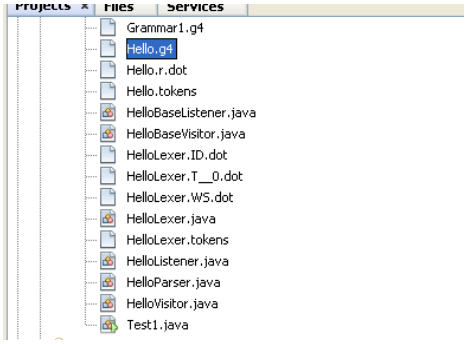
Gambar : Syntax Diagram Hello

Setiap baris di dalam *grammar* ini bisa dilehat di *Syntax Diagram* tentang alur kerjanya. Seperti Kode program di bawah ini tentang *Syntax Diagram* ID a-z.



Gambar: Syantax Diagram ID a-z

Setelah melakukan generating Code *Grammar* , maka akan ditemui berbagai macam class untuk mendukung proses selanjutnya. Berikut ini class yang dihasilkan dari Hello.g4



Gambar: Class yang dihasilkan dari Hello.g4

Untuk mengujinya dibutuhkan *class* tambahan, berikut ini potongan kode untuk menguji sementara *grammar* .

```
NTLRInputStream input = new
ANTLRInputStream(//System.in);
                              amasuk       );
        // create a lexer that feeds off
of input CharStream
        HelloLexer lexer = new
HelloLexer(input);
        // create a buffer of tokens
pulled from the lexer
        CommonTokenStream tokens = new
CommonTokenStream(lexer);
        // create a parser that feeds
off the tokens buffer
        HelloParser parser = new
HelloParser(tokens);
        ParseTree tree = parser.r(); //
begin parsing at init rule
        String s =
tree.toStringTree(parser);
        System.out.println(s);
```

Kode program: Kode Pengujian Grammar

4.3 Rekapitulasi Hasil Angket

Game ini juga dilakukan pengujian kepada kalangan terbatas untuk mengetahui kualitas *Game* secara umum untuk diluncurkan kepada kalangan yang lebih luas. Oelh karena itulah dilakukan proses angket yang mana pilihan dalam angket adalah gambaran pada kulitas *Game* setelah melakukan pemakaian oleh sebagian pengguna.

No	Uraian	SB	B	C	K
Mudahkah memainkan *Java Coding Game*?					
1	Konfigurasi tombol	3	6	1	0
2	Penyajian Persoalan	1	7	2	0
3	Permainan Keseluruhan	2	8	0	0
Bagaimana tanggapan mengenai *Java Coding Game*?					
4	Bentuk Tampilan	7	2	1	0
5	Pilihan Warna	1	8	1	0
6	Penggunaan Font	3	7	0	0
7	Penggunaan Bahasa	2	4	4	0
8	Sound	4	1	5	0
9	Tampilan Animasi	3	7	0	0
10	Ide / Konsep Cerita	4	5	1	0
Total Nilai		30	55	15	0

Tabel Hasil Angket

Dari hasil angket di atas didapat berbagai gambaran yang terjadi pada berbagai orang yang melakukan pengujian atau percobaan pada Java Coding Game. Hal ini terjadi karena banyak karakter yang tentunya berbeda pula dlam menyikapi sebuah game.

5. Kesimpulan

Dari penelitian dan implikasi uji coba penelitian ini, dapat disimpulkan bahwa penggunaan metode *Generating Tree* mempunyai kemudahan dalam inialisasi program untuk menghasilkan *grammar* yang baik untuk dapat dibaca pada pengujian yang menghasilkan pengecekan kebenaran *grammar* yang dimasukkan.

Dari hasil angket didapat 20 % yang melakukan percobaan kemudahan memainkan *Java Coding Game* secara keseluruhan sangat baik, untuk 80% menyatakan permainan secara keseluruhan Baik. Dari sini dapat disimpulkan bahwa penggunakan *Java Coding Game* sebagai game edukasi mendapat respon yang baik. Sedang untuk ide atau konsep cerita dalam *Java Coding Game* mungkin berbeda pada *Game* pada umumnya karena menyisipkan konsep edukasi kode-kode java. Dari hasil angket sebanyak 10 % memberikan penilaian cukup, sedangkan 50 % dari hasil angket memberikan penilaian baik dan 40 % memberikan penilaian sangat baik.

6. Daftar Pustaka

[1] Hariyanto, Bambang. 2004. *Teori Bahasa, Otomata dan Komputasi serta terapannya.* Bandung: Informatika.

[2] Jaya. Novian T. 2010. *Hypno Teaching Bukan Sekedar Mengajar*. Bekasi: D-Brain

[3] Parr, Terence. 2012. *The Definitive ANTLR 4 Reference*. Texas, USA: The Pragmatic Bookshelf.

[4] Setiawan, Iwan. 2006. Perancangan Software Embeded System Berbasis FSM. *Laporan* Tidak Diterbitkan. Semarang: Elektro FT Universitas Diponegoro

[5] Taru, Andi NNW. 2010. *Pemrograman Game dengan Java dan GTGE*. Yogyakarta: Penerbit Andi.