Language Translation Using PCCTS and C++

A Reference Guide

Terence John Parr
MageLang Institute

Automata Publishing Company, San Jose, CA 95129

Cover design: Evolved Design
Interior design: Anu Sethuram
Editing: Martha Cover

Published by *Automata* Publishing Company

In addition to this title, the following HDL titles and software are also available from Automata Publishing Company:

1. ***Digital Design and Synthesis with Verilog HDL***
2. ***Digital Design and Synthesis with VHDL***

Please see the ***order form*** on the last page of the book

Automata Publishing Company
1072 De Anza Blvd.,
San Jose, CA 95129, USA
Phone: 408-255-0705
email: info @apco.com

Printed in the United States of America
10 9 8 7 6 5 4 3 2 1

ISBN 0-9627488-5-4

This is Tom’s fault.

A Completely Serious, No-Nonsense, Startlingly-Accurate Autobiography

Terence John Parr was born in Los Angeles, California, USA in the year of the dragon on August 17, 1964 during the week of the Tonkin Gulf Crisis, which eventually led us into the Vietnam Conflict; coincidence? Terence's main hobbies in California were drooling, covering his body in mud, and screaming at the top of his lungs.

In 1970, Terence moved to Colorado Springs, Colorado with his family in search of better mud and less smog. His formal education began in a Catholic grade school where he became intimately familiar with penguins and other birds of prey. Terence eventually escaped private school to attend public junior high only to return to the private sector—attending Fountain Valley School for the "education" only a prep school can provide. After being turned down by every college he applied to, Terence begged his way into Purdue University's School of Humanities. Much to the surprise of his high school's faculty and the general populace, Terence graduated in 1987 from Purdue with a bachelor's degree in computer science.

After contemplating an existence where he had to get up and go to work, Terence quickly applied to graduate school at Purdue University's School of Electrical Engineering. By sheer tenacity, he was accepted and then promptly ran off to Paris, France after only one semester of graduate work. Terence returned to Purdue in the Fall of 1988, eventually finishing up his master's degree in May 1990 despite his best efforts. Hank Dietz served as major professor and supervised Terence's master's thesis.

A short stint with the folks in blue suits during the summer of 1990, convinced Terence to begin his Ph.D.; again, Hank Dietz was his advisor. He passed the Ph.D. qualifier exam in January of 1991, stunning the local academic community. After three years of course work, research, and general fooling around, Terence finished writing his doctoral dissertation and defended it against a small horde of professors and students on July 1, 1993.

After completing a year of penance with Paul Woodward and Matt O'Keefe at the Army High Performance Computing Research Center at the University of MN as a postdoctoral slave, Terence formed Parr Research Corporation and leapt into the unknown on August 1, 1994.

The Java programming language started its inexorable climb to stardom in early 1995. Terence entered the mad rush of Java startups in late 1995, forming MageLang Institute (`www.MageLang.com`) with Tom Burns and Mel Berman, in order to provide exceptional language training and further the cause of Java. Terence still maintains PCCTS while working 26 hours a day at MageLang and is allegedly having a pretty good time.

Table of Contents

Tables

Foreword

A few years ago, I implemented a programming language called NewtonScript(tm)[1], the application development language for the Newton(R) operating system. You may not have heard of NewtonScript, but you've probably heard of the tool I used to implement it: a crusty old thing called YACC.

YACC--like the C language, Huffman coding, and the QWERTY keyboard--is an example of a standard engineering tool that is standard because it was the first "80% solution". YACC opened up parsing to the average programmer. Writing a parser for a "little language" using YACC was vastly simpler than writing one by hand, which made YACC quite successful. In fact, it was so successful, progress on alternative parsing tools just about stopped.

Not everybody adopted YACC, of course. There were those who needed something better. A lot of serious compiler hackers stuck with hand-coded LL parsers, to get maximum power and flexibility. In many cases, they had to, because languages got more and more complicated--LALR just wasn't good enough without lots of weird hacks. Of course, these people had to forego the advantages of using a parser generator.

So if your language is simple, you use YACC. If your language is too complex, or if you want good error recovery, or if performance is critical, you write a parser from scratch. This has been the status quo for about 20 years.

Terence Parr and PCCTS have the potential to jolt us out of this situation. First, Terence pursued and formalized a new parsing strategy, called predicated LL(k), that combines the robustness and intelligibility of LL with the generality of LALR. Second, he implemented a parser generator, called ANTLR, that makes this power easy to use. Even the dedicated hand-coders may change their minds after a close look at this stuff. Finally, for those situations where you need to traverse a parse tree (and who doesn't?), SORCERER applies the ANTLR philosophy to that problem.

---

1. NewtonScript is a trademark and Newton is a registered trademark of Apple Computer, Inc.

The result is a tool set that I think deserves to take over from YACC and LEX as the default answer to any parsing problem. And as Terence and others point out, a lot of problems are parsing problems.

Finally, let me mention that PCCTS is a tool with a face. Although it's in the public domain, it's actively supported by the tireless Terence Parr, as well as the large and helpful community of users who hang out on comp.compilers.tools.pccts. This book will help the PCCTS community to grow and prosper, so that one day predicated LL(k) will rule, YACC will be relegated to the history books, and Terence will finally achieve his goal of world domination.

Just kidding about that last part.

Walter Smith
Palo Alto, California
January 1996

Preface

I like tools—always have. This is primarily because I'm fundamentally lazy and would much rather work on something that makes others productive rather than actually having to do anything useful myself. For example, as a child, my parents forced me to cut the lawns on our property. I spent hours trying to get the lawn mower to cut the lawn automatically rather than simply firing up the mower and walking around the lawn. This philosophy has followed me into adult life and eventually led to my guiding principle:

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

This is pretty much what has happened to me with regard to language recognition and translation. Towards the end of my undergraduate studies at Purdue, I was working for a robotics company for which I was developing an interpreter/compiler for a language called KAREL. This project was fun the first time (I inadvertently erased the whole thing); the second time, however, I kept thinking "I don't understand YACC. Isn't there a way to automate what I build by hand?" This thought kept rolling around in the back of my head even after I had started EE graduate school to pursue neural net research (my topic was going to be "Can I replace my lazy brain with a neural net possessing the intelligence of a sea slug without anybody noticing?"). As an aside, I decided to take a course on language tool building taught by Hank Dietz.

The parser generator ANTLR eventually arose from the ashes of my course project with Hank and I dropped neural nets in favor of ANTLR as my thesis project. This initial version of ANTLR was pretty slick because it was a shorthand for what I'd build by hand, but at that time ANTLR could only generate LL(1) parsers. Unfortunately, there were many such tools and unless a tool was developed with parsing strength equal to or superior to YACC's, nothing would displace YACC as the *de facto* standard for parser generation; although, I was mainly concerned with making myself more efficient at the time and had no World-domination aspirations.

ANTLR currently has three things that make it generate strong parsers: *(i)* k>1 lookahead, *(ii)* semantic predicates (the ability to have semantic information direct the parse), and *(iii)* syntactic predicates (selective backtracking). Using more than a single symbol of lookahead has always been desirable, but is exponentially complex in time and space; therefore, I decided to change the definition of k>1 lookahead and *voila*: LL(k) became possible. (That's how I escaped Purdue with my Ph.D. before anyone got

wise). The fundamentals of semantic and syntactic predicates are not my ideas, but, together with Russell Quong at Purdue, we substantially augmented these predicates to make them truly useful. These capabilities can be shown in theory and in practice to make ANTLR parsers stronger than YACC's pure LALR(1) parsers (our tricks could easily be added to an LALR(1) parser generator, however). ANTLR also happens to be a flexible and easy-to-use tool and, consequently, ANTLR has become popular.

Near the end of my Ph.D., I started helping out some folks who wanted to build a FORTRAN translator at the Army High Performance Computer Research center at the University of Minnesota. I used ANTLR to recognize their FORTRAN subset and built trees that I later traversed with a number of (extremely similar) tree-walking routines. After building one too many of these tree walkers, I thought "OK, I'm bored. Why can't I make a tool that builds tree walkers?" Such a tool would parse trees instead of text, but would be basically the same as ANTLR. SORCERER was born. Building language translators became much easier because of the ANTLR/SORCERER combination.

The one weak part of these tools has always been their documentation. This book is an attempt to rectify this appalling situation and replaces the series of disjointed release notes for ANTLR, DLG (our scanner generator), and SORCERER—the tools of the Purdue Compiler Construction Tool Set, PCCTS. I've also included Tom Moog's wonderful notes for the newbie as an appendix.

Giving credit to everyone who has significantly aided this project would be impossible, but here is a good guess: Will Cohen and Hank Dietz were coauthors of the original PCCTS as a whole. Russell Quong has been my partner in research for many years and is a coauthor of ANTLR. Gary Funck and Aaron Sawdey are coauthors of SORCERER. Ariel Tamches spent a week of his Christmas vacation in the wilds of Minnesota helping with the C++ output.

Sumana Srinivasan, Mike Monegan, and Steve Naroff of NeXT, Inc., provided extensive help in the definition of the ANTLR C++ output; Sumana also developed the C++ grammar that became the basis for the C++ grammar available for PCCTS. Thom Wood and Randy Helzerman both influenced the C++ output. Randy Helzerman has been a relentless supporter of PCCTS since it was forced upon him. Steve Robenalt pushed through the comp.compilers.tools.pccts newsgroup and wrote the initial FAQ. Peter Dahl, then Ph.D. candidate, and Professor Matt O'Keefe (both at the University of Minnesota) tested versions of ANTLR extensively. Dana Hoggatt (Micro Data Base Systems, Inc.) and Ed Harfmann tested 1.00 heavily. Anthony Green at Visible Decisions, John Hall at Worcester Polytechnic Institute, Devin Hooker at Ellery Systems, Kenneth D. Weinert at Information Handling Services, Steve Hite, Roy Levow at Florida Atlantic University, David Seidel, and Loring Craymer at JPL have been faithful beta testers of PCCTS. Gary Frederick manages the mailing list and offers many suggestions. Scott Haney at Lawrence Livermore National Laboratory developed the Macintosh MPW port.

I thank the planning group for the first annual PCCTS workshop sponsored by Parr Research Corporation and held at NeXT July 25 and 26, 1994: Gary Funck, Steve Robenalt, and Ivan Kissiov. The PCCTS '95 workshop group included Gary Funck, Steve Robenalt, and John D. Mitchell. The following people provided reviews of the initial release of this book (in the order their reviews arrived): Scott Stanchfield, Dan FitzPatrick, Tuan Doan, Kris Kelley, Alistair G. Crooks, John Mitchell, Chiiwen Liou, Jim Coker, Asgeir Olafsson, Glenn Lewis, and Michael Richter (who provided a HUGE number of suggestions). A multitude of PCCTS users have helped refine ANTLR with their suggestions; I apologize for not being able to mention everyone here who has supported the PCCTS project. The following people are mentioned so they will buy a copy of this book: Marjorie Kalman, Jennifer Wilson, Richard Raitt, Jeff Johnson, Cris DuBord, Paul Stahura, Jim Beaver, Tom Burns, Brett Miller, Rick Guptill, Jim Schwarz, Ephram Ajaji, Kurt Fickie, Tim Rohaly, Mike Beranek, Thierry Beauvilain, Mike Hofflinger, Russell "Creature" Cattelan, Kevin "Pugsley" Edgar, Steve Anderson, Paul Woodward, Gary Lutchansky, and Mark Gruenberg.

Bug reports and general words of encouragement are welcome. Please send mail to

```
parrt@parr-research.com
```

You may also wish to visit our newsgroup

```
comp.compilers.tools.pccts
```

or the ftp site:

```
ftp://ftp.parr-research.com/pub/pccts/
```

All of the tools in PCCTS are public domain. As such, there is no guarantee that the software is useful, will do what you want, or will behave as you expect. There are most certainly bugs still lurking in the code and there are probably errors in this book. I hope the benefits of the tools will outweigh any inconvenience in using them.

Terence John Parr

`parrt@MageLang.com`	(general chatting, social commentary)
`parrt@parr-research.com`	(bug reports, PCCTS questions, etc...)
`www.MageLang.com`	(company website)
`www.parr-research.com/~parrt`	(personal web site)

Irreverent in San Francisco, California
August 1996

1 Introduction

Computer language translation has become a common task. While compilers and tools for traditional computer languages (such as C, C++, FORTRAN, SMALLTALK or Java) are still being built, their number is dwarfed by the thousands of mini-languages for which recognizers and translators are being developed. Programmers construct translators for database formats, graphical data files (*e.g.*, SGI Inventor, AutoCAD), text processing files (*e.g.*, HTML, SGML), and application command-interpreters (*e.g.*, SQL, EMACS); even physicists must write code to read in the initial conditions for their finite-element computations.

Many programmers build recognizers (*i.e.*, parsers) and translators by hand. They write a recursive-descent parser that recognizes the input and either generates output directly, if the translation is simple enough to allow this, or builds an intermediate representation of the input for later translation, if the translation is complicated. Generally, some form of tree data-structure is used as an intermediate representation in this case (e.g., the input "3+4" can be conveniently represented by a tree with "+" at the root and "3" and "4" as leaves). In order to manipulate or generate output from a tree, the programmer is again confronted with a recognition problem—that of matching tree templates against an input tree. As an input tree is traversed, each subtree must be recognized in order to determine which translation action to execute.

Many language tools aid in translator construction and can be broadly divided into either parser generator or the translator generator.

- A parser generator is a program that accepts a grammatical language description and generates a parser that recognizes sentences in that language.

- A translator generator is a tool that accepts a grammatical language description along with some form of translation specification and generates a program to recognize and translate sentences in that language.

This book is a reference guide for the parser generator ANTLR, ANother Tool for Language Recognition, and the tree-parser generator SORCERER, which is suited to source-to-source translation. SORCERER does not fit into the translator generator category perfectly because it translates trees, whereas the typical translator generator can only be used to translate text directly, thus hiding any intermediate steps or data structures from the programmer. The ANTLR and SORCERER team more closely supports what programmers would build by hand. Specifically, ANTLR recognizes textual input and generates an intermediate form tree, which can be manipulated by a SORCERER-generated tree-walker; however, both tools can be used independently.

While every tool has its strengths and weaknesses, any evaluation must boil down to this: Programmers want to use tools that employ mechanisms they understand, that are sufficiently powerful to solve their problem, that are flexible, that automate tedious tasks, and that generate output that is easily folded into their application. Most language tools fail one or many of these criteria. Consequently, parsers and translators are still often written by hand. ANTLR and SORCERER have become popular because they were written specifically with these issues in mind.

About this book

This book is intended as a reference manual not a textbook or how-to book on language translation. Nonetheless, this book is valuable to any scientist, engineer, or programmer who has to translate, evaluate, interpret, manipulate or otherwise examine data or language statements of any kind. ANTLR and SORCERER (the two main components of PCCTS) were designed to be usable by people who are not language experts. Indeed, we are aware of two biologists doing biochemical pattern recognition with PCCTS.

For those already familiar with PCCTS, Chapters “ANTLR Reference” on page 81, “SORCERER Reference” on page 161, and “Templates and Quick Reference Guide” on page 217 will be the most useful. Those familiar with other language tools should skim “Chapter 2 - A Tutorial” and then read the ANTLR and SORCERER reference chapters; Chapter “Templates and Quick Reference Guide” on page 217, the Section on ANTLR on page 30, and the Section on SORCERER on page 31 will also be of interest as they summarize the behavior and flavor of the tools. Persons unfamiliar with languages, parsers, and language tools should carefully read “Exactly 1800 Words On Languages and Parsing” on page 25 and “A Tutorial” on page 41; they should finish up by reading the reference chapters on ANTLR and SORCERER.

We assume that you have a working knowledge of C++ or C. Any knowledge of grammars or language tools is extremely helpful.

Exactly 1800 Words On Languages and Parsing

We give only a taste of language theory here and in a very loose fashion. However, it should give you enough information and define enough terms to get you through the rest of the book.

In the Spring of 1983, as first year computer science students at Purdue University, we were assigned the problem of recognizing arithmetic expressions, which could include nested parentheses. We were given a specification that described what the expressions looked like and were asked to produce a Pascal program that recognized such expressions. The specification looked something like

```
expr-> factor
factor-> term ( "+" term )*
term-> atom ( "*" atom )*
atom-> "(" expr ")"
atom-> INTEGER
atom-> IDENTIFIER
```

where `INTEGER` and `IDENTIFIER` were shorthands for strings of digits and strings of letters, respectively; "`( "+" term )*`" indicated that zero or more "`+" term` sequences could be seen. [*Or did we get term and factor mixed up?*]. The rules described the structure of small pieces of the expression language. For example,

```
term-> atom ( "*" atom )*
```

was read by saying, "a `term` is an `atom` followed by zero or more '`"*" atom`' sequences." We remember thinking what a marvelously precise means of describing an infinitely large set of input strings.

We decided that our program would have one function to recognize each rule in the grammar to keep things nice and neat. In this manner, references to rules would become, possibly recursive, procedure calls in our program. References to actual input strings such as "`(`" and `INTEGER` were all hard coded to eat white space and look for the particular string. This got to be repetitive and so we decided to factor out the common operations among all input string matching code. Further, it seemed easier to treat input strings as single "words" when trying to match the grammatical structure of the expressions. We eventually came up with a function called `getword` that returned an integer describing what input vocabulary word was found. It also made sense to assume that some variable would always hold the

next word to be matched; that is, after matching a word, the variable would be set to the result of calling `getword` again.

With the benefit of our current knowledge, now we would say that we were provided with a *grammar* consisting of a set of *rules* that specified the set of all possible sentences in the expression *language*; that is, the grammar specified the *syntax* of the language. Rules with more than one alternative were considered to have multiple *productions*. Our program was a *parser* that made calls to a *lexical analyzer* or *scanner* (our `getword` function) that broke up the input character stream into *vocabulary* symbols, or *tokens*. The program we built was a classic example of a *recursive-descent parser*. A generic term for this type of parsing is *top-down* because when you look at the parse tree, the parse starts at the top (the start symbol) and works its way down the tree. Recursive-descent parsers are a set of mutually recursive procedures that normally use a single symbol of *lookahead* to make parsing decisions. For example, rule `atom` could be encoded in C as

```
int atom()
{
    // use lookahead to decide which alternative applies
    switch ( current_token ) {
        case LPAREN :// -> "(" expr ")"
            current_token = getword();
            expr();
            if ( current_token != RPAREN ) error-clause;
            current_token = getword();
            break;
        case INTEGER :// -> INTEGER
            current_token = getword();
            break;
        case IDENTIFIER :// -> IDENTIFIER
            current_token = getword();
            break;
        default :
            error-clause (missing LPAREN, INTEGER, or IDENTIFIER)
    }
}
```

where the variable `current_token` is the lookahead symbol.

Parsers that follow this simple formula can be classified as LL(1), which is a shorthand indicating that the input is matched from left-to-right (as opposed to backwards) and that parsing decisions are made on the left edge of alternative productions with 1 symbol of lookahead. This amounts to saying that LL(1) parsers must predict which alternative production will be successfully matched by examining only the set of tokens that can be matched first by each production. We loosely define this set of tokens that predicts alternatives to be the *lookahead set*.

Normally, this is set of tokens that can be matched first by a production *p* and is called *FIRST(p)*; *e.g.*,

```
FIRST("(" expr ")")
```

is the singleton set {"("}. Occasionally, the *FOLLOW* set is used to predict alternatives. FOLLOW(*r*) is the set of all tokens that can be matched following references to rule *r*. For example, given the grammar

```
rule             -> optional_ID SEMICOLON
optional_ID      -> IDENTIFIER
optional_ID      ->
```

FOLLOW(`optional_ID`) is { `SEMICOLON` } because `SEMICOLON` follows the reference to `optional_ID`. The lookahead set for the empty production is defined to be the FOLLOW of the invoking rule. Therefore, `SEMICOLON` predicts the empty production of `optional_ID`.

When the lookahead sets from alternative productions are not disjoint, we say that the parsing decision is *nondeterministic* or *ambiguous*. In other words, there is at least one token that predicts more than one alternative. Most of the time, this is a bad thing.

LL(1) parsers may be generalized to LL(k) for k>1. For example, the following grammar is ambiguous upon token `A`:

```
a -> A B C
a -> A D E
```

where `A`, `B`, `C`, `D`, and `E` are some vocabulary tokens. Because both productions have a common prefix of A, an LL(1) parser could not determine which production was going to successfully match. However, if the parser could see ahead to both the `A` and what followed `A` on the input stream, the parser could determine which production was going to match. An LL(2) parser is such a creature; hence, rule `a` is unambiguous in the LL(2) sense. A grammar for which a deterministic LL(k) parser can be built is LL(k). A language for which an LL(k) grammar exists is LL(k).

Because recursive-descent parsers are just piles of code, more sophisticated predictions can be made than simple lookahead buffer comparisons. For example, what if two productions are exactly alike syntactically, but are *semantically* different? What if the productions have different meanings (usually depending upon context or other information)? Consider the following grammar:

```
element -> ID "(" expression_list ")"     // array reference
element -> ID "(" expression_list ")"     // procedure call
```

It is perfectly reasonable to separate these two cases because while they look the same, array references and procedure calls are very different semantically. The definition of the `ID` must

be consulted to determine which production to match. In a hand-built parser, you could do this:

```
element()
{
    if ( current_token == ID && isarray(current_text) ) {
        match an array reference;
    }
    else if ( current_token == ID && isprocedure(current_text) ) {
        match a procedure call;
    }
    else error;
}
```

where `isarray(current_text)` is some function you have defined that returns true if the `ID` is previously defined as an array; `isprocedure` would be defined similarly. We call such a parser a predicated LL(k) parser, *pred-LL(k)*, because at least one parsing decision was predicated upon information not available to a pure LL(k) parser. The forms `isarray(current_text)` and `isprocedure(current_text)` are considered *semantic predicates*. We could modify the grammar as follows:

```
element -> <<isarray(current_text)>>? ID "(" expression_list ")"
element -> <<isprocedure(current_text)>>? ID "(" expression_list ")"
```

where `<<...>>?` is a semantic predicate in ANTLR notation.

Pred-LL(k) parsing covers another type of predicated parsing decision. Consider the following grammar:

```
a -> (A)* B
a -> (A)* C
```

No matter how large we make the k of LL(k), a sequence of k+1 `A`'s could always be presented to the parser, and the parser could not see past the `A`'s to the `B` or `C`. This grammar then is non-LL(k) for any fixed value of k. Because there are real grammars that might have constructs requiring arbitrary lookahead, we introduced another type of predicate called *syntactic predicates.* A syntactic predicate specifies a grammar fragment that uniquely predicts the associated production. The grammar could be modified as follows:

```
a -> ( (A)* B )? (A)* B
a -> (A)* C
```

which indicates that, to predict the first production, zero or more `A`'s must be seen followed by a `B`. If this syntactic predicate fails, the second production will be attempted by default; hence, no predicate is required at its left edge. Clearly, this ability to scan arbitrarily ahead, renders the class of pred-LL(k) languages much larger than the class of LL(k) languages.

Bottom-up Parsers

And now, for something completely different: a bit about the class of languages and parsers called *LR(k)*. LR(k) parsers are considered *bottom-up* parsers because they try to match the leaves of the parse tree as they work their way up the parse tree towards the start symbol at the root. A simple way to illustrate LR parsing is to consider a simple language as described the following grammar.

```
a -> A B C
a -> A B D
```

Loosely speaking, an LR-based parser consumes input symbols until it finds a viable complete production; for the purposes of this discussion, all productions are viable. Input token A would be tested against both productions. Since A matches neither completely, another input token would be consumed, and AB would be compared against the productions. Again, a complete right-hand-side would not be matched. The next input symbol would be consumed, say token D. At this point, ABD matches the second right-hand-side and the parser would report that it had found input for rule a. The process of consuming input is called *shifting*, and the process of matching complete right-hand-sides is called *reducing.* (The right-hand-side is reduced to the left-hand-side.) In our example, no lookahead is required to determine that a valid sentence was found because the entire production can be seen before making a decision. Therefore, this grammar is LR(0).

LR(k) recognizers (and their variants such as LALR(k)) are stronger than LL(k) recognizers because the LR strategy uses more context information. For an LR parser, the context consists of all grammar productions consistent with the previously seen input. This context often includes several “pending” grammar productions. Intuitively, an LR(k) parser attempts to match multiple productions at the same time and postpones making a decision until sufficient input has been seen. In contrast, the context for an LL parser is restricted to the sequence of previously matched productions and the position within the current grammar production being matched. An LL(k) parser must make decisions about which production to match without having seen any portion of the pending productions—it has access to less context information. Hence, LL(k) parsers rely heavily on lookahead. We note that our LR(0) grammar is LL(3) as a case in point.

On the other hand, our pred-LL(k) parsers are stronger than LR(k) parsers for two reasons. First, semantic predicates may be used to parse context sensitive languages. Second, pred-LL(k) parsers have access to arbitrary lookahead. Further, embedding actions in an LR grammar can introduce ambiguities, thus reducing the strength of LR.

ANTLR

ANTLR constructs human-readable recursive-descent parsers in C or C++ from pred-LL(k) grammars, namely LL(k) grammars, for k>1 that support predicates.

Predicates allow arbitrary semantic and syntactic context to direct the parse in a systematic way. As a result, ANTLR can generate parsers for many context-sensitive languages and many non-LL(k)/LR(k) context-free languages. Semantic predicates indicate the semantic validity of applying a production; syntactic predicates are grammar fragments that describe a syntactic context that must be satisfied before recognizing an associated production. In practice, many ANTLR users report that developing a pred-LL(k) grammar is easier than developing the corresponding LR(1) grammar.

In addition to a strong parsing strategy, ANTLR has many features that make it more programmer-friendly than the majority of LR/LALR and LL parser generators.

- ANTLR integrates the specification of lexical and syntactic analysis. A separate lexical specification is unnecessary because lexical regular expressions (token descriptions) can be placed in double-quotes and used as normal token references in an ANTLR grammar.
- ANTLR accepts grammar constructs in Extended Backus-Naur Form (EBNF) notation.
- ANTLR provides facilities for automatic abstract syntax tree construction.
- ANTLR generates recursive-descent parsers in C/C++ so that there is a clear correspondence between the grammar specification and the ANTLR output. Consequently, it is relatively easy for non-parsing experts to design and debug an ANTLR grammar.
- ANTLR has both automatic and manual facilities for error recovery and reporting. The automatic mechanism is simple and effective for many parsing situations; the manual mechanism called “parser exception handling” simplifies development of high-quality error handling.
- ANTLR allows each grammar rule to have parameters and return values, facilitating attribute passing during the parse. Because ANTLR converts each rule to a C/C++ function in a recursive descent parser, a rule parameter is simply a function parameter. Additionally, ANTLR rules can have multiple return values.
- ANTLR has numerous other features that make it a product rather than a research project. ANTLR itself is written in highly portable C; its output can be debugged with existing source-level debuggers and is easily integrated into programmers’ applications.

Ultimately, the true test of a language tool’s usefulness lies with the vast industrial programmer community. ANTLR is widely used in the commercial and academic communities. Thousands of people in virtually all industrialized nations have acquired the software since the original 1.00 release in 1992. Several universities currently teach courses with ANTLR. Many commercial programs use ANTLR.

For example NeXT, Inc. has completed and is testing a unified C/Objective-C/C++ compiler using an ANTLR grammar that was derived directly from the June 1993 ANSI X3J16 C++ grammar. (Measurements show that this ANTLR parser is about 20% slower, in terms of pure parsing speed, than a hand-built recursive-descent parser that parses only C/Objective-C, but not C++. The C++ grammar available for ANTLR was developed using the NeXT grammar as a guide.) C++ has been traditionally difficult for other LL(1) tools and LR(1)-based tools such as YACC. YACC grammars for C++ are extremely fragile with regard to action placement; i.e., the insertion of an action can introduce conflicts into the C++ grammar. In contrast, ANTLR grammars are insensitive to action placement because of their LL(k) nature.

The reference guide for ANTLR begins on page 81.

SORCERER

Despite the sophistication of code-generator generators and source-to-source translator generators (such as attribute grammar based tools), programmers often choose to build tree parsers by hand to solve source translation problems. In many cases, a programmer has a front-end that constructs intermediate form trees and simply wants to traverse the trees and execute a few actions. In such cases, the optimal tree walks of code-generator generators and the powerful attribute evaluation schemes of source-to-source translator systems are overkill. Programmers would rather avoid the overhead and complexity.

A SORCERER description is essentially an unambiguous grammar (collection of rules) in Extended BNF notation that describes the structure and content of a user's trees. The programmer annotates the tree grammar with actions to effect a translation, manipulate a user-defined data structure, or manipulate the tree itself. SORCERER generates a collection of simple C or C++ functions, one for each tree-grammar rule that recognizes tree patterns and performs the programmer's actions in the specified sequence.

Tree pattern matching is done efficiently in a top-down manner with an LL(1)-based[1] parsing strategy augmented with syntactic predicates to resolve non-LL(1) constructs (with selective backtracking) and semantic predicates to specify any context-sensitive tree patterns. Tree traversal speed is linear in the size of the tree unless a non-LL(1) construct is specified—in which case backtracking can be used selectively to recognize the construct while maintaining near-linear traversal speed.

SORCERER can be considered an extension to an existing language rather than a total replacement as other tools aspire to be. Consequently, programmers can use SORCERER to perform the well understood, tedious problem of parsing trees, without limiting themselves to describing the intended translation problem purely as attribute manipulations. SORCERER does not force you to use any particular parser generator or intermediate representation. Its application interface is extremely simple and can be linked with almost any application that constructs and manipulates trees.

SORCERER was designed to work with as many tree structures as possible because it requires nor assumes no pre-existing application such as a parser generator. However, we have made it particularly easy to integrate with trees built by ANTLR-generated parsers. Using the SORCERER C interface, the programmer's trees must have fields `down`, `right`, and `token` (which can be redefined easily with the C preprocessor). The SORCERER C++ interface is much less restrictive. The programmer must only define a small set of functions to allow the tree-parser to walk the programmer's trees. (This set includes `down()`, `right()`, and `type()`.)

SORCERER operates in one of two modes: non-transform mode and transform mode. In non-transform mode (the default case), SORCERER generates a simple tree parser that is best suited to syntax-directed translation. (The tree is not rewritten—a set of actions generates some form of output.) In transform mode, SORCERER generates a parser that assumes a tree transformation will be done. Without programmer intervention, the parser automatically copies the input tree to an output tree. Each rule has an implicit (automatically defined) result tree; the result tree of the start symbol is a pointer to the transformed tree. The various alternatives and grammar elements may be annotated with "`!`" to indicate that they should not be automatically linked into the output tree. Portions of, or entire, subtrees may be rewritten. A set of library functions is available to support tree manipulations. Transform mode is specified with the SORCERER `-transform` command-line option.

1. We build top-down parsers with one symbol of lookahead because they are usually sufficient to recognize intermediate form trees because they are often specifically designed to make translation easy; moreover, recursive-descent parsers provide tremendous semantic flexibility.

Intermediate Representations and Translation

We are often confronted with questions regarding the applicability of SORCERER. Some people ask why intermediate representations are used for translation. Those who are already familiar with the use of trees for translation ask why they should use SORCERER instead of building a tree walker by hand or building a C++ class hierarchy with `walk()` or `action()` virtual member functions. Compiler writers ask how SORCERER differs from code-generator generators and ask what SORCERER is good at. This section and the next address these issues to support our design choices regarding source translation and SORCERER.

The construction of computer language translators and compilers is generally broken down into separate phases such as lexical analysis, syntactic analysis, and translation where the task of translation can be handled in one of two ways:

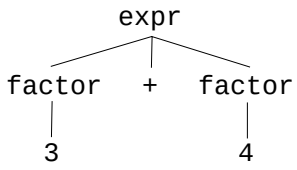
- Actions can be executed during the parse of the input text stream to generate output; when the parser has finished recognizing the input, the translation is complete. This type of translation is often called syntax-directed translation.
- Actions can be executed during the parse of the input text stream to construct an intermediate representation (IR) of the input, which will be re-examined later to perform a translation. These actions can be automatically inserted into the text parser by ANTLR as we have shown in previous chapters.

The advantages of constructing an intermediate representation are that multiple translators can be built without modifying the text parser, multiple simple passes over an IR can be used rather than a single complex pass, and, because large portions of an IR can be examined quickly (i.e., without rewinding an input file), more complicated translations can be performed. Syntax-directed translations are typically sufficient only for output languages that closely resemble the input language or for languages that can be generated without having to examine large amounts of the input stream, that is, with only local information.

For source to source translation, trees (normally called abstract syntax trees or *ASTs*) are the most appropriate implementation of an IR because they encode the grammatical structure used to recognize the input stream. For example, input string "`3+4`" is a valid phrase in an expression language, but does not specify what language structure it satisfies. On the other hand, the tree structure

has the same three input symbols, but additionally encodes the fact that the "+" is an operator and that "3" and "4" are operands. There is no point to parsing the input if your AST does not encode at least some of the language structure used to parse the input. The structure you encode in the AST should be specifically designed to make tree walking easy during a subsequent translation phase.

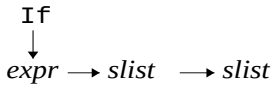
An AST should be distinguished from a *parse tree*, which encodes not only the grammatical structure of the input, but records which rules were applied during the parse. A parse tree for our plus-tree might look like:



which is bulkier, contains information that is unnecessary for translation (namely the rule names), and harder to manipulate than a tree with operators as subtree roots.

If all tree structures were binary (each node had only two children), then a tree node could be described with whatever information was needed for each node plus two child pointers. Unfortunately, AST nodes often need more than two children and, worse still, the number of child varies greatly; that is, the most appropriate tree structure for an `if`-statement may be an `If` root node with three children: one for the conditional, one for the `then` statement list, and one for the `else` clause statement list. In both ANTLR and SORCERER we have adopted the *child-sibling* tree implementation structure where each node has a *first-child* and *next-sibling* pointer. The expression tree above would be structured and illustrated as

and an `if`-statement would be structured as



Child-sibling trees can be conveniently described textually in a LISP-like manner:

(*parent child1 ... childn*)

So, our expression tree could be described as

```
( + 3 4 )
```

and our `If` tree as

`( If` *expr slist slist* `)`

The contents of each AST node may contain a variety of things such as pointers into the symbol table and information about the associated token object. The most important piece of information is the token type associated with the input token from which the node was constructed. It is used during a tree walk to distinguish between trees of identical structure but different contents. For example, the tree

is considered different than

because of the differences in their token types (which we normally identify graphically via node labels). Whether the token type is available as a C `struct` field or a C++ member function is irrelevant.

Tree structures with homogeneous nodes as described here are easy to construct, whereas trees consisting of a variety of node types are very difficult to construct or transform automatically.

SORCERER Versus Hand-Coded Tree Walking

The question "Why is SORCERER useful when you can write a tree parser by hand?" is analogous to asking why you need a parser generator when you can write text parser by hand. The answer is the same, although it is not a perfectly fair comparison because IRs are generally designed to be easier to parse than the corresponding input text. Nonetheless, SORCERER grammars have the advantage over hand-coded tree parsers because grammars:

- Are much easier and faster to write
- Are smaller than programs
- Are more readable as grammars directly specify the IR structure
- Are more maintainable
- Automatically detect malformed input trees

- Can possibly detect ambiguities/nondeterminisms in your tree description (such as when two different patterns have the same root token) that might be missed when writing a tree walker by hand

Further, parsing a tree is the same as parsing a text stream except that the tree parser must match a two-dimensional stream rather than a one-dimensional stream.

Because a variety of techniques are available to perform tree walks and translations, it's worth looking at some common C and C++ hand-coding techniques and understanding why (or how?) SORCERER grammars often represent more elegant solutions.

In C, given homogeneous tree structures, there are two possible tree-walking strategies:

- A simple recursive depth-first search function applies a translation function to each node in the tree indiscriminately. The translation function would have to test the node to which it was applied in order to perform the necessary task. Any translation function would have trouble dealing with multiple-node subtrees such as those constructed for `if`-statements. The structure of the IR is not tested for deformities.
- A hand-built parser explicitly tests for the IR structure such as "if root node is a `If` node and the first child is an expression, ..." SORCERER is a simply a shorthand for this strategy.

Alternatively, you can use C structures with fields that point to the required children rather than a list of children nodes. In C++, given tree nodes with homogeneous behavior, you could make each node in the tree an appropriate class that had a `walk()` member function. The `walk()` function would do the appropriate thing depending on what type of tree node it was. However, you would end up building a complete tree parser by hand. The class member `PLUSNode::walk()` would be analogous to a rule `plus_expr`. For example,

```
class PLUSNode : public AST {
    walk()
    {
        MATCH(PLUS);                        // match the root
        this->down()->walk();               // walk left operand
        this->down()->right()->walk();      // walk right operand
    }
    ...
}
```

versus

```
plus_expr
    :   #( PLUS expr expr )
    ;
```

where `expr` would nicely group all the expression templates in one rule. For example,

```
expr
    :   plus_expr
    |   mult_expr
    ...
    ;
```

whereas in the hand-coded version, there could be no explicit specification for what an expression tree looks like—there is just a collection C++ classes with similar names such as `PLUSNode`, `MULTNode`, and so on:

```
class PLUSNode : public AST { walk(); ... };
class MULTNode : public AST { walk(); ... };
```

On the other hand, if we used a variety of tree node types, a set of class members could point to the appropriate information rather than using a generic list of children. For example,

```
class EXPRNode : public AST {...};

class PLUSNode : public EXPRNode {
    EXPR *left_opnd;
    EXPR *right_opnd;
    walk()
    {
        left_opnd->walk();
        right_opnd->walk();
    }
};
```

However, a `walk()` function is still needed to specify what to do and in what order. A set of member pointers is not nearly as powerful as a grammar because a grammar can specify operand order and sequences of operands. The order of operands is important during translation when you want to generate an appropriate sequence of output. (What if the field names were `Bill` and `Ted` instead of `left_opnd` and `right_opnd`? While these are silly names, the point is made that you have to encode order in the names of the fields.) The ability to specify sequences is analogous to allowing you to specify structures of varying size. For example, the tree structure to describe a function call expression would have to be encoded as follows:

```
class FUNCCallNode : public EXPRNode {
    char *id;
    List<EXPRNode *> arguments;
    walk()
    {
        for ( each element of arguments list )
            arg->walk();
    }
};
```

Because the number of arguments is unknown at compile time, a list of arguments must be maintained and walked by hand; whereas, using SORCERER, because everything is represented as a generic list of nodes, you could easily describe such IR structures:

```
func_call
    :  #( FUNCCall ID ( expr )* )
    ;
```

No matter how fancy you get using C or C++, you must still describe your IR structure at least partially with hand-written code rather than with a grammar.

The only remaining reason to have a variety of node class types is to specify what translation action to execute for each node. Action execution, too, is better handled grammatically. Actions embedded within the SORCERER grammar dictate what to do and, according to action position, when to do it. In this manner, it is very obvious what actions are executed during the tree walk. With the hand-coded C++ approach, you would have to peruse the class hierarchy to discover what would happen at each node. Further, there may be cases where you have two identical nodes in an IR structure for which you want to perform two different actions, depending on their context. With the grammatical approach, you simply place a different action at each node reference. The hand-coded approach forces you to make action member functions sensitive to their surrounding tree context, which is difficult and cumbersome.

We have argued in this section that a grammar is more appropriate for describing the structure of an IR than a hand-coded C function or C++ class hierarchy with a set of `walk()` member functions and that child-sibling trees are very convenient tree-structure implementation. Different tree node types in C/C++ are required only when a grammar cannot be used to describe the tree's structure. Translations can also be performed more easily by embedding actions within a grammar rather than scattering the actions around a class hierarchy. On the other hand, we do not stop you from walking trees with a variety of class types. SORCERER will pretend, however, that your tree consists only of nodes of a single type.

What Is SORCERER Good At and Bad At?

SORCERER is not the "silver bullet" of translation. It was designed specifically to support source-to-source translations through a set of embedded actions that generate output directly or via a set of tree transformation actions.

SORCERER is good at

- Describing tree structures (just as LISP is good at it)
- Syntax-directed translations
- Tree transformations either local such as constant folding and tree normalizations or global such as adding declarations for implicitly defined variables
- Interpreting trees such as for scripting languages

SORCERER is not good at or does not support

- Optimized assembly code generation
- Construction of “use-def” chains, data-flow dependency graphs, and other common compiler data structures, although SORCERER can be used to traverse statement lists to construct these data structures with user-supplied actions

How Is SORCERER Different Than a Code-Generator Generator?

Compiler code-generator generators are designed to produce a stream of assembly langauge instructions from an input tree representing the statements in a source program. Because there may be multiple assembly instructions for a single tree pattern (e.g., integer addition and increment), a code-generator generator must handle ambiguous grammars. An ambiguous grammar is one for which an input sequence may be recognized in more than one way by the resulting parser (i.e., there is more than one grammatical derivation). A "cost" is provided for each tree pattern in the grammar to indicate how "expensive" the instruction issued by the pattern would be in terms of execution speed or memory space. The code-generator finds the optimal walk of the input tree, which results in the optimal assembly instruction stream. SORCERER differs in the following ways:

1. Because code-generators must choose between competing grammar alternatives, they must match the entire alternative before executing a translation action. However, the ability to execute a translation action at any point during the parse is indispensable for source-to-source translation.
2. Code-generator generators were not designed for and cannot perform tree rewrites.

3. Code-generator generators normally do not allow EBNF grammar constructs to specify lists of elements.
4. While code-generator generators handle unambiguous grammars such as SORCERER's as well as ambiguous grammars, they may not handle unambiguous grammars as efficiently as a tool specifically tuned for fast, deterministic parsing.

It is ironic that most translator generators are code-generator generators, even though most translation problems do not involve compilation. Unfortunately, few practical tools like SORCERER exist for the larger scope of source-to-source translation.

2 A Tutorial

In this chapter we walk you through a series of progressively complex recognizers, interpreters, and translators for polynomials using ANTLR and SORCERER. We assume a passing familiarity with the syntax of ANTLR/SORCERER and a basic understanding of scanners (regular expressions) and parsers (grammars).

Evaluating and Differentiating Polynomials

The first application developed in this chapter accepts a sequence of polynomial equations of the form:

r = ax^n + by^m + ... ;

for some real numbers *a*, *b*, *m*, and *n* where the polynomial variables (here, *x*, *y* and *r*) may be any single lowercase letter (which we call a *register* or *identifier*) "`a..z`". Each polynomial is evaluated, and the result is stored into the variable on the left-hand-side of the equation. (Here, the result is stored into *r*.) As output, we print the result after each polynomial evaluation. For example,

```
lonewolf:/projects/Book/tutorial/simple$ poly
a = 5;
storing 5.000000 in a
b = 3a^2 + 2a + 7;
storing 92.000000 in b
```

where the **bold** characters are the polynomials to be entered.

This task uses two equivalent methods: *(i)* an ANTLR grammar to recognize the input and to compute the results, and *(ii)* an ANTLR grammar to recognize the input and to construct ASTs that will be evaluated with a SORCERER grammar.

The second application developed in this tutorial differentiates a polynomial to demonstrate the tree transformation abilities of SORCERER. A polynomial of the form:

ax^n + bx^m + ...

is differentiated by manipulating the associated AST and printing the polynomial back out. (*Those of you who slept through calculus may need to be reminded that the derivative of ax^b is (ab)x^(b-1), the derivative of ax is a (even if a is* `1`*) and the derivative of a is* `0` *where a is real and b is an integer*.) For example,

```
lonewolf:/projects/Book/tutorial/rewrite$ ./poly
2x^5 + x^2 + 3x + 9
10x^4+2x+3
```

where again the **bold** characters represent the input string.

Language Recognition and Syntax-Directed Interpretation

We break up building our polynomial evaluator into three main tasks:

1. Describing the syntax
2. Describing the vocabulary (set of input symbols or tokens)
3. Inserting semantic actions to evaluate the polynomial

Tasks **1** and **2** result in a working polynomial recognizer and task **3** results in a working evaluator. Testing a project at each such stage is recommended.

Syntax

Begin the task of building a recognizer for a language by examining a representative set of input strings and trying to identify the underlying grammatical structure. In our case, equations are strings like:

```
a = 3;
b = a+5a + 2a^2 + a^8 + 4;
```

The best way to describe the input "at a coarse level" is as a series of assignments with an identifier on the left and a polynomial on the right, terminated with a semicolon. In grammar notation, we write:

```
interp
    :   ( ID "=" poly ";" )+
    ;
```

where anything inside double quotes is a regular expression describing an input symbol (e.g., `"="` matches the equal sign) and `ID` is a label for a regular expression defined elsewhere. The `(...)+` construct indicates that the enclosed elements should be matched one or more times. A polynomial looks like a series of terms added together; hence, we can describe a polynomial as:

```
poly
    :   term ( "\+" term )*
    ;
```

where the `(...)*` construct indicates that the enclosed elements can be matched zero or more times; a polynomial may be composed of a single term; hence, we use a zero-or-more rather than one-or-more subrule. Polynomial terms are simple numbers, simple variables, variables with exponents, or variables with exponents and coefficients. This fact we encode as:

```
term
    :   FLOAT
    |   reg
    |   reg "^" exp
    |   coefficient reg "^" exp
    ;

coefficient
    :   FLOAT
    ;

reg :   ID
    ;

exp :   reg
    |   FLOAT
    ;
```

where we create rules `coefficient`, `reg` and `exp` to make rule `term` more clear.

As it appears, rule `term` would result in a parser that needed to see two symbols ahead (over the common production left-prefix `reg`), instead of the normal one symbol, to determine which alternative would match. Specifically, upon input "`a`", the parser could not determine

whether the second or third alternative of `term` applied; that is, the parser could not see if a "^" followed the "a".

In order to demonstrate the principle of "left-factoring" and to make the resulting parser behave more naturally, we left-factor rule `term` so that the resulting parser requires only a single symbol of lookahead. (Note that in general, left-factoring is not always possible and more than a single symbol of lookahead is beneficial.) Left-factoring a rule means that you combine common left-prefixes among the productions. In our case, it is necessary to merge alternatives two and three, and one and four into two new, slightly more complicated, alternatives:

```
term
    :coefficient { reg { "^" exp } } // merged alts 1 and 4
    |reg { "^" exp }                 // merged alts 2 and 3
    ;
```

where the `{..}` subrule implies that the enclosed elements are optional. In this form, we have preserved the grammatical structure, but have reduced the lookahead requirements. For example, rule `term` attempts to match alternative one if the input is a number and attempts alternative two if the input is a register.

Vocabulary

Once the grammatical structure of a language is described, the set of vocabulary symbols called tokens must be specified. Our polynomial language has only six tokens, which are described in Table 1 on page 44. The grammar we have developed so far provides implicit

TABLE 1. Vocabulary Symbols for Polynomial Language

Regular Expression	Description
`"="`	The equals sign
`";"`	The semicolon
`"\+"`	The plus sign where the "\" escape character indicates that the actual plus sign is required—an unescaped "+" is a reserved symbol (meaning one or more as it does at the grammatical level)
`"^"`	The exponentiation operator or caret

TABLE 1. (Continued) Vocabulary Symbols for Polynomial Language

Regular Expression	Description
`"[0-9]+ {.[0-9]+}"`	A floating point number. Match a series of one or more digits (`0` through `9`) followed optionally by a decimal point and more digits
`"[a-z]"`	An identifier. Our language restricts identifiers to single lower-case letters called registers, which simplifies associating a value with an identifier.

definitions (by simply referring to them) for all but numbers and registers, which can be described as

```
#token ID       "[a-z]"
#token FLOAT    "[0-9]+ { . [0-9]+ }"
```

where we have labeled them for grammar readability.

We must also specify that white space is to be ignored, which is conveniently done with a regular expression. The scanner (the code that breaks up the input character stream into vocabulary symbols) normally returns after matching a regular expression. However, we don't want to mention a white space vocabulary symbol everywhere within our grammar. So, we attach an action to the regular expression, indicating that the matched input symbol should be ignored:

```
#token "[\ \t]+"    <<skip();>>
#token "\n"         <<skip(); newline();>>
```

where we have separated out the recognition of the newline character so that we can tell the scanner to increment the line count (for error reporting).

The recognizer for our language is now complete. In order to test it, we must specify a grammar class for ANTLR, inform ANTLR what the type of our token objects is, and provide a main program. The following code section is a complete description that will result in an executable parser:

```
<<
 #include "PBlackBox.h"               // Define a black box for main()
 #include "DLGLexer.h"                // What's the scanner called?
 typedef ANTLRCommonToken ANTLRToken;  // Use a predefined token object

main()
{
```

```
    ParserBlackBox<DLGLexer, PolyParser, ANTLRToken> p(stdin);
     p.parser()->interp();// start up the parser
}
>>

#token "[\ \t]+"    <<skip();>>
#token "\n"         <<skip(); newline();>>
 #token ID "[a-z]"
 #token FLOAT "[0-9]+ { . [0-9]+ }"

class PolyParser {
interp
    :   ( ID "=" poly ";" )+
    ;

poly
    :   term ( "\+" term )*
    ;

term
    :   coefficient { reg { "^" exp } }
    |   reg { "^" exp }
    ;

coefficient
    :   FLOAT
    ;

reg :   ID
    ;

exp :   reg
    |   FLOAT
    ;
}
```

A makefile that constructs the executable is written automatically by the `genmk` tool as:

```
genmk -CC -class PolyParser -project poly poly.g > makefile
```

The makefile then has to be modified so that makefile variables `PCCTS` and `CCC` (the C++ compiler) are set properly. For example,

```
PCCTS = /usr/local/src/pccts
CCC=g++
```

The executable called `poly` can be used to check the syntax of our language, but nothing else. For example,

```
lonewolf:/projects/Book/tutorial/simple$ poly
a = a + =
line 1: syntax error at "=" missing { ID FLOAT }
line 2: syntax error at "" missing ;
```

Semantic Actions

To actually compute the values described by the polynomials, we must embed actions within the grammar.

Begin by making a basic assumption: every rule except `interp` returns the value indicated by its part of the computation. The result of `poly` is then the overall result of evaluating the polynomial:

```
poly > [float r]
    :   <<float f;>>
        term>[$r] ( "\+" term>[f] <<$r += f;>> )*
    ;
```

Rule `poly` is defined to have a return value called `$r` via the "`> [float r]`" notation; this is similar to the output redirection character of `UNIX` shells. Setting the value of `$r` sets the return value of `poly`.

The first action after the "`:`" is an init-action (because it is the first action of a rule or subrule). The init-action defines a local variable called `f` that will be used in the `(...)*` loop to hold the return value of the term.

The result of each polynomial is stored into the register specified on the left-hand side of the equation.

```
interp
    :   <<float r;>>
        (   lhs:ID "=" poly>[r] ";" <<store($lhs->getText(), r);>>
        )+
    ;
```

The result of `poly` is placed into local variable `r,` and the register (`a..z,` in our case) is accessed by labeling the token reference that matches the left-hand side. The label `lhs` becomes an `ANTLRTokenPtr` in the output C++ code, and the standard method `getText()` is used to obtain the text associated with a token object. Our use of the `store()` function is postponed until after the remainder of the grammar is explained.

The rules in the parse tree at the lowest level are easily augmented to return the value of the specified polynomial portion.

```
coefficient > [float r]
    :   flt:FLOAT       <<$r = atof($flt->getText());>>
    ;
```

Here, the `atof()` library function is used compute the floating point value of the text matched for the `FLOAT` token. (That is, string `"3.14"` is converted to floating value `3.14`.) Again, the return value is set by assigning a value to `$r`.

To compute the value of a variable, we call a function to return the value of the referenced register; `value()` is explained shortly.

```
reg > [float r]
    :   id:ID           <<$r = value($id->getText());>>
    ;
```

Rule `exp` is just as simple:

```
exp > [float r]
    :   reg > [$r]
    |   flt:FLOAT       <<$r = atof($flt->getText());>>
    ;
```

Rule `term,` on the other hand, is a bit more complicated. Given input "`3x^2`", rule `term` chooses alternative one and begins by using a rule reference to `coefficient` to match "`3`". The "`x`" is matched by choosing the first alternative of the outermost subrule. After having matched rule reference `reg`, the parser sees the "`^2`" and applies the first alternative of the nested subrule. At that point, the parser has collected the coefficient value (variable `c`, the value stored in the register (variable `v`), and the value of the exponent (variable `e`). The result of the rule is `c` times `v` to the power `e`.

You may have noticed that what was previously an optional subrule in the grammar without actions is now a subrule with an empty alternative. The two are functionally equivalent from a language recognition standpoint, but allow us to attach an action to the case when nothing is matched for that subrule. For example, input "`3x`" follows the same path through rule term as "`3x^2`" except that the exponent is missing. The empty path of the nested subrule is taken instead. We have added an action "`$r=c*v;`" to the empty path to compute the correct value when the exponent is missing. The other paths of `term` can be inferred from this description.

```
term > [float r]
    :   <<float f=0.0, e=0.0, c=0.0, v=0.0;>>
        coefficient > [c]
        (   reg > [v]
            (   "^" exp>[e]     <<$r = c*pow(v,e);>>
            |                   <<$r = c*v;>>
            )
        |                       <<$r = c;>>
        )
    |   reg > [f]
```

```
        (   "^" exp > [e]       <<$r = pow(f,e);>>
        |                       <<$r = f;>>
        )
    ;
```

Trigger Functions. The functions `store()` and `value()` isolate our implementation from the grammar a bit and avoid placing actions within the grammar. This is a good principle to follow because it allows the behavior or implementation details of a translator to be changed without having to actually go inside a possibly dark and scary grammar. Trigger functions are defined easily by adding some virtual functions to the class definition:

```
class PolyParser {
<<
protected:
    virtual void store(char *reg, float v) {;}
    virtual float value(char *reg) {;}
>>
    ...
}
```

where we have defined only the two most basic operations needed by our parser. When `PolyParser` is subclassed, these parser triggers can be easily overridden:

```
class InterpretingPolyParser : public PolyParser {
protected:
    float regs['z'-'a'+1];
    virtual void store(char *reg, float v)
        {
            regs[reg[0]-'a'] = v;
            printf("storing %f in %s\n", v, reg);
        }
    virtual float value(char *reg) { return regs[reg[0]-'a']; }
public:
    InterpretingPolyParser(ANTLRTokenBuffer *input)
        : PolyParser(input)
        {
            for (int i='a'; i<='z'; i++) regs[i-'a'] = 0.0;
        }
};
```

Note that we have used a `float` array indexed by the registers `a..z` to store and retrieve the register values.

The main program is modified to use the subclassed parser:

```
main()
{
    ParserBlackBox<DLGLexer,
```

```
                    InterpretingPolyParser,
                    ANTLRToken> p(stdin);
    p.parser()->interp();
}
```

We have now successfully augmented our grammar to compute and print out the values of input polynomials. The entire grammar file looks like this:

```
<<
#include <math.h>

class InterpretingPolyParser : public PolyParser {
protected:
    float regs['z'-'a'+1];
    virtual void store(char *reg, float v)
        {
            regs[reg[0]-'a'] = v;
            printf("storing %f in %s\n", v, reg);
        }
    virtual float value(char *reg)
        { return regs[reg[0]-'a']; }
public:
    InterpretingPolyParser(ANTLRTokenBuffer *input)
        : PolyParser(input)
        {
            for (int i='a'; i<='z'; i++) regs[i-'a'] = 0.0;
        }
};

#include "PBlackBox.h"
#include "DLGLexer.h"
typedef ANTLRCommonToken ANTLRToken;

main()
{
    ParserBlackBox<DLGLexer,
                    InterpretingPolyParser,
                    ANTLRToken> p(stdin);
    p.parser()->interp();
}
>>

#token "[\ \t]+"<<skip();>>
#token "\n"<<skip(); newline();>>
#token ID  "[a-z]"
#token FLOAT"[0-9]+ { . [0-9]+ }"
```

```
class PolyParser {
<<
protected:
    virtual void store(char *reg, float v) {;}
    virtual float value(char *reg) {;}
>>

interp
    :      <<float r;>>
           (   lhs:ID "=" poly>[r] ";" <<store($lhs->getText(),r);>>
           )+
    ;

poly   >   [float r]
    :      <<float f;>>
           term>[$r] ( "\+" term>[f] <<$r += f;>> )*
    ;

term   >   [float r]
    :      <<float f=0.0, e=0.0, c=0.0, v=0.0;>>
           coefficient > [c]
           (      reg > [v]
                  (      "^" exp>[e]<<$r = c*pow(v,e);>>
                  |                   <<$r = c*v;>>
                  )
           |                          <<$r = c;>>
           )
    |      reg > [f]
           (      "^" exp > [e]<<$r = pow(f,e);>>
           |                          <<$r = f;>>
           )
    ;

coefficient > [float r]
    :      flt:FLOAT<<$r = atof($flt->getText());>>
    ;

reg    >   [float r]
    :      id:ID      <<$r = value($id->getText());>>
    ;

exp    >   [float r]
    :      reg > [$r]
```

```
    |       flt:FLOAT<<$r = atof($flt->getText());>>
    ;

}
```

The `makefile` has not changed, and the executable can be generated by simply remaking. Here is some sample input ouput.

```
lonewolf:/projects/Book/tutorial/simple$ poly
a=3;
storing 3.000000 in a
b = 10a^2 + 4a + 3;
storing 105.000000 in b
```

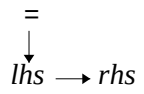
Constructing and Walking ASTs

Another way to evaluate the polynomial equations in the previous section is to have the parser construct trees and then walk those trees to compute the results. In this section, we remove the actions from the previous example's grammar, annotate the grammar with a few symbols and actions to construct trees, then build a SORCERER grammar to walk our trees and compute the results.

AST Design

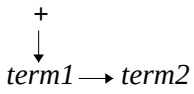
The structure of intermediate trees is important. The fundamental design goal is that an intermediate form should contain not only the contents of the input stream, but should represent the structure of the underlying language as well. For example, a linked list of the input token objects has complete contents, but has no structure to indicate how the input was parsed.

The top-level tree structure for our ASTs will represent the assignment operation as:

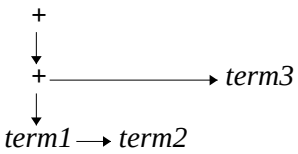


where *lhs* and *rhs* are left-hand side and right-hand side, respectively. For example, "`a=3;`" will be represented by:

Polynomials are sums of terms:

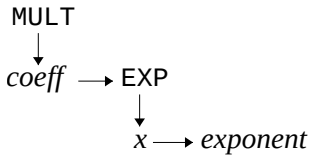


for "*term1* + *term2*". For simplicity, we have exactly two children for "+" nodes; hence, multiple additions are represented by using the result of one addition as the operand of another addition. For example, "*term1* + *term2* + *term*3" would be represented as:

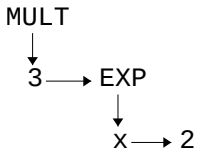


With a depth-first walk of the tree, the order of operations is correct—left to right for addition.

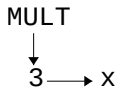
Terms themselves have the following template:



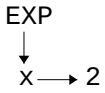
For example, "3x^2" is represented by:



Terms without exponents such as "3x" look like:



and terms without coefficients such as "x^2" look like:



Notice that there is no corresponding input token for the multiply operation because it is implicit that the term variable is multiplied by its coefficient. This node must be created manually with a grammar action.

Constructing ASTs

In this section, we define the appearance of AST node and modify the grammar to construct ASTs.

For simplicity, define an AST node to contain simply a pointer to the associated input token object. The type of an AST must be called AST, and we have placed its definition in a file called AST.h:

```
#include "ASTBase.h"
#include "AToken.h"

class AST : public ASTBase {
protected:
    ANTLRTokenPtr token; // pointer to the token found in input
public:

    // called when #[tokentype,string] is seen in an action
    AST(ANTLRTokenType tok, char *s) { token = new ANTLRToken(tok,s); }

    // constructor called by parser for token references in grammar
    AST(ANTLRTokenPtr t) { token = t; }

    // define what happens at a node when preorder() is called
    void preorder_action() {
        char *s = token->getText();
```

```
        printf(" %s", s);
    }
};
```

The main program and required definitions become:

```
<<
#include "PBlackBox.h"
#include "DLGLexer.h"
typedef ANTLRCommonToken ANTLRToken;

#include "AST.h"

main()
{
    ParserBlackBox<DLGLexer, PolyParser, ANTLRToken> p(stdin);
    ASTBase *root = NULL;
    p.parser()->interp(&root);
}
>>
```

ANTLR uses a return parameter to return the AST constructed for each rule. As a result, the call to the starting rule must pass the address of a tree pointer where the result will be stored; this pointer must be initialized to `NULL`.

Tree building. Eventually, each polynomial is passed to the tree walker for evaluation one at a time; a list of all equations is not maintained. The previous starting rule

```
interp
    :   ( ID "=" poly ";" )+
    ;
```

is therefore split into the following:

```
interp!
    :   ( assign )+
    ;
assign
    :   ID "="^ poly ";"!       <<#0->preorder(); printf("\n");>>
    ;
```

where the `!` on the `interp` header indicates that ANTLR is not to construct trees in that rule (a list of assignments is not required). Had we left the rule as written and added the tree construction grammar operators (as shown in the `assign` rule), the trees would not be constructed correctly after the first polynomial. The second iteration of the `(...)+` loop would continue to add to the same tree because each rule constructs exactly one tree.

The `!` on the `";"` token in rule `assign` indicates that a node is not to be constructed in the AST for that token. The `^` suffix on the `"="` token indicates that the assignment is to be made the root of the current subtree (whatever that happens to be at the time the assignment operator is matched)—in this case, a lone `ID` node. Any other token is assumed to be a leaf node in the AST. All rule references without `!` suffixes return subtrees whose roots are made children of the current subtree root. For example, the tree returned from `poly` is made a child of the assignment node with the target of the assignment as the other child.

The call to `preorder()` for the return tree, `#0`, in `assign` walks the tree and prints it out in LISP form. For the moment, we print out the tree rather than invoke a SORCERER tree walker so that this portion of the evaluator can be tested separately.

To collect the sum of terms is easy to do:

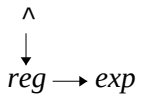
```
poly
    :   term ( "\+"^ term )*
    ;
```

The `^` suffix on the `"\+"` tells ANTLR to create an addition node and place it as the root of whatever subtree has been constructed up until that point for rule `poly`. The subtrees returned by the `term` references are collected as children of the addition nodes.

The simple terms of the *x^e* are constructed by the second alternative of term:

```
    |   reg { "^"^ exp }
```

where we have converted the `(exp|)` back to the simpler optional subrule. The "`"^"^`" may be a bit confusing. The input token is the up-arrow and the grammar operator for AST root is also up-arrow. Hence, trees of the form



are created (assuming an exponent is found on the input).

Constructing trees for terms with coefficients is complicated by the fact that a multiply node must be created and placed in the tree for which there is no corresponding input symbol. We will therefore have to turn off ANTLR's default tree construction mechanism to build the AST manually. Because the automatic AST mechanism can only be turned off at a rule-level granularity, we have `term` call another rule which builds the appropriate tree manually, thus leaving the automatic mechanism to work its magic for the other alternatives in `term`:

```
term
    :   bigterm
    |   reg { "^"^ exp }
    ;
```

```
bigterm!
    :  c:coefficient
       (   r:reg
           (   "^" e:exp
               <<#0 = #(#[MULT,"MULT"], #c, #(#[EXP,"EXP"], #r, #e));>>
           |   <<#0 = #(#[MULT,"MULT"], #c, #r);>>
           )
       |       <<#0 = #c;>>
       )
    ;
```

where "`#[`*arglist*`]`" is a node constructor translated to "`new AST(`*arglist*`)`" and

```
#(root, child1, ..., childn)
```

is converted to a call to the tree constructor

```
ASTBase::tmake(root, child1, ..., childn, NULL);
```

The rule references in `bigterm` are labeled so that the resulting ASTs can be referenced and placed in the tree for `bigterm`. The simplest path through the rule is to match `coefficient` followed by nothing, in which case the `<<#0=#c;>>` action is executed to set the return value (`#0`) of `bigterm`. If a coefficient and a register are found, but no exponent, the

```
<<#0 = #(#[MULT,"MULT"], #c, #r);>>
```

action is executed. It creates a tree with a `MULT` node at the root and the coefficient and register as children, where the `#[...]` node constructor is translated to a call to `AST(...)` by ANTLR. The `#(...)` is translated to a call to the tree constructor where the first argument is the root node and all subsequent arguments are the children of that node. Trees for full terms are constructed in a similar fashion.We define the `MULT` token type referenced within the above actions with

```
#token MULT
```

even though no input symbol corresponds to this token type and, therefore, we do not specify a regular expression. We also need to access the token type of the exponent operator so that the manual tree construction actions can build the appropriate nodes:

```
#token EXP    "^"
```

Rules `coefficient, reg`, and `exp` automatically construct nodes for the single tokens they match; nothing further must be specified.

The grammar does not yet pass anything to a tree walker for evaluation, but may be tested to see that the correct trees are being produced. In the complete ANTLR description, (`ASTBase::preorder()` is called to dump out the trees):

```
<<
#include "PBlackBox.h"
#include "DLGLexer.h"
typedef ANTLRCommonToken ANTLRToken;

#include "AST.h"

main()
{
    ParserBlackBox<DLGLexer, PolyParser, ANTLRToken> p(stdin);
    ASTBase *root = NULL;
    p.parser()->interp(&root);
}
>>

#token "[\ \t]+"    <<skip();>>
#token "\n"         <<skip(); newline();>>
#token ID           "[a-z]"
#token FLOAT        "[0-9]+ { . [0-9]+ }"

#token EXP      "^"
#token MULT

class PolyParser {

interp!
    :      ( a:assign )+
    ;

assign
    :      ID "="^ poly ";"!   <<#0->preorder(); printf("\n");>>
    ;

poly
    :      term ( "\+"^ term )*
    ;

term
    :      bigterm
    |      reg { "^"^ exp }
    ;

bigterm!
    :      c:coefficient
           (      r:reg
                  (   "^" e:exp
```

```
                    <<#0 = #(#[MULT,"MULT"], #c, #(#[EXP,"EXP"], #r,
#e));>>
                |   <<#0 = #(#[MULT,"MULT"], #c, #r);>>
                )
          |         <<#0 = #c;>>
          )
    ;

coefficient
    :     FLOAT
    ;

reg :     ID
    ;

exp :     reg
    |     FLOAT
    ;
}
```

A makefile may be constructed by invoking

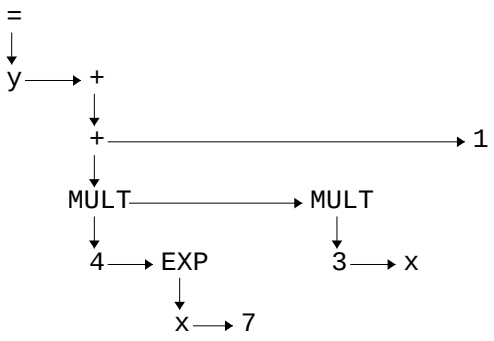
```
genmk -CC -class PolyParser -project poly -trees poly.g
```

and placing the output in `makefile`. Do not forget to fill in appropriate values for the `PCCTS` and `CCC` make variables. Here is some sample input/output:

```
lonewolf:/projects/Book/tutorial/testAST$ ./poly
x = 2;
 ( = x 2 )
y = 4x^7 + 3x + 1;
 ( = y ( + ( + ( MULT 4 ( EXP x 7 ) ) ( MULT 3 x ) ) 1 ) )
```

The corresponding ASTs look like

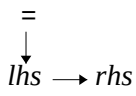
and



Describing ASTs With SORCERER

Now that our parser constructs trees, we can evaluate the polynomials by walking them, either with a hand-built tree walker or by having SORCERER generate a tree walker automatically. SORCERER accepts a grammatical description of your AST structure and generates a recursive-descent tree walker that looks very much like what you would build by hand. However, tree grammars have the same advantages over hand built tree walkers that ANTLR grammars has over hand-built conventional text parsers.

As in our AST design, assignments are of the form



which can easily be described with a rule in SORCERER:

```
assign
    :   #( ASSIGN ID poly )
    ;
```

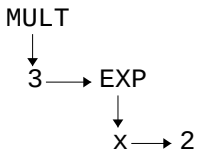
Where the `ASSIGN` token type will be attached to the `"="` token in the ANTLR grammar later.

Our AST design for polynomials can be described as:

```
poly:  #( MULT poly poly )
    |  #( ADD poly poly )
    |  #( EXP poly poly )
    |  ID
```

```
    |   FLOAT
    ;
```

where the ADD token type is attached to the "\+" token in the ANTLR grammar later. Rule poly lists all the possible subtrees and leaf nodes. Each subtree has a binary operator at the root. We do not have to specify a priority in which these particular subtrees can be matched against the input tree because the incoming tree has the precedence encoded in the structure itself. For example, in the tree



it is clear that the exponentiation is to be done first because it is farther down in the tree than the multiplication operator. All children must be computed before the parent operation can take place.

The experienced reader will note that our SORCERER grammar is a bit looser than necessary. For example, the third alternative can describe the AST structure more precisely:

```
    |   #( EXP ID FLOAT )
```

because "*x*^*e*" is only allowed for *x* as identifiers and *e* as floats. However, for simplicity, we make all operands references to rule poly. Later, we will only need one action to compute the value of a FLOAT node (i.e., in the fourth alternative) rather than at each place in a "tighter" grammar where FLOAT is referenced.

To satisfy the SORCERER programmer's interface, we must define what the input trees look like by providing type SORAST. We include AST.h and then specify

```
typedef AST SORAST;
```

The SORCERER programmer's interface requires also that each node be able to identify its token type. For convenience, we have added a function to retrieve the text associated with the token object stored in each AST node. The augmented AST definition looks like this:

```
#include "ASTBase.h"
#include "AToken.h"
#include "ATokPtr.h"
class AST : public ASTBase {
protected:
    ANTLRTokenPtr token;// pointer to the token found on input
public:

    // called when #[tokentype,string] is seen in an action
```

```
    AST(ANTLRTokenType tok, char *s) {token = new ANTLRToken(tok, s);}

    // constructor called by parser for token references in grammar
    AST(ANTLRTokenPtr t) { token = t; }

    // define what happens at a node when preorder() is called
    void preorder_action() {
    char *s = token->getText();
    printf(" %s", s);
    }

    // every node must know it's token type
    virtual int type() { return token->getType(); }

    // convenient to get text of associated input token
    char *getText()    { return token->getText(); }
};
```

We also need to give a class definition around the SORCERER grammar along with our two trigger functions.

```
class EvalPoly {
<<
protected:
    virtual float value(char *r) = 0;
    virtual void store(char *r, float v) = 0;
>>
    ...
}
```

We subclass `EvalPoly` and place it in `MyEvalPoly.h` to define these trigger functions.

```
#include "EvalPoly.h"

class MyEvalPoly : public EvalPoly {
protected:
    float regs['z'-'a'+1];
    virtual float value(char *r){ return regs[r[0]-'a']; }
    virtual void store(char *r, float v){ regs[r[0]-'a'] = v; }
public:
    MyEvalPoly()
        {
            for (int i='a'; i<='z'; i++) regs[i-'a'] = 0.0;
        }
};
```

We have again used a `float` array indexed by the registers `a..z` to store and retrieve the register values.

Adding Actions to Compute Polynomial Values

To actually compute the value of a polynomial, we add actions to the SORCERER grammar. As with the ANTLR only version, we use a few trigger functions to store and retrieve register values. Assigning the result of evaluating a polynomial is done in rule `assign` as follows:

```
assign
    :   <<float r;>>
        #( ASSIGN id:ID poly>[r] )
        <<
        store(id->getText(), r);
        printf("storing %f in %s\n", r, id->getText());
        >>
    ;
```

We define a local variable, `r`, with the init-action (first action of a rule or subrule) and place the result of `poly` into it. We call our trigger function `store()` with the register and the result value and print it out. The position of this action is important: It must be done after the reference to `poly` so that `poly's` value is computed before the `printf` tries to use it.

Computing the value of the polynomial trees is straightforward (one of the design goals of our AST, remember?):

```
poly > [float r]
    :   <<float p1,p2;>>
        #( MULT poly>[p1] poly>[p2] )      <<r = p1*p2;>>

    |   <<float p1,p2;>>
        #( ADD poly>[p1] poly>[p2] )       <<r = p1+p2;>>

    |   <<float p1,p2;>>
        #( EXP poly>[p1] poly>[p2] )       <<r = pow(p1,p2);>>

    |   id:ID                              <<r = value(id->getText());>>
    |   f:FLOAT                           <<r = atof(f->getText());>>
    ;
```

Rule `poly` returns the floating point result as a return value. Starting with the simplest alternatives, you will note that for a floating point tree node, we simply compute the floating point value `(atof())` of the text found on the input stream for that token. For an identifier (register) node, we call our trigger function `value()` with the register identifier and return the result. The other alternatives of `poly` take the results of the two operands and perform the appropriate operation. Again, the actions must appear after the calls to the operands so that both results are available before being used.

Because rule `assign` will be called from the ANTLR grammar, no main program is required or specified in the SORCERER description. The entire specification is

```
#header <<
#include <math.h>
#include <stdio.h>
#include <string.h>
#include "tokens.h"
#include "AToken.h"
typedef ANTLRCommonToken ANTLRToken;
#include "AST.h"
typedef AST SORAST;
>>

class EvalPoly {
<<
protected:
    virtual float value(char *r) = 0;
    virtual void store(char *r, float v) = 0;
>>

assign
    :   <<float r;>>
        #( ASSIGN id:ID poly>[r] )
        <<
        store(id->getText(), r);
        printf("storing %f in %s\n", r, id->getText());
        >>
    ;

poly > [float r]
    :   <<float p1,p2;>>
        #( MULT poly>[p1] poly>[p2] )      <<r = p1*p2;>>

    |   <<float p1,p2;>>
        #( ADD poly>[p1] poly>[p2] )       <<r = p1+p2;>>

    |   <<float p1,p2;>>
        #( EXP poly>[p1] poly>[p2] )       <<r = pow(p1,p2);>>
```

```
    |   id:ID                                <<r = value(id->getText());>>
    |   f:FLOAT                              <<r = atof(f->getText());>>
    ;

}
```

Now the question is, how do we link our SORCERER tree-walker into our AST-constructing ANTLR grammar? First, we need to include the tree walker definition:

```
#header <<
// must be visible to all generated files; hence, must
// be in #header action
#include "AToken.h"
typedef ANTLRCommonToken ANTLRToken;
#include "MyEvalPoly.h"
>>
```

Second, we need to associate labels with the assignment and addition operator tokens so that they may be referenced in our SORCERER description.

```
#token ASSIGN      "="
#token ADD     "\+"
```

Third, and finally, we need to place a call in the `assign` ANTLR rule to invoke the SORCERER `assign` rule (which will evaluate the polynomial):

```
assign
    :   ID "="^ poly ";"!
        <<walker.assign((SORASTBase **)&#0);>>
    ;
```

where our tree walker is defined as a member variable of our parser:

```
class PolyParser {
<<
protected:
    MyEvalPoly walker;
>>
    ...
}
```

The action invokes the `assign` rule of the tree walker we have declared. The cast is required because SORCERER generates tree-walking functions that take generic `SORASTBase` tree pointers not `ASTBase` pointers (the type of `#0`).

The augmented ANTLR description is

```
#header <<
// must be visible to all generated files; hence, must put in #header
#include "AToken.h"
```

```
typedef ANTLRCommonToken ANTLRToken;
#include "MyEvalPoly.h"
>>

<<
#include "PBlackBox.h"
#include "DLGLexer.h"

main()
{
    ParserBlackBox<DLGLexer, PolyParser, ANTLRToken> p(stdin);
    ASTBase *root = NULL;
    p.parser()->interp(&root);
}
>>

#token "[\ \t]+" <<skip();>>
#token "\n"              <<skip(); newline();>>
#token ID  "[a-z]"
#token FLOAT "[0-9]+ { . [0-9]+ }"

#token EXP       "^"
#token ASSIGN    "="
#token ADD       "\+"
#token MULT

class PolyParser {

<<
protected:
    MyEvalPoly walker;
>>

interp!
    :   ( a:assign )+
    ;

assign
    :   ID "="^ poly ";"!
        <<walker.assign((SORASTBase **)&#0);>>
    ;

poly
    :   term ( "\+"^ term )*
    ;
```

```
term:bigterm
    |   reg { "^"^ exp }
    ;

bigterm!
    :   c:coefficient
        (   r:reg
            (   "^" e:exp
                <<#0 = #(#[MULT,"MULT"], #c, #(#[EXP,"EXP"], #r, #e));>>
            |   <<#0 = #(#[MULT,"MULT"], #c, #r);>>
            )
        |       <<#0 = #c;>>
        )
    ;

coefficient
    :   FLOAT
    ;

reg :   ID
    ;

exp :   reg
    |   FLOAT
    ;

}
```

The previous makefile created with

```
genmk -CC -class PolyParser -project poly -trees poly.g
```

for testing the AST construction can be modified for use with a SORCERER phase in the following ways:

- MAKE variables for SORCERER support code and binaries are added:

  ```
  SOR_H = $(PCCTS)/sorcerer/h
  SOR_LIB = $(PCCTS)/sorcerer/lib
  SOR = $(PCCTS)/sorcerer/sor
  ```

- `CFLAGS` variable is changed to add the SORCERER directory:

  ```
  CFLAGS = -I. -I$(ANTLR_H) -I$(SOR_H)
  ```

- `SRC` variable is changed to add the SORCERER phase and support file:

```
SRC = poly.cpp \
      PolyParser.cpp \
      $(ANTLR_H)/AParser.cpp $(ANTLR_H)/DLexerBase.cpp \
      $(ANTLR_H)/ASTBase.cpp $(ANTLR_H)/PCCTSAST.cpp \
      $(ANTLR_H)/ATokenBuffer.cpp $(SCAN).cpp \
      eval.cpp EvalPoly.cpp \
      $(SOR_LIB)/STreeParser.cpp
```

- `OBJ` variable is changed in a similar way.
- Target `poly.o` is changed to depend on the SORCERER phase include file:

```
poly.o : $(TOKENS) $(SCAN).h poly.cpp EvalPoly.h
        $(CCC) -c $(CFLAGS) -o poly.o poly.cpp
```

- Finally, we add targets for all the SORCERER output and support code:

```
EvalPoly.o : EvalPoly.cpp
        $(CCC) -c $(CFLAGS) EvalPoly.cpp

eval.o : eval.cpp
        $(CCC) -c $(CFLAGS) eval.cpp

eval.cpp EvalPoly.cpp EvalPoly.h : eval.sor
        $(SOR) -CPP eval.sor

STreeParser.o : $(SOR_LIB)/STreeParser.cpp
        $(CCC) -o STreeParser.o -c $(CFLAGS) \
                $(SOR_LIB)/STreeParser.cpp
```

Tree Transformations and Multiple SORCERER Phases

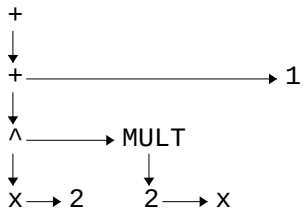
The previous example used SORCERER to evaluate polynomial assignments. We now consider how to use SORCERER to perform tree rewrites. We shall differentiate polynomials of the form:

ax^n + bx^m + ...

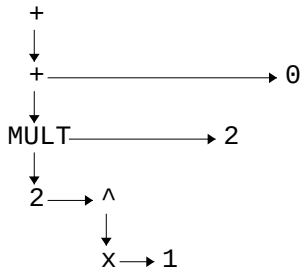
for integers *a*, *b*, *n* and *m*; we assume a single free variable *x*. As before, we will construct trees for the polynomials using an ANTLR grammar, but then manipulate the trees using three SORCERER phases. For example, given input

```
x^2 + 2x + 1
```

the ANTLR grammar constructs the tree



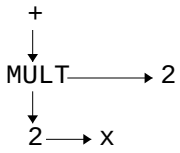
The first SORCERER phase differentiates the polynomial, yielding:



or

```
2x^1 + 2 + 0
```

The second phase normalizes the polynomial so that additions of zero are removed and exponents of one are removed:



or

```
2x + 2
```

The third and final phase prints the tree back out in polynomial form.

SORCERER will be used in *transform* mode where it makes the following assumptions:

- There is an input tree from which an output tree is derived.

- If given no instructions to the contrary, SORCERER automatically copies the input tree to the output tree.
- Each rule has a result tree, and the result tree of the first rule called is considered the final, transformed tree. This added functionality does not affect the normal rule argument and return value mechanism.
- Labels attached to grammar elements are generally referred to as *label*, where *label* refers to the input tree subtree in nontransform mode.

  The output tree in transform mode is referred to as *label*. The input node, *for token references only,* can be obtained with *label*`_in`. The input subtree associated with rule references is unavailable after the rule has been matched—the tree pointer points to where that rule left off parsing. Input nodes in transform mode are not needed very often.
- A C++ variable exists for any labeled token reference even if it is never set by SORCERER.
- The output tree of a rule can be set and/or referenced as #*rule*.

Tree Definition

To satisfy SORCERER in transform mode (in this situation), you must tell SORCERER how to construct new trees with `shallowCopy()` where `t->shallowCopy()` returns a duplicate of node `t` with all node pointers `NULL`. If you refer to `#[...]` in an action, you must also define a constructor with the appropriate arguments. The AST used in this application differs from the previous in that we have added

- An integer `iconst` field to make decrementing exponents easier
- Constructors for all `#[...]` node constructor references
- A definition of `shallowCopy()`

The AST is defined as follows:

```
#ifndef AST_h
#define AST_h

#include "ASTBase.h"
#include "AToken.h"
// use smart pointers ANTLRTokenPtr for garbage collection
#include "ATokPtr.h"

class AST : public ASTBase {
protected:
    ANTLRTokenPtr token;
    int iconst;
```

```
public:
/* These ctor are called when you ref node constructor #[tok,s] */
AST(ANTLRTokenType tok, char *s)
    {
        token = new ANTLRToken(tok, s);
        if ( token->getType() == INT )
             iconst = atoi(token->getText());
    }
AST(ANTLRTokenType tok, int i)
    {
        token = new ANTLRToken(tok, "");
        iconst = i;
    }
// called by ANTLR grammar during initial tree construction
AST(ANTLRTokenPtr t)
    {
        token = t;
        if ( token->getType() == INT )
        {
            iconst = atoi(token->getText());
        }
    }
AST(const AST &t)     // copy constructor
    {
        token = t.token;
        iconst = t.iconst;
        setDown(NULL);
        setRight(NULL);
    }
void preorder_action() {
    char *s = token->getText();
    if ( token->getType()==INT ) printf(" %d", iconst);
    else printf(" %s", s);
}
virtual int type() { return token->getType(); }
char *getText()    { return token->getText(); }
void setText(char *s) { token->setText(s); }
virtual PCCTS_AST *shallowCopy() { return new AST(*this); }
int getIConst() { return iconst; }
void setIConst(int i) { iconst = i; }
void decIConst() { iconst--; }
};

#endif
```

Building Trees For Differentiation

The grammar needed to build the ASTs is a slightly modified version of `poly.g` used in the previous examples. The differences are that assignments are not recognized and coefficients and exponents must be simple integers. (Support code differences are commented in the grammar.)

```
<<
#include “PBlackBox.h"
#include "DLGLexer.h"
typedef ANTLRCommonToken ANTLRToken;

#include "AST.h"
#include "DiffPoly.h"          // include the .h files for 3 phases
#include "SimplifyPoly.h"
#include "PrintPoly.h"

main()
{
    ParserBlackBox<DLGLexer, PolyParser, ANTLRToken> p(stdin);
    ASTBase *root = NULL;
    p.parser()->poly(&root)
}
>>

#token "[\ \t]+"<<skip();>>
#token "\n" <<skip(); newline();>>

#token EXP "^"
#token ADD "\+"                // def used by SORCERER phases
#token MULT                    // used by SORCERER phases

class PolyParser {

poly
    :  <<
       AST *result=NULL, *nresult=NULL;
       DiffPoly dp;                // define the 3 phases
       PrintPoly pp;
       SimplifyPoly sp;
       >>
       term ( "\+"^ term )*
       <<
       // execute the 3 phases, creating new tree after phase 1,2
       dp.poly((SORASTBase **)&(#0), (SORASTBase **)&result);
       sp.poly((SORASTBase **)&result, (SORASTBase **)&nresult);
```

```
        pp.poly((SORASTBase **)&nresult);
        printf("\n");
        >>
    ;

term:bigterm
    |   reg { "^"^ INT }
    ;

bigterm!
    :   c:coefficient
        (   r:reg
            (   "^" e:exp
                <<#0 = #(#[MULT,"MULT"], #c, #(#[EXP,"EXP"], #r, #e));>>
            |   <<#0 = #(#[MULT,"MULT"], #c, #r);>>
            )
        |       <<#0 = #c;>>
        )
    ;

coefficient
    :   INT
    ;

reg :   ID
    ;

exp :   INT
    ;

}

#token ID     "[a-z]"
#token INT "[0-9]+"
```

Differentiation Phase

Differentiating our polynomial trees follows these rules:

TABLE 2. Differentiation of Polynomial Trees

Isolated integers	Set the value to 0.
Isolated identifiers	Replace with an integer node whose value is 1
Terms with exponents and no coefficient	Make a new tree with a multiply at the root, the previous exponent as the first child, and the previous term as the second child. Decrement the exponent of the term.
Terms with no exponent, but with coefficient	Replace the term with the coefficient node.
Terms with coefficients and exponents	Multiply the exponent into the coefficient and decrement the exponent.

The complete SORCERER diffierentiation grammar looks like this:

```
#header <<
#include <math.h>
#include <stdio.h>
#include <string.h>
#include "tokens.h"
#include "AToken.h"
typedef ANTLRCommonToken ANTLRToken;
#include "AST.h"
typedef AST SORAST;
>>

class DiffPoly {

poly:   #( ADD poly poly )
    |   term
    ;

term!:  i:INT                    <<#term = #[INT,"0"];>>
    |   id:ID                    <<#term = #[INT,"1"];>>
    |   #( ex:EXP id:ID e:INT )
        <<
    #term = #(#[MULT,"MULT"], #[INT,e->getIConst()], #(ex,id,e));
        e->decIConst();             // decrement exponent
```

```
        >>
    |   #(  m:MULT ct:INT
          (      #( ex:EXP idt:ID et:INT )
                 <<
                 // just reset the integer values everywhere
                 ct->setIConst(ct->getIConst()*et->getIConst());
                 et->decIConst();
                 #term = #(m, ct, #(ex,idt,et));
                 >>
           |     ID
                 <<#term = ct;>> // just return the INT node
           )
        )
    ;

}
```

The action

```
#term = #[INT,"0"];
```

sets the output tree for rule `term` to be a single node (using the `#[...]` node constructor). The action

```
#term = #(#[MULT,"MULT"], #[INT,e->getIConst()], #(ex,id,e));
```

creates a tree with a new `MULT` node as the root, a new `INT` node as the first child and the previous term, `#(ex,id,e),` because the second child where nodes labeled by `ex, id` and `e` are duplicates made automatically from the input nodes.

Simplification Phase

Differentiation can leave unusual terms such as additions of zero and exponents of one. A SORCERER phase to simply polynomials is useful. The following simple rules are used:

TABLE 3. Simplification of Polynomial Trees

Addition of 0+0	Return NULL.
Addition of integer with value 0 to any other term, *t*	Return *t*.
Terms *t* with exponents of value 1	Return just *t*.

The following SORCERER transform mode phase implements these rules.

```
#header <<
#include <math.h>
#include <stdio.h>
#include <string.h>
#include "tokens.h"
#include "AToken.h"
typedef ANTLRCommonToken ANTLRToken;
#include "AST.h"
typedef AST SORAST;
>>

class SimplifyPoly {

poly:   #( MULT poly poly )
    |!  #( a:ADD p:poly q:poly )
        <<
        if ( p==NULL ) #poly = q;
        else if ( q==NULL ) #poly = p;
        else {
              int leftIdentity=( p->type()==INT && p->getIConst()==0 );
              int rightIdentity=( q->type()==INT && q->getIConst()==0);
              if ( leftIdentity && !rightIdentity ) #poly = q; //0+x
              else if ( !leftIdentity && rightIdentity ) #poly = p; //x+0
              else if (!leftIdentity&&!rightIdentity) #poly = #(a,p,q);
              else #poly = NULL; //0+0
        }
        >>
    |!  #( e:EXP v:poly ex:INT )
        <<
        if ( ex->getIConst()==1 ) #poly = v;
        else #poly = #(e,v,ex);
        >>
    |   id:ID
    |   i:INT
    ;

}
```

Printing Phase

After simplification, the only remaining task is to print the differentiated tree back out in polynomial form, which is done in SORCERER nontransform mode:

```
#header <<
#include <math.h>
#include <stdio.h>
#include <string.h>
#include "tokens.h"
#include "AToken.h"
typedef ANTLRCommonToken ANTLRToken;
#include "AST.h"
typedef AST SORAST;
>>

class PrintPoly {

poly:   #( MULT poly poly )
    |   #( ADD poly <<printf("+");>> poly )
    |   #( EXP poly <<printf("^");>> poly )
    |   id:ID   <<printf("%s",id->getText());>>
    |   i:INT   <<printf("%d",i->getIConst());>>
    ;

}
```

The order of action execution is important. Because we wish to print out polynomials in infix notation (as opposed to postfix, for example), we insert the

```
printf("+");
```

action in between the printing of the two operands of the `ADD` root node.

Makefile

The following makefile has targets for the parser and three tree-walking phases. This was initially generated by the `genmk` program, but was modified by hand for the SORCERER targets.

```
#
# PCCTS makefile for: poly.g
#
# Created from: genmk -CC -class PolyParser -project poly -trees poly.g
#
```

```
# PCCTS release 1.32
# Project: poly
# C++ output
# DLG scanner
# ANTLR-defined token types
#
TOKENS = tokens.h
#
# The following filenames must be consistent with ANTLR/DLG flags
DLG_FILE = parser.dlg
ERR = err
HDR_FILE =
SCAN = DLGLexer
PCCTS = /projects/pccts
ANTLR_H = $(PCCTS)/h
SOR_H = $(PCCTS)/sorcerer/h
SOR_LIB = $(PCCTS)/sorcerer/lib
BIN = $(PCCTS)/bin
ANTLR = $(BIN)/antlr
DLG = $(BIN)/dlg
SOR = $(PCCTS)/sorcerer/sor
CFLAGS = -I. -I$(ANTLR_H) -g -I$(SOR_H)
AFLAGS = -CC -gt
DFLAGS = -C2 -i -CC
GRM = poly.g
SRC = poly.cpp \
    PolyParser.cpp \
    $(ANTLR_H)/AParser.cpp $(ANTLR_H)/DLexerBase.cpp \
    $(ANTLR_H)/ASTBase.cpp $(ANTLR_H)/PCCTSAST.cpp \
    $(ANTLR_H)/ATokenBuffer.cpp $(SCAN).cpp \
    diff.cpp DiffPoly.cpp print.cpp PrintPoly.cpp simplify.cpp \
    SimplifyPoly.cpp \
    $(SOR_LIB)/STreeParser.cpp
OBJ = poly.o \
    PolyParser.o \
    AParser.o DLexerBase.o \
    ASTBase.o PCCTSAST.o \
    ATokenBuffer.o $(SCAN).o \
    diff.o DiffPoly.o print.o PrintPoly.o simplify.o SimplifyPoly.o \
    STreeParser.o
ANTLR_SPAWN = poly.cpp PolyParser.cpp \
    PolyParser.h $(DLG_FILE) $(TOKENS)
DLG_SPAWN = $(SCAN).cpp $(SCAN).h
CCC=g++
CC=$(CCC)
```

```
poly : $(OBJ) $(SRC)
    $(CCC) -o poly $(CFLAGS) $(OBJ)

poly.o : $(TOKENS) $(SCAN).h poly.cpp PrintPoly.h DiffPoly.h
SimplifyPoly.h
    $(CCC) -c $(CFLAGS) -o poly.o poly.cpp

PolyParser.o : $(TOKENS) $(SCAN).h PolyParser.cpp PolyParser.h
    $(CCC) -c $(CFLAGS) -o PolyParser.o PolyParser.cpp

$(SCAN).o : $(SCAN).cpp $(TOKENS)
    $(CCC) -c $(CFLAGS) -o $(SCAN).o $(SCAN).cpp

$(ANTLR_SPAWN) : $(GRM)
    $(ANTLR) $(AFLAGS) $(GRM)

$(DLG_SPAWN) : $(DLG_FILE)
    $(DLG) $(DFLAGS) $(DLG_FILE)

AParser.o : $(ANTLR_H)/AParser.cpp
    $(CCC) -c $(CFLAGS) -o AParser.o $(ANTLR_H)/AParser.cpp

ATokenBuffer.o : $(ANTLR_H)/ATokenBuffer.cpp
    $(CCC) -c $(CFLAGS) -o ATokenBuffer.o $(ANTLR_H)/ATokenBuffer.cpp

DLexerBase.o : $(ANTLR_H)/DLexerBase.cpp
    $(CCC) -c $(CFLAGS) -o DLexerBase.o $(ANTLR_H)/DLexerBase.cpp

ASTBase.o : $(ANTLR_H)/ASTBase.cpp
    $(CCC) -c $(CFLAGS) -o ASTBase.o $(ANTLR_H)/ASTBase.cpp

PCCTSAST.o : $(ANTLR_H)/PCCTSAST.cpp
    $(CCC) -c $(CFLAGS) -o PCCTSAST.o $(ANTLR_H)/PCCTSAST.cpp

#
# SORCERER crud
#
PrintPoly.o : PrintPoly.cpp
    $(CCC) -c $(CFLAGS) PrintPoly.cpp

print.o : print.cpp
    $(CCC) -c $(CFLAGS) print.cpp

print.cpp PrintPoly.cpp PrintPoly.h : print.sor
    $(SOR) -CPP print.sor
```

```
DiffPoly.o : DiffPoly.cpp
    $(CCC) -c $(CFLAGS) DiffPoly.cpp

diff.o : diff.cpp
    $(CCC) -c $(CFLAGS) diff.cpp

diff.cpp DiffPoly.cpp DiffPoly.h : diff.sor
    $(SOR) -transform -CPP diff.sor

SimplifyPoly.o : SimplifyPoly.cpp
    $(CCC) -c $(CFLAGS) SimplifyPoly.cpp

simplify.o : simplify.cpp
    $(CCC) -c $(CFLAGS) simplify.cpp

simplify.cpp SimplifyPoly.cpp SimplifyPoly.h : simplify.sor
    $(SOR) -transform -CPP simplify.sor

STreeParser.o : $(SOR_LIB)/STreeParser.cpp
    $(CCC) -o STreeParser.o -c $(CFLAGS) $(SOR_LIB)/STreeParser.cpp

clean:
    rm -f *.o core poly

scrub:
    rm -f *.o core poly $(ANTLR_SPAWN) $(DLG_SPAWN)
```

3 ANTLR Reference

This chapter tells you what you need to know so you can construct parsers via ANTLR grammars, how to interface a parser to your application, and how to insert actions to generate output. Unless otherwise specified, actions and other source code is C++.

[*Professors Russell Quong, Hank Dietz, and Will Cohen all have contributed greatly to the overall development of PCCTS in general. In particular, much of the intellectual property of ANTLR was conceived with Russell Quong.*]

ANTLR Descriptions

Generally speaking, an ANTLR description consists of a collection of lexical and syntactic rules describing the language to be recognized and a collection of user-defined semantic actions describing what to do with the input sentences as they are recognized. A single grammar may be broken up into multiple files and multiple grammars may be specified within a single file, but the basic sequence follows something like:

```
header action
actions
token definitions
rules
actions
token definitions
```

For example, the following is a complete ANTLR description that recognizes the vocabulary of B. Simpson:

```
<<
typedef ANTLRCommonToken ANTLRToken;
#include "DLGLexer.h"
#include "PBlackBox.h"
main() {
    ParserBlackBox<DLGLexer,              // create a parser
                   BSimpsonParser,
                   ANTLRToken> bart(stdin);
    bart.parser()->a();                   // invoke parser
}
>>

#token     "[\ \t\n]+"     <<skip();>>   // ignore whitespace
#token MAN "man"

class BSimpsonParser {
a   :   "no" "way" MAN
    |   "don't" "have" "a" "cow" "man"
    ;
}
```

More precisely, ANTLR descriptions conform to the following grammar:

```
grammar
    :   ( "#header" ACTION
        | "#parser" STRING
        | "#tokdefs" STRING
        )*
        { "class" ID "\{" }
        ( ACTION | lexaction | directive | global_exception_handler )*
        ( rule | directive )+
        ( ACTION | directive )*
        { "\}" }
        ( ACTION | directive )*
    ;

directive
    :   lexclass | token_def | errclass_def | tokclass_def
    ;
```

where the lexical items in Table 4 on page 83 apply:

There is no start rule specification *per se* because any rule can be invoked first.

TABLE 4. Lexical Items in an ANTLR Description

Token Name	Form	Example
ACTION	<<...>>	<<int i;>> <<define(id->getText());>>
STRING	"..."	"[a-z]+" "begin" "includefile.h" "test"
TOKEN	*[A-Z][a-zA-Z0-9_]**	ID KeyBegin Int_Form1
RULE	*[a-z][a-zA-Z0-9_]**	expr statement func_def
ARGBLOCK	[...]	[34] [int i, float j]
ID	*[a-zA-Z][a-z0-9_]**	CParser label
SEMPRED	<<...>>?	<<isType(id->getText())>>?

Comments

Both C and C++ style comments are allowed within the grammar (outside of actions) regardless of the language used within actions. For example,

```
/* here is a rule */
args : ID ( "," ID )* ;           // match a list of ID's
```

The comments used within your actions is determined by your language.

#header Directive

Any C or C++ code that must be visible to files generated by ANTLR must placed in an action at the start of your description preceded by the `#header` directive. This directive is necessary when using the C interface and is optional with the C++ interface. Turn on ANTLR command line option `-gh` when using the C interface if the function that invokes the parser is in a non-ANTLR-generated file.

#parser Directive

Because C does not have the notion of a package or module, linking ANTLR-generated parser causes multiply defined symbol errors (because of the global variables defined in

each parser). The solution to the problem is to prefix all global ANTLR symbols with a user-defined string in order to make the symbols unique to the C linker. The `#parser` is used to specify this prefix. A file called `remap.h` is generated that contains a sequence of redefinitions for the global symbols. For example,

```
#parser foo
```

generates a `remap.h` file similar to:

```
#define your_rule       foo_your_rule
#define zztokenLA       xyz_zztokenLA
#define AST             xyz_AST
...
```

Parser Classes

When using the C++ interface, you must specify the name of the parser class by enclosing all rules in

```
class Parser {
    ...
}
```

A parser class results in a subclass of `ANTLRParser` in the parser. A parser object is simply a set of actions and routines for recognizing and performing operations on sentences of a language. Consequently, it is natural to have many separate parser objects; for example, one for recognizing include files.

Exactly one parser class may be defined. For the defined class, ANTLR generates a derived class of `ANTLRparser`.

Actions may be placed within the parser class scope and may contain any C++ code that is valid within a C++ class definition. Any variable or function declarations will become members of the class in the resulting C++ output. For example,

```
class Parser {
   <<public: int i;>>
   <<int f() { blah; }>>

rule : A B <<f();>> ; <<fail-action for rule;>>

   <<final action;>>
}
```

Here, variable `i` and function `f` are members of class `Parser` that become a subclass of `ANTLRParser` in the resulting C++ code.

The actions at the head of the parser class are collected and placed near the head of the resulting C++ class; the actions at the end of the parser class are similarly collected and placed near the end of the resulting parser class definition.

The `Parser.h` file generated by ANTLR for this parser class would look something like this:

```
class Parser : public ANTLRParser {
protected:
        static ANTLRChar *_token_tbl[];
public: int i;
int f() { blah; }
        static SetWordType setwd1[4];
public:
        Parser(ANTLRTokenBuffer *input);
        Parser(ANTLRTokenBuffer *input, ANTLRTokenType eof);
        void rule(void);
        final action;
};
```

Rules

An ANTLR rule describes a portion of the input language and consists of a list of alternatives; rules also contain code for error handling and argument or return value passing. A rule looks like:

```
rule   :  alternative1
       |  alternative2
          ...
       |  alternativen
       ;
```

where each alternative production is composed of a list of elements that can be references to rules, references to tokens, actions, predicates, and subrules. Argument and return value definitions looks like the following where there are *n* arguments and *m* return values:

```
rule[arg1,...,argn] > [retval1,...,retvalm] : ... ;
```

The syntax for using a rule mirrors its definition:

```
a   :   ... rule[arg1,...,argn] > [v1,...,vm] ...
    ;
```

Here, the various v_i receive the return values from the rule `rule`, each v_i must be an l-value. For example,

```
start
   :  <<int r;>>              // init-action declares local var r
```

```
        expr[3,4] > [r]         <<printf("result %d\n");>>
    ;
expr[int a, int b] > [int result]
    :   i:INT <<$result = $a+$b+atoi($i->getText());>>
    ;
```

The reference to rule `expr` in rule `start` passes two arguments, `3` and `4`, which correspond to `a` and `b` in rule `expr` just like a normal programming language. The return value of expr is an integer called `result`, which is set in the action. The integer value of the text for the incoming integer is added to the two arguments to compute the result.

We make special note of the first action of rule `start`: if the first element of the rule is an action, that action is an init-action and is executed once before recognition of the rule begins and is the place to define local variables.

The exact syntax of a rule is the following:

```
rule
    :   RULE { "!" } { ARGBLOCK }
        { ">" ARGBLOCK }
        { STRING }       // error string to print instead of rule name
        ":"
        block ";"
        { ACTION }       // fail action
        ( exception_group )*
    ;

block
    :   alt ( exception_group )* ( "\|" alt ( exception_group )* )*
    ;

alt :   { "\@" } ( { "\~" } element )*
    ;

token: TOKEN | STRING
    ;

element
    :   { ID ":" }
        (   token { ".." token } { "^" | "!" } { "\@" }
        |   "." { "^" | "!" }
        |   RULE { "!" } { ARGBLOCK } { ">" ARGBLOCK }
        )
    |   ACTION        // <<...>>
    |   SEMPRED       // <<...>>?
    |   "\(" block "\)" { "\*" | "\+" | "?" }
```

```
|   "\{" block "\}"
;
```

Subrules (EBNF Descriptions)

A subrule is the same as a rule without a label and, hence, has no arguments or return values. The four subrules to choose from are listed in Table 5 on page 87.

TABLE 5. ANTLR Subrule Format

Name	Form	Example
plain subrule	`(...)`	`(ID | INT)`
zero-or-more	`(...)*`	`ID ( “,” ID )*`
one-or-more	`(...)+`	`( declaration )+`
optional	`{...}`	`{ “else” statement }`

If the first element of the whole subrule is an action, that action is an init-action and is executed once before recognition of the subrule begins—even if the subrule is a looping construct. Further, the action is always executed even if the subrule matches nothing.

Rule Elements

In this section, we summarize the elements that can appear in rules. Most elements (i.e., predicates, actions, tree operators, and exceptions) are described in more detail later.

Actions

Actions are of the form `<<...>>` and contain user-supplied C or C++ code that must be executed during the parse. Init-actions are actions that are the very first element of a rule or subrule; they are executed before the rule or subrule recognizes anything and can be used to define local variables. Fail-actions are placed after the ‘`;`’ in a rule definition and are executed if an error occurred while parsing the rule (unless exception handlers are used). Any other action is executed immediately after the preceding rule element and before any following elements.

Semantic Predicate

A semantic predicate has two forms:

- `<<...>>?` This form represents a C or C++ expression that must evaluate to true before recognition of elements beyond it in the rule are authorized for recognition.
- `( lookahead-context )? => <<...>>?` This form is simply a more specific form as it indicates that the predicate is only valid under a particular lookahead context; e.g., the following predicate indicates that the `isTypeName()` test is only valid if the first symbol of lookahead is an identifier:

```
( ID )? => <<isTypeName(LT(1)->getText())>>?
```

Typically, semantic predicates are used to specify the semantic validity of a particular production and, therefore, most often are placed at the extreme left edge of productions.

You should normally allow ANTLR to compute the lookahead context (ANTLR command line option "`-prc on`"). See "Predicates" on page 127.

Syntactic Predicate

Syntactic predicates are of the form `(...)?` specify the syntactic context under which a production will successfully match. They are useful in situations where normal LL(k) parsing is inadequate. For example,

```
a   :   ( list "=" )? list "=" list
    |   list
    ;
```

Tokens, Token Classes and Token Operators

Token references indicate the token that must be matched on the input stream and are either identifiers beginning with an upper case letter or are regular expressions enclosed in double quotes. A token class looks just like a token reference, but has an associated `#tokclass` definition and indicates the set of tokens that can be matched on the input stream.

The *range* operator has the form T_1 `..` T_n and specifies a token class containing the set of token type values from T_1 up to T_n inclusively. Any token found on the input stream that is contained in this set is considered a valid match.

The *not* operator has the form ~*T* and specifies the set of all tokens defined in the grammar except for *T*.

Rule References

Rule references indicate that another rule must be invoked to recognize part of the input stream. The rule may be passed to some arguments and may return some values. Rules are indentifiers that begin with a lower case letter. For example,

```
a   :   <<int i;>> b[34] > [i]
    ;
b[int j] > [int k]
    :   A B <<$k = $j + 1;>>     //return argument + 1
    ;
```

Labels

All rules, token, and token class references may be labeled with an identifier. Identifiers are generally used to access the attribute (C interface) or token object (C++ interface) of tokens. Rule labels are used primarily by the exception handling mechanism to make a group of handlers specific to a rule invocation.

Labels may begin with either an upper or lower case letter; e.g., `id:ID ER:expr`.

Actions in an ANTLR grammar may access attributes by using labels of the form `$`*`label`* attached to token rather than the conventional $*i* for some integer *i*. By using symbols instead of integer identifiers, grammars are more readable and action parameters are not sensitive to changes in rule element positions. The form of a label is:

```
label:element
```

where `element` is either a token reference or a rule reference. To refer to the attribute (C interface) or token pointer (C++ interface) of that element in an action, use

```
$label
```

within an action or rule argument list. For example,

```
a : t:ID <<printf("%s\n", $t->getText());>>
  ;
```

using the C++ interface. To reference the tree variable associated with *`element`*, use

```
#label
```

When using parser exception handling, simply reference *`label`* to attach a handler to a particular rule reference. For example,

```
a : t:b
    exception[t]
        default : <<trap any error found during call to 'b'>>
  ;
```

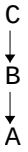
Labels must be unique for each rule as they have rule scope. Labels may be accessed from parser exception handlers.

AST Operators

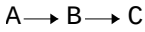
When constructing ASTs, ANTLR assumes that any nonsuffixed token is a leaf node in the resulting tree. To inform ANTLR that a particular token should not be included in the output AST, suffix the token with "`!`." Rules may also be suffixed with "`!`" to indicate that the tree constructed by the invoked rule should not be linked into the tree constructed for the current rule. Any token suffixed with the "`^`" operator is considered a root token. A tree node is constructed for that token and is made the root of whatever portion of the tree has been built; e..g,

```
a : A B^ C^ ;
```

results in the following tree:



First `A` is matched and made a lonely child, followed by `B` which is made the parent of the current tree, `A`. Finally, `C` is matched and made the parent of the current tree—making it the parent of the `B` node. Note that the same rule without any operators results in:



Exception Operator

When parser exception handlers are being used in a grammar, token references suffixed with the `@` operator do not throw `MismatchedToken` upon a token mismatch. The error is handled within `_match_wdfltsig()`.

Multiple ANTLR Description Files

ANTLR descriptions may be broken up into many different files, but the sequence mentioned above in the grammatical structure of ANTLR descriptions must be maintained.

For example, if file `f1.g` contained

```
#header <<#include "int.h">>
<< main() { ANTLR(start(), stdin); } >>
```

and file f2.g contained

```
start   :   "begin" VAR "=" NUM ";" "end" "." "@" ;
```

and file f3.g contained

```
#token VAR "[a-z]+"
#token NUM "[0-9]+"
```

the correct ANTLR invocation would be

```
antlr f1.g f2.g f3.g
```

Note that the order of files f2.g and f3.g could be switched. In this case, to comply with ANTLR's description meta-language, the only restriction is that file f1.g must be mentioned first on the command line.

Other files may be included into the parser files generated by ANTLR via actions containing a #include directive. For example,

```
<<#include "support_code.h">>
```

If a file (or anything else) must be included in all parser files generated by ANTLR, the #include directive must be placed in the #header action. In other words,

```
#header <<#include "necessary_type_defs_for_all_files.h">>
```

Note that #include can be used to define any ANTLR object (Attrib, AST, etc...) by placing it in the #header action.

Lexical Directives

Token Definitions

Tokens are defined either explicitly with #token or implicitly by using them as rule elements. Implicitly defined tokens can be either regular expressions (non-identified tokens) or token names (identified). Token names begin with an upper case letter (rules begin with a lower case letter). More than one occurrence of the same regular expression in a grammar description produces a single regular expression in lexical description passed to DLG (parser.dlg) and is assigned one token type number. Regular expressions and token identifiers that refer to the same lexical object (input pattern) may be used interchangeably. Token identifiers that are referenced, but not attached to a regular expression are simply assigned a token type and result in a #define definition only. It is not necessary to label regular expressions with an identifier in ANTLR. However, all token types that you wish to explicitly refer to in an action must be declared with a #token instruction.

You may introduce tokens, lexical actions, and token identifiers with the `#token` directive. Specifically,

- Simply declare a token for use in a user action:

  ```
  #token VAR
  ```

  This is useful for defining a token type that has no associated regular expression. For example, an abstract syntax tree may need a "dummy" node with a token type that does not class with an input token.

- Associate a token with a regular expression and, optionally, an action:

  ```
  #token ID "[a-zA-Z][a-zA-Z0-9]*"
  #token Eof "@" << printf("Eof Found\n"); >>
  ```

- Specify what must occur upon a regular expression:

  ```
  #token "[0-9]+" <<printf("Found an int\n");>>
  ```

Important: All token identifiers result in either `#define` definitions or enum elements in the resulting parser. Be careful not to use C++ keywords as token identifiers like `if`.

Lexical actions tied to a token definition may access the variables, functions, and macros in Table 6 on page 92:

TABLE 6. C++ Interface Symbols Available to Lexical Actions

Symbol	Description
`replchar(DLGchar c)`	Replace the text of the most recently matched lexical object with `c`. You can erase the current expression text by sending in a `'\0'`.
`replstr(DLGchar *s)`	Replace the text of the most recently matched lexical object with `s`.
`int line()`	The current line number being scanned by DLG.
`newline()`	Maintain `DLGLexer::_line` by calling this function when a newline character is seen; just increments `_line`.
`more()`	Set a flag that tells DLG to continue looking for another token; future characters are appended to the current token text.
`skip()`	Set a flag that tells DLG to continue looking for another token; future characters are not appended to the current token text.

TABLE 6. (Continued) C++ Interface Symbols Available to Lexical Actions

Symbol	Description
`advance()`	Instruct DLG to consume another input character. `ch` will be set to this next character.
`int ch`	The most recently scanned character.
`DLGchar *lextext()`	The entire lexical buffer containing all characters matched thus far since the last token type was returned. See `more()` and `skip()`.
`DLGchar *begexpr()`	Beginning of last token matched.
`DLGchar *endexpr()`	Pointer to the last character of last token matched.
`trackColumns()`	Call this function to get DLG to track the column numbers.
`int begcol()`	The column number starting from 1 of the first character of the most recently matched token.
`int endcol()`	The column number starting from 1 of the last character of the most recently matched token. Reset the column to 0 when a newline character is encountered. Also adjust the column in the lexical action when a character is not one print position wide (e.g., tabs or non-printing characters). The column information is not immediately updated if a token's action calls `more()`.
`set_begcol(int a)`	Set the current token column number for the beginning of the token.
`set_endcol(int a)`	Set the current token column number for the beginning of the token.
`DLGchar`	The type name of a character read by DLG. This is linked by `typedef` to `char` by default, but it could be a class or another atomic type.
`errstd(char *)`	Called automatically by DLG to print an error message indicating that the input text matches no defined lexical expressions. Override in a subclass to redefine.

TABLE 6. (Continued) C++ Interface Symbols Available to Lexical Actions

Symbol	Description
`mode(int m)`	Set the lexical mode (i.e., lexical class or automaton) corresponding to a lex class defined in an ANTLR grammar with the `#lexclass` directive.
`setInputStream( DLGInputStream *)`	Specify that the scanner should read characters from the indicated input stream (e.g., file, string, function).
`saveState(DLGState *)`	Save the current state of the scanner. You need this function for include files and so on; i.e., save the state of DLG, reset the file pointer, process the other file, and then restore the state.
`restoreState( DLGState *)`	Restore the state of the scanner from a state buffer.

Regular Expressions

The input character stream is broken up into vocabulary symbols (tokens) via regular expressions—a meta-language similar to the ANTLR EBNF description language. ANTLR collects all of the regular expressions found within your grammar (both those defined implicitly within the grammar and those defined explicitly via the `#token` directive) and places them in a file that is converted to a scanner by DLG. Table 7 on page 94 describes the set of regular expressions.

TABLE 7. Regular Expression Syntax

Expression	Description
a`|`*b*	Matches either the pattern *a* or the pattern *b*.
`(a)`	Matches the pattern *a*. Pattern *a* is kept as an indivisible unit.
`{a}`	Matches *a* or nothing, i.e., the same as (*a*`|`).
`[a]`	Matches any single character in character list *a*; e.g., `[abc]` matches either an `a`, `b` or `c` and is equivalent to `(a|b|c)`.
`[a-b]`	Matches any of the single characters whose ASCII codes are between *a* and *b* inclusively, i.e., the same as (*a*`|...|`*b*).

TABLE 7. (Continued) Regular Expression Syntax

Expression	Description
`~[a]`	Matches any single character except for those in character list *a*.
`~[]`	Matches any single character; literally "not nothing."
`a*`	Matches zero or more occurrences of pattern *a*.
`a+`	Matches one or more occurrences of pattern *a*, i.e., the same as *aa**.
`@`	Matches end-of-file.
`\t`	Tab character.
`\n`	Newline character.
`\r`	Carriage return character.
`\b`	Backspace character.
`\a`	Matches the single character *a*—even if *a* by itself would have a different meaning, e.g., `\+` would match the + character.
`\0nnn`	Matches character that has octal value *nnn*.
`\0x`*nn*	Matches character that has hexadecimal value *nnn*.
`\`*mnn*	Matches character with decimal value *mnn*, 1≤m≤9.

Token Order and Lexical Ambiguities

The order in which regular expressions are found in the grammar description file(s) is significant. When the input stream contains a sequence of characters that match more than one regular expression, (i.e., one regular expression is a subset of another) the scanner is confronted with a dilemma. The scanner does not know which regular expression to match, so it does not know which action should be performed. To resolve the ambiguity, DLG (the scanner generator) assumes that the regular expression defined earliest in the grammar should take precedence over later definitions. Therefore, tokens that are special cases of other regular expressions *should* be defined before the more general regular expressions. For example, a keyword is a special case of a variable and thus needs to occur before the variable definition.

```
#token KeywordBegin "begin"
   ...
#token ID "[a-zA-Z][a-zA-Z0-9]*"
```

Token Definition Files (#tokdefs)

You will probably be interested in specifying the token types rather than having ANTLR generate its own; typically, this situation arises when you want to link an ANTLR-generated parser with a non-DLG-based scanner (perhaps an existing scanner). To get ANTLR to use pre-assigned token types, specify

```
#tokdefs "mytokens.h"
```

before any token definitions, where `mytokens.h` is a file with only a list of `#define`s or an `enum` definition with optional comments.

When this directive is used, new token identifier definitions are not allowed (either explicit definitions like "`#token A`" or implicit definitions such as a reference to a token label in a rule). However, you may attach regular expressions and lexical actions to the token labels defined in `mytokens.h`. For example, if `mytokens.h` contained:

```
#define A 2
```

and `t.g` contained:

```
#tokdefs "mytokens.h"
#token A "blah"
a : A B;
```

ANTLR would report the following error message:

```
Antlr parser generator   Version 1.32   1989-1995
t.g, line 3: error: implicit token definition not allowed with #tokdefs
```

This refers to the fact that token identifier `B` was not defined in `mytokens.h` and ANTLR has no idea how to assign the token identifier a token type number.

Only one token definition file is allowed.

As is common in C and C++ programming, "gates" are used to prevent multiple inclusions of include files. ANTLR knows to ignore the following two lines at the head of a token definition file:

```
#ifndef id1
#define id2
```

No check is made to ensure that id_1 and id_2 are the same or that they conform to any particular naming convention (such as the name of the file suffixed with "`_H`").

The following items are ignored inside your token definition file: white space, C style comments, C++ style comments, `#ifdef`, `#if`, `#else`, `#endif`, `#undef`, `#import`. Anything other than these ignored symbols, `#define`, `#ifndef`, or a valid `enum` statement yield lexical errors.

Token Classes

A token class is set of tokens that can be referenced as one entity; token classes are equivalent to subrules consisting of the member tokens separated by "|"s. The basic syntax is:

```
#tokclass Tclass { T1 ... Tn }
```

where *Tclass* is a valid token identifier (begins with an upper case letter) and T_i is a token reference (either a token identifier or a regular expression in double-quotes) or a token class reference; token classes may have overlapping tokens. Referencing *Tclass* is the same as referencing a rule of the form

```
tclass : T1 | ... | Tn ;
```

To reference the bitset created for token class *Tclass* in a grammar action is done as *Tclass*_set; e.g.,

```
#tokclass stop { ";" "end" }
statement
    :  ... ;
    exception
       default: <<consumeUntil(stop_set);>>
```

The difference between a token class and a rule lies in efficiency. A reference to a token class is a simple set membership test during parser execution rather than a linear search of the tokens in a rule (or subrule). Furthermore, the set membership will be much smaller than a series of if-statements in a recursive-descent parser. Note that automaton-based parsers (both LL and LALR) automatically perform this type of set membership (specifically, a table lookup), but lack the flexibility of recursive-descent parsers such as those constructed by ANTLR.

A predefined wildcard token class, identified by a dot, is available to represent the set of all defined tokens. For example,

```
ig :  "ignore_next_token" . ;
```

The wildcard is sometimes useful for ignoring portions of the input stream; however, lexical classes are often more efficient at ignoring input. A wildcard can also be used for error handling as an "else-alternative".

```
if  :   "if" expr "then" stat
    |   .  <<fprintf(stderr, "malformed if-statement");>>
    ;
```

Be careful not to do things like this:

```
ig  :   "begin"
        ( . )*
```

```
        "end"
    ;
```

because the loop generated for the "`(.)*`" block will never terminate because "`end`" is also matched by the wildcard. Rather than using the wildcard to match large token classes, it is often best to use the *not* operator. For example,

```
ig  :   "begin"
        ( ~"end" )*
        "end"
    ;
```

where "~" is the *not* operator and implies a token class containing all tokens defined in the grammar except the token (or tokens in a token class) modified by the operator. The `if` example could be rewritten as:

```
if  :   "if" expr "then" stat
    |   ~"if" <<fprintf(stderr, "malformed if-statement");>>
    ;
```

The *not* operator may be applied to token class references and token references only. It may not be applied to subrules, for example. The wildcard operator and the *not* operator never result in a set containing the end-of-file token type.

Token classes can also be created via the *range* operator of the form T_1 `..` T_n. The token type of T_1 must be less than T_n and the values between T_1 and T_n must be valid token types. In general, this feature should be used in conjunction with `#tokdefs` so that you control the token type values. An example range operator is:

```
#tokdefs "mytokens.h"
a : operand OpStart .. OpEnd operand ;
```

where `mytokens.h` contains

```
#define Add      1
#define Sub      2
#define Mul      3
#define OpStart  1
#define OpEnd 3
```

This feature might not be needed because of the more powerful token class directive:

```
#tokclass Op { Add Sub Mul }
a : operand Op operand ;
```

Lexical Classes

ANTLR parsers use DFAs (Deterministic Finite Automata) created by DLG to match tokens found on the character input stream. More than one automaton (lexical class) may be defined in PCCTS. Multiple scanners are useful in two ways. First, more than one grammar can be described within the same PCCTS input file(s). Second, multiple automatons can be used to recognize tokens that seriously conflict with other regular expressions within the same lexical analyzer (e.g., comments, quoted-strings, etc...).

Actions attached to regular expressions (which are executed when that expression has been matched on the input stream) may switch from one lexical analyzer to another. Each analyzer (lex class) has a label used to enter that automaton. A predefined lexical class called `START` is in effect from the beginning of the PCCTS description until the user issues a `#lexclass` directive or the end of the description is found.

When more than one lexical class is defined, it is possible to have the same regular expression and the same token label defined in multiple automatons. Regular expressions found in more than one automaton are given different token type numbers, but token labels are unique across lexical class boundaries. For instance,

```
#lexclass A
#token LABEL "expr1"

#lexclass B
#token LABEL "expr2"
```

In this case, `LABEL` is the same token type number (`#define` in C or `enum` in C++) for both `expr1` and `expr2`. A reference to `LABEL` within a rule can be matched by two different regular expressions depending on which automaton is currently active.

Hence, the `#lexclass` directive marks the start of a new set of lexical definitions. Rules found after a `#lexclass` can only use tokens defined within that class—i.e., all tokens defined until the next `#lexclass` or the end of the PCCTS description, whichever comes first. Any regular expressions used explicity in these rules are placed into the current lexical class. Since the default automaton, `START`, is active upon parser startup, the start rule must be defined within the boundaries of the `START` automaton. Typically, a multiple-automaton grammar will begin with

```
#lexclass START
```

immediately before the rule definitions to ensure that the rules use the token definitions in the "main" automaton.

Tokens are given sequential token numbers across all lexical classes so that no conflicts arise. This also allows you to reference `ANTLRParser::token_tbl[`*`token_num`*`]` (which

is a string representing the label or regular expression defined in the grammar) regardless of which class *token_num* is defined in.

Multiple grammars, multiple lexical analyzers

Different grammars generally require separate lexical analyzers to break up the input stream into tokens. What may be a keyword in one language may be a simple variable in another. The `#lexclass` directive is used to group tokens into different lexical analyzers. For example, to separate two grammars into two lexical classes,

```
#lexclass GRAMMAR1
rules for grammar1
#lexclass GRAMMAR2
rules for grammar2
```

All tokens found beyond the `#lexclass` directive are considered to be of that class.

Single grammar, multiple lexical analyzers

For most languages, some characters are interpreted differently, depending on the syntactic context; comments and character strings are the most common examples. Consider the recognition of C style comments:

```
#lexclass C_COMMENT
#token "[\n\r]" <<skip(); newline();>>
#token "\*/" <<mode(START); skip();>>
#token "\*~[/]" <<skip();>>
#token "~[\*\n\r]+"     <<skip();>>

#lexclass START
#token "/\*"     <<mode(C_COMMENT); skip();>>
```

Lexical Actions

It is sometimes convenient or necessary to have a section of user C code constructed automatically by DLG placed in the lexical analyzer; for example, you may need to provide `extern` definitions for variables or functions defined in the parser, but used in token actions. Normally, actions not associated with a `#token` directive or embedded within a rule are placed in the parser generated by ANTLR. However, preceding an action appearing outside of any rule with the `#lexaction` pseudo-op directs the action to the lexical analyzer file. For example,

```
<< /* a normal action outside of the rules */ >>
#lexaction
    << /* this action is inserted into the lexical
```

```
        * analyzer created by DLG
        */
    >>
```

All `#lexaction` actions are collected and placed as a group into the C or C++ file where the "lexer" resides. Typically, this code consists of functions or variable declarations needed by `#token` actions.

Error Classes

The default syntax error reporting mechanism generates a list of tokens that could be possibly matched when the erroneous token was encountered. Often, this list is large and means little to the user for anything but small grammars. For example, an expression recognizer might generate the following error message for an invalid expression, `"a b"`:

```
syntax error at "b" missing { "\+" "\-" "\*" "/" ";" }
```

A better error message would be

```
syntax error at "b" missing { operator ";" }
```

This modification can be accomplished by defining the error class:

```
#errclass "operator" { "\+" "\-" "\*" "/" }
```

The general syntax for the `#errclass` directive is as follows:

```
#errclass label { T1 ... Tn }
```

where *`label`* is either a quoted string or a label (capitalized just like token labels). Any quoted string must not conflict with any rule name, token identifier or regular expression. Groups of expressions are replaced with this string.

The error class elements, T_i, can be

- labeled tokens or regular expressions

  Tokens (identifiers or regular expressions) referenced within an error class must at some point in the grammar be referenced in a rule or explicitly defined with `#token`. The definition need not appear before the `#errclass` definition.
- other error classes

  See the example following "rules."

- rules

  the *FIRST* set (set of all tokens that can be recognized first upon entering a rule) for that rule is included in the error class. The `-ge` command-line option can be used to have ANTLR generate an error class for each rule of the form:

  ```
  #errclass Rule { rule }
  ```

  where the error class name is the same as the rule except that the first character is converted to uppercase.

The ability to reference other error classes error class hierarchies. For example,

```
#errclass Fruit { CHERRY APPLE }
#errclass Meat { COW PIG }
#errclass "stuff you can eat" { Fruit Meat }

yum    :  (CHERRY | APPLE) PIE
       |  (COW | PIG) FARM
       |  THE (CHERRY | APPLE) TREE
       ;
```

Different error messages result depending upon where in rule `yum` a syntax error is detected. If the input were

```
THE PIG TREE
```

the following error message would result:

```
syntax error at “PIG” missing { Fruit }
```

However, if the input were

```
FARM COW
```

the decent error message

```
syntax error at “FARM” missing { “stuff you can eat” THE }
```

would result. Note that without the error class definitions, the error message would have been:

```
syntax error at “FARM” missing { CHERRY APPLE COW PIG THE }
```

which conveys the same information, but at a much more detailed level.

How ANTLR Uses Error Classes

ANTLR attempts to construct sets of tokens for error reporting—error sets. The sets are created wherever a parsing decision will be made in the generated parser. At every point in the parsing process, there is a set of currently recognizable or acceptable token. This set can be decoded and printed out when a syntax error is detected. ANTLR attempts to replace subsets of all error sets with error classes defined by the user. For example, rule `a` below contains a subrule with more than one alternative implying that a parsing decision will be required at run-time to determine which alternative to choose.

```
a : (Happy | Sad | Funny | Carefree) Person ;
```

If, upon entering rule `a`, the current token is not one of the four tokens found in the alternatives, a syntax error will have occurred and the following message would be generated (if "`huh`" were the input token):

```
syntax error at “huh” missing { Happy Sad Funny Carefree }
```

Let us define an error class called `Adjective` that groups those same four tokens together.

```
#errclass Adjective { Happy Sad Funny Carefree }
```

Now the error message would be:

```
syntax error at “huh” missing { Adjective }
```

ANTLR repeatedly trys to replace subsets of the error set until no more substitutions can be made. At each replacement iteration, the largest error class that is completely contained within the error set is substituted for that group of tokens. One replacement iteration may perform some substitution that makes another, previously inviable, substitution possible. This allows the hierarchy mechanism described above in the error class description section. The sequence of substitutions for the `yum` example in the previous section would be:

1. `{ CHERRY APPLE COW PIG THE }`
2. `{ Fruit COW PIG THE }`
3. `{ Fruit Meat THE }`
4. `{ "stuff you can eat" THE }`

The error class mechanism leads to smaller error sets and can be used to provide more informative error messages.

Actions

Actions are embedded within your grammar to effect a translation. Without actions, ANTLR grammars result in a simple recognizer, which answers yes or no as to whether an input sentence was valid. This section describes where actions may occur within an ANTLR grammar, when they are executed, and what special terms they may reference (e.g., for attributes). Actions are of the form "<<...>>" (normal action) or "[...]" (argument or return value block).

Placement

There are three main positions where actions may occur:

- **Outside of any rule**. These actions may not contain executable code unless it occurs within a completely-specified function. Typically, these actions contain variable and function declarations as would normally be found in a C or C++ program. These actions are placed in the global scope in the resulting parser. Consequently, all other actions have access to the declarations given in these global actions. For example,

```
<<
extern int from_elsewhere;
enum T { X, Y, Z };
main()
{
      ...
}
>>
a: <<T b=X; printf("starting a");>>
      blah
   ;
```

- **Within a rule or immediately following the rule**. These actions are executed during the recognition of the input and must be executable code unless they are init-actions, in which case, they may contain variable declarations as well. Actions immediately following the ';' of a rule definition are fail-actions and are used to clean up after a syntax error. (These are less useful now due to parser exception handlers.) For example,

```
rule   :  <<init-action>>
             ... <<normal action>> ...
       ;
       <<fail-action>>
```

- **As a rule argument or return value block**. These actions either define arguments and return values or they specify the value of arguments and return values; their behavior is identical to that of normal C/C++ functions except that ANTLR allows you to define more than one return value. For example,

```
code_block[Scope s] > [SymbolList localsyms]
       :   <<Symbol *sym;>>
           "begin" decl[$s] > [sym] <<$localsyms.add(sym);>> "end"
       ;
```

  where `s` is an input argument to `code_block`, `localsyms` is a return value, and `sym` is a local variable in `code_block` that holds the result of calling rule `decl`.

Time of Execution

Actions placed among the elements in the productions of a rule are executed immediately following the recognition of the preceding grammar element, whether that element is a simple token reference or large subrule.

Init-actions are executed before anything has been recognized in a subrule or rule. Init-actions of subrules are executed regardless of whether or not anything is matched by the subrule. Further, init-actions are always executed during guess mode; i.e., while evaluating a syntactic predicate.

Fail-actions are used only when parser exception handlers are not used and are executed upon a syntax error within that rule.

Interpretation of Action Text

ANTLR generally ignores what you place inside actions with the exception that certain expression terms are available to allow easy access to attributes (C interface), token pointers (C++ interface), and trees. The following tables describe the various special symbols recognized by ANTLR inside `[...]` and `<<...>>` actions for the C and C++ interface.

Comments (both C and C++), characters, and strings are ignored by ANTLR. To escape '`$`' and '`#`', use '`\$`' and '`\#`'.

TABLE 8. C++ Interface Interpretation of Terms in Actions

Symbol	Meaning
`$`*`j`*	The token pointer for the *j*th element (which must be a token reference) of the current alternative. The counting includes actions. Subrules embedded within the alternative are counted as one element. There is no token pointer associated with subrules, actions, or rule references.
`$`*`i.j`*	The token pointer for the *j*th element of *i*th level starting from the outermost (rule) level at 1. .
`$0`	Invalid. No translation. There is no token pointer associated with rules.
`$$`	Invalid. No translation.
`$`*`arg`*	The rule argument labeled *`arg`*.
`$`*`rule`*	Invalid. No translation.
`$`*`rv`*	The rule return result labeled *`rv`*. (l-value)
`$[`*`token_type`*`,`*`text`*`]`	Invalid. There are no attributes using the C++ interface.
`$[]`	Invalid.

Table 9 on page 107 provides a brief description of the available AST expressions. See Table 10 on page 126 for a more complete description

TABLE 9. Synopsis of C/C++ Interface Interpretation of AST Terms in Actions

Symbol	Meaning
`#0`	A pointer to the result tree of the enclosing rule. (l-value).
`#i`	A pointer to the AST built (or returned from) the *i*th element of the enclosing alternative.
`#label`	A pointer to the AST built (or returned from) the elemented labeled with *label*. Translated to *label*`_ast`.
`#[args]`	Tree node constructor. Translated to a call to `zzmk_ast(zzastnew(), args)` in C. In C++, it is translated to "`new AST(args)`".
`#[]`	Empty tree node constructor.
`#(root, child1, ..., childn)`	Tree constructor.
`#()`	`NULL`.

Init-Actions

Init-actions are used to define local variables and optionally to execute some initialization code for a rule or subrule. The init-action of a rule is executed exactly once—before any in the rule has been executed. It is not executed unless the rule is actually invoked by another rule or a user action (such as main routine). For example,

```
a  :  <<int i;>>
a  :  INT    <<i = atoi(a->getText());>>
   |  ID     <<i = 0;>>
   ;
```

The init-action of a subrule is always executed regardless of whether the subrule matches any input. For example,

```
a  :  ( <<int i=3;>> ID )*
      /* i is local to the (...)* loop and initialized only once */
      { <<f = 0;>> b:FLOAT <<f=atof(b->getText());>> }
      /* f is 0 if a FLOAT was not found */
   ;
```

Init-actions can not reference attribute or token pointer symbols such as $*label*.

Fail Actions

Fail actions are actions that are placed immediately following the ";" rule terminator. They are executed after a syntax error has been detected but before a message is printed and the attributes have been destroyed (optionally with zzd_attr()). However, attributes are not valid here because we do not know at what point the error occurred and which attributes even exist. Fail actions are often useful for cleaning up data structures or freeing memory. For example,

```
a   :   <<List *p=NULL;>>
        ( Var <<append(p, $1);>> )+
        <<OperateOn(p); rmlist(p);>>
    ;
        <<rmlist(p);>>
```

The ( )+ loop matches a lists of variables (Vars) and collections them in a list. The fail-action <<rmlist(p);>> specifies that if and when a syntax error occurs, the elements are to be freed.

Fail-actions should not reference attribute or token pointer symbols such as $*label*.

Fail-actions are executed right before the rule returns to the invoking rule.

Accessing Token Objects From Grammar Actions

The C++ interface parsing-model specifies that the parser accepts a stream of token pointers rather than a stream of simple token types, such as is done using the C interface parsing-model. Rather than accessing attributes computed from the text and token type of the input token, the C++ interface allows direct access to the stream of token objects created by the scanner. You may reference $*label* within the actions of a rule where *label* is a label attached to a token element defined within the same alternative. For example,

```
def :   "var" id:ID ";" <<behavior->defineVar($id->getText());>>
```

In this case, $id is a pointer to the token object created by the scanner (with the makeToken() function) for the token immediately following the keyword var on the input stream. Normally, you will subclass ANTLRRefCountToken or simply use ANTLRCommonToken as the token object class. Functions getText() and getLine() can be used to access the attributes of the token object.

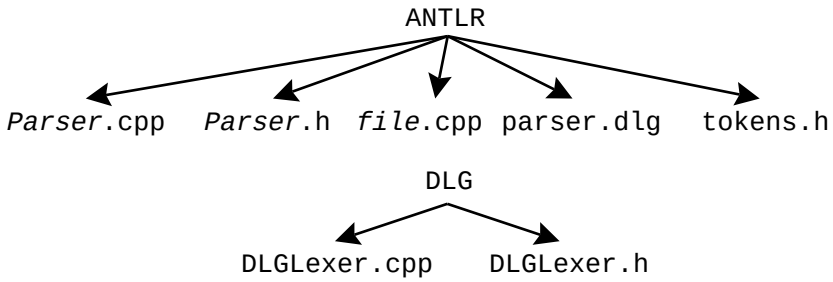
C++ Interface

When generating recursive-descent parsers in C++, ANTLR uses the flexibility of C++ classes in two ways to create modular, reusable code. First, ANTLR will generate parser classes in which the class member functions, rather than global functions, contain the code

- to recognize rules and
- to perform semantic actions

Second, ANTLR uses snap-together classes for the input, the lexer, and the token buffer. Figure 1 on page 109 shows the files generated by ANTLR and DLG for grammar class `Parser` and grammar file `file.g`.

FIGURE 1 Files Generated By ANTLR, DLG



An ANTLR parser consists of one or more C++ classes, called parser classes. Each parser class recognizes and translates part (or all) of a language. The recursive-descent recognition routines and the semantic actions are member functions of this class. A parser object is an instantiation (or variable) of the parser class.

To specify the name of the parser class in an ANTLR grammar description, enclose the appropriate rules and actions in a C++ class definition, as follows:

```
class Expr {
   <<int i;>>
<<
public:
       void print();
>>
e  :   INT ("\*" INT)* ;
       ... // other grammar rules
}
```

ANTLR then generates a parser class `Expr` that looks like the following:

```
class Expr : public ANTLRParser {
public:
       Expr(ANTLRTokenBuffer *input);
       Expr(ANTLRTokenBuffer *input, ANTLRTokenType eof);
       int i;
       void print();
       void e();
private:
       internal-Expr-specific-data;
};
```

The Utility of C++ Classes in Parsing

It is natural to have many separate parser objects. For example, if parsing ANSI C code, we might have three parser classes: for C expressions, C declarations, and C statements. Parsing multiple languages or parts of languages simply involves switching parser objects. For example, if you had a working C language front end for a compiler, to evaluate C expressions in a debugger, just use the parser object for C expressions (and modify the semantic actions with virtual functions.)

Using parser classes has the standard advantages of C++ classes involving name spaces and encapsulation of state. Because all routines are class member functions, they belong in the class name space and do not clutter the global name space, reducing (or greatly simplifying) the problem of name clashes. A parser object encapsulates the various state needed during a parse or translation.

While the ability to cleanly instantiate and invoke multiple parsers is useful, the main advantage of parser classes is that they can be extended in an object-oriented fashion. By using the inheritance and virtual functions mechanisms of C++, a parser class can be used as the base class (superclass) for a variety of similar but non-identical uses. Derived parser classes can be specialized for different activities; in many cases, these derived classes need only redefine translation actions because they inherit the grammar rules (these recursive-descent routines are member functions) from the base class. For example,

```
class CPP_Parser {
<<
virtual void defineClass(char *cl);
>>
cdef
    :   "class" id:ID "\{" ... "\}" <<defineClass(id->getText());>>
    ;
...
}
```

To construct a browser, you might subclass `CPP_Parser` to override `defineClass()` so that the function would highlight the class name on the screen; e.g.,

```
class CPP_Browser {
    // nondefault constructor is required.
    CPP_Browser(ANTLRTokenBuffer *in) : CPP_Parser(in) { }
    void defineClass(char *cl) { highlight(cl); }
};
```

A C++ compiler might override `defineClass()` to add the symbol to the symbol table.

Alternatively, the behavior of a parser can be delegated to a behavior object such that actions in the parser would be of the form

```
<<behavior->triggerSomeAction();>>
```

This approach has the advantage that behavior of the parser can be changed at runtime.

Invoking ANTLR Parsers

The second way ANTLR uses C++ classes is to have separate C++ classes for the input stream, the lexical analyzer (scanner), the token buffer, and the parser. Conceptually, these classes fit together as shown in Figure 2 on page 111. In fact, the ANTLR-generated classes "snap together" in an identical fashion. To initialize the parser, you

1. Attach an input stream object to a DLG-based scanner; if the user has constructed their own scanner, they would attach it here.
2. Attach a scanner to a token buffer object.
3. Attach the token buffer to a parser object generated by ANTLR.

FIGURE 2 Overview of the C++ classes generated by ANTLR.

The following code illustrates how these classes fit together for a parser object `Expr`.

```
main()
{
    DLGFileInput in(stdin);     // get an input stream for DLG
    DLGLexer scan(&in);         // connect a scanner to an input stream
    ANTLRTokenBuffer pipe(&scan);  // connect scanner, parser via pipe
    // DLG needs vtbl to access virtual func, pass a token.
    // mytoken(aToken) converts aToken to a (ANTLRToken *).
    // You don't need mytoken if you don't use garbage-
    // collected token objects.
    ANTLRTokenPtr aToken = new ANTLRToken;
    scan.setToken(mytoken(aToken));
    Expr parser(&pipe);         // make a parser connected to the pipe
    parser.init();              // initialize the parser
    parser.e();                 // begin parsing; e = start symbol
}
```

where `ANTLRToken` is programmer-defined and must be a subclass of `ANTLRAbstractToken`. To start parsing, it is sufficient to call the `Expr` member function associated with the grammar rule; here, `e` is the start symbol. Naturally, this explicit sequence is a pain to type so we have provided a "black box" template:

```
main()
{
    ParserBlackBox<DLGLexer, Expr, ANTLRToken> p(stdin);
    p.parser()->e();
}
```

To ensure compatibility among different input streams, lexers, token buffers, and parsers, all objects are derived from one of the four common bases classes `DLGInputStream`, `DLGLexerBase`, `ANTLRTokenBuffer` or `ANTLRParser`. All parsers are derived from a common base class `ANTLRParser`.

ANTLR C++ Class Hierarchy

Figure 3 on page 121 shows an overview of important class relationships defined by the C++ interface. Each element of the class hierarchy includes rules, behaviors, and design tips for building hierarchies that is a benefit to a user of good hierarchies.

Token Classes

Each token object passed to the parser must satisify at least the interface defined by class `ANTLRAbstractToken` if ANTLR is to compile and report errors for you. Specifically, ANTLR token objects know their token type, line number, and associated input text.

```
class ANTLRAbstractToken {
public:
    virtual ANTLRTokenType getType();
    virtual void setType(ANTLRTokenType t);     // optional
    virtual int getLine();
    virtual void setLine(int line);             // optional
    virtual ANTLRChar *getText();
    virtual void setText(ANTLRChar *);          // optional
    virtual ANTLRAbstractToken *
           makeToken(ANTLRTokenType t, ANTLRChar *txt, int ln);
};
```

Most of the time you will want your token objects to be garbage collected to avoid memory leaks. The `ANTLRRefCountToken` class is provided for this purpose. All subclasses are garbage collected (assuming you use the provide "smart pointer" class `ANTLRTokenPtr`).

The common case is that you will subclass the `ANTLRRefCountToken` interface. For your convenience, however, a token object class, `ANTLRCommonToken`, that will work "out of the box." It does garbage collection and has a fixed text field that stores the text of token found in the input stream.

Why function `makeToken()` is required at all and why you have to pass the address of an `ANTLRToken` into the DLG-based scanner during parser initialization may not be obvious. Why cannot the constructor be used to create a token and so on? The reason lies with the scanner, which must construct the token objects. The DLG support routines are typically in a precompiled object file that is linked, regardless of your token definition. Hence, DLG must be able to create tokens of any type.

Because objects in C++ are not "self-conscious" (i.e., they do not know their own type), DLG has no idea of the appropriate constructor. Constructors cannot be virtual anyway; so, we provide a constructor that is virtual and that acts like a factory. It returns the address of a new token object upon each invocation rather than just initializing an existing object.

Because classes are not first-class objects in C++ (i.e., you cannot pass class names around), we must pass DLG the address of an `ANTLRToken` token object so DLG has access to the appropriate virtual table and is, thus, able to call the appropriate `makeToken()`. This weirdness would disappear if all objects knew their type or if class names were first-class objects. Here is the code fragment in DLG that constructs the token objects that are passed to the parser via the `ANTLRTokenBuffer`:

```
ANTLRAbstractToken *DLGLexerBase::
getToken()
{
    if ( token_to_fill==NULL ) panic("NULL token_to_fill");
    ANTLRTokenType tt = nextTokenType();
    DLGBasedToken *tk = (DLGBasedToken *)
        token_to_fill->makeToken(tt, _lextext, _line);
```

```
    return tk;
}
```

Token Object Garbage Collection

Token objects are created via `ANTLRToken::makeToken()`, but how are they deleted? The class `ANTLRCommonToken` is garbage collected through a "smart pointer" called `ANTLRTokenPtr` using reference counting. Any token object not referenced by your grammar's actions is destroyed by the `ANTLRTokenBuffer` when it makes room for more token objects. (Calling function `ANTLRParser::noGarbageCollection()` will turn of this mechanism.) Token objects referenced by your actions are destroyed when local `ANTLRTokenPtr` objects are deleted. For example,

```
a : label:ID ;
```

would be converted to something like:

```
void yourclass::a(void)
{
    zzRULE;
    ANTLRTokenPtr label=NULL;
    zzmatch(ID);
    label = (ANTLRTokenPtr)LT(1);
    consume();
    ...
}
```

When the `label` object is destroyed (it is just a pointer to your input token object obtained from `LT(1)`), it decrements the reference count on the object created for the `ID`. If the count goes to zero, the object pointed by `label` is deleted.

To correctly manage the garbage collection, use `ANTLRTokenPtr` instead of "`ANTLRToken *`." Unforunately, the smart pointers can only be pointers to the abstract token class, which causes trouble in your actions. If you subclass ANTLRCommonToken and then attempt to refer to one of your token members via a token pointer in your grammar actions, the C++ compiler will complain that your token object does not have that member. For example, the following results in a compile-time error:

```
<<
class ANTLRToken : public ANTLRCommonToken {
        int muck;
        ...
};
>>

class Foo {
a : t:ID << t->muck = ...; >> ;
}
```

The `t->muck` reference will convert the `t` to "`ANTLRAbstractToken *`" resulting from `ANTLRTokenPtr::operator->()`. Instead, you must do the following:

```
a : t:ID << mytoken(t)->muck = ...; >> ;
```

in order to downcast `t` to be an "`ANTLRToken *`". Macro `mytoken(`*`aSmartTokenPtr`*`)` gets an "`ANTLRToken *`" from a smart pointer.

The reference counting interface used by `ANTLRTokenPtr` is as follows:

```
class ANTLRRefCountToken : public ANTLRAbstractToken {

/*  define to satisfy ANTLRTokenBuffer's need to determine
    whether or not a token object can be destroyed.  If
    nref()==0, no one has a reference, and the object may be
    destroyed.  This function defaults to 1, hence, if you use
    ANTLRParser::garbageCollectTokens() message with a token
    object not derived from ANTLRCommonRefCountToken, the parser
    will compile but will not delete objects after they leave the
    token buffer. */

protected:
    unsigned refcnt_;
public:
    // these 3 functions are called by ANTLRTokenPtr class
    virtual unsigned nref() { return 1; }
    virtual void ref();
    virtual void deref();
};
```

Scanners and Token Streams

The raw stream of tokens coming from a scanner is accessed via an `ANTLRTokenStream`. The required interface is simply that the token stream must be able to answer the message `getToken()`:

```
class ANTLRTokenStream {
public:
    virtual ANTLRAbstractToken *getToken() = 0;
};
```

To use your own scanner, subclass `ANTLRTokenStream` and define `getToken()` or have `getToken()` call the appropriate function in your scanner. For example,

```
class MyLexer : public ANTLRTokenStream {
private:
    int c;
public:
    MyLexer();
```

```
    virtual ANTLRAbstractToken *getToken();
};
```

DLG scanners are all subclasses of `ANTLRTokenStream`.

Token Buffer

The parser is "attached" to an `ANTLRTokenBuffer` by interface functions: `getToken()` and `bufferedToken()`. The object that actually consumes characters and constructs tokens, a subclass of `ANTLRTokenStream`, is connected to the `ANTLRTokenBuffer` via interface function `ANTLRTokenStream::getToken()`. This strategy isolates the infinite lookahead mechanism (used for syntactic predicates) from the parser and provides a "sliding window" into the token stream.

The `ANTLRTokenBuffer` begins with *k* token object pointers where *k* is the size of the lookahead specified on the ANTLR command line. The buffer is circular when the parser is not evaluating a syntactic predicate (that is, when ANTLR is guessing during the parse); when a new token is consumed, the least recently read token pointer is discarded. When the end of the token buffer is reached during a syntactic predicate evaluation, however, the buffer grows so that the token stream can be rewound to the point at which the predicate was initiated. The buffer can only grow, never shrink.

By default, the token buffer deletes token objects when they are no longer needed. A reference count is used to determine how many references exist to each token object. When the count reaches zero, the token object is subject to deletion. If your grammar references a token object in a grammar action, the token buffer will not delete that object. The "smart pointer" to the token object used by your action will delete it.

The token object pointers in the token buffer may be accessed from your actions with `ANTLRParser::LT(`*i*`)`, where *i*=1..*n* where *n* is the number of token objects remaining in the file; `LT(1)` is a pointer to the next token to be recognized. This function can be used to write sophisticated semantic predicates that look deep into the rest of the input token stream to make complicated decisions. For example, the C++ qualified item construct is difficult to match because there may be an arbitrarily large sequence of scopes before the object can be identified (e.g., `A::B::~B()`).

The `ANTLRParser::LA(`*i*`)` function returns the token type of the i^{th} lookahead symbol, but is valid only for *i*=1..*k*. This function uses a cache of *k* tokens stored in the parser itself. The token buffer itself is not queried.

The commonly used ANTLRTokenBuffer functions are:

`virtual ANTLRAbstractToken *`**`getToken()`**`;`
: Return the next token from the buffer.

`virtual ANTLRAbstractToken *`**`bufferedToken`**`(int i);`
: Return the token `i` ahead where `i` = 1..*n* with *n* equal to the number of tokens remaining in the input.

`void` **`noGarbageCollectTokens`**`();`
: Turn off deletion of token objects by buffer.

`void` **`garbageCollectTokens`**`();`
: Turn on deletion of token objects by buffer; this is the default.

`virtual void` **`setMinTokens`**`(int k_new);`
: Specify the minimum number of token objects held by the buffer. The `k_new` element must as large as the k specified to the `ANTLRTokenBuffer` constructor.

Parsers

ANTLR generates a subclass of `ANTLRParser` called `P` for definitions in your grammar file of the form:

```
class P {
...
}
```

The commonly used functions that you may wish to invoke or override are:

```
class ANTLRParser {
public:
```

`virtual void` **`init()`**`;`
: Note: you must call `ANTLRParser::init()`if you override `init()`.

`ANTLRTokenType` **`LA`**`(int i);`
: The token type of the `i`th symbol of lookahead where `i`=1..k.

`ANTLR AbstractToken *`**`LT`**`(int i);`
: The token object pointer of the `i`th symbol of lookahead where `i`=1..*n* (*n* is the number of tokens remaining in the input).

`void` **`setEofToken`**`(ANTLRTokenType t);`
: When using non-DLG-based scanners, you must inform the parser what token type should be considered end-of-input. This token type is then used by the errorecovery facilities to scan past bogus tokens without going beyond the end of the input.

```
    void garbageCollectTokens();
```
Any token pointer discarded from the token buffer is `deleted` if this function is called (assuming the reference count is zero for that token.) This is the default.

```
    void noGarbageCollectTokens();
```
The token buffer does not delete any tokens.

```
    virtual void syn (ANTLRAbstractToken *tok,ANTLChar*egroup, SetWordType
    *eset, ANTLRTokenType etok, int k);
```
You can redefine `syn()` to change how ANTLR resports error messages; see `edecode()` below.

```
    virtual void panic(char *msg);
```
Call this if something really bad happens. The parser will terminate.

```
    virtual void consume();
```
Get another token of input.

```
    void consumeUntil(SetWordType *st); // for exceptions
```
This function forces the parser to consume tokens until a token in the token class specified (or end-of-input) is found. That token is not consumed. You may want to call `consume()` afterwards.

```
    void consumeUntilToken(int t);
```
Consume tokens until the specified token is found(or end of input). That token is not consumed—you may want to `consume()` afterwards.

```
protected:
```

```
    void edecode(SetWordType *);
```
Print out in set notation the specified token class. Given a token class called *T* in your grammar, the set name will be called *T*`_set` in an action.

```
    virtual void tracein(char *r);
```
This function is called upon exit from rule r.

```
    virtual void traceout(char *r);
```
This function is called upon exit from rule r.

```
};
```

AST Classes

ANTLR's AST definitions are subclasses of `ASTBase`, which is derived from `PCCT_AST` (so that the SORCERER and ANTLR trees have a common base). The interesting functions are as follows:

class PCCTS_AST {

```
// minimal SORCERER interface
    virtual PCCTS_AST *right();
```
Return next sibling.

```
    virtual PCCTS_AST *down();
```
Return first child.

```
    virtual void setRight(PCCTS_AST *t);
```
Set the next sibling.

```
    virtual void setDown(PCCTS_AST *t);
```
Set the first child.

```
    virtual int type();
```
What is the node type (used by SORCERER).

```
    virtual void setType(int t);
```
Set the node type (used by SORCERER)?

```
    virtual PCCTS_AST *shallowCopy();
```
Return a copy of the node (used for SORCERER in transform mode). When you implement this, you must NULL the child-sibling pointers. You can define a copy constructor and have `shallowCopy()` call that. If you you want to use `dup()` with either ANTLR or SORCERER or `-transform` mode with SORCERER, you must define `shallowCopy()`.

```
    // not needed by ANTLR—support functions; see SORCERER doc
    virtual PCCTS_AST *deepCopy();
    virtual void addChild(PCCTS_AST *t);
    virtual void insert_after(PCCTS_AST *a, PCCTS_AST *b);
    virtual void append(PCCTS_AST *a, PCCTS_AST *b);
    virtual PCCTS_AST *tail(PCCTS_AST *a);
    virtual PCCTS_AST *bottom(PCCTS_AST *a);
    virtual PCCTS_AST *cut_between(PCCTS_AST *a, PCCTS_AST *b);
    virtual void tfree(PCCTS_AST *t);
    virtual int nsiblings(PCCTS_AST *t);
    virtual PCCTS_AST*sibling_index(PCCTS_AST *t, int i);

    virtual void panic(char *err);
       Print an error message and terminate the program.
};
```

ASTBase is a subclass of PCCTS_AST and adds the functionality:

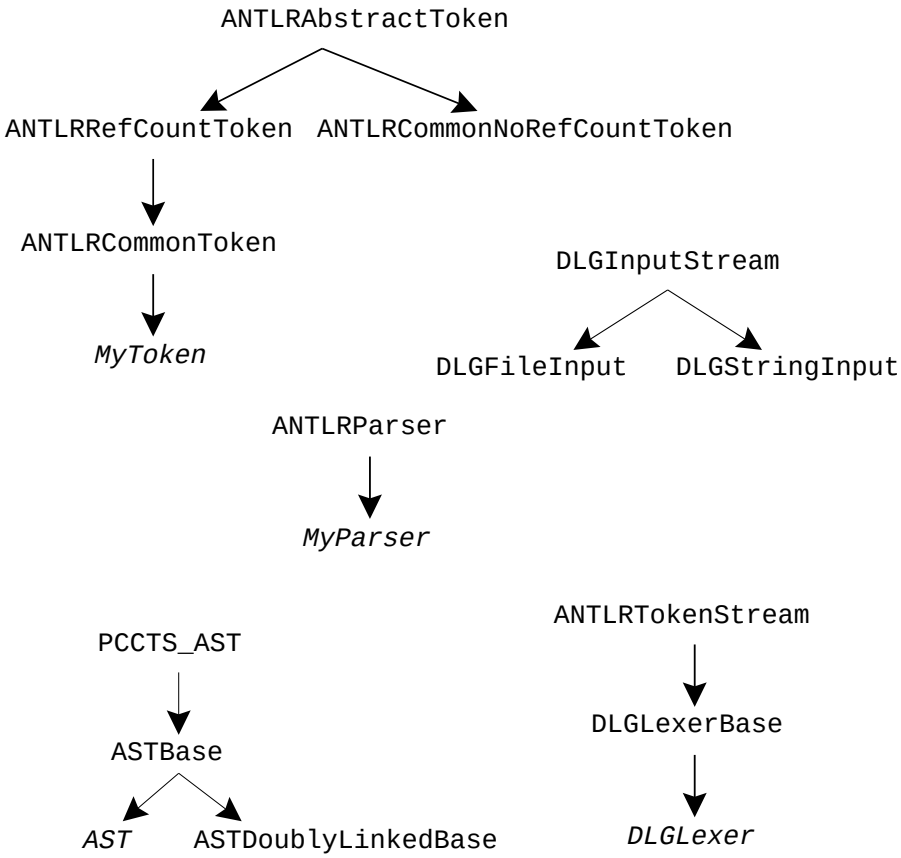
```
class ASTBase : public PCCTS_AST {
public:
    ASTBase *dup();
            Return a duplicate of the tree.

    void destroy();
            Delete the entire tree.

    static ASTBase *tmake(ASTBase *, ...);
            Construct a tree from a possibly NULL root (first argument) and a list of
            children. Followed by a NULL argument.

    void preorder();
            Preorder traversal of a tree (normally used to print out a tree in LISP form).

    virtual void preorder_action();
            What to do at each node during the traversal.

    virtual void preorder_before_action ();
            What to do before descending a down pointer link (i.e., before visiting the
            children list). Prints a left parenthesis by default.

    virtual void preorder_after_action();
            What to do upon return from visiting a children list. Prints a right parenthesis
            by default.

};
```

To use doubly linked child-sibling trees, subclass ASTDoublyLinkedBase instead:

```
class ASTDoublyLinkedBase : public ASTBase {
public:
    void double_link(ASTBase *left,ASTBase *up);
            Set the parent (up) and previous child (left) pointers of the whole tree.
            Initially, left and up arguments to this function must be NULL.

    PCCTS_AST *left() { return _left; }
            Return the previous child.

    PCCTS_AST *up() { return _up; }
            Return the parent (works for any sibling in a sibling list).

};
```

Note, however, that the tree routines from ASTBase do not update the left and up pointers. You must call double_link() to update all the links in the tree.

FIGURE 3 C++ Class Hierarchies



Intermediate-Form Tree Construction

You may embed actions within an ANTLR grammar to construct abstract syntax trees (ASTs) but ANTLR provides an automatic mechanism for implicitly or explicitly specifying tree structures. Using the automatic mechanism, you must define what an AST node looks like and how to construct an AST node given an attribute (C) or token pointer (C++). The ANTLR `-gt` command line option must be turned on so that ANTLR knows to generate the extra code in the resulting parser to construct and manipulate trees. In this section, we describe the required C or C++ definitions, the available support functions, the AST operators, and the special symbols used in actions to construct nodes and trees.

Required AST Definitions

The C++ interface requires that you derive a class called AST from ASTBase. The derived class contains the fields you need for your purposes and a constructor that accepts an ANTLRToken pointer; the constructor fills in the AST node from the contents of the token object. Here is a sample AST node definition that merely makes a reference to the token object for which the node was created:

```
class AST : public ASTBase {
public:
    ANTLRTokenPtr token;
    AST(ANTLRTokenPtr t) { token = t; }
};
```

The calling of grammar rules from C++ code is slightly different when trees are being built. As with the C interface, the address of a NULL-initialized tree pointer must be passed to the starting rule. The pointer comes back set to the tree constructed for that rule:

```
main()
{
    ParserBlackBox<...> p(stdin);

    ASTBase *root = NULL;
    p.parser->start_rule(&root);
}
```

AST Support Functions

The following PCCTS_AST members are not needed by ANTLR or SORCERER, but are support functions available to both; they are useful in SORCERER applications

```
void addChild(PCCTS_AST *t);
```

Add a child t of this.

```
PCCTS_AST *ast_find_all(PCCTS_AST *t,
                        PCCTS_AST *u,
                        PCCTS_AST **cursor);
```

Find all occurrences of u in t. The cursor must be initialized to NULL before calling this function and is used to keep track of where in t the function left off between function calls. Returns NULL when no more occurrences are found. Useful for iterating over every occurrence of a particular subtree.

```
int match(PCCTS_AST *t,PCCTS_AST *u);
```

Returns true if t and u have the same structure (the trees have the same tree structure and token type fields.) If both trees are NULL, true is returned.

`void` **`insert_after`**`(PCCTS_AST *a,PCCTS_AST *b);`
Add `b` immediately after `a` as its sibling. If `b` is a sibling list at its top level, then the last sibling of `b` points to the previous right sibling of `a`. Inserting a `NULL` pointer has no effect.

`void` **`append`**`(PCCTS_AST *a, PCCTS_AST *b);`
Add `b` to the end of `the` sibling list for `a`. Appending a `NULL` pointer is illegal.

`PCCTS_AST *`**`tail`**`(PCCTS_AST *a);`
Find the end of the sibling list for `a` and return a pointer to it.

`PCCTS_AST *`**`bottom`**`(PCCTS_AST *a);`
Find the bottom of `the` child list for `a` (going straight "down".)

`PCCTS_AST *`**`cut_between`**`(PCCTS_AST *a, PCCTS_AST *b);`
Unlink all siblings between `a` and `b` and return a pointer to the first element of the sibling list that was unlinked. This routine makes `a` point to `b` and makes sure that the tail of the sibling list, which was unlinked, does not point to `b`. The routine ensures that `a` and `b` are (perhaps indirectly) connected.

`void` **`tfree`**`(PCCTS_AST *t);`
Recursively walk `t`, deleting all the nodes in a depth-first order.

`int` **`nsiblings`**`(PCCTS_AST *t);`
Returns the number of siblings of t.

`PCCTS_AST *`**`sibling_index`**`(PCCTS_AST *t, int i);`
Return a pointer to the i^{th} sibling where the sibling to the right of `t` is the first. An index of 0, returns t.

The following `ASTBase` members are specific to ANTLR:

`static ASTBase *`**`tmake`**`(ASTBase *, ...);`
See the `#(...)` in “Interpretation of C/C++ Actions Related to ASTs” on page 125.

`ASTBase *`**`dup`**`();`
Duplicate the current tree.

`void` **`preorder`**`();`
Perform a preorder walk of the tree using the following member functions.

`void` **`preorder_action`**`();`
What to do in each node as you do a preorder walk. Typically, `preorder()` is used to print out a tree in lisp-like notation. In that case, it is sufficient to redefine this function alone.

`void` **`preorder_before_action`**`();`
What to print out before walking down a tree.

```
void preorder_after_action();
```

What to print out after walking down a tree.

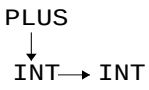
Operators

ANTLR grammars may be annotated with AST construction operators. The operators are sufficient to describe all but the strangest tree structures.

Consider the "root" operator "^." The token modified by the root operator is considered the root of the currently-built AST. As a result, the rule

```
add :   INT PLUS^ INT ;
```

results in an AST of the form:



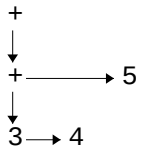
The way to "read" rule `add` with regards to AST building is to say

> "Make a node for `INT` and add it to the sibling list (it is a parent-less only child now). Make a node for `PLUS` and make it the root of the current tree (which makes it simply the parent of the only child). Make a node for the second `INT` and add it to the sibling list."

Think of the AST operators as actions that are executed as they are encountered not as something that specifies a tree structure known at ANTLR analysis time. For example, what if a looping subrule is placed in the rule?

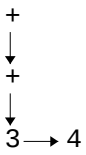
```
add :   INT (PLUS^ INT)* ;
```

Input "3+4+5" would yield:



After the "3+4" has been read, the current tree for rule add would be:

just as before. However, because the `(...)*` allows you to match another addition term, two more nodes are added to the tree for `add`, one of which is a root node. After the recognition of the second + in the input, the tree for add would look like this:

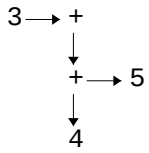


because the + was made the root of the current tree due to the root operator. The `5` is a simple leaf node (since it was not modified with either `^` or `!`) and is added to the current sibling list. The addition of the new root changes the current sibling list from the 3 and 4 list to the first + that was added to the tree; i.e., the first child of the new root. Hence, the 5 is added to the second level in the tree and we arrive at the final tree structure.

Subrules merely modify the tree being built for the current rule, whereas each rule has its own tree. For example, if the `(...)*` in add were moved to a different rule:

```
add :   INT addTerms
    ;
addTerms
    :   (PLUS^ INT)*
    ;
```

then the following, very different, tree would be generated for input "`3+4+5`:"



While this tree structure is not totally useless, it is not as useful as the previous structure because the + operators are not at the subtree roots.

Interpretation of C/C++ Actions Related to ASTs

Actions within ANTLR grammar rules may reference expression terms that are not valid C or C++ expressions, but are understood and translated by ANTLR. These terms are useful, for example, when you must construct trees too complicated for simple grammar annotation, when nodes must be added to the trees built by ANTLR, or when partial trees must be examined.

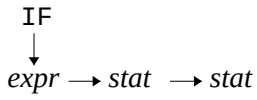
TABLE 10. C/C++ Interface Interpretation of AST Terms in Actions

Symbol	Meaning
`#0`	A pointer to the result tree of the enclosing rule (l-value).
`#i`	A pointer to the AST built (or returned from) the i^{th} element of the enclosing alternative. You should really use the `#label` instead.
`#label`	A pointer to the AST built (or returned from) the element labeled with `label`. Translated to `label_ast`.
`#[args]`	Tree node constructor. Translated to a call to `zzmk_ast(zzastnew(), args)` in C where `zzastnew()` allocates and returns a pointer to a new, initialized AST node. You must define `zzmk_ast()` if this term is used. In C++, translated to "`new AST(args)`".
`#[]`	Empty tree node constructor. Translated to a call to `zzastnew()` in C and to "`new AST`" in C++.
`#(root child1, ..., childn)`	Tree constructor. If `root` is `NULL`, then a sibling list is returned. The `childi` arguments are added to the sibling list until the first `NULL` pointer (not counting the `root` pointer) is encountered .
`#()`	`NULL`.

Consider how one might build an AST for an `if` statement. A useful tree may quickly and easily be constructed via grammar annotations:

```
if :  IF^ expr THEN! stat { ELSE! stat } ;
```

Here, the `IF` is identified as the root of the tree, the `THEN` and `ELSE` are omitted as unnecessary, and the trees constructed by `expr` and `stat` are linked in as children of the `IF` node:



To construct the same tree structure manually, the following grammar is sufficient:

```
if! :   IF e:expr THEN st:stat { ELSE el:stat }
        <<#if = #(#[IF], #e, #st, #el);>>
    ;
```

where the '!' in the rule header indicates that no automatic tree construction should be done by ANTLR. The "#[IF]" term constructs a tree node via "new AST(IF)" (assuming you have defined an AST constructor taking a ANTLRTokenType argument) and the "#(...)" tree constructor puts the IF node above the children matched for the conditional and statements. The el label is initialized to NULL and contributes nothing to the resulting tree if an else clause is not found on the input.

Predicates

Predicates are used to recognize difficult language constructs such as those that are context-sensitive or those that require unbounded lookahead to recognize. This section provides a brief description of how predicates are defined and used to alter the normal LL(k) parsing strategy.

Semantic Predicates

Semantic predicates alter the parse based upon run-time information. Generally, this information is obtained from a symbol table and is used to recognize context-sensitive constructs such as those that are syntactically identical but semantically very different; e.g., variables and type names are simple identifiers, but are used in completely different contexts.

The basic semantic predicate takes the form of an action suffixed with the "?" operator: "<<...>>?." No white space is allowed between the ">>" and the "?." Predicates must be boolean expressions and may not have side effects (i.e., they should not modify variables.) Alternatives without predicates are assumed to have a predicate of "<<1>>?." Also, because predicates can be "hoisted," out of rules as we will see shortly, predicates that refer to rule parameters or local variables are often undesirable.

Validating Semantic Predicates

All semantic predicates behave at least as *validating predicates*. That is, all predicates must evaluate to true as the parser encounters them during the parse or a semantic error occurs. For example in,

```
typename
    :   <<isTypeName(LT(1)->getText())>>? ID
    ;
```

When `typename` is invoked, the predicate is tested before attempting to match the `ID` token reference; where `isTypeName()` is some user-defined boolean function. If the first symbol of lookahead is not a valid type name, ANTLR generates an error message indicating that the predicate failed.

A fail action may be specified by appending a "`[...]`" action; this action is executed upon failure of the predicate when acting as a validating predicate:

```
typename
    :   <<isTypeName(LT(1)->getText())>>?[dialogBox(BadType)] ID
    ;
```

where we can presume that `dialogBox(BadType)` is a user-defined function that opens a dialog box to display an error message. ANTLR generates code similar to the following:

```
void Parser::typename(void)
{
    if (!(isTypeName(LT(1)->getText()))) dialogBox(BadType) ;
    zzmatch(ID);
    consume();
    return;
}
```

using the C++ interface.

Disambiguating Semantic Predicates

When ANTLR finds a syntactic ambiguity in your grammar, ANTLR attempts to resolve the ambiguity with semantic information. In other words, ANTLR searches the grammar for any predicates that provide semantic information concerning the tokens in the lookahead buffer. A predicate that is tested during the parse to make a parsing decision (as opposed to merely checking for validity once a decision has been made) is considered a *disambiguating predicate*. We say that disambiguating predicates are *hoisted* into a parsing decision from a rule or rules. A predicate that may be hoisted into a decision is said to be *visible* to that decision. In this section, we describe which predicates are visible, how multiple predicates are combined, and how visible predicates are incorporated into parsing decisions.

ANTLR searches for semantic predicates when a syntactically ambiguous parsing decision is discovered. The set of visible predicates is collected and tested in the appropriate prediction expression. We say that predicate *p* is visible to an alternative (and, hence, may be used to predict that alternative) if *p* can be evaluated correctly without consuming another input token and without executing a user action. Generally, visible predicates reside on the

left edge of productions; predicates not on the left edge usually function as validating predicates only. For example,

```
a   :   <<p1>>? ID
    |   b
    ;

b   :   <<p2>>? ID
    |   <<action>> <<p3>>? ID
    |   INT <<p4>>? ID
    |   { FLOAT } <<p5>>? ID
    ;
```

First we observe that a lookahead of `ID` predicts both alternatives of rule `a`. ANTLR searchs for predicates that potentially disambiguate the parsing decision. Here, we see that *p1* may be used to predict alternative one because it can be evaluated without being hoisted over a grammar element or user action. Alternative two of rule `a` has no predicate, but the alternative references rule `b` which has two predicates. Predicate *p2* is visible, but *p3* is not because *p3* would have to be executed before *action,* which would change the action execution order. Predicate *p4* is not visible because an `INT` token would have to be consumed before *p4* could be evaluated in the correct context. Predicate *p5* is not visible because a `FLOAT` token may have to be consumed to gain the correct context. Rule `a` could be coded something like the following:

```
a()
{
   if ( LA(1)==ID && (p1) ) {
      MATCH(ID);
      consume();
   }
   else ( LA(1)==ID && (p2) ) {
      b();
   }
}
```

Predicates may be hoisted over init-actions because init-actions are assumed to contain merely local variable allocations. For example,

```
a   :   <<init-action>>// does not affect hoisting
        <<p1>>? ID
    |   b
    ;
```

Care must be taken so that predicates do not refer to local variables or rule parameters if the predicate could be hoisted out of that rule. In this example,

```
a   :   b | ID
    ;
```

```
b[int ctx]
    :   <<ctx>>? ID
    ;
```

predicate `ctx` is hoisted into rule `a`, resulting in a C or C++ compilation error because `ctx` exists only in rule `b`.

Alternatives without predicates are assumed to be semantically valid; hence, predicates on some alternatives are redundant. For example,

```
a   :   <<flag>>? ID
    |   <<!flag>>?ID
    ;
```

The predicate on the second alternative is unnecessary because if `flag` evaluates to false, `!flag` is redundant.

A predicate used to help predict an alternative may or may not apply to all lookahead sequences predicting that alternative. We say that the lookahead must be consistent with the *context* of the predicate for it to provide useful information. Consider the following example.

```
a   :   (var | INT)
    |   ID
    ;
var :   <<isVar(LATEXT(1))>>? ID
    ;
```

Because `ID` predicts both alternatives of rule `a`, ANTLR hoists the predicate `isVar()` into the prediction expression for the first alternative. However, both `INT` and `ID` predict the first alternative—evaluating `isVar()` when the lookahead is `INT` would be incorrect as it would return false when in fact no semantic information is known about `INT`s. The first alternative of rule `a` would never be able to match an `INT`.

When hoisting a predicate, ANTLR computes and then carries along the context under which the predicate was found (with "`-prc on`" command-line option). The required depth, *k*, for the predicate context is determined by examining the actual predicate to see what lookahead depths are used; predicates that do not reference `LT(`*k*`)` or `LATEXT(`*k*`)` are assumed to have *k*=1. Normally, *k*=1 as predicates usually test only the next lookahead symbol.

TABLE 11. Sample Predicates and Their Lookahead Contexts

Predicate	Context
`a : <<p(LT(1))>>? ID ;`	`ID`
`a : <<p(LT(1))>>? b ;` `b : ID | INT ;`	`ID, INT`
`a : <<p(LT(2))>>? LPAREN ID ;`	`LPAREN ID`

The third predicate in the table provides context information for the `ID` following the `LPAREN`; hence, the context is `LPAREN` followed by `ID`. The other two examples require a lookahead depth of k=1.

Predicates normally apply to exactly one lookahead sequence. ANTLR will give you a warning for any predicate that applies to more than one sequence.

There are situations when you would not wish ANTLR to compute the lookahead context of predicates:

- When ANTLR would take too long to compute the context and
- When the predicate applies only to a subset of the full context computed by ANTLR

In these situations, a predicate context-guard is required, which allows you to specify the syntactic context under which the predicate is valid. The form of a context-guarded predicate is

`( `*`context-guard`*` )? => <<`*`semantic-predicate`*`>>?`

Where the *`context-guard`* can be any grammar fragment that specifies a set of *k*-sequences where *k* is the depth referenced in *`semantic-predicate`*. For example,

```
cast_expr
    :   ( LPAREN ID )? => <<isTypeName(LT(2))>>?
        LPAREN abstract_type RPAREN
    ;
```

This predicate dictates that when the lookahead is `LPAREN` followed by `ID`, then the `isTypeName()` predicate provides semantic information and may be evaluated. Without the guard, ANTLR assumes that the lookahead was `LPAREN` followed by all tokens that could begin an `abstract_type`.

Multiple lookahead *k*-sequences can also be specified inside the context guard:

`a : ( ID | KEYWORD )? => <<`*`predicate`*`>>? b ;`

The use of EBNF looping-constructs such as `(...)*` are not valid in context-guards.

Because there may be more than one predicate visible to an alternative, ANTLR has rules for combining multiple predicates.

- Predicates or groups of predicates taken from alternative productions are `||`'d together.
- Predicates or groups of predicates taken from the same production are `&&`'d together.

For example,

```
decl
    :   typename declarator ";"
    |   declarator ";"
    ;
declarator
    :   ID
    ;
typename
    :   classname
    |   enumname
    ;
classname
    :   <<isClass(LATEXT(1))>>? ID
    ;
enumname
    :   <<isEnum(LATEXT(1))>>? ID
    ;
```

The decision for the first alternative of rule `decl` would hoist both predicates and test them in a decision similar to the following:

```
if ( LA(1)==ID && (isClass(LATEXT(1)||isEnum(LATEXT(1)) ) { ...
```

Adding a predicate to rule `decl`:

```
decl
    :   <<flag>>? typename declarator ";"
    |   declarator ";"
    ;
```

would result in flag being `&&`'d with the result of the combination of the other two predicates:

```
if (LA(1)==ID && (flag&&(isClass(LATEXT(1)||isEnum(LATEXT(1)))) { ...
```

In reality, ANTLR inserts code to ensure that the predicates are tested only when the parser lookahead is consistent with the context associated with each predicate; here, all predicates have `ID` as their context, and the redundant tests have been removed for clarity.

Semantic Predicates Effect upon Syntactic Predicates

During the evaluation of a syntactic predicate, semantic predicates that have been hoisted into prediction expressions are still evaluated. Success or failure of these disambiguating predicates simply alters the parse and does not directly cause syntax errors.

Validation predicates (those that have not been hoisted) are also still evaluated. However, a failed validation predicate aborts the current syntactic predicate being evaluated whereas, normally, a failure causes a syntax error.

Syntactic Predicates

Just as semantic predicates indicate when a production is valid, syntactic predicates also indicate when a production is a candidate for recognition. The difference lies in the type of information used to predict alternative productions. Semantic predicates employ information about the *meaning* of the input (e.g., symbol table information), whereas syntactic predicates employ structural information like normal LL(k) parsing decisions. Syntactic predicates specify a grammatical construct that must be seen on the input stream to make a production valid. Moreover, this construct may match input streams that are arbitrarily long; normal LL(k) parsers are restricted to using the next *k* symbols of lookahead.

Syntactic Predicate Form and Meaning

Syntactic predictions have the form

$(\alpha)? \beta$

or, the shorthand form

$(\alpha)?$

which is identical to

$(\alpha)? \alpha$

where α and β are arbitrary Extended BNF (EBNF) grammar fragments that do not define new nonterminals. The meaning of the long form syntactic predicate is:

"If α is matched on the input stream, attempt to recognize β."

Note the similarity to the semantic predicate

<<α>>? β

which means

"If α evaluates to true at parser run-time, attempt to match β."

Syntactic predicates may occur only at the extreme left edge of alternatives because they are only useful during the prediction of alternatives—not during the subsequent recognition of the alternatives.

Alternative productions that are syntactically unambiguous, but non-LL(k), should be rewritten, left-factored, or modified to use syntactic predicates. Consider the following rule:

```
type   :  ID
       |  ID
       ;
```

The alternatives are syntactically ambiguous because they can both match the same input. The rule is a candidate for semantic predicates, not syntactic predicates. The following example is unambiguous. It is just not deterministic to a normal LL(k) parser.

Consider a small chunk of the vast C++ declaration syntax. Can you tell exactly what type of object `f` is after having seen the left parenthesis?

```
int f(
```

The answer is "no." Object `f` could be an integer initialized to some previously defined symbol `a`:

```
int f(a);
```

or a function prototype or definition:

```
int f(float a) {...}
```

The following is a greatly simplified grammar for these two declaration types:

```
decl:  type ID "\(" expr_list "\)" ";"
   |   type ID "\(" arg_decl_list "\)" func_def
   ;
```

Left-factoring "`type ID "\("`" would be trivial because our grammar is so small and the left-prefixes are identical:

```
decl:  type ID
       "\("
          (  expr_list "\)" ";"
          |  arg_decl_list
          )
       "\)" func_def
   ;
```

However, if a user action were required before recognition of the reference to rule `type`, left-factoring would not be possible:

```
decl
    :   <<// dummy init action; next action isn't init action>>
        <<printf("var init\n");>>
        type ID "\(" expr_list "\)" ";"
    |   <<printf("func def\n");>>
        type ID "\(" arg_decl_list "\)" func_def
    ;
```

The solution to the problem involves looking arbitrarily ahead (`type` could be arbitrarily big, in general) to determine what appears after the left parenthesis. This problem is easily solved implicitly by using a syntactic predicate:

```
decl
    :   (   <<//dummy action>>
            <<printf("var init\n");>>
            type ID "\(" expr_list "\)" ";"
        )?
    |   <<printf("func def\n");>>
        type ID "\(" arg_decl_list "\)" func_def
    ;
```

The `(...)?` indicates that it is impossible to decide from the left edge of rule `decl` with a finite amount of lookahead, which production to predict. Any grammar construct inside a `(...)?` block is attempted and, if it fails, the next alternative production that could match the input is attempted. This represents selective backtracking and is similar to allowing ANTLR parsers to guess without being "penalized" for being wrong. Note that the first action of any block is considered an init-action and hence, because it may define local variables it cannot be gated out with an `if`-statement. (Local variables would not be visible outside the `if`-statement.)

Modified Parsing Strategy

Decisions that are not augmented with syntactic predicates are parsed deterministically with finite lookahead up to depth *k* as is normal for ANTLR-generated parsers. When at least one syntactic predicate is present in a decision, rule recognition proceeds as follows:

1. Find the first *viable* production (i.e., the first production in the alternative list predicted by the current finite lookahead) according to the associated finite-lookahead prediction-expression.
2. If the first grammar element in that production is not a syntactic predicate, predict that production and go to Step 3 else attempt to match the predicting grammar fragment of the syntactic predicate.

3. If the predicting grammar fragment is matched, predict the associated production and go to Step 4 else find the next viable production and go to Step 2.
4. Proceed with the normal recognition of the production predicted in Steps 2 or 3.

For successful predicates, both the predicting grammar fragment and the remainder of the production are actually matched: hence, the short form, `(α)?`, actually matches α twice—once to predict and once to apply α normally, executing any embedded actions.

Nested Syntactic Predicate Invocation

Because syntactic predicates may reference any defined nonterminal and because of the recursive nature of grammars, it is possible for the parser to return to a point in the grammar that had already requested backtracking. This nested invocation poses no problem from a theoretical point of view, but it can cause unexpected parsing delays in practice.

Efficiency

The order of alternative productions in a decision is significant. Productions in an ANTLR grammar are always attempted in the order specified. For example, the parsing strategy outline above indicates that the following rule is most efficient when *`foo`* is less complex than *`bar`*.

```
a   :   (foo)?
    |   bar
    ;
```

because they testing the simplest possibility first is faster.

Any parsing decisions made inside a `(..)?` block are made deterministically unless they themselves are prefixed with syntactic predicates. For example,

```
a   :   ( (A)+ X | (B)+ X )?
    |   (A)* Y
    ;
```

specifies that the parser should attempt to match the nonpredicated grammar fragment

```
(   (A)+ X
|   (B)+ X
)
```

using normal the normal finite-lookahead parsing strategy. If a phrase recognizable by this grammar fragment is found on the input stream, the state of the parser is restored to what it was before the predicate invocation and the grammar fragment is parsed again. If not, if the grammar fragment failed to match the input, apply the next production in the outer block:

```
(A)* Y
```

Effect of Syntactic Predicates on Actions and Semantic Predicates

While evaluating a syntactic predicate, user actions, such as adding symbol table entries, are not executed because in general, they cannot be "undone"; this conservative approach avoids affecting the parser state in an irreversible manner. Upon successful evaluation of a syntactic predicate, actions are once again enabled—unless the parser was in the process of evaluating another syntactic predicate.

Because semantic predicates are restricted to side-effect-free expressions, they are always evaluated when encountered. However, during syntactic predicate evaluation, the semantic predicates evaluated must be functions of values computed when actions were enabled. For example, if your grammar has semantic predicates that examine the symbol table, all symbols needed to direct the parse during syntactic predicate evaluation must be entered into the table before this backtracking phase has begun.

Because init-actions are always executed, it is possible to make ANTLR into actually executing an action during the evaluation of a syntactic predicate by simply enclosing the action in a subrule:

```
(<<action>>)
```

Syntactic Predicates effect upon Grammar Analysis

ANTLR constructs normal LL(k) decisions throughout predicated parsers, only resorting to arbitrary lookahead predictors when necessary. Calculating the lookahead sets for a full LL(k) parser can be quite expensive in terms of (time and space), so by default, ANTLR uses a linear approximation to the lookahead and only uses full LL(k) analysis when required. When ANTLR encounters a syntactic predicate, it generates the instructions for selective backtracking as you would expect, but also generates an approximate decision. Although no finite lookahead decision is actually required (the arbitrary lookahead mechanism will accurately predict the production without it) the approximate portion of the decision reduces the number of times backtracking is attempted without hope of a successful match. An unexpected, but important, benefit of syntactic predicates is that they provide a convenient method for preventing ANTLR from attempting full LL(k) analysis when doing so would cause unacceptable analysis delays.

Parser Exception Handlers

Parser exception handlers provide a more sophisticated alternative to the automatic error reporting and recovery facility provided by ANTLR. The notion of throwing and catching parser error signals is similar to C++ exception handling: however, our implementation allows both the C and C++ interface to use parser exception handling. This section provides a short description of the syntax and semantics of ANTLR exceptions.

When a parsing error occurs, the parser throws an exception. The most recently encountered exception handler that `catch`es the appropriate signal is executed. The parse continues after the exception by prematurely returning from the rule that handled the exception. Generally, the rule that catches the exception is not the rule that throws the exception; e.g., a `statement` rule may be a better place to handle an error than the depths of an expression evaluator as the `statement` rule has unambiguous context information with which to generate a good error message and recover.

Exception handlers may be specified:

- **After any alternative**. These handlers apply only to signals thrown while recognizing the elements of that alternative.
- **After the ';' of a rule definition**. These handlers apply to any signal thrown while recognizing any alternative of the rule unless the handler references a element label, in which case the handler applies only to recognition of that rule element. Non-labeled handlers attached to a rule catch signals not caught by handlers attached to an alternative.
- **Before the list of rules**. These *global* exception handlers apply when a signal is not caught by a handler attached to a rule or alternative. Global handlers behave slightly differently in that they are always executed in the rule that throws the signal; the rule is still prematurely exited.

Exception Handler Syntax

The syntax for an exception group is as follows:

```
exception_group
    :   "exception" { "\[" ID "\]" } ( exception_handler )*
        { "default" ":" ACTION }
    ;

exception_handler
    :   "catch" SIGNAL ":" { ACTION }
    ;
```

where `SIGNAL` is one of:

TABLE 12. Predefined Parser Exception Signals

Signal Name	Description
`NoViableAlt`	Exception thrown when none of the alternatives in a pending rule or subrule were predicted by the current lookahead.
`NoSemViableAlt`	Exception thrown when no alternatives were predicted in a rule or subrule and at least one semantic predicate (for a syntactically viable alternative) failed.
`MismatchedToken`	Exception thrown when the pending token to match did not match the first symbol of lookahead.

A "`default :`" clause may also be used in your exception group to match any signal that was thrown. Currently, you cannot define your own exception signals.

You can define multiple signals for a single handler. For example,

```
exception
    catch MismatchedToken :
    catch NoViableAlt :
    catch NoSemViableAlt :
        <<
        printf("stat:caught predefined signal\n");
        consumeUntil(DIE_set);
        >>
```

If a label attached to a rule reference is specified for an exception group, that group may be specified after the end of the '`;`' rule terminator. Because element labels are unique for each rule, ANTLR can still uniquely identify the appropriate rule reference to associate the exception group. It often makes a rule cleaner to have most of the exception handlers at the end of the rule. For example,

```
a   :   A t:expr B
    |   ...
    ;
    exception[t]
        catch ...
        catch ...
```

The `NoViableAlt` signal only makes sense for labeled exception groups and for rule exception groups.

Exception Handler Order of Execution

Given a signal, *S*, the handler that is invoked is determined by looking through the list of enabled handlers in a specific order. Loosely speaking, we say that a handler is *enabled* (becomes active) and pushed onto an exception stack when it has been seen by the parser on its way down the parse tree. A handler is *disabled* and taken off the exception stack when the associated grammar fragment is successfully parsed. The formal rules for enabling are:

- All global handlers are enabled upon initial parser entry.
- Exception handlers specified after an alternative become enabled when that alternative is predicted.
- Exception handlers specified for a rule become enabled when the rule is invoked.
- Exception handlers attached with a label to a particular rule reference within an alternative are enabled just before the invocation of that rule reference.

Disabling rules are:

- All global handlers are disabled upon parser exit.
- Exception handlers specified after an alternative are disabled when that alternative has been (successfully) parsed completely.
- Exception handlers specified for a rule become disabled just before the rule returns.
- Exception handlers tied to a particular rule reference within an alternative are disabled just after the return from that rule reference.

Upon an error condition, the parser with throw an exception signal, *S*. Starting at the top of the stack, each exception group is examined looking for a handler for *S*. The first *S* handler found on the stack is executed. In practice, the run time stack and hardware program counter are used to search for the appropriated handler. This amounts to the following:

1. If there is an exception specified for the enclosing alternative, then look for *S* in that group first.
2. If there is no exception for that alternative or that group did not specify an *S* handler, then look for *S* in the enclosing rule's exception group.
3. Global handlers are like macros that are inserted into the rule exception group for each rule.
4. If there is no rule exception or that group did not specify an *S* handler, then return from the enclosing rule with the current error signal still set to *S*.
5. If there is an exception group attached (via label) to the rule that just returned, check that exception group for *S*.
6. If an exception group attached to a rule reference does not have an *S* handler, then look for *S* in the enclosing rule's exception group.

This process continues until an *S* handler is found or a return instruction is executed in starting rule. When either happens, the start symbol would have a return-parameter set to *S*.

These guidelines are best shown in an example:

```
a   :  A c B
       exception                                          /* 1 */
          catch MismatchedToken : <<ACTION1>>
   |   C t:d D
       exception                                          /* 2 */
             catch MismatchedToken :<<ACTION2>>
             catch NoViableAlt : <<ACTION3>>
   ;
   exception[t]                                           /* 3 */
       catch NoViableAlt : <<ACTION4>>
   exception                                              /* 4 */
       catch NoViableAlt : <<ACTION5>>

c  :  E ;
d  :  e ;
e  :  F | G
   ;
   exception                                        /* 5 */
       catch MismatchedToken : <<ACTION6>>
```

Table 13 on page 141 summarizes the sequence in which the exception groups are tested.

TABLE 13. Sample Order of Search for Exception Handlers

Input	Exception group search sequence	Action Executed
D E B	4	5
A E D	1	1
A F B	1	1
C F B	2	2
C E D	5, 2	3

Note that action 4 is never executed because rule `d` has no tokens to mismatch and mismatched tokens in rule `e` are caught in that rule.

The global handlers are like macro insertions. For example:

```
exception catch NoViableAlt : <<blah blah>>
a   :  A
    ;
    exception
       catch MismatchedToken : <<ack;>>
b   :  B
    ;
```

This grammar fragment is functionally equivalent to:

```
a   :  A
    ;
    exception
       catch MismatchedToken : <<ack;>>
       catch NoViableAlt : <<blah blah>>
b   :  B
    ;
    exception
       catch NoViableAlt : <<blah blah>>
```

Modifications to Code Generation

The following items describe the changes to the output parser C or C++ code when at least one exception handler is specified:

- Each rule reference acquires a signal parameter that returns `0` if no error occurred during that reference or it returns a nonzero signal *S*.
- The `MATCH()` macro throws `MismatchedToken` rather than calling `zzsyn()`, the standard error reporting and recovery function.
- When no viable alternative is found, `NoViableAlt` is signaled rather than calling the `zzsyn()` routine.
- The parser no longer resynchronizes automatically.

Semantic Predicates and NoSemViableAlt

When the input stream does not predict any of the alternatives in the current list of possible alternatives, `NoViableAlt` is thrown. However, what happens when semantic predicates are specified in that alternative list? There are cases where it would be very misleading to just throw `NoViableAlt` when in fact one or more alternatives were syntactically viable; i.e., the reason that no alternative was predicted was due to a semantic invalidity and a different signal must be thrown in such a case. For example,

```
expr   :  <<P1>>? ID ...   /* function call */
       |  <<P2>>? ID ...   /* array reference */
       |  INT
       ;
       exception
          catch NoViableAlt :
             <<no ID or INT was found>>
          catch NoSemViableAlt :
          <<an ID was found, but it was not valid>>
```

Typically, you would want to give very different error messages for the two different situations. Specifically, reporting a message such as

```
syntax error at ID missing { ID INT }
```

would be very misleading (i.e., wrong).

The rule for distinguishing between `NoViableAlt` and `NoSemViableAlt` is:

> If `NoViableAlt` would be thrown and at least one semantic predicate (for a syntactically viable alternative) failed, signal `NoSemViableAlt` instead of `NoViableAlt`.

(*Semantic predicates that are not used to predict alternatives do not yet throw signals. You must continue to use the fail-action attached to individual predicates in these cases.*)

Resynchronizing the Parser

When an error occurs while parsing rule *R*, the parser will generally not be able to continue parsing anywhere within that rule. It will return immediately after executing any exception code. The one exception is for handlers attached to a particular rule reference. In this case, the parser knows exactly where in the alternative you would like to continue parsing from—immediately after the rule reference.

After reporting an error, your handler must resynchronize the parser by consuming zero or more tokens. More importantly, this consumption must be appropriate given the point where the parser will attempt to continue parsing. For example, given when an error occurs during the recognition of the conditional of an `if`-statement, a good way to recover would be to consume tokens until the `then` is found on the input stream.

```
stat   :  IF e:expr THEN stat
       ;
       exception[e]
          default : <<print error; consumeUntilToken(THEN);>>
```

The parser will continue with the parse after the `expr` reference (because we attached the exception handler to the rule reference) and look for the `then` right away.

To allow this type of manual resynchronization of the parser, two functions are provided:

TABLE 14. Resynchronization Functions

Function	Description
`consumeUntil(`*X*`_set)`	Consume tokens until a token in the token class *X* is seen. Recall that ANTLR generates a packed bit-set called *X*_set for each token class *X*. The C interface prefixes the function name with "`zz`".
`consumeUntilToken(`*T*`)`	Consume tokens until token *T* is seen. The C interface prefixes the function name with "`zz`".

For example,

```
#tokclass RESYNCH { A C }
a   :   b A
    |   b C
    ;
b   :   B
    ;
    exception
        catch MismatchedToken : // consume until FOLLOW(b)
            <<print error message; zzconsumeUntil(RESYNCH_set);>>
```

You may also use function `set_el(`*T*`, `*TC*`_set)` (prefix with "`zz`" in C interface) to test token type *T* for membership in a token class *TC*. For example,

```
<<if ( zzset_el(LA(1), TC_set) ) blah blah blah;>>
```

The @ Operator

You may suffix any token reference with the `@` operator, which indicates that if that token is not seen on the input stream, errors are to be handled immediately rather than throwing a `MismatchedToken` exception. In particular, [*for the moment*] the macros `zzmatch_wdfltsig()` or `zzsetmatch_wdfltsig()` is called in both C and C++ mode for simple token or token class references. In C++, you can override functions `ANTLRParser` member functions `_match_wdfltsig()` and `_setmatch_wdfltsig()`.

The `@` operator may also be placed at the start of any alternative to indicate that all token references in that alternative (and enclosed subrules) are to behave as if they had been suffixed with the '`@`' operator individually. Consider the following grammar:

```
stat
    :@ "if" INT "then" stat { "else" stat }
       <<printf("found if\n");>>
    |  id:ID@ "="@ INT@ ";"@
       <<printf("found assignment to %s\n", $id->getText());>>
    ;
```

The @ on the front of alternative one indicates that each token reference in the alternative is to be handled without throwing an exception. The match routine will catch the error. The second alternative explicitly indicates that each token is to be handled locally without throwing an exception.

ANTLR Command Line Arguments

ANTLR understands the following command line arguments:

`-CC`	Generate C++ output from ANTLR.
`-ck` *n*	Use up to *n* symbols of lookahead when using compressed (linear approximation) lookahead. This type of lookahead is very cheap to compute and is attempted before full LL(k) lookahead, which is of exponential complexity in the worst case. In general, the compressed lookahead can be much deeper (e.g, `-ck 10`) than the full lookahead (which usually must be less than 4).
`-cr`	Generate a cross-reference for all rules. For each rule, print a list of all other rules that reference it.
`-e1`	Ambiguities/errors shown in low detail (default).
`-e2`	Ambiguities/errors shown in more detail.
`-e3`	Ambiguities/errors shown in excruciating detail.
`-fe` *f*	File Rename `err.c` to *f*.
`-fh` *f*	File Rename `stdpccts.h` header (turns on -gh) to *f*.
`-fl` *f*	File Rename lexical output, `parser.dlg`, to *f*.
`-fm` *f*	File Rename file with lexical mode definitions, `mode.h`, to *f*.
`-fr` *f*	File Rename file which remaps globally visible symbols, `remap.h`, to *f*.
`-ft` *f*	File Rename tokens.h to *f*.
`-gc`	Indicates that antlr should generate no C code, i.e., only perform analysis on the grammar.

`-gd` C/C++ code is inserted in each of the ANTLR generated parsing functions to provide for user-defined handling of a detailed parse trace. The inserted code consists of calls to the user-supplied macros or functions called zzTRACEIN and zzTRACEOUT in C and calls to `ANTLRParser::tracein()` and `traceout()` in C++. The only argument is a `char *` pointing to a C-style string, which is the grammar rule recognized by the current parsing function. If no definition is given for the trace functions upon rule entry and exit, a message is printed indicating that a particular rule as been entered or exited.

`-ge` Generate an error class for each rule.

`-gh` Generate `stdpccts.h` for non-ANTLR-generated files to include. This file contains all defines needed to describe the type of parser generated by ANTLR (e.g. how much lookahead is used and whether or not trees are constructed) and contains the header action specified by the user. If your `main()` is in another file, you should include this file in C mode. C++ can ignore this option.

`-gk` Generate parsers that delay lookahead fetches until needed. Without this option, ANTLR generates parsers which always have *k* tokens of lookahead available. This option is incompatible with semantic predicates and renders references to `LA(`*i*`)` invalid as one never knows when the i^{th} token of lookahead will be fetched. [*This is broken in C++ mode*.]

`-gl` Generate line info about grammar actions in the generated C/C++ code of the form
`#` *line* `"`*file.g*`"`

which makes error messages from the C/C++ compiler more sensible because they point into the grammar file, not the resulting C/C++ file. Debugging is easier too, because you will step through the grammar, not C/C++ file.

`-gs` Do not generate sets for token expression lists; instead generate a `"||"`-separated sequence of `LA(1)==`*token_number*. The default is to generate sets.

`-gt` Generate code for Abstract Syntax Trees.

`-gx` Do not create the lexical analyzer files (dlg-related). This option should be given when you need to provide a customized lexical analyzer. It may also be used in make scripts to cause only the parser to be rebuilt when a change not affecting the lexical structure is made to the input grammars.

`-k` *n* Set *k* of LL(*k*) to *n*; i.e., set the number of tokens of look-ahead (default==1).

`-o` *dir* Directory where output files should go (default="."). This keeps the source directory clear of ANTLR and DLG spawn.

`-p` The complete grammar, collected from all input grammar files and stripped of all comments and embedded actions is listed to `stdout`. This enables viewing the entire grammar as a whole and eliminates the need to keep actions concisely stated so that the grammar is easier to read.

`-pa` This option is the same as `-p` except that the output is annotated with the first sets determined from grammar analysis.

`-prc on` Turn on the computation of predicate context (default is not to compute the context).

`-prc off` Turn off the computation and hoisting of predicate context (default case).

`-rl` *n* Limit the maximum number of tree nodes used by grammar analysis to *n*. Occasionally, ANTLR is unable to analyze a grammar. This rare situation occurs when the grammar is large and the amount of lookahead is greater than one. A nonlinear analysis algorithm is used by PCCTS to handle the general case of LL(k) parsing. The average complexity of analysis, however, is near linear due to some fancy footwork in the implementation which reduces the number of calls to the full LL(k) algorithm. An error message will be displayed, if this limit is reached, which indicates the grammar construct being analyzed when ANTLR hit a nonlinearity. Use this option if ANTLR seems to go out to lunch and your disk start thrashing; try *n*=80000 to start. Once the offending construct has been identified, try to remove the ambiguity that antlr was trying to overcome with large lookahead analysis. The introduction of `(...)?` backtracking predicates eliminates some of these problems—antlr does not analyze alternatives that begin with `(...)?` (it simply backtracks, if necessary, at run time).

`-w1` Set low warning level. Do not warn if semantic predicates and/or (...)? blocks are assumed to cover ambiguous alternatives.

`-w2` Ambiguous parsing decisions yield warnings even if semantic predicates or (...)? blocks are used. Warn if predicate context computed and semantic predicates incompletely disambiguate alternative productions.

`-` Read grammar from standard input and generate `stdin.c` as the parser file.

DLG Command Line Arguments

These are the command line arguments understood by DLG (normally, you can ignore these and concentrate on ANTLR):

`-CC` Generate C++ output. The output file is not specified in this case.

`-C`*`level`* Where level is the compression level used. 0 indicates no compression, 1 removes all unused characters from the transition from table, and 2 maps equivalent characters into the same character classes. Using level -C2 significantly reduces the size of the DFA produced for lexical analyzer.

`-m` *`f`* Produces the header file for the lexical mode with a name other than the default name of `mode.h`.

`-i` An interactive, or as interactive as possible, scanner is produced. A character is obtained only when required to decide which state to go to. Some care must be taken to obtain accept states that do not require look ahead at the next character to determine if that is the stop state. Any regular expression with a `"e*"` at the end is guaranteed to require another character of lookahead.

`-cl`*`class`* Specify a class name for DLG to generate. The default is `DLGLexer`.

`-ci` The DFA will treat upper and lower case characters identically. This is accomplished in the automaton; the characters in the lexical buffer are unmodified.

`-cs` Upper and lower case characters are treated as distinct. This is the default.

`-o` *`dir`* Directory where output files should go (default=”.”). This is very nice for keeping the source directory clear of ANTLR and DLG spawn.

`-Wambiguity`

Warns if more than one regular expression could match the same character sequence. The warnings give the numbers of the expressions in the DLG lexical specification file. The numbering of the expressions starts at one. Multiple warnings may be print for the same expressions.

`-` Used in place of file names to get input from `stdin` or send output to `stdout`.

C Interface

(*The C interface gradually evolved from a simplistic attributed-parser built in 1988. Unfortunately for backward compatibility reasons, the interface has been augmented but not changed in any significant way.*)

The C interface parsing model assumes that a scanner (normally built by DLG) returns the token type of tokens found in the input stream when it is asked to do so by the parser. The parser provides attributes that are computed from the token type and text of the token, to grammar actions to facilitate translations. The line and column information are directly accessed from the scanner. The interface requires only that you define what an attribute looks like and how to construct one from the information provided by the scanner. Given this information, ANTLR can generate a parser that will correctly compile and recognize sentences in the prescribed language.

The type of an attribute must be called `Attrib`; the function or macro to convert the text and token type of a token to an attribute is called `zzcr_attr()`.

This chapter describes the invocation of C interface parsers, the definition of special symbols and functions available to grammatical actions, the definition of attributes, and the definition of AST nodes.

Invocation of C Interface Parsers

C interface parsers are invoked via the one of the macros defined in Table 15 on page 149.

TABLE 15. C Interface Parser Invocation Macros

Macro	Description
`ANTLR(r,f)`	Begin parsing at rule `r`, reading characters from stream `f`.
`ANTLRm(r,f,m)`	Begin parsing at rule `r`, reading characters from stream `f`; begin in lexical class `m`.
`ANTLRf(r,f)`	Begin parsing at rule `r`, reading characters by calling function `f` for each character.
`ANTLRs(r,s)`	Begin parsing at rule `r`, reading characters from string `s`.

The rule argument must be a valid C function call, including any parameters required by the starting rule. For example, to read an `expr` from `stdin`:

```
ANTLR(expr(), stdin);
```

To read an `expr` from string `buf`:

```
char buf[] = "3+4";
ANTLRs(expr(), buf);
```

To read an `expr` and build an AST:

```
char buf[] = "3+4";
AST *root;
ANTLRs(expr(&root), buf);
```

To read an `expr` and build an AST where `expr` has a single integer parameter:

```
#define INITIAL      0
char buf[] = "3+4";
AST *root;
ANTLRs(expr(&root,INITIAL), buf);
```

A simple template for a C interface parser is the following:

```
#header <<
#include "charbuf.h"
>>
#token "[\ \t]+"        <<zzskip();>>
#token "\n"             <<zzskip(); zzline++;>>
<<
main() { ANTLR(start(), stdin); }
>>

start : ;
```

Functions and Symbols in Lexical Actions

Table 16 on page 151 describes the functions and symbols available to actions that are executed upon the recognition of an input token (In rare cases, however, these functions need to be called from within a grammar action).

TABLE 16. C Interface Symbols Available to Lexical Actions

Symbol	Description
`zzreplchar(char c)`	Replace the text of the most recently matched lexical object with `c`.
`zzreplstr(char c)`	Replace the text of the most recently matched lexical object with `c`.
`int zzline`	The current line number being scanned by DLG. This variable must be maintained by the user; this variable is normally maintained by incrementing it upon matching a newline character.
`zzmore()`	This function merely sets a flag that tells DLG to continue looking for another token; future characters are appended to `zzlextext`.
`zzskip()`	This function merely sets a flag that tells DLG to continue looking for another token; future characters are not appended to `zzlextext`.
`zzadvance()`	Instruct DLG to consume another input character. `zzchar` will be set to this next character.
`int zzchar`	The most recently scanned character.
`char *zzlextext`	The entire lexical buffer containing all characters matched thus far since the last token type was returned. See `zzmore()` and `zzskip()`.
`NLA`	To change token type to *t*, do "`NLA = t;`". This feature is not really needed anymore as semantic predicates are a more elegant means of altering the parse with run time information.
`NLATEXT`	To change token type text to *foo*, do "`strcpy(NLATEXT,foo);`". This feature sets the actual token lookahead buffer, not the lexical buffer `zzlextext`.

TABLE 16. (Continued) C Interface Symbols Available to Lexical Actions

Symbol	Description
`char *zzbegexpr`	Beginning of last token matched.
`char *zzendexpr`	End of last token matched.
`ZZCOL`	Define this preprocessor symbol to get DLG to track the column numbers.
`int zzbegcol`	The column number starting from 1 of the first character of the most recently matched token.
`int zzendcol`	The column number starting from 1 of the last character of the most recently matched token. Reset `zzendcol` to 0 when a newline is encountered. Adjust `zzendcol` in the lexical action when a character is not one print position wide (e.g., tabs or non-printing characters). The column information is not immediately updated if a token's action calls `zzmore()`.
`void (*zzerr)(char *)`	You can set `zzerr` to point to a routine of your choosing to handle lexical errors (e.g., when the input does not match any regular expression).
`zzmode(int m)`	Set the lexical mode (i.e., lexical class or automaton) corresponding to a lex class defined in an ANTLR grammar with the `#lexclass` directive.
`int zzauto`	What automaton (i.e., lexical mode) is DLG in?
`zzrdstream(FILE *)`	Specify that the scanner should read characters from the stream argument.
`zzclose_stream()`	Close the current stream.
`zzrdstr(zzchar_t *)`	Specify that the scanner should read characters from the string argument.
`zzrdfunc(int (*)())`	Specify that the scanner should obtain characters by calling the indicated function.
`zzsave_dlg_state( struct zzdlg_state *)`	Save the current state of the scanner. This is useful for scanning nested includes files, etc...
`zzrestore_dlg_state( struct zzdlg_state *)`	Restore the state of the scanner from a state buffer.

Attributes Using the C Interface

Attributes are objects that are associated with all tokens found on the input stream. Typically, attributes represent the text of the input token, but may include any information that you require. The type of an attribute is specified via the `Attrib` type name, which you must provide. A function `zzcr_attr()` is also provided by you to inform the parser how to convert from the token type and text of a token to an `Attrib`. [*In early versions of ANTLR, attributes were also used to pass information to and from rules or subrules. Rule arguments and return values are a more sophisticated mechanism and, hence, in this section, we will pretend as if attributes are only used to communicate with the scanner.*]

Attribute Definition and Creation

The attributes associated with input tokens must be a function of the text and the token type associated with that lexical object. These values are passed to `zzcr_attr()` which computes the attribute to be associated with that token. The user must define a function or macro that has the following form:

```
void zzcr_attr(attr, type, text)
Attrib *attr;    /* pointer to attribute associated with this lexeme */
int type;        /* the token type of the token */
char *text;      /* text associated with lexeme */
{
       /* *attr = f(text,token); */
}
```

Consider the following `Attrib` and `zzcr_attr()` definition.

```
typedef union {
          int ival; float fval;
       } Attrib;

zzcr_attr(Attrib *attr, int type, char *text)
{
       switch ( type )   {
       case INT : attr->ival = atoi(text); break;
       case FLOAT : attr->fval = atof(text); break;
   }
}
```

The `typedef` specifies that attributes are integer or floating point values. When the regular expression for a floating point number (which has been identified as `FLOAT`) is matched on the input, `zzcr_attr()` converts the string of characters representing that number to a C `float`. Integers are handled analogously.

You can specify the C definition or `#include` statements the file needed to define `Attrib` (and `zzcr_attr()` if it is a macro) using the ANTLR `#header` directive. The action associated with `#header` is placed in every C file generated from the grammar files. Any C file created by the user that includes `antlr.h` must once again define `Attrib` before using `#include antlr.h`. A convenient way to handle this is to use the `-gh` ANTLR command line option to have ANTLR generate the `stdpccts.h` file and then simply include `stdpccts.h`.

Attribute References

Attributes are referenced in user actions as `$`*`label`* where `label` is the label of a token referenced anywhere before the position of the action. For example,

```
#header <<
typedef int Attrib;
#define zzcr_attr(attr, type, text) *attr = atoi(text);
>>
#token "[\ \t\n]+"<<zzskip();>>       /* ignore whitespace */

add   :  a:"[0-9]+" "\+" b:"[0-9]+"
      <<printf("addition is %d\n", a+b);>>
      ;
```

If `Attrib` is defined to be a structure or union, then `$`*`label.field`* is used to access the various fields. For example, using the `union` example above,

```
#header <<
typedef union { ... };
>>
void zzcr_attr(...) { ... };
#token "[\ \t\n]+"<<zzskip();>>       /* ignore whitespace */

add   : a:INT "\+" b:FLOAT
        <<printf("addition is %f\n", $a.ival+$b.fval);>>
      ;
```

For backward compatibility reasons, ANTLR still supports the notation $*i* and $*i.j*, where *i* and *j* are a positive integers. The integers uniquely identify an element within the currently active block and within the current alternative of that block. With the invocation of each new block, a new set of attributes becomes active and the previously active set is temporarily inactive. The $*i* and $*i.j* style attributes are scoped exactly like local stack-based variables in C. Attributes are stored and accessed in stack fashion. With the recognition of each element in a rule, a new attribute is pushed on the stack. Consider the following simple rule:

```
a: B | C ;
```

Rule a has 2 alternatives. The $*i* refers to the i^{th} rule element in the current block and within the same alternative. So, in rule *a*, both B and C are $1.

Subrules are like code blocks in C—a new scope exists within the subrule. The subrules themselves are counted as a single element in the enclosing alternative. For example,

```
b  :  A ( B C <<action1>> | D E <<action2>> ) F <<action3>>
   |  G <<action4>>
   ;
```

Table 17 on page 155 describes the attributes that are visible to each action.

TABLE 17. Visibility and Scoping of Attributes

Action	Visible Attributes
action1	B as $1 (or $2.1), C as $2 (or $2.2), A as $1.1
action2	D as $1, E as $2, A as $1.1
action3	A as $1, F as $3
action4	G as $1

Attribute destruction

You may elect to "destroy" all attributes created with zzcr_attr(). A macro called zzd_attr(), is executed once for every attribute when the attribute goes out of scope. Deletions are done collectively at the end of every block. The zzd_attr() is passed the address of the attribute to destroy. This can be useful when memory is allocated with zzcr_attr() and needs to be free()ed; make sure to NULL the pointers. For example, sometimes zzcr_attr() needs to make copies of some lexical objects temporarily. Rather than explicitly inserting code into the grammar to free these copies, zzd_attr() can be used to do it implicitly. This concept is similar to the constructors and destructors of C++. Consider the case when attributes are character strings and copies of the lexical text buffer are made which later need to be deallocated. This can be accomplished with code similar to the following.

```
#header <<
typedef char *Attrib;
#define zzd_attr(attr) {free(*(attr));}
>>
<<
zzcr_attr(Attrib *attr, int type, char *text)
```

```
{
       if ( type == StringLiteral ) {
          *attr = malloc( strlen(text)+1 );
          strcpy(*attr, text);
       }
}
>>
```

Standard Attribute Definitions

Some typical attribute types are defined in the PCCTS include directory. These standard attribute types are contained in the following include files:

- `charbuf.h`. Attributes are fixed-size text buffers, each 32 characters in length. If a string longer than 31 characters (31 + 1 '`\0`' terminator) is matched for a token, it is truncated to 31 characters. You can change this buffer length from the default 32 by redefining `ZZTEXTSIZE` before the point where `charbuf.h` is included. The text for an attribute must be referenced as `$i.text`.
- `int.h`. Attributes are `int` values derived from tokens using the `atoi()` function.
- `charptr.h`, `charptr.c`. Attributes are pointers to dynamically allocated variable-length strings. Although generally both more efficient and more flexible than `charbuf.h`, these attribute handlers use `malloc()` and `free()`, which are not the fastest library functions. The file `charptr.c` must be used with `#include`, or linked with the C code ANTLR generates for any grammar using `charptr.h`.

Interpretation of Symbols in C Actions

TABLE 18. C Interface Interpretation of Attribute Terms in Actions

Symbol	Meaning
`$`*`j`*	The attribute for the *j*th element of the current alternative. The attribute counting includes actions. Subrules embedded within the alternative are counted as one element.
`$`*`i.j`*	The attribute for the *j*th element of *i*th level starting from the outermost (rule) level at 1.
`$0`	The result attribute of the immediately enclosing subrule or rule. (l-value)
`$$`	The result attribute of the enclosing rule. (l-value)
`$`*`arg`*	The rule argument labeled *`arg`*.
`$`*`rule`*	The result attribute of the enclosing rule; this is the same as `$$`. (l-value)
`$`*`rv`*	The rule return result labeled *`rv`*. (l-value)
`$[`*`token_type`*`,`*`text`*`]`	Attribute constructor; this is translated to a call to `zzconstr_attr(`*`token`*`,`*`text`*`)`.
`$[]`	An empty, initialized attribute; this is translated to a call to `zzempty_attr()`.

AST Definitions

AST nodes using the C interface always have the following structure:

```
struct _ast {
       struct _ast *right, *down;
       user_defined_fields
};
```

where you must fill in *`user_defined_fields`* using the `AST_FIELDS` `#define` and must be provided in the `#header` action or in an include file included by the `#header` action. Only the user-defined fields should be modified by the user as `right` and `down` are

handled by the code generated by ANTLR. The type of an AST is provided by ANTLR and is always called `AST`.

You must define what the AST nodes contain and also how to fill in the nodes. To accomplish this, you supply a macro or function to convert from the attribute, text, and token type of an input token to a tree node. ANTLR calls the function

```
zzcr_ast(AST *ast, Attrib *attr, int token_type, char *text);
```

to fill in tree node `ast` by calling

```
zzcr_ast(zzastnew(), attrib-of-current-token, LA(0), LATEXT(0))
```

The following template can be used for C interface tree building:

```
#header <<
#define AST_FIELDS                what-you-want-in-the-AST-node
#define zzcr_ast(ast,attr,ttype,text)ast->field(s) = ... ;
>>.
```

TABLE 19. C Interface AST Support Functions

Function	Description
`zzchild(AST *t)`	Return the child of `t`.
`zzsibling(AST *t)`	Return the next sibling to the right of `t`.
`zzpre_ast(AST *t, FuncPtr n, FuncPtr before, FuncPtr after)`	Do a depth-first walk of the tree applying `n` to each node, `before` before each subtree, and `after` after each subtree.
`zzfree_ast(AST *t)`	Free all AST nodes in the tree. Calls `zztfree()` on each node before `free()`ing.
`AST *zztmake(AST *t, ...)`	Build and return a tree with `t` as the root and any other arguments as children. A `NULL` argument (except for `t`) terminates the list of children. Arguments other than the root can themselves be trees.
`AST *zzdup_ast(AST *t)`	Duplicate the entire tree `t` using `zzastnew()` for creating new nodes.
`zztfree(AST *t)`	If macro `zzd_ast()` is defined, invokes `zzd_ast()` on `t` and then frees `t`.

TABLE 19. (Continued) C Interface AST Support Functions

Function	Description
`zzdouble_link(AST *t, AST *up, AST *left)`	Set the `up` and `left` pointers of all nodes in `t`. The initial call should have `up` and `left` as `NULL`.
`AST *zzastnew(void)`	Use `calloc()` to create an AST node.

The invocation of the start symbol must pass the address of a `NULL`-initialized tree pointer because ANTLR passes the address of the tree to fill in to each rule when the `-gt` option is turned on:

```
main()
{
    AST *root=NULL;
    ANTLR(starting_rule(&root), stdin);
}
```

After the parse has completed, `root` will point to the tree constructed by *`starting_rule`*.

4 SORCERER Reference

SORCERER is a simple tree-parser and translator generator that has the notation and the "flavor" of ANTLR. It accepts a collection of rules that specifies the contents and structure of your trees and generates a top-down, recursive-descent program that will walk trees of the indicated form. Semantic actions may be interspersed among the rule elements to effect a translation—either by constructing an output tree (from the input tree) or by generating output directly. SORCERER has some pretty decent tree rewriting and miscellaneous support libraries.

This chapter describes how to construct tree parsers via SORCERER grammars, how to interface a tree parser to a programmer's application, how to insert actions to generate output, and how to perform tree transformations. Unless otherwise specified, actions and other source code is C++.

(*Aaron Sawdey, Ph.D. candidate at University of MN, and Gary Funck of Intrepid Technology are coauthors of SORCERER.*)

Introductory Examples

It is often best to introduce a language tool with a simple example that illustrates its primary function. We consider the task of printing out a simple expression tree in both postfix and infix notation. Assuming that expression trees have operators at subtree parent nodes and operands are children, the expression "`3+4`" could be represented with the tree structure:

where the 3 and 4 children nodes (of token type INT) are operands of the addition parent node. We will also assume that INT type nodes have a field called val that contains the integer value associated with the node.

```
expr   :  #( PLUS expr expr )         <<printf(" +");>>
       |  i:INT                       <<printf(" %d",i->val);>>
       ;
```

The labeled element "i:INT" specifies that an INT node is to be matched and a pointer called i is to be made available to embedded actions within that alternative. Given the indicated expression tree, this rule would generate "3 4 +".

To construct an infix version, all we have to do is move the action, thus changing the sequence of action executions. Naturally, the structure of the tree has not changed and the grammar itself is no different.

```
expr   :  #( PLUS expr <<printf(" +");>> expr )
       |  i:  INT     <<printf(" %d",i->val);>>
       ;
```

The output generated by expr is now "3 + 4" for the same input tree.

Constructing tree walkers by hand to perform the same tasks is simple as well, but becomes more difficult as the tree structure becomes more complicated. Further, a program is harder to read, modify, and maintain than a grammar. The following C function is equivalent to the postfix grammar.

```
expr(SORAST *t)
{
   if ( t->token==PLUS ) {
      MATCH(PLUS);
      expr(t->down);
      expr(t->down->right);
      printf(" +");
   }
   else {
      MATCH(INT);
      printf(" %d",i->val);
   }
}
```

This hand-built function is not as robust as the SORCERER generated function, which would check for NULL trees and for trees that did not have PLUS or INT as root nodes.

SORCERER is suitable for rewriting input trees as well as generating output via embedded actions. Consider how one might swap the order of the operands for the same expression tree given above. Having to manipulate child-sibling tree pointers manually is tedious and error prone. SORCERER supports tree rewriting via a few library functions, grammar element labels, and a tree constructor. Again, because our tree structure has not changed, our grammar remains the same; but the actions are changed from generating text output to generating a slightly modified tree.

```
expr   :! #( a:PLUS b:expr c:expr )   <<#expr=#(a,c,b);>>
       |  i:INT
       ;
```

SORCERER is informed that a tree transformation is desired via the `-transform` command line option. In transform mode, SORCERER generates an output tree for every input tree and by default copies the input tree to the output tree unless otherwise directed. The first alternative in our example is annotated with a "`!`" to indicate that SORCERER should not generate code to construct an output tree as our embedded action will construct the output tree. Action

```
#expr=#(a,c,b);
```

indicates that the output tree of rule `expr` has the same root, the node pointed to by `a`, and the same children, but with the order reversed. Given the input tree for "`3+4`", the output tree of `expr` would be

C++ Programming Interface

SORCERER generates parsers that can walk any tree that implements a simple interface. You must specify:

1. The type of a tree node, called `SORAST` (derived from class `SORASTBase` and given in the `#header` action)
2. How to navigate a tree with member functions `down()` and `right()`
3. And how to distinguish between different nodes via member function `type()` which returns the token type associated with a tree node

If you wish to perform tree rewrites, you must also specify how to construct new trees via `setDown()`, `setRight()`, and `shallowCopy()` where `t->shallowCopy()` returns a duplicate of node `t`. If you refer to `#[...]` in an action, you must also define a constructor with the appropriate arguments.

For example, the most natural implementation of a child-sibling tree conforming to the SORCERER tree interface (transform or nontransform mode) and can be used is the following:

```
class SORCommonAST : public SORASTBase {
protected:
    SORCommonAST *_right, *_down;
    int _type;

public:
    PCCTS_AST *right()                { return _right; }
    PCCTS_AST *down()                 { return _down; }
    int type()                        { return _type; }
    void setRight(PCCTS_AST *t)       { _right = (SORCommonAST *)t; }
    void setDown(PCCTS_AST *t)        { _down = (SORCommonAST *)t; }
    PCCTS_AST *shallowCopy()
        {
            SORCommonAST *p = new SORCommonAST;
            if ( p==NULL ) panic();
            *p = *this;
            p->setDown(NULL);
            p->setRight(NULL);
            return (PCCTS_AST *)p;
        }
};
```

This definition is also satisfactory for performing tree rewrites as SORCERER knows how to set the pointers and the token type and knows how to duplicate a node.

A SORCERER grammar contains a class definition that results in a C++ class derived from `STreeParser`. For example,

```
class MyTreeParser {
some-actions
some-rules
}
```

Each of the rules defined within the class definition will become public member functions of `MyTreeParser`. Any actions defined within the class definition must contain valid C++ member declarations or definitions.

Invoking a tree parser is a simple matter of creating an instance of the parser and calling one of the rules with the address of the root of the tree you wish to walk and any arguments you may have defined for that rule; for example.,

```
main()
{
    MyTreeParser myparser;
    SORAST *some_tree = ... ;
    myparser.some_rule((SORASTBase **)&some_tree);
}
```

The cast on the input tree is required because SORCERER must generate code for *some_rule* that can walk any kind of conformant tree:

```
void MyTreeParser::some_rule(SORASTBase **_root);
```

Unfortunately, without the cast you get a compiler error complaining that "SORASTBase **" is not the same as "SORAST **".

If *some_rule* had an argument and return value such as

```
some_rule[int i] > [float j] : ... ;
```

the invocation would change to

```
main()
{
    MyTreeParser myparser;
    SORAST *some_tree = ... ;
    float result=myparser.some_rule((SORASTBase **)&some_tree, 34);
}
```

Table 20 on page 165 describes the files generated by SORCERER from a tree description in file(s) *f1*.sor ... *fn*.sor.

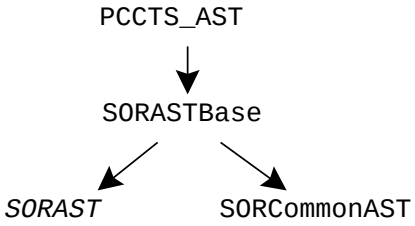
TABLE 20. Files Written by SORCERER For C++ Interface

File	Description
f1.cpp ... *fn*.cpp	Definition of rules specified in *f1*.sor ... *fn*.sor.
Parser.h	Declaration of class *Parser* where *Parser* is defined in the SORCERER grammar. All grammar rules become member functions.
Parser.cpp	Definition of support member functions of class *Parser*.

There are no global variables defined by SORCERER for the C++ interface and, hence, multiple SORCERER tree parsers may easily be linked together; e.g., C++ files generated by SORCERER for different grammars can be compiled and linked together without fear of multiply-defined symbols. This also implies that SORCERER parsers are "thread-safe" or "re-entrant."

C++ Class Hierarchy

Trees walked by SORCERER are all derived directly or indirectly from `PCCTS_AST` so that ANTLR and SORCERER both have access to the same member functions through the same C++ object *vtables*:



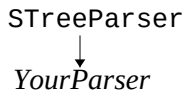
The classes are described as follows:

- `PCCTS_AST`. SORCERER and ANTLR trees must be derived from the same base class if they both are to manipulate the same trees; i.e., both tools must have access to a common object virtual table for the tree nodes. The standard SORCERER public interface is described in `PCCTS_AST`:

```
virtual PCCTS_AST *right();
virtual PCCTS_AST *down();
virtual void setRight(PCCTS_AST *t);
virtual void setDown(PCCTS_AST *t);
virtual int type();
virtual PCCTS_AST *shallowCopy();
```

- `SORASTBase`. This is currently `typedef`'d to `PCCTS_AST` so that the return type of the C++ tree library routines can be `SORASTBase` rather than `PCCTS_AST`, which could cause some confusion for those using SORCERER without ANTLR.
- `SORAST`. You must define this type name as the type of tree node that SORCERER should walk or transform.
- `SORCommonAST`. This class is useful when ANTLR is not used to construct the input trees of the tree parser. It defines a typical child-sibling implementation.

The parser class defined in your SORCERER grammar description is constructed as a derived class of `STreeParser`:



which defines the standard SORCERER tree parser behavior.

TABLE 21. C++ Files

File	Defines
`h/PCCTSAST.h`, `lib/PCCTSAST.cpp`	Class `PCCTS_AST` (which contains the standard ANTLR/SORCERER tree interface).
`h/SASTBase.h`	`SORASTBase`. Currently this is only a typedef to `PCCTS_AST`.
`h/SCommonAST.h`	`SORCommonAST`.
`h/SList.h`, `lib/SList.cpp`	`SList` class.
`h/STreeParser.h`, `lib/STreeParser.cpp`	`STreeParser` class.
`h/config.h`	Configuration information for the various platforms.

Token Type Definitions

SORCERER generates a parser that recognizes both the structure and a portion of the contents of a tree node. Member `type()` is the only information used to determine the contents of a node; however, semantic predicates can be used to alter the parse depending on other node information. SORCERER-generated parsers test the content of a node with an expression of the form "`_t->type()==`*T*". For the SORCERER output to compile, token type *T* must have been defined.

SORCERER-generated translators can use either user-defined token types or can have SORCERER assign token types to each terminal referenced in the grammar description. The "`-def-tokens` *`file`*" option is used to generate a file with `#define`s for each referenced token. When folding a SORCERER parser into an existing application, the token types will already be defined. These definitions can be explicitly included via a C or C++ action, or the

file containing the token types can be specified with a `#tokdefs` directive. The file may only contain the definitions, but may be in the form of `#define`s or an `enum`. For example,

```
<<enum TokenType { A=1, B=2 };>>
a : A B;
```

or,

```
#tokdefs "mytokens.h"
a : A B;
```

To use the range operator, $T_1 .. T_2$, the `#tokdefs` directive must be used because the actual value of the token types must be known to verify the pred-LL(1) nature of a programmer's grammar.

A token type of `0` is illegal.

Using ANTLR and SORCERER Together

To have ANTLR construct trees and have SORCERER walk them, do the following:

1. Define a `type()` field in the `AST` definition for ANTLR. E.g.,

   ```
   #include "ATokPtr.h"
   class AST : public ASTBase {
   protected:
       int _type;
   public:
   AST(ANTLRTokenPtr t)        { _type = t->getType(); }
   AST()                       { _type = 0; }
   int type()                  { return _type; }
   };
   ```

2. Construct trees via ANTLR as you normally would. Ensure that any token type that you will refer to in the SORCERER grammar has a label in the ANTLR grammar. For example,

   ```
   #token ASSIGN "="
   ```

3. In your SORCERER description, include the `AST` definition you gave to ANTLR and define `SORAST` to be `AST`. For example,

   ```
   #header <<
   #include "AST.h"         /* include your ANTLR tree def */
   typedef AST SORAST;
   >>
   ```

4. Create a main program that calls both ANTLR and SORCERER routines.

```
#include "tokens.h"
#include "TextParser.h"
typedef ANTLRCommonToken ANTLRToken;
#include "TreeParser.h"
#include "DLGLexer.h"
#include "PBlackBox.h"
main()
{
    ParserBlackBox<DLGLexer,
                   TextParser,
                   ANTLRToken> lang(stdin);
    AST *root=NULL;
    TreeParser tparser;
    lang.parser()->stat((ASTBase **)&root);
    tparser.start_symbol((SORASTBase **)&root);
}
```

SORCERER Grammar Syntax

Just as ANTLR grammars specify a sequence of actions to perform for a given input sentence, SORCERER descriptions specify a sequence of actions to perform for a given input tree. The only difference between a conventional text language parser and a tree parser is that tree parsers have to recognize tree structure as well as grammatical structure. For this reason, the only significant difference between ANTLR input and SORCERER input is that SORCERER grammar productions can use an additional grouping construct—a construct to identify the elements and structure of a tree. This section summarizes SORCERER input syntax.

A SORCERER description is a collection of rules in Extended BNF (EBNF) form and user-defined actions preceded by a header action where the programmer defines the type of a SORCERER tree:

```
#header <<header action>>
actions
rules
actions
```

where *actions* are enclosed in European quotes <<...>> and rules are defined as follows:

```
rule   :  alternative1
       |  alternative2
       ...
       |  alternativen
       ;
```

Each alternative production is composed of a list of elements where an element can be one of the items in Table 22 on page 170. The "..." within the grouping constructs can themselves be lists of alternatives or items. C and C++ style comments are ignored by SORCERER

TABLE 22. SORCERER Description Elements

Item	Description	Example
leaf	Token type	`ID`
$T_1..T_2$	Token range	`OPSTART..OPEND`
`.`	Wild card	`#( FUNC ID (.)* )`
rule-name	Reference to another rule	`expr`
label`:`*elem*	Label an element	`id:ID`
`#(...)`	Tree pattern	`#(ID expr slist slist)`
`<<...>>`	User-defined semantic action	`<<printf("%d",i->val);>>`
`(...)`	Subrule	`(STRING | ID | FLOAT)`
`(...)*`	Zero-or-more closure subrule	`args : (expr)* ;`
`(...)+`	One-or-more positive closure	`slist: #(SLIST (stat)+ )`
`{...}`	Optional subrule	`#(IF expr stat {stat})`
`<<...>>?`	Semantic predicate	`id : <<isType()>>? ID`

`(...)?`	Syntactic predicate	`( #( MINUS expr expr )?`
		`| #( MINUS expr )`
		`)`

Rule Definitions: Arguments and Return Values

All rules begin with a lowercase letter and may declare arguments and a return value in a manner similar to C and C++:

`rule[`arg_1`, `arg_2`, ..., `arg_n`] > [`*return-type id*`] : ... ;`

which declares a `rule` to have *n* arguments and a return value; either may be omitted. For example, consider

```
a[int a, int b] > [int r]
    :   some-tree <<r=a+b;>>
    ;
```

which matches *some-tree* and returns the sum of the two arguments passed to the rule. The return value of a rule is set by assigning to the variable given in the return block `[...]`.

The invocation of rule `a` would be of this form:

```
b   :   <<int local;>>
        blah a[3,4] > [local] foo
    ;
```

The result of `7` will be stored into `local` after the invocation of rule `a`. Note that the syntax of rule return value assignment is like UNIX I/O redirection and mirrors the rule declaration.

As a less abstract example, consider that it is often desirable to pass a value to an expression rule indicating whether it is on the left or right hand side of an assignment:

```
<<enum SIDE { LHS, RHS };>>
stat   :   #( ASSIGN expr[LHS] expr[RHS] )
       |   ...
       ;
expr[SIDE s] : ...
       ;
```

Return values are also very useful. The following example demonstrates how the number of arguments in a function call can be returned and placed into a local variable of the invoking rule.

```
expr
    :  ID
    |  FLOAT
    |<<int n;>>
       fc > [n]
       <<printf("func call has \%d arguments\\n", n);>>
    ;
fc > [int nargs]
    :  <<int i=0;>> #( FUNC ID ( . <<i++;>> )* )
       <<nargs = i;>>
    ;
```

Special Actions

The first action of any production is considered an *init-action* and can be used to declare local variables and perform some initialization code before recognition of the associated production begins. The code is executed only if that production is predicted by the look ahead of the tree parser. Even when normal actions are shut off during the execution of a syntactic predicate, `(...)?`, init-actions are executed. They cannot be enclosed in curly-braces because they define local variables that must be visible to the entire production—not just that action.

Actions suffixed with a "?" are semantic predicates and discussed below.

Special Node References

The simplest node specifier is a token, which begins with an uppercase letter. To specify that any token may reside in a particular node position, the wildcard "`.`" is used.

The wildcard sometimes behaves in a strange, but useful, manner as the "match anything else" operator. For example,

```
a : A | . ;
```

matches `A` in the first production and anything else in the second. This effect is caused by SORCERER's parsing strategy. It tries the productions in the order specified. Any non-`A` node will be bypassed by the first production and matched by the second.

Wildcards cannot be used to predict two different productions; that is,

```
a  :  .
   |  .
   |  A
   ;
```

results in a warning from SORCERER:

```
f.sor, line 1: warning: alts 1 and 2 of (...) nondeterministic upon {.}
```

To indicate that a specific range of tokens is to be matched, the range operator "$T_1 . . T_2$" is used. For example,

```
<<
#define Plus      1
#define Minus     2
#define Mult      3
#define Div       4
#define INT       10
#define OpStart   Plus
#define OpEnd     Div
>>
expr
   :  #( OpStart..OpEnd expr expr )
   |  INT
   ;
```

matches any expression tree comprised of integers and one of the four arithmetic operators.

For the range operator to work properly, $T_1 \leq T_2$ must hold and all values from T_1 ,and T_2 inclusively must be valid token types. Furthermore, the `#tokdefs` directive must be used to provide SORCERER with the actual token type values.

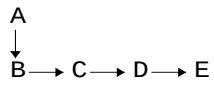
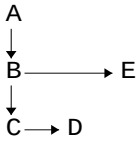
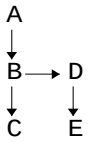
Tree Patterns

Tree patterns are specified in a LISP-like notation of the form:

`#(` *`root-item item ... item`* `)`

where the "#" distinguishes a parenthetical expression from the EBNF grouping construct, `(...)`, and *`root-item`* is a leaf node identifier such as `ID`, the wildcard, or a token range. For example, consider the tree specification and graphical-representation pairs given in Table 23 on page 174.

TABLE 23. Sample Tree Specification and Graphical Representation Pairs

Tree Description	Tree Structure
`#(A B C D E)`	
`#(A #(B C D) E)`	
`#(A #(B C) #(D E))`	

Flat trees (lists of items without parents) are of the form:

`item ... item`

Rule references and subrules may not be tree roots because they may embody more than a single node. By definition, the root of a tree must be a single node.

EBNF Constructs in the Tree-Matching Environment

A tree without any EBNF subrules is a tree with fixed structure. When tree patterns contain EBNF subrule specifications, the structure of the tree language may be difficult to see for humans. This section provides numerous examples that illustrate the types of trees that can be matched with EBNF constructs and the tree specifier, `#( . . . )`. Table 24 on page 175, Table 25 on page 175, Table 24 on page 175, and Table 24 on page 175 illustrate the various EBNF constructs and the tree structures the constructs match. One final note: EBNF constructs may not be used as subtree roots..

TABLE 24. EBNF Subrules

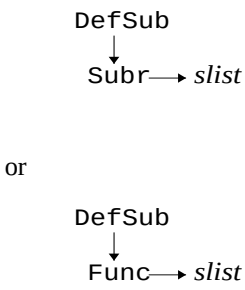
Tree Description	Possible Tree Structures
`#(DefSub (Subr|Func) slist)`	

TABLE 25. EBNF Optional Subrules

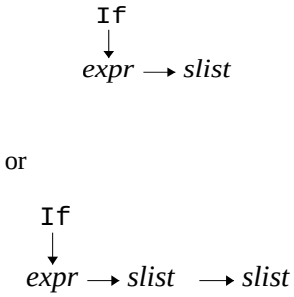
Tree Description	Possible Tree Structures
`#(If expr slist {slist})`	

TABLE 26. EBNF Zero-Or-More Subrules

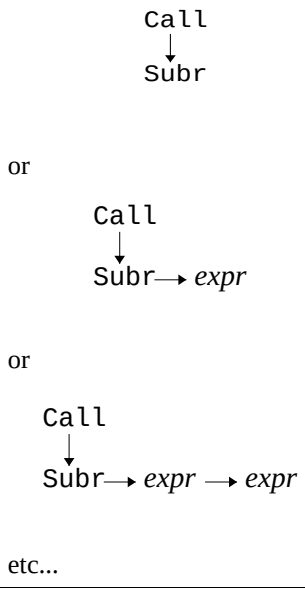
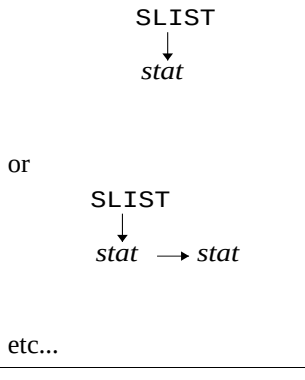
Tree Description	Possible Tree Structures
`#(Call Subr (expr)*)`	

TABLE 27. EBNF One-Or-More Subrules

Tree Description	Possible Tree Structures
`#(SLIST (stat)+)`	

Element Labels

Rule elements can be labeled with an identifier (any uppercase or lowercase string), which is automatically defined (and initialized to `NULL`) as a `SORAST` node pointer, that can be referenced by user actions. The syntax is:

`t:element`

where *t* may be any upper or lower case identifier and *`element`* is either a token reference, rule reference, the wild-card, or a token range specification.

Subtrees may be labeled by labeling the root node. The following grammar fragment illustrates a typical use of an element labels.

```
a   :   #( DO u:ID expr expr #( v:SLIST (stat)* ) )
        <<
        printf("induction var is %s\n", u->symbol());
        analyze_code(v);
        >>
    ;
```

where `symbol()` is some member function that the user has defined as part of a `SORAST`. These labels are pointers to the nodes associated with the referenced element and are only available after the recognition of the associated tree element; the one exception is a labeled rule reference, whose label is available for use as an argument to the referenced rule; e.g.,

```
a : p:b[p->symbol()] ;
```

Labels have rule scope—they are defined at the rule level and are not local to a particular subrule or alternative.

Labels can be used to test for the presence of an optional element. Therefore, in:

```
expr_list: oprnd:expr { Comma oprnd:expr } ;
```

variable `oprnd` will have the value of the second operand if the `Comma` is present and the first operand if not. It can also be used to track the last element added to a list:

```
expr_list: oprnd:expr ( Comma oprnd:expr )* ;
```

Note that there are no `$`-variables such as there are in ANTLR.

In transform mode, *`label`* refers to the output node associated with the labeled grammar element. To access the associated input node use *`label`*`_in`.

@-Variables

Local stack-based variables are convenient because a new instance is created upon each invocation of a function. However, unlike global variables, the symbol is only visible within that function (or SORCERER rule). Another function cannot access the variable. A stack-based variable that had global scope would be extremely useful; it would also be nice if that variable did not require the use of any "real" global variables. We have created just such creatures and called them *@-variables* for lack of a better name (the concept was derived from NewYacc). (*@-variables are mainly useful with the C interface because the C++ interface allows you to define tree-parser class member variables.*)

An @-variable is defined like a normal local variable, but to inform SORCERER of its existence, you must use a special syntax:

`@(`*`simple-type-specifier id = init-value`*`)`

To reference the variable, you must also inform SORCERER with `@`*`id`*. For example,

```
a   :   <<@(int blah)>> /* define reference var: "int blah;" */
        <<@blah = 3;>>  /* set blah to 3 */
        b
        <<printf("blah = %d\n", @blah);>>  /* prints "blah = 5 */
    ;
b   :   c ;
c   :   d ;
d   :   A
        <<
        printf("blah = %d\n", @blah);      /* prints "blah = 3 */
        @blah = 5;
        >>
    ;
```

where the output should be

```
blah = 3
blah = 5
```

The notation `@`*`id`* is used just like plain *`id`*; i.e., as the object of an assignment or as part of a normal C/C++ expression.

As another example, consider a grammar that provides definitions for implicitly defined scalars.

```
routine
    :   #( DefSub ID slist )
    ;
slist
    :   <<@(AST * defs)>>
```

```
        #( SLIST
            ( v:vardef <<define(v);>> )*
            <<@defs=v;>>
            ( stat )*
        )
    ;
vardef
    :   #( Type ID )
    ;
stat:   #( Assign scalar:ID INT )
        <<if ( scalar is-not-defined )
            ast_append(@defs, #(#[Type,"real"], #[ID,id->symbol]));
        >>
    ;
```

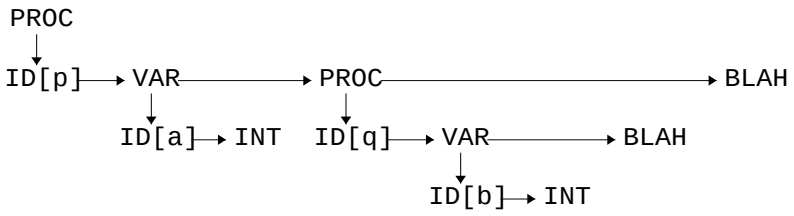
An @-variable is saved upon entry to the rule that defines it and restored upon exit from the rule. To demonstrate this stack-oriented nature, consider the following example,

```
proc:   #( p:PROC          <<printf("enter proc %s\n", p->symbol);>>
            ID (decl|proc)* (stat)*
        )                 <<printf("exit proc %s\n", p->symbol);>>
    ;
decl:   <<@(AST * lastdecl)>>
        #( VAR d:ID INT )  <<@lastdecl = d;
                             printf("def %s\n", d->symbol);>>
    ;
stat:   BLAH    <<printf("last decl is %s\n", @lastdecl->symbol;>>
    ;
```

Given some input such as:

```
procedure p;
var a : integer;
    procedure q;
    var b : integer;
    begin
        blah;
    end;
begin
    blah;
end;
```

with the intermediate form structure:



The output for this is

```
enter proc
def a
enter proc
def b
last decl is b
exit proc
last decl is a
exit proc
```

If you want the functionality of a normal C/C++ global variable but do not want the problems associated with a global variable, @-variables should also be used. When an @-variable definition is prefixed with `static`, it is never save or restored like other @-variables. For example,

```
a : <<@(static int blah)>> ... b ... ;
b : ... <<@blah = 3;>> ... ;
```

Essentially, `@blah` is a global variable; there just happens to be a new copy for every `STreeParser` that you define.

While @-variables, strictly speaking, provide no more functionality than passing the address of local variables around, @-variables are much more convenient.

When using the C++ interface, simple parser member variables are functionally equivalent to static @-variables.

Embedding Actions For Translation

In many cases, translation problems are simple enough that a single pass over an intermediate form tree is sufficient to generate the desired output. This type of translation is very straightforward and is the default mode of operation.

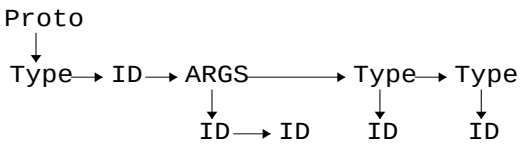
Translations of this type are done with a series of print statements sprinkled around the SORCERER grammar. For example, consider how one might convert function prototypes from K&R C to ANSI C (assuming the arguments and the declarations are in the same order):

```
void f(i,j)
int i;
float j;
```

would be converted to

```
void f(int i, float j);
```

in source form. Graphically, a prototype could be structured as follows:



where `Proto` and `ARGS` are imaginary tokens (tokens with no corresponding input symbol) used to structure the intermediate form tree. The `ID` directly under the `Proto` node is the function name.

The following tree grammar fragment could be used to reconize these simplified C prototypes:

```
proto
    :   #( Proto Type ID #( ARGS ( ID )* ) ( decl )* )
    ;

decl
    :   #( Type ID )
    ;
```

To perform the transformation, the grammar could be augmented with actions in the following manner:

```
proto
    :   #(  Proto t:Type f:ID
            <<printf("%s %s(", t->symbol(), f->symbol());>>
            #( ARGS ( ID )* )
            (  d:decl      <<if (d->right()!=NULL) printf(",");>>
            )*             <<printf(");\n");>>
        )
     ;

decl
    :   #( t:Type id:ID )
        <<printf("%s %s", t->symbol(), id->symbol());>>
    ;
```

where `symbol()` is a member that returns the textual representation of the type or function name.

Embedding Actions for Tree Transformations

While the syntax-directed scheme presented in the previous section is sometimes enough to handle an entire translation, it will not handle translations requiring multiple passes. In fact, if the translation can be handled with a simple syntax-directed translation from the intermediate form, it could probably be handled as a syntax-directed translation directly from the original, text input. Why even discuss syntax-directed translation for intermediate forms? Because a programmer can rewrite a tree innumerable times but must eventually convert the intermediate form to an output form.

This section describes the support available to the programmer in rewriting portions of an intermediate form. We provide information about how SORCERER rewrites trees, about the tree library, and about the other support libraries.

When tree transformations are to be made, the command-line option `-transform` must be used. In transform mode, SORCERER makes the following assumptions:

1. An input tree exists from which an output tree is derived.
2. If given no instructions to the contrary, SORCERER automatically copies the input tree to the output tree.
3. Each rule has a result tree, and the result tree of the first rule called is considered the final, transformed tree. This added functionality does not affect the normal rule argument and return value mechanism.

4. Labels attached to grammar elements are generally referred to as *label*, where *label* refers to the input tree subtree in nontransform mode.

   The output tree in transform mode is referred to as *label*. The input node, *for token references only*, can be obtained with *label*_in. The input subtree associated with rule references is unavailable after the rule has been matched. The tree pointer points to where that rule left off parsing. Input nodes in transform mode are not needed very often.

5. A C/C++ variable exists for any labeled token reference even if it is never set by SORCERER.
6. The output tree of a rule can be set and/or referenced as #*rule*.

The following sections describe the three types of tree manipulations.

Deletion

When portions of a SORCERER tree are to be deleted, the programmer has only to suffix the items to delete with a "!"; this effectively filters which nodes of the input tree are copied to the output tree. For example, if all exponent operators were to be removed from an expression tree, something similar to the following grammar could be used:

```
expr:  #( Plus expr expr )
    |  #( Mult expr expr )
    |! #( Exp expr expr )
    ;
```

where a "!" appended to an alternative operator "!" indicates that the entire alternative should not be included in the output tree. Token and rule references can also be individually deleted. The functionality of previous example, can be specified equivalently as:

```
expr:  #( Plus expr expr )
    |  #( Mult expr expr )
    |  #( Exp! expr! expr! )
    ;
```

No output tree nodes are constructed for the token references in the examples above. However, a labeled token reference always results in the generation of an output tree node regardless of the "!" suffixes. If you do not link the output node into the result tree or delete it explicitly in an action, a "memory leak" will exist.

Modification

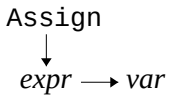
To rewrite a portion of a tree, you specify that the nodes of the tree are not to be automatically linked into the output tree via an appropriately-placed "!" operator. It is then up to you to describe the tree result for the enclosing rule. For example, let's assume that we want to translate assignments of the form

expr `->` *var*

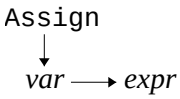
to

var `:=` *expr*

Assuming that the AST transformation was from



to



the following grammar fragment could perform this simple operation:

```
assign
    :!     #(a:Assign e:expr id:ID)   <<#assign = #(a, id, e);>>
    ;
```

The "`#(a, id, e)`" is a tree constructor that behaves exactly like it does in an ANTLR grammar. `a` is the root of a tree with two children: `id` (a pointer to a copy of the input node containing `ID`) and `e` (the result returned by `expr`).

You must be careful not to refer to the same output node more than once. Cycles can appear in the output tree, thus making it a graph not a tree. Also, be careful not to include any input tree nodes in the output tree.

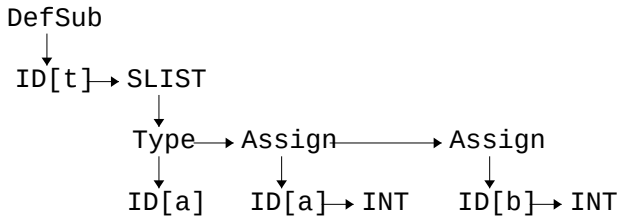
Augmentation

The result tree of a rule can be augmented as well as rearranged. This section briefly describes how the tree library routines (illustrated in the next section) can be used to add subtrees and nodes to the output tree.

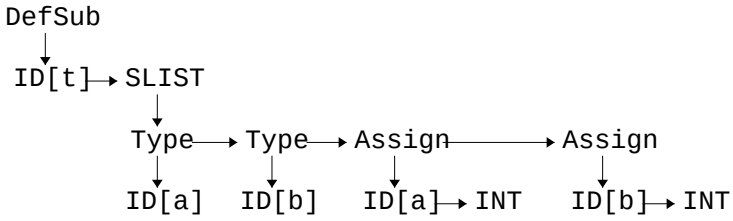
Consider the task of adding variable definitions for implicitly defined scalars in FORTRAN. Let's assume that the tree structure of a simple FORTRAN subroutine with scalar definitions and assignments such as:

```
subroutine t
real a
a = 1
b = 2
end
```

looks like



We would like the tree to be rewritten as follows



where `ID[a]` represents an `ID` node with a symbol field of `a`. In other words, we would like to add definitions before the statement list for implicitly defined scalars.

The following grammar fragment could be used to recognize implicitly defined scalars and add a definition for it above in the tree after the last definition.

```
class FortranTranslate {
<<
   SORAST *defs;        // instance var tracking variable definitions
public:
   FortranTranslate() { defs=NULL; }
>>

routine
```

```
    :   #( DefSub ID slist )
    ;
slist
    :   #( SLIST
            ( v:vardef )*   <<defs=v; define(v);>>
            ( stat )*
        )
    ;
vardef
    :   #( Type ID )
    ;
stat:   #( Assign scalar:ID INT )
        <<if ( is-not-defined(scalar) )
            ast_append(defs,#(#[Type,"real"],#[ID,scalar->symbol()]));
        >>
    ;
}
```

where the notation "`#[args]`" is a node constructor and is translated to

`new SORAST(args)`

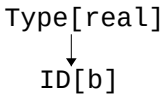
(or function `ast_node(args)` using the C interface). For example, in our case, you would define

```
class SORAST : public SORASTBase {
...
   SORAST(int token_type, char *sym);
...
};
```

The tree constructor

```
 #(#[Type,"real"], #[ID,id->symbol()])
```

builds a tree like



The `ast_append(defs,tree)` function call adds *tree* to the end of the sibling list pointed to by `defs`.

C++ Support Classes and Functions

SORCERER ASTs are defined by a class called `SORAST` that must be derived from `SORASTBase`, which inherits the following member functions (defined in `lib/PCCTAST.C`):

`addChild`
`void addChild(t)`. Add `t` to the list of children for `this`.

`append`
`void` **`append`**`(b)`. Add `b` to the end of the sibling list. Appending a `NULL` pointer is illegal.

`bottom`
`SORASTBase` **`*bottom`**`()`. Find the bottom of the child list (going straight "down").

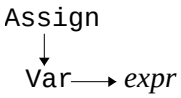
`cut_between`
`SORASTBase` **`*cut_between`**`(a,b)`.
Unlink all siblings between `a` and `b` and return a pointer to the first element of the sibling list that was unlinked. Basically, all this routine does is to make `b` a sibling of `a` and make sure that the tail of the sibling list, which was unlinked, does not point to `b`. The routine ensures that `a` and `b` are (perhaps indirectly) connected to start with. This routine returns `NULL` if either of `a` or `b` are `NULL` or if `a` and `b` are not connected.

`insert_after`
`void` **`insert_after`**`(b)`.
Add subtree `b` immediately after `this` as its sibling. If `b` is a sibling list at its top level, then the last sibling of `b` points to the previous right-sibling of `this`. If `b` is `NULL`, nothing is done. Inserting a `NULL` pointer has no effect.

`ast_find_all`
`SORASTBase` **`*ast_find_all`**`(u, cursor)`.
Find all occurrences of `u` in the tree pointed to by `this`.`cursor` (a pointer to a `SORAST` pointer) must be initialized to `this`. It eventually returns `NULL` when no more occurrences of `u` are found. This function is useful for iterating over every occurrence of a particular subtree. For example,

```
/* find all scalar assignments withing a statement list */
SORAST *scalar_assign = #( #[Assign], #[Var] );
PCCTS_AST *cursor = slist;
SORAST *p;
while ((p=(SORAST *)slist->ast_find_all(scalar_assign,&cursor)))
{
     /* perform an operation on 'p' */
}
```

where assignments are structured as



This function does not seem to work if you make nested calls to it; i.e., a loop containing an `ast_find_all()` that contains a call to another `find_all()`.

`tfree`

`void` **`tfree`**`()`. Recursively walk a tree `delete`'ing all the nodes in a depth-first order.

`make`

`static SORASTBase` **`*make`**`(root,child`$_1$`, ..., child`$_n$`,NULL)`. Create a tree with `root` as the root of the specified n children. If `root` is `NULL`, then a sibling list is constructed. If `child`$_i$ is a list of sibling, then `child`$_{i+1}$ will be attached to the last sibling of `child`$_i$. Any `NULL` `child`$_i$ results in `child`$_{i-1}$ being the last sibling. The root must not have children to begin with. A shorthand can be used in a description read by SORCERER:

`#(root, child`$_1$`, ..., child`$_n$`)`

`match`

`int` **`match`**`(u)`. Returns true if `this` and `u` are the same (the trees have the same tree structure and token types); else it returns false. If `u` is `NULL`, false is returned.

`nsiblings`

`int` **`nsiblings`**`().` Returns the number of siblings.

`ast_scan`

`int` **`ast_scan`**`(template, labelptr`$_1$`, ..., labelptr`$_n$`).` This function is analogous to `scanf`. It tries to match tree `this` against `template` and return the number of labels that were successfully mapped. The `template` is a string consisting of a SORCERER tree description with an optional set of node labels. For every label specified in template, the address of a SORAST pointer must be passed. Upon return from `ast_scan()`, the pointers will point to the requested nodes in `this`. This function can only be conveniently used from within a SORCERER description file and requires the use of the `#tokdefs` directive; it can be used in non-transform mode.

Consider the following example.

```
n = t->ast_scan("#( %1:A %2:B %3:C )", &x, &y, &z);
```

which SORCERER converts to before the call to `ast_scan()}`:

```
n = t->ast_scan("#( %1:7 %2:8 %3:9 )", &x, &y, &z);
```

where the token types of A, B, and C are 7, 8, and 9, respectively. After the call, pointers x, y, and z will point to the root, the first child and the second child, respectively; *n* will be 3.

sibling_index
SORASTBase ***sibling_index**(i). Return a pointer to the i^{th} sibling where the first sibling to the right is the index 2. An index of i==0, returns NULL and i==1 returns this.

tail
SORASTBase ***tail**(). Find the end of the sibling list and return a pointer to it.

Error Detection and Reporting

The following STreeParser member functions are called for the various possible parsing errors:

mismatched_token
mismatched_token(int looking_for, AST *found). The parser was looking for a particular token that was not matched.

mismatched_range
mismatched_range(int lower, int upper, AST *found). The parser was looking for a token in a range and did not find one.

missing_wildcard()
missing_wildcard(). The parser was looking for any tree element or subtree and found a NULL pointer.

no_viable_alt
no_viable_alt(char *which_rule, AST *current_root). The parser entered a rule for which no alternative's lookahead predicted that the input subtree would be matched.

sorcerer_panic
sorcerer_panic(char *err). This is called explicitly by you or by the support code when something terrible has happened.

Command Line Arguments

The basic form of a SORCERER command line is

`sor [`*`options`*`] file`$_1$`.sor ... file`$_n$`.sor`

where `file`$_1$ is the only one that may begin with a `#header` directive and *options* may be taken from:

`-CPP`
: Turn on C++ output mode. You must define a class around your grammar rules. An "`.h`" and "`.C`" file are created for the class definition as well as the normal "`.C`" file for the parser your grammar rules.

`-def-tokens`
: [*C++ mode only*] For each token referenced in the grammar, generate an `enum STokenType` definition in the class definition file. This should not be used with the `#tokdefs` directive, which specifies token types you've already defined.

`-def-tokens-file` *`file`*
: [*C mode only*] For each token referenced in the grammar, generate a `#define` in the specified file. This should not be used with the `#tokdefs` directive, which specifies token types you've already defined.

`-funcs` *`style`*
: Specify the style of the function headers and prototypes to be generated by SORCERER. *`style`* must be one of *`ANSI`* (the default), *KR*, or *`both`*..

`-inline`
: Only generate actions and functions for the given rules. Do not generate header information in the output. The usefulness of this option for anything but documentation has not been established.

`-out-dir style`
: Directory where all output files go; the default is "`.`".

`-prefix` *`s`*
: Prefix all globally visible symbols with *s* , including the error routines. Actions that call rules must prefix the function with *s* as well. This option can be used to link multiple SORCERER-generated parsers together by generating them with different prefixes. This is not useful in C++ mode.

`-proto-file` *`file`*
: Put all prototypes for rule functions in this file.

`-transform`
: Assume that a tree transformation will take place.

`-`
: Take input from `stdin` rather than a file.

C Programming Interface

Invocation of C Interface SORCERER Parsers

As with the C++ interface, the C interface requires that you specify the type of a tree node, how to navigate the tree, and the type of a node:

1. You must define tree node type `SORAST` in the `#header` action. SORAST must contain the fields in point **2** and **3**.
2. Your trees must be in child-sibling form; i.e., the trees must have fields `down` (points to the first child) and `right` (points to the next sibling).
3. Your tree must have a `token` field, which is used to distinguish between tree nodes. (*To be consistent with ANTLR and the SORCERER C++ interface, this field should be called* `type`*, but we have left it as* `token` *for backward compatibility reasons. The C interface of SORCERER is well enough established that changing it would invalidate too many grammars*).

A conforming tree using the C interface is the following:

```
typedef struct _node {
            struct _node *right, *down;
            int token;
            /* add fields you need here */
         } SORAST;
```

Table 28 on page 192 describes the files generated by SORCERER from a tree description in file(s) *`f1`*`.sor ... `*`fn`*`.sor.`

TABLE 28. Files Written by SORCERER for C Interface

File	Description
f1.C ... *fn*.C	Definition of rules specified in *f1*.sor ... *fn*.sor.
tokens.h	If SORCERER command-line option "-def-tokens-file *tokens*.h" is specified, this file contains a series of #defines for token types assigned by SORCERER for all node references in the grammar.

Using the C interface, SORCERER may also be used as a filter from stdin, in which case, the parser functions are written to stdout and no files are written (unless "-def-tokens-file *tokens*.h" is specified).

There are no global variables defined by SORCERER for the C interface, so multiple SORCERER tree parsers may easily be linked together; e.g., C files generated by SORCERER for different grammars can be compiled and linked together without fear of multiply defined symbols. This is accomplished by simulating a this pointer. The first argument of every parser function is a pointer to an STreeParser structure containing all variables needed to by the parser. Having a parser structure that is passed around from rule to rule implies that SORCERER parsers are "thread-safe" or "re-entrant."

You may add variables to the STreeParser structure by defining _PARSER_VARS; e.g.,

```
#define _PARSER_VARSint value; Sym *p;
```

Invoking a tree parser is a matter of creating an STreeParser variable, initializing it, and calling one of the parsing functions created from the rules. Parsing functions are called with the address of the parser variable, the address of the root of the tree you wish to walk, and any arguments you may have defined for that rule; e.g.,

```
main()
{
    MyTreeParser myparser;
    SORAST *some_tree = ... ;
    STreeParserInit(&myparser);
    rule(&myparser, &some_tree);
}
```

If *rule* had an argument and return value such as

```
rule[int i] > [float j] : ... ;
```

the invocation would change to

```
main()
{
    MyTreeParser myparser;
    SORAST *some_tree = ... ;
    float result;
    STreeParserInit(&myparser);
    result = rule(&myparser, &some_tree, 34);
}
```

C Types

The following types are used with C interface (see Figure on page 189):

`SORAST`
: You must provide the definition of this type, which represents the type of tree nodes that SORCERER is to walk or transform.

`SIntStack`
: A simple stack of integers.

`SList`
: A simple linked list of `void` pointers.

`SStack`
: A simple stack of `void` pointers.

`STreeParser`
: This type defines the variables needed by a tree walker. The address of one of these objects is passed to every parser function as the first argument.

C Files

The following files are used with the C interface.

`h/astlib.h, lib/astlib.c`
: Define the SORCERER AST library routines.

`lib/CASTBase.h`
: This is only used to compile the library routines. You can force them to compile with your `SORAST` definition if you want; that way, the order of the fields is irrelevant.

`h/config.h`
: Defines configuration information for the various platforms.

`h/sorcerer.h, lib/sorcerer.c`
: Define `STreeParser` and the support functions needed by SORCERER.

`h/sorlist.h,lib/sorlist.c`
: The `SList` manager.

`h/sintstack.h, lib/sintstack.c`
: The `SIntStack` manager.

`h/sstack.h, lib/sstack.c`
: The `SStack` manager.

`lib/errsupport.c`
: Defines error support code for SORCERER parser that you can link in, includes `mismatched_range()`, `missing_wildcard()`, `mismatched_token()`, `no_viable_alt()`, and `sorcerer_panic()`.

Combined Usage of ANTLR and SORCERER

To get SORCERER to walk an ANTLR-generated tree using the C interface is straightforward:

1. Define a `token` field in the AST definition for ANTLR in your ANTLR grammar file. For example,

   ```
   #define AST_FIELDSchar text[50]; int token;
   ```

2. Have your ANTLR parser construct trees as you normally would. Ensure that any token type that you will refer to in the SORCERER grammar has a label in the ANTLR grammar. For example,

   ```
   #token ASSIGN "="
   ```

3. In your SORCERER description, include the `AST` definition you gave to ANTLR and define `SORAST` to be `AST`. If you have used the ANTLR `-gh` option, you can simply include `stdpccts.h`. For example,

   ```
   #header <<
   #include "stdpccts.h" /* define AST and ANTLR token types */
   typedef AST SORAST;
   >>
   ```

4. A main program that calls both ANTLR and SORCERER routines looks like this:

   ```
   main()
   {
       AST *root=NULL;
       STreeParser tparser;
       STreeParserInit(&tparser);
       /* get the tree to walk with SORCERER */
       ANTLR(stat(&root), stdin);
       printf("input tree:"); lisp(root); printf("\n");
       /* walk the tree */
       start_symbol(&tparser, &root);
   }
   ```

C Support Libraries

Tree Library

The AST tree library, in `lib/astlib.c`, is used in transform mode to perform tree rewriting, although some of them may be useful in nontransform mode.

`ast_append`
: `void` **`ast_append`**`(a,b)`. Add `b` to the end of `a`'s sibling list. Appending a `NULL` pointer is illegal.

`*ast_bottom`
: `AST` **`*ast_bottom`**`(a)`. Find the bottom of `a`'s child list (going straight down).

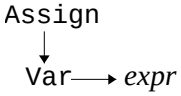
`*ast_cut_between`
: `AST` **`*ast_cut_between`**`(a,b).` Unlink all siblings between `a` and `b` and return a pointer to the first element of the sibling list that was unlinked. Basically, all this routine does is to make `a` point to `b` and make sure that the tail of the sibling list, which was unlinked, does not point to `b`. The routine ensures that `a` and `b` are (perhaps indirectly) connected to start with.

`ast_insert_after`
: `void` **`ast_insert_after`**`(a,b)`. Add subtree `b` immediately after `a` as its sibling. If `b` is a sibling list at its top level, then the last sibling of `b` points to the previous `right` sibling of `a`.Inserting a `NULL` pointer has no effect.

`*ast_find_all`
: `AST *`**`ast_find_all`**`(t, u, cursor).` Find all occurrences of `u` in `t`. cursor (a pointer to an AST pointer) must be initialized to `t`. It eventually returns `NULL` when no more occurrences of `u` are found. This function is useful for iterating over every occurrence of a particular subtree. For example,

```
/* find all scalar assignments withing a statement list */
AST *scalar_assign = #( #[Assign], #[Var] );
AST *cursor = statement_list;
while ((p=ast_find_all(statement_list, scalar_assign, &cursor)))
{
     /* perform an operation on 'p' */
}
```

where `ast_node()` (the function invoked by references to `#[]`) is assumed to take a single argument—a token type—and assignments are structured as:



This function does not seem to work if you make nested calls to it; i.e., a loop containing an `ast_find_all()` which contains another call to `ast_find_all()`.

`ast_free`

`void` **`ast_free`**`(t)`. Recursively walk `t`, `free`ing all the nodes in a depth-first order. This is perhaps not very useful because more than a `free()` may be required to properly destroy a node.

`*ast_make`

`AST` **`*ast_ make`**`(root, child`$_1$`, ..., child`$_n$`, NULL)`. Create a tree with `root` as the root of the specified *n* children. If root is `NULL`, then a sibling list is constructed. If child$_i$ is a list of sibling, then child$_{i+1}$ will be attached to the last sibling of child$_i$. Any `NULL` child$_i$ results in child$_{i-1}$ being the last sibling. The root must not have children to begin with.

A shorthand can be used in a description read by SORCERER:

```
#(root, child_1, ..., child_n) }
```

`ast_match`

`int` **`ast_match`**`(t,u)`. Returns true if `t` and `u` are the same (the trees have the same tree structure and `token` `SORAST` fields); else it returns false. If both trees are `NULL`, true is returned.

`ast_match_partial`

`int` **`ast_match_partial`**`(t,u)`. Returns true if `u` matches `t` (beginning at root), but `u` can be smaller than `t` (i.e., a subtree of `t`).

`ast_nsiblings`

`int` **`ast_nsiblings`**`(t)`. Returns the number of siblings of `t`.

`ast_scan`

`int` **`ast_scan`**`(template, t, labelptr`$_1$`, ..., labelptr`$_n$`)`. This function is analogous to `scanf`. It tries to match tree `t` against template and return the number of labels that were successfully mapped. The template is a string consisting of a SORCERER tree description with an optional set of node labels. For every label specified in template, the address of a `SORAST` pointer must be passed. Upon return from `ast_scan()`, the pointers will point to the requested nodes in `t`. This function can only be conveniently used from within a SORCERER description file and requires the use of the `#tokdefs` directive; it can be used in nontransform mode. Consider the following example.

```
n = ast_scan("#( %1:A %2:B %3:C )", t, &x, &y, &z);
```

which SORCERER converts to before the call to ast_scan()}:

```
n = ast_scan("#( %1:7 %2:8 %3:9 )", t, &x, &y, &z);
```

where the token types of `A`, `B`, and `C` are 7, 8, and 9, respectively. After the call, pointers `x`, `y`, and `z` will point to the root, the first child and the second child, respectively; `n` will be 3.

The order of label specification in `template` is not significant.

`*ast_sibling_index`

`AST` **`*ast_sibling_index`**`(t,i)`. Return a pointer to the i^{th} sibling where the sibling to the right of `t` is `i==2`. A index of `i==1`, returns `t`.

`*ast_tail`

`AST` **`*ast_tail`**`(a).` Find the end of `a`'s sibling list and return a pointer to it.

`*ast_to_slist`

`SList` **`*ast_to_slist`**`(t).` Return a list containing the siblings of `t`. This can be useful for walking a list of subtrees without having to parse it. For example,

```
<<SList *stats;>>
slist
     : ( stat )*  <<stats = ast_to_list(_root);>>
   ;
```

where `_root` is the argument passed to every function that points to the input tree node. In this case, the variable `stats` will be a list with an element (`SORAST *`) for each statement in the statement list.

`*slist_to_ast`

`AST` **`*slist_to_ast`**`(list).` Return a tree composed of the elements of `list` with a sibling for each element in list.

List Library

The SORCERER list library, `lib/sorlist.c`, is a simple linked-list manager that makes lists of pointers. The pointer for a new list must be initialized to `NULL` as any non-empty list has a sentinel node whose `elem` field pointer is really a pointer to the last element.

`*slist_iterate`

`void` **`*slist_iterate`**`(list,cursor)`. Iterate over a list of elements in `list`; return a pointer to a new element in list upon every call and `NULL` when no more are left. It can be used like this:

```
cursor = mylist;
while ( (p=slist_iterate(mylist,\&cursor)) ) {
     /* place with element p */
}
```

Lists can also be traversed with

```
SList *p;
for (p = list->next; p!=NULL; p=p->next)
{
     /* process (MyElement *)p->elem */
}
```

slist_add

`void` **`slist_add`**`(list, e).` Add element `e` to `list` . Any non-empty list has a sentinel node whose `elem` pointer is really a pointer to the last element. Elements are appended to the list. For example,

```
SList *Strings = NULL;
list_add(&Strings, "hi");
list_add(&Strings, "there");
```

slist_free

`void` **`slist_free`**`(list)`. Free a list (frees all of the nodes used to hold pointers to actual elements). It does not effect the elements tracked by the list.

Stack Library

The SORCERER stack library, `lib/sstack.c`, is a simple linked-list style stack of pointers. There is no sentinel node, and a pointer for a stack must be initialized to `NULL` initially.

sstack_push

`void` **`sstack_push`**`(st,e)`. Push element `e` on stack `st` where `e` can be a pointer to any object. For example,

```
SStack *st = NULL;
sstack_push(&st, "I push");
sstack_push(&st, "therefore, I'm a stack");
```

***sstack_pop**

`void` **`*sstack_pop`**`(st).` Pop the top element off of stack `st` and return it. For example,

```
SStack *st = NULL;
char *s;
sstack_push(&st, "over the edge");
s = sstack_pop(&st);
printf("%s\n", s);
```

should print "`over the edge`".

Integer Stack Library

The SORCERER `SIntStack` library, `lib/sintstack.c`, is a simple array-based stack of integers. Stacks of integers are common (such as saving the scope/level of programming language); hence, we have constructed a stack which is much easier to use and faster than the normal stack routines. Overflow and underflow conditions are trapped.

***sint_newstack**

`SIntStack` **`*sint_newstack`**`(size).` Make a new stack with a maximum depth of `size` and return a pointer to it.

sint_freestack

`void` **`sint_freestack`**`(SIntStack *st).` Destroys a stack created by `sint_newstack()`.

sint_push

`void` **`sint_push`**`(st, i).` Push integer `i` onto stack `st`. For example,

```
SIntStack *st = sint_newstack(100);
sint_push(st, 42);
sint_push(st, 3);
```

sint_pop

`int` **`sint_pop`**`(st)`. Pop the top integer off the stack and return it.

```
SIntStack *st = sint_newstack(10);
sint_push(st, 3);
printf("%d\n", sint_pop(st));
```

would print "3".

sint_stacksize

`int` **`sint_stacksize`**`(st)`. Returns the number of integers currently on the stack.

sint_stackreset

`void` **`sint_stackreset`**`(st)`. Remove all integers from the stack.

sint_stackempty

`int` **`sint_stackempty`**`(st)`. Returns true if there are no integers on stack `st`.

sint_top

`int` **`sint_top`**`(st)`. Return the top integer on the stack without affecting the state of the stack.

5 ANTLR Warning and Error Messages

This chapter describes error and warning messages that can be generated by ANTLR. They are organised by categories of ANTLR functionality.The actual messages displayed by ANTLR are shown in bold and are followed by a brief description.

Token and Lexical Class Definition Messages

Warnings

redefinition of token *t*; ignored
Token *t* was previously seen in either a `#token` directive or rule.

token label has no associated rexpr: *t*
A token type is associated with token t, but no regular expression has been provided; i.e., no input character sequence will result in this token type.

token name *t* and rexpr *re* already defined; ignored
t and *re* were previously attached to other regular expressions or tokens, respectively. For example:

```
#token T “foo”
#token U
#token U “foo”
```

no regular expressions found in grammar

You did not specify even one input character sequence to combine into a token. This is an uninteresting grammar.

lexclass name conflicts with token/errclass label 't'

A lexclass definition tried to reuse a previously defined symbol *t*.

Errors

redefinition of token *t*; ignored

Another definition of *t* was seen previously.

action cannot be attached to a token name *(t)*; ignored

Actions can only be attached to regular expressions. If only a token label is specified, an action is meaningless.

redefinition of action for *re;* ignored

An action has already been attached to regular expression *re.*

#token definition '*t*' not allowed with #tokdefs; ignored

When the `#tokdefs` directive is used, all tokens are assumed to be defined inside the specified file. New token labels may not be introduced in the grammar specificatiion, however, regular expressions may be attached to the token labels.

implicit token definition not allowed with #tokdefs

When the `#tokdefs` directive is used, all tokens are assumed to be defined inside the specified file. New token labels may not be introduced in the grammar specification, however, regular expressions may be attached to the token labels via the `#token` directive.

#token requires at least token name or rexpr

A lone `#token` directive is meaningless (even if an action is given).

redefinition of action for expr *re*; ignored

Regular expression *re* already has an attached action.

redefinition of expr *re*; ignored

Regular expression *re* has already been defined.

Grammatical Messages

Warnings

rule *r* **not defined**

Rule *r* was referenced in your grammar, but you have not defined it.

alts *i* **and** *j* **of** *decision-type* **ambiguous upon** *k-seqs*

The specified alternatives (counting from 1) of the decision cannot be distinguished. At least one input sequence of length *k* could be matched by both alternatives. For example, the following rule is ambiguous at *k*=1 upon tokens `{A,B}`:

```
a : A B | A C ;
```

It is not ambiguous at *k*=2. The following rule is ambiguous upon 2-sequence `AB` (or, as ANTLR would print it out: `{A},{B}`):

```
a : A B C | A B ;
```

This is only a warning, but some decisions are inherently ambiguous like the proverbial dangling else clause:

```
stat : IF expr THEN stat { ELSE stat } | ...;
```

The optional clause is ambiguous upon `ELSE`.

optional/exit path and alt(s) of *decision-type* **ambiguous upon** *k-seqs*

The same interpretation applies to this message as for the previous message. The difference lies in that no alternative number can be associated with the exit path of a loop. For example,

```
a : ( A B )* A C ;
```

is ambiguous upon `A` for *k*=1, but unambiguous at *k*=2.

Errors

infinite left-recursion to rule *a* **from rule** *b*

Without consuming a token of input, the parser may return to a previously visited state. Naturally, your parser may never terminate. For example,

```
a : A | b ;
b : c B ;
c : a | C ;
```

All rules in this grammar can cause infinite recursion.

only one grammar class allowed in this release

Only one grammar class may be specified. To include rules from multiple files in one class, repeat the class header in each file.

file 1: `class T {` *some rules* `}`

file 2: `class T {` *more rules* `}`

Implementation Messages

action buffer overflow; size *n*

One of your actions was too long for an internal buffer. Increase `ZZLEXBUFSIZE` in `pccts/antlr/generic.h` and recompile ANTLR. Or, break up your action into two actions.

predicate buffer overflow; size *n*

One of your semantic predicates was too long for an internal buffer. Increase `ZZLEXBUFSIZE` in `pccts/antlr/generic.h` and recompile ANTLR. Or, break up your predicate into two actions.

parameter buffer overflow; size *n*

One of your actions was too long for an internal buffer. Increase `ZZLEXBUFSIZE` in `pccts/antlr/generic.h` and recompile ANTLR. Or, break up your action into two actions.

#$%%*&@# internal error: *error*

[**complain to nearest government official or send hate-mail to parrt@parr-research.com; please pray to the ‘‘bug’’ gods that there is a**

trival fix.]
Something bad happened. Send in a bug report.

hit analysis resource limit while analyzing alts *i* **and** *j* **of** *decision-type*
ANTLR was busily computing the lookahead sets needed construct your parser, but ran out of resources (you specify a resource cap with `-rl` command line option). For large grammars, this indicates what decision was taking so long to handle. You can simplify the decision, reduce the size of the grammar or lookahead, or use syntactic predicates.

out of memory while analyzing alts *i* **and** *j* **of** *decision-type*
ANTLR tried to call `malloc()`, which failed.

Action, Attribute, and Rule Argument Messages

Warnings

$*t* not parameter, return value, or element label
You referenced $*t* within an action, but it is not a parameter or return value of the enclosing rule nor is it a label on a rule or token.

invalid parameter/return value: '*param-or-ret-val-definition*'
Your parameter or return value definition was poorly formed C or C++; e.g., missing argument name.

rule *r* accepts no parameter(s)
You specified parameters to *r* in some rule of your grammar, but r does not accept parameters.

rule *r* requires parameter(s)
You specified no parameters to *r* in some rule of your grammar, but *r* accepts at least one parameter.

rule *r* yields no return value(s)
You specified a return value assignment from *r* in some rule of your grammar, but *r* does not return any values.

rule *r* returns a value(s)

You specified no return value assignment from *r* in some rule of your grammar, but *r* returns at least one value.

Errors

$$ use invalid in C++ mode

$$ can only be used in C mode. C++ mode does not have attributes. Use return arguments or return values.

$[] use invalid in C++ mode

`$[]` can only be used in C mode. C++ mode does not have attributes.

cannot mix old-style $i with new-style labels

You cannot reference $*i* for some integer *i* in your actions and $*label* for some label attached to a rule or token reference.

one or more $i in action(s) refer to non-token elements

In C++ mode, $*i* for some integer *i* variables do not exist for rules since attributes are not defined. Use return arguments or return values.

cannot mix with new-style labels with old-style $i

You referenced $*label* at this point, but previously referenced $*i* for some integer *i* in your actions.

label definition clashes with token/tokclass definition: '*t*'

You attached a label to a rule or token that is already defined as a token or token class.

label definition clashes with rule definition: '*t*'

You attached a label to a rule or token that is already defined as a rule.

label definitions must be unique per rule: '*t*'

You attached a label to a rule or token that is already defined as a label within that rule.

Command-Line Option Messages

Warnings

#parser meta-op incompatible with -CC; ignored
`#parser` directive can only be used in C mode. Use a class definition in C++ mode.

#parser meta-op incompatible with '-gp prefix'; '-gp' ignored
`#parser` directive should be used instead of `-gp`, but we left it in for backward compatibility reasons. Use a class definition in C++ mode.

-gk option could cause trouble for <<...>>? predicates
The `-gk` option delays the fetching of lookahead, hence, predicates that refer to future lookahead values may be referring to unfetched values.

-gk incompatible with semantic predicate usage; -gk turned off
See previous.

-gk conflicts with -pr; -gk turned off
See previous.

analysis resource limit (# of tree nodes) must be greater than 0
You have not specified a value or specified a negative number for the `-rl` option.

must have at least one token of look-ahead (setting to 1)
You have not specified a value or specified a negative number for the `-k` option.

must have compressed lookahead >= full LL(k) lookahead (setting -ck to -k)
You have not specified a value or specified a negative number for the `-ck` option.

Errors

class meta-op used without C++ option
You cannot give grammar class definitions without the `-CC` option.

Token and Error Class Messages

default errclass for '*t*' would conflict with token/errclass/tokclass

ANTLR cannot create an error class for rule `t` because the error class would conflict with a known symbol. Default error class names are created from rule names by capitalizing the first letter of the rule name.

errclass name conflicts with regular expression '*t*'

The specified error class conflicts with a previously-defined regular expression.

redefinition of errclass or conflict w/token or tokclass '*t*'; ignored

You have defined an error class with the same name as a previously-defined symbol.

undefined rule '*t*' referenced in errclass '*t*'; ignored

You referenced a rule in your error class that does not have a definition.

self-referential error class '*t*'; ignored

Your error class refers to itself directly or indirectly (through another error class).

undefined token '*t*' referenced in errclass '*t*'; ignored

Your error class refers to a token that has not been defined.

redefinition of tokclass or conflict w/token '*t*'; ignored

Your token class name conflicts with a previously-defined token label.

redefinition of #tokclass '*t*' to #token not allowed; ignored

You have redefined token class *t*.

Predicate Messages

Warnings

alt *`i`* of *`decision-type`* has no predicate to resolve ambiguity

With options `-w2` and `-prc on` ANTLR warns you that one of the lookahead sequences predicts more than one alternative and that you have specified a

predicate to resolve the ambiguity for one of the alternatives, but not the other. For example,

```
a : <<f(LT(1))>>? ID | ID ;
```

will result in

stdin, line 2: warning: alt 2 of the rule itself has no predicate to resolve ambiguity

cannot compute context of predicate in front of (..)? block

You used option `-prc on` and used a `<<...>>?` in front of a `(...)?` for which lookahead cannot be computed. For example,

```
a : <<blah>>? (UGH)? | ICK ;
```

(...)? as only alternative of block is unnecessary

You specified something like:

```
a : (foo)? ;
```

which is the same as

```
a : foo ;
```

Errors

(...)? predicate must be first element of production

You specified a syntactic predicate with a grammar element in front of it. All syntactic predicates must be the first element of a production in order to predict it.

Exception Handling Messages

duplicate exception handler for label '*t*'

You specified more than one handler for a single label *t*.

unknown label in exception handler: '*t*'

You specified a handler for an unknown label *t*.

6 SORCERER Warning and Error Messages

This chapter describes error and warning messages that can be generated by SORCERER. We have broken the descriptions into categories rather than grouping them into error and warning sections. The actual messages displayed by SORCERER are shown in bold and are followed by a brief description.

Syntax Messages

Warnings

unknown meta-op: *m*
Meta-operation #*m* is not valid.

missing #header statement
You forgot a `#header` statement in C mode.

extra #header statement
You have more than one `#header` statement.

extra #tokdef statement
You have more than one `#tokdef` statement.

Errors

Missing /*; found dangling */
You forgot to start your comment with a `/*`.

Missing <<; found dangling >>"
You forgot to start your action or predicate with a <<.

Missing /*; found dangling */ in action
You forgot to start your comment with a `/*` in an action.

missing class definition for trailing '}'
The end of the class definition was seen, but the header was missing.

rule definition clashes with *r* **definition: '***t***'**
You have defined a rule that has the same name as a previously defined symbol such as a label.

rule multiply defined: '*r***'**
You have defined a rule with this name already.

label definition clashes with *t* **definition: '***u***'**
Label *u* classes with a previously defined symbol such as a rule name.

cannot label this grammar construct
You can only label rule and token references (including the wildcard).

redefinition of token *t***; ignored**
You have already defined token *t*.

token definition clashes with *symbol-type* **definition: '***t***'**
The definition of token *t* clashes with a predefined symbol.

Action Messages

Warnings

eoln found in string
You did not terminate your string on the same line as your started it.

eoln found in string (in user action)
You did not terminate your string on the same line as your started it.

eoln found in char literal (in user action)
You did not terminate your character literal on the same line as your started it.

Errors

Reference variable clashes with *t*: '*v*'
You have defined an @-variable that clashes with a previously-defined symbol such as a rule name.

#id used in action outside of rule; ignored
`#id` is only valid as the result or input tree of a rule. Placing a reference to `#id` outside of a rule makes no sense.

Grammatical Messages

infinite recursion from rule *a* to rule *b*
Rule *a* can reach rule *b* without having moved anywhere in the tree. Naturally, infinite-recursion can result.

rule not defined: *'r'*
You have referenced rule *r*, but not defined it in your grammar.

alts *i* and *j* of (...) nondeterministic upon *tree-node*
Alternatives *i* and *j* both begin with the same root node or sibling node.

(...)? predicate in block with one alternative; will generate bad code

A syntactic predicate in a rule or subrule with only one alternative makes no sense because the tree-walker will not have to guess which alternative to choose.

predicate not at beginning of alternative; ignored

A predicate not at the left-edge of a production cannot aid in the prediction of that alternative.

Implementation Messages

action buffer overflow; size *n*

One of your actions was too long for an internal buffer. Increase `ZZLEXBUFSIZE` in `sorcerer/sor.g` and recompile SORCERER. Or break up your action into two actions.

parameter buffer overflow; size *n*

One of your actions was too long for an internal buffer. Increase `ZZLEXBUFSIZE` in `sorcerer/sor.g` and recompile ANTLR. Or break up your action into two actions.

Command-Line Option Messages

Warnings

-def-tokens valid with C++ interface only; ignored
-def-tokens-file not valid with C++ interface; ignored

C++ mode SORCERER generates a list of token definitions (unless `#tokdef`) is used in the output class definition file.

-def-tokens-file conflicts with -inline; ignored

Cannot generate a token definition file if the output of SORCERER will be inline.

don’t you want filename with that -def-tokens-file?; ignored

You forgot to specify a file name.

-prefix conflicts with C++ interface; ignored
C++ does not need to prefix symbols with a prefix because of the information hiding capabilities of C++.

don't you want string with that -prefix?; ignored
You forgot to specify a string.

don't know how to generate '*t*' style functions; assuming ANSI
You gave an invalid or missing `-funcs` argument

-proto-file not valid with C++ interface; ignored
Prototypes are placed in the parser class definition file in C++ mode.

don't you want filename with that -proto-file?; ignored
You forgot a file name.

'-' (stdin) ignored as files were specified first
You specified both inline mode and some grammar files.

'-' (stdin) cannot be used with C++ interface; ignored
C++ requires a bunch of output that cannot be just concatenated together.

-inline conflicts with -def-tokens; ignored
Cannot generate token definitions when output is inline.

-inline conflicts with C++ interface; ignored
C++ requires a bunch of output that cannot be just concatenated together.

tokens file not generated; it conflicts with use of #tokdefs
If a token definition file is used to specify token type values, you cannot write another version of this file out.

Can't open prototype file '*f*'; ignored
For some reason, file *f* could not be open for writing.

invalid parameter/return value: '*r*'
You provided a poorly formed C/C++ parameter or return value.

Errors

-funcs option makes no sense in C++ mode; ignored
Functions are always prototyped in C++.

class meta-op used without C++ option
You had a parser class definition, but forgot to turn on C++ option.

file '*f*' ignored as '-' (stdin option) was specified first
You specified `stdin` mode and then a file name.

Token Definition File Messages

cannot write token definition file *f*
For some reason, file *f* could not be open for writing.

cannot open token defs file '*f*'
File *f* could not be found.

range operator is illegal without #tokdefs directive
In order to use the range operator, you must tell SORCERER what the token type values are for all your tokens. The only to do this is to use `#tokdef`.

implicit token definition of '*t*' not allowed with #tokdefs
Token *t* was not defined in the token definition file. Its token type is therefore unknown.

7 Templates and Quick Reference Guide

In this chapter, we provide a collection of examples and summaries that illustrate the major features of ANTLR and DLG; we include an example linking ANTLR and SORCERER. Much of the code is taken from the `testcpp` directory in the PCCTS distribution.

Templates

This section provides templates for using ANTLR in C++ mode when not using trees, when using trees and when using ANTLR with SORCERER.

Basic ANTLR Template

```
#header <<
// put things here that need to be defined in all output files
>>

<<
#include "DLGLexer.h"
typedef ANTLRCommonToken ANTLRToken;
#include "PBlackBox.h"

class MyVersionOfParser : public Parser {
// override triggers declared in actual parser class def below
```

```
public:
   MyVersionOfParser(ANTLRTokenBuffer *input) : Parser(input)
   {
      printf("start parse\n");
   }
   ~MyVersionOfParser()
   {
      printf("end parse\n");
   }
};

int main()
{
   ParserBlackBox<DLGLexer, MyVersionOfParser, ANTLRToken> p(stdin);
   p.parser()->startrule();
   return 0;
}
>>

/* Ignore whitespace */
#token "[\ \t]+"    <<skip();>>
#token "\n"         <<skip(); newline();>>

class Parser {

<< Define member functions (triggers) and variables here;
   or, in subclass above.
>>

startrule
   :  alternative 1
   |  alternative 2
   |  ...
   ;

}

// Sample tokens that normally appear at the end of a grammar
#token INT   "[0-9]+"
#token ID "[a-zA-Z_][a-zA-Z0-9_]*"
```

Using ANTLR With ASTs

```
<<
typedef ANTLRCommonToken ANTLRToken;
#include "DLGLexer.h"
#include "PBlackBox.h"

// ASTs are simply smart pointers to input token objects
class AST : public ASTBase {
   ANTLRTokenPtr token;
public:
   AST(ANTLRTokenPtr t) { token = t; }
   void preorder_action() {    // what to print out at each node
       char *s = token->getText();
       printf(" %s", s);
   }
   PCCTS_AST *shallowCopy() { define if you use dup or deepCopy }
};

int main()
{
   ParserBlackBox<DLGLexer, Parser, ANTLRToken> p(stdin);
   ASTBase *root = NULL;
   p.parser()->start(&root);   // parse and build trees
   root->preorder();           // print out the tree in LISP form
   printf("\n");
   root->destroy();            // delete the nodes
   return 0;
}
>>

// token definitions

class Parser {

start
   :  ...
   ;

}
```

Using ANTLR With SORCERER

This section contains a number of files representing a complete ANTLR/SORCERER application that reads in expressions and generates a simple stack code.

File: lang.g

```
<<
typedef ANTLRCommonToken ANTLRToken;
#include "AST.h"
>>

#token "[\ \t]+"       <<skip();>>
#token "\n"            <<newline(); skip();>>
#token ASSIGN    "="
#token ADD       "\+"
#token MULT      "\*"

class SimpleParser {

stat:ID "="^ expr ";"!
   ;

expr:mop_expr ( "\+"^ mop_expr )*
   ;

mop_expr
   :  atom ( "\*"^ atom )*
   ;

atom:ID
   |  INT
   ;

}
#token ID    "[a-z]+"
#token INT   "[0-9]+"
```

File: AST.h

```
#include "ASTBase.h"
#include "AToken.h"

#define AtomSize    20
```

```
#include "ATokPtr.h"

class AST : public ASTBase {
protected:
   char text[AtomSize+1];
   int _type;
public:
   AST(ANTLRTokenPtr t)
      { _type = t->getType(); strcpy(text, t->getText()); }
   AST()                    { _type = 0; }
   int type()               { return _type; }
   char *getText()          { return text; }
   void preorder_action()   { printf(" %s", text); }
};

typedef AST SORAST;     // define the type of a SORCERER tree
```

File: gen.sor

```
#header <<
#include "tokens.h"
#include "AST.h"
>>

class SimpleTreeParser {

gen_stat
   :   #( ASSIGN t:ID gen_expr )
      <<printf("\tstore %s\n", t->getText());>>
   ;

gen_expr
   :   #( ADD gen_expr gen_expr )      <<printf("\tadd\n");>>
   |   #( MULT gen_expr gen_expr )     <<printf("\tmult\n");>>
   |   t:ID          <<printf("\tpush %s\n", t->getText());>>
   |   t:INT         <<printf("\tpush %s\n", t->getText());>>
   ;

}
```

File: main.cpp

```
#include "tokens.h"
#include "SimpleParser.h"               // define the parser
```

```
typedef ANTLRCommonToken ANTLRToken;
#include "SimpleTreeParser.h"            // define the tree walker
#include "DLGLexer.h"                    // define the lexer
#include "PBlackBox.h"

main()
{
   ParserBlackBox<DLGLexer, SimpleParser, ANTLRToken> lang(stdin);
   AST *root=NULL, *result;
   SimpleTreeParser tparser;

   lang.parser()->stat((ASTBase **)&root); // get the tree to walk
// printf("input tree:"); root->preorder(); printf("\n");
   tparser.gen_stat((SORASTBase **)&root); // walk tree
}
```

File: makefile

This makefile was modified from the original created with the `genmk` command indicated in the makefile comment.

```
#
# PCCTS makefile for: lang.g
#
# Created from:  /circle/s13/parrt/PCCTS/bin/genmk -CC -class \
#                SimpleParser -project t4 -trees lang.g
#
# PCCTS release 1.33
# Project: t
# C++ output
# DLG scanner
# ANTLR-defined token types
#
TOKENS = tokens.h
#
# The following filenames must be consistent with ANTLR/DLG flags
DLG_FILE = parser.dlg
ERR = err
HDR_FILE =
SCAN = DLGLexer
PCCTS = /usr/local/pccts
ANTLR_H = $(PCCTS)/h
SOR_H = ../../h
SOR_LIB = ../../lib
BIN = $(PCCTS)/bin
ANTLR = $(BIN)/antlr
DLG = $(BIN)/dlg
```

```
SOR = ../../sor
CFLAGS = -I. -I$(ANTLR_H) -I$(SOR_H) -I$(SOR_LIB) -g
AFLAGS = -CC -gt
DFLAGS = -C2 -i -CC
GRM = lang.g
SRC = lang.cpp main.cpp test4.cpp $(SOR_LIB)/STreeParser.cpp \
 SimpleParser.cpp \
 SimpleTreeParser.cpp \
 $(ANTLR_H)/AParser.cpp $(ANTLR_H)/DLexerBase.cpp \
 $(ANTLR_H)/ASTBase.cpp $(ANTLR_H)/PCCTSAST.cpp \
 $(ANTLR_H)/ATokenBuffer.cpp $(SCAN).cpp
OBJ = lang.o main.o test4.o STreeParser.o \
 SimpleParser.o \
 SimpleTreeParser.o \
 AParser.o DLexerBase.o \
 ASTBase.o PCCTSAST.o \
 ATokenBuffer.o $(SCAN).o
ANTLR_SPAWN = lang.cpp SimpleParser.cpp \
 SimpleParser.h $(DLG_FILE) $(TOKENS)
DLG_SPAWN = $(SCAN).cpp $(SCAN).h
CCC=g++
CC=$(CCC)

t : $(OBJ) $(SRC)
   $(CCC) -o t4 $(CFLAGS) $(OBJ)

main.o : main.cpp SimpleTreeParser.h SimpleParser.h
   $(CCC) -c $(CFLAGS) main.cpp

lang.o : $(TOKENS) $(SCAN).h lang.cpp
   $(CCC) -c $(CFLAGS) -o lang.o lang.cpp

SimpleParser.o : $(TOKENS) $(SCAN).h SimpleParser.cpp SimpleParser.h
   $(CCC) -c $(CFLAGS) -o SimpleParser.o SimpleParser.cpp

SimpleTreeParser.o : $(TOKENS) $(SCAN).h SimpleTreeParser.cpp tokens.h
   $(CCC) -c $(CFLAGS) SimpleTreeParser.cpp

test4.cpp SimpleTreeParser.h SimpleTreeParser.cpp : test4.sor
   $(SOR) -CPP test4.sor

test4.o : test4.cpp
   $(CCC) -c $(CFLAGS) test4.cpp

STreeParser.o : $(SOR_LIB)/STreeParser.cpp
   $(CCC) -o STreeParser.o -c $(CFLAGS) $(SOR_LIB)/STreeParser.cpp
```

```
$(SCAN).o : $(SCAN).cpp $(TOKENS)
   $(CCC) -c $(CFLAGS) -o $(SCAN).o $(SCAN).cpp

$(ANTLR_SPAWN) : $(GRM)
   $(ANTLR) $(AFLAGS) $(GRM)

$(DLG_SPAWN) : $(DLG_FILE)
   $(DLG) $(DFLAGS) $(DLG_FILE)

AParser.o : $(ANTLR_H)/AParser.cpp
   $(CCC) -c $(CFLAGS) -o AParser.o $(ANTLR_H)/AParser.cpp

ATokenBuffer.o : $(ANTLR_H)/ATokenBuffer.cpp
   $(CCC) -c $(CFLAGS) -o ATokenBuffer.o $(ANTLR_H)/ATokenBuffer.cpp

DLexerBase.o : $(ANTLR_H)/DLexerBase.cpp
   $(CCC) -c $(CFLAGS) -o DLexerBase.o $(ANTLR_H)/DLexerBase.cpp

ASTBase.o : $(ANTLR_H)/ASTBase.cpp
   $(CCC) -c $(CFLAGS) -o ASTBase.o $(ANTLR_H)/ASTBase.cpp

PCCTSAST.o : $(ANTLR_H)/PCCTSAST.cpp
   $(CCC) -c $(CFLAGS) -o PCCTSAST.o $(ANTLR_H)/PCCTSAST.cpp

clean:
   rm -f *.o core t4

scrub:
   rm -f *.o core t4 $(ANTLR_SPAWN) $(DLG_SPAWN) test4.cpp
```

Defining Your Own Tokens

In an action before the grammar, you may specify or include the definition of `ANTLRToken` rather than use the predefined `ANTLRCommonToken`.

```
class ANTLRToken : public ANTLRRefCountToken {
protected:
   ANTLRTokenType _type;    // what's the token type of the token object
   int _line;               // track line info for errors
   ANTLRChar _text[30];     // hold the text of the input token

public:
   ANTLRToken(ANTLRTokenType t, ANTLRChar *s)
```

```
      :   ANTLRRefCountToken(t,s)
      { setType(t); _line = 0; setText(s); }

   // Your derived class MUST have a blank constructor.
   ANTLRToken()
      { setType((ANTLRTokenType)0); _line = 0; setText(""); }

   // how to access the token type and line number stuff
   ANTLRTokenType getType() { return _type; }
   void setType(ANTLRTokenType t) { _type = t; }
   virtual int getLine() { return _line; }
   void setLine(int line) { _line = line; }

   ANTLRChar *getText() { return _text; }
   void setText(ANTLRChar *s) { strncpy(_text, s, 30); }

   // WARNING: you must return a stream of distinct tokens
   // This function will disappear when we can use templates
   virtual ANTLRAbstractToken *makeToken(    ANTLRTokenType tt,
                                             ANTLRChar *txt,
                                             int line)
   {
      ANTLRAbstractToken *t = new ANTLRToken(tt,txt);
      t->setLine(line);
      return t;
   }
};
```

Defining Your Own Scanner

To use your own scanner with an ANTLR grammar, you must define a subclass of `ANTLRTokenStream` and then include that definition in the grammar file within an action (instead of the usual "`#include "DLGLexer.h"`". Here is a sample lexer definition:

```
#include "config.h"
#include "tokens.h"     // let's say it defines DIGIT, PUNCT
typedef ANTLRCommonToken ANTLRToken;
#include ATOKENBUFFER_H
#include <ctype.h>

class MyLexer : public ANTLRTokenStream {
private:
   int c;
public:
```

```
   MyLexer() { c = getchar(); }
   virtual ANTLRAbstractToken *getToken()
       {
           char buf[2];
           buf[0] = c;
           if ( isdigit(c) ) return new ANTLRToken(DIGIT,buf);
           if ( ispunct(c) ) return new ANTLRToken(PUNCT,buf);
           return NULL;
       }
};
```

The genmk Program

The genmk program is provided so that most makefiles for ANTLR can be automatically generated. To begin most projects, you only need provide a parser class name, decide whether you are going to build trees, decide on the name of the project (the executable), and decide on the grammar file name. For example, the most common `genmk` line is:

`genmk -CC -class` *MyParser* `-project` *myproj* *file*`.g > makefile`

This line creates a makefile that uses ANTLR with the C++ interface, with parser class *MyParser*, and with a resulting executable called *myproj*; your grammar file is *file*`.g`. The following lines in the makefile need to be modified to suit your environment:

```
 PCCTS = .    # normally something like /usr/local/src/pccts
#CCC=g++  # uncomment and define to your C++ compiler
```

If you will be using trees, use the `-tree` option with `genmk` also.

Rules

Rule With Multiple Alternatives

```
rule
   :  alternative1
   |  alternative2
   ...
   |  alternativen
   ;
```

Rule With Arguments and Return Values

rule[arg_1,...,arg_n] > [$retval_1$,...,$retval_m$] : ... ;

where the arguments and return values are well-formed C++ definitions such as "`int i`" or "`char *p`".

EBNF Constructs

Subrule

(alternative$_1$ | alternative$_2$ ... | alternative$_n$)

Optional Subrule

{ alternative$_1$ | alternative$_2$ ... | alternative$_n$ }

Zero Or More Subrule

(alternative$_1$ | alternative$_2$ ... | alternative$_n$)*

One Or More Subrule

(alternative$_1$ | alternative$_2$ ... | alternative$_n$)+

Alternative Elements

Token References

1. Token identifiers. Identifiers begin with an uppercase letter; e.g., `ID`, `INT`.
2. Regular expressions enclosed in double-quotes; e.g. `"[a-z]+"`, `"begin"`.
3. Token class references; e.g.,
   ```
   #tokclass Operators { Plus Minus }
   e : e2 ( Operators e2 )* ;
   ```

4. Token type ranges—two tokens separated by two dots;
   `FirstOperator .. LastOperator`
5. "Not" operator—match any token except end-of-file or *TOKEN*; e.g., ~*TOKEN*.

Rule References

Rule references invoke other rules possibly with arguments and return values. For example, "`b[34] > [i]`" invokes rule `b` with one argument, `34`, and stores the return value of `b` in some variable `i`.

Labels

Rule and token references may be labeled with an identifier (either upper or lower case); e.g.,

```
rule : label:ID ;
```

Labels are referenced in user actions as $*label*.

Labels are useful for:

1. Accessing the token object associated with the referenced token.
2. Attaching a parser exception handler to the invocation of either a rule or token reference.

Actions

Actions are enclosed in double angle-brackets; e.g., <<*action*>>. If the first element of any subrule or rule is an action, that action is an init-action; e.g.,

```
rule : <<int i=3;>> id:ID <<i=atoi($id->getText());>> ;
```

An action placed immediately after the terminating ';' on a rule is considered a fail-action. Fail-actions are executed upon syntax error before ANTLR prints out a message and before the rule is exited.

Predicates

Semantic Predicates

Semantic predicates are actions followed by '?'. For example,

```
typename : <<isTypeName(LT(1)->getText())>>? ID ;
```

The `LT(1)` is the trigger to ANTLR that only one symbol of context is needed for this predicate when the `-prc` command line option is used.

Syntactic Predicates

Syntactic predicates are subrules followed by a '?'. For example,

```
statement
   :  ( decl )?
   |  expr
   ;
```

Generalized Predicate

Sometimes it is necessary to specify the context under which a semantic predicate is valid manually rather than allowing ANTLR to compute the context. The following predicate form can be used to specify a semantic predicate with specific syntactic context:

`( `*context*` )? => <<`*predicate*`>>?`

For example,

```
(ID)? => <<qualifiedItemIs()==Constructor>>?
```

This generalized predicate indicates that the semantic predicate should only be evaluated if `ID` is the next symbol of lookahead. You may use only simple strings of tokens inside the context "guard" (e.g., `( A B | C D )? => <<`*blah*`>>?`).

Tree operators

token-reference`!`

Do not create an AST node in the output tree for this reference.

rule-reference`!`

Do not link in the AST created by the referenced rule into the current output tree.

token-reference^

Create an AST node in the output tree for this token reference. It becomes the root of the tree being built for the enclosing rule.

Lexical Directives

`#token` *LABEL* `"`*regular-expression*`"` `<<`*action*`>>`
where any of the items may be omitted. However, actions may only be tied to regular expressions and at least one item must be specified.

`#lexclass` *LCLASS*
start a new automaton or lexical class in your grammar.

`#tokclass` *TCLASS* `{ T1 T2 ... Tn }`
Define *TCLASS* as a set of tokens.

`#tokdefs "`*file*`"`
Specify a file containing `#define`s or an `enum` of all token labels for ANTLR to use.

Parser Exception Handling

Rule With Exception Handlers

```
rule
   :  alternative1
      exception
         catch ... <<...>>
      exception[label]
         catch ... <<...>>
   |  alternative2
   ...
   |  alternativen
   ;
   exception
      catch ... <<...>>
      catch ... <<...>>
      default : <<...>>
   exception[label1]
```

```
      ...
   exception[label2]
      ...
```

where *label*, *label1*, and *label2* are labels attached to either rule or token references within the alternatives; specifically, *label* must be contained within `alternative`$_1$.

Token Exception Operator

```
stat
   :@ "if" INT "then" stat { "else" stat }
      <<printf("found if\n");>>
   |  id:ID@ "="@ INT@ ";"@
      <<printf("found assignment to %s\n", $id->getText());>>
   ;
```

The `@` on the front of alternative one indicates that each token reference in the alternative is to be handled without throwing an exception—the match routine will catch the error. The second alternative explicitly indicates that each token is to be handled locally without throwing an exception.

8 History

The PCCTS project began as a parser-generator project for a graduate course at Purdue University in the Fall of 1988 taught by Hank Dietz--"translator-writing systems". Under the guidance of Professor Dietz, the parser generator, ANTLR (originally called YUCC), continued after the termination of the course and eventually became the subject of Terence Parr's Master's thesis. Originally, lexical analysis was performed via a simple scanner generator which was soon replaced by Will Cohen's DLG in the Fall of 1989 (DFA-based lexical-analyzer generator, also an offshoot of the graduate translation course).

The alpha version of ANTLR was totally rewritten resulting in 1.00B. Version 1.00B was released via an internet newsgroup (comp.compilers) posting in February of 1990 and quickly gathered a large following. 1.00B generated only LL(1) parsers, but allowed the merged description of lexical and syntactic analysis. It had rudimentary attribute handling similar to that of YACC and did not incorporate rule parameters or return values; downward inheritance was very awkward. 1.00B-generated parsers terminated upon the first syntax error. Lexical classes (modes) were not allowed and DLG did not have an interactive mode.

Upon starting his Ph.D. at Purdue in the Fall of 1990, Terence Parr began the second total rewrite of ANTLR. The method by which grammars may be practically analyzed to generate LL(k) lookahead information was discovered in August of 1990 just before Terence's return to Purdue. Version 1.00 incorporated this algorithm and included the AST mechanism, lexical classes, error classes, and automatic error recovery; code quality and portability were higher. In February of 1992 1.00 was released via an article in SIGPLAN Notices. Peter Dahl, then Ph.D. candidate, and Professor Matt O'Keefe (both at the University of Minnesota) tested this version extensively. Dana Hoggatt (Micro Data Base Systems, Inc.) tested 1.00 heavily.

Version 1.06 was released in December 1992 and represented a large feature enhancement over 1.00. For example, rudimentary semantic predicates were introduced, error messages were significantly improved for k>1 lookahead and ANTLR parsers could indicate that lookahead fetches were to occur only when necessary for the parse (normally, the lookahead "pipe" was constantly full). Russell Quong joined the project in the Spring of 1992 to aid in the semantic predicate design. Beginning and advanced tutorials were created and released as well. A makefile generator was included that sets up dependencies and such correctly for ANTLR and DLG. Very few 1.00 incompatibilities were introduced (1.00 was quite different from 1.00B in some areas).

Version 1.10 was released on August 31, 1993 after Terence's release from Purdue and incorporated bug fixes, a few feature enhancements and a major new capability--an arbitrary lookahead operator (syntactic predicate), "`(`α`)?`β". This feature was codesigned with Professor Russell Quong also at Purdue. To support infinite lookahead, a preprocessor flag, `ZZINF_LOOK`, was created that forced the `ANTLR()` macro to tokenize all input prior to parsing. Hence, at any moment, an action or predicate could see the entire input sentence. The predicate mechanism of 1.06 was extended to allow multiple predicates to be hoisted; the syntactic context of a predicate could also be moved along with the predicate.

In February of 1994, SORCERER was released. This tool allowed the user to parse child-sibling trees by specifying a grammar rather than building a recursive-descent tree walker by hand. Aaron Sawdey at The University of Minnesota became a second author of SORCERER after the initial release.

On April 1, 1994, PCCTS 1.20 was released. This was the first version to actively support C++ output. It also included important fixes regarding semantic predicates and (..)+ subrules. This version also introduced token classes, the "not" operator, and token ranges.

On June 19, 1994, SORCERER 1.00B9 was released. Gary Funck of Intrepid Technology joined the SORCERER team and provided very valuable suggestions regarding the "transform" mode of SORCERER.

On August 8, 1994, PCCTS 1.21 was released. It mainly cleaned up the C++ output and included a number of bug fixes.

From the 1.21 release forward, the maintenance and support of all PCCTS tools was picked up by Parr Research Corporation.

A sophisticated error handling mechanism called "parser exception handling" was released for version 1.30. 1.31 fixed a few bugs.

Release 1.33 is the version corresponding to this initial book release.

APPENDIX

Notes for New Users of PCCTS

Thomas H. Moog
Polhode, Inc.
tmoog@polhode.com

These notes are based on my own experiences and a year of observing the PCCTS mailing list and the omp.compilers.tools.pccts news group. These notes have an emphasis on C++ mode. Those who are using C mode may wish to consult the first version of these notes mentioned prior to Item #1. The Notes consist of a table of contents and the Notes themselves. If an entry in the table-of-contents contains a dash ("–") instead of a page number than the title is the entire item, so there's no point in referring to another page for additional information. The code mentioned in the section of examples can be obtained via web browser or FTP from the site mentioned prior to Item #1 and at most PCCTS archive sites.

Where is

Basics

Checklist

#token

#tokclass

#lexclass

Lexical Lookahead

Line and Column Information

C++ Mode

ASTs

Rules

Init-Actions

Inheritance

Syntactic Predicates

Semantic Predicates

Debugging Tips for New Users of PCCTS

Switches and Options

Multiple Source Files

Source Code Format

Miscellaneous

(C Mode) LA/LATEXT and NLA/NLATEXT

(C Mode) Execution-Time Routines

(C Mode) Attributes

(C Mode) ASTs

Extended Examples and Short Descriptions of Distributed Source Code

Where is

#1. These notes, related examples, and an earlier version with an emphasis on C mode, are available on the net.

Primary Site:

web browser: http://www.mcs.net/~tmoog/pccts.html
anonymous ftp: ftp://ftp.mcs.net/mcsnet.users/tmoog/*

Europe:

anonymous ftp: ftp://ftp.th-darmstadt.de/pub/programming/languages/compiler-compiler/pccts/notes.newbie/*

#2. FTP sites for the Purdue Compiler Construction Tool Set (PCCTS).

Primary Site:

Node:	ftp.parr-research.com	(Parr Research, Inc.)
	ftp-mount.ee.umn.edu [128.101.146.5]	(University of Minnesota)
Files:	The PCCTS distribution kit:	/pub/pccts/*latest-version*/pccts.tar.Z *and* pccts.tar.gz
	FAQ comp.compilers.tools.pccts	/pub/pccts/documentation/FAQ
	Contributed files:	/pub/pccts/contrib/*
	Pre-built binaries for PCCTS:	/pub/pccts/binaries/PC /pub/pccts/binaries/SGI etc.

Note: There is no guarantee that these binaries will be up-to-date. They are contributed by users of these machines rather than thePCCTS developers.

Europe:

Node: ftp.th-darmstadt.de [130.83.55.75]
Directory: /pub/programming/languages/compiler-compiler/pccts

(This is updated weekly on Sunday.)

Also:

Node: ftp.uu.net [192.48.96.9]
Directory: languages/tools/pccts

#3. The FAQ is maintained by Michael T. Richter (mtr@globalx.net) and is available at the FTP site.

#4. Archive sites for MS-DOS programs for unpacking .tar and .gzip files (the format of the PCCTS distribution kit).

Node: oak.oakland.edu (Oakland University in Rochester, Michigan)
File: simtel/msdos/archiver/tar4dos.zip
File: simtel/msdos/compress/gzip124.zip

Node:	wuarchive.wustl.edu	(Washington University in St. Louis, Missouri)
File:	/archive/systems/ibmpc/simtel/msdos/archiver/tar4dos.zip	
File:	/archive/systems/ibmpc/simtel/msdos/compress/gzip124.zip	

Contributed by Bill Tutt
(rassilon@cs.simpson.edu)

#5. Example grammars for C++, ANSI C, Java, Fortran 77, and Objective C.

All the above mentioned grammars are located at the FTP site in /pub/pccts/contrib/*

The C++ grammar (FTP file pccts/contrib/cplusplus.tar), written in C++ mode, is the best demonstration available of the use of PCCTS capabilities. The desire to handle the C++ grammar in an elegant fashion led to a number of improvements in PCCTS.

The Fortran 77 grammar (C mode) by Ferhat Hajdarpasic (ferhath@ozemail.com.au) includes Sorcerer routines.

#6. Parr-Research web page: http://www.parr-research.com/~parrt/prc

Basics

#7. Invoke ANTLR or DLG with no arguments to get a switch summary

#8. Tokens begin with uppercase characters, rules begin with lowercase characters

#9. Even in C mode you can use C++ style comments in the non-action portion of ANTLR source code

Inside an action you have to obey the comment conventions of your compiler.

#10. In #token regular expressions spaces and tabs which are not escaped are ignored

This makes it easy to add white space to a regular expression:

```
#token  Symbol  "[a-z A-Z] [a-z A-Z 0-9]*"
```

#11. Never choose names which coincide with compiler reserved words or library names

You'd be surprised how often someone has done something like one of the following:

```
#token FILE   "file"
#token EOF    "@"
const:        "[0-9]*" ;
```

#12. Write <<predicate>>? not <<predicate *semi-colon*>>? (semantic predicates go in "if" conditions)

#13. Some constructs which cause warnings about ambiguities and optional paths

```
rule : a { ( b | c )* } ;
rule : a { b } ;
b    : ( c )* ;
rule : a c* ;
a    : b { c } ;
rule : a { b | c | } ;
```

Checklist

#14. Locate incorrectly spelled #token symbols using ANTLR –w2 switch or by inspecting *parserClassName*.C

If a #token symbol is spelled incorrectly ANTLR will assign it a new #token number which, of course, will never be matched.

#15. Duplicate definition of a #token name is not reported

ANTLR will simply use the later definition and forget the earlier one. Using the ANTLR –w2 option does not help

#16. Use ANTLR cross-reference option –cr to detect orphan rules when ambiguities are reported

#17. `LT`(*i*) and `LATEXT`(*i*) are magical names in semantic predicates — punctuation is critical

ANTLR wants to determine the amount of lookahead required for evaluating a semantic predicate. It does this by searching in C++ mode for strings of the form "`LT(`" and in C mode for strings of the form "`LATEXT(`". If there are spaces before the open "(" it won't make a match. It evaluates the expression following the "(" under the assumption that it is an integer literal (e.g."1"). If it is something like "`LT(1+i)`" then you'll have problems. With ANTLR switch –w2 you will receive a warning if ANTLR doesn't find at least one `LT`(*i*) in a semantic predicate.

#token

See also #8, #10, #14, #15, #22, #133, #134.

#18. To match any single character use: "`~[]`", to match everything to a newline use: "`~[\n]*`"

#19. To match an "@" in your input text use "`\@`", otherwise it will be interpreted as the end-of-file symbol

#20. The escaped literals in #token regular expressions are: `\t \n \r \b` (not the same as ANSI C)

#21. In #token expressions " \12 " is decimal " \012 " is octal, and " \0x12 " is hex (not the same as ANSI C)

Contributed by John D. Mitchell (johnm@alumni.eecs.berkeley.edu).

#22. DLG wants to find the longest possible string that matches

The regular expression "`~[]*`" will cause problems. It will gobble up everything to the end-of-file.

#23. When two regular expressions of equal length match a regular expression the first one is chosen

Thus more specific regular expressions should appear in the grammar file before more general ones:

```
#token  HELP  "help"  /*  should appear before "symbol" */
#token  Symbol  "[a-z A-Z]*"   /*  should appear after keywords  */
```

Some of these may be caught by using the DLG switch –Wambiguity. In the following grammar the input string "HELP" will never be matched:

```
#token  WhiteSpace      "[\ \t]"        <<skip();>>
#token  ID             "[a-z A-Z]+"
#token  HELP           "HELP"

statement
    : HELP "@"   <<printf("token HELP\n");>>           /* a1 */
    | "inline" "@" <<printf("token inline\n");>>       /* a2 */
    | ID "@"        <<printf("token ID\n");>>          /* a3 */
    ;
```

The best advice may be to follow the practice of TJP: place "#token ID" at the end of the grammar file.

#24. Inline regular expression are no different than #token statements

PCCTS code does *not* check for a match to "inline" (Item #23 line a2) before attempting a match to the regular expressions defined by #token statements. The first two alternatives ("a1" and "a2") will *never* be matched. All of this will be clear from examination of the file "parser.dlg" (the name does *not* depend on the parser's class name).

Another way of looking at this is to recognize that the conversion of character strings to tokens takes place in class DLGLexer, not class ANTLRParser, and that all that is happening with an inline regular expression is that ANTLR is allowing you to define a token's regular expression in a more convenient fashion — not changing the fundamental behavior.

If one builds the example above using the DLG switch –Wambiguity one gets the message:

```
dlg warning: ambigious regular expression  3  4
dlg warning: ambigious regular expression  3  5
```

The numbers which appear in the DLG message refer to the assigned token numbers. Examine the array _token_tbl in *parserClassName*.C to find the regular expression which corresponds to the token number reported by DLG:

```
ANTLRChar *Parser::_token_tbl[]={
    /* 00 */        "Invalid",
    /* 01 */        "@",
    /* 02 */        "WhiteSpace",
    /* 03 */        "ID",
    /* 04 */        "HELP",
    /* 05 */        "inline"
};
```

Well, there is one important difference for those using Sorcerer. With in-line regular expressions there is no symbolic name for the token, hence it can't be referenced in a Sorcerer rule. Contributed by John D. Mitchell (johnm@alumni.eecs.berkeley.edu).

#25. Watch out when you see ~[*list-of-characters*] at the end of a regular expression

What the user usually wants to express is that the regular expression should stop *before* the *list-of-characters*. However the expression will include the complement of that list as part of the regular expression. Often users forget about what happens to the characters which are in the complement of the set.

Consider for example a #lexclass for a C style comment:

```
/* C-style comment handling */
#lexclass COMMENT   /* a1 */
#token "\*/"     << mode(START); skip();>>              /* a2 */
#token "~[\*]+"  << skip();>>                          /* a3 */
#token "\*~[/]"  << skip(); >>  /* WRONG*/              /* a4 */
 /* Should be "\*"                          */          /* a5 */
 /* Correction due to Tim Corringham     */            /* a6 */
 /* tim@ramjam.demon.co.uk  20-Dec-94   */             /* a7 */
```

The RE at line a2 accepts "*/" and changes to #lexclass START. The RE at line a4 accepts a "*" which is *not* followed by a "/". The problem arises with comments of the form:

```
/* this comments breaks the example **/
```

The RE at line a4 consumes the "**" at the end of the comment leaving nothing to be matched by " */ ".

This is a relatively efficient way to span a comment. However it is not the simplest. A simpler description is:

```
#token "\*/"    << mode(START); skip(); >>   /* b1 */
#token "~[]"    << skip(); >>                /* b2 */
```

This works because b1 ("*/") is two characters long while b2 is only one character long — and DLG always prefers the longest expression which matches.

For those who are concerned with the efficiency of scanning:

```
#token        "[\n\r]"      <<skip();newline();>>
#token        "\*/"         <<mode(START);skip();>>
#token        "\*"          <<skip();>>
#token        "~[\*\n\r]+"  <<skip();>>
```

Contributed by Brad Schick (schick@interaccess.com)

#26. Watch out when one regular expression is the prefix of another

If the shorter regular expression is followed by something which can be the first character of the suffix of the longer regular expression, DLG will happily assume that it is looking at the longer regular expression. See Item #41 for one approach to this problem.

#27. DLG is not able to backtrack

Consider the following example:

```
#token          "[\ \t]*"       <<skip();>>
#token ELSE     "else"
```

```
#token ELSEIF   "else [\ \t]* if"
#token STOP     "stop"
```

with input:

```
else stop
```

When DLG gets to the end of "else" it realizes that the space will allow it to match a longer string than "else" by itself. So DLG accept the spaces. Everything is fine until DLG gets to the initial "s" in "stop". It then realizes it has no match — but it can't backtrack. It passes back an error status to ANTLR which (normally) prints out something like:

```
invalid token near line 1 (text was 'else ') ...
```

There is an "extra" space between the "else" and the closing single quote mark.

This problem is not detected by the DLG option –Wambiguity.

For this particular problem "else" and "if" can be treated as separate tokens. For more difficult cases work-arounds are (a) to push the problem onto the parser by using syntactic predicates or (b) to use Vern Paxson's lexical analyzer "flex" which has powerful backtracking capabilities. See Item #44 and Example #17.

#28. The lexical routines mode(), skip(), and more() have simple, limited use!

All they do is set status bits or fields in a structure owned by the lexical analyzer and then return immediately. Thus it is OK to call these routines anywhere from within a lexical action. You can even call them from within a subroutine called from a lexical action routine.

#29. lextext() includes strings accumulated via more() — begexpr()/endexpr() refer only to the last matched RE

#30. Use "`if (_lextext != _begexpr) {...}`" to test for RE being appended to lextext using more()

To track the line number of the *start* of a lexical element that may span several lines I use the following test:

```
if (_lextext == _begexpr) {startingLine=_line;}//user-defined var
```

#31. #token actions can access protected variables of the DLG base class (such as _line) if necessary

#32. Replace semantic routines in #token actions with semantic predicates.

In early versions on PCCTS it was common to change the token code based on semantic routines in the #token actions. With semantic predicates this technique is now frowned upon:

Old style:

```
#token TypedefName
#token ID  "[a-z A-Z]*"
       <<if (isTypedefName(lextext)) return TypedefName;>>
```

New Style C Mode:

```
#token ID  "[a-z A-Z]*"
typedefName : <<isTypedefName(LATEXT(1))>>? ID;
```

The old technique is appropriate for making *lexical* decisions based on the input; for instance, treating a number appearing in columns 1 through 5 as a statement label rather than a number. The new style is important because of the buffer between the lexer and parser introduced by large amounts of lookahead, especially syntactic predicates. For instance a declaration of a type may not have been entered into the symbol table by the parser by the time the lexer encounters a declaration of a variable of that type. An extreme case is infinite lookahead in C mode: parsing doesn't even begin until the entire input has been processed by the lexer. See Item #121 for an extended discussion of semantic predicates. Example #12 shows how some semantic decisions can be moved from the lexer to the token buffer.

#33. For 8 bit characters in DLG, make `char` variables unsigned by default (g++ option –funsigned-char).

For Unix systems this should be combined with a call to `setlocale(LC_ALL,"")` to replace the default locale of "C" with the user's native locale. Contributed by Ulfar Erlingsson (ulfarerl@rhi.hi.is).

#34. The maximum size of a DLG token is set by an optional argument of the ctor DLGLexer() — default is 2000.

The maximum size of a character string stored in an ANTLRToken is independent of the maximum size of a DLG token. See Item #54 and Example #5.

#35. If a token is recognized using more() and its #lexclass ignores end-of-file, then the very last token will be lost.

When a token is recognized in several pieces using more(), an end-of-file may have been detected before the entire token is recognized. Without treatment of this special case, the portions of the token already recognized will be ignored and the error of a lexically incomplete token will be ignored. Since all appearances of the regular expression "@", regardless of #lexclass, are mapped to the same #token value, proper handling requires some work-arounds.

Suppose you want to recognize C style comments using:

```
#lexclass START
#token  Comment_Begin  "/\*"  <<skip();mode(LC_Comment);more();>>
#token  Eof            "@"
...
#lexclass LC_Comment
#token  Unexpected_Eof  "@"   <<mode(START);>>
#token  Comment_End     "\*/" <<skip();mode(START);>>
#token                  "~[]" <<skip();more()>>
...
```

The token code "Unexpected_Eof" will never be seen by the parser. The result is that C style comments which omit the trailing "*/" can swallow all the input to the end-of-file and not give any error message. My solution to this problem is to fool PCCTS by using the following definition:

```
#token  Unexpected_Eof  "@@"   <<mode(START);>>
```

This exploits a characteristic of DLG character streams: once they reach end-of-file they must return end-of-file to every request for another character until explicitly reset.

Another example of this pitfall, with more serious implications, is the recognition of C style strings.

#tokclass

See also #41, #87, #88, #92.

#36. #tokclass provides an efficient way to combine reserved words into reserved word sets

```
#token Read     "read"
#token Write    "write"
#token Exec     "exec"
#token ID       "[a-z A-Z] [a-z A-Z 0-9 \@]*"
#tokclass Any     {ID Read Write Exec}
#tokclass Verb    {Read Write Exec}
command: Verb Any ;
```

#37. Use ANTLRParser::set_el() to test whether an ANTLRTokenType is in a #tokclass

To test whether a token "t" is in the #tokclass "Verb":

```
if (set_el(t->getType(),Verb_set)) {...}
```

There are several variations of this routine in the ANTLRParser class.

#lexclass

See also #41, #56.

#38. Inline regular expressions are put in the most recently defined lexical class

If the most recently defined lexical class is not START you may be surprised:

```
#lexclass START
...
#lexclass LC_Comment
...
inline_example: symbol "=" expression ;
```

This will place "=" in the #lexclass LC_Comment (where it will never be matched) rather than the START #lexclass where the user meant it to be. Since it is okay to specify a #lexclass in several pieces it might be a good idea when using #lexclass to place "#lexclass START" just before the first rule — then any inline definitions of tokens will be placed in the START #lexclass automatically:

```
#lexclass START
...
#lexclass COMMENT
```

```
...
#lexclass START
```

#39. Use a stack of #lexclass modes in order to emulate lexical subroutines

Consider a grammar in which lexical elements have internal structure. An example of this is C strings and character literals which may contain elements like:

escaped characters	`\" and \'`
symbolic codes	`\t`
numbers	`\xff  \200  \0`

Rather than implementing a separate #lexclass to handle these sequences for both character literals and string literals it would be possible to have a single #lexclass which would handle both. To implement such a scheme one needs something like a subroutine stack to remember the previous #lexclass. See Example #9 for a set of such routines.

#40. Sometimes a stack of #lexclass modes isn't enough

Consider a log file consisting of clauses, each of which has its own #lexclass and in which a given word is reserved in some clauses and not others:

```
#1;1-JAN-94 01:23:34;enable;forge bellows alarm;move to station B;
#2;1-JAN-94 08:01:56;operator;john bellows;shift change at 08:00;
#3;1-JAN-94 09:10:11;move;old pos=5.0 new pos=6.0;operator request;
#4;1-JAN-94 10:11:12;alarm;bellows;2-JAN-94 00:00:01;
```

If the item is terminated by a separator, there is a problem because the separator will be consumed in the recognition of the most nested item — with nothing left over to be consumed by other elements which end at the separator. The problem appears when it is necessary to leave a #lexclass and return more than one level. To be more specific, a #token action can only be executed when one or more characters is consumed. Therefore, to return through three levels of #lexclass calls would appear to require the consumption of at least three characters. In the case of balanced constructs like `"..."` and `'...'` this is not a problem since the terminating character can be used to trigger the #token action. However, if the scan is terminated by a *separator* such as the semi-colon above (;), you cannot use the same technique. Once the semi-colon is consumed, it is unavailable for the other #lexclass routines on the stack to see.

One solution is to allow the user to specify (during the call to pushMode) a "lookahead" routine to be called when the corresponding element of the mode stack is popped. At that point the "lookahead" routine can examine `ch` to determine whether it also wants to pop the stack, and so on up the mode stack. The consumption of a single character can result in popping multiple modes from the mode stack based on a single character of lookahead.

If your approach is more complicated than this, you might as well write a second parser just to handle the so-called lexical elements.

Continuing with the example of the log file (above): each statement type has its fields in a specific order. When the statement type is recognized, a pointer is set to a list of the

#lexclasses which is in the same order as the remaining fields of that kind of statement. An action is attached to every #token which recognizes a semi-colon (";") advances a pointer in the list of #lexclasses and then changes the #lexclass by calling mode() to set the #lexclass for the next field of the statement.

Lexical Lookahead

#41. One extra character of lookahead is available to the #token action routine in `ch` (except in interactive mode)

In interactive mode (DLG switch –i) DLG fetches a character only when it needs it to determine if the end of a token has been reached. In non-interactive mode the content of `ch` is always valid. The debug code described in Item #126 can help debug problems with interactive lookahead.

For the remainder of this discussion assume that DLG is in non-interactive mode.

Consider the problem of distinguishing floating point numbers from range expressions such as those used in Pascal:

range: 1..23 float: 1.23

As a first effort one might try:

```
#token  Int     "[0-9]+"
#token  Range   ".."
#token  Float   "[0-9]+.[0-9]*"
```

The problem is that "1..23" looks like the floating point number "1." with an illegal "." at the end. DLG always takes the longest matching string, so "1." will always look more appetizing than "1". What one needs to do is to look at the character following "1." to see if it is another ".", and if it is to assume that it is a range expression. The flex lexer has trailing context, but DLG doesn't — except for the single character in `ch`.

A solution in DLG is to write the #token Float action routine to look at what's been accepted, and at `ch`, in order to decide what to do:

```
#token Float    "[0-9]*.[0-9]*"
    <<if (*endexpr() == '.' && /* might use more complex   test    */
         ch == '.') {
          mode(LC_Range);   /* treat it like a range expression  */
          return Int;       /* looks like an int followed by ".."*/
      };
     >>

#lexclass LC_Range
#token Range    "."      <<mode(START);>>  // consume second "."
                                            of range
```

#42. The lex operators "^" and "$" (anchor pattern to start/end of line) can sometimes be simulated by DLG

DLG doesn't have operators to anchor a pattern match to the start or end of a line. However, a requirement that a string start at column 1 can sometimes be simulated by a

combination of #lexclass and #token action routines. A requirement that the string end at the end-of-line can sometimes be simulated in a #token action routine by testing whether ch is a newline.

In the following example, a "*" in column 1 is treated as a different lexical element than a "*" appearing elsewhere. This example depends on having column information enabled by use of trackColumns():

```
#token  Star_Col1
#token  Star            "*"      <<if (get_endcol() == 1) {
                                     return Star_Col1;}
                                 >>
#token  WhiteSpace      "[\ \t]"   <<skip();>>
#token  ID              "[a-z A-Z]+"
#token  NEWLINE         "\n"     <<newline(); set_endcol(0);>>

expr!   : (Star     <<printf ("\nThe * is NOT in column 1\n");>>
             | Star_Col1  <<printf ("\nThe * is in column 1\n");>>
             | ID         <<printf ("\nFirst token is an ID\n");>>
          )* "@" ;
```

#43. When the first non-blank token on a line may have a special interpretation

If the set of tokens which can appear at the start of a line is relatively small then code the newline #token action to switch to another #lexclass where just the start-of-line tokens will be recognized:

```
#lexclass START
#token NL      "\n [\t\ ]*" <<newline();skip();mode(StartOfLine);>>
#token Number  "[0-9]+"
#token Mult    "\*"

#lexclass StartOfLine
#token Label    "[0-9]+"      <<mode(START);>>
#token Comment  "\* ~[\n]*"   <<mode(START);>>
```

If the regular expressions that can appear only at the start of a line is a subset of the "anywhere" tokens then one can use a flag to determine which interpretation to assign to the regular expression just matched. Checking for begcol()==0 could also serve as the flag:

```
#lexclass START
#token NL     "\n [\ ]*" <<newline();skip();firstTokenOnLine=1;>>
#token Number "[0-9]+"   <<if (firstTokenOnLine) {return Label;};>>
#token Mult   "\*"       <<if(firstTokenOnLine){
                               skip();mode(LC_Comment);more();
                               };>>
#lexclass LC_Comment
#token   Comment "~[\n]*" <<skip();mode(START);>>
```

This requires that the flag "firstTokenOnLine" be cleared for every token but that of a newline. This would be rather tedious to code for every token #action. It's convenient to put it in a class derived from DLGLexer or from ANTLRTokenBuffer. It would be natural to put it in the makeToken routine, but it is difficult for makeToken to exchange information with the #token action routines. See Item #62.

Another approach is to include the newline as part of the regular expression:

```
#lexclass START
#token Number  "[0-9]+"
#token Label   "\n [\ ]* [0-9]+"
#token Mult    "\*"
#token Comment "\n [\ ]* \* ~[\n]*"
```

This requires that a newline be prepended to the input character stream and that the line number be decremented by 1 at the start to compensate for the extra newline. The problem is that a DLGInputStream doesn't know how to adjust its caller's line information (Item #62). In any case, the line information in the 1.32b6 is a protected variable. The result is that it is necessary to have a rather inelegant class derived from DLGLexer in order to accomplish this. See Example #8.

#44. For more powerful forms of lexical lookahead one can use Vern Paxson's flex

If more than one character of lookahead is necessary and it appears difficult to solve using #lexclass, semantic predicates, or other mechanisms you might consider using flex by Vern Paxson (University of California – Berkeley). Flex is a superset of lex. For an example of how to use flex with ANTLR in C++ mode see Example #17. For C mode visit the FTP site (Item #2) for file /pub/pccts/contrib/NOTES.flex.

See also #27, #42, #56, Example #15.

Line and Column Information

Most names in this section refer to members of class DLGLexerBase or DLGLexer

Before C++ mode the proper handling of line and column information was a large part of these notes.

#45. If you want column information for error messages (or other reasons) use C++ mode

#46. If you want accurate line information even with many characters of lookahead use C++ mode

#47. Call trackColumns() to request that DLG maintain column information

#48. To report column information in syntax error messages override ANTLRParser::syn() — See Example #6

#49. Call newline() and then set_endcol(0) in the #token action when a newline is encountered

#50. Adjusting column position for tab characters

Assume that tabs are set every eight characters starting with column 9.

Computing the column position will be simple if you match tab characters in isolation:

```
#token Tab  "\t"    <<_endcol=((_endcol-1) & ~7) + 8;>>
```

This would be off by 1, except that DLG, on return from the #token action, computes the next column using:

```
        _begcol=_endcol+1;
```

If you include multiple tabs and other forms of whitespace in a single regular expression, the computation of _endcol by DLG must be backed out by subtracting the length of the string. Then you can compute the column position by inspecting the string character by character.

#51. Computing column numbers when using more() with strings that include tab characters and newlines

```
/* what is the column and line position when the comment includes
   or is followed by tabs tab  tab */  tab  tab  i++;
```

> Note: This code excerpt requires a change to PCCTS 1.32b6 file pccts/dlg/output.c in order to inject code into the DLGLexer class header. The modified source code is distributed as part of the notes in file notes/changes/dlg/output.c and output_diff.c An example of its use is given in Example #7.

My feeling is that the line and column information should be updated at the same time more() is called because it will lead to more accurate position information in messages. At the same time one may want to identify the *first* line on which a construct begins rather than the line on which the problem is detected: it's more useful to know that an unterminated string started at line 123 than that is was still unterminated at the end-of-file.

```
void DLGLexer::tabAdjust () { // requires change to output.c
   char * p;              //   to add user code to DLGLexer
   if (_lextext == _begexpr) startingLineForToken=_line;
   _endcol=_endcol-(_endexpr-_begexpr)+1;  // back out DLG
                                              computation
   for (p=_begexpr;*p != 0; p++) {
      if (*p == '\n') {                 // newline() by itself
       newline();_endcol=0;            //   doesn't reset column
      } else if (*p == '\t') {
         _endcol=((_endcol-1) & ~7) + 8;   // traditional tab stops
      };
     _endcol++;
   };
   _endcol--;         // DLG will compute begcol=endcol+1
}
```

See Example #7 for a more complete description.

See also #42, #56.

C++ Mode

#52. The destructors of base classes should be virtual in almost all cases

If you don't know why, you should read Scott Meyers' excellent book, *Effective C++, Fifty Specific Ways* ....

#53. Why must the AST root be declared as ASTBase rather than AST ?

The functions which implement the rules of the grammar are declared with the prototype:

```
void aRule(ASTBase ** _root) {...};
```

The underlying support code of ANTLR depends only on the behaviors of ASTBase. There are two virtues to this design:

No recompilation of the underlying routines is necessary when the definition of AST changes

The same object code can be used with multiple parsers in the same program each with its own kind of AST

This is in contrast to C++ templates which are designed to provide source code reuse, not object code reuse.

An "AST *" can be passed to an "ASTBase *" why not an "AST **" for an "ASTBase **" ?

This is a C++ FAQ. Consider the following (invalid) code fragment:

```
struct B {};                                          /* a1 */
structD1:B {inti;};                                   /* a2 */
struct D2 : B {doubled;};                             /* a3 */
void func(B ** ppB) {*ppB=new D2;}; /* WRONG */     /* a4 */
D1 * pD1=newD1;                                       /* a5 */
func(&pD1);                                           /* a6 */
```

At line a5, pD1 is declared to be a pointer to a D1. This pointer is passed to "func" at line a6. The function body at line a4 replaces a pointer to a D1 with a pointer to a D2, which violates the declaration at line a5.

The following *is* legal, although it may not do what is expected:

```
void func2(B * pB) {D1d1;*pB=d1;};             /* b1 */
func2(pD1);                                     /* b2 */
```

The assignment at line b5 *slices* d1 and assigns only the B part of d1 to the object pointed to by pB because the assignment operator chosen is that of class B, not class D1.

#54. ANTLRCommonToken text field has maximum length fixed at compile time – but there's an alternative

For ANTLRCommonToken the length of the text field is fixed by #define ANTLRCommonTokenTEXTSIZE. The default is 100 characters. If you want an already written routine which will handle tokens which are limited by the size of the DLG token buffers look at the definition of ANTLRToken in Example #5 file varToken.*.

#55. C++ Mode makes multiple parsers easy.

pccts/testcpp/5/test.g Uses multiple instances of a single parse class (thus a single grammar)

pccts/testcpp/6/main.C Program uses parsers for two different grammars (test.g and test2.g)

If two parsers share the same DLG automaton it may be necessary to save DLG state. See Item #56.

#56. Use DLGLexerBase routines to save/restore DLG state when multiple parsers share a token buffer.

When the second parser "takes control" the DLGLexer doesn't know about it and doesn't reset the state variables such as #lexclass, line number, column tracking, etc.

Use DLGLexerBase::saveState (DLGState *) and restoreState(DLGState *) to save and restore DLG state.

#57. Required AST constructors: AST(), AST(ANTLRToken), and AST(X x,Y y) for #[X x,Y y]

#58. In C++ Mode ASTs and ANTLRTokens do not use stack discipline as they do in C mode.

In C mode ASTs and ANTLRTokens are allocated on a stack. This is an efficient way to allocates space for structs and is not a serious limitation because in C it is customary for a structure to be of fixed size. In C++ mode it would be a serious limitation to assume that all objects of a given type were of the same size because derived classes may have additional fields. For instance one may have a "basic" AST with derived classes for unary operators, binary operators, variables, and so on. As a result the C++ mode implementation of symbolic tags for elements of the rule uses simple pointer variables. The pointers are initialized to 0 at the start of the rule and remain well defined for the entire rule. The things they point will normally also remained well defined, even objects defined in sub-rules:

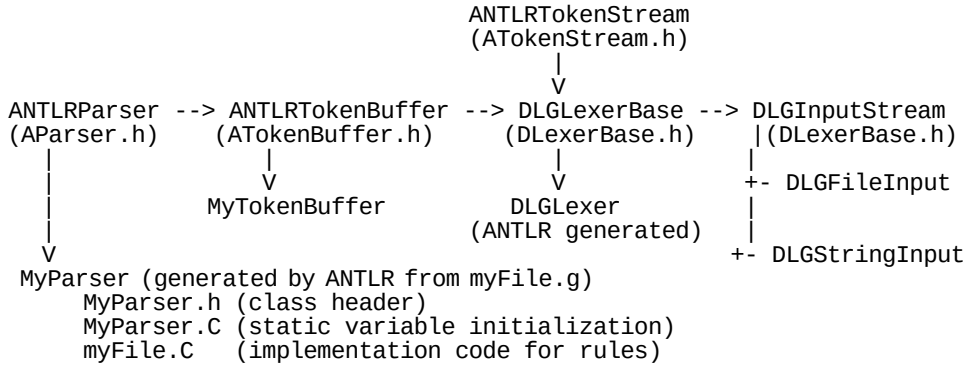
```
rule ! : a:rule2 {b:B} <<#0=#(#a,#[$b]);>> ; // OK only in C++ mode
```

This fragment is not be well defined in C mode because B would become undefined on exit from "`{...}`".

#59. Summary of Token class inheritance in file AToken.h.

```
ANTLRAbstractToken — (empty class) no virtual table
  |
  V
ANTLRLightweightToken — (token type) no virtual table
  |
  V
ANTLRTokenBase — (token type, text, line) virtual table
  |
  V
DLGBasedToken — (token type, text, line) virtual table
  |
  +-- ANTLRCommonToken — (token type, text, line) virtual table
  |      using fixed length text fields
  |
  +-- MyToken — (token type, text, line, ...) virtual table
        notes/var/varToken.h — variable length text fields
        notes/col/myToken.h — variable length text with
          column info
```

#60. Diagram showing relationship of major classes.



#61. Tokens are supplied as demanded by the parser. They are "pulled" rather than "pushed".

```
ANTLRParser::consume()
--> ANTLRTokenBuffer::getToken()
--> ANTLRTokenBuffer::getANTLRToken()
--> DLGLexer::getToken()
--> MyToken::makeToken(ANTLRtokenType,lexText,line)
```

#62. Because tokens are "pulled" it is easy to access downstream objects but difficult to access upstream objects

There is a pointer from the ANTLRParser to the ANTLRTokenBuffer, from the ANTLRTokenBuffer to the DLGLexer, and from the DLGLexer to the DLGInputStream. However if the DLGInputStream wants to reset the DLGLexer line number, there's no pointer in the DLGInputStream object which points to the "parent" DLGLexer object. The linked list is one-way.

The user can may want to derive a class from DLGInputStream in which there is a member function setParser() thereby changing a one-way linked-list into a circular list.

#63. Additional notes for users converting from C to C++ mode

In general:	`zz`*name*	=> *name*, *_name*, or *name*`()`
example:	`zzlextext`	=> `_lextext`, `lextext()`
except for:	`zzchar`	=> `ch`
In DLGLexerBase:	`NLA=tokenCode`	=> `return tokenCode`
	`line++`	=> `newline()`
	`line=value`	=> `_line=value`
	`zztokens[`*i*`]`	=> *parserClassName*`::tokenName(`*i*`)`
	The tokenName() function is promised for the next release of PCCTS — or see Example #7 for how to define your own tokenName function.	
	`zzendcol`	=> `_endcol`, `set_endcol()`, `get_endcol()`

```
zzbegcol      => _begcol, set_begcol(),
                 get_begcol()
```

ASTs

#64. To enable AST construction (automatic or explicit) use the ANTLR –gt switch

#65. Use symbolic tags (rather than numbers) to refer to tokens and ASTs in rules

prior to version 1.30:
```
rule! : x y          <<#0=#(#1,#2);>>   ;
```
with version 1.30:
```
rule! : xx:x yy:y    <<#0=#(#xx,#yy);>> ;
```

The symbolic tags are implemented as pointers to ASTs. The pointers are initialized to 0 at the start of the rule and remain defined for the entire rule. See Item #58. Rules no longer return pointers to tokens (Item #83

#66. Constructor AST(ANTLRToken *) is automatically called for terminals when ANTLR –gt switch is used

This can be suppressed using the "!" operator.

#67. If you use ASTs you have to pass a root AST to the parser

```
ASTBase     *root=NULL;
...
Parser.startRule(&root,otherArguments);
root->preorder();
root->destroy();
```

#68. Use ast–>destroy() to recursively descend the AST tree and free all sub-trees

#69. Don't confuse `#[...]` with `#(...)`

The first creates a single AST node using an AST constructor (which is usually based on an ANTLRToken or an ANTLRTokenType). It converts lexical information to an AST.

The second creates an AST tree or list (usually more than a single node) from other ASTs by filling in the "down" field of the first node in the list to create a root node, and the "sibling" fields of each of the remaining ASTs in the lists. It combines existing ASTs to create a more complex structure.

```
#token ID        "[a-z]*"
#token COLON     ":"
#token Stmt_With_Label

id! : name:ID    <<#0=#[Stmt_With_Label,$name->getText()];>> ;
                                                    /*a1*/
```

The new AST (a single node) contains Stmt_With_Label in the token field, given a traditional version of AST::AST(ANTLRTokenType,char *).

```
rule! : name:id COLON e:expr   <<#0=#(#name,#e);>> ;
                                                 /* a2 */
```

Creates an AST list with "name" at its root and "e" as its first (and only) child.

The following example (a3) is equivalent to a1, but more confusing, because the two steps above have been combined into a single action:

```
rule! : name:ID COLON e:expr                          /* a3 */
             <<#0=#(#[Stmt_With_Label,$name>getText()],#e);>> ;
```

#70. The make-a-root operator for ASTs ("^") can be applied only to terminals (#token, #tokclass, #tokdef)

A child rule might return a tree rather than a single AST. Were this to happen it could not be made into a root as it is *already* a root and the corresponding fields of the structure are in use. To make an AST returned by a called rule a root use the expression: #(root-rule, sibling1, sibling2, sibling3).

```
                addOp               : "\+" | "\-";
               #tokclass AddOp      { "\+"   "\-"}
/* OK    */    add !                expr ("\+"^ expr) ;
 /* Wrong */   addExpr !            : expr (addOp^ expr) ;
 /* OK    */    addExpr !           : expr (AddOp^ expr);
```

#71. An already constructed AST tree cannot be the root of a new tree

An AST tree (unless it's a trivial tree with no children) already has made use of the "down" field in its structure. Thus one should be suspicious of any constructs like the following:

```
rule! : anotherRule:rule2........ <<#0=#(#anotherRule,...);>> ;
```

#72. Don't assign to #0 unless automatic construction of ASTs is disabled using the "!" operator on a rule

```
a! : xx:x yy:y zz:z <<#0=#(#xx,#yy,#zz);>> ; // ok
a  : xx:x yy:y zz:z <<#0=#(#xx,#yy,#zz);>> ; // NOT ok
```

The reason for the restriction is that assignment to #0 will cause any ASTs pointed to by #0 to be lost when the pointer is overwritten.

#73. The statement in Item #72 is stronger than necessary

You can assign to #0 even when using automated AST construction if the old tree pointed to by #0 is part of the new tree constructed by `#(...)`. For example:

```
#token Comma      ","
#token Stmt_List

stmt_list: stmt (Comma stmt)*  <<#0=#(#[Stmt_List],#0);>> ;
```

The automatically constructed tree pointed to by #0 is just put at the end of the new list, so nothing is lost. If you reassign to #0 in the middle of the rule, automatic tree construction will result in the addition of remaining elements at the end of the new tree. This is not recommended by TJP.

Special care must be used when combining the make-a-root operator (e.g. rule: expr Op^ expr) with this transgression (assignment to #0 when automatic tree construction is selected).

#74. A rule that constructs an AST returns an AST even when its caller uses the "!" operator

#75. (C++ Mode) In AST mode a token which isn't incorporated into an AST will result in lost memory

For a rule like the following:

```
rule : FOR^ lValue EQ! expr TO! expr BY! expr ;
```

the tokens "EQ", "TO", and "BY" are not incorporated into any AST. In C mode the memory they occupied (they are called *attributes* in C mode) would be recovered on rule exit. In C++ mode their memory will be lost unless special action is taken or the user enables the ANTLR reference counting option. Another approach is to use the NoLeakToken class from Example #5.

#76. When passing `#(...)` or `#[...]` to a subroutine it must be cast from "ASTBase *" to "AST *"

Most of the PCCTS internal routines are declared using ASTBase rather than AST because they don't depend on behavior added by the user to class AST. Usually PCCTS hides this by generating explicit casts, but in the case of subroutine arguments the hiding fails and the user needs to code the cast manually. See also Item #135.

#77. Some examples of `#(...)` notation using the PCCTS list notation

See page 45 of the 1.00 manual for a description of the PCCTS list notation.

```
a: A ;
b: B ;
c: C ;

#token T_abc

r : a b c    <<;>>           ;/* AST list (0 A B C) without root */
r!: a b c    <<#0=#(0,#1,#2,#3);>> ;/* AST list (0 A B C) without
                                                        root */
r : a! b! c! <<#0=#(0,#1,#2,#3);>> ;/* AST list (0 A B C) without
                                                        root */
r : a^ b c                 ;/* AST tree (A B C) with root A    */
r!: a b c    <<#0=#(#1,#2,#3);>>   ;/* AST tree (A B C) with
                                                    root A     */
r!: a b c    <<#0=#(#[T_abc],#1,#2,#3);>>
                              ;/* AST tree (T_abc_node A B C)     */
                               /*   with root T_abc_node          */
r : a b c    <<#0=#(#[T_abc],#0);>>     ;  /* the same as above  */
r : a! b! c! <<#0=#(#[T_abc],#1,#2,#3);>> ; /* the same as above */
```

#78. A rule which derives epsilon can short circuit its caller's explicitly constructed AST

When a rule derives epsilon it will return an AST value of 0. As the routine which constructs the AST tree (ASTBase::tmake) has a variable length argument list which is terminated by 0, this can cause problem with `#(...)` lists that have more than two elements:

```
rule ! : DO body:loop_body END_DO <<#0=#(#[DO],#body,#[END_DO];>> ;
loop_body : { statement_list } ;  /* can return 0 on DO END_DO  */
```

Although this particular example could be handled by automatic tree construction, the problem is a real one when constructing a tree by adding more than one sibling at a time. This problem does not exist for automatically constructed AST trees because those trees are constructed one element at a time. Contributed by T. Doan (tdoan@bnr.ca).

#79. How to use automatic AST tree construction when a token code depends on the alternative chosen

Suppose one wants to make the following transformation:

```
rule :  lv:lhs ;          => #(#[T_simple],#lv)
rule :  lv:lhs  rv:rhs ;  => #(#[T_complex],#lv,#rv)
```

Both lhs and rhs considered separately may be suitable for automatic construction of ASTs, but the change in token type from "T_simple" to "T_complex" appears to require manual tree construction. Use the following idiom:

```
rule : lhs (
             ()      <<#0=#(#[T_simple],#0);>>
             | rhs   <<#0=#(#[T_complex],#0);>>
           ) ;
```

Another solution:

```
rule : <<ANTLRTokenType t=T_simple;>>
           l:lhs { r:rhs <<t=T_complex;>> }
                                    <<#0=#(#[t],#0);>> ;
```

#80. For doubly linked ASTs derive from class ASTDoublyLinkedBase and call tree–>double_link(0,0).

The ASTDoublyLinkedBase class adds "up" and "left" fields to the AST definition, but it does not cause them to be filled in during AST construction. After the tree is built call tree->double_link(0,0) to traverses the tree and fill in the up and left fields.

#81. When ASTs are constructed manually the programmer is responsible for deleting them on rule failure.

It is worth a little bit of extra trouble to let PCCTS construct the AST for a rule automatically in order to obviate the need for writing a fail action for a rule. A safer implementation might be to maintain a doubly linked list of all
ASTs from which an AST is removed when it is destroyed. See class NoLeakAST from Example #6.

See also #100, #102.

Rules

#82. To refer to a field of an ANTLRtOKEN within a rule's action use `<<... mytoken($x)->field...>>`

ANTLR puts all "ANTLRToken*" variables in an ANTLRTokenPtr object in order to maintain reference counts for tokens. When the reference counter goes to zero the token is deleted (assuming that the ANTLRToken definition is derived from ANTLRRefCountToken). One result of this is that rule actions which need to refer to a real ANTLRToken field must first convert an ANTLRTokenPtr to an "ANTLRToken*" using the macro "mytoken":

```
number: n:Number <<if (mytoken($n)->value < 0)   {...};>>
```

#83. Rules don't return tokens values, thus this won't work: `rule: r1:rule1 <<...$r1...>>`

In earlier versions of PCCTS (C mode) it was accepted practice to assign an attribute to a rule:

```
rule : rule1  <<$0=$1;>>
```

However, with the introduction of symbolic tags for labels (Item #65) this feature became deprecated for C mode (Item #152) and is not even supported for C++ mode. To return a pointer to a token (ANTLRToken *) from a rule use inheritance (See Item #99):

```
statement
       : <<ANTLRToken * t;>>  rule > [t] ;
rule >  [ANTLRToken *t]
       : r1:rule1   <<$t=someAction($r1);>>
```

It's still standard practice to pass back AST information using assignment to #0 and to refer to such return values using labels on rules. It's also standard practice to refer to tokens associated with *terminals*:

```
rule : xx:X    <<...$xx...>>    // okay: "X" is a terminal (token)
rule : xx:x    <<...$xx...>>    // won't work: "x" is a rule rather
x    : xx:X    <<$x=$xx;>>      //  than a terminal (token)
```

#84. A simple example of rewriting a grammar to remove left recursion

ANTLR can't handle left-handed recursion. A rule such as:

```
expr : expr Op expr
     | Number
     | String
     ;
```

will have to be rewritten to something like this:

```
expr : Number (Op expr)*
     | String (Op expr)*
     ;
```

#85. A simple example of left-factoring to reduce the amount of ANTLR lookahead

Another sort of transformation required by ANTLR is left-factoring:

```
rule : STOP WHEN expr
     | STOP ON expr
     | STOP IN expr
     ;
```

These are easily distinguishable when *k*=2, but with a small amount of work it can be cast into a *k*=1 grammar:

```
rule : STOP ( WHEN expr
            | ON expr
            | IN expr
            ) ;
```

or:

```
rule        : STOP rule_suffix
            ;
rule_suffix : WHEN expr
            | ON expr
            | IN expr
            ;
```

An extreme case of a grammar requiring a rewrite is in Example #13.

#86. ANTLR will guess where to match "@" if the user omits it from the start rule

ANTLR attempts to deduce "start" rules by looking for rules which are not referenced by any other rules. When it finds such a rule it assumes that an end-of-file token ("@") should be there and adds one if the user did not code one. This is the only case, according to TJP, when ANTLR adds something to the user's grammar.

#87. To match any token use the token wild-card expression "." (dot)

This can be useful for providing a context dependent error message rather than the all purpose message "syntax error".

```
if-stmt : IF "\(" expr "\)" stmt
        | IF .   <<printf("If statement requires expression"
                          "enclosed in parenthesis");
                   PARSE_FAIL;      // user defined
                 >>
        ;
```

This particular case is better handled by the parser exception facility.

A simpler example:

```
quoted : "quote" . ;            // quoted terminal
```

#88. The "~" (tilde) operator applied to a #token or #tokclass is satisfied when the input token does *not* match

```
anything : (~ Newline)* Newline ;
```

The "~" operator cannot be applied to rules. Use syntactic predicates to express the idea "if this rule doesn't match try to match this other rule".

#89. To list the rules of the grammar grep *parserClassName*.h for "_root" or edit the output from ANTLR –cr

#90. The ANTLR –gd trace option can be useful in sometimes unexpected ways

For example, by suitably defining the functions ANTLRParser::tracein and ANTLRParser::traceout one can accumulate information on how often each rule is invoked. They could be used to provide a traceback of active rules following an error provided that the havoc caused by syntactic predicates' use of setjmp/longjmp is properly dealt with.

#91. Associativity and precedence of operations is determined by nesting of rules

In the example below "=" associates to the right and has the lowest precedence. Operators "+" and "*" associate to the left with "*" having the highest precedence.

```
expr0   : expr1 {"="^expr0};                /* a1 */
expr1   : expr2 ("\+"^ expr2)*;             /* a2 */
expr2   : expr3 ("\*"^ expr3)* ;            /* a3 */
expr3   : ID;                               /* a4 */
```

The more deeply nested the rule the higher the precedence. Thus precedence is "*" > "+" > "=". Consider the expression "x=y=z". Will it be parsed as "x=(y=z)" or as "(x=y)=z" ? The first part of expr0 is expr1. Because expr1 and its descendants cannot match an "=" it follows that all derivations involving a *second* "=" in an expression must arise from the "`{...}`" term of expr0. This implies right association.

In the following samples the ASTs are shown in the root-and-sibling format used in PCCTS documentation. The numbers in brackets are the serial number of the ASTs. This was created by code from Example #6.

```
a=b=c=d
( = <#2>  a <#1>  ( = <#4>  b <#3>  ( = <#6>  c <#5>  d <#7>  ) )
                                                    ) NL <#8>
a+b*c
( + <#2>  a <#1>  ( * <#4>  b <#3>  c <#5>  ) ) NL <#6>
a*b+c
( + <#4>  ( * <#2>  a <#1>  b <#3>  ) c <#5>  ) NL <#6>
```

#92. #tokclass can replace a rule consisting only of alternatives with terminals (no actions)

One can replace:

```
addOp          : "\+" | "\-" ;
```

with:

```
#tokclass AddOp { "\+"  "\-" }
```

This replaces a modest subroutine with a simple bit test. A #tokclass identifier may be used in a rule wherever a simple #token identifier may be used.

The other work-around is much more complicated:

```
expr1! : left:expr2 <<#0=#l;>>
            (op:addOp right:expr2  <<#0=#(#op,#left,#right);>> )* ;
addOp  : "\+" | "\-" ;
```

The "!" for rule "expr1" disables automatic constructions of ASTs in the rule. This allows one to manipulate #0 manually. If the expression had no addition operator then the sub-rule "(addOp expr)*" would not be executed and #0 will be assigned the AST constructed by #left. However if there *is* an addOp present then each time the sub-rule is rescanned due to the "`(...)*`" the current tree in #0 is placed as the first of two siblings underneath a new tree. This new tree has the AST returned by addOp as the root. It is a left-leaning tree.

#93. Rather than comment out a rule during testing, add a nonsense token which never matches — See Item #96

See also #8, #13,#16,#66, #92, #95, #116.

Init-Actions

#94. Don't confuse init-actions with leading-actions (actions which precede a rule).

If the first element following the start of a rule or sub-rule is an action it is always interpreted as an init-action. An init-action occurs in a scope which includes the entire rule or sub-rule. An action which is *not* an init-action is enclosed in "{" and "}" during generation of code for the rule and has essentially zero scope — the action itself.

The difference between an init-action and an action which precedes a rule can be especially confusing when an action appears at the start of an alternative. These *appear* to be almost identical, but they aren't:

```
b  : <<int i=0;>>  b1 > [i] /* b1  <<...>> is an init-action*/
   | <<int j=0;>>  b2 > [j] /* b2  <<...>> is part of the rule*/
   ;                        /* and will cause a compilation error*/
```

On line "b1" the `<<...>>` appears immediately after the beginning of the rule making it an init-action. On line "b2" the `<<...>>` does *not* appear at the start of a rule or sub-rule, thus it is interpreted as a leading action which happens to precede the rule.

This can be especially dangerous if you are in the habit of rearranging the order of alternatives in a rule.

For instance, changing this:

```
b  : <<int i=0,j=0;>> <<i++;>>  b1 > [i]   /* c1 */
   | <<j++;>>  b1 > [i]                    /* c2 */
   ;
```

to this:

```
b  : /* empty production */                /* d1 */
   | <<int i=0,j=0;>> <<i++;>>  b1 > [i]   /* d2 */
   | <<j++;>>  b1 > [i]
   ;
```

or to this:

```
b
   : <<j++;>>  b1 > [i]                    /* e1 */
   | <<int i=0,j=0;>> <<i++;>>  b1 > [i]   /* e2 */
   ;
```

changes an init-action into a non-init action, and vice-versa.

#95. An empty sub-rule can change a regular action into an init-action.

A particularly nasty form of the init-action problem is when an empty sub-rule has an associated action:

```
rule!: name:ID (
                  /* empty */
                    <<#0=#[ID,$name];>>
                | ab:array_bounds
                   <<#0=#[T_array_declaration,$name],#ab);>>
                );
```

Since there is no reserved word in PCCTS for epsilon, the action for the empty arm of the sub-rule becomes the init-action. For this reason it's wise to follow one of the following conventions

– Represent epsilon with an empty rule "()"

– Put the null rule as the last rule in a list of alternatives:

```
rule!: name:ID (
                 ()  <<#0=#[ID,$name];>>
                 | ab:array_bounds
                       <<#0=#[T_array_declaration,$name],#ab);>>
               );
```

The cost of using "()" to represent epsilon is small.

#96. Commenting out a sub-rule can change a leading-action into an init-action.

Suppose one comments out a rule in the grammar in order to test an idea:

```
rule                            /* a1 */
        : <<init-action;>>      /* a2 */
////      rule_a                /* a3 */
        | rule_b                /* a4 */
        | rule_c                /* a5 */
        ;
```

In this case one only wanted to comment out the "rule_a" reference in line a3. The reference is indeed gone, but the change has introduced an epsilon production, which probably creates a large number of ambiguities. Without the init-action the ":" would have probably have been commented out also, and ANTLR would report a syntax error — thus preventing one from shooting oneself in the foot. See Item #93.

Commenting out a rule can create orphan rules, which can lead to misleading reports of ambiguity in the grammar. To detect orphan rules use the ANTLR –cr (cross-reference) switch.

#97. Init-actions are executed just once for sub-rules: `(...)+`, `(...)*`, and `{...}`

Consider the following example from section 3.6.1 (page 29) of the 1.00 manual:

```
a : <<List *p=NULL;>>   // initialize list
    Type
    (  <<int i=0;>>    // initialize index
```

```
            v:Var <<append(p,i++,$v);>>
          )*
          <<OperateOn(p);>>
        ;
```

See also #104, #112, #116.

Inheritance

#98. Downward inherited variables are just normal C arguments to the function which recognizes the rule

If you are using downward inheritance syntax to pass results back to the caller (really upward inheritance !), then it is necessary to pass the *address* of the variable which will receive the result.

#99. Upward inheritance returns arguments by passing back values

If the rule has more than one item passed via upward inheritance, then ANTLR creates a `struct` to hold the result and then copies each component of the structure to the upward inheritance variables.

```
#token  T_int
#token  T_real
#token  T_complex

class P {
...
number : <<int useRadix=10;int iValue;double rValue;double
                                                    rPart,iPart;>>
         { radix > [useRadix] }
                intNumber [useRadix] > [iValue]
        | realNumber > [rValue]
        | complexNumber > [rPart,iPart]
;
complexNumber > [double rPart,double iPart] :
        "\[" realNumber > [$rPart] "," realNumber > [$iPart] "\]"
;
realNumber > [double result] :
        v:"[0-9]+.[0-9]*"            <<$result=toDouble($v);>>
;
radix > [int i] : v:"%[0-9]+"        <<$i=toInt($v);>>
;
intNumber [int radix] > [int result] :
        v:"[0-9]+"                   <<$result=toInt($v);>>
;
}
```

This example depends on the use of several constructors for ASTs and user defined routines toInt() and toDouble().

#100. ANTLR –gt code will include the AST with downward inheritance values in the rule's argument list

See also #113.

Syntactic Predicates

The terms "infinite lookahead," "guess mode," and "syntactic predicate" all imply use of the same facility in PCCTS to provide a limited amount of backtracking by the parser. In this case we are not referring to backtracking in DLG or other lexers. The term "syntactic predicate" emphasizes that it is handled by the parser. The term "guess mode" emphasizes that the parser may have to backtrack. The term "guess mode" may also be used to distinguish two mutually exclusive modes of operation in the ANTLR parser:

> — Normal mode: A failure of the input to match the rules is a syntax error. The parser executes actions, constructs ASTs, reports syntax errors it finds (or invokes parser exception handling) and attempts automatic recovery from the syntax errors. There is no provision for backtracking in this mode.
>
> — Guess mode: The parser attempts to match a "`(...)?`" block and knows that it must be able to backtrack if the match fails. In this case the parser does *not* execute user-actions (except init-actions), nor does it construct ASTs. Failed validation predicates do not result in backtracking even when in guess mode.

In C++ mode there lookahead using a sliding window of tokens whose initial size is specified when the ANTLRTokenBuffer is constructed. In C mode the entire input is read, processed, and tokenized by DLG before ANTLR begins parsing. The term "infinite lookahead" derives from the initial implementation in ANTLR C mode.

#101. Regular actions are suppressed while in guess mode because they have side effects

#102. Automatic construction of ASTs is suppressed during guess mode because it is a side effect

#103. Syntactic predicates should not have side-effects

If there is no match then the rule which uses the syntactic predicate won't be executed.

#104. How to use init-actions to create side-effects in guess mode (despite Item #103)

If you absolutely have to have side-effects from syntactic predicates one can use exploit the fact that ANTLR always executes init-actions, even in guess mode:

```
rule   : (prefix)? A
       | B
       ;
prefix : <<regular-init-action-that's-always-executed>>
         A ( <<init-action-for-empty-subrule>> ) B
       ;
```

The init-actions in "prefix" will always be executed (perhaps several times) in guess-mode. Contributed by TJP.

#105. With values of *k*>1 or infinite lookahead mode one cannot use feedback from parser to lexer.

As infinite lookahead mode can cause large amounts of the input to be scanned by DLG before ANTLR begins parsing one cannot depend on feedback from the parser to the lexer to handle things like providing special token codes for items which are in a symbol table (the "lex hack" for `typedefs` in the C language). Instead one *must* use semantic predicates which allow for such decisions to be made by the parser or place such checks in the ANTLRTokenBuffer routine getToken() which is called every time the parser needs another token. See Example #12.

#106. Can't use interactive scanner (ANTLR –gk option) with ANTLR infinite lookahead.

#107. Syntactic predicates are implemented using setjmp/longjmp — beware C++ objects requiring destructors.

Semantic Predicates

#108. (Bug) Semantic predicates can't contain string literals

A predicate containing a string literal is incorrectly "string-ized" in the call to zzfailed_predicate.

```
rule:  <<containsCharacter("!@#$%^&*",LT(1)->getText()>>?  ID ;
                              /*  Will not work */
```

The work-around is to place the literal in a string constant and use the variable name.

#109. (Bug) Semantic predicates can't cross lines without escaped newline

```
rule: <<do_test();\
           this_is_a_workaround)>>? x y z ; /*** Note escaped
                                                 newline ***/
```

#110. Semantic predicates have higher precedence than alternation: `<<>>? A|B` means `(<<>>? A)|B`

#111. Any actions (except init-actions) inhibit the hoisting of semantic predicates

Here is an example of an empty leading action whose sole purpose is to inhibit hoisting of semantic predicates appearing in rule2 into the prediction for rule1. Note the presence of the empty init-action (See Item #94).

```
rule1    : <<;>> <<>> rule2
         | rule3
         ;
rule2    : <<semanticPred(LT(1)->getText())>>? ID ;
```

#112. Semantic predicates that use local variables or require init-actions must inhibit hoisting

#113. Semantic predicates that use inheritance variables must not be hoisted

You cannot use downward inheritance to pass parameters to semantic predicates which are *not* validation predicates. The problem appears when the semantic predicate is hoisted into a parent rule to predict which rule to call:

For instance:

```
a  :  b1 [flag]
   |  b2
   ;
b1 [int flag]
   : <<flag && hasPropertyABC(LT(1)->getText())>>? ID ;
b2 : ID ;
```

When the semantic predicate is evaluated within rule "`a`" to determine whether to call `b1, b2,` or `b3` the compiler will discover that there is no variable named "`flag`" for procedure "`a()`". If you are unlucky enough to have a variable named "`flag`" in `a()`, then you will have a *very* difficult-to-find bug.

#114. A semantic predicate which is not at the left edge of a rule becomes a validation predicate

Decisions about which rule of a grammar to apply are made before entering the code which recognizes the rule. If the semantic predicate is not at the left edge of the production, then the decision has already been made and it is too late to change rules based on the semantic predicate. In this case the semantic predicate is evaluated only to verify that it is true and is termed a "validation predicate."

#115. Semantic predicates are not always hoisted into the prediction expression

Even if a semantic predicate is on the left edge, there is no guarantee that it will be part of the prediction expression. Consider the following two examples:

```
a  :  <<semantic-predicate>>?  ID glob       /* a1 */
   |  ID glob                               /* a2 */
  ;
b  :  <<semantic-predicate>>?  ID glob       /* b1 */
   |  Number glob                          /* b2 */
  ;
```

With *k*=1 rule "a" requires the semantic predicate to disambiguate alternatives a1 and a2 because the rules are otherwise identical. Rule "b" has a token type of Number in alternative b2 so it can be distinguished from b1 without evaluation of the semantic predicate during prediction. In both cases the semantic predicate will be validated by evaluation inside the rule.

#116. Semantic predicates can't be hoisted into a sub-rule: "`{x} y`" is not exactly equivalent to "`x y | y`"

Consider the following grammar extract:

```
class Expr {
    e1 : (e2)+ END ;
    xid: <<is_xid(LT(1)->getText())>>? ID ;
    yid: <<is_yid(LT(1)->getText())>>? ID ;

  /* Works        */  e2:  xid "." yid | yid ;/* a1 */
  /* Doesn't work */  e2:  {xid "."} yid ;    /* a2 */
}
```

Alternatives a1 and a2 appear to be equivalent, but a1 works on input "abc" and a2 doesn't because only the semantic predicate of xid is hoisted into production e1 (but not the semantic predicate of yid).

Explanation by TJP: These alternatives are not really the same. The *language* described however is the same. The rule:

```
e2: {xid "."} yid ;
```

is shorthand for:

```
e2: (xid "." | /* epsilon */ ) yid ;
```

Rule e2 has no decision to make here — hence, yid does not get its predicate hoisted. The decision to be made for the empty alternative does not get the predicate from yid hoisted because one can't hoist a predicate *into* a subrule from beyond the subrule. The program might alter things in the subrule so that the predicate is no longer valid or becomes valid.

Contributed by Kari Grano (grano@cc.Helsinki.fi).

#117. How to change the reporting of failed semantic predicates

To make a global change #define the macro zzfailed_predicate(string) prior to the #include of pccts/h/AParser.h

One can change the handling on a case-by-case basis by using the "failed predicate" action which is enclosed in "[" and "]" and follows immediately after the predicate:

```
a : <<isTypedef(LT(1)->getText())>>?
                [{printf("Not a typedef\n");};]  ID ;
```

Douglas Cuthbertson (Douglas_Cuthbertson.JTIDS@jtids_qmail.hanscom.af.mil) has pointed out that ANTLR doesn't put the fail action inside "`{...}`". This can lead to problems when the action contains multiple statements.

For an example of conversion of a failed semantic predicate into a parser exception see Example #16.

#118. A semantic predicate should be free of side-effects because it may be evaluated multiple times.

Even in simple grammars semantic predicate are often evaluated twice: once in the prediction expression for a rule and once inside the rule as a validation predicate to make sure the semantic predicate is valid.

A semantic predicate may be hoisted into more than one prediction expressions.

A prediction expression may be evaluated more than once as part of syntactic predicates (guess mode).

#119. There's no simple way to avoid evaluation of a semantic predicate for validation after use in prediction.

#120. What is the "context" of a semantic predicate ?

Answer according to TJP: The context of a predicate is the set of *k*-strings (comprised of lookahead symbols) that can be matched following the execution of a predicate. For example,

```
a : <<p>>? alpha ;
```

The context of "`p`" is Look(`alpha`) where Look(`alpha`) is the set of lookahead *k*-strings for `alpha`.

```
class_name: <<isClass(LT(1)->getText())>>? ID ;
```

The context of `<<isClass ...>>?` is ID for *k*=1. Only *k*=1 is used since only LT(1) is referenced in the semantic predicate. It is important to use "–prc on" for proper operation. The old notation:

```
class_name: <<LA(1)==ID ? isClass(LT(1)->getText()) : 1>>? ID ;
                  /* Obsolete notation incompatiable with -prc on */
```

shouldn't be used for new grammars. It is not compatible with "–prc on". The only reason "–prc on" is not the default is backward compatibility.

Here is an example that won't work because it doesn't have context check in the predicates:

```
a          : ( class_name | Num )
           | type_name
           ;
class_name : <<isClass(LT(1)->getText())>>? ID ;
type_name  : <<isType(LT(1)->getText())>>? ID ;
```

The prediction for production one of rule "a" is:

```
if ( LA(1) in { ID, Num } && isClass(LT(1)->getText()))  {...}
```

Clearly, Num will never satisfy isClass(), so the production will never match.

When you ask ANTLR to compute context, it can check for missing predicates. With –prc on, for this grammar:

```
a   : b
    | <<isVar(LT(1)->getText())>>?      ID
    | <<isPositive(LT(1)->getText()>>?  Num
    ;
b   : <<isType(LT(1)->getText())>>?     ID
    | Num
    ;
```

ANTLR reports:

```
warning alt 1 of rule itself has no predicate to resolve
ambiguity upon { Num }
```

#121. Semantic predicates, predicate context, and hoisting

The interaction of semantic predicates with hoisting is sometimes subtle. Hoisting involves the evaluation of semantic predicates in a rule's parent in order to determine whether the rule associated with the semantic predicate is "viable". There are two ways to generate code for semantic predicates which are "hoisted" into a parent rule. With "–prc off", the default, the behavior of semantic predicates resembles gates which enable

or disable various productions. With "–prc on" the behavior of semantic predicates resemble a token for which its token type is determined by run-tine information rather than by purely lexical information. It is important to understand what "-prc on" does, when to use semantic predicates, and when to choose an alternative method of using semantic information to guide the parse. We start with a grammar excerpt which does not require hoisting, then add a rule which requires hoisting and show the difference in code with predicate context computation off (the default) and on.

```
statement
        : upper
        | lower
        | number
        ;
upper   : <<isU(LT(1)->getText())>>? ID ;
lower   : <<isL(LT(1)->getText())>>? ID ;
number  : Number ;
```

The code generated (with one ambiguity warning) resembles:

```
if (LA(1)==ID && isU) {
    upper();
} else if (LA(1)==ID && isL) {
    lower();
} else if (LA(1)==Number) {
    number();
...
```

Now the need for a non-trivial prediction expression is introduced:

```
parent  : statement
        | ID
        ;
statement
        : upper
        | number
        ;
```

Running ANTLR causes one ambiguity warning. The code for "statement" resembles:

```
if ( (LA(1)==ID || LA(1)==Number) && isU) {
     statement();
} else if (LA(1)==ID) {
...
```

Even if LA(1) is a Number, the semantic predicate isU() will be evaluated. Depending on the way that isU is written it may or may not be meaningful. This is exactly the problem addressed by predicate computation. With "–prc on" one receives two ambiguity warnings and the code for "statement" resembles:

Code –prc on	Outline format –prc on

`if ( (LA(1)==ID ||`
`  LA(1)==Number) &&`
`  ( !(LA(1)==ID) ||`
`  (LA(1)==ID && isU)) {`
`  statement();`
`} else if (LA(1)==ID) {`
`  ...`
	`&&`
	`  ||`
	`  LA(1)==ID`
	`  LA(1)==Number`
	`  ||`
	`  !  <===== not ...`
	`  LA(1)==ID  <===== an ID`
	`  isU(LT(1)->getText())`

The important thing to notice is the call to isU() is guarded by a test that insures that the token is indeed an ID.

The following does not change anything because ANTLR already knows that the lookahead context for the semantic predicates can only be "ID":

```
upper   : (ID)? => <<isU(LT(1)->getText())>>? ID ;
```

Consider the following grammars excerpts all built with –k 2 and "–prc on":

```
#token  X       "x"
#token  Y       "y"
#token  A_or_B  "a | b"

class P {
statement : ( ax_or_by | bx )* "@"
          ;
ax_or_by  : aName X
          | bName Y
          ;
bx        : bName X
          ;
aName     : <<isa(LT(1)->getText())>>? A_or_B ;
bName     : <<isb(LT(1)->getText())>>? A_or_B ;
```

With input "bx" the above example issues an error message when the semantic predicate "aName" fails. The rule "statement" predicts "ax_or_by" because the gate "bName" is true. In searching for a viable rule to call "statement" finds "ax_or_by" to be the first alternative with a semantic predicate which is true (or with no semantic predicate). With option "–prc off" this is the intended mode of operation. ANTLR doesn't realize that the second token doesn't match because the second token isn't part of the semantic predicate.

Outline format –prc off	Outline format –prc on

`Alternative ax_or_by`
 	
`&&`
`  LA(1)==A_or_B`
`  ||`
`  LA(2)==X`
`  LA(2)==Y`
`  ||`
`  isa(LT(1)->getText())`
`  isb(LT(1)->getText())`
	`Alternative ax_or_by`
	
	`&&`
	`  LA(1)==A_or_B`
	`  ||`
	`  LA(2)==X`
	`  LA(2)==Y`
	`  ||`
	`  !`
	`  ||`
	`  LA(1)==A_or_B`
	`  LA(1)==A_or_B`
	`  ||`
	`  &&`
	`  LA(1)==A_or_B`
	`  isa(LT(1)->getText())`
	`  &&`
	`  LA(1)==A_or_B`
	`  isb(LT(1)->getText())`

`Alternative bx`
 	
`&&`
`  LA(1)==A_or_B`
`  LA(2)==X`
`  isb(LT(1)->getText())`
	`Alternative bx`
	
	`&&`
	`  LA(1)==A_or_B`
	`  LA(2)==X`
	`  ||`
	`  !`
	`  LA(1)==A_or_B`
	`  isb(LT(1)->getText())`

If the semantic predicates are expanded inline one gets:

```
ax_or_by
        : <<isa(LT(1)->getText())>>? A_or_B X
        | <<isb(LT(1)->getText())>>? A_or_B Y
        ;
bx      : <<isb(LT(1)->getText())>>? A_or_B X
        ;
```

One still gets a failure of the semantic predicate for "A_or_B X". By adding a reference to LT(2) one lets ANTLR know that the context is two tokens:

```
ax_or_by : <<LT(2),isa(LT(1)->getText())>>? A_or_B X
         | <<LT(2),isb(LT(1)->getText())>>? A_or_B Y
         ;
bx       : <<LT(2),isb(LT(1)->getText())>>? A_or_B X
         ;
```

This performs exactly as desired for the inputs "ax", "by", and "bx".

Outline format –prc off	Outline format –prc on

`Alternative ax_or_by`
 	
`&&`
`  LA(1)==A_or_B`
`  ||`
`  LA(2)==X`
`  LA(2)==Y`
`  ||`
`  LT(2),isa(LT(1)->getText())`
`  LT(2),isb(LT(1)->getText())`
	`Alternative ax_or_by`
	
	`&&`
	`  LA(1)==A_or_B`
	`  ||`
	`  LA(2)==X`
	`  LA(2)==Y`
	`  ||`
	`  !`
	`  ||`
	`  &&`
	`  LA(1)==A_or_B`
	`  LA(2)==X`
	`  &&`
	`  LA(1)==A_or_B`
	`  LA(2)==Y`
	`  ||`
	`  &&`
	`  &&`
	`  LA(1)==A_or_B`
	`  LA(2)==X`
	`  LT(2),isa(LT(1)->getText())`
	`  &&`
	`  &&`
	`  LA(1)==A_or_B`
	`  LA(2)==Y`
	`  LT(2),isb(LT(1)->getText())`

`Alternative bx`
 	
`&&`
`  LA(1)==A_or_B`
`  LA(2)==X`
`  LT(2),isb(LT(1)->getText())`
	`Alternative bx`
	
	`&&`
	`  LA(1)==A_or_B`
	`  LA(2)==X`
	`  ||`
	`  !`
	`  &&`
	`  LA(1)==A_or_B`
	`  LA(2)==X`
	`  LT(2),isb(LT(1)->getText())`

You can't test more context than is available at the point of definition. The following won't work:

```
/* Wrong */  aName : <<LT(2),isa(LT(1)->getText())>>? A_or_B ;
/* Wrong */  bName : <<LT(2),isb(LT(1)->getText())>>? A_or_B ;
```

One can often avoid problems by rearranging the code:

```
ax_by_bx : aName X
         | bName Y
         | bName X
         ;
```

or even:

```
bx_or_by : bName X
         | bName Y
         ;
ax       : aName X
         ;
```

This code works without special effort because the semantic predicates in each alternative of "statement" are mutually exclusive. Whether this matches what one needs for translation is a separate question.

I consider semantic predicates and hoisting to be a part of ANTLR which requires some vigilance. The ANTLR –w2 switch should be used and reports of ambiguities should be checked.

The code used to format the "if" conditions of the semantic predicates is notes/diagram/*.

See also #12, #17, #142, Example #12, Example #16.

#122. Another example of predicate hoisting

Consider the following grammar fragment which uses semantic predicates to disambiguate an ID in rules ca and cb:

```
a : ( { b | X } Eol)* "@" ;    /* a1 */
b : c ID ;                     /* a2 */
c : {ca} {cb} ;                /* a3 */

ca: <<pa(LATEXT(1))>>? ID;     /* a4 */
cb: <<pb(LATEXT(1))>>? ID;     /* a5 */
```

The code generated for rule c resembles:

```
if (LA(1)==ID) && pa(LATEXT(1))) {  /* b1 */
   ca();                            /* b2 */
} else {                            /* b3 */
   goto exit;                       /* b4 */
};                                  /* b5 */
```

The test of "pb" does not even appear. The problem is that the element "{cb}" is not at the left edge of rule c – even though "{ca}" is an optional element. Although "ca" may match epsilon, its presence in rule c still blocks the hoisting of the predicate in rule cb.

A first effort to solve this problem is to rewrite rule c so as to place "cb" on the left edge of the production:

```
c : ()                         /* c1 */
  | ca {cb}                    /* c2 */
  | cb                         /* c3 */
  ;                            /* c4 */
```

The code generated for rule c now resembles:

```
if (LA(1)==ID) {                /* d1 */
  ;                             /* d2 */
} else if (LA(1)==ID && pa(LATEXT(1))) {/* d3 */
   ...                           /* d4 */
```

It is clear that rules ca and cb are now unreachable because any ID will always match the test at line d1. The order of alternatives should be changed to:

```
c : ca {cb}                     /* e1 */
  | cb                          /* e2 */
  | ()                          /* e3 */
  ;                             /* e4 */
```

However our problems aren't over yet. The code generate for the "(...)*" test in rule "a" resembles:

```
while ( (LA(1)==X || LA(1)==Eol || LA(1)==ID) &&   /* f1 */
        (pa(...) || pb(...)) {                  /* f2 */
  ...                                                /*
f3 */
```

If both pa and pb are false then the body of the rule is never entered even though it should match an X or and ID using the rule on line a2 when rule c derives epsilon. I believe this is a problem in the handling of semantic predicates when confronted with productions which can derive epsilon.

Contributed by Sigurdur Asgeirsson (sigurasg@meanandmice.is).

See also #12, #17, #142, Example #12, Example #16.

Debugging Tips for New Users of PCCTS

#123. A syntax error with quotation marks on separate lines means a problem with newline

```
line 1: syntax error at "
" missing ID
```

#124. Use the ANTLR –gd switch to debug via rule trace

#125. Use the ANTLR –gs switch to generate code with symbolic names for token tests

#126. How to track DLG results

If you can't figure out what the DLG lexer is doing, try inserting the following code in class DLGLexerBase member nextToken() near line 140 of pccts/h/DLexer.C. This is one of the code samples — please see Example #10.

Just below:

```
tk=(this->*actions[accepts[state]])();/* invokes action routine */
```

Add this:

```
#ifdef DEBUG_LEXER
        printf("\ntoken type=%s lextext=(%s) mode=%d",
                parserClassName::tokenName(tk),
                (_lextext[0]=='\n' && _lextext[1]==0) ?
                        "newline" : _lextext,
                automaton);
          if (interactive && !charfull) {
          printf(" char=empty");
          } else {
           if (ch=='\n') {
             printf(" char=newline");
```

```
            } else {
              printf(" char=(%c)",ch);
            };
          };
          printf(" %s\n",
                 (add_erase==1 ? "skip()" :
                  add_erase==2 ? "more()" :
                  ""));
#endif
```

tk:	token number of the token just identified
lextext:	text of the token just identified
ch:	lookahead character
parserClassName:	name of the user's parser class
	This must be "hard-coded". In 1.32b6 there is no way for a DLGLexerBase object to determine the parser which requested the token. See Item #62.
tokenName	static member function of the *parserClassName* class
	Promised for the next release of PCCTS — or see Example #7 for how to define your own tokenName function.

Switches and Options

#127. Use ANTLR –gx switch to suppress regeneration of the DLG code and recompilation of DLGLexer.C

#128. Can't use an interactive scanner (ANTLR –gk option) with ANTLR infinite lookahead

#129. To make DLG case insensitive use the DLG –ci switch

The analyzer does not change the text, it just ignores case when matching it against the regular expressions.

See also #7, #14, #16, #66, #90, #100, #106, #124, #125.

Multiple Source Files

#130. To see how to place main() in a .C file rather than a grammar file (".g") see pccts./testcpp/8/main.C

```
#include "tokens.h"
#include "myParserClass.h"
#include "DLGLexer.h"
```

#131. How to put file scope information into the second file of a grammar with two .g files

If one did place a file scope action in the second file, ANTLR would interpret it as the fail action of the last rule appearing in the first grammar file.

To place file scope information in the second file #include the generated file in yet another file which has the file scope declarations.

Source Code Format

#132. To place the C right shift operator ">>" inside an action use "\>>"

If you forget to do this you'll get the error message:

```
warning: Missing <<; found dangling >>
```

This doesn't work with #lexaction or #header because the ">>" will be passed on to DLG which has exactly the same problem as ANTLR. The only work-around I've found for these special cases was to place the following in an #include file "shiftr.h":

```
#define SHIFTR  >>
```

where it is invisible to ANTLR and DLG. Then I placed a #include "shiftr.h" in the #lexaction.

No special action is required for the shift left operator.

#133. One cannot continue a regular expression in a #token statement across lines

If one tries to use "\" to continue the line DLG will think you are trying to match a newline character. A workaround (not completely equivalent) is to break the regular expression into several parts and use more() to combine them into a single token.

#134. A #token without an action will attempt to swallow an action which immediately follows it

This is a minor problem when the #token is created for use with attributes or ASTs nodes and has no regular expression:

```
#token  CastExpr
#token  SubscriptExpr
#token  ArgumentList
<<
... Code related to parsing
>>
```

You'll receive the message:

```
warning: action cannot be attached to a token name
        (...token name...); ignored
```

See also #9, #164.

Miscellaneous

#135. Given `rule[A a,B b] > [X x]` the proto is `X rule(ASTBase* ast,int* sig,A a,B b)`

The argument "sig" is the status value returned when using parser exception handling.

If automatic generation of ASTs is not selected, exceptions are not in use, or there are no inheritance variables then the corresponding arguments are dropped from the argument list. Thus with ASTs disabled, no parser exception support, and neither upward nor downward inheritance variables the prototype of a rule would be:

```
void rule()
```

See also #53 and #76.

#136. To remake ANTLR changes must be made to the makefile as currently distributed

The first problem is that generic.h does not appear in the dependency lists. The second problem is that the rebuild of antlr.c from antlr.g and of scan.c from parser.dlg have been commented out so as to allow building ANTLR on a machine without ANTLR the first time when there are problems with tar restoring modification dates for files.

#137. ANTLR reports "... action buffer overflow ..."

There are several approaches:

Usually one can bypass this problem with several consecutive action blocks. Contributed by M.T. Richter (mtr@globalx.net).

One can place the code in a separate file and use #include. Contributed by Dave Seidel (dave@numega.com or 75342.2034@compuserve.com).

One can change ANTLR itself. Change ZZLEXBUFSIZE near line 38 of pccts/antlr/generic.h and re-make.

#138. Exception handling uses status codes and `switch` statements to unwind the stack rule by rule

#139. For tokens with complex internal structure add #token expressions to match frequent errors

Suppose one wants to match something like a floating point number, character literal, or string literal. These have a complex internal structure. It is possible to describe them exactly with DLG. But is it wise to do so ? Consider:

`'\ff'` for `'\xff'` or `"\mThe result is: "` for `"\nThe result is: "`

If DLG fails to tolerate small errors like the ones above the result could be dozens of error messages as it searches for the closing quotation mark or apostrophe.

One solution is to create additional #token definitions which recognize common errors and either generates an appropriate error message or return a special #token code such as "Bad_String_Const". This can be combined with a special #lexclass which scans (in a very tolerant manner) to the end of the construct and generates no additional errors. This is the approach used by John D. Mitchell (johnm@alumni.eecs.berkely.edu) in the recognizer for C character and string literals in Example #2.

Another approach is to try to scan to the end of the token in the most forgiving way possible and then to validate the token's syntax in the DLG action routine.

#140. See pccts/testcpp/2/test.g and testcpp/3/test.g for examples of how to intergrate non-DLG lexers with PCCTS

The examples were written by Ariel Tamches (tamches@cs.wisc.edu).

#141. Ambiguity, full LL(*k*), and the linear approximation to LL(*k*)

It took me a while to understand in an intuitive way the difference between full LL(*k*) lookahead given by the ANTLR –k switch and the linear approximation given by the ANTLR –ck switch. Most of the time I run ANTLR with –k 1 and –ck 2. Because I didn't understand the linear approximation I didn't understand the warnings about ambiguity. I couldn't understand why ANTLR would complain about something which I thought was obviously parse-able with the lookahead available. I would try to make the messages go away totally, which was sometimes very hard. If I had understood the linear approximation I might have been able to fix them easily or at least have realized that there was no problem with the grammar, just with the limitations of the linear approximation.

I will restrict the discussion to the case of "–k 1" and "–ck 2".

Consider the following example:

```
rule1   : rule2a | rule2b | rule2c ;
rule2a  : A X | B Y | C Z ;
rule2b  : B X | B Z ;
rule2c  : C X ;
```

It should be clear that with the sentence being only two tokens this should be parseable with LL(2).

Instead, because k=1 and ck=2 ANTLR will produce the following messages:

```
/pccts120/bin/antlr -k 1 -gs -ck 2 -gh example.g
ANTLR parser generator   Version 1.20   1989-1994
example.g, line 23: warning: alts 1 and 2 of the rule itself
         ambiguous upon { B }, { X Z }
example.g, line 23: warning: alts 1 and 3 of the rule itself
         ambiguous upon { C }, { X }
```

The code generated resembles the following:

```
if      (LA(1)==A || LA(1)==B || LA(1)==C) &&
        (LA(2)==X || LA(2)==Y || LA(2)==Z) then rule2a()
else if (LA(1)==B) &&
        (LA(2)==X || LA(2)==Z) then rule2b()
else if (LA(1)==C) &&
        (LA(2)==X) then rule3a()
        ...
```

This might be called "product-of-sums". There is an "or" part for LA(1), an "or" part for LA(2), and they are combined using "and". To match, the first lookahead token must be in the first set and the second lookahead token must be in the second set. It doesn't matter that what one really wants is:

```
if      (LA(1)==A && LA(2)==X) ||
        (LA(1)==B && LA(2)==Y) ||
        (LA(1)==C && LA(2)==Z) then rule2a()
else if (LA(1)==B && LA(2)==X) ||
        (LA(1)==B && LA(2)==Z) then rule2b()
else if (LA(1)==C && LA(2)==X) then rule2c()
```

This happens to be "sums-of-products" but the real problem is that each product involves one element from LA(1) and one from LA(2) and as the number of possible tokens increases the number of terms grows as N^2. With the linear approximation the number of terms grows (surprise) linearly in the number of tokens.

ANTLR won't do this with k=1 (it would for k=2). It will only do "product-of-sums". However, all is not lost — you simply add a few well chosen semantic predicates which you have computed using your LL(*k*>1), mobile, water-resistant, all purpose, guaranteed-for-a-lifetime, carbon based, analog computer.

The linear approximation selects for each branch of the "if" a set which may include *more* than what is wanted but never selects a *subset* of the correct lookahead sets. We simply insert a hand-coded version of the LL(2) computation. It's ugly, especially in this case, but it fixes the problem. In large grammars it may not be possible to run ANTLR with k=2, so this fixes a few rules which cause problems. The generated parser may run faster because it will have to evaluate fewer terms at execution time.

```
<<
int use_rule2a() {
  if ( LA(1)==A && LA(2)==X ) return 1;
  if ( LA(1)==B && LA(2)==Y ) return 1;
  if ( LA(1)==C && LA(2)==Z ) return 1;
  return 0;
}
>>

rule1   :
        <<use_rule2a()>>? rule2a | rule2b | rule2c ;
rule2a  : A X | B Y | C Z ;
rule2b  : B X | B Z ;
rule2c  : C X ;
```

Correction due to Monty Zukowski
(monty@tbyte.com)

The real cases I've coded have shorter code sequences in the semantic predicate. I coded this as a function to make it easier to read and because there is a bug in ANTLR 1.3x which prevents semantic predicates from crossing lines. Another reason to use a function (or macro) is to make it easier to read the generated code to determine when your semantic predicate is being hoisted too high. It's easy to find references to a function name with the editor — but difficult to locate a particular sequence of "LA(1)" and "LA(2)" tests. Predicate hoisting is a separate issue which is described in Item #121.

In some cases of reported ambiguity it is not necessary to add semantic predicates because no *valid* token sequence could get to the wrong rule. If the token sequence were invalid it would be detected by the grammar eventually, although perhaps not where one might wish. In other cases the only necessary action is a reordering of the ambiguous rules so that a more specific rule is tested first. The error messages still appear, but one can ignore them or place a trivial semantic predicate (i.e. <<1>>?) in front of the later

rules. This makes ANTLR happy because it thinks you've added a semantic predicate which fixes things.

#142. What is the difference between "`(...)? <<...>>?` *x*" and "`(...)? => <<...>>?` *x*" ?

The first expression is a syntactic predicate followed by a semantic predicate. The syntactic predicate can perform arbitrary lookahead and backtracking before committing to the rule. However it won't encounter the semantic predicate until already committed to the rule — this makes the semantic predicate merely a validation predicate. Not a very useful semantic predicate.

The second expression is a semantic predicate with a convenient notation for specifying the look-ahead context. The context expression is used to generate an "if" condition similar to that used to predict which rule to invoke. It isn't any more powerful than the grammar analysis implied by the values you've chosen for the ANTLR switches –k and –ck. It doesn't have any of the machinery of syntactic predicates and does *not* allow arbitrarily large lookahead.

#143. Memory leaks and lost resources

Syntactic predicates use setjmp/longjmp and can cause memory leaks (Item #107).
Delete temporary attributes on rule failure and exceptions (Item #156).
Delete temporary ASTs on rule failure and exceptions (Item #81).
A rule that constructs an AST returns an AST even when its caller uses the "!" operator (Item #74).
(C++ Mode) A rule which applies "!" to a terminal loses the token (Item #75) unless the ANTLR reference counting option is enabled.
(C Mode) Define a zzd_ast() routine if you define a zzcr_ast() or zzmk_ast() (Item #163).

#144. Some ambiguities can be fixed by introduction of new #token numbers

For instance in C++ with a suitable definition of the class "C" one can write:

```
C a,b,c                         /* a1 */
a.func1(b);                     /* a2 */
a.func2()=c;                    /* a3 */
a = b;                          /* a4 */
a.operator =(b);                /* a5 */
```

Statement a5 happens to place an "=" (or any of the usual C++ operators) in a token position where it can cause a lot of ambiguity in the lookahead. set. One can solve this particular problem by creating a special #lexclass for things which follow "operator" with an entirely different token number for such operator strings — thereby avoiding the whole problem.

```
//
//  C++ operator sequences (somewhat simplified for these
                                    notes)
//
```

```
 //  operator <type_name>
 //  operator <special characters>
 //
 // There must be at least one non-alphanumeric character
// between "operator" and operator name - otherwise they
//  would be run together - ("operatorint" instead of
// "operator int")
//
#lexclass LEX_OPERATOR
#token  FILLER_C1              "[\ \t]*"
          <<skip();
            if( isalnum(ch) ) mode(START);
          >>
#token  OPERATOR_STRING      "[\+\-\*\/%\^\&\|\~\!\=\<\>]*"
                        <<mode(START);>>
#token  FILLER_C2               "\(\) | \[\] "
                      <<mode(START);return OPERATOR_STRING;>>
```

#145. Use "#pragma approx" to replace full LL(*k*) analysis of a rule with the linear approximation

To be supplied.

(C Mode) LA/LATEXT and NLA/NLATEXT

#146. Do not use LA(*i*) or LATEXT(*i*) in the action routines of #token

To refer to the token code (in a #token action) of the token just recognized use NLA. NLA is an lvalue (can appear on the left hand side of an assignment statement). To refer to the text just recognized use zzlextext (the entire text) or NLATEXT. One can also use zzbegexpr/zzendexpr which refer to the last regular expression matched. The char array pointed to by zzlextext may be larger than the string pointed to by zzbegexpr and zzendexpr because it includes substrings accumulated through the use of zzmore().

#147. Care must be taken in using LA(*i*) and LATEXT(*i*) in interactive mode (ANTLR switch –gk)

In interactive mode ANTLR doesn't guarantee that it will fetch lookahead tokens until absolutely necessary. It is somewhat safer to refer to lookahead information in semantic predicates, but care is still required.

In this table the entries "prev" and "next" means that the item refers to the token which precedes (or follows) the action which generated the output. For semantic predicate entries think of the following rule:

```
rule : <<semantic-predicate>>? Next NextNext ;
```

For rule-action entries think of the following rule:

```
        rule : Prev <<action>> Next NextNext;
-------------------------------------------------------------------
               k=1        k=1        k=3        k=3        k=3
             standard  infinite   standard  interactive  infinite
-------------------------------------------------------------------
for a semantic predicate
------------------------
  LA(0)         Next       Next       --         --          --
  LA(1)         Next       Next      Next       Next        Next
  zzlextext     Next       Next      Next        --         Next
  ZZINF_LA(0)              Next                             Next
  ZZINF_LA(1)              NextNext
NextNext
-----------------
for a rule action
-----------------
  LA(0)         Prev       Prev       --        Prev         --
  LA(1)         Prev       Prev      Prev       Next        Prev
  zzlextext     Prev       Prev      Prev        --         Prev
  ZZINF_LA(0)              Prev                             Prev
  ZZINF_LA(1)              Next                             Next
-------------------------------------------------------------------
```

(C Mode) Execution-Time Routines

#148. Calls to zzskip() and zzmore() should appear only in #token actions (or in subroutines they call)

#149. Use ANTLRs or ANTLRf in line-oriented languages to control the prefetching of characters and tokens

Write your own input routine and then use ANTLRs (input supplied by string) or ANTLRf (input supplied by function) rather than plain ANTLR which is used in most of the examples.

#150. Saving and restoring parser state in order to parse other objects (input files)

Suppose one wants to parse files that "include" other files. The code in ANTLR (antlr.g) for handling #tokdefs statements demonstrates how this may be done:

```
grammar:  ...
          | "#tokdefs"  QuotedTerm
            <<{
            zzantlr_state      st;    /* defined in antlr.h  */
            struct zzdlg_state  dst;   /*defined in dlgdef.h */
            FILE                    *f;
            UserTokenDefsFile = mystrdup(LATEXT(1));
            zzsave_antlr_state(&st);
            zzsave_dlg_state(&dst);
            f = fopen(StripQuotes(LATEXT(1)),"r");
            if ( f==NULL ) {
                warn(eMsg1("cannot open token defs file '%s'",
                           LATEXT(1)+1));}
            else {
```

```
        ANTLRm( enum_file(), f, PARSE_ENUM_FILE);
        UserDefdTokens = 1;
    }
    zzrestore_antlr_state(&st);
    zzrestore_dlg_state(&dst);
    }>>
```

The code uses zzsave_antlr_state() and zzsave_dlg_state() to save the state of the current parse. The ANTLRm macro specifies a starting rule for ANTLR of "enum_file" and starts DLG in the PARSE_ENUM_FILE state rather than the default state (which is the current state — whatever it might be). Because enum_file() is called without any arguments it appears that enum_file() does not use ASTs nor pass back any attributes. Contributed by TJP.

(C Mode) Attributes

#151. Use symbolic tags (rather than numbers) to refer to attributes and ASTs in rules

prior to version 1.30: `rule : X Y  <<printf("%s %s",$1,$2);>> ;`

with version 1.30:

```
rule : xx:X yy:Y<<printf("%s
                          %s",$xx,$yy);>> ;
```

#152. Rules no longer have attributes: `rule : r1:rule1 <<...$r1...;>>` won't work

Actually this still works if one restricts oneself to C mode and uses *numeric* labels like $1 and $2. However numeric labels are a deprecated feature, can't be used in C++ mode, and can't be used in the same source file as symbolic labels, so it's best to avoid them.

#153. Attributes are built automatically only for terminals

To construct attributes under any other circumstances one must use `$[`*TokenCode*`,...]` or zzcr_attr().

#154. How to access the text or token part of an attribute

The way to access the text, token, (or whatever) part of an attribute depends on the way the attribute is stored. If one uses the PCCTS supplied routine "pccts/h/charbuf.h" then:

```
id : "[a-z]+"        <<printf("Token is %s\n",$1.text);>> ;
```

If one uses the PCCTS supplied routine "pccts/h/charptr.c" and "pccts/h/charptr.h" then:

```
id : "[a-z]+"        <<printf("Token is %s\n",$1);>> ;
```

If one uses the PCCTS supplied routine "pccts/h/int.h" (which stores numbers only) then:

```
number : "[0-9]+"     <<printf ("Token is %d\n",$1);>> ;
```

(Note the use of "%d" rather than "%s" in the printf() format).

#155. The $0 and $$ constructs are no longer supported — use inheritance instead (Item #99)

#156. If you use attributes then define a zzd_attr() to release resources (memory) when an attribute is destroyed

#157. Don't pass automatically constructed attributes to an outer rule or sibling rule — they'll be out of scope

The PCCTS generated variables which contain automatically generated attributes go out of scope at the end of the rule or sub-rule that contains them. Of course you can copy the attribute to a variable that won't go out of scope. If the attribute contains a pointer which is copied (e.g. pccts/h/charptr.c) then extra caution is required because of the actions of zzd_attr(). See Item #158.

#158. A charptr.c attribute must be copied before being passed to a calling rule

The pccts/h/charptr.c routines use a pointer to a string. The string itself will go out of scope when the rule or sub-rule is exited. Why ? The string is copied to the heap when ANTLR calls the routine zzcr_attr() supplied by charptr.c — however ANTLR also calls the charptr.c supplied routine zzd_attr() (which frees the allocated string) as soon as the rule or sub-rule exits. The result is that in order to pass charptr.c strings to outer rules via inheritance it is necessary to make an independent copy of the string (using strdup for example) or else by zeroing the original pointer to prevent its deallocation.

#159. Attributes created in a rule should be assumed not valid on entry to a fail action

Fail action are "... executed after a syntax error is detected but before a message is printed and the attributes have been destroyed. However, attributes are not valid here because one does not know at what point the error occurred and which attributes even exist. Fail actions are often useful for cleaning up data structures or freeing memory." (Page 29 of 1.00 manual)

Example of a fail action:

```
a : <<List *p=NULL;>>
    ( v:Var <<append(p,$v);>> )+
        <<operateOn(p);rmlist(p);>>
  ; <<rmlist(p);>>
    ^^^^^^^^^^^^^^    <--- Fail Action
```

#160. Use a fail action to destroy temporary attributes when a rule fails

If you construct temporary, local, attributes in the middle of the recognition of a rule, remember to deallocate the structure should the rule fail. The code for failure goes after the ";" and before the next rule. For this reason it is sometimes desirable to defer some processing until the rule is recognized rather than the most convenient place:

```
#include "pccts/h/charptr.h"
;statement!
        : <<char *label=0;>>
          {name:ID COLON  <<label=MYstrdup($name);>> }
                 s:statement_without_label
                        <<#0=#(#[T_statement,label],#s);
                          if (label!=0) free(label);
                        >>
;<<if (label !=0) free(label);>>
```

In the above example attributes are handled by charptr.* (see the warning, Item #158). The call to MYstrdup() is necessary because $name will go out of scope at the end of the subrule "{name:ID COLON}". The routine written to construct ASTs from attributes (invoked by `#[int,char *]`) knows about this behavior and always makes a copy of the character string when it constructs the AST. This makes the copy created by the explicit call to MYstrdup redundant once the AST has been constructed. If the call to "statement_without_label" fails then the temporary copy must be deallocated.

#161. When you need more information for a token than just token type, text, and line number

Passing accurate column information along with the token in C mode when using syntactic predicates requires workarounds. P.A. Keller (P.A.Keller@bath.ac.uk) has worked around this limitation of C mode by passing the address of a user-defined struct (rendered as text using format codes "%p" or "%x") along with (or instead) of the token's actual text. This requires changes in syntax error routines and other places where the token text might be displayed.

#162. About the pipeline between DLG and ANTLR (C Mode)

I find it helpful to think of lexical processing by DLG as a process which fills a pipeline and of ANTLR as a process which empties a pipeline. (This relationship is exposed in C++ mode because of the ANTLRTokenBuffer class).

With LL_K=1 the pipeline is only one item deep, trivial, and invisible. It is invisible because one can make a decision in ANTLR to change the DLG #lexclass with zzmode() and have the next token (the one following the one just parsed by ANTLR) parsed according to the new #lexclass.

With LL_K>1 the pipeline is not invisible. DLG will put a number of tokens into the pipeline and ANTLR will analyze them in the same order. How many tokens are in the pipeline depends on options one has chosen.

Case 1: Infinite lookahead mode ("`(...)?`"). The pipeline is as huge as the input since the entire input is tokenized by DLG before ANTLR even begins analysis.

Case 2: Demand lookahead (interactive mode). There is a varying amount of lookahead depending on how much ANTLR thinks it needs to predict which rule to execute next. This may be zero tokens (or maybe it's one token) up to *k* tokens. Naturally, it takes extra work by ANTLR to keep track of how many tokens are in the pipe and how many are needed to parse the next rule.

Case 3: Normal mode. DLG stays exactly *k* tokens ahead of ANTLR. This is a half-truth. It rounds *k* up to the next power of 2 so that with k=3 it actually has a pipeline of 4 tokens. If one says "-k 3" the analysis is still *k*=3, but the pipeline size is rounded up because TJP decided it was faster to use a bit-wise "and" to compute the next position in a circular buffer rather than (n+1) mod k.

(C Mode) ASTs

#163. Define a zzd_ast() to recover resources when an AST is deleted

#164. How to place prototypes for routines using ASTs in the #header

Add #include "ast.h" after the #define AST_FIELDS and before any references to AST:

```
#define AST_FIELDS int token;char *text;
#include "ast.h"
#define zzcr_ast(ast,attr,tok,astText) \
        create_ast(ast,attr,tok,text)
void create_ast (AST *ast,Attr *attr,int tok,char *text);
```

#165. To free an AST tree use zzfree_ast() to recursively descend the AST tree and free all sub-trees

The user should supply a routine zzd_ast() to free any resources used by a single node — such as pointers to character strings allocated on the heap.

#166. Use #define zzAST_DOUBLE to add support for doubly linked ASTs

There is an option for doubly linked ASTs in the module ast.c. It is controlled by #define zzAST_DOUBLE. Even with zzAST_DOUBLE only the right and down fields are filled while the AST tree is constructed. Once the tree is constructed the user must call the routine zzdouble_link(tree,0,0) to traverse the tree and fill in the left and up fields.

Extended Examples and Short Descriptions of Distributed Source Code

Examples mentioned in these notes are available as .tar files at the sites mentioned in Item #1. In keeping with the restrictions in PCCTS, I have used neither templates nor multiple inheritance in these examples.

All these examples use AST classes and token classes which are derived from NoLeakAST and NoLeakToken respectively. These classes maintain a doubly-linked list of all ASTs (or tokens) which have been created but not yet deleted making it possible to recover memory for these objects.

#1. Modifications to pccts/dlg/output.c to add member functions and data to DLGLexer header

See files notes/changes/dlg/output*.c

This modification to output.c adds the following code to the DLGLexer class header:

```
#ifdef DLGLexerIncludeFile
#include DLGLexerIncludeFile
#endif
```

#2. DLG definitions for C and C++ comments, character literals, and string literals

See files in notes/cstuff/cstr.g (C mode) or notes/cstuff/cppstr.g (C++ mode). Contributed by John D. Mitchell (johnm@alumni.eecs.berkeley.edu).

#3. A simple floating point calculator implemented using PCCTS attributes and inheritance

This is the PCCTS equivalent of the approach used in the canonical yacc example. See notes/calctok/*.

#4. A simple floating point calculator implemented using PCCTS ASTs and C++ virtual functions

See notes/calcAST/*.

In this example an expression tree is built using ASTs. For each operator in the tree there is a different class derived from AST with an operator specific implementation of the virtual function "eval()". Evaluation of the expression is performed by calling eval() for the root node of the AST tree. Each node invokes eval() for its children nodes, computes its own operation, and passed the result to its parent in a recursive manner.

#5. An ANTLRToken class for variable length strings allocated from the heap

See files in notes/var/varToken.*

#6. How to extend PCCTS C++ classes using the example of adding column information

See files in notes/col/*

This demonstrates how to add column information to tokens and to produce syntax error messages using this information. This example derives classes from ANTLRToken and ANTLRTokenBuffer.

#7. How to pass whitespace through DLG for pretty-printers

See files in notes/ws/*

This demonstrates how to combine several separate DLG tokens (whitespace for this example) into a single ANTLR token. It also demonstrates careful processing of tab characters to generate accurate column information even within comments or other constructs which use more().

#8. How to prepend a newline to the DLGInputStream via derivation from DLGLexer

See files in notes/prependnl/*

This demonstrates how to derive from DLGLexer in order to replace a user-supplied DLGInputStream routine with another which can perform additional processing on the input character stream before the characters are passed to DLG. In this case a single newline is prepended to the input. This is done to make it easier to treat the first non-blank token on a line as a special case, even when it appears on the very first line of the input file.

#9. How to maintain a stack of #lexclass modes

See files in notes/modestack/*

This is based on routines written by David Seidel (dave@numega.com or 75342.2034@compuserve.com) which allow the user to pass a a routine to be executed when the mode is popped from the stack.

#10. Changes to pccts/h/DLexer.C to aid in debugging of DLG lexers as outlined in Item #126

See files in notes/changes/h/DLexer*.C

#11. AT&T Cfront compatible versions of some 1.32b6 files

See files in notes/changes/h/PCCTSAST*.*

#12. When you want to change the token type just before passing the token to the parser

See files in notes/predbuf/*

This program shows how to reassign token codes to tokens at the time they are fetched by the parser by deriving from class ANTLRTokenBuffer and changing the behavior of getToken().

#13. Rewriting a grammar to remove left recursion and perform left factoring

The original grammar:

```
command := SET var BECOMES expr
         |  SET var BECOMES QUOTE QUOTE
         |  SET var BECOMES QUOTE expr QUOT
         |  SET var BECOMES QUOTE command QUOTE

expr    := QUOTE anyCharButQuote QUOTE
         |  expr AddOp expr
         |  expr MulOp expr
```

The repetition of "SET var BECOMES" for command would require k=4 to get to the interesting part. The first step is to left-factor "command":

```
command := SET var BECOMES
                ( expr
                | QUOTE QUOTE
                | QUOTE expr QUOTE
                | QUOTE command QUOTE
                )
```

The definition of expr uses left recursion which must be eliminated when using ANTLR:

```
op      := AddOp
        |  MulOp
expr    := QUOTE anyCharButQuote QUOTE (op expr)*
```

Since expr begins with QUOTE and all the alternatives of the sub-rule of command also start with QUOTE. This too can be left-factored:

```
 command := SET var BECOMES QUOTE
                ( expr_suffix
                | QUOTE
                | expr QUOTE
                | command QUOTE
                )
expr_suffix := anyCharButQuote QUOTE (op expr)*
expr        := QUOTE expr_suffix
```

The final grammar can be built by ANTLR with k=2.

```
#token Q        "\""
#token SVB      "svb"            //  "SET var BECOMES"
#token Qbar     "[a-z A-Z]*"
#token AddOp    "\+"
#token MulOp    "\*"
#token WS       "\ "    <<zzskip();>>
#token NL       "\n"    <<zzskip();>>

repeat      : ( command )+ "@";
command     : SVB Q ( expr_suffix
                    | expr Q
                    | Q          <<printf("null command\n");>>
                    | command Q  <<printf("command\n");>>
                    );
expr_suffix : Qbar Q <<printf("The Qbar expr is (%s)\n",$1.text);>>
                 { op expr };
expr        : Q expr_suffix;
op          : AddOp | MulOp ;
```

#14. Using the GNU gperf (generate perfect hashing function) with PCCTS

The scanner generated by DLG can be very large. For grammars which contain a large number of keywords it might make sense to the use of the GNU program "gperf". The gperf programs attempts to generate a "minimal perfect hash function" for testing whether an argument is among a fixed set of strings such as those used in the reserved words of languages. It has a large number of options to specify space/time trade-offs

and the style of the code generated (e.g. C++ vs. C, case sensitivity, arrays vs. case statements, etc.).

As a test I found that a grammar with 25 keywords caused DLG to generate a file DLGLexer.C with 22,000 characters. Changing the lexical analysis code to use gperf resulted in a file DLGLexer.C that was 2,800 characters. The file generated by gperf was about 3,000 characters.

The gperf program was originally written by Douglas C. Schmidt. It is based on an algorithm developed by Keith Bostic. The gperf program is covered by the GNU General Public License. I do not know what restrictions there are on the output of gperf. The source code can be found in comp.sources.unix, volume 20.

Among the many FTP sites with comp.sources.unix here are two:

ftp.cis.ohio-state.edu/pub/comp.sources.unix/Volume20/gperf
ftp.informatik.tu-muenchen.de/pub/comp/usenet/comp.sources.unix/gperf

File: keywords.h

```
#ifndef KEYWORDS_H
#define KEYWORDS_H

#include "tokens.h"

struct Keywords {
   char *       name;
   ANTLRTokenType    tokenType;
};

Keywords * in_cli_word_set(const char *,int);

#endif
```

File: clikeywords.gperf:

```
%{
/*
  gperf -a -k 1,3 -H cliHash -N in_cli_word_set -ptT >clikeywords.C

        a - ansi prototypes
        k - character positions to use in hash
        H - override name of hash function
        N - override namae of in_word_set function
        p - return pointer to struct or 0
        t - use structure type declaration
        T - don't copy struct definition to output
*/

#include <string.h>
#include "keywords.h"

%}
Keywords;
%%
char,           CHAR
```

```
string,         STRING
...
void,           VOID
```

File: main grammar

```
...
#lexaction <<

#include "keywords.h"
#include <string.h>

Keywords *      pKeyword;
>>

#token  Eof     "@"
#token  CHAR    "char"
#token  STRING  "string"
...
#token  VOID    "void"
#token ID "[a-z A-Z]+"
        <<pKeyword=in_cli_word_set(lextext(),strlen(lextext()));
          if (pKeyword != 0) return pKeyword->tokenType;
        >>

class P {
...
}
```

#15. Processing counted strings in DLG

Sometimes literals are preceded by a count field.

```
3abc identifier 4defg
```

This example works by storing the count which precedes the string in a local variable and then switching to a #lexclass which accepts characters one at a time while decrementing a counter. When the counter reaches zero (or a newline in this example) the DLG routine switches back to the usual #lexclass.

```
...
#lexaction <<static int count;>>

#token HOLLERITH        "[0-9]*"
        <<count=atoi(lextext());
         printf("Count is %d\n",count);
         mode(COUNT);
        >>
#token Eof              "@"
#token ID               "[a-z]*"<<printf("ID is %s\n",lextext());>>
#token WS               "\ "      <<skip();>>
#token NL               "\n"

#lexclass COUNT
#token  STRING          "~[]"
        <<count--;
          if (count == 0) {
              mode(START);
```

```
                printf ("Hollerith string is \"%s\"\n",lextext());
            } else if (ch == '\n') {
                mode(START);
                printf("Hollerith string %s terminated by newline\n",
                  lextext());
            } else {
                more();
            };
          >>
class P {
  statement      : ( (HOLLERITH STRING | ID )* NL)+ "@";
}
```

See files in notes/hollerith/*

#16. How to convert a failed validation predicate into a signal for treatment by parser exception handling

See files notes/sim/*

This program intercepts a failed validation predicate in order to pass a signal back up the call tree. The example includes code which takes the signal returned by the start rule and invokes the default handler

This example is not as clean as I would like because of the difficulty of adding new behavior to a parser class.

#17. How to use Vern Paxson's flex with PCCTS in C++ mode by inheritance from ANTLRTokenStream

See files example.flex and flexLexer.* in notes/flex/*

Index

Symbols

A

B

C

D

E

F

G

O

P

R

S

T

U

V

W

Z