



ANTLR

jGuru

ANTLR

Reference Manual

Credits

Project Lead

[Terence Parr](#)

Support from

[jGuru.com](#)

Your View of the Java Universe

Help with initial coding

John Lilly, [Empathy Software](#)

C++ code generator by

[Peter Wells](#) and [Ric Klaren](#)

Sather code generator by

[Mika Illouz](#)

Infrastructure support from [Perforce](#):

The world's best source code control system

Substantial intellectual effort donated by

[John Mitchell](#)

[Scott Stanchfield](#)

[Jim Coker](#)

[Monty Zukowski](#) (*Streams, translation*)

[Chapman Flack](#) (*UNICODE, streams*)

[ParseView](#) parser debugger by

[Scott Stanchfield](#)

ANTLR Version 2.7.1

Version: \$Id: //depot/code/org.antlr/main/main/doc/index.html#8 \$

October 1, 2000

Check out the [ANTLR FAQ at jguru.com!](#)

Check out the [Bug Database](#) (ok, so it's out of date)

Check out <http://www.ANTLR.org>

ParseView debugger is [available](#). (for prior versions)

If you are looking for the previous main version (PCCTS 1.33) of ANTLR rather than the Java/C++/Sather version (2.x), see [Getting started with PCCTS](#).

Download [ANTLR 2.7.1](#).

[ANTLR 2.7.1 release notes](#)

[ANTLR Meta-Language](#)

- [Meta-Language Vocabulary](#)
- [Header Section](#)
- [Parser Class Definitions](#)
- [Lexical Analyzer Class Definitions](#)
- [Tree-parser Class Definitions](#)
- [Options Section](#)
- [Tokens Section](#)
- [Grammar Inheritance](#)
- [Rule Definitions](#)
- [Atomic Production elements](#)
- [Simple Production elements](#)
- [Production Element Operators](#)
- [Token Classes](#)
- [Predicates](#)
- [Element Labels](#)
- [EBNF Rule Elements](#)
- [Interpretation Of Semantic Actions](#)
- [Semantic Predicates](#)
- [Syntactic Predicates](#)
 - [Fixed depth lookahead and syntactic predicates](#)
- [ANTLR Meta-Language Grammar](#)

[Lexical Analysis with ANTLR](#)

- [Lexical Rules](#)
 - [Return values](#)
 - [Skipping characters](#)
 - [Distinguishing between lexer rules](#)
 - [Definition order and lexical ambiguities](#)
 - [Keywords and literals](#)
 - [Common prefixes](#)

- [Token definition files](#)
- [Character classes](#)
- [Token Attributes](#)
- [Lexical lookahead and the end-of-token symbol](#)
- [Scanning Binary Files](#)
- [Scanning Unicode Characters](#)
- [Manipulating Token Text and Objects](#)
 - [Manipulating the Text of a Lexical Rule](#)
 - [Token Object Creation](#)
 - [Heterogeneous Token Object Streams](#)
- [Filtering Input Streams](#)
 - [ANTLR Masquerading as SED](#)
- [Nongreedy Subrules](#)
 - [Greedy Subrules](#)
 - [Nongreedy Lexer Subrules](#)
 - [Limitations of Nongreedy Subrules](#)
- [Lexical States](#)
- [The End Of File Condition](#)
- [Case sensitivity](#)
- [Ignoring whitespace in the lexer](#)
- [Tracking Line Information](#)
- [Tracking Column Information](#)
- [Using Explicit Lookahead](#)
- [A Surprising Use of A Lexer: Parsing](#)
- [But... We've Always Used Automata For Lexical Analysis!](#)

[ANTLR Tree Parsers](#)

- [What's a tree parser?](#)
- [What kinds of trees can be parsed?](#)
- [Tree grammar rules](#)
 - [Syntactic predicates](#)
 - [Semantic predicates](#)
 - [An Example Tree Walker](#)
- [Transformations](#)
 - [An Example Tree Transformation](#)
- [Examining/Debugging ASTs](#)

Token Streams

- [Introduction](#)
- [Pass-Through Token Stream](#)
- [Token Stream Filtering](#)
- [Token Stream Splitting](#)
 - [Example](#)
 - [Filter Implementation](#)
 - [How To Use This Filter](#)
 - [Tree Construction](#)
 - [Garbage Collection Issues](#)
 - [Notes](#)
- [Token Stream Multiplexing \(aka "Lexer states"\)](#)
 - [Multiple Lexers](#)
 - [Lexers Sharing Same Character Stream](#)
 - [Parsing Multiplexed Token Streams](#)
 - [The Effect of Lookahead Upon Multiplexed Token Streams](#)
 - [Multiple Lexers Versus Calling Another Lexer Rule](#)
- [The Future](#)

Token Vocabularies

- [Introduction](#)
 - [How does ANTLR decide which vocabulary symbol gets what token type?](#)
 - [Why do token types start at 4?](#)
 - [What files associated with vocabulary does ANTLR generate?](#)
 - [How does ANTLR synchronize the symbol-type mappings between grammars in the same file and in different files?](#)
- [Grammar Inheritance and Vocabularies](#)
- [Recognizer Generation Order](#)
- [Tricky Vocabulary Stuff](#)

Error Handling and Recovery

- [ANTLR Exception Hierarchy](#)
- [Modifying Default Error Messages With Paraphrases](#)
- [Parser Exception Handling](#)
- [Specifying Parser Exception-Handlers](#)
- [Default Exception Handling in the Lexer](#)

Java Runtime Model

- [Programmer's Interface](#)
 - [What ANTLR generates](#)
- [Multiple Lexers/Parsers With Shared Input State](#)
- [Parser Implementation](#)
 - [Parser Class](#)
 - [Parser Methods](#)
 - [EBNF Subrules](#)
 - [Production Prediction](#)
 - [Production Element Recognition](#)
 - [Standard Classes](#)
- [Lexer Implementation](#)
 - [Lexer Form](#)
 - [Creating Your Own Lexer](#)
 - [Lexical Rules](#)
- [Token Objects](#)
- [Token Lookahead Buffer](#)

C++ Runtime model

- [C++ notes](#)
 - [Building the runtime](#)
 - [Getting C++ output](#)
 - [Changing the AST Type](#)
 - [Using Heterogeneous AST types](#)
 - [A grammar template](#)

Sather Runtime Model

- [Programmer's Interface](#)
 - [What ANTLR generates](#)
- [Multiple Lexers/Parsers With Shared Input State](#)
- [Parser Implementation](#)
 - [Parser Class](#)
 - [Parser Methods](#)
 - [EBNF Subrules](#)
 - [Production Prediction](#)
 - [Production Element Recognition](#)
- [Lexer Implementation](#)

- [Lexer Form](#)
- [Creating Your Own Lexer](#)
- [Lexical Rules](#)

[ANTLR Tree Construction](#)

- [Notation](#)
- [Controlling AST construction](#)
- [Grammar annotations for building ASTs](#)
 - [Leaf nodes](#)
 - [Root nodes](#)
 - [Turning off standard tree construction](#)
 - [Tree node construction](#)
 - [AST Action Translation](#)
- [Invoking parsers that build trees](#)
- [AST Factories](#)
- [Heterogeneous ASTs](#)
 - [An Expression Tree Example](#)
 - [Describing Heterogeneous Trees With Grammars](#)
- [AST \(XML\) Serialization](#)
- [AST enumerations](#)
- [A few examples](#)
- [Labeled subrules](#)
- [Reference nodes](#)
- [Required AST functionality and form](#)

[Grammar Inheritance](#)

- [Introduction and motivation](#)
- [Functionality](#)
- [Where Are Those Supergrammars?](#)
- [Error Messages](#)

[Options](#)

- [File, Grammar, and Rule Options](#)
 - [Options supported in ANTLR](#)
 - [language: Setting the generated language](#)
 - [k: Setting the lookahead depth](#)
 - [importVocab: Initial Grammar Vocabulary](#)

- [exportVocab: Naming Output Vocabulary](#)
- [testLiterals: Generate literal-testing code](#)
- [defaultErrorHandler: Controlling default exception-handling](#)
- [codeGenMakeSwitchThreshold: controlling code generation](#)
- [codeGenBitsetTestThreshold: controlling code generation](#)
- [buildAST: Automatic AST construction](#)
- [ASTLabelType: Setting label type](#)
- [charVocabulary: Setting the lexer character vocabulary](#)
- [warnWhenFollowAmbig](#)
- [Command Line Options](#)



ANTLR

jGuru

ANTLR 2.7.1

Release Notes

October 1, 2000

The ANTLR 2.7.1 release is a bug fix release, brought to you by those hip cats at jGuru.com. One of the bug fixes, however, allows UNICODE characters to be recognized for the first time. :)

Enhancements

ANTLR 2.7.1 has a few enhancements:

- ANTLR now allows UNICODE characters because Terence made case-statement expressions more efficient ;) See the unicode example in the distribution and the brief blurb in the [documentation](#).
- Massively improved C++ code generator (see below).
- Added automatic column setting support. See updated [doc](#) and new examples/java/columns directory.
- Ter added `throws` to tree and regular parsers .
- Added an `antlr/extras` directory, currently containing only `antlr-emacs.el` by Christoph.Wedler@sap-ag.de. Thanks, Christoph!

C++ Code Generation

Pete Wells and Ric Klaren have pretty much gutted the C++ code generator to use templates and so on. Here are few notes (with lib/cpp/Changelog having more goodies). Ric has totally worked his ass off to make the C++ what it is now! :)

Enhancements to C++ code generator for:

- * #line generation for easier debugging of action code. Turn on/off with option `genHashLines` (grammar option).
- * Cleaner generated code, by providing options to specify namespace prefixes. Grammar options `namespaceAntlr` and `namespaceStd` can be set to "antlr::" and "std::" or to blank if your compiler doesn't support namespaces.
- * Generate comments to explain what the bitsets represent.
- * Fix bug with `-traceTreeParser` code.
- * Avoid warnings about unused variable `_saveIndex`.
- * Remove final, illegal comma in token types enum.

Enhancements to C++ support library for:

- * Performance enhancements. Thanks to several people for suggestions/patches here. Improvements to memory management for building strings, and buffering of tokens.
- * Support for Metrowerks Codewarrior, and Sun CC 5.0.
- * Fix problem with multi-threaded lexers using static variable.
- * Slight tidy up (more planned).

Additionally, there have been enhancements made to the C++ side to mirror the Java side changes.

Ric Klaren (2.7.1a3 C++ changes) says:

- - action.g allow ':' in ID rule so C++ namespace qualifiers work.
- - CppCodegenerator the '::' fix for namespaceXXX options. As requested by Michael Schmitt.
- - Several cleanups in the Exception classes (basically a hoisting of code) and one or two new constructors with more line/column param's.
- - Default value for column in LexerInputState to 1. as suggested by someone on the list.. (name I would have to look up)
- - A makefile for the C++ lib directory. Not yet the autoconf stuff posted by someone (whose name I would also have to look up) it would imply a bigger workover of the lib/cpp directory. Which is harder to do with diff's.
- - Some changes I made after enabling the effective C++ warnings on g++ (minor drive basically.. in most places not really needed)
- - Several virtuals added to methods. Based on a suggestion also by Ernest Pasour. It makes the error messages from the thrown exceptions a lot better

Bug Fixes

In no particular order, here are the improvements/fixes made to 2.7.0 to arrive at 2.7.1 (via 2.7.1a1..a4):

- columns started at 0 for line 1. fixed.
- Bob McWhirter added -o fix so that antlr looks for import vocab stuff in -o director if not found in \$CWD (current working directory).
- Added optimization so that large unicode ranges don't result in giant switch case expressions. For example, added charVocabulary='\u0003..\uffff' to java.g. Took antlr 24s to generate 51k lexer file vs 9sec without. New 2.7.1 did it with big vocab in 14 sec. Oh, and the interesting thing is that with the big vocab and new optimization, it's actually smaller than with vocab set to ASCII. :)
- added a build script.
- Robert Colquhoun rjc@trump.net.au gave me a patch to pull stuff out of Tool.java that was causing it to be required for runtime even.
- Jerry James (james@eecs.ukans.edu) gave me a patch to make the labels for heterogeneous tree nodes match the specific AST type rather than plain AST.
- ANTLR didn't like curlyes in quotes (preproc.g was hosed). It now parses:

```
class A extends Parser;

tokens {
// hi |}
/*
fds
*}*/
TOK_LBRACE= " { ";
TOK_RBRACE= " } ";
}

a : "{ B "};
```

- Fixed C++ code generator to allow ~(Z|G)
- Parser.getInputState called setInputState.
- ANTLR now allows comments between header, options, and tokens and then '{' now. Examples:

```
options //fdkjfds
{
k = 1;
}
```

```
tokens //testing
{
A = "a";
}
```

- Made fields of CommonToken protected (open to subclasses), added col. added column tracking support; tabs are counted as 1 unless you override tab(). Called from consume(); bumps by one by default. Overhead is minimal; only called on tabs. extra increment for all consume()s now extra int in CommonToken.

```
/**
advance the current column number by an appropriate amount. If
you do not override this to specify how much to jump for a
tab, then tabs are counted as one char. This method is called
from consume().
*/
public void tab() {
// update inputState.column as function of
// inputState.column and tab stops.
// For example, if tab stops are columns 1
// and 5 etc... and column is 3, then add 2
// to column.
    inputState.column++;
}
```

- added CharScanner.setColumn
- warnings were going to stdout, make go to stderr.
- added check for unterminated rules. Labels in column 1 result in a warning.
- wasn't providing always exactly 4 digits for \u chars in JavaCharFormatter.escapeChar.
- Fixed that nasty follow cycle grammar analysis bug Tom Moog and others found.
- C++: CharScanner.cpp toLower, changed arg from char to int.
- added column support to C++ output
- Sather fixes put in, brought up to snuff with Java/C++.
- ANTLR continued on after discovering duplicate grammar. 'caused later exception.
- Bug fix: \$setType(w); didn't work because of the leading space.
- For the java.tree.g grammar: the NEW operator didn't allow an optional (objBlock)?
- HTML: Added lots of tweaks to html.g, Made blockquote handle nested content. Fixed bug in COMMENT_DATA that wouldn't let '-' appear in comment. Made COMMENT scarf WS after comment
- Added to runtime jars (bigger but too lazy to weed out unnecessary var refs that force inclusion):

```
antlr/DefineGrammarSymbols.class
antlr/ANTLRGrammarParseBehavior.class
antlr/MakeGrammar.class
antlr/ANTLRParser.class
antlr/ANTLRTokenTypes
antlr/LLkGrammarAnalyzer
antlr/GrammarAnalyzer
```

- Added constructors.

```
public CommonASTWithHiddenTokens() {
    super();
}

public CommonASTWithHiddenTokens(Token tok) {
    super(tok);
}
```

ANTLR Installation

ANTLR comes as a single zip or compressed tar file. Unzipping the file you receive will produce a directory called antlr-2.7.1 with subdirectories antlr, doc, examples, cpp, and examples.cpp. You need to place the antlr-2.7.1 directory in your CLASSPATH environment variable. For example, if you placed antlr-2.7.1 in directory /tools, you need to append

```
/tools/antlr-2.7.1
```

to your CLASSPATH or.

```
\tools\antlr-2.7.1
```

if you work on an NT or Win95 box.

References to antlr.* will map to /tools/antlr-2.7.1/antlr/*.class.

You must have at least JDK 1.1 installed properly on your machine. The ASTFrame AST viewer uses Swing 1.1.

JAR FILE

Try using the **runtime** library `antlr.jar` file. Place it in your CLASSPATH instead of the antlr-2.7.1 directory. The jar includes all **parse-time** files needed (if it is missing a file, email parrt@jguru.com) You cannot run the antlr tool itself with the jar, but your parsers should run with just this jar file. It's pretty small, around 75k uncompressed.

RUNNING ANTLR

ANTLR is a command line tool (although many development environments let you run ANTLR on grammar files from within the environment). The main method within antlr.Tool is the ANTLR entry point.

```
java antlr.Tool file.g
```

The command-line option is `-diagnostic`, which generates a text file for each output parser class that describes the lookahead sets. Note that there are number of options that you can specify at the grammar class and rule level.

Options `-trace`, `-traceParser`, `-traceTreeParser` may be used to track the lexer, parser, and tree parser invocations.

Try the new `-html` option to generate HTML output of your grammar(s); this is only partially done.

If you have trouble running ANTLR, ensure that you have Java installed correctly and then ensure that you have the appropriate CLASSPATH set.

Version: \$Id: //depot/code/org.antlr/release/antlr-2.7.1/doc/antlr271release.html#2 \$

**ANTLR****jGuru**

ANTLR 2 Meta-Language

ANTLR 2 accepts three types of grammar specifications -- parsers, lexers, and tree-parsers (also called tree-walkers). Because ANTLR 2 uses LL(k) analysis for all three grammar variants, the grammar specifications are similar, and the generated lexers and parsers behave similarly.

Note: in this document, the word "parser" usually includes tree-parsers as well as token stream parsers, except where noted.

Meta-Language Vocabulary

Whitespace. Spaces, tabs, and newlines are separators in that they can separate ANTLR vocabulary symbols such as identifiers, but are ignored beyond that. For example, "FirstName LastName" appears as a sequence of two token references to ANTLR not token reference, space, followed by token reference.

Comments. ANTLR accepts C-style block comments and C++-style line comments. Java-style documenting comments are allowed on grammar classes and rules, which are passed to the HTML output if requested. For example,

```
/**This grammar recognizes simple expressions
 * @author Terence Parr
 */
class ExprParser;

/**Match a factor */
factor : ... ;
```

Characters. Character literals are specified just like in Java. They may contain octal-escape characters (e.g., '\377'), Unicode sequences (e.g., '\uFFFF'), and the usual special character escapes recognized by Java ('\b', '\r', '\t', '\n', '\f', '\', '\'). In lexer rules, single quotes represent a character to be matched on the input character stream. Single-quoted characters are not yet supported in parser rules.

End of file. The EOF token is automatically generated for use in parser rules:

```
rule : (statement)+ EOF;
```

You can test for EOF_CHAR in actions of lexer rules:

```
// make sure nothing but newline or
// EOF is past the #endif
ENDIF
{
    boolean eol=false;
}
:    "#endif"
    ( ('\n' | '\r') {eol=true;} )?
    {
        if (!eol) {
            if (LA(1)==EOF_CHAR) {error("EOF");}
            else {error("Invalid chars");}
        }
    }
```

```

    }
  }
;

```

While you can test for end-of-file as a character, it is not really a character--it is a condition. You should instead override `CharScanner.uponEOF()`, in your lexer grammar:

```

/** This method is called by YourLexer.nextToken()
 *  when the lexer has
 *  hit EOF condition. EOF is NOT a character.
 *  This method is not called if EOF is reached
 *  during syntactic predicate evaluation or during
 *  evaluation of normal lexical rules, which
 *  presumably would be an IOException. This
 *  traps the "normal" EOF * condition.
 *
 *  uponEOF() is called after the complete evaluation
 *  of the previous token and only if your parser asks
 *  for another token beyond that last non-EOF token.
 *
 *  You might want to throw token or char stream
 *  exceptions like: "Heh, premature eof" or a retry
 *  stream exception ("I found the end of this file,
 *  go back to referencing file").
 */
public void uponEOF()
    throws TokenStreamException, CharStreamException
{
}

```

The end-of-file situation is a bit nutty (version 2.7.1) because Terence used `-1` as a char not an int (`-1` is `\uFFFF`...oops). Terence will fix as we get better at Unicode.

Strings. String literals are sequences of characters enclosed in double quotes. The characters in the string may be represented using the same escapes (octal, Unicode, etc.) that are valid in character literals.

In lexer rules, strings are interpreted as sequences of characters to be matched on the input character stream (e.g., `"for"` is equivalent to `'f' 'o' 'r'`).

In parser rules, strings represent tokens, and each unique string is assigned a token type. However, ANTLR does not create lexer rules to match the strings. Instead, ANTLR enters the strings into a literals table in the associated lexer. ANTLR will generate code to test the text of each token against the literals table, and change the token type when a match is encountered before handing the token off to the parser. You may also perform the test manually -- the automatic code-generation is controllable by a [lexer option](#).

You may want to use the token type value of a string literal in your actions, for example in the synchronization part of an error-handler. For string literals that consist of alphabetic characters only, the string literal value will be a constant with a name like `LITERAL_xxx`, where `xxx` is the name of the token. For example, the literal `"return"` will have an associated value of `LITERAL_return`. You may also assign a specific label to a literal using the `tokens` section.

Token references. Identifiers beginning with an uppercase letter are token references. The subsequent characters may be any letter, digit, or underscore. A token reference in a parser rule results in matching the specified token. A token reference in a lexer rule results in a call to the lexer rule for matching the characters of the token. In other words, token references in the lexer are treated as rule references.

Token definitions. Token definitions in a lexer have the same syntax as parser rule definitions, but refer to tokens, not parser rules. For example,

```
class MyParser extends Parser;
idList : ( ID )+; // parser rule definition

class MyLexer extends Lexer;
ID : ( 'a'..'z' )+ ; // token definition
```

Rule references. Identifiers beginning with a lowercase letter are references to ANTLR parser rules. The subsequent characters may be any letter, digit, or underscore. Lexical rules may not reference parser rules.

Actions. Character sequences enclosed in (possibly nested) curly braces are semantic actions. Curly braces within string and character literals are not action delimiters.

Arguments Actions. Character sequences in (possibly nested) square brackets are rule argument actions. Square braces within string and character literals are not action delimiters. The arguments within [] are specified using the syntax of the generated language, and should be separated by commas.

```
codeBlock
[int scope, String name] // input arguments
returns [int x]          // return values
: ... ;
```

```
a {int y}:
// pass 2 args, get return y=cblock[1,"John"]
```

Many people would prefer that we use normal parentheses for arguments, but parentheses are best used as grammatical grouping symbols for EBNF.

Symbols. The following table summarizes punctuation and keywords in ANTLR.

Symbol	Description
(...)	subrule
(...)*	closure subrule
(...)+	positive closure subrule
(...)?	optional
{ ... }	semantic action
[...]	rule arguments
{ ... }?	semantic predicate
(...)=>	syntactic predicate

	alternative operator
..	range operator
~	not operator
.	wildcard
=	assignment operator
:	label operator, rule start
;	rule end
<...>	element option
class	grammar class
extends	specifies grammar base class
returns	specifies return type of rule
options	options section
tokens	tokens section
header	header section
tokens	token definition section

Header Section

A header section contains source code that must be placed before any ANTLR-generated code in the output parser. This is mainly useful for C or C++ output due to their requirement that elements be declared before being referenced. In Java, this can be used to specify a package for the resulting parser, and any imported classes. A header section looks like:

```
header {
    source code in the language generated by ANTLR;
}
```

The header section is the first section in a grammar file.

Parser Class Definitions

All parser rules must be associated with a parser class. A grammar (.g) file may contain an arbitrary number of parser class definitions (along with lexers and tree-parsers). Each parser class specification precedes the options and rule definitions of the parser. A parser specification in a grammar file often looks like:

```
{ optional class preamble }
class YourParserClass extends Parser;
options
tokens...
parser rules...
```

When generating code in an object-oriented language, parser classes result in classes in the

output, and rules become member methods of the class. In C, classes would result in structs, and some name-mangling would be used to make the resulting rule functions globally unique.

The optional class preamble is some arbitrary text enclosed in `{}`. The preamble, if it exists, will be output to the generated class file immediately before the definition of the class.

Enclosing curly braces are not used to delimit the class because it is hard to associate the trailing right curly brace at the bottom of a file with the left curly brace at the top of the file. Instead, a parser class is assumed to continue until the next `class` statement.

Lexical Analyzer Class Definitions

A parser class results in parser objects that know how to apply the associated grammatical structure to an input stream of tokens. To perform lexical analysis, you need to specify a lexer class that describes how to break up the input character stream into a stream of tokens. The syntax is similar to that of a parser class:

```
{ optional class preamble }
class YourLexerClass extends Lexer;
options...
tokens...
lexer rules...
```

Lexical rules contained within a lexer class become member methods in the generated class. Each grammar (.g) file may contain an arbitrary number of lexer classes. The parser and lexer classes may appear in any order.

The optional class preamble is some arbitrary text enclosed in `{}`. The preamble, if it exists, will be output to the generated class file immediately before the definition of the class.

Tree-parser Class Definitions

A tree-parser is like a parser, except that it processes a two-dimensional tree of AST nodes instead of a one-dimensional stream of tokens. Tree parsers are specified identically to parsers, except that the rule definitions may contain a [special form](#) to indicate descent into the tree.

```
{ optional class preamble }
class YourTreeParserClass extends TreeParser;
options
tokens...
tree parser rules...
```

Options Section

Rather than have the programmer specify a bunch of command-line arguments to the parser generator, an [options section](#) within the grammar itself serves this purpose. This solution is preferable because it associates the required options with the grammar rather than ANTLR invocation. The section is preceded by the `options` keyword and contains a series of option/value assignments. An options section may be specified on both a per-file, per-grammar, and per-rule basis.

You may also specify an option on an element, such as a token, reference.

Tokens Section

If you need to define an "imaginary" token, one that has no corresponding real input symbol, use the tokens section to define them. Imaginary tokens are used often for tree nodes that mark or group a subtree resulting from real input. For example, you may decide to have an `EXPR` node be the root of every expression subtree and `DECL` for declaration subtrees for easy reference during tree walking. Because there is no corresponding input symbol for `EXPR`, you cannot reference it in the grammar to implicitly define it. Use the following to define those imaginary tokens.

```
tokens {
    EXPR;
    DECL;
}
```

The formal syntax is:

```
tokenSpec : "tokens" LCURLY
           (tokenItem SEMI)+
           RCURLY
           ;

tokenItem : TOKEN ASSIGN STRING (tokensSpecOptions)?
          | TOKEN (tokensSpecOptions)?
          | STRING (tokensSpecOptions)?
          ;

tokensSpecOptions
: "<"
  id ASSIGN optionValue
  ( SEMI id ASSIGN optionValue )*
">"
;
```

You can also define literals in this section and, most importantly, assign to them a valid label as in the following example.

```
tokens {
    KEYWORD_VOID="void";
    EXPR;
    DECL;
    INT="int";
}
```

Strings defined in this way are treated just as if you had referenced them in the parser.

If a grammar imports a vocabulary containing a token, say *T*, then you may attach a literal to that token type simply by adding `T="a literal"` to the tokens section of the grammar. Similarly, if the imported vocabulary defines a literal, say `"_int32"`, without a label, you may attach a label via `INT32="_int32"` in the tokens section.

You may define options on the tokens defined in the `tokens` section. The only option available so far is `AST=class-type-to-instantiate`.

```
// Define a bunch of specific AST nodes to build.
// Can override at actual reference of tokens in
// grammar.
tokens {
    PLUS<AST=PLUSNode>;
    STAR<AST=MULTNode>;
}
```

Grammar Inheritance

Object-oriented programming languages such as C++ and Java allow you to define a new object as it differs from an existing object, which provides a number of benefits. "Programming by difference" saves development/testing time and future changes to the *base* or *superclass* are automatically propagated to the *derived* or *subclass*. ANTLR 2.x supports [grammar inheritance](#) as a mechanism for creating a new grammar class based on a base class. Both the grammatical structure and the actions associated with the grammar may be altered independently.

Rule Definitions

Because ANTLR considers lexical analysis to be parsing on a character stream, both lexer and parser rules may be discussed simultaneously. When speaking generically about rules, we will use the term *atom* to mean an element from the input stream (be they characters or tokens).

The structure of an input stream of atoms is specified by a set of mutually-referential rules. Each rule has a name, optionally a set of arguments, optionally a "throws" clause, optionally an init-action, optionally a return value, and an alternative or alternatives. Each alternative contains a series of elements that specify what to match and where.

The basic form of an ANTLR rule is:

```
rulename
    : alternative_1
    | alternative_2
    ...
    | alternative_n
    ;
```

If parameters are required for the rule, use the following form:

```
rulename[formal parameters] : ... ;
```

If you want to return a value from the rule, use the `returns` keyword:

```
rulename returns [type id, type id...] : ... ;
```

where *type* is a type specifier of the generated language, and *id* is a valid identifier of the generated language. In Java, a single type identifier would suffice most of the time, but returning an array of strings, for example, would require brackets:

```
ids returns [String[] s]: ( ID { ... } ) * ;
```

Also, when generating C or C++, the return type could be complex such as:

```
ids returns [char *[] s]: ... ;
```

The ids of the returns statement will be passed to the output code. An action may assign directly to these ids to affect the return values.

To specify that your parser (or tree parser rule) can throw a non-ANTLR specific exception, use the exceptions clause. For example, here is a simple parser specification with a rule that throws MyException:

```
class P extends Parser;

a throws MyException
  : A
  ;
```

ANTLR generates the following for rule a:

```
public final void a()
    throws RecognitionException,
           TokenStreamException,
           MyException
{
    try {
        match(A);
    }
    catch (RecognitionException ex) {
        reportError(ex);
        consume();
        consumeUntil(_tokenSet_0);
    }
}
```

Lexer rules may not specify exceptions as of ANTLR 2.7.1.

Init-actions are specified before the colon. Init-actions differ from normal actions because they are always executed regardless of guess mode. In addition, they are suitable for variable declarations in languages like C where all declarations must be at the start of the rule.

```
rule
{
    init-action
}
: ...
;
```

Lexer rules. Rules defined within a lexer grammar must have a name beginning with an uppercase letter. These rules implicitly match characters on the input stream instead of tokens on the token stream. Referenced grammar elements include token references (implicit lexer rule references), characters, and strings. Lexer rules are processed in the exact same manner as parser rules and, hence, may specify arguments and return values; further, lexer rules can also have local variables and use recursion. See more about [lexical analysis with ANTLR](http://www.antlr.org/doc/metalang.html).

Parser rules. Parser rules apply structure to a stream of tokens whereas lexer rules apply structure to a stream of characters. Parser rules, therefore, must not reference character literals. Double-quoted strings in parser rules are considered token references and force ANTLR to squirrel away the string literal into a table that can be checked by actions in the associated lexer.

All parser rules must begin with lowercase letters.

Tree-parser rules. In a tree-parser, an additional special syntax is allowed to specify the match of a two-dimensional structure. Whereas a parser rule may look like:

```
rule : A B C;
```

which means "match A B and C sequentially", a tree-parser rule may also use the syntax:

```
rule : #(A B C);
```

which means "match a node of type A, and then descend into its list of children and match B and C". This notation can be nested arbitrarily, using #(...) anywhere an EBNF construct could be used, for example:

```
rule : #(A B #(C D (E)*) );
```

Atomic Production elements

Character literal. A character literal can only be referred to within a lexer rule. The single character is matched on the character input stream. There are no need to escape regular expression meta symbols because regular expressions are not used to match lexical atoms. For example, ' { ' need not have an escape as you are specifying the literal character to match. Meta symbols are used outside of characters and string literals to specify lexical structure.

String literal. Referring to a string literal within a parser rule defines a token type for the string literal, and causes the string literal to be placed in a hash table of the associated lexer. The associated lexer must check the hash table in an action. References to string literals within the parser may be suffixed with an element option; see token references below.

Referring to a string within a lexer rule matches the indicated sequence of characters and is a shorthand notation. For example, consider the following lexer rule definition:

```
BEGIN : "begin" ;
```

This rule can be rewritten in a functionally equivalent manner:

```
BEGIN : 'b' 'e' 'g' 'i' 'n' ;
```

There are no need to escape regular expression meta symbols because regular expressions are not used to match characters in the lexer.

Token reference. Referencing a token in a parser rule implies that you want to recognize a token with the specified token type. This does not actually call the associated lexer rule--the lexical analysis phase delivers a stream of tokens to the parser.

A token reference within a lexer rule implies a method call to that rule, and carries the same analysis semantics as a rule reference within a parser. In this situation, you may specify rule arguments and return values. See the next section on rule references.

You may also specify an option on a token reference. Currently, you can only specify the AST node type to create from the token. For example, the following rule instructs ANTLR to build `INTNode` objects from the `INT` reference:

```
i : INT<AST=INTNode> ;
```

The syntax of an element option is

```
<option=value; option=value; ...>
```

Wildcard. The "." wildcard within a parser rule matches any single token; within a lexer rule it matches any single character. For example, this matches any two tokens between the B and C:

```
r : A B . . C;
```

Simple Production elements

Rule reference. Referencing a rule implies a method call to that rule at that point in the parse. You may pass parameters and obtain return values. For example, formal and actual parameters are specified within square brackets:

```
funcdef
    :   type ID "(" args ")" block[1]
    ;
block[int scope]
    :   "{" ... { /*use arg scope/* } "}"
    ;
```

Return values that are stored into variables use a simple assignment notation:

```
set
{ Vector ids=null; } // init-action
:   "(" ids=idList ")"
;
idList returns [Vector strs]
{ strs = new Vector(); } // init-action
:   id:ID
    { strs.appendElement(id.getText()); }
    (
        ", " id2:ID
        { strs.appendElement(id2.getText()); }
    )*
;
```

Semantic action. Actions are blocks of source code (expressed in the target language) enclosed in curly braces. The code is executed after the preceding production element has been recognized and before the recognition of the following element. Actions are typically used to generate output, construct trees, or modify a symbol table. An action's position dictates when it is recognized relative to the surrounding grammar elements.

If the action is the first element of a production, it is executed before any other element in that production, but only if that production is predicted by the lookahead.

The first action of an EBNF subrule may be followed by ':'. Doing so designates the action as an *init-action* and associates it with the subrule as a whole, instead of any production. It is

executed immediately upon entering the subrule -- before lookahead prediction for the alternates of the subrule -- and is executed even while guessing (testing syntactic predicates). For example:

```
(
  {init-action}:
  {action of 1st production} production_1
|  {action of 2nd production} production_2
)?
```

The init-action would be executed regardless of what (if anything) matched in the optional subrule.

Production Element Operators

Element complement. The "~" not unary operator must be applied to an atomic element such as a token identifier. For some token atom T , $\sim T$ matches any token other than T except end-of-file. Within lexer rules, $\sim 'a'$ matches any character other than character $'a'$. The sequence $\sim .$ ("not anything") is meaningless and not allowed.

The vocabulary space is very important for this operator. In parsers, the complete list of token types is known to ANTLR and, hence, ANTLR simply clones that set and clears the indicated element. For characters, you must specify the character vocabulary; the only other choice is to add elements 0..255 and clear the indicated character. For Unicode, complementing a character would mean creating a set with 2^{16} elements unless the programmer specified the vocabulary. The character vocabulary includes the characters specified in the `charVocabulary` option and any characters referenced in the lexer rules. Here is a sample use of the character vocabulary option:

```
class L extends Lexer;
options { charVocabulary = '\1'..'377'; }

DIGIT : '0'..'9';
SL_COMMENT : "//" (~'\n')* '\n';
```

Set complement. the not operator can also be used to construct a token set or character set by *complementing* another set. This is most useful when you want to match tokens or characters until a certain delimiter set is encountered. Rather than invent a special syntax for such sets, ANTLR allows the placement of \sim in front of a subrule containing only simple elements and no actions. In this specific case, ANTLR will not generate a subrule, and will instead create a set-match. The simple elements may be token references, token ranges, character literals, or character ranges. For example:

```
class P extends Parser;
r : T1 (~ (T1 | T2 | T3))* (T1 | T2 | T3);

class L extends Lexer;
SL_COMMENT : "//" (~ ('\n' | '\r'))* ('\n' | '\r');

STRING : '"' (ESC | ~ ('\\" | '"'))* '"';
protected ESC : '\\ ('n' | 'r');
```

Range operator. The range binary operator implies a range of atoms may be matched. The expression $'c1' .. 'c2'$ in a lexer matches characters inclusively in that range. The expression $T .. U$ in a parser matches any token whose token type is inclusively in that

range, which is of dubious value unless the token types are generated externally.

AST root operator. When generating abstract syntax trees (ASTs), token references suffixed with the "^" root operator force AST nodes to be created and added as the root of the current tree. This symbol is only effective when the [buildAST option](#) is set. [More information](#) about ASTs is also available.

AST exclude operator. When generating abstract syntax trees, token references suffixed with the "!" exclude operator are not included in the AST constructed for that rule. Rule references can also be suffixed with the exclude operator, which implies that, while the tree for the referenced rule is constructed, it is not linked into the tree for the referencing rule. This symbol is only effective when the [buildAST option](#) is set. [More information](#) about ASTs is also available.

Token Classes

By using a range operator, a not operator, or a subrule with purely atomic elements, you implicitly define an "anonymous" token or character class--a set that is very efficient in time and space. ANTLR will soon let you define explicit or labeled token/character classes. For example, you can define a lexer rule such as:

```
OPS : ( PLUS | MINUS | MULT | DIV ) ;
```

or

```
WS : ( ' ' | '\n' | '\t' ) ;
```

These describe sets of tokens and characters respectively that are easily optimized to simple, single, bit-set rather than series of token and character comparisons.

Predicates

Semantic predicate. Semantics predicates are conditions that must be met at parse-time before parsing can continue past them. The functionality of [semantic predicates](#) is explained in more detail later. The syntax of a semantic predicate is a semantic action suffixed by a question operator:

```
{ expression }?
```

The expression must not have side-effects and must evaluate to true or false (`boolean` in Java, `int` in C, or `bool` in C++). Since semantic predicates can be executed while guessing, they should not rely upon the results of actions or rule parameters.

Syntactic predicate. Syntactic predicates specify the lookahead language needed to predict an alternative. [Syntactic predicates](#) are explained in more detail later. The syntax of a syntactic predicate is a subrule with a `=>` operator suffix:

```
( lookahead-language ) => production
```

Where the lookahead-language can be any valid ANTLR construct including references to other rules. Actions are not executed, however, during the evaluation of a syntactic predicate.

Element Labels

Any atomic or rule reference production element can be labeled with an identifier (case not significant). In the case of a labeled atomic element, the identifier is used within a semantic action to access the associated Token object or character. For example,

```
assign
  :   v:ID "=" expr ";"
    { System.out.println(
      "assign to "+v.getText()); }
  ;
```

No "\$" operator is needed to reference the label from within an action as was the case with ANTLR 1.xx.

The AST node constructed for a token reference or rule reference may be accessed from within actions as *label_AST*.

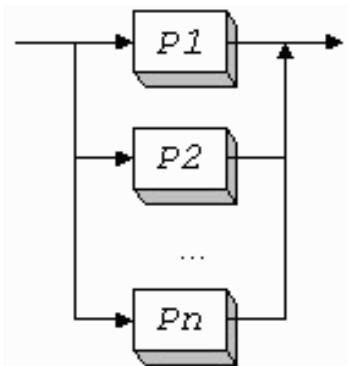
Labels on token references can also be used in association with parser [exception handlers](#) to specify what happens when that token cannot be matched.

Labels on rule references are used for [parser exception handling](#) so that any exceptions generated while executing the labeled rule can be caught.

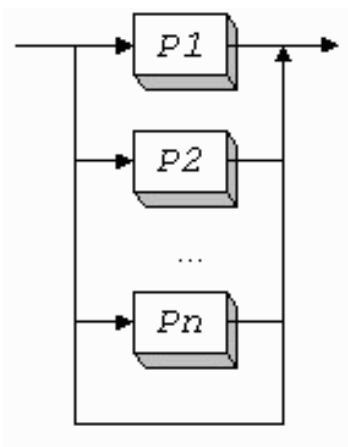
EBNF Rule Elements

ANTLR supports extended BNF notation according to the following four subrule syntax / syntax diagrams:

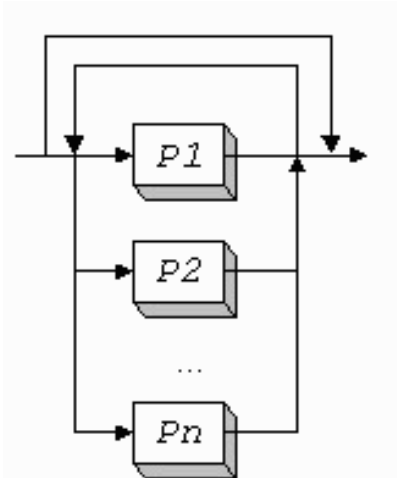
(*P1* | *P2* | ... | *Pn*)



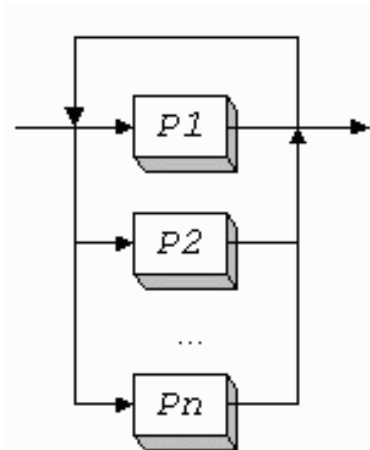
(*P1* | *P2* | ... | *Pn*)?



$(P1 \mid P2 \mid \dots \mid Pn)^*$



$(P1 \mid P2 \mid \dots \mid Pn)^+$



Interpretation Of Semantic Actions

Semantic actions are copied to the appropriate position in the output parser verbatim with the exception of [AST action translation](#).

None of the $\$$ -variable notation from ANTLR 1.xx carries forward into ANTLR 2.00.

Semantic Predicates

A semantic predicate specifies a condition that must be met (at run-time) before parsing may proceed. We differentiate between two types of semantic predicates: (i) *validating* predicates that throw exceptions if their conditions are not met while parsing a production and (ii) *disambiguating* predicates that are *hoisted* into the prediction expression for the associated production.

Semantic predicates are syntactically semantic actions suffixed with a question mark operator:

`{ semantic-predicate-expression }?`

The expression may use any symbol provided by the programmer or generated by ANTLR that is visible at the point in the output the expression appears.

The position of a predicate within a production determines which type of predicate it is. For

example, consider the following validating predicate (which appear at any non-left-edge position) that ensures an identifier is semantically a type name:

```
decl: "var" ID ":" t:ID
      { isTypeName(t.getText()) }?
      ;
```

Validating predicates generate parser exceptions when they fail. The thrown exception is of type `SemanticException`. You can catch this and other parser exceptions in an [exception handler](#).

Disambiguating predicates are always the first element in a production because they cannot be hoisted over actions, token, or rule references. For example, the first production of the following rule has a disambiguating predicate that would be hoisted into the prediction expression for the first alternative:

```
stat: // declaration "type varName;"
      {isTypeName(LT(1))}? ID ID ";"
      | ID "=" expr ";" // assignment
      ;
```

If we restrict this grammar to LL(1), it is syntactically nondeterministic because of the common left-prefix: ID. However, the semantic predicate correctly provides additional information that disambiguates the parsing decision. The parsing logic would be:

```
if ( LA(1)==ID && isTypeName(LT(1)) ) {
    match production one
}
else if ( LA(1)==ID ) {
    match production one
}
else error
```

ANTLR provides a *guarded predicate* to allow you to specify the lookahead context under which a predicate should be evaluated. The syntax is:

```
(lookahead-context-for-predicate)=>{predicate}?
```

A guarded predicate is useful in situations where the semantic predicate should be hoisted into the prediction decision only when the lookahead is consistent with some context. For example:

```
a : (ID)=>{isType(LT(1))}? (ID|INT)
  | ID
  ;
```

Here, the predicate is only applicable when an ID is found on the input stream. It should not be evaluated when an INT is found.

Formally, in ANTLR 1.xx, semantic predicates represented the semantic context of a production. As such, the semantic AND syntactic context (lookahead) could be hoisted into other rules. In the public-domain version of ANTLR 2.00, predicates are not currently be hoisted outside of their enclosing rule. Consequently, rules such as:

```
type : {isType(t)}? ID ;
```

are meaningless. On the other hand, this "semantic context" feature caused considerable confusion to many ANTLR 1.xx folks.

Syntactic Predicates

There are occasionally parsing decisions that cannot be rendered deterministic with finite lookahead. For example:

```
a    :    ( A )+ B
      |    ( A )+ C
      ;
```

The common left-prefix renders these two productions nondeterministic in the LL(k) sense for any value of k. Clearly, these two productions can be *left-factored* into:

```
a    :    ( A )+ ( B | C )
      ;
```

without changing the recognized language. However, when actions are embedded in grammars, left-factoring is not always possible. Further, left-factoring and other grammatical manipulations do not result in natural (readable) grammars.

The solution is simply to use arbitrary lookahead in the few cases where finite LL(k) for $k > 1$ is insufficient. ANTLR allows you to specify a lookahead language with possibly infinite strings using the following syntax:

```
( prediction block ) => production
```

For example, consider the following rule that distinguishes between sets (comma-separated lists of words) and parallel assignments (one list assigned to another):

```
stat:    ( list "=" )=> list "=" list
        |    list
        ;
```

If a `list` followed by an assignment operator is found on the input stream, the first production is predicted. If not, the second alternative production is attempted.

Syntactic predicates are a form of selective backtracking and, therefore, actions are turned off while evaluating a syntactic predicate so that actions do not have to be undone.

Syntactic predicates are implemented using exceptions in the target language if they exist. When generating C code, `long jmp` would have to be used.

We could have chosen to simply use arbitrary lookahead for any non-LL(k) decision found in a grammar. However, making the arbitrary lookahead explicit in the grammar is useful because you don't have to guess what the parser will be doing. Most importantly, there are language constructs that are ambiguous for which there exists no deterministic grammar! For example, the infamous if-then-else construct has no LL(k) grammar for any k. The following grammar is ambiguous and, hence, nondeterministic:

```
stat:    "if" expr "then" stat ( "else" stat )?
        |    ...
        ;
```

Given a choice between two productions in a nondeterministic decision, we simply choose

the first one. This works out well in most situations. Forcing this decision to use arbitrary lookahead would simply slow the parse down.

Fixed depth lookahead and syntactic predicates

ANTLR cannot be sure what lookahead can follow a syntactic predicate (the only logical possibility is whatever follows the alternative predicted by the predicate, but erroneous input and so on complicates this), hence, ANTLR assumes anything can follow. This situation is similar to the computation of lexical lookahead when it hits the end of the token rule definition.

Consider a predicate with a (...) * whose implicit exit branch forces a computation attempt on what follows the loop, which is the end of the syntactic predicate in this case.

```
class parse extends Parser;
a      :      (A (P)*) => A (P)*
      |      A
      ;
```

The lookahead is artificially set to "any token" for the exit branch. Normally, the P and the "any token" would conflict, but ANTLR knows that what you mean is to match a bunch of P tokens if they are present--no warning is generated.

If more than one path can lead to the end of the predicate in any one decision, ANTLR will generate a warning. The following rule results in two warnings.

```
class parse extends Parser;
a      :      (A (P|)*) => A (P)*
      |      A
      ;
```

The empty alternative can indirectly be the start of the loop and, hence, conflicts with the P. Further, ANTLR detects the problem that two paths reach end of predicate. The resulting parser will compile but never terminate the (P|)* loop.

The situation is complicated by k>1 lookahead. When the nth lookahead depth reaches the end of the predicate, it records the fact and then code generation ignores the lookahead for that depth.

```
class parse extends Parser;
options {
    k=2;
}
a      :      (A (P B|P)*) => A (P)*
      |      A
      ;
```

ANTLR generates a decision of the following form inside the (...) * of the predicate:

```
if ((la_1==P) && (la_2==B)) {
    match(P);
    match(B);
}
else if ((la_1==P) && (true)) {
    match(P);
}
```

```
}  
else {  
    break _loop4;  
}
```

This computation works in all grammar types.

ANTLR 2.x.x Meta-Language Grammar

See `antlr/antlr.g` for the grammar that describes ANTLR 2.x.x input grammar syntax in ANTLR 2.x.x meta-language itself.

Version: \$Id: //depot/code/org.antlr/test/antlr-2.7.0a11/doc/metalang.html#3 \$



ANTLR

jGuru

Lexical Analysis with ANTLR

A *lexer* (often called a scanner) breaks up an input stream of characters into vocabulary symbols for a parser, which applies a grammatical structure to that symbol stream. Because ANTLR employs the same recognition mechanism for lexing, parsing, and tree parsing, ANTLR-generated lexers are much stronger than DFA-based lexers such as those generated by DLG (from PCCTS 1.33) and `lex`.

The increase in lexing power comes at the cost of some inconvenience in lexer specification and indeed requires a serious shift your thoughts about lexical analysis. See a [comparison of LL\(k\) and DFA-based lexical analysis](#).

ANTLR generates predicated-LL(k) lexers, which means that you can have semantic and syntactic predicates and use $k > 1$ lookahead. The other advantages are:

- You can actually read and debug the output as its very similar to what you would build by hand.
- The syntax for specifying lexical structure is the same for lexers, parsers, and tree parsers.
- You can have actions executed during the recognition of a single token.
- You can recognize complicated tokens such as HTML tags or "executable" comments like the javadoc @-tags inside `/** ... */` comments. The lexer has a stack, unlike a DFA, so you can match nested structures such as nested comments.

The overall structure of a lexer is:

```
class MyLexer extends Lexer;
options {
    some options
}
{
    lexer class members
}
lexical rules
```

Lexical Rules

Rules defined within a lexer grammar must have a name beginning with an uppercase letter. These rules implicitly match characters on the input stream instead of tokens on the token stream. Referenced grammar elements include token references (implicit lexer rule references), characters, and strings. Lexer rules are processed in the exact same manner as parser rules and, hence, may specify arguments and return values; further, lexer rules can also have local variables and use recursion. The following rule defines a rule called `ID` that is available as a token type in the parser.

```
ID : ( 'a' .. 'z' )+
    ;
```

This rule would become part of the resulting lexer and would appear as a method called `mID()`.

Lexer rules allow your parser to match *context-free* structures on the input character stream

as opposed to the much weaker *regular* structures (using a DFA--deterministic finite automaton). For example, consider that matching nested curly braces with a DFA must be done using a counter whereas nested curlies are trivially matched with a context-free grammar:

```
ACTION
    :   '{' ( ACTION | ~' }' )* '}'
    ;
```

The recursion, of course, is the dead giveaway that this is not an ordinary lexer rule.

Because the same algorithms are used to analyze lexer and parser rules, lexer rules may use more than a single symbol of lookahead, can use semantic predicates, and can specify syntactic predicates to look arbitrarily ahead, thus, providing recognition capabilities beyond the LL(k) languages into the *context-sensitive*. Here is a simple example that requires $k > 1$ lookahead:

```
ESCAPE_CHAR
    :   '\\\t' // two char of lookahead needed,
    |   '\\\n' // due to common left-prefix
    ;
```

To illustrate the use of syntactic predicates for lexer rules, consider the problem of distinguishing between floating point numbers and ranges in Pascal. Input 3 . . 4 must be broken up into 3 tokens: INT, RANGE, followed by INT. Input 3 . 4, on the other hand, must be sent to the parser as a REAL. The trouble is that the series of digits before the first '.' can be arbitrarily long. The scanner then must consume the first '.' to see if the next character is a '.', which would imply that it must back up and consider the first series of digits an integer. Using a non-backtracking lexer makes this task essentially impossible. However, a syntactic predicate can be used to specify what arbitrary lookahead is necessary:

```
class Pascal;

prog:    INT
        (    RANGE INT
            { System.out.println("INT .. INT"); }
        |    EOF
            { System.out.println("plain old INT"); }
        )
        |    REAL { System.out.println("token REAL"); }
        ;

lexclass LexPascal;

WS    :    (' '
        |   '\t'
        |   '\n'
        |   '\r')
        { $setType(Token.SKIP); }
        ;

protected
INT    :    ('0'..'9')+

```

```

;

protected
REAL:  INT '.' INT
;

RANGE
:  ".."
;

RANGE_OR_INT
:  ( INT ".." ) => INT  { $setType(INT); }
|  ( INT '.' )  => REAL { $setType(REAL); }
|  INT          { $setType(INT); }
;

```

ANTLR lexer rules are even able to handle FORTRAN assignments and other difficult lexical constructs. Consider the following DO loop:

```
DO 100 I = 1,10
```

If the comma were replaced with a period, the loop would become an assignment to a weird variable called "DO100I":

```
DO 100 I = 1.10
```

The following rules correctly differentiate the two cases:

```
DO_OR_VAR
:  (DO_HEADER)=> "DO" { $setType(DO); }
|  VARIABLE { $setType(VARIABLE); }
;

```

```
protected
DO_HEADER
options { ignore=WS; }
:  "DO" INT VARIABLE '=' EXPR ','
;

```

```
protected INT : ('0'..'9')+;
```

```
protected WS : ' ';
```

```
protected
VARIABLE
:  'A'..'Z'
   ('A'..'Z' | ' ' | '0'..'9')*
   { /* strip space from end */ }
;

```

```
// just an int or float
protected EXPR
:  INT ( '.' (INT)? )?

```


;

Return values

All rules return a token object (conceptually) automatically, which contains the text matched for the rule and its token type at least. To specify a user-defined return value, define a return value and set it in an action:

```
protected
INT returns [int v]
    : ('0'..'9')+ { v=Integer.valueOf($getText); }
    ;
```

Non-protected rules cannot have a return type as the parser cannot access the return value leading to confusion.

Skipping characters

To have the characters matched by a rule ignored, set the token type to `Token.SKIP`. For example,

```
WS : ( ' ' | '\t' | '\n' { newline(); } | '\r' )+
    { $setType(Token.SKIP); }
    ;
```

Distinguishing between lexer rules

ANTLR generates a rule called `nextToken` which has an alternative containing a lexer rule reference, one alternative for each non-protected lexer rule. The first few characters of the token are used to route the lexer to the appropriate lexical rule. The alternatives of `nextToken` are analyzed for determinism; i.e., with k characters of lookahead can the lexer determine which lexer rule to attempt.

The ANTLR input:

```
INT : ('0'..'9')+;
WS  : ' ' | '\t' | '\r' | '\n';
```

produces a switch statement in `nextToken`:

```
switch (LA(1)) {
    case '0': case '1': case '2': case '3':
    case '4': case '5': case '6': case '7':
    case '8': case '9':
        mINT(); break;
    case '\t': case '\n': case '\r': case ' ':
        mWS(); break;
    default: // error
}
}
```

Definition order and lexical ambiguities

ANTLR 2.0 does not follow the common lexer rule of "first definition wins" (the alternatives within a rule, however, still follow this rule). Instead, sufficient power is given to handle the two most common cases of ambiguity, namely "keywords vs. identifiers", and

"common prefixes"; and for especially nasty cases you can use syntactic predicates.

Keywords and literals

Many languages have a general "identifier" lexical rule, and keywords that are special cases of the identifier pattern. A typical identifier token is defined as:

```
ID : LETTER ( LETTER | DIGIT ) * ;
```

This is often in conflict with keywords. ANTLR 2.0 solves this problem by letting you put fixed keywords into a literals table. The literals table (which is usually implemented as a hash table in the lexer) is checked after each token is matched, so that the literals effectively override the more general identifier pattern. Literals are created in one of two ways. First, any double-quoted string used in a parser is automatically entered into the literals table of the associated lexer. Second, literals may be specified in the lexer grammar by means of the [literal option](#). In addition, the [testLiterals option](#) gives you fine-grained control over the generation of literal-testing code.

Common prefixes

Fixed-length common prefixes in lexer rules are best handled by increasing the [lookahead depth](#) of the lexer. For example, some operators from Java:

```
class MyLexer extends Lexer;
options {
    k=4;
}
GT : ">";
GE : ">=";
RSHIFT : ">>";
RSHIFT_ASSIGN : ">>=";
UNSIGNED_RSHIFT : ">>>";
UNSIGNED_RSHIFT_ASSIGN : ">>>=";
```

Token definition files

Token definitions can be transferred from one grammar to another by way of token definition files. This is accomplished using the [importVocab](#) and [exportVocab](#) options.

Character classes

Use the ~ operator to invert a character or set of characters. For example, to match any character other than newline, the following rule references ~'\n'.

```
SL_COMMENT : "//" (~'\n')* '\n';
```

The ~ operator also inverts a character set:

```
NOT_WS : ~( ' ' | '\t' | '\n' | '\r' );
```

The range operator can be used to create sequential character sets:

```
DIGIT : '0'..'9' ;
```

Token Attributes

See the next section.

Lexical lookahead and the end-of-token symbol

A unique situation occurs when analyzing lexical grammars, one which is similar to the end-of-file condition when analyzing regular grammars. Consider how you would compute lookahead sets for the ('b'|) subrule in following rule B:

```
class L extends Lexer;
```

```
A      :      B 'b'
      ;
```

```
protected // only called from another lex rule
```

```
B      :      'x' ( 'b' | )
      ;
```

The lookahead for the first alternative of the subrule is clearly 'b'. The second alternative is empty and the lookahead set is the set of all characters that can follow references to the subrule, which is the follow set for rule B. In this case, the 'b' character follows the reference to B and is therefore the lookahead set for the empty alt indirectly. Because 'b' begins both alternatives, the parsing decision for the subrule is nondeterminism or ambiguous as we sometimes say. ANTLR will justly generate a warning for this subrule (unless you use the `warnWhenFollowAmbig` option).

Now, consider what would make sense for the lookahead if rule A did not exist and rule B was not protected (it was a complete token rather than a "subtoken"):

```
B      :      'x' ( 'b' | )
      ;
```

In this case, the empty alternative finds only the end of the rule as the lookahead with no other rules referencing it. In the worst case, **any** character could follow this rule (i.e., start the next token or error sequence). So, should not the lookahead for the empty alternative be the entire character vocabulary? And should not this result in a nondeterminism warning as it must conflict with the 'b' alternative? Conceptually, yes to both questions. From a practical standpoint, however, you are clearly saying "heh, match a 'b' on the end of token B if you find one." I argue that no warning should be generated and ANTLR's policy of matching elements as soon as possible makes sense here as well.

Another reason not to represent the lookahead as the entire vocabulary is that a vocabulary of '\u0000'..\uFFFF' is really big (one set is $2^{16} / 32$ long words of memory!). Any alternative with '<end-of-token>' in its lookahead set will be pushed to the ELSE or DEFAULT clause by the code generator so that huge bitsets can be avoided.

The summary is that lookahead purely derived from hitting the end of a lexical rule (unreferenced by other rules) cannot be the cause of a nondeterminism. The following table summarizes a bunch of cases that will help you figure out when ANTLR will complain and when it will not.

X	:	'q' ('a')? ('a')?	The first subrule is nondeterministic as 'a' from second subrule (and end-of-token) are in the lookahead for exit branch of (...)?
X	:	'q' ('a')? ('c')?	No nondeterminism.
Y	:	'y' X 'b'	Nondeterminism in rule X.
protected X	:	'b'	
X	:	'x' ('a' 'c' 'd')+ 'z' ('a')+ ;	No nondeterminism as exit branch of loops see lookahead computed purely from end-of-token.
Y	:	'y' ('a')+ ('a')?	Nondeterminism between 'a' of (...)+ and exit branch as the exit can see the 'a' of the optional subrule. This would be a problem even if ('a')? were simply 'a'. A (...)* loop would report the same problem.
X	:	'y' ('a' 'b')+ 'a' 'c'	At k=1, this is a nondeterminism for the (...)? since 'a' predicts staying in and exiting the loop. At k=2, no nondeterminism.
Q	:	'q' ('a')?	Here, there is an empty alternative inside an optional subrule. A nondeterminism is reported as two paths predict end-of-token.

You might be wondering why the first subrule below is ambiguous:

('a')? ('a')?

The answer is that the NFA to DFA conversion would result in a DFA with the 'a' transitions merged into a single state transition! This is ok for a DFA where you cannot have actions anywhere except after a complete match. Remember that ANTLR lets you do the following:
 ('a' {do-this})? ('a' {do-that})?

One other thing is important to know. Recall that alternatives in lexical rules are reordered according to their lookahead requirements, from highest to lowest.

```
A      :      'a'
      |      'a' 'b'
      ;
```

At k=2, ANTLR can see 'a' followed by '<end-of-token>' for the first alternative and 'a' followed by 'b' in the second. The lookahead at depth 2 for the first alternative being '<end-of-token>' suppressing a warning that depth two can match any character for the first alternative. To behave naturally and to generate good code when no warning is generated, ANTLR reorders the alternatives so that the code generated is similar to:

```
A() {
    if ( LA(1)=='a' && LA(2)=='b' ) { // alt 2
        match('a'); match('b');
    }
    else if ( LA(1)=='a' ) { // alt 1
        match('a')
    }
    else {error;}
}
```

Note the lack of lookahead test for depth 2 for alternative 1. When an empty alternative is present, ANTLR moves it to the end. For example,

```
A      :      'a'
      |
      |      'a' 'b'
      ;
```

results in code like this:

```
A() {
    if ( LA(1)=='a' && LA(2)=='b' ) { // alt 2
        match('a'); match('b');
    }
    else if ( LA(1)=='a' ) { // alt 1
        match('a')
    }
    else {
    }
}
```

Note that there is no way for a lexing error to occur here (which makes sense because the rule is optional--though this rule only makes sense when protected).

Semantic predicates get moved along with their associated alternatives when the alternatives are sorted by lookahead depth. It would be weird if the addition of a {true}? predicate

(which implicitly exists for each alternative) changed what the lexer recognized! The following rule is reordered so that alternative 2 is tested for first.

```
B      :      {true}? 'a'
        |      'a' 'b'
        ;
```

Syntactic predicates are **not** reordered. Mentioning the predicate after the rule it conflicts with results in an ambiguity such as is in this rule:

```
F      :      'c'
        |      ('c')=> 'c'
        ;
```

Other alternatives are, however, reordered with respect to the syntactic predicates even when a switch is generated for the LL(1) components and the syntactic predicates are pushed the default case. The following rule illustrates the point.

```
F      :      'b'
        |      { /* empty-path */
        |      ('c')=> 'c'
        |      'c'
        |      'd'
        |      'e'
        ;
```

Rule F's decision is generated as follows:

```
switch ( la_1 ) {
case 'b':
{
    match('b');
    break;
}
case 'd':
{
    match('d');
    break;
}
case 'e':
{
    match('e');
    break;
}
default:
    boolean synPredMatched15 = false;
    if (((la_1=='c'))) {
        int _m15 = mark();
        synPredMatched15 = true;
        guessing++;
        try {
            match('c');
        }
        catch (RecognitionException pe) {
```

```

        synPredMatched15 = false;
    }
    rewind(_m15);
    guessing--;
}
if ( synPredMatched15 ) {
    match('c');
}
else if ((la_1=='c')) {
    match('c');
}
else {
    if ( guessing==0 ) {
        /* empty-path */
    }
}
}
}

```

Notice how the empty path got moved after the test for the 'c' alternative.

Scanning Binary Files

[Prior to 2.2.3, ANTLR was unable to parse binary files.] Character literals are not limited to printable ASCII characters. To demonstrate the concept, imagine that you want to parse a binary file that contains strings and short integers. To distinguish between them, marker bytes are used according to the following format:

format	description
<code>"\0' highbyte lowbyte</code>	Short integer
<code>"\1' string of non-\2' chars \2'</code>	String

Sample input (274 followed by "a test") might look like the following in hex (output from UNIX `od -h` command):

```
0000000000    00 01 12 01 61 20 74 65 73 74 02
```

or as viewed as characters:

```
0000000000    \0 001 022 001 a      t e s t 002
```

The parser is trivially just a (...) around the two types of input tokens:

```
class DataParser extends Parser;
```

```

file:    (    sh:SHORT
            {System.out.println(sh.getText());}
        |    st:STRING
            {System.out.println("\ "+
                                st.getText()+"\ ");}
        )+
        ;

```

All of the interesting stuff happens in the lexer. First, define the class and set the vocabulary to be all 8 bit binary values:

```

class DataLexer extends Lexer;
options {
    charVocabulary = '\u0000'..' \u00FF';
}

```

Then, define the two tokens according to the specifications, with markers around the string and a single marker byte in front of the short:

```

SHORT
    :    // match the marker followed by any 2 bytes
        '\0' high:. lo:.
        {
            // pack the bytes into a two-byte short
            int v = (((int)high)<<8) + lo;
            // make a string out of the value
            $setText(""+v);
        }
    ;

STRING
    :    '\1'!    // begin string (discard)
        ( ~'\2' )*
        '\2'!    // end string (discard)
    ;

```

To invoke the parser, use something like the following:

```

import java.io.*;

class Main {
    public static void main(String[] args) {
        try {
            // use DataInputStream to grab bytes
            DataLexer lexer =
                new DataLexer(
                    new DataInputStream(System.in)
                );
            DataParser parser =
                new DataParser(lexer);
            parser.file();
        } catch (Exception e) {
            System.err.println("exception: "+e);
        }
    }
}

```

Scanning Unicode Characters

ANTLR (as of 2.7.1) allows you to recognize input composed of Unicode characters; that is, you are not restricted to 8 bit ASCII characters. I would like to emphasize that ANTLR *allows*, but does yet not *support* Unicode as there is more work to be done. For example, end-of-file is currently incorrectly specified:


```
CharScanner.EOF_CHAR=(char)-1;
```

This must be an integer -1 not char, which is actually narrowed to 0xFFFF via the cast. I have to go through the entire code base looking for these problems. Plus, we should really have a special syntax to mean "java identifier character" and some standard encodings for non-Western character sets etc...

Another problem is that ANTLR generates bit sets when a few range checks would work. This makes for some big output files (all data), but at least the Java compilers will compile the output now (previously giant switch expressions would prevent compilation).

The following is a very simple example of how to match a series of space-separated identifiers.

```
class L extends Lexer;

options {
    // Allow any char but \uFFFF (16 bit -1)
    charVocabulary='\u0000'..\uFFFE';
}

{
    private static boolean done = false;

    public void uponEOF()
        throws TokenStreamException, CharStreamException
    {
        done=true;
    }

    public static void main(String[] args) throws Exception {
        L lexer = new L(System.in);
        while ( !done ) {
            Token t = lexer.nextToken();
            System.out.println("Token: "+t);
        }
    }
}

ID      :      ID_START_LETTER ( ID_LETTER )*
;

WS      :      ( ' ' | '\n' ) {$setType(Token.SKIP);}
;

protected
ID_START_LETTER
:      '$'
|      '_'
|      'a'..'z'
|      '\u0080'..\ufffe'
;
```

```
protected
ID_LETTER
    :      ID_START_LETTER
    |      '0'..'9'
    ;
```

A final note on Unicode. The `~x` "not" operator includes everything in your specified vocabulary (up to 16 bit character space) except `x`. For example,

```
~( '$' | 'a'..'z' )
```

results in every unicode character except '\$' and lowercase latin-1 letters, assuming your `charVocabulary` is `0..FFFF`.

Manipulating Token Text and Objects

Once you have specified what to match in a lexical rule, you may ask "what can I discover about what will be matched for each rule element?" ANTLR allows you to label the various elements and, at parse-time, access the text matched for the element. You can even specify the token object to return from the rule and, hence, from the lexer to the parser. This section describes the text and token object handling characteristics of ANTLR.

Manipulating the Text of a Lexical Rule

There are times when you want to look at the text matched for the current rule, alter it, or set the text of a rule to a new string. The most common case is when you want to simply discard the text associated with a few of the elements that are matched for a rule such as quotes.

ANTLR provides the `!` operator that lets you indicate certain elements should not contribute to the text for a token being recognized. The `!` operator is used just like when building trees in the parser. For example, if you are matching the HTML tags and you do not want the `<` and `>` characters returned as part of the token text, you could manually remove them from the token's text before they are returned, but a better way is to suffix the unwanted characters with `!`. For example, the `
` tag might be recognized as follows:

```
BR : '<!' "br" '>!' ; // discard < and >
```

Suffixing a lexical rule reference with `!` forces the text matched by the invoked rule to be discarded (it will not appear in the text for the invoking rule). For example, if you do not care about the mantissa of a floating point number, you can suffix the rule that matches it with a `!`:

```
FLOAT : INT ('.'! INT!)? ; // keep only first INT
```

As a shorthand notation, you may suffix an alternative or rule with `!` to indicate the alternative or rule should not pass any text back to the invoking rule or parser (if nonprotected):

```
// ! on rule: nothing is auto added to text of rule.
rule! : ... ;
```

```
// ! on alt: nothing is auto added to text for alt
rule : ... |! ...;
```

Item suffixed with '!'	Effect
char or string literal	Do not add text for this atom to current rule's text.
rule reference	Do not add text for matched while recognizing this rule to current rule's text.
alternative	Nothing that is matched by alternative is added to current rule's text; the enclosing rule contributes nothing to any invoking rule's text. For nonprotected rules, the text for the token returned to parser is blank.
rule definition	Nothing that is matched by any alternative is added to current rule's text; the rule contributes nothing to any invoking rule's text. For nonprotected rules, the text for the token returned to parser is blank.

While the '!' implies that the text is not added to the text for the current rule, you can label an element to access the text (via the token if the element is a rule reference).

In terms of implementation, the characters are always added to the current text buffer, but are carved out when necessary (as this will be the exception rather than the rule, making the normal case efficient).

The '!' operator is great for discarding certain characters or groups of characters, but what about the case where you want to insert characters or totally reset the text for a rule or token? ANTLR provides a series of special methods to do this (we prefix the methods with '\$' because Java does not have a macro facility and ANTLR must recognize the special methods in your actions). The following table summarizes.

Method	Description/Translation
<code>\$append(x)</code>	Append <code>x</code> to the text of the surrounding rule. Translation: <code>text.append(x)</code>
<code>\$setText(x)</code>	Set the text of the surrounding rule to <code>x</code> . Translation: <code>text.setLength(_begin); text.append(x)</code>
<code>\$getText</code>	Return a String of the text for the surrounding rule. Translation; <code>new String(text.getBuffer(), _begin, text.length() - _begin)</code>
<code>\$setToken(x)</code>	Set the token object that this rule is to return. See the section on Token Object Creation . Translation: <code>_token = x</code>
<code>\$setType(x)</code>	Set the token type of the surrounding rule. Translation: <code>_ttype = x</code>
<code>setText(x)</code>	Set the text for the entire token being recognized regardless of what rule the action is in. No translation.
<code>getText()</code>	Get the text for the entire token being recognized regardless of what rule the action is in. No translation.

One of the great things about an ANTLR generated lexer is that the text of a token can be modified incrementally as the token is recognized (an impossible task for a DFA-based lexer):

```
STRING: ' ' ( ESCAPE | ~( ' ' | '\\ ' ) ) * ' ' ;
```

```
protected
ESCAPE
    :    '\\\ '
        ( 'n' { $setText( "\n" ); }
        | 'r' { $setText( "\r" ); }
        | 't' { $setText( "\t" ); }
        | '"' { $setText( "\"" ); }
        )
    ;
```

Token Object Creation

Because lexical rules can call other rules just like in the parser, you sometimes want to know what text was matched for that portion of the token being matched. To support this, ANTLR allows you to label lexical rules and obtain a `Token` object representing the text, token type, line number, etc... matched for that rule reference. This ability corresponds to be able to access the text matched for a lexical state in a DFA-based lexer. For example, here is a simple rule that prints out the text matched for a rule reference, `INT`.

```
INDEX    :    '[' i:INT ']'
           {System.out.println(i.getText());}
           ;

INT      :    ('0'..'9')+ ;
```

If you moved the labeled reference and action to a parser, it would do the same thing (match an integer and print it out).

All lexical rules *conceptually* return a `Token` object, but in practice this would be inefficient. ANTLR generates methods so that a token object is created only if any invoking reference is labeled (indicating they want the token object). Imagine another rule that calls `INT` without a label.

```
FLOAT    :    INT ( '.' INT ) ? ;
```

In this case, no token object is created for either reference to `INT`. You will notice a boolean argument to every lexical rule that tells it whether or not a token object should be created and returned (via a member variable). All nonprotected rules (those that are "exposed" to the parser) must always generate tokens, which are passed back to the parser.

Heterogeneous Token Object Streams

While token creation is normally handled automatically, you can also manually specify the token object to be returned from a lexical rule. The advantage is that you can pass heterogeneous token objects back to the parser, which is extremely useful for parsing languages with complicated tokens such as HTML (the `` and `<table>` tokens, for example, can have lots of attributes). Here is a rule for the `` tag that returns a token object of type `ImageToken`:

```
IMAGE
{
    Attributes attrs;
}
```

```

    : "<img " attrs=ATTRIBUTES '>'
      {
        ImageToken t = new ImageToken(IMAGE,$getText);
        t.setAttributes(attrs);
        $setToken(t);
      }
    ;
ATTRIBUTES returns [Attributes a]
    : ...
    ;

```

The `$setToken` function specifies that its argument is to be returned when the rule exits. The parser will receive this specific object instead of a `CommonToken` or whatever else you may have specified with the `Lexer.setTokenObjectClass` method. The action in rule `IMAGE` references a token type, `IMAGE`, and a lexical rule references, `ATTRIBUTES`, which matches all of the attributes of an image tag and returns them in a data structure called `Attributes`.

What would it mean for rule `IMAGE` to be protected (i.e., referenced only from other lexical rules rather than from `nextToken`)? Any invoking labeled rule reference would receive the object (not the parser) and could examine it, or manipulate it, or pass it on to the invoker of that rule. For example, if `IMAGE` were called from `TAGS` rather than being nonprotected, rule `TAGS` would have to pass the token object back to the parser for it.

```

TAGS : IMG:IMAGE
      { $setToken(img); } // pass to parser
    | PARAGRAPH // probably has no special token
    | ...
    ;

```

Setting the token object for a nonprotected rule invoked without a label has no effect other than to waste time creating an object that will not be used.

We use a `CharScanner` member `_returnToken` to do the return in order to not conflict with return values used by the grammar developer. For example,

```
PTAG: "<p>" { $setToken(new ParagraphToken($$)); } ;
```

which would be translated to something like:

```

protected final void mPTAG()
    throws RecognitionException, CharStreamException,
           TokenStreamException {
    Token _token = null;
    match("<p>");
    _returnToken =
        new ParagraphToken(text-of-current-rule);
}

```

Filtering Input Streams

You often want to perform an action upon seeing a pattern or two in a complicated input stream, such as pulling out links in an HTML file. One solution is to take the HTML

grammar and just put actions where you want. Using a complete grammar is overkill and you may not have a complete grammar to start with.

ANTLR provides a mechanism similar to AWK that lets you say "here are the patterns I'm interested in--ignore everything else." Naturally, AWK is limited to regular expressions whereas ANTLR accepts context-free grammars (Uber-AWK?). For example, consider pulling out the <p> and
 tags from an arbitrary HTML file. Using the filter option, this is easy:

```
class T extends Lexer;
options {
    k=2;
    filter=true;
}
```

```
P : "<p>" ;
BR: "<br>" ;
```

In this "mode", there is no possibility of a syntax error. Either the pattern is matched exactly or it is filtered out.

This works very well for many cases, but is not sophisticated enough to handle the situation where you want "almost matches" to be reported as errors. Consider the addition of the <table...> tag to the previous grammar:

```
class T extends Lexer;
options {
    k=2;
    filter = true;
}
```

```
P : "<p>" ;
BR: "<br>" ;
TABLE : "<table" (WS)? (ATTRIBUTE)* (WS)? '>' ;
WS : ' ' | '\t' | '\n' ;
ATTRIBUTE : ... ;
```

Now, consider input "<table 8 = width ;>" (a bogus table definition). As is, the lexer would simply scarf past this input without "noticing" the invalid table. What if you want to indicate that a bad table definition was found as opposed to ignoring it? Call method

```
setCommitToPath(boolean commit)
```

in your TABLE rule to indicate that you want the lexer to commit to recognizing the table tag:

```
TABLE
    :    "<table" (WS)?
        {setCommitToPath(true);}
        (ATTRIBUTE)* (WS)? '>'
    ;
```

Input "<table 8 = width ;>" would result in a syntax error. Note the placement after the whitespace recognition; you do not want <tabletop> reported as a bad table (you want to ignore it).

One further complication in filtering: What if the "skip language" (the stuff in between valid tokens or tokens of interest) cannot be correctly handled by simply consuming a character and trying again for a valid token? You may want to ignore comments or strings or whatever. In that case, you can specify a rule that scarfs anything between tokens of interest by using option `filter=RULE`. For example, the grammar below filters for `<p>` and `
` tags as before, but also prints out any other tag (`<...>`) encountered.

```
class T extends Lexer;
options {
    k=2;
    filter=IGNORE;
    charVocabulary = '\3'..'177';
}

P : "<p>" ;
BR: "<br>" ;

protected
IGNORE
    :   '<' (~'>')* '>'
        {System.out.println("bad tag:"+getText);}
    |   ("\r\n" | '\r' | '\n' ) {newline();}
    |   .
    ;
```

Notice that the filter rule must track newlines in the general case where the lexer might emit error messages so that the line number is not stuck at 0.

The filter rule is invoked either when the lookahead (in `nextToken`) predicts none of the nonprotected lexical rules or when one of those rules fails. In the latter case, the input is rolled back before attempting the filter rule. Option `filter=true` is like having a filter rule such as:

```
IGNORE : . ;
```

Actions in regular lexical rules are executed even if the rule fails and the filter rule is called. To do otherwise would require every valid token to be matched twice (once to match and once to do the actions like a syntactic predicate)! Plus, there are few actions in lexer rules (usually they are at the end at which point an error cannot occur).

Is the filter rule called when `commit-to-path` is true and an error is found in a lexer rule? No, an error is reported as with `filter=true`.

What happens if there is a syntax error in the filter rule? Well, you can either put an exception handler on the filter rule or accept the default behavior, which is to consume a character and begin looking for another valid token.

In summary, the filter option allows you to:

1. Filter like awk (only perfect matches reported--no such thing as syntax error)
2. Filter like awk + catch poorly-formed matches (that is, "almost matches" like `<table 8=3;>` result in an error)
3. Filter but specify the skip language

ANTLR Masquerading as SED

To make ANTLR generate lexers that behave like the UNIX utility sed (copy standard in to standard out except as specified by the replace patterns), use a filter rule that does the input to output copying:

```
class T extends Lexer; options { k=2; filter=IGNORE;
charVocabulary = '\3'..'177'; }
P : "<p>" {System.out.print("<P>");};
BR : "<br>" {System.out.print("<BR>");};
protected
IGNORE
: ( "\r\n" | '\r' | '\n' )
  {newline(); System.out.println("");};
| c: . {System.out.print(c);}
;
```

This example dumps anything other than <p> and
 tags to standard out and pushes lowercase <p> and
 to uppercase. Works great.

Nongreedy Subrules

Quick: What does the following match?

```
BLOCK : '{' (.)* '}';
```

Your first reaction is that it matches any set of characters inside of curly quotes. In reality, it matches '{' followed by every single character left on the input stream! Why? Well, because ANTLR loops are *greedy*--they consume as much input as they can match. Since the wildcard matches any character, it consumes the '}' and beyond. This is a pain for matching strings, comments and so on.

Why can't we switch it around so that it consumes only until it sees something on the input stream that matches what **follows** the loop, such as the '}'? That is, why can't we make loops *nongreedy*? The answer is we can, but sometimes you want greedy and sometimes you want nongreedy (PERL has both kinds of closure loops now too). Unfortunately, parsers usually want greedy and lexers usually want nongreedy loops. Rather than make the same syntax behave differently in the various situations, Terence decided to leave the semantics of loops as they are (greedy) and make a subrule option to make loops nongreedy.

Greedy Subrules

I have yet to see a case when building a parser grammar where I did not want a subrule to match as much input as possible. For example, the solution to the classic if-then-else clause ambiguity is to match the "else" as soon as possible:

```
stat : "if" expr "then" stat ("else" stat)?
      | ...
;
```

This ambiguity (which statement should the "else" be attached to) results in a parser nondeterminism. ANTLR warns you about the (. . .)? subrule as follows:

```
warning: line 3: nondeterminism upon
```



```

k==1:"else"
between alts 1 and 2 of block

```

If, on the other hand, you make it clear to ANTLR that you want the subrule to match greedily (i.e., assume the default behavior), ANTLR will not generate the warning. Use the greedy subrule option to tell ANTLR what you want:

```

stat : "if" expr "then" stat
      ( options {greedy=true;} : "else" stat)?
    | ID
    ;

```

You are not altering the behavior really, since ANTLR was going to choose to match the "else" anyway, but you have avoided a warning message.

There is no such thing as a nongreedy (. . .)? subrule because telling an optional subrule not to match anything is the same as not specifying the subrule in the first place. If you make the subrule nongreedy, you will see:

```

warning in greedy.g: line(4),
    Being nongreedy only makes sense
    for (...) + and (...) *
warning: line 4: nondeterminism upon
    k==1:"else"
    between alts 1 and 2 of block

```

Greedy subrules are very useful in the lexer also. If you want to grab any whitespace on the end of a token definition, you can try (WS)? for some whitespace rule WS:

```

ID : ('a'..'z')+ (WS)? ;

```

However, if you want to match ID in a loop in another rule that could also match whitespace, you will run into a nondeterminism warning. Here is a contrived loop that conflicts with the (WS)? in ID:

```

LOOP : ( ID
        | WS
        )+
    ;

```

The whitespace on the end of the ID could be matched in ID or in LOOP now. ANTLR chooses to match the WS immediately, in ID. To shut off the warning, simply tell ANTLR that you mean for it to be greedy, it's default behavior:

```

ID : ('a'..'z')+ (options {greedy=true;}:WS)? ;

```

Nongreedy Lexer Subrules

ANTLR's default behavior of matching as much as possible in loops and optional subrules is sometimes not what you want in lexer grammars. Most loops that match "a bunch of characters" in between markers, like curly braces or quotes, should be nongreedy loops. For example, to match a nonnested block of characters between curly braces, you want to say:

```

CURLY_BLOCK_SCARF
:   '{' (.)* '}'
;

```

Unfortunately, this does not work--it will consume everything after the '{' until the end of the input. The wildcard matches anything including '}' and so the loop merrily consumes past the ending curly brace.

To force ANTLR to break out of the loop when it sees a lookahead sequence consistent with what follows the loop, use the greedy subrule option:

```
CURLY_BLOCK_SCARF
    :   '{'
        (
            options {
                greedy=false;
            }
            .
        )*
        '}'
    ;
```

To properly take care of newlines inside the block, you should really use the following version that "traps" newlines and bumps up the line counter:

```
CURLY_BLOCK_SCARF
    :   '{'
        (
            options {
                greedy=false;
            }
            : '\r' ('\n')? {newline();}
            | '\n'          {newline();}
            | .
        )*
        '}'
    ;
```

Limitations of Nongreedy Subrules

What happens when what follows a nongreedy subrule is not as simple as a single "marker" character like a right curly brace (i.e., what about when you need $k > 1$ to break out of a loop)? ANTLR will either "do the right thing" or warn you that it might not.

First, consider the matching C comments:

```
CMT : "/*" (.)* "*/" ;
```

As with the curly brace matching, this rule will not stop at the end marker because the wildcard matches the "*/" end marker as well. You must tell ANTLR to make the loop nongreedy:

```
CMT : "/*" (options {greedy=false;} :.)* "*/" ;
```

You will not get an error and ANTLR will generate an exit branch

```
do {
    // nongreedy exit test
    if ((LA(1)=='*')) break _loop3;
    ...
}
```

Ooops. $k=1$, which is not enough lookahead. ANTLR did not generate a warning because it assumes you are providing enough lookahead for all nongreedy subrules. ANTLR cannot determine how much lookahead to use or how much is enough because, by definition, the decision is ambiguous--it simply generates a decision using the maximum lookahead.

You must provide enough lookahead to let ANTLR see the full end marker:

```
class L extends Lexer;
options {
    k=2;
}
```

```
CMT : "/*" (options {greedy=false;} :.)* "*/" ;
```

Now, ANTLR will generate an exit branch using $k=2$.

```
do {
    // nongreedy exit test
    if ((LA(1)=='*' && (LA(2)=='/'))
        break _loop3;
    ...
```

If you increase k to 3, ANTLR will generate an exit branch using $k=3$ instead of 2, even though 2 is sufficient. We know that $k=2$ is ok, but ANTLR is faced with a nondeterminism at it will use as much information as it has to yield a deterministic parser.

There is one more issue that you should be aware of. Because ANTLR generates linear approximate decisions instead of full $LL(k)$ decisions, complicated "end markers" can confuse ANTLR. Fortunately, ANTLR knows when it is confused and will let you know.

Consider a simple contrived example where a loop matches either `ab` or `cd`:

```
R : ( options {greedy=false;}
    : ("ab" | "cd")
    )+
    ("ad" | "cb")
    ;
```

Following the loop, the grammar can match `ad` or `cb`. These exact sequences are not a problem for a full $LL(k)$ decision, but due to the extreme compression of the linear approximate decision, ANTLR will generate an inaccurate exit branch. In other words, the loop will exit, for example, on `ab` even though that sequence cannot be matched following the loop. The exit condition is as follows:

```
// nongreedy exit test
if ( _cnt10>=1 && (LA(1)=='a' || LA(1)=='c') &&
    (LA(2)=='b' || LA(2)=='d')) break _loop10;
```

where the `_cnt10` term ensures the loop goes around at least once (but has nothing to do with the nongreedy exit branch condition really). Note that ANTLR has compressed all characters that can possibly be matched at a lookahead depth into a single set, thus, destroying the sequence information. The decision matches the cross product of the sets, including the spurious lookahead sequences such as `ab`.

Fortunately, ANTLR knows when a decision falls between its approximate decision and a

full LL(k) decision--it warns you as follows:

```
warning in greedy.g: line(3),
    nongreedy block may exit incorrectly due
    to limitations of linear approximate lookahead
    (first k-1 sets in lookahead not singleton).
```

The parenthetical remark gives you a hint that some $k > 1$ lookahead sequences are correctly predictable even with the linear approximate lookahead compression. The idea is that if all sets for depths $1..(k-1)$ are singleton sets (exactly one lookahead sequence for first $k-1$ characters) then linear approximate lookahead compression does not weaken your parser. So, the following variant does not yield a warning since the exit branch is linear approximate as well as full LL(k):

```
R : ( options {greedy=false;}
    :
    )+
    ("ad" | "ae" )
    ;
```

The exit branch decision now tests lookahead as follows:

```
(LA(1) == 'a' ) && (LA(2) == 'd' | LA(2) == 'e' )
```

which accurately predicts when to exit.

Lexical States

ANTLR has the ability to switch between multiple lexers using a token stream multiplexor. Please see the discussion in [streams](#).

The End Of File Condition

A method is available for reacting to the end of file condition as if it were an event; e.g., you might want to pop the lexer state at the end of an include file. This method, `CharScanner.uponEOF()`, is called from `nextToken()` right before the scanner returns an `EOF_TYPE` token object to parser:

```
public void uponEOF() |
    throws TokenStreamException, CharStreamException;
```

This event is not generated during a syntactic predicate evaluation (i.e., when the parser is guessing) nor in the middle of the recognition of a lexical rule (that would be an IO exception). This event is generated only after the complete evaluation of the last token and upon the next request from the parser for a token.

You can throw exceptions from this method like "Heh, premature eof" or a retry stream exception. See the `includeFile/P.g` for an example usage.

Case sensitivity

You may use option `caseSensitive=false` in the lexer to indicate that you do not want case to be significant when matching characters against the input stream. For example, you want element 'd' to match either upper or lowercase D, however, you do not want to change the case of the input stream. We have implemented this feature by having the lexer's

`LA()` lookahead method return lowercase versions of the characters. Method `consume()` still adds the original characters to the string buffer associated with a token. We make the following notes:

- The lowercasing is done by a method `toLowerCase()` in the lexer. This can be overridden to get more specific case processing. Using option `caseSensitive` calls method `CharScanner.setCaseSensitive(...)`, which you can also call before (or during I suppose) the parse.
- ANTLR issues a warning when `caseSensitive=false` and uppercase ASCII characters are used in character or string literals.

Case sensitivity for literals is handled separately. That is, set lexer option `caseSensitiveLiterals` to false when you want the literals testing to be case-insensitive. Implementing this required changes to the literals table. Instead of adding a `String`, it adds an `ANTLRHashString` that implements a case-insensitive or case-sensitive hashing as desired.

Note: ANTLR checks the characters of a lexer string to make sure they are lowercase, but does not process escapes correctly--put that one on the "to do" list.

Ignoring whitespace in the lexer

One of the great things about ANTLR is that it generates full predicated-LL(k) lexers rather than the weaker (albeit sometimes easier-to-specify) DFA-based lexers of DLG. With such power, you are tempted (and encouraged) to do real parsing in the lexer. A great example of this is HTML parsing, which begs for a two-level parse: the lexer parses all the attributes and so on within a tag, but the parser does overall document structure and ordering of the tags etc... The problem with parsing within a lexer is that you encounter the usual "ignore whitespace" issue as you do with regular parsing.

For example, consider matching the `<table>` tag of HTML, which has many attributes that can be specified within the tag. A first attempt might yield:

```
OTABLE      :      "<table" (ATTR)* '>'
              ;
```

Unfortunately, input "`<table border=1>`" does not parse because of the blank character after the `table` identifier. The solution is not to simply have the lexer ignore whitespace as it is read in because the lookahead computations must see the whitespace characters that will be found in the input stream. Further, defining whitespace as a rudimentary set of things to ignore does not handle all cases, particularly difficult ones, such as comments inside tags like

```
<table <!--wow...a comment--> border=1>
```

The correct solution is to specify a rule that is called after each lexical element (character, string literal, or lexical rule reference). We provide the lexer rule option `ignore` to let you specify the rule to use as whitespace. The solution to our HTML whitespace problem is therefore:

```
TABLE
options { ignore=WS; }
:      "<table" (ATTR)* '>'
```

```

        ;
// can be protected or non-protected rule
WS      :      ' ' | '\n' | COMMENT | ...
        ;

```

We think this is cool and we hope it encourages you to do more and more interesting things in the lexer!

Oh, almost forgot. There is a **bug** in that an extra whitespace reference is inserted after the end of a lexer alternative if the last element is an action. The effect is to include any whitespace following that token in that token's text.

Tracking Line Information

Each lexer object has a `line` member that can be incremented by calling `newline()` or by simply changing its value (e.g., when processing `#line` directives in C).

```
SL_COMMENT : "//" (~'\n')* '\n' {newline();} ;
```

Do not forget to split out `'\n'` recognition when using the not operator to read until a stopping character such as:

```

BLOCK : '('
        ( '\n' { newline(); }
          | ~( '\n' | ')' )
        )*
        ')'
        ;

```

Another way to track line information is to override the `consume()` method:

Tracking Column Information

ANTLR (2.7.1 and beyond), tracks character column information so that each token knows what column it starts in; columns start at 1 just like line numbers. The `CharScanner.consume()` method asks method `tab()` to update the column number if it sees a tab, else it just increments the column number:

```

...
if ( c=='\t' ) {
    tab();
}
else {
    inputState.column++;
}

```

By default, `tab()` is defined as follows:

```

/**
advance the current column number by an appropriate
amount. If you do not override this to specify how
much to jump for a tab, then tabs are counted as
one char. This method is called from consume().
*/

```

```

public void tab() {
    // update inputState.column as function of
    // inputState.column and tab stops.
    // For example, if tab stops are columns 1
    // and 5 etc... and column is 3, then add 2
    // to column.
    inputState.column++;
}

```

Upon new line, the lexer needs to reset the column number to 1. Here is the default implementation of `CharScanner.newline()`:

```

public void newline() {
    inputState.line++;
    inputState.column = 1;
}

```

Do not forget to call `newline()` in your lexer rule that matches `'\n'` lest the column number not be reset to 1 at the start of a line.

The shared input state object for a lexer is actually the critter that tracks the column number (as well as the starting column of the current token):

```

public class LexerSharedInputState {
    protected int column=1;
    protected int line=1;
    protected int tokenStartColumn = 1;
    protected int tokenStartLine = 1;
    ...
}

```

If you want to handle tabs in your lexer, just implement a method like the following to override the standard behavior.

```

/** set tabs to 4, just round column up to next tab + 1
12345678901234567890
   x  x  x  x
*/
public void tab() {
    int t = 4;
    int c = getColumn();
    int nc = (((c-1)/t)+1)*t+1;
    setColumn( nc );
}

```

See the examples/`java/columns` directory for the complete example.

Using Explicit Lookahead

On rare occasions, you may find it useful to explicitly test the lexer lookahead in say a semantic predicate to help direct the parse. For example, `/*...*/` comments have a two character stopping symbol. The following example demonstrates how to use the second symbol of lookahead to distinguish between a single `'/'` and a `"*/"`:

```

ML_COMMENT

```

```

:    "/"*
    ( { LA(2)!='/' }? '*'
    | '\n' { newline(); }
    | ~('*' | '\n')
    )*
    "*" /
;

```

The same effect might be possible via a syntactic predicate, but would be much slower than a semantic predicate. A DFA-based lexer handles this with no problem because they use a bunch of (what amount to) gotos whereas we're stuck with structured elements like while-loops.

A Surprising Use of A Lexer: Parsing

The following set of rules match arithmetical expressions in a lexer **not** a parser (whitespace between elements is not allowed in this example but can easily be handled by specifying rule option `ignore` for each rule):

```

EXPR
{ int val; }
:    val=ADDEXPR
    { System.out.println(val); }
;

protected
ADDEXPR returns [int val]
{ int tmp; }
:    val=MULTEXPR
    ( '+' tmp=MULTEXPR { val += tmp; }
    | '-' tmp=MULTEXPR { val -= tmp; }
    )*
;

protected
MULTEXPR returns [int val]
{ int tmp; }
:    val=ATOM
    ( '*' tmp=ATOM { val *= tmp; }
    | '/' tmp=ATOM { val /= tmp; }
    )*
;

protected
ATOM returns [int val]
:    val=INT
    | '(' val=ADDEXPR ')'
;

protected
INT returns [int val]

```



```

:    ('0'..'9')+
    {val=Integer.valueOf($getText);}
;

```

But...We've Always Used Automata For Lexical Analysis!

Lexical analyzers were all built by hand in the early days of compilers until DFAs took over as the scanner implementation of choice. DFAs have several advantages over hand-built scanners:

- DFAs can easily be built from terse regular expressions.
- DFAs do automatic left-factoring of common (possibly infinite) left-prefixes. In a hand-built scanner, you have to find and factor out all common prefixes. For example, consider writing a lexer to match integers and floats. The regular expressions are straightforward:

```

integer : "[0-9]+" ;
real    : "[0-9]+{.[0-9]*}|.[0-9]+" ;

```

Building a scanner for this would require factoring out the common `[0-9]+`. For example, a scanner might look like:

```

Token nextToken() {
    if ( Character.isDigit(c) ) {
        match an integer
        if ( c=='.' ) {
            match another integer
            return new Token(REAL);
        }
        else {
            return new Token(INT);
        }
    }
    else if ( c=='.' ) {
        match a float starting with .
        return new Token(REAL);
    }
    else ...
}

```

Conversely, hand-built scanners have the following advantages over DFA implementations:

- Hand-built scanners are not limited to the regular class of languages. They may use semantic information and method calls during recognition whereas a DFA has no stack and is typically not semantically predicated.
- Unicode (16 bit values) is handled for free whereas DFAs typically have fits about anything but 8 bit characters.
- DFAs are tables of integers and are, consequently, very hard to debug and examine.
- A tuned hand-built scanner can be faster than a DFA. For example, simulating the DFA to match `[0-9]+` requires n DFA state transitions where n is the length of the integer in characters.

Tom Pennello of Metaware back in 1986 ("Very Fast LR Parsing") generated

LR-based parsers in machine code that used the program counter to do state transitions rather than simulating the PDA. He got a huge speed up in parse time. We can extrapolate from this experiment that avoiding a state machine simulator in favor of raw code results in a speed up.

So, what approach does ANTLR 2.xx take? Neither! ANTLR 2.xx allows you to specify lexical items with expressions, but generates a lexer for you that mimics what you would generate by hand. The only drawback is that you still have to do the left-factoring for some token definitions (but at least it is done with expressions and not code). This hybrid approach allows you to build lexers that are much stronger and faster than DFA-based lexers while avoiding much of the overhead of writing the lexer yourself.

In summary, specifying regular expressions is simpler and shorter than writing a hand-built lexer, but hand-built lexers are faster, stronger, able to handle unicode, and easy to debug. This analysis has led many programmers to write hand-built lexers even when DFA-generation tools such as `lex` and `d1g` are commonly-available. ANTLR 1.xx made a parallel argument concerning PDA-based LR parsers and recursive-descent LL-based parsers. As a final justification, we note that writing lexers is trivial compared to building parsers; also, once you build a lexer you will reuse it with small modifications in the future.

Version: \$Id: //depot/code/org.antlr/release/antlr-2.7.1/doc/lexer.html#2 \$

ANTLR Tree Parsers

Or, The Entity Formerly Known As SORCERER

ANTLR 2.xx helps you build intermediate form trees (ASTs) by augmenting a grammar with tree operators, rewrite rules, and actions. ANTLR also allows you to specify the grammatical structure of ASTs, thus, supporting the manipulation or simple walking of trees to produce translations.

Formerly, a separate tool, SORCERER, was used to generate tree parsers, but ANTLR has taken over this role. ANTLR now builds recognizers for streams of characters, tokens, or tree nodes.

What's a tree parser?

Parsing is the application of grammatical structure to a stream of input symbols. ANTLR takes this further than most tools and considers a tree to be a stream of nodes, albeit in two dimensions. In fact, the only real difference in ANTLR's code generation for token stream parsing versus tree parsing lies in the testing of lookahead, rule-method definition headers, and the introduction of a two-dimensional tree structure code-generation template.

What kinds of trees can be parsed?

ANTLR tree parsers can walk any tree that implements the AST interface, which imposes a child-sibling like structure to whatever tree data-structure you may have. The important navigation methods are:

- `getFirstChild`: Return a reference to the first child of the sibling list.
- `getNextSibling`: Return a reference to the next child in the list of siblings.

Each AST node is considered to have a list of children, some text, and a "token type". Trees are self-similar in that a tree node is also a tree. An AST is defined completely as:

```
public interface AST {
    /** Get the token type for this node */
    public int getType();
    /** Set the token type for this node */
    public void setType(int ttype);
    /** Get the token text for this node */
    public String getText();
    /** Set the token text for this node */
    public void setText(String text);
    /** Get the first child of this node; null if no children */
    public AST getFirstChild();
    /** Set the first child of a node. */
    public void setFirstChild(AST c);
    /** Get the next sibling in line after this one */
    public AST getNextSibling();
    /** Set the next sibling after this one. */
    public void setNextSibling(AST n);
    /** Add a (rightmost) child to this node */
    public void addChild(AST c);
}
```

Tree grammar rules

As with the SORCERER tool of PCCTS 1.33 and the ANTLR token grammars, tree grammars are collections of EBNF rules embedded with actions, semantic predicates, and syntactic predicates.

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

Each alternative production is composed of a list of elements where an element can be one of the items in a regular ANTLR grammar with the addition of the tree pattern element, which has the form:

```
#( root-token child1 child2 ... childn )
```

For example, the following tree pattern matches a simple PLUS-rooted tree with two INT children:==

```
#( PLUS INT INT )
```

The root of a tree pattern must be a token reference, but the children elements can even be subrules. For example, a common structure is an if-then-else tree where the else-clause statement subtree is optional:

```
#( IF expr stat (stat)? )
```

An important thing to remember when specifying tree patterns and tree grammars in general is that sufficient matches are done not exact matches. As long as the tree satisfies the pattern, a match is reported, regardless of how much is left unparsed. For example, `#(A B)` will report a match for any larger tree with the same structure such as `#(A #(B C) D)`.

Syntactic predicates

ANTLR tree parsers use only a single symbol of lookahead, which is normally not a problem as intermediate forms are explicitly designed to be easy to walk. However, there is occasionally the need to distinguish between similar tree structures. Syntactic predicates can be used to overcome the limitations of limited fixed lookahead. For example, distinguishing between the unary and binary minus operator is best done by using operator nodes of differing token types, but given the same root node, a syntactic predicate can be used to distinguish between these structures:

```
expr:   ( #(MINUS expr expr) )=> #( MINUS expr expr )
      |   #( MINUS expr )
      . . .
      ;
```

The order of evaluation is very important as the second alternative is a "subset" of the first alternative.

Semantic predicates

Semantic predicates at the start of an alternative are simply incorporated into the alternative prediction expressions as with a regular grammar. Semantic predicates in the middle of productions throw exceptions

when they evaluate to false just like a regular grammar.

An Example Tree Walker

Consider how you would build a simple calculator. One approach would be to build a parser that computed expression values as it recognized the input. For the purposes of illustration, we will build a parser that constructs a tree intermediate representation of the input expression and a tree parser that walks the intermediate representation, computing the result.

Our recognizer, `CalcParser`, is defined via the following grammar.

```
class CalcParser extends Parser;
options {
    buildAST = true;    // uses CommonAST by default
}

expr:    mexpr (PLUS^ mexpr)* SEMI!
        ;

mexpr
:    atom (STAR^ atom)*
    ;

atom:    INT
        ;
```

The `PLUS` and `STAR` tokens are considered operators and, hence, subtree roots; they are annotated with the `^` character. The `SEMI` token reference is suffixed with the `!` character to indicate it should not be included in the tree.

The scanner for this calculator is defined as follows:

```
class CalcLexer extends Lexer;

WS      :      ( ' '
              |  '\t'
              |  '\n'
              |  '\r' )
          { _ttype = Token.SKIP; }
        ;

LPAREN:  '('
        ;

RPAREN:  ')'
        ;

STAR:    '*'
        ;

PLUS:    '+'
        ;

SEMI:    ';'
        ;
```

```

INT      :      ('0'..'9')+
          ;

```

The trees generated by this recognizer are simple expression trees. For example, input "3*4+5" results in a tree of the form #(+ (* 3 4) 5). In order to build a tree walker for trees of this form, you have to describe its structure recursively to ANTLR:

```

class CalcTreeWalker extends TreeParser;

expr     :      #(PLUS expr expr)
          |      #(STAR expr expr)
          |      INT
          ;

```

Once the structure has been specified, you are free to embed actions to compute the appropriate result. An easy way to accomplish this is to have the `expr` rule return an integer result and then have each alternative compute the subresult for each subtree. The following tree grammar and actions produces the desired effect:

```

class CalcTreeWalker extends TreeParser;

expr returns [int r]
{
    int a,b;
    r=0;
}
:      #(PLUS a=expr b=expr) {r = a+b;}
|      #(STAR a=expr b=expr) {r = a*b;}
|      i:INT                  {r = Integer.parseInt(i.getText());}
;

```

Notice that no precedence specification is necessary when computing the result of an expression--the structure of the tree encodes this information. That is why intermediate trees are much more than copies of the input in tree form. The input symbols are indeed stored as nodes in the tree, but the structure of the input is encoded as the relationship of those nodes.

The code needed to launch the parser and tree walker is:

```

import java.io.*;
import antlr.CommonAST;
import antlr.collections.AST;

class Calc {
    public static void main(String[] args) {
        try {
            CalcLexer lexer =
                new CalcLexer(new DataInputStream(System.in));
            CalcParser parser = new CalcParser(lexer);
            // Parse the input expression
            parser.expr();
            CommonAST t = (CommonAST)parser.getAST();
            // Print the resulting tree out in LISP notation
            System.out.println(t.toStringList());
        }
    }
}

```

```

        CalcTreeWalker walker = new CalcTreeWalker();
        // Traverse the tree created by the parser
        int r = walker.expr(t);
        System.out.println("value is "+r);
    } catch(Exception e) {
        System.err.println("exception: "+e);
    }
}
}
}

```

Transformations

While tree parsers are useful to examine trees or generate output from a tree, they must be augmented to handle tree transformations. ANTLR tree parsers support the `buildAST` option just like regular parsers; this is analogous to the transform mode of SORCERER. Without programmer intervention, the tree parser will automatically copy the input tree to a result tree. Each rule has an implicit (automatically defined) result tree; the result tree of the start symbol can be obtained from the tree parser via the `getAST` method. 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.

Actions embedded within the rules can set the result tree based upon tests and tree constructions. See the section on [grammar action translations](#).

An Example Tree Transformation

Revisiting the simple Calc example from above, we can perform a few tree transformations instead of computing the expression value. The action in the following tree grammar optimizes away the addition identity operation (addition to zero).

```

class CalcTreeWalker extends TreeParser;
options{
    buildAST = true;    // "transform" mode
}

expr:! #(PLUS left:expr right:expr)
// '!' turns off auto transform
{
    // x+0 = x
    if ( #right.getType()==INT &&
        Integer.parseInt(#right.getText())==0 )
    {
        #expr = #left;
    }
    // 0+x = x
    else if ( #left.getType()==INT &&
             Integer.parseInt(#left.getText())==0 )
    {
        #expr = #right;
    }
    // x+y
    else {
        #expr = #(PLUS, left, right);
    }
}

```

```

        }
    }
    |   #(STAR expr expr) // use auto transformation
    |   i:INT
    ;

```

The code to launch the parser and tree transformer is:

```

import java.io.*;
import antlr.CommonAST;
import antlr.collections.AST;

class Calc {
    public static void main(String[] args) {
        try {
            CalcLexer lexer =
                new CalcLexer(new DataInputStream(System.in));
            CalcParser parser = new CalcParser(lexer);
            // Parse the input expression
            parser.expr();
            CommonAST t = (CommonAST)parser.getAST();
            // Print the resulting tree out in LISP notation
            System.out.println(t.toLispString());

            CalcTreeWalker walker = new CalcTreeWalker();
            // Traverse the tree created by the parser
            walker.expr(t);
            // Get the result tree from the walker
            t = (CommonAST)walker.getAST();
            System.out.println(t.toLispString());
        } catch(Exception e) {
            System.err.println("exception: "+e);
        }
    }
}

```

Examining/Debugging ASTs

Often when developing a tree parser, you will get parse errors. Unfortunately, your trees are usually very large, making it difficult to determine where your AST structure error is. To help the situation (I found it VERY useful when building the Java tree parser), I created an ASTFrame class (a JFrame) that you can use to view your ASTs in a Swing tree view. It does not copy the tree, but uses a TreeModel. Run `antlr.debug.misc.ASTFrame` as an application to test it out or see the new Java grammar `Main.java`. I am not sure it will live in the same package as I'm not sure how debugging etc... will shake out with future ANTLR versions. Here is a simple example usage:

```

public static void main(String args[]) {
    // Create the tree nodes
    ASTFactory factory = new ASTFactory();
    CommonAST r = (CommonAST)factory.create(0, "ROOT");
    r.addChild((CommonAST)factory.create(0, "C1"));
    r.addChild((CommonAST)factory.create(0, "C2"));
    r.addChild((CommonAST)factory.create(0, "C3"));
}

```



```
    ASTFrame frame = new ASTFrame("AST JTree Example", r);  
    frame.setVisible(true);  
}
```

Version: \$Id: //depot/code/org.antlr/release/antlr-2.7.1/doc/sor.html#1 \$



ANTLR

jGuru

Token Vocabularies

Every grammar specifies language structure with rules (substructures) and vocabulary symbols. These symbols are equated with integer "token types" for efficient comparison at run-time. The files that define this mapping from symbol to token type are fundamental to the execution of ANTLR and ANTLR-generated parsers. This document describes the files used and generated by ANTLR plus the options used to control the vocabularies.

Introduction

A parser grammar refers to tokens in its vocabulary by symbol that will correspond to Token objects, generated by the lexer or other token stream, at parse-time. The parser compares a unique integer token type assigned to each symbol against the token type stored in the token objects. If the parser is looking for token type 23, but finds that the first lookahead token's token type, `LT(1).getType()`, is not 23, then the parser throws `MismatchedTokenException`.

A grammar may have an import vocabulary and always has an export vocabulary, which can be referenced by other grammars. Imported vocabularies are never modified and represent the "initial condition" of the vocabulary. Do not confuse `importVocabulary`

The following represent the most common questions:

How does ANTLR decide which vocabulary symbol gets what token type?

Each grammar has a token manager that manages a grammar's export vocabulary. The token manager can be preloaded with symbol / token type pairs by using the grammar `importVocab` option. The option forces ANTLR to look for a file with mappings that look like:

```
PLUS=44
```

Without the `importVocab` option, the grammar's token manager is empty (with one caveat you will see later).

Any token referenced in your grammar that does not have a predefined token type is assigned a type in the order encountered. For example, in the following grammar, tokens A and B will be 4 and 5, respectively:

```
class P extends Parser;
a : A B ;
```

Vocabulary file names are of the form: `NameTokenTypes.txt`.

Why do token types start at 4?

Because ANTLR needs some special token types during analysis. User-defined token types must begin after 3.

What files associated with vocabulary does ANTLR generate?

ANTLR generates `VTokenTypes.txt` and `VTokenTypes.java` for vocabulary `V` where `V` is either the name of the grammar or specified in an `exportVocab=V` option. The

text file is sort of a "freezedried" token manager and represents the persistent state needed by ANTLR to allow a grammar in a different file to see a grammar's vocabulary including string literals etc... The Java file is an interface containing the token type constant definitions. Generated parsers implement one of these interfaces to obtain the appropriate token type definitions.

How does ANTLR synchronize the symbol-type mappings between grammars in the same file and in different files?

The export vocabulary for one grammar must become the import vocabulary for another or the two grammars must share a common import vocabulary.

Imagine a parser P in p.g:

```
// yields PTokenTypes.txt
class P extends Parser;
// options {exportVocab=P;} ---> default!
decl : "int" ID ;
```

and a lexer L in l.g

```
class L extends Lexer;
options {
  importVocab=P; // reads PTokenTypes.txt
}
ID : ('a'..'z')+ ;
```

ANTLR generates LTokenTypes.txt and LTokenTypes.java even though L is primed with values from P's vocabulary.

Grammars in different files that must share the same token type space should use the `importVocab` option to preload the same vocabulary.

If these grammars are in the same file, ANTLR behaves in exactly same way. However, you can get the two grammars to share the vocabulary (allowing them both to contribute to the same token space) by setting their export vocabularies to the same vocabulary name. For example, with P and L in one file, you can do the following:

```
// yields PTokenTypes.txt
class P extends Parser;
// options {exportVocab=P;} ---> default!
decl : "int" ID ;
class L extends Lexer;
options {
  exportVocab=P; // shares vocab P
}
ID : ('a'..'z')+ ;
```

If you leave off the vocab options from L, it will choose to share the first export vocabulary in the file; in this case, it will share P's vocabulary.

```
// yields PTokenTypes.txt
class P extends Parser;
decl : "int" ID ;
// shares P's vocab
```

```
class L extends Lexer;
ID : ('a'..'z')+ ;
```

The token type mapping file looks like this

```
P // exported token vocab name
LITERAL_int="int "=4
ID=5
```

Grammar Inheritance and Vocabularies

Grammars that extend supergrammars inherit rules, actions, and options but what vocabulary does the subgrammar use and what token vocabulary does it use? ANTLR sees the subgrammar as if you had cut and paste all of the nonoverridden rules of the supergrammar into the subgrammar like an include. Therefore, the set of tokens in the subgrammar is the union of the tokens defined in the supergrammar and in the supergrammar. All grammars export a vocabulary file and so the subgrammar will export and use a different vocabulary than the supergrammar. The subgrammar always imports the vocabulary of the supergrammar unless you override it with an `importVocab` option in the subgrammar.

A grammar Q that extends P primes its vocabulary with P's vocabulary as if Q had specified option `importVocab=P`. For example, the following grammar has two token symbols.

```
class P extends Parser;
a : A Z ;
```

The subgrammar, Q, initially has the same vocabulary as the supergrammar, but may add additional symbols.

```
class Q extends P;
f : B ;
```

In this case, Q defines one more symbol, B, yielding a vocabulary for Q of {A,B,C}.

The vocabulary of a subgrammar is always a superset of the supergrammar's vocabulary. Note that overriding rules does not affect the initial vocabulary.

If your subgrammar requires new lexical structures, unused by the supergrammar, you probably need to have the subparser use a sublexer. Override the initial vocabulary with an `importVocab` option that specifies the vocabulary of the sublexer. For example, assume parser P uses PL as a lexer. Without an `importVocab` override, Q's vocabulary would use P's vocab and, consequently, PL's vocabulary. If you would like Q to use token types from another lexer, say QL, do the following:

```
class Q extends P;
options {
  importVocab=QL;
}
f : B ;
```

Q's vocab will now be the same or a superset of QL's vocabulary.

Recognizer Generation Order

If all of your grammars are in one file, you do not have to worry about which grammar file ANTLR should process first, however, you still need to worry about the order in which ANTLR sees the grammars within the file. If you try to import a vocabulary that will be exported by a grammar later in the file, ANTLR will complain that it cannot load the file. The following grammar file will cause antlr to fail:

```
class P extends Parser;
options {
importVocab=L;
}

a : "int" ID;

class L extends Lexer;
ID : 'a';
```

ANTLR will complain that it cannot find LTokenTypes.txt because it has not seen grammar L yet in the grammar file. On the other hand, if you happened to have LTokenTypes.txt lying around (from a previous run of ANTLR on the grammar file when P did not exist?), ANTLR will load it for P and then overwrite it again for L. ANTLR must assume that you want to load a vocabulary generated from another file as it cannot know what grammars are approaching even in the same file.

In general, if you want grammar B to use token types from grammar A (regardless of grammar type), then you must run ANTLR on grammar A first. So, for example, a tree grammar that uses the vocabulary of the parser grammar should be run after ANTLR has generated the parser.

When you want a parser and lexer, for example, to share the same vocabulary space, all you have to do is place them in the same file with their export vocabs pointing at the same place. If they are in separate files, have the parser's import vocab set to the lexer's export vocab unless the parser is contributing lots of literals. In this case, reverse the import/export relationship so the lexer uses the export vocabulary of the parser.

Tricky Vocabulary Stuff

What if your grammars are in separate files and you still want them to share all or part of a token space. There are two solutions: (1) have the grammars import the same vocabulary or (2) have the grammars all inherit from the same base grammar that contains the common token space.

The first solution applies when you have two lexers and two parsers that must parse radically different portions of the input. The example in examples/java/multiLexer of the ANTLR 2.6.0 distribution is such a situation. The javadoc comments are parsed with a different lexer/parser than the regular Java portion of the input. The "*" terminating comment lexical structure is necessarily recognized by the javadoc lexer, but it is natural to have the Java parser enclose the launch of the javadoc parser with open/close token references:

```
javadoc
  : JAVADOC_OPEN
  {
    DemoJavaDocParser jdocparser =
```

```

        new DemoJavaDocParser(getInputState());
        jdocparser.content();
    }
    JAVADOC_CLOSE
;

```

The problem is: the javadoc lexer defines JAVADOC_CLOSE and hence defines its token type. The vocabulary of the Java parser is based upon the Java lexer not the javadoc lexer, unfortunately. To get the javadoc lexer and Java lexer to both see JAVADOC_CLOSE (and have the same token type), have both lexers import a vocabulary file that contains this token type definition. Here are the heads of DemoJavaLexer and DemoJavaDocLexer:

```

class DemoJavaLexer extends Lexer;
options {
    importVocab = Common;
}
...
class DemoJavaDocLexer extends Lexer;
options {
    importVocab = Common;
}
...

```

CommonTokenTypes.txt contains:

```

Common // name of the vocab
JAVADOC_CLOSE=4

```

The second solution to vocabulary sharing applies when you have say one parser and three different lexers (e.g., for various flavors of C). If you only want one parser for space efficiency, then the parser must see the vocabulary of all three lexers and prune out the unwanted structures grammatically (with semantic predicates probably). Given CLexer, GCCLexer, and MSCLexer, make CLexer the supergrammar and have CLexer define the union of all tokens. For example, if MSCLexer needs "_int32" then reserve a token type visible to all lexers in CLexer:

```

tokens {
    INT32;
}

```

In the MSCLexer then, you can actually attach a literal to it.

```

tokens {
    INT32="_int32"
}

```

In this manner, the lexers will all share the same token space allowing you to have a single parser recognize input for multiple C variants.

Version: \$Id: //depot/code/org.antlr/release/antlr-2.7.1/doc/vocab.html#1 \$

Error Handling and Recovery



All syntactic and semantic errors cause parser exceptions to be thrown. In particular, the methods used to match tokens in the parser base class (`match` et al) throw `MismatchedTokenException`. If the lookahead predicts no alternative of a production in either the parser or lexer, then a `NoViableAltException` is thrown. The methods in the lexer base class used to match characters (`match` et al) throw analogous exceptions.

ANTLR ANTLR will generate default error-handling code, or you may specify your own exception handlers. Either case results (where supported by the language) in the creation of a `try/catch` block. Such `try{ }` blocks surround the generated code for the grammar element of interest (rule, alternate, token reference, or rule reference). If no exception handlers (default or otherwise) are specified, then the exception will propagate all the way out of the parser to the calling program.

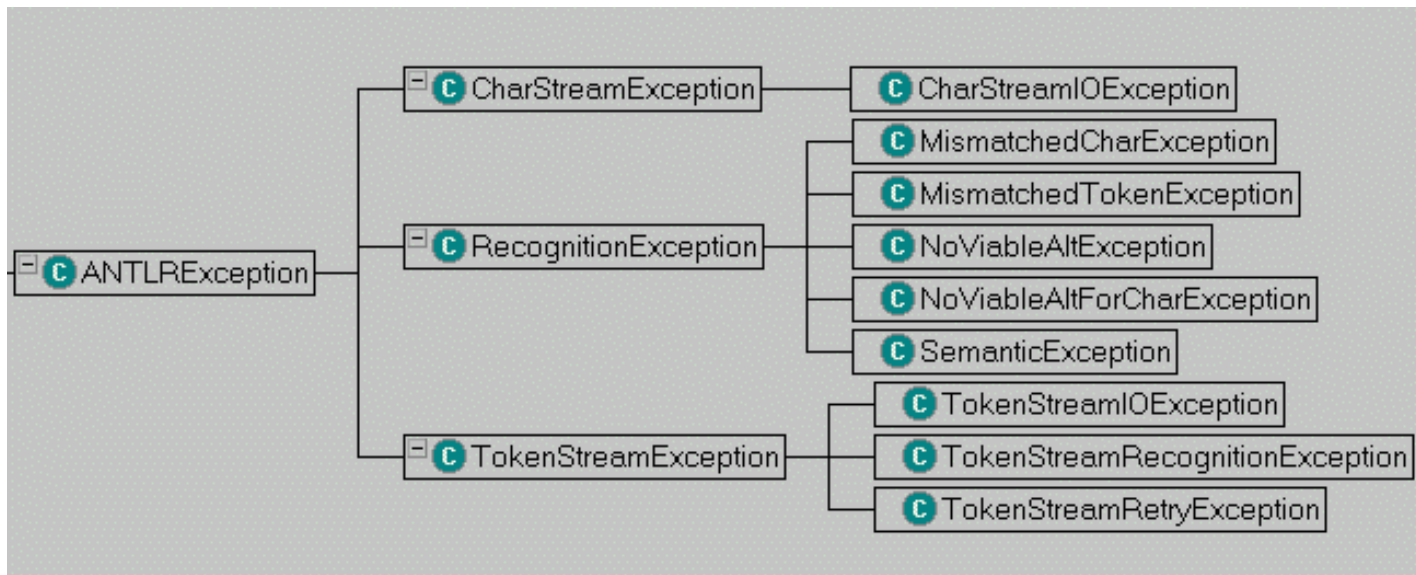
jGuru

ANTLR's default exception handling is good to get something working, but you will have more control over error-reporting and resynchronization if you write your own exception handlers.

Note that the '@' exception specification of PCCTS 1.33 does not apply to ANTLR 2.0.

ANTLR Exception Hierarchy

ANTLR-generated parsers throw exceptions to signal recognition errors or other stream problems. All exceptions derive from `ANTLRException`. The following diagram shows the hierarchy:



Exception	Description
<code>ANTLRException</code>	Root of the exception hierarchy. You can directly subclass this if you want to define your own exceptions unless they live more properly under one of the specific exceptions below.
<code>CharStreamException</code>	Something bad that happens on the character input stream. Most of the time it will be an IO problem, but you could define an exception for input coming from a dialog box or whatever.
<code>CharStreamIOException</code>	The character input stream had an IO exception (e.g., <code>CharBuffer.fill()</code> can throw this). If <code>nextToken()</code> sees this, it will convert it to a <code>TokenStreamIOException</code> .

RecognitionException

A generic recognition problem with the input. Use this as your "catch all" exception in your main() or other method that invokes a parser, lexer, or treeparser. All parser rules throw this exception.

MismatchedCharException

Thrown by CharScanner.match() when it is looking for a character, but finds a different one on the input stream.

MismatchedTokenException

Thrown by Parser.match() when it is looking for a token, but finds a different one on the input stream.

NoViableAltException

The parser finds an unexpected token; that is, it finds a token that does not begin any alternative in the current decision.

NoViableAltForCharException

The lexer finds an unexpected character; that is, it finds a character that does not begin any alternative in the current decision.

SemanticException

Used to indicate syntactically valid, but nonsensical or otherwise bogus input was found on the input stream. This exception is thrown automatically by failed, validating semantic predicates such as:

```
a : A {false}? B ;
```

ANTLR generates:

```
match(A);
if (!(false)) throw new
    SemanticException("false");
match(B);
```

You can throw this exception yourself during the parse if one of your actions determines that the input is wacked.

TokenStreamException

Indicates that something went wrong while generating a stream of tokens.

TokenStreamIOException

Wraps an IOException in a TokenStreamException

TokenStreamRecognitionException

Wraps a RecognitionException in a TokenStreamException so you can pass it along on a stream.

TokenStreamRetryException

Signals aborted recognition of current token. Try to get one again. Used by TokenStreamSelector.retry() to force nextToken() of stream to re-enter and retry. See the examples/java/includeFile directory.

This a great way to handle nested include files and so on or to try out multiple grammars to see which appears to fit the data. You can have something listen on a socket for multiple input types without knowing which type will show up when.

The typical main or parser invoker has try-catch around the invocation:

```
try {
    ...
}
catch(TokenStreamException e) {
    System.err.println("problem with stream: "+e);
}
catch(RecognitionException re) {
    System.err.println("bad input: "+re);
}
```


}

Lexer rules throw `RecognitionException`, `CharStreamException`, and `TokenStreamException`.

Parser rules throw `RecognitionException` and `TokenStreamException`.

Modifying Default Error Messages With Paraphrases

The name or definition of a token in your lexer is rarely meaningful to the user of your recognizer or translator. For example, instead of seeing

```
Error: line(1), expecting ID, found ';' 
```

you can have the parser generate:

```
Error: line(1), expecting an identifier,
      found ';' 
```

ANTLR provides an easy way to specify a string to use in place of the token name. In the definition for ID, use the paraphrase option:

```
ID
options {
  paraphrase = "an identifier";
}
: ('a'..'z' | 'A'..'Z' | '_' )
  ('a'..'z' | 'A'..'Z' | '_' | '0'..'9')*
;
```

Note that this paraphrase goes into the token types text file (ANTLR's persistence file). In other words, a grammar that uses this vocabulary will also use the paraphrase.

Parser Exception Handling

ANTLR 2.0 generates recursive-descent recognizers. Since recursive-descent recognizers operate by recursively calling the rule-matching methods, this results in a call stack that is populated by the contexts of the recursive-descent methods. Parser exception handling for grammar rules is a lot like exception handling in a language like C++ or Java. Namely, when an exception is thrown, the normal thread of execution is stopped, and functions on the call stack are exited sequentially until one is encountered that wants to catch the exception. When an exception is caught, execution resumes at that point.

In ANTLR 2.0, parser exceptions are thrown when (a) there is a syntax error, (b) there is a failed validating semantic predicate, or (c) you throw a parser exception from an action.

In all cases, the recursive-descent functions on the call stack are exited until an exception handler is encountered for that exception type or one of its base classes (in non-object-oriented languages, the hierarchy of exception types is not implemented by a class hierarchy). Exception handlers arise in one of two ways. First, if you do nothing, ANTLR will generate a default exception handler for every parser rule. The default exception handler will report an error, sync to the follow set of the rule, and return from that rule. Second, you may specify your own exception handlers in a variety of ways, as described later.

If you specify an exception handler for a rule, then the default exception handler is not generated for that rule. In addition, you may control the generation of default exception handlers with a [per-grammar or per-rule option](#).

Specifying Parser Exception-Handlers

You may attach exception handlers to a rule, an alternative, or a labeled element. The general form for specifying an exception handler is:

```
exception [label]
```

```

catch [exceptionType exceptionVariable]
  { action }
catch ...
catch ...

```

where the label is only used for attaching exceptions to labeled elements. The `exceptionType` is the exception (or class of exceptions) to catch, and the `exceptionVariable` is the variable name of the caught exception, so that the action can process the exception if desired. Here is an example that catches an exception for the rule, for an alternate and for a labeled element:

```

rule:   a:A B C
      |   D E
        exception // for alternate
          catch [RecognitionException ex] {
            reportError(ex.toString());
          }
      ;
      exception // for rule
        catch [RecognitionException ex] {
          reportError(ex.toString());
        }
      exception[a] // for a:A
        catch [RecognitionException ex] {
          reportError(ex.toString());
        }

```

Note that exceptions attached to alternates and labeled elements **do not** cause the rule to exit. Matching and control flow continues as if the error had not occurred. Because of this, you must be careful not to use any variables that would have been set by a successful match when an exception is caught.

Default Exception Handling in the Lexer

Normally you want the lexer to keep trying to get a valid token upon lexical error. That way, the parser doesn't have to deal with lexical errors and ask for another token. Sometimes you want exceptions to pop out of the lexer--usually when you want to abort the entire parsing process upon syntax error. To get ANTLR to generate lexers that pass on `RecognitionException`'s to the parser as `TokenStreamException`'s, use the `defaultErrorHandler=false` grammar option. Note that IO exceptions are passed back as `TokenStreamIOException`'s regardless of this option.

Here is an example that uses a bogus semantic exception (which is a subclass of `RecognitionException`) to demonstrate blasting out of the lexer:

```

class P extends Parser;
{
public static void main(String[] args) {
    L lexer = new L(System.in);
    P parser = new P(lexer);
    try {
        parser.start();
    }
    catch (Exception e) {
        System.err.println(e);
    }
}
}

```

```

start : "int" ID (COMMA ID)* SEMI ;

```

```
class L extends Lexer;
options {
    defaultErrorHandler=false;
}

{int x=1;}

ID : ('a'..'z')+ ;

SEMI: ';'
    {if ( expr )
     throw new
       SemanticException("test",
                           getFilename(),
                           getLine());} ;

COMMA: ',' ;

WS : (' '|'\n'){newline();}+
    {$setType(Token.SKIP);}
    ;
```

When you type in, say, "int b;" you get the following as output:

```
antlr.TokenStreamRecognitionException: test
Version: $Id: //depot/code/org.antlr/release/antlr-2.7.1/doc/err.html#1 $
```



Java Runtime Model

Programmer's Interface

ANTLR

jGuru

In this section, we describe what ANTLR generates after reading your grammar file and how to use that output to parse input. The classes from which your lexer, token, and parser classes are derived are provided as well.

What ANTLR generates

ANTLR generates the following types of files, where *MyParser*, *MyLexer*, and *MyTreeParser* are names of grammar classes specified in the grammar file. You may have an arbitrary number of parsers, lexers, and tree-parsers per grammar file; a separate class file will be generated for each. In addition, token type files will be generated containing the token vocabularies used in the parsers and lexers. One or more token vocabularies may be defined in a grammar file, and shared between different grammars. For example, given the grammar file:

```
class MyParser extends Parser;
options {
    exportVocab=My;
}
... rules ...
```

```
class MyLexer extends Lexer;
options {
    exportVocab=My;
}
... rules ...
```

```
class MyTreeParser extends TreeParser;
options {
    exportVocab=My;
}
... rules ...
```

The following files will be generated:

- *MyParser.java*. The parser with member methods for the parser rules.
- *MyLexer.java*. The lexer with the member methods for the lexical rules.
- *MyTreeParser.java*. The tree-parser with the member methods for the tree-parser rules.
- *MyTokenTypes.java*. An interface containing all of the token types defined by your parsers and lexers using the exported vocabulary named *My*.
- *MyTokenTypes.txt*. A text file containing all of the token types, literals, and paraphrases defined by parsers and lexers contributing vocabulary *My*.

The programmer uses the classes by referring to them:

1. Create a lexical analyzer. The constructor with no arguments implies that you want to read from standard input.
2. Create a parser and attach it to the lexer (or other TokenStream).
3. Call one of the methods in the parser to begin parsing.

If your parser generates an AST, then get the AST value, create a tree-parser, and invoke one of the tree-parser rules using the AST.

```
MyLexer lex = new MyLexer();
MyParser p =
    new MyParser(lex, user-defined-args-if-any);
p.start-rule();
// and, if you are tree parsing the result...
MyTreeParser tp = new MyTreeParser();
tp.start-rule(p.getAST());
```

You can also specify the name of the token and/or AST objects that you want the lexer/parser to create. Java's support of dynamic programming makes this quite painless:

```
MyLexer lex = new MyLexer();
lex.setTokenObjectClass("MyToken");
// defaults to "antlr.CommonToken"
...
parser.setASTNodeClass("MyASTNode");
// defaults to "antlr.CommonAST"
```

The lexer and parser can cause `IOExceptions` as well as `RecognitionExceptions`, which you must catch:

```
CalcLexer lexer =
    new CalcLexer(new DataInputStream(System.in));
CalcParser parser = new CalcParser(lexer);
// Parse the input expression
try {
    parser.expr();
}
catch (IOException io) {
    System.err.println("IOException");
}
catch (RecognitionException e) {
    System.err.println("exception: "+e);
}
```

Multiple Lexers/Parsers With Shared Input State

Occasionally, you will want two parsers or two lexers to share input state; that is, you will want them to pull input from the same source token stream or character stream. The section on [multiple lexer "states"](#) describes such a situation.

ANTLR 2.6.0 factored the input variables such as line number, guessing state, input stream, etc... into a separate object so that another lexer or parser could share that state. The `LexerSharedInputState` and `ParserSharedInputState` embody this factoring. Method `getInputState()` can be used on either `CharScanner` or `Parser` objects. Here is how to construct two lexers sharing the same input stream:

```
// create Java lexer
JavaLexer mainLexer = new JavaLexer(input);
// create javadoc lexer; attach to shared
// input state of java lexer
JavaDocLexer doclexer =
    new JavaDocLexer(mainLexer.getInputState());
```

Parsers with shared input state can be created similarly:

```
JavaDocParser jdocparser =
    new JavaDocParser(getInputState());
jdocparser.content(); // go parse the comment
```

Sharing state is easy, but what happens upon exception during the execution of the "subparser"? What about syntactic predicate execution? It turns out that invoking a subparser with the same input state is exactly the same as calling another rule in the same parser as far as error handling and syntactic predicate guessing are concerned. If the parser is guessing before the call to the subparser, the subparser must continue guessing, right? Exceptions thrown inside the subparser must exit the subparser and return to enclosing error handler or syntactic predicate handler.

Parser Implementation

Parser Class

ANTLR generates a parser class (an extension of `LLkParser`) that contains a method for every rule in your grammar. The general format looks like:

```
public class MyParser extends LLkParser
    implements MyLexerTokenTypes {
    public T(TokenBuffer tokbuf) {
        super(tokbuf);
    }
    public P(TokenStream lexer) {
        this(lexer, 1);
    }
    ...
    // add your own constructors here...
    rule-definitions
}
```

Parser Methods

ANTLR generates recursive-descent parsers, therefore, every rule in the grammar will result in a method that applies the specified grammatical structure to the input token

stream. The general form of a parser method looks like:

```
public void rule()
    throws RecognitionException,
           TokenStreamException
{
    init-action-if-present
    if ( lookahead-predicts-production-1 ) {
        code-to-match-production-1
    }
    else if ( lookahead-predicts-production-2 ) {
        code-to-match-production-2
    }
    ...
    else if ( lookahead-predicts-production-n ) {
        code-to-match-production-n
    }
    else {
        // syntax error
        throw new NoViableAltException(LT(1));
    }
}
```

This code results from a rule of the form:

```
rule:  production-1
      |  production-2
      |  ...
      |  production-n
      ;
```

If you have specified arguments and a return type for the rule, the method header changes to:

```
/* generated from:
 *   rule(user-defined-args)
 *   returns return-type : ... ;
 */
public return-type rule(user-defined-args)
    throws RecognitionException,
           TokenStreamException
{
    ...
}
```

Token types are integers and we make heavy use of bit sets and range comparisons to avoid excessively-long test expressions.

EBNF Subrules

Subrules are like unlabeled rules, consequently, the code generated for an EBNF subrule mirrors that generated for a rule. The only difference is induced by the EBNF subrule operators that imply optionality or looping.

(. . .) ? optional subrule. The only difference between the code generated for an optional subrule and a rule is that there is no default `else`-clause to throw an exception--the recognition continues on having ignored the optional subrule.

```
{
  init-action-if-present
  if ( lookahead-predicts-production-1 ) {
    code-to-match-production-1
  }
  else if ( lookahead-predicts-production-2 ) {
    code-to-match-production-2
  }
  ...
  else if ( lookahead-predicts-production-n ) {
    code-to-match-production-n
  }
}
```

Not testing the optional paths of optional blocks has the potential to delay the detection of syntax errors.

(. . .) * closure subrule. A closure subrule is like an optional looping subrule, therefore, we wrap the code for a simple subrule in a "forever" loop that exits whenever the lookahead is not consistent with any of the alternative productions.

```
{
  init-action-if-present
loop:
  do {
    if ( lookahead-predicts-production-1 ) {
      code-to-match-production-1
    }
    else if ( lookahead-predicts-production-2 ) {
      code-to-match-production-2
    }
    ...
    else if ( lookahead-predicts-production-n ) {
      code-to-match-production-n
    }
    else {
      break loop;
    }
  }
}
```



```

    while (true);
}

```

While there is no need to explicitly test the lookahead for consistency with the exit path, the grammar analysis phase computes the lookahead of what follows the block. The lookahead of what follows much be disjoint from the lookahead of each alternative otherwise the loop will not know when to terminate. For example, consider the following subrule that is nondeterministic upon token A.

```
( A | B ) * A
```

Upon A, should the loop continue or exit? One must also ask if the loop should even begin. Because you cannot answer these questions with only one symbol of lookahead, the decision is non-LL(1).

Not testing the exit paths of closure loops has the potential to delay the detection of syntax errors.

As a special case, a closure subrule with one alternative production results in:

```

{
    init-action-if-present
loop:
    while ( lookahead-predicts-production-1 ) {
        code-to-match-production-1
    }
}

```

This special case results in smaller, faster, and more readable code.

(. . .) + positive closure subrule. A positive closure subrule is a loop around a series of production prediction tests like a closure subrule. However, we must guarantee that at least one iteration of the loop is done before proceeding to the construct beyond the subrule.

```

{
    int _cnt = 0;
    init-action-if-present
loop:
    do {
        if ( lookahead-predicts-production-1 ) {
            code-to-match-production-1
        }
        else if ( lookahead-predicts-production-2 ) {
            code-to-match-production-2
        }
        ...
        else if ( lookahead-predicts-production-n ) {
            code-to-match-production-n
        }
    }
}

```

```

        else if ( _cnt>1 ) {
            // lookahead predicted nothing and we've
            // done an iteration
            break loop;
        }
        else {
            throw new NoViableAltException(LT(1));
        }
        _cnt++; // track times through the loop
    }
    while (true);
}

```

While there is no need to explicitly test the lookahead for consistency with the exit path, the grammar analysis phase computes the lookahead of what follows the block. The lookahead of what follows must be disjoint from the lookahead of each alternative otherwise the loop will not know when to terminate. For example, consider the following subrule that is nondeterministic upon token A.

$$(A \mid B)^+ A$$

Upon A, should the loop continue or exit? Because you cannot answer this with only one symbol of lookahead, the decision is non-LL(1).

Not testing the exit paths of closure loops has the potential to delay the detection of syntax errors.

You might ask why we do not have a `while` loop that tests to see if the lookahead is consistent with any of the alternatives (rather than having series of tests inside the loop with a `break`). It turns out that we can generate smaller code for a series of tests than one big one. Moreover, the individual tests must be done anyway to distinguish between alternatives so a `while` condition would be redundant.

As a special case, if there is only one alternative, the following is generated:

```

{
    init-action-if-present
    do {
        code-to-match-production-1
    }
    while ( lookahead-predicts-production-1 );
}

```

Optimization. When there are a large (where large is user-definable) number of strictly LL(1) prediction alternatives, then a `switch`-statement can be used rather than a sequence of `if`-statements. The non-LL(1) cases are handled by generating the usual `if`-statements in the default case. For example:

```

switch ( LA(1) ) {
    case KEY_WHILE :

```

```

    case KEY_IF :
    case KEY_DO :
        statement();
        break;
    case KEY_INT :
    case KEY_FLOAT :
        declaration();
        break;
    default :
        // do whatever else-clause is appropriate
}

```

This optimization relies on the compiler building a more direct jump (via jump table or hash table) to the *i*th production matching code. This is also more readable and faster than a series of bit set membership tests.

Production Prediction

LL(1) prediction. Any LL(1) prediction test is a simple set membership test. If the set is a singleton set (a set with only one element), then an integer token type == comparison is done. If the set degree is greater than one, a bit set is created and the single input token type is tested for membership against that set. For example, consider the following rule:

```

a : A | b ;
b : B | C | D | E | F ;

```

The lookahead that predicts production one is {A} and the lookahead that predicts production two is {B, C, D, E, F}. The following code would be generated by ANTLR for rule a (slightly cleaned up for clarity):

```

public void a() {
    if ( LA(1)==A ) {
        match(A);
    }
    else if (token_set1.member(LA(1))) {
        b();
    }
}

```

The prediction for the first production can be done with a simple integer comparison, but the second alternative uses a bit set membership test for speed, which you probably didn't recognize as testing `LA(1) member {B, C, D, E, F}`. The complexity threshold above which bitset-tests are generated is user-definable.

We use arrays of long ints (64 bits) to hold bit sets. The *i*th element of a bitset is stored in the word number $i / 64$ and the bit position within that word is $i \% 64$. The divide and modulo operations are extremely expensive and, but fortunately, a strength reduction can be done. Dividing by a power of two is the same as shifting right and modulo a power of two is the same as masking with that power minus one. All of these

details are hidden inside the implementation of the `BitSet` class in the package `antlr.collections.impl`.

The various bit sets needed by ANTLR are created and initialized in the generated parser (or lexer) class.

Approximate LL(k) prediction. An extension of LL(1)...basically we do a series of up to k bit set tests rather than a single as we do in LL(1) prediction. Each decision will use a different amount of lookahead, with LL(1) being the dominant decision type.

Production Element Recognition

Token references. Token references are translated to:

```
match(token-type);
```

For example, a reference to token `KEY_BEGIN` results in:

```
match(KEY_BEGIN);
```

where `KEY_BEGIN` will be an integer constant defined in the `MyParserTokenType` interface generated by ANTLR.

String literal references. String literal references are references to automatically generated tokens to which ANTLR automatically assigns a token type (one for each unique string). String references are translated to:

```
match(T);
```

where `T` is the token type assigned by ANTLR to that token.

Character literal references. Referencing a character literal implies that the current rule is a lexical rule. Single characters, `'t'`, are translated to:

```
match('t');
```

which can be manually inlined with:

```
if ( c=='t' ) consume();
else throw new MismatchedCharException(
    "mismatched char: '"+(char)c+"'");
```

if the method call proves slow (at the cost of space).

Wildcard references. In lexical rules, the wildcard is translated to:

```
consume();
```

which simply gets the next character of input without doing a test.

References to the wildcard in a parser rule results in the same thing except that the

consume call will be with respect to the parser.

Not operator. When operating on a token, $\sim T$ is translated to:

```
matchNot(T);
```

When operating on a character literal, 't' is translated to:

```
matchNot('t');
```

Range operator. In parser rules, the range operator ($T1..T2$) is translated to:

```
matchRange(T1,T2);
```

In a lexical rule, the range operator for characters $c1..c2$ is translated to:

```
matchRange(c1,c2);
```

Labels. Element labels on atom references become Token references in parser rules and ints in lexical rules. For example, the parser rule:

```
a : id:ID {System.out.println("id is "+id);} ;
```

would be translated to:

```
public void a() {
    Token id = null;
    id = LT(1);
    match(ID);
    System.out.println("id is "+id);
}
```

For lexical rules such as:

```
ID : w:. {System.out.println("w is "+(char)w);} ;
```

the following code would result:

```
public void ID() {
    int w = 0;
    w = c;
    consume(); // match wildcard (anything)
    System.out.println("w is "+(char)w);
}
```

Labels on rule references result in AST references, when generating trees, of the form *label_ast*.

Rule references. Rule references become method calls. Arguments to rules become arguments to the invoked methods. Return values are assigned like Java assignments. Consider rule reference `i=list[1]` to rule:

```
list[int scope] returns int
    :    { return scope+3; }
    ;
```

The rule reference would be translated to:

```
i = list(1);
```

Semantic actions. Actions are translated verbatim to the output parser or lexer except for the [translations required for AST generation](#).

To add members to a lexer or parser class definition, add the class member definitions enclosed in {} immediately following the class specification, for example:

```
class MyParser;
{
    protected int i;
    public MyParser(TokenStream lexer,
        int aUsefulArgument) {
        i = aUsefulArgument;
    }
}
... rules ...
```

ANTLR collects everything inside the {...} and inserts it in the class definition before the rule-method definitions. When generating C++, this may have to be extended to allow actions after the rules due to the wacky ordering restrictions of C++.

Semantic predicates.

Standard Classes

ANTLR constructs parser classes that are subclasses of the `antlr.LLkParser` class, which is a subclass of the `antlr.Parser` class. We summarize the more important members of these classes here. See `Parser.java` and `LLkParser.java` for details of the implementation.

```
public abstract class Parser {
    protected ParserSharedInputState inputState;
    protected ASTFactory ASTFactory;
    public abstract int LA(int i);
    public abstract Token LT(int i);
    public abstract void consume();
    public void consumeUntil(BitSet set) { ... }
    public void consumeUntil(int tokenType) { ... }
    public void match(int t)
        throws MismatchedTokenException { ... }
    public void matchNot(int t)
```

```

        throws MismatchedTokenException { ... }
    ...
}

public class LLkParser extends Parser {
    public LLkParser(TokenBuffer tokenBuf, int k_)
        { ... }
    public LLkParser(TokenStream lexer, int k_)
        { ... }
    public int LA(int i) { return input.LA(i); }
    public Token LT(int i) { return input.LT(i); }
    public void consume() { input.consume(); }
    ...
}

```

Lexer Implementation

Lexer Form

The lexers produced by ANTLR 2.0 are a lot like the parsers produced by ANTLR 2.0. They only major differences are that (a) scanners use characters instead of tokens, and (b) ANTLR generates a special `nextToken` rule for each scanner which is a production containing each public lexer rule as an alternate. The name of the lexical grammar class provided by the programmer results in a subclass of `CharScanner`, for example

```

public class MyLexer extends CharScanner
    implements MyTokenTypes {
    public MyLexer() {
        this(System.in);
    }
    public MyLexer(InputStream in) {
        super(in);
    }
    public Token nextToken() {
        scanning logic
        ...
    }
    recursive and other non-inlined lexical methods
    ...
}

```

When an ANTLR-generated parser needs another token from its lexer, it calls a method called `nextToken`. The general form of the `nextToken` method is:

```

public Token nextToken()
    throws TokenStreamException {
    int tt;

```

```

    for (;;) {
        try {
            resetText();
            switch ( c ) {
                case for each char predicting lexical rule
                call lexical rule gets token type -> tt
                default :
                    throw new NoViableAltForCharException(
                        "bad char: '"+(char)c+"'");
            }
            if ( tt!=Token.SKIP ) {
                return makeToken(tt);
            }
        }
        catch (RecognitionException ex) {
            reportError(ex.toString());
        }
    }
}

```

For example, the lexical rules:

```

lexclass Lex;

WS    : ( '\t' | '\r' | ' ' ) {_ttype=Token.SKIP;} ;
PLUS  : '+';
MINUS : '-';
INT   : ( '0'..'9' )+ ;
ID    : ( 'a'..'z' )+ ;
UID   : ( 'A'..'Z' )+ ;

```

would result in something like:

```

public class Lex extends CharScanner
    implements TTokenTypes {
    ...
    public Token nextToken()
        throws TokenStreamException {
        int _tt = Token.EOF_TYPE;
        for (;;) {
            try {
                resetText();
                switch ( _c ) {
                    case '\t': case '\r': case ' ':
                        _tt=mWS();
                        break;
                    case '+':
                        _tt=mPLUS();
                        break;

```



```

    case '-':
        _tt=mMINUS();
        break;
    case '0': case '1': case '2': case '3':
    case '4': case '5': case '6': case '7':
    case '8': case '9':
        _tt=mINT();
        break;
    case 'a': case 'b': case 'c': case 'd':
    case 'e': case 'f': case 'g': case 'h':
    case 'i': case 'j': case 'k': case 'l':
    case 'm': case 'n': case 'o': case 'p':
    case 'q': case 'r': case 's': case 't':
    case 'u': case 'v': case 'w': case 'x':
    case 'y': case 'z':
        _tt=mID();
        break;
    case 'A': case 'B': case 'C': case 'D':
    case 'E': case 'F': case 'G': case 'H':
    case 'I': case 'J': case 'K': case 'L':
    case 'M': case 'N': case 'O': case 'P':
    case 'Q': case 'R': case 'S': case 'T':
    case 'U': case 'V': case 'W': case 'X':
    case 'Y': case 'Z':
        _tt=mUID();
        break;
    case EOF_CHAR :
        _tt = Token.EOF_TYPE;
        break;
    default :
        throw new NoViableAltForCharException(
            "invalid char "+_c);
    }
    if ( _tt!=Token.SKIP ) {
        return makeToken(_tt);
    }
} // try
    catch (RecognitionException ex) {
        reportError(ex.toString());
    }
} // for
}

public int mWS()
    throws RecognitionException,
        CharStreamException,
        TokenStreamException {
    int _ttype = WS;

```

```

switch ( _c ) {
case '\t':
    match('\t');
    break;
case '\r':
    match('\r');
    break;
case ' ':
    match(' ');
    break;
default :
{
    throw new NoViableAltForException(
        "no viable for char: "+(char)_c);
}
}
_ttype = Token.SKIP;
return _ttype;
}

public int mPLUS()
    throws RecognitionException,
        CharStreamException,
        TokenStreamException {
    int _ttype = PLUS;
    match('+');
    return _ttype;
}

public int mMINUS()
    throws RecognitionException,
        CharStreamException,
        TokenStreamException {

    int _ttype = MINUS;
    match('-');
    return _ttype;
}

public int mINT()
    throws RecognitionException,
        CharStreamException,
        TokenStreamException {

    int _ttype = INT;
    {
    int _cnt=0;
    _loop:

```

```

do {
    if ( _c>='0' && _c<='9' )
        { matchRange('0','9'); }
    else
        if ( _cnt>=1 ) break _loop;
    else {
        throw new ScannerException(
            "no viable alternative for char: "+
                (char)_c);
    }
    _cnt++;
} while (true);
}
return _ttype;
}

public int mID()
    throws RecognitionException,
        CharStreamException,
        TokenStreamException {
    int _ttype = ID;
    {
    int _cnt=0;
    _loop:
    do {
        if ( _c>='a' && _c<='z' )
            { matchRange('a','z'); }
        else
            if ( _cnt>=1 ) break _loop;
        else {
            throw new NoViableAltForCharException(
                "no viable alternative for char: "+
                    (char)_c);
        }
        _cnt++;
    } while (true);
    }
    return _ttype;
}

public int mUID()
    throws RecognitionException,
        CharStreamException,
        TokenStreamException {

    int _ttype = UID;
    {
    int _cnt=0;

```

```

        _loop:
        do {
            if ( _c>='A' && _c<='Z' )
                { matchRange('A','Z'); }
            else
                if ( _cnt>=1 ) break _loop;
            else {
                throw new NoViableAltForCharException(
                    "no viable alternative for char: "+
                    (char)_c);
            }
            _cnt++;
        } while (true);
    }
    return _ttype;
}
}

```

ANTLR-generated lexers assume that you will be reading streams of characters. If this is not the case, you must create your own lexer.

Creating Your Own Lexer

To create your own lexer, your Java class that will doing the lexing must implement interface `TokenStream`, which simply states that you must be able to return a stream of tokens via `nextToken`:

```

/**This interface allows any object to
 * pretend it is a stream of tokens.
 * @author Terence Parr, MageLang Institute
 */
public interface TokenStream {
    public Token nextToken();
}

```

ANTLR will not generate a lexer if you do not specify a lexical class.

Launching a parser with a non-ANTLR-generated lexer is the same as launching a parser with an ANTLR-generated lexer:

```

HandBuiltLexer lex = new HandBuiltLexer(...);
MyParser p = new MyParser(lex);
p.start-rule();

```

The parser does not care what kind of object you use for scanning as long as it can answer `nextToken`.

If you build your own lexer, and the token values are also generated by that lexer, then you should inform the ANTLR-generated parsers about the token type values generated

by that lexer. Use the [importVocab](#) in the parsers that use the externally-generated token set, and create a token definition file following the requirements of the `importVocab` option.

Lexical Rules

Lexical rules are essentially the same as parser rules except that lexical rules apply a structure to a series of characters rather than a series of tokens. As with parser rules, each lexical rule results in a method in the output lexer class.

Alternative blocks. Consider a simple series of alternatives within a block:

```
FORMAT : 'x' | 'f' | 'd';
```

The lexer would contain the following method:

```
public int mFORMAT() {
    if ( c=='x' ) {
        match('x');
    }
    else if ( c=='f' ) {
        match('f');
    }
    else if ( c=='d' ) {
        match('d');
    }
    else {
        throw new NoViableAltForCharException(
            "no viable alternative: '"+(char)c+"'");
    }
    return FORMAT;
}
```

The only real differences between lexical methods and grammar methods are that lookahead prediction expressions do character comparisons rather than `LA(i)` comparisons, `match` matches characters instead of tokens, a `return` is added to the bottom of the rule, and lexical methods throw `CharStreamException` objects in addition to `TokenStreamException` and `RecognitionException` objects.

Optimization: Non-Recursive lexical rules. Rules that do not directly or indirectly call themselves can be inlined into the lexer entry method: `nextToken`. For example, the common identifier rule would be placed directly into the `nextToken` method. That is, rule:

```
ID : ( 'a'..'z' | 'A'..'Z' )+
    ;
```

would not result in a method in your lexer class. This rule would become part of the resulting lexer as it would be probably inlined by ANTLR:

```
public Token nextToken() {
    switch ( c ) {
        cases for operators and such here
        case '0': // chars that predict ID token
        case '1':
        case '2':
        case '3':
        case '4':
        case '5':
        case '6':
        case '7':
        case '8':
        case '9':
            while ( c>='0' && c<='9' ) {
                matchRange('0','9');
            }
            return makeToken(ID);
        default :
            check harder stuff here like rules
            beginning with a..z
    }
}
```

If not inlined, the method for scanning identifiers would look like:

```
public int mID() {
    while ( c>='0' && c<='9' ) {
        matchRange('0','9');
    }
    return ID;
}
```

where token names are converted to method names by prefixing them with the letter m. The nextToken method would become:

```
public Token nextToken() {
    switch ( c ) {
        cases for operators and such here
        case '0': // chars that predict ID token
        case '1':
        case '2':
        case '3':
        case '4':
        case '5':
        case '6':
        case '7':
```

```

    case '8':
    case '9':
        return makeToken(mID());
    default :
        check harder stuff here like rules
        beginning with a..z
}

```

Note that this type of range loop is so common that it should probably be optimized to:

```

while ( c>='0' && c<='9' ) {
    consume();
}

```

Optimization: Recursive lexical rules. Lexical rules that are directly or indirectly recursive are not inlined. For example, consider the following rule that matches nested actions:

```

ACTION
    :   '{' ( ACTION | ~'}' )* '}'
    ;

```

ACTION would be result in (assuming a character vocabulary of 'a..'z', '{', '}'):

```

public int mACTION()
    throws RecognitionException,
           CharStreamException,
           TokenStreamException {

    int _ttype = ACTION;
    match('{');
    {
    _loop:
    do {
        switch ( _c ) {
        case '{':
            mACTION();
            break;
        case 'a': case 'b': case 'c': case 'd':
        case 'e': case 'f': case 'g': case 'h':
        case 'i': case 'j': case 'k': case 'l':
        case 'm': case 'n': case 'o': case 'p':
        case 'q': case 'r': case 's': case 't':
        case 'u': case 'v': case 'w': case 'x':
        case 'y': case 'z':
            matchNot('}');
            break;
        default :
            break _loop;
        }
    }
}

```

```

    }
    } while (true);
    }
    match('}');
    return _ttype;
}

```

Token Objects

The basic token knows only about a token type:

```

public class Token {
    // constants
    public static final int MIN_USER_TYPE = 3;
    public static final int INVALID_TYPE = 0;
    public static final int EOF_TYPE = 1;
    public static final int SKIP = -1;

    // each Token has at least a token type
    int type=INVALID_TYPE;

    // the illegal token object
    public static Token badToken =
        new Token(INVALID_TYPE, "");

    public Token() {;}
    public Token(int t) { type = t; }
    public Token(int t, String txt) {
        type = t; setText(txt);
    }

    public void setType(int t) { type = t; }
    public void setLine(int l) {;}
    public void setColumn(int c) {;}
    public void setText(String t) {;}

    public int getType() { return type; }
    public int getLine() { return 0; }
    public int getColumn() { return 0; }
    public String getText() {...}
}

```

The raw Token class is not very useful. ANTLR supplies a "common" token class that it uses by default, which contains the line number and text associated with the token:

```

public class CommonToken extends Token {
    // most tokens will want line, text information
    int line;
    String text = null;
}

```



```

public CommonToken() {}
public CommonToken(String s) { text = s; }
public CommonToken(int t, String txt) {
    type = t;
    setText(txt);
}

public void setLine(int l) { line = l; }
public int getLine() { return line; }
public void setText(String s) { text = s; }
public String getText() { return text; }
}

```

ANTLR will generate an interface that defines the types of tokens in a token vocabulary. Parser and lexers that share this token vocabulary are generated such that they implement the resulting token types interface:

```

public interface MyLexerTokenTypes {
    public static final int ID = 2;
    public static final int BEGIN = 3;
    ...
}

```

ANTLR defines a token object for use with the `TokenStreamHiddenTokenFilter` object called `CommonHiddenStreamToken`:

```

public class CommonHiddenStreamToken
    extends CommonToken {
    protected CommonHiddenStreamToken hiddenBefore;
    protected CommonHiddenStreamToken hiddenAfter;

    public CommonHiddenStreamToken
        getHiddenAfter() {...}
    public CommonHiddenStreamToken
        getHiddenBefore() {...}
}

```

Hidden tokens are weaved amongst the normal tokens. Note that, for garbage collection reasons, hidden tokens never point back to normal tokens (preventing a linked list of the entire token stream).

Token Lookahead Buffer

The parser must always have fast access to k symbols of lookahead. In a world without syntactic predicates, a simple buffer of k `Token` references would suffice. However, given that even LL(1) ANTLR parsers must be able to backtrack, an arbitrarily-large buffer of `Token` references must be maintained. `LT(i)` looks into the token buffer.

Fortunately, the parser itself does not implement the token-buffering and lookahead

algorithm. That is handled by the `TokenBuffer` object. We begin the discussion of lookahead by providing an LL(k) parser framework:

```
public class LLkParser extends Parser {
    TokenBuffer input;
    public int LA(int i) {
        return input.LA(i);
    }
    public Token LT(int i) {
        return input.LT(i);
    }
    public void consume() {
        input.consume();
    }
}
```

All lookahead-related calls are simply forwarded to the `TokenBuffer` object. In the future, some simple caching may be performed in the parser itself to avoid the extra indirection, or ANTLR may generate the call to `input.LT(i)` directly.

The `TokenBuffer` object caches the token stream emitted by the scanner. It supplies `LT()` and `LA()` methods for accessing the k^{th} lookahead token or token type, as well as methods for consuming tokens, guessing, and backtracking.

```
public class TokenBuffer {
    ...
    /** Mark another token for
     * deferred consumption */
    public final void consume() {...}

    /** Get a lookahead token */
    public final Token LT(int i) { ... }

    /** Get a lookahead token value */
    public final int LA(int i) { ... }

    /**Return an integer marker that can be used to
     * rewind the buffer to its current state. */
    public final int mark() { ... }

    /**Rewind the token buffer to a marker.*/
    public final void rewind(int mark) { ... }
}
```

To begin backtracking, a mark is issued, which makes the `TokenBuffer` record the current position so that it can rewind the token stream. A subsequent rewind directive will reset the internal state to the point before the last mark.

Consider the following rule that employs backtracking:

```

stat:    (list EQUAL) => list EQUAL list
        |    list
        ;
list:    LPAREN (ID)* RPAREN
        ;

```

Something like the following code would be generated:

```

public void stat()
    throws RecognitionException,
           TokenStreamException
{
    boolean synPredFailed;
    if ( LA(1)==LPAREN ) { // check lookahead
        int marker = tokenBuffer.mark();
        try {
            list();
            match(EQUAL);
            synPredFailed = false;
        }
        catch (RecognitionException e) {
            tokenBuffer.rewind(marker);
            synPredFailed = true;
        }
    }
    if ( LA(1)==LPAREN && !synPredFailed ) {
        // test prediction of alt 1
        list();
        match(EQUAL);
        list();
    }
    else if ( LA(1)==LPAREN ) {
        list();
    }
}

```

The token lookahead buffer uses a circular token buffer to perform quick indexed access to the lookahead tokens. The circular buffer is expanded as necessary to calculate $LT(i)$ for arbitrary i . `TokenBuffer.consume()` does not actually read more tokens. Instead, it defers the read by counting how many tokens have been consumed, and then adjusts the token buffer and/or reads new tokens when `LA()` or `LT()` is called.

Version: \$Id: //depot/code/org.antlr/test/antlr-2.7.0a11/doc/runtime.html#3 \$

C++ Notes



The C++ runtime and generated grammars look very much the same as the java ones. There are some subtle differences though, but more on this later.

[ANTLR](#)

Building the runtime

[jGuru](#)

The runtime files are located in the lib/cpp subdirectory of the ANTLR distribution. This release is the first to include preliminary automake/autoconf support. Building it is in general done by doing the following:

```
./configure --prefix=/usr/local
make
```

Installing the runtime is done by typing

```
make install
```

This installs the runtime library libantlr.a in /usr/local/lib and the header files in /usr/local/include/antlr.

Using the runtime

Generally you will compile the ANTLR generated files with something similar to:

```
c++ -c MyParser.cpp -I/usr/local/include
```

Linking is done with something similar to:

```
c++ -o MyExec <your .o files> -L/usr/local/lib -lantlr
```

Getting ANTLR to generate C++

To get ANTLR to generate C++ code you have to add

```
language="Cpp" ;
```

to the global options section. After that things are pretty much the same as in java mode except that all token and AST classes are wrapped by a reference counting class (this to make live easier). The reference counting class uses

```
operator->
```

to reference the object it is wrapping. As a result of this you use -> in C++ mode instead of the '.' of java. See the examples in examples/c++ for some illustrations.

Using Custom AST types

In C++ mode it is also possible to override the AST type used by the code generated by ANTLR. To do this you have to do the following:

- Define a custom AST class like the following:

```
#include <antlr/CommonAST.hpp>

typedef antlr::ASTRefCount<My_AST> RefMyAST;

class MyAST : public antlr::CommonAST {
```



```

public:
    MyAST( void ) : down(), right()
    {
    }
    ~MyAST( void )
    {
    }
    void initialize( antlr::RefToken t )
    {
        antlr::CommonAST::initialize(t);
        // more stuff....
        // ...
    }
    void initialize(int t,const ANTLR_USE_NAMESPACE(std)string& txt)
    {
        setType(t);
        setText(txt);
    }
    void addChild( RefMy_AST c )
    {
        antlr::BaseAST::addChild( static_cast<antlr::RefAST>(c) );
    }
    static antlr::RefAST factory( void )
    {
        antlr::RefAST ret = static_cast<antlr::RefAST>(RefMyAST(new MyAST));
        return ret;
    }
private:
    RefMyAST down;        // are these really necessary...
    RefMyAST right;
};

```

- Tell ANTLR's C++ codegenerator to use your RefMyAST by including the following in the options section:

```
ASTLabelType = "RefMyAST";
```

After that you only need to tell the parser before every invocation of a new instance that it should use the AST factory defined in your class. This is done like this:

```
My_Parser parser(lexer);
parser.setASTNodeFactory( MyAST::factory );
```

If you do not do this only CommonAST objects get created and used as if they were MyAST's. (In future versions this might be done automatically) Now all ANTLR generated code uses RefMyAST/MyAST as type. As a result you can access extra members and methods without typecasting.

Using Heterogeneous AST types

This is largely untested. Small examples seem to work. Functionality from duptree and the likes will not work, this may be fixed in the next release, in general inspection of the trees will work, transformations 90% sure not.. Basically follow the java instructions and look at the generated code. If someone would be willing to share some experiences?

A template grammar file for C++

```
header "pre_include_hpp" {
    // gets inserted before antlr generated includes in the header file
}
```

```

header "post_include_hpp" {
    // gets inserted after antlr generated includes in the header file
    // outside any generated namespace specifications
}

header "pre_include_cpp" {
    // gets inserted after the antlr generated includes in the cpp file
}

header "post_include_cpp" {
    // gets inserted after the antlr generated includes in the cpp file
}

header {
    // gets inserted after generated namespace specifications in the header
    // file. But outside the generated class.
}

options {
    language="Cpp";
    namespace="something";           // encapsulate code in this namespace
// namespaceStd="std";             // cosmetic option to get rid of long defines
//                                 // in generated code
// namespaceAntlr="antlr";         // cosmetic option to get rid of long defines
//                                 // in generated code
    genHashLines = true;           // generated #line's or turn it off.
}

{
    // global stuff in the cpp file
    ...
}
class MyParser extends Parser;
options {
    exportVocab=My;
}
{
    // additional methods and members
    ...
}
... rules ...

{
    // global stuff in the cpp file
    ...
}
class MyLexer extends Lexer;
options {
    exportVocab=My;
}
{
    // additional methods and members
    ...
}
... rules ...

{
    // global stuff in the cpp file

```

```
    ...  
}  
class MyTreeParser extends TreeParser;  
options {  
    exportVocab=My;  
}  
{  
    // additional methods and members  
    ...  
}  
... rules ...
```

Version: \$Id: //depot/code/org.antlr/release/antlr-2.7.1/doc/cpp-runtime.html#3 \$

Sather Runtime Model



Programmer's Interface

ANTLR

In this section, we describe what ANTLR generates after reading your grammar file and how to use that output to parse input. The classes from which your lexer, token, and parser classes are derived are provided as well.

jGuru

What ANTLR generates

ANTLR generates the following types of files, where *MY_PARSER*, *MY_LEXER*, and *MY_TREE_PARSER* are names of grammar classes specified in the grammar file. You may have an arbitrary number of parsers, lexers, and tree-parsers per grammar file; a separate class file will be generated for each. In addition, token type files will be generated containing the token vocabularies used in the parsers and lexers. One or more token vocabularies may be defined in a grammar file, and shared between different grammars. For example, given the grammar file:

```
options {
    language="Sather";
}

class MyParser extends Parser;
options {
    exportVocab=MY;
}
... rules ...

class MY_LEXER extends Lexer;
options {
    exportVocab=MY;
}
... rules ...

class MY_TREE_PARSER extends TreeParser;
options {
    exportVocab=MY;
}
... rules ...
```

The following files will be generated:

- *MY_PARSER.sa*. The parser with member methods for the parser rules.
- *MY_LEXER.sa*. The lexer with the member methods for the lexical rules.
- *MY_TREE_PARSER.sa*. The tree-parser with the member methods for the tree-parser rules.
- *MY_TOKENTYPES.sa*. An interface containing all of the token types defined by your parsers and lexers using the exported vocabulary named MY.
- *MY_TokenTypes.txt*. A text file containing all of the token types, literals, and paraphrases defined by parsers and lexers contributing vocabulary MY.

The programmer uses the classes by referring to them:

1. Create a lexical analyzer.
2. Create a parser and attach it to the lexer (or another \$ANTLR_TOKEN_STREAM).
3. Call one of the methods in the parser to begin parsing.

If your parser generates an AST, then get the AST value, create a tree-parser, and invoke one of the tree-parser rules using the AST.

```
lex ::= #MY_LEXER{ANTLR_COMMON_TOKEN}( file_stream );
parser ::= MY_PARSER{ANTLR_COMMON_TOKEN,ANTLR_COMMON_AST}( lex,
user-defined-args-if-any );
parser.start-rule;
```

```
-- and, if you are tree parsing the result...
tree_parser ::= #MY_TREE_PARSER{ANTLR_COMMON_AST};
tree_parser.start-rule( parser.ast );
```

The lexer and parser can cause exceptions of type \$ANTLR_RECOGNITION_EXCEPTIONS, which you may catch:

```
lexer ::= #CALC_LEXER{ANTLR_COMMON_TOKEN}( file_stream );
parser ::= #CALC_PARSER{ANTLR_COMMON_TOKEN,ANTLR_COMMON_AST}(lexer);
-- Parse the input expression
protect
  parser.expr;
when $ANTLR_RECOGNITION_EXCEPTION
  #ERR + exception.str + "\n";
end;
```

Multiple Lexers/Parsers With Shared Input State

Occasionally, you will want two parsers or two lexers to share input state; that is, you will want them to pull input from the same source token stream or character stream.

ANTLR 2.6.0 factored the input variables such as line number, guessing state, input stream, etc... into a separate object so that another lexer or parser could share that state. The ANTLR_LEXER_SHARED_INPUT_STATE and ANTLR_PARSER_SHARED_INPUT_STATE embody this factoring. Attribute `input_state` can be used on either ANTLR_CHAR_SCANNER or ANTLR_PARSER objects. Here is how to construct two lexers sharing the same input stream:

```
-- create a Java lexer
main_lexer ::= #JAVA_LEXER{ANTLR_COMMON_TOKEN}( input );
-- create javadoc lexer
-- attach to shared input state of java lexer
doclexer ::= #JAVADOC_LEXER{ANTLR_COMMON_TOKEN}( main_lexer.input_state );
```

Parsers with shared input state can be created similarly:

```
jdocparser ::= #JAVA_DOC_PARSER{ANTLR_COMMON_TOKEN,ANTLR_COMMON_AST}( input_state );
jdocparser.content; -- go parse the comment
```

Sharing state is easy, but what happens upon exception during the execution of the "subparser"? What about syntactic predicate execution? It turns out that invoking a subparser with the same input state is exactly the same as calling another rule in the same parser as far as error handling and syntactic predicate guessing are concerned. If the parser is guessing before the call to the subparser, the subparser must continue guessing, right? Exceptions thrown inside the subparser must exit the subparser and return to enclosing error handler or syntactic predicate handler.

Parser Implementation

Parser Class

ANTLR generates a parser class (an extension of ANTLR_LLKPARSER) that contains a method for every rule in your grammar. The general format looks like:

```
class MY_PARSER{ TOKEN < $ANTLR_TOKEN, AST < $ANTLR_AST{AST} } is

  include ANTLR_LLKPARSER{ TOKEN, AST } create -> super_create;
  include CALC_PARSER_TOKENTYPES;

  create ( token_buf : ANTLR_TOKEN_BUFFER{TOKEN} , k : INT ) : SAME is
    res : SAME := super_create( token_buf, k );
    res.token_names := sa_token_names;
    return res;
  end;

  create ( token_buf : ANTLR_TOKEN_BUFFER{TOKEN} ) : SAME is
    return #SAME( token_buf, 1);
  end;
```

```

create ( lexer : $ANTLR_TOKEN_STREAM{TOKEN} , k : INT ) : SAME is
    res : SAME := super_create( lexer, k );
    res.token_names := sa_token_names;
    return res;
end;

create( lexer : $ANTLR_TOKEN_STREAM{TOKEN} ) : SAME is
    res : SAME := #SAME( lexer, 1);
    return res;
end;

create ( state : ANTLR_PARSER_SHARED_INPUT_STATE{TOKEN} ) : SAME is
    res : SAME := super_create( state,1);
    res.token_names := sa_token_names;
    return res;
end;
...
-- add your own constructors here...
rule-definitions
end;

```

Parser Methods

ANTLR generates recursive-descent parsers, therefore, every rule in the grammar will result in a method that applies the specified grammatical structure to the input token stream. The general form of a parser method looks like:

```

rule is
    init-action-if-present
    if ( lookahead-predicts-production-1 ) then
        code-to-match-production-1
    elsif ( lookahead-predicts-production-2 ) then
        code-to-match-production-2
    ...
    elsif ( lookahead-predicts-production-n ) then
        code-to-match-production-n
    else
        -- syntax error
        raise #ANTLR_NO_VIABLE_ALT_EXCEPTION(LT(1));
    end;
end;

```

This code results from a rule of the form:

```

rule:  production-1
      |  production-2
      ...
      |  production-n
      ;

```

If you have specified arguments and a return type for the rule, the method header changes to:

```

(* generated from:
 *   rule(user-defined-args)
 *   returns return-type : ... ;
 *)
rule( user-defined-args ) : return-type is
    ...
end;

```

Token types are integers and we make heavy use of sets and range comparisons to avoid excessively-long test expressions.

EBNF Subrules

Subrules are like unlabeled rules, consequently, the code generated for an EBNF subrule mirrors that generated for a rule. The only difference is induced by the EBNF subrule operators that imply optionality or looping.

(. . .) ? optional subrule. The only difference between the code generated for an optional subrule and a rule is that there is no default `else`-clause to throw an exception--the recognition continues on having ignored the optional subrule.

```

init-action-if-present
if ( lookahead-predicts-production-1 ) then
    code-to-match-production-1

elseif ( lookahead-predicts-production-2 ) then
    code-to-match-production-2

...
elseif ( lookahead-predicts-production-n ) then
    code-to-match-production-n
end;

```

Not testing the optional paths of optional blocks has the potential to delay the detection of syntax errors.

(. . .) * closure subrule. A closure subrule is like an optional looping subrule, therefore, we wrap the code for a simple subrule in a "forever" loop that exits whenever the lookahead is not consistent with any of the alternative productions.

```

init-action-if-present
loop
    if ( lookahead-predicts-production-1 ) then
        code-to-match-production-1

        elseif ( lookahead-predicts-production-2 ) then
            code-to-match-production-2

        ...
        elseif ( lookahead-predicts-production-n ) then
            code-to-match-production-n

    else
        break!;

end;
end;

```

While there is no need to explicitly test the lookahead for consistency with the exit path, the grammar analysis phase computes the lookahead of what follows the block. The lookahead of what follows must be disjoint from the lookahead of each alternative otherwise the loop will not know when to terminate. For example, consider the following subrule that is nondeterministic upon token A.

```
( A | B ) * A
```

Upon A, should the loop continue or exit? One must also ask if the loop should even begin. Because you cannot answer these questions with only one symbol of lookahead, the decision is non-LL(1).

Not testing the exit paths of closure loops has the potential to delay the detection of syntax errors.

As a special case, a closure subrule with one alternative production results in:

```

init-action-if-present
loop while!( lookahead-predicts-production-1 );
    code-to-match-production-1
end;

```

This special case results in smaller, faster, and more readable code.

(...)+ positive closure subrule. A positive closure subrule is a loop around a series of production prediction tests like a closure subrule. However, we must guarantee that at least one iteration of the loop is done before proceeding to the construct beyond the subrule.

```

sa_cnt : INT := 0;
init-action-if-present
loop
  if ( lookahead-predicts-production-1 ) then
    code-to-match-production-1

  elseif ( lookahead-predicts-production-2 ) then
    code-to-match-production-2

  ...
  elseif ( lookahead-predicts-production-n ) then
    code-to-match-production-n

  elseif ( sa_cnt>1 ) then
    -- lookahead predicted nothing and we've
    -- done an iteration
    break!;

  else
    raise #ANTLR_NO_VIABLE_ALT_EXCEPTION(LT(1));

end;
sa_cnt := sa_cnt + 1; -- track times through the loop

end;

```

While there is no need to explicitly test the lookahead for consistency with the exit path, the grammar analysis phase computes the lookahead of what follows the block. The lookahead of what follows must be disjoint from the lookahead of each alternative otherwise the loop will not know when to terminate. For example, consider the following subrule that is nondeterministic upon token A.

$(A \mid B)^+ A$

Upon A, should the loop continue or exit? Because you cannot answer this with only one symbol of lookahead, the decision is non-LL(1).

Not testing the exit paths of closure loops has the potential to delay the detection of syntax errors.

You might ask why we do not have a `while` loop that tests to see if the lookahead is consistent with any of the alternatives (rather than having series of tests inside the loop with a `break`). It turns out that we can generate smaller code for a series of tests than one big one. Moreover, the individual tests must be done anyway to distinguish between alternatives so a `while` condition would be redundant.

As a special case, if there is only one alternative, the following is generated:

```

init-action-if-present
loop
  code-to-match-production-1
  if ( lookahead-predicts-production-1 ) then
    break!;
  end;
end;

```

Optimization. When there are a large (where large is user-definable) number of strictly LL(1) prediction alternatives, then a case-statement can be used rather than a sequence of `if`-statements. The non-LL(1) cases are handled by generating the usual `if`-statements in the `else` case. For example:

```

case ( LA(1) )

```

```

    when KEY_WHILE, KEY_IF, KEY_DO then
        statement;
    when KEY_INT, KEY_FLOAT then
        declaration;
    else
        -- do whatever else-clause is appropriate
    end;
end;

```

This optimization relies on the compiler building a more direct jump (via jump table or hash table) to the *i*th production matching code. This is also more readable and faster than a series of set membership tests.

Production Prediction

LL(1) prediction. Any LL(1) prediction test is a simple set membership test. If the set is a singleton set (a set with only one element), then an integer token type = comparison is done. If the set degree is greater than one, a set is created and the single input token type is tested for membership against that set. For example, consider the following rule:

```

a : A | b ;
b : B | C | D | E | F;

```

The lookahead that predicts production one is {A} and the lookahead that predicts production two is {B, C, D, E, F}. The following code would be generated by ANTLR for rule a (slightly cleaned up for clarity):

```

a is
  if ( LA(1) = A ) then
    match(A);

  elsif ( token_set1.member(LA(1)) ) then
    b;

  end;

end;

```

The prediction for the first production can be done with a simple integer comparison, but the second alternative uses a set membership test for speed, which you probably didn't recognize as testing `LA(1) member {B, C, D, E, F}`. The complexity threshold above which set-tests are generated is user-definable. We use arrays of BOOLS to hold sets. The various sets needed by ANTLR are created and initialized in the generated parser (or lexer) class.

Approximate LL(k) prediction. An extension of LL(1)...basically we do a series of up to *k* set tests rather than a single as we do in LL(1) prediction. Each decision will use a different amount of lookahead, with LL(1) being the dominant decision type.

Production Element Recognition

Token references. Token references are translated to:

```
match(token-type);
```

For example, a reference to token KEY_BEGIN results in:

```
match(KEY_BEGIN);
```

where KEY_BEGIN will be an integer constant defined in the `MY_PARSER_TOKENTYPES` class generated by ANTLR.

String literal references. String literal references are references to automatically generated tokens to which ANTLR automatically assigns a token type (one for each unique string). String references are translated to:

```
match(T);
```

where *T* is the token type assigned by ANTLR to that token.

Character literal references. Referencing a character literal implies that the current rule is a lexical rule. Single characters, 't', are translated to:

```
match( 't' );
```

which can be manually inlined with:

```
if ( c = 't' ) then
  consume;
else
  raise #ANTLR_NO_VIABLE_ALT_FOR_CHAR_EXCEPTION( LA(1), file_name, line );
end;
```

if the method call proves slow (at the cost of space).

Wildcard references. In lexical rules, the wildcard is translated to:

```
consume;
```

which simply gets the next character of input without doing a test.

References to the wildcard in a parser rule results in the same thing except that the `consume` call will be with respect to the parser.

Not operator. When operating on a token, $\sim T$ is translated to:

```
match_not( T );
```

When operating on a character literal, $'t'$ is translated to:

```
match_Not( 't' );
```

Range operator. In parser rules, the range operator ($T1..T2$) is translated to:

```
match_range( T1, T2 );
```

In a lexical rule, the range operator for characters $c1..c2$ is translated to:

```
match_range( c1, c2 );
```

Labels. Element labels on atom references become `TOKENS` references in parser rules and `INTs` in lexical rules. For example, the parser rule:

```
a : id:ID { OUT::create + "id is " + id + '\n'; }
;
```

would be translated to:

```
a is
  id : TOKEN := void;
  id := LT(1);
  match(ID);
  OUT::create + "id is " + id + '\n';
end;
```

For lexical rules such as:

```
ID : w:. { OUT::create + "w is "+ w + '\n'; }
;
```

the following code would result:

```
ID is
  w : CHAR;
  w := c;
  consume; -- match wildcard (anything)
  OUT::create + "w is "+ w + '\n';
end;
```

Labels on rule references result in AST references, when generating trees, of the form `label_ast`.

Rule references. Rule references become method calls. Arguments to rules become arguments to the invoked methods. Return values are assigned like Sather assignments. Consider rule reference `i=list[1]` to rule:

```
list[scope:INT] returns INT
:   { return scope+3; }
;
```

The rule reference would be translated to:

```
i := list(1);
```

Semantic actions. Actions are translated verbatim to the output parser or lexer except for the [translations required for AST generation](#).

To add members to a lexer or parser class definition, add the class member definitions enclosed in `{}` immediately following the class specification, for example:

```
class MY_PARSER extends Parser;
{
  private i : INT;
  create ( lexer : ANTLR_TOKEN_STREAM{TOKEN}, aUsefulArgument : INT ) : SAME is
    i := aUsefulArgument;
  end;
}
... rules ...
```

ANTLR collects everything inside the `{...}` and inserts it in the class definition before the rule-method definitions.

Semantic predicates.

Lexer Implementation

Lexer Form

The lexers produced by ANTLR 2.x are a lot like the parsers produced by ANTLR 2.x. They only major differences are that (a) scanners use characters instead of tokens, and (b) ANTLR generates a special `next_token` rule for each scanner which is a production containing each public lexer rule as an alternate. The name of the lexical grammar class provided by the programmer results in a subclass of `ANTLR_CHARS_CANNER`, for example

```
class MY_LEXER{TOKEN} < $ANTLR_TOKEN_STREAM{TOKEN} , $ANTLR_FILE_CURSOR is

  include ANTLR_CHAR_SCANNER{TOKEN} create -> private char_scanner_create;
  include CALC_PARSER_TOKENTYPES;

  create ( istr : $ISTREAM ) : SAME is
    ...
  end;

  create ( bb : ANTLR_BYTE_BUFFER ) : SAME is
    ...
  end;

  create ( state : ANTLR_LEXER_SHARED_INPUT_STATE ) : SAME is
    ...
  end;

  next_token : TOKEN is
    scanning logic
    ...
  end;
  recursive and other non-inlined lexical methods
```

```
...
```

```
end;
```

When an ANTLR-generated parser needs another token from its lexer, it calls a method called `next_token`. The general form of the `next_token` method is:

```
next_token : TOKEN is
  ss_ttype : INT;
  loop
    protect
      reset_text;
      case ( LA(1) )
      case for each char predicting lexical rule
        call lexical rule gets token type -> sa_ttype
      else
        raise #ANTLR_NO_VIABLE_ALT_FOR_CHAR_EXCEPTION( LA(1), file_name, line );
      end;

      if ( sa_ttype /= ANTLR_COMMON_TOKEN::SKIP ) then
        return make_token( sa_ttype );
      end;

    when $ANTLR_RECOGNITION_EXCEPTION then
      report_error( exception.str );

  end;
end;
end;
```

For example, the lexical rules:

```
class LEX extends Lexer;

WS   : ( '\t' | '\r' | ' ' ) {sa_ttype := ANTLR_COMMON_TOKEN::SKIP;} ;
PLUS : '+';
MINUS: '-';
INT  : ( '0'..'9' )+ ;
ID   : ( 'a'..'z' )+ ;
UID  : ( 'A'..'Z' )+ ;
```

would result in something like:

```
class LEX{TOKEN} < $ANTLR_TOKEN_STREAM{TOKEN} , $ANTLR_FILE_CURSOR is

  next_token : TOKEN is
    sa_rettoken : TOKEN;
    continue : BOOL := true;
    loop
      sa_ttype : INT := ANTLR_COMMON_TOKEN::INVALID_TYPE;
      reset_text;
      protect -- for char stream error handling
        protect -- for lexical error handling
          case ( LA(1) )
          when '\t' , '\r' , ' '
          then
            mWS( true );
            sa_rettoken := sa_return_token;
          when '+'
          then
            mPLUS( true );
            sa_rettoken := sa_return_token;
```

```

        when '-'
        then
            mMINUS( true );
            sa_rettoken := sa_return_token;
        when '0', '1', '2', '3', '4', '5', '6', '7', '8', '9'
        then
            mINT( true );
            sa_rettoken := sa_return_token;
        when 'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l', 'm',
            'n', 'o', 'p', 'q', 'r', 's', 't', 'u', 'v', 'w', 'x', 'y',
'z' then
            mID( true );
            sa_rettoken := sa_return_token;
        when 'A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K', 'L', 'M',
            'N', 'O', 'P', 'Q', 'R', 'S', 'T', 'U', 'V', 'W', 'X', 'Y',
'Z' then
            mUID( true );
            sa_rettoken := sa_return_token;
        else
            -- default
            if ( LA(1) = EOF_CHAR ) then
                upon_eof;
                sa_return_token := make_token( ANTLR_COMMON_TOKEN::EOF_TYPE);
            else
                raise #ANTLR_NO_VIABLE_ALT_FOR_CHAR_EXCEPTION( LA(1), file_name,
line );
            end; -- if
        end; -- case

        if ( ~void(sa_return_token) and continue ) then;
            sa_ttype := sa_return_token.ttype;
            sa_ttype := test_literals_table(sa_ttype);
            sa_return_token.ttype := sa_ttype;
            return sa_return_token;
        end; -- if
        when $ANTLR_RECOGNITION_EXCEPTION then
            report_error( exception );
            consume;
        end; -- protect
        when $ANTLR_CHAR_STREAM_EXCEPTION then
            raise #ANTLR_TOKEN_STREAM_EXCEPTION( exception.message );
        end; -- protect
    end; -- loop
end; -- next_token

mWS( sa_create_token : BOOL ) is
    sa_ttype : INT;
    sa_token : TOKEN;
    sa_begin : INT := text.length;
    sa_ttype := WS;
    sa_save_index : INT;

    case ( LA(1) )
    when '\t'
    then
        match('\t');
    when '\r'
    then
        match('\r');
    when ' '
    then
        match(' ');

```

```

else
    raise #ANTLR_NO_VIABLE_ALT_FOR_CHAR_EXCEPTION( LA(1), file_name, line );
end; -- case

sa_ttype := ANTLR_COMMON_TOKEN::SKIP;
if ( sa_create_token and void(sa_token) and sa_ttype /=
ANTLR_COMMON_TOKEN::SKIP ) then
    sa_token := make_token( sa_ttype );
    sa_token.text := text.substring( sa_begin, text.length - sa_begin );
end; -- if
sa_return_token := sa_token;
end; -- rule

mPLUS( sa_create_token : BOOL ) is
    sa_ttype : INT;
    sa_token : TOKEN;
    sa_begin : INT := text.length;
    sa_ttype := PLUS;
    sa_save_index : INT;

    match('+');
    if ( sa_create_token and void(sa_token) and sa_ttype /=
ANTLR_COMMON_TOKEN::SKIP ) then
        sa_token := make_token( sa_ttype );
        sa_token.text := text.substring( sa_begin, text.length - sa_begin );
    end; -- if
    sa_return_token := sa_token;
end; -- rule

mMINUS( sa_create_token : BOOL ) is
    sa_ttype : INT; sa_token : TOKEN; sa_begin : INT := text.length;
    sa_ttype := MINUS;
    sa_save_index : INT;

    match('-');
    if ( sa_create_token and void(sa_token) and sa_ttype /=
ANTLR_COMMON_TOKEN::SKIP ) then
        sa_token := make_token( sa_ttype );
        sa_token.text := text.substring( sa_begin, text.length - sa_begin );
    end; -- if
    sa_return_token := sa_token;
end; -- rule

mINT( sa_create_token : BOOL ) is
    sa_ttype : INT;
    sa_token : TOKEN;
    sa_begin : INT := text.length;
    sa_ttype := INT;
    sa_save_index : INT;

    sa0_cnt7 : INT := 0;
    loop
        if ((LA(1) >= '0' and LA(1) <= '9')) then
            match_range( '0', '9' );
        else
            if ( sa0_cnt7 >= 1 ) then
                break!
            else
                raise #ANTLR_NO_VIABLE_ALT_FOR_CHAR_EXCEPTION( LA(1), file_name, line
);
            end; -- if

```



```

        end; -- if

        sa0_cnt7 := sa0_cnt7 + 1;
    end; -- loop
    if ( sa_create_token and void(sa_token) and sa_ttype /=
ANTLR_COMMON_TOKEN::SKIP ) then
        sa_token := make_token( sa_ttype );
        sa_token.text := text.substring( sa_begin, text.length - sa_begin );
    end; -- if
    sa_return_token := sa_token;
end; -- rule

mID( sa_create_token : BOOL ) is
    sa_ttype : INT; sa_token : TOKEN; sa_begin : INT := text.length;
    sa_ttype := ID;
    sa_save_index : INT;

    sa1_cnt10 : INT := 0;
    loop
        if (((LA(1) >= 'a' and LA(1) <= 'z'))) then
            match_range( 'a', 'z' );
        else
            if ( sa1_cnt10 >= 1 ) then
                break!
            else
                raise #ANTLR_NO_VIABLE_ALT_FOR_CHAR_EXCEPTION( LA(1), file_name, line
);
            end; -- if
        end; -- if

        sa1_cnt10 := sa1_cnt10 + 1;
    end; -- loop
    if ( sa_create_token and void(sa_token) and sa_ttype /=
ANTLR_COMMON_TOKEN::SKIP ) then
        sa_token := make_token( sa_ttype );
        sa_token.text := text.substring( sa_begin, text.length - sa_begin );
    end; -- if
    sa_return_token := sa_token;
end; -- rule

mUID( sa_create_token : BOOL ) is
    sa_ttype : INT; sa_token : TOKEN; sa_begin : INT := text.length;
    sa_ttype := UID;
    sa_save_index : INT;

    sa2_cnt13 : INT := 0;
    loop
        if (((LA(1) >= 'A' and LA(1) <= 'Z'))) then
            match_range( 'A', 'Z' );
        else
            if ( sa2_cnt13 >= 1 ) then
                break!
            else
                raise #ANTLR_NO_VIABLE_ALT_FOR_CHAR_EXCEPTION( LA(1), file_name, line
);
            end; -- if
        end; -- if

        sa2_cnt13 := sa2_cnt13 + 1;
    end; -- loop
    if ( sa_create_token and void(sa_token) and sa_ttype /=

```

```

ANTLR_COMMON_TOKEN::SKIP ) then
    sa_token := make_token( sa_ttype );
    sa_token.text := text.substring( sa_begin, text.length - sa_begin );
end; -- if
sa_return_token := sa_token;
end; -- rule

end; -- class

```

ANTLR-generated lexers assume that you will be reading streams of characters. If this is not the case, you must create your own lexer.

Creating Your Own Lexer

To create your own lexer, your Sather class that will doing the lexing must implement interface `$ANTLR_TOKEN_STREAM`, which simply states that you must be able to return a stream of tokens conforming to `$ANTLR_TOKEN` via `next_token`:

```

abstract class $ANTLR_TOKEN_STREAM{TOKEN < $ANTLR_TOKEN} is
    next_token : TOKEN;
end;

```

ANTLR will not generate a lexer if you do not specify a lexical class.

Launching a parser with a non-ANTLR-generated lexer is the same as launching a parser with an ANTLR-generated lexer:

```

lex ::= #HAND_BUILT_LEXER{MY_TOKEN}( ... );
p ::= #MY_PARSER{MY_TOKEN, ANTLR_COMMON_AST}( lex );
p.start-rule;

```

The parser does not care what kind of object you use for scanning as long as it can answer `next_token`.

If you build your own lexer, and the token values are also generated by that lexer, then you should inform the ANTLR-generated parsers about the token type values generated by that lexer. Use the [importVocab](#) in the parsers that use the externally-generated token set, and create a token definition file following the requirements of the `importVocab` option.

Lexical Rules

Lexical rules are essentially the same as parser rules except that lexical rules apply a structure to a series of characters rather than a series of tokens. As with parser rules, each lexical rule results in a method in the output lexer class.

Alternative blocks. Consider a simple series of alternatives within a block:

```

FORMAT : 'x' | 'f' | 'd';

```

The lexer would contain the following method:

```

mFORMAT is
    if ( c = 'x' ) then
        match('x');

    elsif ( c = 'f' ) then
        match('f');

    elsif ( c = 'd' ) then
        match('d');

    else
        raise #ANTLR_NO_VIABLE_ALT_FOR_CHAR_EXCEPTION( ... );
    end;

```

```

    return FORMAT;
end;

```

The only real differences between lexical methods and grammar methods are that lookahead prediction expressions do character comparisons rather than `LA(i)` comparisons, `match` matches characters instead of tokens, and a `return` is added to the bottom of the rule.

For example, the common identifier rule would be placed directly into the `next_token` method. That is, rule:

```

ID : ( 'a'..'z' | 'A'..'Z' )+
    ;

```

would not result in a method in your lexer class. This rule would become part of the resulting lexer as it would be probably inlined by ANTLR:

```

next_token : TOKEN is
  case ( LA(1) )
    cases for operators and such here
    -- chars that predict ID token
    case '0', '1', '2', '3', '4', '5', '6', '7', '8', '9' then
      loop while!( c >= '0' and c <= '9' );
        match_range( '0' , '9' );
      end;
      return make_token(ID);
    else
      check harder stuff here like rules
      beginning with a..z
    end;
end;

```

If not inlined, the method for scanning identifiers would look like:

```

mID : TOKEN is
  loop while!( c >= '0' and c <= '9' )
    match_range( '0' , '9' );
  end;
  return ID;
end;

```

where token names are converted to method names by prefixing them with the letter `m`. The `next_token` method would become:

```

next_token : TOKEN is
  case ( c )
    cases for operators and such here
    -- chars that predict ID token
    when '0', '1', '2', '3', '4', '5', '6', '7', '8', '9'
      return make_token( mID );
    else
      check harder stuff here like rules beginning with a..z
    end;
end;

```

Note that this type of range loop is so common that it should probably be optimized to:

```

loop while! ( c >= '0' and c <= '9' );
  consume;
end;

```

Optimization: Recursive lexical rules. Lexical rules that are directly or indirectly recursive are not inlined. For example, consider the following rule that matches nested actions:

```

ACTION
  : '{' ( ACTION | ~'}' )* '}'

```

```
;
```

ACTION would be result in (assuming a character vocabulary of 'a'..'z', '{', '}'):

```
mACTION : INT is
  sa_ttype : INT := ACTION;
  match('{');
  loop
    case ( LA(1) )
    when '{' then
      mACTION;
    when 'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l', 'm',
         'n', 'o', 'p', 'q', 'r', 's', 't', 'u', 'v', 'w', 'x', 'y', 'z' then
      match_not('}');
    else
      break!;
    end;
  end;
  match('}');
  return sa_ttype;
end;
```

Version: \$Id: //depot/code/org.antlr/release/antlr-2.7.1/doc/sa-runtime.html#1 \$



ANTLR

jGuru

ANTLR Tree Construction

ANTLR helps you build intermediate form trees, or abstract syntax trees (ASTs), by providing grammar annotations that indicate what tokens are to be treated as subtree roots, which are to be leaves, and which are to be ignored with respect to tree construction. As with PCCTS 1.33, you may manipulate trees using tree grammar actions.

It is often the case that programmers either have existing tree definitions or need a special physical structure, thus, preventing ANTLR from specifically defining the implementation of AST nodes. ANTLR specifies only an interface describing minimum behavior. Your tree implementation must implement this interface so ANTLR knows how to work with your trees. Further, you must tell the parser the name of your tree nodes or provide a tree "factory" so that ANTLR knows how to create nodes with the correct type (rather than hardcoding in a new `AST()` expression everywhere). ANTLR can construct and walk any tree that satisfies the AST interface. A number of common tree definitions are provided.

Notation

In this and other documents, tree structures are represented by a LISP-like notation, for example:

```
#( A B C )
```

is a tree with A at the root, and children B and C. This notation can be nested to describe trees of arbitrary structure, for example:

```
#( A B #( C D E ) )
```

is a tree with A at the root, B as a first child, and an entire subtree as the second child. The subtree, in turn, has C at the root and D,E as children.

Controlling AST construction

AST construction in an ANTLR Parser, or AST transformation in a Tree-Parser, is turned on and off by the [buildAST option](#).

From an AST construction and walking point of view, ANTLR considers all tree nodes to look the same (i.e., they appear to be homogeneous). Through a tree factory or by specification, however, you can instruct ANTLR to create nodes of different types. See the section below on heterogeneous trees.

Grammar annotations for building ASTs

Leaf nodes

ANTLR assumes that any nonsuffixed token reference or token-range is a leaf node in the resulting tree for the enclosing rule. If no suffixes at all are specified in a grammar, then a Parser will construct a linked-list of the tokens (a degenerate AST), and a Tree-Parser will copy the input AST.

Root nodes

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

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

results in tree `#(C #(B A))`.

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 the flat tree A B C.

Turning off standard tree construction

Suffix a token reference with "!" to prevent incorporation of the node for that token into the resulting tree (the AST node for the token is still constructed and may be referenced in actions, it is just not added to the result tree automatically). Suffix a rule reference "!" to indicate that the tree constructed by the invoked rule should not be linked into the tree constructed for the current rule.

Suffix a rule definition with "!" to indicate that tree construction for the rule is to be turned off. Rules and tokens referenced within that rule still create ASTs, but they are not linked into a result tree. The following rule does no automatic tree construction. Actions must be used to set the return AST value, for example:

```
begin!
  :   INT PLUS i:INT
    { #begin = #(PLUS INT i); }
  ;
```

For finer granularity, prefix alternatives with "!" to shut off tree construction for that alternative only. This granularity is useful, for example, if you have a large number of alternatives and you only want one to have manual tree construction:

```
stat:
  ID EQUALS^ expr // auto construction
  ... some alternatives ...
  |! RETURN expr
    {#stat = #([IMAGINARY_TOKEN_TYPE] expr);}
  ... more alternatives ...
  ;
```

Tree node construction

With automatic tree construction off (but with `buildAST` on), you must construct your own tree nodes and combine them into tree structures within embedded actions. There are several ways to create a tree node in an action:

1. use `new T(arg)` where *T* is your tree node type and *arg* is either a single token type, a token type and token text, or a `Token`.
2. use `ASTFactory.create(arg)` where *T* is your tree node type and *arg* is either a single token type, a token type and token text, or a `Token`. Using the factory is

more general than creating a new node directly, as it defers the node-type decision to the factory, and can be easily changed for the entire grammar.

3. use the shorthand notation `#[TYPE]` or `#[TYPE,"text"]`. The shorthand notation results in a call to `ASTFactory.create()`.
4. use the shorthand notation `#id`, where `id` is either a token matched in the rule, a label, or a rule-reference.

To construct a tree structure from a set of nodes, you can set the first-child and next-sibling references yourself or call the factory `make` method or use `#(. . .)` notation described below.

AST Action Translation

In parsers and tree parsers with `buildAST` set to true, ANTLR will translate portions of user actions in order to make it easier to build ASTs within actions. In particular, the following constructs starting with `#` will be translated:

`#label`

The AST associated with a labeled token-reference or rule-reference may be accessed as `#label`. The translation is to a variable containing the AST node built from that token, or the AST returned from the rule.

`#rule`

When `rule` is the name of the enclosing rule, ANTLR will translate this into the variable containing the result AST for the rule. This allows you to set the return AST for a rule or examine it from within an action. This can be used when AST generation is on or suppressed for the rule or alternate. For example:

```
r! : a:A { #r = #a; }
```

Setting the return tree is very useful in combination with normal tree construction because you can have ANTLR do all the work of building a tree and then add an imaginary root node such as:

```
decl : ( TYPE ID )+
      { #decl = #([DECL,"decl"], #decl); }
      ;
```

ANTLR allows you to assign to `#rule` anywhere within an alternative of the rule. ANTLR ensures that references of and assignments to `#rule` within an action force the parser's internal AST construction variables into a stable state. After you assign to `#rule`, the state of the parser's automatic AST construction variables will be set as if ANTLR had generated the tree rooted at `#rule`. For example, any children nodes added after the action will be added to the children of `#rule`.

`#label_in`

In a tree parser, the **input** AST associated with a labeled token reference or rule reference may be accessed as `#label_in`. The translation is to a variable containing the input-tree AST node from which the rule or token was extracted. Input variables are seldom used. You almost always want to use `#label` instead of `#label_in`.

`#id`

ANTLR supports the translation of unlabeled token references and rule references as a shorthand notation, as long as the token or rule name is unique within the scope of a single alternative. In these cases, the use of an unlabeled token reference or rule reference is identical to using a label. For example, this:

```
r! : A { #r = #A; }
```

is equivalent to:

```
r! : a:A { #r = #a; }
```

`#id_in` is given similar treatment to `#label_in`.

```
#[TOKEN_TYPE] or #[TOKEN_TYPE, "text"]
```

AST node constructor shorthand. The translation is a call to the `ASTFactory.create()` method. For example, `#[T]` is translated to:

```
ASFFactory.create(T)
```

```
 #(root, c1, ..., cn)
```

AST tree construction shorthand. ANTLR looks for the comma character to separate the tree arguments. Commas within method call tree elements are handled properly; i.e., an element of `"foo(#a, 34)"` is ok and will not conflict with the comma separator between the other tree elements in the tree. This tree construct is translated to a "make tree" call. The "make-tree" call is complex due to the need to simulate variable arguments in languages like Java, but the result will be something like:

```
ASTFactory.make(root, c1, ..., cn);
```

In addition to the translation of the `#(...)` as a whole, the root and each child `c1..cn` will be translated. Within the context of a `#(...)` construct, you may use:

- `id` or `label` as a shorthand for `#id` or `#label`.
- `[...]` as a shorthand for `#[...]`.
- `(...)` as a shorthand for `#(...)`.

The target code generator performs this translation with the help of a special lexer that parses the actions and asks the code-generator to create appropriate substitutions for each translated item.

Invoking parsers that build trees

Assuming that you have defined a lexer `L` and a parser `P` in your grammar, you can invoke them sequentially on the system input stream as follows.

```
L lexer = new L(System.in);
P parser = new P(lexer);
parser.setASTNodeType("MyAST");
parser.startRule();
```

If you have set `buildAST=true` in your parser grammar, then it will build an AST, which can be accessed via `parser.getAST()`. If you have defined a tree parser called `T`, you

can invoke it with:

```
T walker = new T();
walker.startRule(parser.getAST()); // walk tree
```

If, in addition, you have set `buildAST=true` in your tree-parser to turn on transform mode, then you can access the resulting AST of the tree-walker:

```
AST results = walker.getAST();
DumpASTVisitor visitor = new DumpASTVisitor();
visitor.visit(results);
```

Where `DumpASTVisitor` is a predefined `ASTVisitor` implementation that simply prints the tree to the standard output.

You can also use get a LISP-like print out of a tree via

```
String s = parser.getAST().toStringList();
```

AST Factories

ANTLR uses a factory pattern to create and connect AST nodes. This is done to primarily to separate out the tree construction facility from the parser, but also gives you a hook in between the parser and the tree node construction. Subclass `ASTFactory` to alter the create methods.

If you are only interested in specifying the AST node type at runtime, use the

```
setASTNodeClass(String className)
```

method on the parser or factory. By default, trees are constructed of nodes of type `CommonAST`.

The `ASTFactory` has some generically useful methods:

```
/** Copy a single node. clone() is not used
 * because we want to return an AST not a plain
 * object...type safety issue. Further, we want
 * to have all AST node creation go through the
 * factory so creation can be tracked. Returns
 * null if t is null.
 */
public AST dup(AST t);

/** Duplicate tree including siblings
 * of root.
 */
public AST dupList(AST t);

/** Duplicate a tree, assuming this is a
 * root node of a tree--duplicate that node
 * and what's below; ignore siblings of root
 * node.
 */
public AST dupTree(AST t);
```

Heterogeneous ASTs

Each node in an AST must encode information about the kind of node it is; for example, is it an ADD operator or a leaf node such as an INT? There are two ways to encode this: with a token type or with a Java (or C++ etc...) class type. In other words, do you have a single class type with numerous token types or no token types and numerous classes? For lack of better terms, I (Terence) have been calling ASTs with a single class type *homogeneous* trees and ASTs with many class types *heterogeneous* trees.

The only reason to have a different class type for the various kinds of nodes is for the case where you want to execute a bunch of hand-coded tree walks or your nodes store radically different kinds of data. The example I use below demonstrates an expression tree where each node overrides `value()` so that `root.value()` is the result of evaluating the input expression. From the perspective of building trees and walking them with a generated tree parser, it is best to consider every node as an identical AST node. Hence, the schism that exists between the hetero- and homogeneous AST camps.

ANTLR supports both kinds of tree nodes--at the same time! If you do nothing but turn on the "buildAST=true" option, you get a homogeneous tree. Later, if you want to use physically separate class types for some of the nodes, just specify that in the grammar that builds the tree. Then you can have the best of both worlds--the trees are built automatically, but you can apply different methods to and store different data in the various nodes. Note that the structure of the tree is unaffected; just the type of the nodes changes.

ANTLR applies a "scoping" sort of algorithm for determining the class type of a particular AST node that it needs to create. The default type is `CommonAST` unless, prior to parser invocation, you override that with a call to:

```
myParser.setASTNodeClass("com.acme.MyAST");
```

In the grammar, you can override the default class type by setting the type for nodes created from a particular input token. Use the element option `<AST=typename>` in the `tokens` section:

```
tokens {
    PLUS<AST=PLUSNode>;
    ...
}
```

You may further override the class type by annotating a particular token reference in your parser grammar:

```
anInt : INT<AST=INTNode> ;
```

This reference override is super useful for tokens such as `ID` that you might want converted to a `TYPENAME` node in one context and a `VARREF` in another context.

ANTLR uses the AST factory to create nodes for which it does not know a specific type. In other words, ANTLR generates code similar to the following:

```
AST tmp2_AST = (AST)astFactory.create(LT(1));
```

On the other hand, if you specify a class to use, either in the `tokens` section or on a particular reference, ANTLR generates the more appropriate:


```
INTNode tmp3_AST = new INTNode(LT(1));
```

Besides being faster and more obvious, this code alleviates another problem with homogeneous ASTs: you have to cast like mad from your type to AST and back (though ANTLR does let you set the AST label with grammar option `ASTLabelType`).

An Expression Tree Example

This example includes a parser that constructs expression ASTs, the usual lexer, and some AST node class definitions.

Let's start by describing the AST structure and node types. Expressions have plus and multiply operators and integers. The operators will be subtree roots (nonleaf nodes) and integers will be leaf nodes. For example, input `3+4*5+21` yields a tree with structure:

```
( + ( + 3 ( * 4 5 ) ) 21 )
```

or:

```

+
|
+--21
|
3--*
   |
   4--5

```

All AST nodes are subclasses of `CalcAST`, which are `BaseAST`'s that also answer method `value()`. Method `value()` evaluates the tree starting at that node. Naturally, for integer nodes, `value()` will simply return the value stored within that node. Here is `CalcAST`:

```
public abstract class CalcAST
    extends antlr.BaseAST
{
    public abstract int value();
}
```

The AST operator nodes must combine the results of computing the value of their two subtrees. They must perform a depth-first walk of the tree below them. For fun and to make the operations more obvious, the operator nodes define `left()` and `right()` instead, making them appear even more different than the normal child-sibling tree representation. Consequently, these expression trees can be treated as both homogeneous child-sibling trees and heterogeneous expression trees.

```
public abstract class BinaryOperatorAST extends
    CalcAST
{
    /** Make me look like a heterogeneous tree */
    public CalcAST left() {
        return (CalcAST)getFirstChild();
    }

    public CalcAST right() {
```

```

        CalcAST t = left();
        if ( t==null ) return null;
        return (CalcAST)t.getNextSibling();
    }
}

```

The simplest node in the tree looks like:

```

import antlr.BaseAST;
import antlr.Token;
import antlr.collections.AST;
import java.io.*;

/** A simple node to represent an INT */
public class INTNode extends CalcAST {
    int v=0;

    public INTNode(Token tok) {
        v = Integer.parseInt(tok.getText());
    }

    /** Compute value of subtree; this is
     * heterogeneous part :)
     */
    public int value() {
        return v;
    }

    public String toString() {
        return " "+v;
    }

    // satisfy abstract methods from BaseAST
    public void initialize(int t, String txt) {
    }
    public void initialize(AST t) {
    }
    public void initialize(Token tok) {
    }
}

```

The operators derive from BinaryOperatorAST and define value() in terms of left() and right(). For example, here is PLUSNode:

```

import antlr.BaseAST;
import antlr.Token;
import antlr.collections.AST;
import java.io.*;

/** A simple node to represent PLUS operation */
public class PLUSNode extends BinaryOperatorAST {
    public PLUSNode(Token tok) {

```

```

    }

    /** Compute value of subtree;
     * this is heterogeneous part :)
     */
    public int value() {
        return left().value() + right().value();
    }

    public String toString() {
        return " +";
    }

    // satisfy abstract methods from BaseAST
    public void initialize(int t, String txt) {
    }
    public void initialize(AST t) {
    }
    public void initialize(Token tok) {
    }
}

```

The parser is pretty straightforward except that you have to add the options to tell ANTLR what node types you want to create for which token matched on the input stream. The tokens section lists the operators with element option AST appended to their definitions. This tells ANTLR to build PLUSNode objects for any PLUS tokens seen on the input stream, for example. For demonstration purposes, INT is not included in the tokens section--the specific token reference is suffixed with the element option to specify that nodes created from that INT should be of type INTNode (of course, the effect is the same as there is only that one reference to INT).

```

class CalcParser extends Parser;
options {
    buildAST = true; // uses CommonAST by default
}

// define a bunch of specific AST nodes to build.
// can override at actual reference of tokens in
// grammar below.
tokens {
    PLUS<AST=PLUSNode>;
    STAR<AST=MULTNode>;
}

expr:    mexpr (PLUS^ mexpr)* SEMI!
        ;

mexpr
:    atom (STAR^ atom)*
        ;

```

```
// Demonstrate token reference option
atom:  INT<AST=INTNode>
      ;
```

Invoking the parser is done as usual. Computing the value of the resulting AST is accomplished by simply calling method `value()` on the root.

```
import java.io.*;
import antlr.CommonAST;
import antlr.collections.AST;

class Main {
    public static void main(String[] args) {
        try {
            CalcLexer lexer =
                new CalcLexer(
                    new DataInputStream(System.in)
                );
            CalcParser parser =
                new CalcParser(lexer);
            // Parse the input expression
            parser.expr();
            CalcAST t = (CalcAST)parser.getAST();

            System.out.println(t.toStringTree());

            // Compute value and return
            int r = t.value();
            System.out.println("value is "+r);
        } catch(Exception e) {
            System.err.println("exception: "+e);
            e.printStackTrace();
        }
    }
}
```

For completeness, here is the lexer:

```
class CalcLexer extends Lexer;

WS   :   ( ' '
        |   '\t'
        |   '\n'
        |   '\r' )
        { $setType(Token.SKIP); }
      ;

LPAREN: '(' ;

RPAREN: ')' ;
```

```

STAR:    '*' ;

PLUS:    '+' ;

SEMI:    ';' ;

protected
DIGIT
    :    '0'..'9' ;

INT :    (DIGIT)+ ;

```

Describing Heterogeneous Trees With Grammars

So what's the difference between this approach and default homogeneous tree construction? The big difference is that you need a tree grammar to describe the expression tree and compute resulting values. But, that's a good thing as it's "executable documentation" and negates the need to handcode the tree parser (the `value()` methods). If you used homogeneous trees, here is all you would need beyond the parser/lexer to evaluate the expressions: [*This code comes from the examples/java/calc directory.*]

```

class CalcTreeWalker extends TreeParser {

    expr returns [float r]
    {
        float a,b;
        r=0;
    }
    :    #(PLUS a=expr b=expr)    {r = a+b;}
    |    #(STAR a=expr b=expr)    {r = a*b;}
    |    i:INT
        {r = (float)
         Integer.parseInt(i.getText());}
    ;
}

```

Because Terence wants you to use tree grammars even when constructing heterogeneous ASTs (to avoid handcoding methods that implement a depth-first-search), implement the following methods in your various heterogeneous AST node class definitions:

```

/** Get the token text for this node */
public String getText();
/** Get the token type for this node */
public int getType();

```

That is how you can use heterogeneous trees with a tree grammar. Note that your token types must match the PLUS and STAR token types imported from your parser. I.e., make sure `PLUSNode.getType()` returns `CalcParserTokenTypes.PLUS`. The token types are generated by ANTLR in interface files that look like:

```

public interface CalcParserTokenTypes {
    ...
    int PLUS = 4;
    int STAR = 5;
}

```

```

    ...
}

```

AST (XML) Serialization

[Oliver Zeigermann olli@zeigermann.de provided the initial implementation of this serialization. His [XTAL](#) XML translation code is worth checking out; particularly for reading XML-serialized ASTs back in.]

For a variety of reasons, you may want to store an AST or pass it to another program or computer. Class `antlr.BaseAST` is `Serializable` using the Java code generator, which means you can write ASTs to the disk using the standard Java stuff. You can also write the ASTs out in XML form using the following methods from `BaseAST`:

- `public void xmlSerialize(Writer out)`
- `public void xmlSerializeNode(Writer out)`
- `public void xmlSerializeRootOpen(Writer out)`
- `public void xmlSerializeRootClose(Writer out)`

All methods throw `IOException`.

You can override `xmlSerializeNode` and so on to change the way nodes are written out. By default the serialization uses the class type name as the tag name and has attributes `text` and `type` to store the text and token type of the node.

The output from running the simple heterogeneous tree example, `examples/java/heteroAST`, yields:

```

( + ( + 3 ( * 4 5 ) ) 21 )
<PLUS><PLUS><int>3</int><MULT>
<int>4</int><int>5</int>
</MULT></PLUS><int>21</int></PLUS>
value is 44

```

The LISP-form of the tree shows the structure and contents. The various heterogeneous nodes override the open and close tags and change the way leaf nodes are serialized to use `<int>value</int>` instead of tag attributes of a single node.

Here is the code that generates the XML:

```

Writer w = new OutputStreamWriter(System.out);
t.xmlSerialize(w);
w.write("\n");
w.flush();

```

AST enumerations

The AST `findAll` and `findAllPartial` methods return enumerations of tree nodes that you can walk. Interface

```
antlr.collections.ASTEnumeration
```

and

```

class antlr.Collections.impl.ASTEnumerator

implement this functionality. Here is an example:
// Print out all instances of
// a-subtree-of-interest
// found within tree 't'.
ASTEnumeration enum;
enum = t.findAll(a-subtree-of-interest);
while ( enum.hasMoreNodes() ) {
    System.out.println(
        enum.nextNode().toStringList()
    );
}

```

A few examples

```

sum :term ( PLUS^ term)*
    ;

```

The "^" suffix on the PLUS tells ANTLR to create an additional node and place it as the root of whatever subtree has been constructed up until that point for rule `sum`. The subtrees returned by the `term` references are collected as children of the addition nodes. If the subrule is not matched, the associated nodes would not be added to the tree. The rule returns either the tree matched for the first `term` reference or a PLUS-rooted tree.

The grammar annotations should be viewed as operators, not static specifications. In the above example, each iteration of the `(...)*` will create a new PLUS root, with the previous tree on the left, and the tree from the new `term` on the right, thus preserving the usual associativity for "+".

Look at the following rule that turns off default tree construction.

```

decl!:
    modifiers type ID SEMI;
    { #decl = #([DECL], ID, ([TYPE] type),
                ([MOD] modifiers) ); }
    ;

```

In this example, a declaration is matched. The resulting AST has an "imaginary" DECL node at the root, with three children. The first child is the ID of the declaration. The second child is a subtree with an imaginary TYPE node at the root and the AST from the `type` rule as its child. The third child is a subtree with an imaginary MOD at the root and the results of the `modifiers` rule as its child.

Labeled subrules

[THIS WILL NOT BE IMPLEMENTED AS LABELED SUBRULES...We'll do something else eventually.]

In 2.00 ANTLR, each rule has exactly one tree associated with it. Subrules simply add

elements to the tree for the enclosing rule, which is normally what you want. For example, expression trees are easily built via:

```
expr: ID ( PLUS^ ID )*
      ;
```

However, many times you want the elements of a subrule to produce a tree that is independent of the rule's tree. Recall that exponents must be computed before coefficients are multiplied in for exponent terms. The following grammar matches the correct syntax.

```
// match exponent terms such as "3*x^4"
eterm
  :   expr MULT ID EXPONENT expr
      ;
```

However, to produce the correct AST, you would normally split the ID EXPONENT expr portion into another rule like this:

```
eterm:
    expr MULT^ exp
    ;

exp:
    ID EXPONENT^ expr
    ;
```

In this manner, each operator would be the root of the appropriate subrule. For input $3 * x^4$, the tree would look like:

```
 #(MULT 3 #(EXPONENT ID 4))
```

However, if you attempted to keep this grammar in the same rule:

```
eterm
  :   expr MULT^ (ID EXPONENT^ expr)
      ;
```

both "^" root operators would modify the same tree yielding

```
 #(EXPONENT #(MULT 3 ID) 4)
```

This tree has the operators as roots, but they are associated with the wrong operands.

Using a labeled subrule allows the original rule to generate the correct tree.


```

eterm
    :   expr MULT^ e:(ID EXPONENT^ expr)
    ;

```

In this case, for the same input $3*x^4$, the labeled subrule would build up its own subtree and make it the operand of the MULT tree of the eterm rule. The presence of the label alters the AST code generation for the elements within the subrule, making it operate more like a normal rule. Annotations of "^" make the node created for that token reference the root of the tree for the e subrule.

Labeled subrules have a result AST that can be accessed just like the result AST for a rule. For example, we could rewrite the above decl example using labeled subrules (note the use of ! at the start of the subrules to suppress automatic construction for the subrule):

```

decl!:
    m:(! modifiers { #m = #([MOD] modifiers); } )
    t:(! type { #t = #([TYPE] type); } )
    ID
    SEMI;
    { #decl = #( [DECL] ID t m ); }
    ;

```

What about subrules that are closure loops? The same rules apply to a closure subrule--there is a single tree for that loop that is built up according to the AST operators annotating the elements of that loop. For example, consider the following rule.

```

term:   T^ i:(OP^ expr)+
    ;

```

For input T OP A OP B OP C, the following tree structure would be created:

```

#(T #(OP #(OP #(OP A) B) C) )

```

which can be drawn graphically as

```

T
|
OP
|
OP--C
|
OP--B
|
A

```

The first important thing to note is that each iteration of the loop in the subrule operates on

the same tree. The resulting tree, after all iterations of the loop, is associated with the subrule label. The result tree for the above labeled subrule is:

```
 #(OP #(OP #(OP A) B) C)
```

The second thing to note is that, because T is matched first and there is a root operator after it in the rule, T would be at the bottom of the tree if it were not for the label on the subrule.

Loops will generally be used to build up lists of subtree. For example, if you want a list of polynomial assignments to produce a sibling list of ASSIGN subtrees, then the following rule you would normally split into two rules.

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

Normally, the following would be required

```
interp
  : ( assign )+
  ;
assign
  : ID ASSIGN^ poly "!"
  ;
```

Labeling a subrule allows you to write the above example more easily as:

```
interp
  : ( r:(ID ASSIGN^ poly ";") )+
  ;
```

Each recognition of a subrule results in a tree and if the subrule is nested in a loop, all trees are returned as a list of trees (i.e., the roots of the subtrees are siblings). If the labeled subrule is suffixed with a "!", then the tree(s) created by the subrule are not linked into the tree for the enclosing rule or subrule.

Labeled subrules within labeled subrules result in trees that are linked into the surrounding subrule's tree. For example, the following rule results in a tree of the form X #(A #(B C) D) Y.

```
a    : X r:( A^ s:(B^ C) D) Y
      ;
```

Labeled subrules within nonlabeled subrules result in trees that are linked into the surrounding rule's tree. For example, the following rule results in a tree of the form #(A X #(B C) D Y).

```
a    :    X ( A^ s:(B^ C) D) Y
      ;
```

Reference nodes

Not implemented. A node that does nothing but refer to another node in the tree. Nice for embedding the same tree in multiple lists.

Required AST functionality and form

The data structure representing your trees can have any form or type name as long as they implement the AST interface:

```
package antlr.collections;

/** Minimal AST node interface used by ANTLR
 *  AST generation and tree-walker.
 */
public interface AST {
    /** Get the token type for this node */
    public int getType();

    /** Set the token type for this node */
    public void setType(int ttype);

    /** Get the token text for this node */
    public String getText();

    /** Set the token text for this node */
    public void setText(String text);

    /** Get the first child of this node;
     *  null if no children */
    public AST getFirstChild();

    /** Set the first child of a node */
    public void setFirstChild(AST c);

    /** Get the next sibling in line after this
     *  one
     */
    public AST getNextSibling();

    /** Set the next sibling after this one */
    public void setNextSibling(AST n);

    /** Add a (rightmost) child to this node */
    public void addChild(AST node);
```

```
    /** Are two nodes exactly equal? */
    public boolean equals(AST t);
    /** Are two lists of nodes/subtrees exactly
     * equal in structure and content? */
    public boolean equalsList(AST t);
    /** Are two lists of nodes/subtrees
     * partially equal? In other words, 'this'
     * can be bigger than 't'
     */
    public boolean equalsListPartial(AST t);
    /** Are two nodes/subtrees exactly equal? */
    public boolean equalsTree(AST t);
    /** Are two nodes/subtrees exactly partially
     * equal? In other words, 'this' can be
     * bigger than 't'.
     */
    public boolean equalsTreePartial(AST t);
    /** Return an enumeration of all exact tree
     * matches for tree within 'this'.
     */
    public ASTEnumeration findAll(AST tree);
    /** Return an enumeration of all partial
     * tree matches for tree within 'this'.
     */
    public ASTEnumeration findAllPartial(
        AST subtree);
    /** Init a node with token type and text */
    public void initialize(int t, String txt);
    /** Init a node using content from 't' */
    public void initialize(AST t);
    /** Init a node using content from 't' */
    public void initialize(Token t);
    /** Convert node to printable form */
    public String toString();
    /** Treat 'this' as list (i.e.,
     * consider 'this'
     * siblings) and convert to printable
     * form
     */
    public String toStringList();
    /** Treat 'this' as tree root
     * (i.e., don't consider
     * 'this' siblings) and convert
     * to printable form */
    public String toStringTree();
}
```

This scheme does not preclude the use of heterogeneous trees versus homogeneous trees. However, you will need to write extra code to create heterogeneous trees (via a subclass of `ASTFactory`) or by specifying the node types at the token reference sites or in the `tokens` section, whereas the homogeneous trees are free.

Version: \$Id: //depot/code/org.antlr/release/antlr-2.7.1/doc/trees.html#1 \$



ANTLR

jGuru

Grammar Inheritance

Object-oriented programming languages such as C++ and Java allow you to define a new object as it differs from an existing object, which provides a number of benefits. "Programming by difference" saves development/testing time and future changes to the *base* or *superclass* are automatically propagated to the *derived* or *subclass*.

Introduction and motivation

Allowing the ANTLR programmer to define a new grammar as it differs from an existing grammar provides significant benefits. Development time goes down because the programmer only has to specify the rules that are different or that need to be added. Further, when the base grammar changes, all derived grammars will automatically reflect the change. Grammar inheritance is also an interesting way to change the behavior of an existing grammar. A rule or set of rules can be respecified with the same structure, but with different actions.

The most obvious use of grammar inheritance involves describing multiple dialects of the same language. Previous solutions would require multiple grammar versions or a single grammar that recognized all dialects at once (using semantics to constrain the input to a single dialect). With grammar inheritance, one could write a base grammar for the common components and then have a derived grammar for each dialect. Code sharing would occur at the grammar and output parser class level.

Consider a simple subset of English:

```
class PrimarySchoolEnglish;

sentence
    :   subject predicate
    ;
subject
    :   NOUN
    ;
predicate
    :   VERB
    ;
```

This grammar recognizes sentences like: `Dilbert speaks.`

To extend this grammar to include sentences manageable by most American college students, we might add direct objects to the definition of a sentence. Rather than copying and modifying the `PrimarySchoolEnglish` grammar, we can simply extend it:

```
class AmericanCollegeEnglish extends
    PrimarySchoolEnglish;

sentence
    :   subject predicate object
```

```

    ;
    object
    :   PREPOSITION ARTICLE NOUN
    ;

```

This grammar describes sentences such as `Dilbert speaks to a dog`. While this looks trivial to implement (just add the appropriate `extends` clause in Java to the output parser class), it involves grammar analysis to preserve grammatical correctness. For example, to generate correct code, ANTLR needs to pull in the base grammar and modify it according to the overridden rules. To see this, consider the following grammar for a simple language:

```

class Simple;

stat:   expr ASSIGN expr
      |   SEMICOLON
      ;

expr:   ID
      ;

```

Clearly, the `ID` token is the lookahead set that predicts the recognition of the first alternative of `stat`. Now, examine a derived dialect of `Simple`:

```

class Derived extends Simple;

expr:   ID
      |   INT
      ;

```

In this case, `{ ID, INT }` predicts the first alternative of `stat`. Unfortunately, a derived grammar affects the recognition of rules inherited from the base grammar! ANTLR must not only override `expr` in `Derived`, but it must override `stat`.

Determining which rules in the base grammar are affected is not easy, so our implementation simply makes a copy of the base grammar and generates a whole new parser with the appropriate modifications. From the programmer's perspective, code/grammar sharing would have occurred, however, from an implementation perspective a copy of the base grammar would be made.

Functionality

Grammar `Derived` inherits from Grammar `Base` all of the rules, options, and actions of `Base` including formal/actual rule parameters and rule actions. `Derived` may override any option or rule and specify new options, rules, and member action. The subgrammar does not inherit actions outside of classes or file options. Consider rule `Base` defined as:

```

class Base extends Parser;
options {
    k = 2;
}

```

```

{
  int count = 0;
}
a : A B {an-action}
  | A C
  ;
c : C
  ;

```

A new grammar may be derived as follows:

```

class Derived extends Base;
options {
  k = 3;          // need more lookahead; override
  buildAST=true; // add an option
}
{
  int size = 0; // override; no 'count' def here
}
a : A B {an-action}
  | A C {an-extra-action}
  | Z          // add an alt to rule a
  ;
b : a
  | A B D      // requires LL(3)
  ;

```

ANTLR will actually interpret the subgrammar as if you had typed:

```

class Derived extends Parser;
options {
  k=3;
  buildAST=true;
}
{
  int size = 0; // override Base action
}
a : A B {an-action}
  | A C {an-extra-action}
  | Z          // add an alt to rule a
  ;

b : a
  | A B D      // requires LL(3)
  ;

// inherited from grammar Base
c : C
  ;

```

Rules may be overridden to change their signatures such as their parameters or return

types:

```
class Base extends Parser;
a[int x] returns [int y]
    : A
    ;
```

```
class Derived extends Base;
a[float z]
    : A
    ;
```

ANTLR will generate a warning, however:

```
warning: rule Derived.a has different signature than Base.a
```

Because of this ability, the subgrammars do not actually inherit, in the Java-sense, from the supergrammar. Different signatures on the generated methods would prevent the parser from compiling.

Where Are Those Supergrammars?

The set of potential "supergrammars" available to some grammar P includes any other grammar in the same file as P and any listed on the ANTLR command line with

```
-glib f1.g;f2.g
```

where the files must include path names if they are located in another directory.

How is supergrammar P found? The grammars defined in the supergrammar list are read in and an inheritance hierarchy is constructed; any repeated grammar definition in this is ignored. The grammars in the normally specified grammar file are also included in the hierarchy. Incomplete hierarchies results in an error message from ANTLR. Grammars in the same file as P are given precedence to those obtained from other files.

The type of grammar (Lexer,Parser,TreeParser) is determined by the type of the highest grammar in the inheritance chain.

Error Messages

ANTLR generates a file called `expandedT.g`, given a grammar input file (not the `-glib` files) called `T.g`. All error messages are relative to this as you really want to see the whole grammar when dealing with ambiguities etc... In the future, we may have a better solution.

Version: \$Id: //depot/code/org.antlr/release/antlr-2.7.1/doc/inheritance.html#1 \$



ANTLR

jGuru

Options

File, Grammar, and Rule Options

Rather than have the programmer specify a bunch of command-line arguments to the parser generator, an options section within the grammar itself serves this purpose. This solution is preferable because it associates the required options with the grammar rather than ANTLR invocation. The section is preceded by the `options` keyword and contains a series of option/value assignments surrounded by curly braces such as:

```
options {
    k = 2;
    tokenVocabulary = IDL;
    defaultErrorHandler = false;
}
```

The options section for an entire (.g) file, if specified, immediately follows the (optional) file header:

```
header { package X; }
options { language="FOO"; }
```

The options section for a grammar, if specified, must immediately follow the ';' of the class specifier:

```
class MyParser extends Parser;
options { k=2; }
```

The options section for a rule, if specified, must immediately follow the rule name:

```
myrule[args] returns [retval]
    options { defaultErrorHandler=false; }
    : // body of rule...
    ;
```

The option names are not keywords in ANTLR, but rather are entries in a symbol table examined by ANTLR. The scope of option names is limited to the `options` section; identifiers within your grammar may overlap with these symbols.

The only ANTLR options not specified in the `options` section are things that do not vary with the grammar, but rather than invocation of ANTLR itself. The best example is debugging information. Typically, the programmer will want a makefile to change an ANTLR flag indicating a debug or release build.

Options supported in ANTLR

Key for the type column: F=file, G=grammar, R=rule, L=lexer, S=subrule, C=C++ only.

Symbol	Type	Description
language	F	Set the generated language
k	G	Set the lookahead depth

importVocab	G	Initial grammar vocabulary
exportVocab	G	Vocabulary exported from grammar
testLiterals	LG,LR	Generate literal-testing code
defaultErrorHandler	G,R	Control default exception-handling
codeGenMakeSwitchThreshold	G	Control code generation
codeGenBitsetTestThreshold	G	Control code generation
buildAST	G	Set automatic AST construction in Parser (transform mode in Tree-Parser)
analyzerDebug	G	Spit out lots of debugging information while performing grammar analysis.
codeGenDebug	G	Spit out lots of debugging information while doing code generation.
ASTLabelType	G	Specify the type of all user-defined labels, overrides default of AST.
charVocabulary	LG	Set the lexer character vocabulary
interactive	G	Both the lexer and the parser have an interactive option, which defaults to "false". See the parser speed section above.
caseSensitive	LG	Case is ignored when comparing against character and string literals in the lexer. The case of the input stream is maintained when stored in the token objects.

<code>ignore</code>	LR	Specify a lexer rule to use as whitespace between lexical rule atomic elements (chars, strings, and rule references). The grammar analysis and, hence, the lookahead sets are aware of the whitespace references. This is a lexer rule option.
paraphrase	LR	An easy way to specify a string to use in place of the token name during error processing.
<code>caseSensitiveLiterals</code>	LG	Case is ignored when comparing tokens against the literals table.
<code>classHeaderSuffix</code>	G	Append a string to the enclosing class definition. In Java, this amounts to a comma-separated list of interfaces that your lexer, parser, or tree walker must implement.
<code>mangleLiteralPrefix</code>	F	Sets the prefix for the token type definitions of literals rather than using the default of "TOKEN_".
warnWhenFollowAmbig	S	Warnings will be printed when the lookahead set of what follows a subrule containing an empty alternative conflicts with a subrule alternative or when the implicit exit branch of a closure loop conflicts with an alternative. The default is true.

generateAmbigWarnings	S	<p>When true, no ambiguity/nondeterminism warning is generated for the decision associated with the subrule. Use this very carefully--you may change the subrule and miss an ambiguity because of the option. Make very sure that the ambiguity you mask is handled properly by ANTLR.</p> <p>ANTLR-generated parsers resolve ambiguous decisions by consuming input as soon as possible (or by choosing the alternative listed first).</p> <p>See the Java and HTML grammars for proper use of this option. A comment should be supplied for each use indicating why it is ok to shut off the warning.</p>
filter	LG	<p>When true, the lexer ignores any input not exactly matching one of the nonprotected lexer rules. When set to a rule name, the filter option using the rule to parse input characters between valid tokens or those tokens of interest.</p>
namespace	FGC	<p>When set, all the C++ code generated is wrapped in the namespace mentioned here.</p>
namespaceStd	FGC	<p>When set, the ANTLR_USE_NAMESPACE(std) macros in the generated C++ code are replaced by this value. This is a cosmetic option that only makes the code more readable. It does not replace this macro in the support C++ files. Note: use this option directly after setting the language to C++.</p>
namespaceAntlr	FGC	<p>When set, the ANTLR_USE_NAMESPACE(antlr) macros in the generated C++ code are replaced by this value. This is a cosmetic option that only makes the code more readable. It does not replace this macro in the support C++ files. Note: use this option directly after setting the language to C++.</p>

genHashLines	FGC	Boolean toggle, when set to 'true' #line <linenumber> "filename" lines are inserted in the generated code so compiler errors/warnings refer the .g files.
--------------	-----	---

language: Setting the generated language

ANTLR supports multiple, installable code generators. Any code-generator conforming to the ANTLR specification may be invoked via the language option. The default language is "Java", but "Cpp", and "Sather" are also supported. The language option is specified at the file-level, for example:

```
header { package zparse; }
options { language="Java"; }
... classes follow ...
```

k: Setting the lookahead depth

You may set the lookahead depth for any grammar (parser, lexer, or tree-walker), by using the k= option:

```
class MyLexer extends Lexer;
options { k=3; }
...
```

Setting the lookahead depth changes the maximum number of tokens that will be examined to select alternative productions, and test for exit conditions of the EBNF constructs (...)?, (...)+, and (...)*. The lookahead analysis is *linear approximate* (as opposed to full LL(k)). This is a bit involved to explain in detail, but consider this example with k=2:

```
r : ( A B | B A )
   | A A
   ;
```

Full LL(k) analysis would resolve the ambiguity and produce a lookahead test for the first alternate like:

```
if ( (LA(1)==A && LA(2)==B) || (LA(1)==B && LA(2)==A) )
```

However, linear approximate analysis would logically OR the lookahead sets at each depth, resulting in a test like:

```
if ( (LA(1)==A || LA(1)==B) && (LA(2)==A || LA(2)==B) )
```

Which is ambiguous with the second alternate for {A,A}. Because of this, setting the lookahead depth very high tends to yield diminishing returns in most cases, because the lookahead sets at large depths will include almost everything.

importVocab: Initial Grammar Vocabulary

[See the documentation on [vocabularies](#) for more information]

To specify an initial vocabulary (tokens, literals, and token types), use the importVocab grammar option.

```
class MyParser extends Parser;
options {
    importVocab=V;
}
```

ANTLR will look for VTokenTypes.txt in the current directory and preload the token manager for MyParser with the enclosed information.

This option is useful, for example, if you create an external lexer and want to connect it to an ANTLR parser. Conversely, you may create an external parser and wish to use the token set with an ANTLR lexer. Finally, you may find it more convenient to place your grammars in separate files, especially if you have multiple tree-walkers that do not add any literals to the token set.

The vocabulary file has an identifier on the first line that names the token vocabulary that is followed by lines of the form ID=value or "literal"=value. For example:

```
ANTLR // vocabulary name
"header"=3
ACTION=4
COLON=5
SEMI=6
...
```

A file of this form is automatically generated by ANTLR for each grammar.

Note: you must take care to run ANTLR on the vocabulary-generating grammar files **before** you run ANTLR on the vocabulary-consuming grammar files.

exportVocab: Naming Export Vocabulary

[See the documentation on [vocabularies](#) for more information]

The vocabulary of a grammar is the union of the set of tokens provided by an importVocab option and the set of tokens and literals defined in the grammar. ANTLR exports a vocabulary for each grammar whose default name is the same as the grammar. So, the following grammar yields a vocabulary called P:

```
class P extends Parser;
a : A;
```

ANTLR generates files PTokenTypes.txt and PTokenTypes.java.

You can specify the name of the exported vocabulary with the exportVocab option. The following grammar generates a vocabulary called V not P.

```
class P extends Parser;
options {
    exportVocab=V;
}
a : A;
```

All grammars in the same file with the same vocabulary name contribute to the same vocabulary (and resulting files). If the grammars were in separate files, on the other hand, they would all overwrite the same file. For example, the following parser and lexer

grammars both may contribute literals and tokens to the MyTokens vocabulary.

```
class MyParser extends Parser;
options {
    exportVocab=MyTokens;
}
...
```

```
class MyLexer extends Lexer;
options {
    exportVocab=MyTokens;
}
...
```

testLiterals: Generate literal-testing code

By default, ANTLR will generate code in all lexers to test each token against the literals table (the table generated for literal strings), and change the token type if it matches the table. However, you may suppress this code generation in the lexer by using a grammar option:

```
class L extends Lexer;
options { testLiterals=false; }
...
```

If you turn this option off for a lexer, you may re-enable it for specific rules. This is useful, for example, if all literals are keywords, which are special cases of ID:

```
ID
options { testLiterals=true; }
    : LETTER (LETTER | DIGIT)*
    ;
```

If you want to test only a portion of a token's text for a match in the literals table, explicitly test the substring within an action using method:

```
public int testLiteralsTable(String text, int ttype) {...}
```

For example, you might want to test the literals table for just the tag word in an HTML word.

defaultErrorHandler: Controlling default exception-handling

By default, ANTLR will generate default exception handling code for a parser or tree-parser rule. The generated code will catch any parser exceptions, synchronize to the follow set of the rule, and return. This is simple and often useful error-handling scheme, but it is not very sophisticated. Eventually, you will want to install your own exception handlers. ANTLR will automatically turn off generation of default exception handling for rule where an exception handler is specified. You may also explicitly control generation of default exception handling on a per-grammar or per-rule basis. For example, this will turn off default error-handling for the entire grammar, but turn it back on for rule "r":

```
class P extends Parser;
options {defaultErrorHandler=false;}
```



```
r
options {defaultErrorHandler=true;}
: A B C;
```

For more information on exception handling in the lexer, go [here](#).

codeGenMakeSwitchThreshold: controlling code generation

ANTLR will optimize lookahead tests by generating a switch statement instead of a series of if/else tests for rules containing a sufficiently large number of alternates whose lookahead is strictly LL(1). The option `codeGenMakeSwitchThreshold` controls this test. You may want to change this to control optimization of the parser. You may also want to disable it entirely for debugging purposes, by setting it to a large number:

```
class P extends Parser;
options { codeGenMakeSwitchThreshold=999; }
...
```

codeGenBitsetTestThreshold: controlling code generation

ANTLR will optimize lookahead tests by generating a bitset test instead of an if statement, for very complex lookahead sets. The option `codeGenBitsetTestThreshold` controls this test. You may want to change this to control optimization of the parser:

```
class P extends Parser;
// make bitset if test involves five or more terms
options { codeGenBitsetTestThreshold=5; }
...
```

You may also want to disable it entirely for debugging purposes, by setting it to a large number:

```
class P extends Parser;
options { codeGenBitsetTestThreshold=999; }
...
```

buildAST: Automatic AST construction

In a Parser, you can tell ANTLR to generate code to construct ASTs corresponding to the structure of the recognized syntax. The option, if set to true, will cause ANTLR to generate AST-building code. With this option set, you can then use all of the [AST-building syntax and support methods](#).

In a Tree-Parser, this option turns on "transform mode", which means an output AST will be generated that is a transformation of the input AST. In a tree-walker, the default action of `buildAST` is to generate a copy of the portion of the input AST that is walked. Tree-transformation is almost identical to building an AST in a Parser, except that the input is an AST, not a stream of tokens.

ASTLabelType: Setting label type

When you must define your own AST node type, your actions within the grammar will require lots of downcasting from AST (the default type of any user-defined label) to your

tree node type; e.g.,

```
decl : d:ID {MyAST t=(MyAST)#d;}
      ;
```

This makes your code a pain to type in and hard to read. To avoid this, use the grammar option `ASTLabelType` to have ANTLR automatically do casts and define labels of the appropriate type.

```
class ExprParser extends Parser;
```

```
options {
    buildAST=true;
    ASTLabelType = "MyAST";
}
```

```
expr : a:term ;
```

The type of #a within an action is `MyAST` not `AST`.

charVocabulary: Setting the lexer character vocabulary

ANTLR processes Unicode. Because of this this, ANTLR cannot make any assumptions about the character set in use, else it would wind up generating huge lexers. Instead ANTLR assumes that the character literals, string literals, and character ranges used in the lexer constitute the entire character set of interest. For example, in this lexer:

```
class L extends Lexer;
A : 'a';
B : 'b';
DIGIT : '0' .. '9';
```

The implied character set is { 'a', 'b', '0', '1', '2', '3', '4', '5', '6', '7', '8', '9' }. This can produce unexpected results if you assume that the normal ASCII character set is always used. For example, in:

```
class L extends Lexer;
A : 'a';
B : 'b';
DIGIT : '0' .. '9';
STRING: '"' (~'"")* '"';
```

The lexer rule `STRING` will only match strings containing 'a', 'b' and the digits, which is usually not what you want. To control the character set used by the lexer, use the `charVocabulary` option. This example will use a general eight-bit character set.

```
class L extends Lexer;
options { charVocabulary = '\3'..'377'; }
...
```

This example uses the ASCII character set in conjunction with some values from the extended Unicode character set:

```
class L extends Lexer;
options {
```

```

        charVocabulary = '\3'..'377' | '\u1000'..'1ffff';
    }
    ...

```

warnWhenFollowAmbig

[**Warning:** you should know what you are doing before you use this option. I deliberately made it a pain to shut warnings off (rather than a single character operator) so you would not just start turning off all the warnings. I thought for long time before implementing this exact mechanism. I recommend a comment in front of any use of this option that explains why it is ok to hush the warning.]

This subrule option is true by default and controls the generation of nondeterminism (ambiguity) warnings when comparing the FOLLOW lookahead sets for any subrule with an empty alternative and any closure subrule such as `(..)+` and `(...)*`. For example, the following simple rule has a nondeterministic subrule, which arises from a language ambiguity that you could attach an ELSE clause to the most recent IF **or** to an outer IF because the construct can nest.

```

stat      :      "if" expr "then" stat ("else" stat)?
          |      ID ASSIGN expr SEMI
          ;

```

Because the language is ambiguous, the context-free grammar must be ambiguous and the resulting parser nondeterministic (in theory). However, being the practical language folks that we are, we all know you can trivially solve this problem by having ANTLR resolve conflicts by consuming input as soon as possible; I have yet to see a case where this was the wrong thing to do, by the way. This option, when set to false, merely informs ANTLR that it has made the correct assumption and can shut off an ambiguity related to this subrule and an empty alternative or exit path. Here is a version of the rule that does not yield a warning message:

```

stat      :      "if" expr "then" stat
          (
            // standard if-then-else ambig
            options {
                warnWhenFollowAmbig=false;
            }
            :      "else" stat
          )?
          |      ID ASSIGN expr SEMI
          ;

```

One important note: This option does not affect non-empty alternatives. For example, you will still get a warning for the following subrule between alts 1 and 3 (upon lookahead A):

```

(
    options {
        warnWhenFollowAmbig=false;
    }
:   A
|   B

```

```
|      A
)
```

Further, this option is insensitive to lookahead. Only completely empty alternatives count as candidate alternatives for hushing warnings. So, at $k=2$, just because ANTLR can see past alternatives with single tokens, you still can get warnings.

Command Line Options

<code>-o outputDir</code>	specify output directory where all output generated.
<code>-glib supergrammarFile</code>	Specify a file with a supergrammar for the generated file.
<code>-debug</code>	launch the ParseView debugger upon parser invocation. Unless you have downloaded and unzipped the debugger over the top of the standard ANTLR distribution, the code emanating from ANTLR with this option will not compile (likewise for Swing).
<code>-html</code>	generate an html file from your grammar without actions and so on. This is only a prototype , but seems to do something useful. It only works for parsers, not lexers or tree parsers.
<code>-diagnostic</code>	generate a text file from your grammar with a lot of debugging info.
<code>-trace</code>	have all rules call <code>traceIn/traceOut</code> .
<code>-traceParser</code>	have parser rules call <code>traceIn/traceOut</code> .
<code>-traceLexer</code>	have lexer rules call <code>traceIn/traceOut</code> .
<code>-traceTreeWalker</code>	have tree walker rules call <code>traceIn/traceOut</code> .

Version: \$Id: //depot/code/org.antlr/test/antlr-2.7.0a11/doc/options.html#3 \$