

1. 1. Document Snapshot API	2
2. 2. Parsing API	3
3. 3. Syntax Highlighting	5
4. 4. Code Completion	5
5. 5. Refactoring	5
6. Home	5
6.1 ANTLR 4 Documentation	7
6.1.1 Getting Started with ANTLR v4	7
6.1.2 Grammar Lexicon	10
6.1.3 Grammar Structure	13
6.1.4 Parser Rules	17
6.1.5 Left-recursive rules	26
6.1.6 Actions and Attributes	27
6.1.7 Lexer Rules	31
6.1.8 Wildcard Operator and Nongreedy Subrules	36
6.1.9 Parse Tree Listeners	39
6.1.10 Parse Tree Matching and XPath	40
6.1.11 Semantic Predicates	43
6.1.12 Options	46
6.1.13 ANTLR Tool Command Line Options	47
6.1.14 Runtime Libraries and Code Generation Targets	51
6.1.14.1 C# Target	51
6.1.14.2 C++ Target	53
6.1.14.3 JavaScript Target	53
6.1.14.4 Java Target	55
6.1.14.5 Python Target	56
6.1.15 Parser and lexer interpreters	58
6.1.16 Integrating ANTLR into Development Systems	59
6.1.16.1 Java IDE Integration	60
6.1.16.2 C# IDE Integration	65
6.1.16.3 C++ IDE Integration	65
6.2 ANTLR v4 FAQ	65
6.2.1 FAQ - Getting Started	66
6.2.1.1 How to I install and run a simple grammar?	66
6.2.1.2 Why does my parser test program hang?	66
6.2.2 FAQ - Installation	66
6.2.2.1 Why can't ANTLR (grun) find my lexer or parser?	67
6.2.2.2 Why can't I run the ANTLR tool?	67
6.2.2.3 Why doesn't my parser compile?	67
6.2.3 FAQ - General	67
6.2.3.1 Why do we need ANTLR v4?	68
6.2.3.2 What is the difference between ANTLR 3 and 4?	69
6.2.3.3 Why is my expression parser slow?	69
6.2.4 FAQ - Grammar syntax	70
6.2.5 FAQ - Lexical analysis	70
6.2.5.1 How can I parse non-ASCII text and use characters in token rules?	70
6.2.5.2 How do I replace escape characters in string tokens?	70
6.2.5.3 Why are my keywords treated as identifiers?	71
6.2.5.4 Why are there no whitespace tokens in the token stream?	71
6.2.6 FAQ - Parse Trees	71
6.2.6.1 What if I need ASTs not parse trees for a compiler, for example?	72
6.2.6.2 When do I use listener/visitor vs XPath vs Tree pattern matching?	72
6.2.7 FAQ - Translation	72
6.2.7.1 ASTs vs parse trees	72
6.2.7.2 Decoupling input walking from output generation	72
6.2.8 FAQ - Actions and semantic predicates	73
6.2.8.1 How do I test if an optional rule was matched?	73
6.2.9 FAQ - Error handling	73
6.2.9.1 How do I perform semantic checking with ANTLR?	73
6.3 Articles and Resources	73
6.4 Stale	74
6.4.1 ANTLRWorks 2	74
6.4.1.1 1. Overview	74
6.4.1.2 2. Grammar Editing Features	75
6.4.1.3 3. StringTemplate 4 Editing Features	78
6.4.1.4 4. Lexer Debugging	78
6.4.1.5 5. Parser Debugging	80
6.4.2 Building ANTLR 4 with Maven	80
7. TODO list	82

1. Document Snapshot API

The Document Snapshot API is primarily a consolidation and extension of several low-level features which previously existed in the NetBeans source code but were not usable as a whole to third-party extensions. Most of the central features of the API draw from the NetBeans [Editor Library 2](#) and [Parsing API](#).

The core interfaces in the Document Snapshot API are `VersionedDocument` and `DocumentSnapshot`, which are most similar to the `Source` and `Snapshot` classes (respectively) in the Parsing API. Here are some particular similarities.

Similarities between `VersionedDocument` and `Source`

- Each can represent a file on disk or document currently opened for editing.
- Each can provide a "snapshot" of the document contents.

Similarities between `DocumentSnapshot` and `Snapshot`

- Each represents an immutable view of document contents at some point in time.

At this point, the APIs quickly diverge. The NetBeans Parsing API extends on the above to provide specific parsing features such as access to the [TokenHierarchy](#) and the ability to represent [Embeddings](#). The Document Snapshot API extends on the above to address synchronization issues which arise when asynchronous operations are occurring while documents are being edited. Since the Parsing API is already well documented, from here on I'll focus on the specifics of the Document Snapshot API.

Document versioning

Document versions are a way of expressing the relative points in time for which multiple snapshots apply. If the user types text in a document while an asynchronous operation is processing a `DocumentSnapshot`, the operation's snapshot represents an earlier version of the text, and the text after the user's edits is the most recent or *current* version of the document. The `VersionedDocument.getCurrentSnapshot()` method provides a `DocumentSnapshot` for the current version of the document.

The `DocumentVersion` interface provides specific information about document versions. The `getVersionNumber()` method returns an integer, where higher values represent later versions of a document. The `getNext()` method returns the following version (or `null` if called on the most recent version), and the `getChanges()` method returns a collection of changes which were applied to the current version to get the next version.

Positions and regions within a document

An *offset* is a single location represented by an integer. The offset *n* represents the location before the *n*-th character (0-based). In the Document Snapshot API, offsets have the primitive type `int`.

A *position* is an offset within the context of a specific document. In the Document Snapshot API, the `SnapshotPosition` class is used to represent an offset within a specific `DocumentSnapshot`. The `getSnapshot()` and `getOffset()` methods return the snapshot and offset for the position.

A *region* is a pair of offsets. Regions are expressed either by their start offset and length or by their start and end (exclusive) offsets. In the Document Snapshot API, a region is represented by the `OffsetRegion` class.

A *position region* is a pair of positions, also described as a region within the context of a specific document. In the Document Snapshot API, a position region has the type `SnapshotPositionRegion`. The `getSnapshot()` and `getRegion()` methods return the snapshot and `OffsetRegion` for the position region. The methods `getStart()` and `getEnd()` return the start and end (exclusive) positions of the region.

Tracking positions and regions between document versions

The Document Snapshot API really shows its strength in the ability to create a position or position region in one snapshot, and then map the value to a new snapshot. The snapshot and mapping methods are thread-safe, so they may be used by asynchronous operations or called on the foreground thread with the current document version to consistently mark locations found during an asynchronous operation (eg. marking errors from a background parse).

The `TrackingPosition` and `TrackingPositionRegion` interfaces represent positions and position regions which can be translated from one document version to another. The tracking operations can be expressed with a *bias* for specific handling of document changes at the position (or endpoints for a position region).

The `TrackingPosition.Bias` enumeration provides `Forward` and `Backward`, which results in the position moving towards the end or beginning of a document (respectively) in response to text insertions at the position.

The `TrackingPositionRegion.Bias` enumeration provides values for each combination of `TrackingPosition.Bias` for the start and end points of the position region. The values `Forward` and `Backward` map both the start and end positions towards the end or beginning of a document (respectively) in response to text insertions at the start or end positions of the region. The value `Exclusive` maps the start position `Forward`

ward and the end position Backward. The value Inclusive maps the start position Backward and the end position Forward.

To allow substantial optimization in the implementation, tracking positions and position regions are also expressed with a tracking *fidelity* (with the `TrackingFidelity` enumeration). By default the fidelity is `Forward`, which means operations only support translating a position or position region from an older document version to a newer document version. This value is appropriate for almost all mapping scenarios encountered during asynchronous operations in the IDE.

Similar concepts in other APIs

The above concepts are certainly not new. Advanced processing for new features in IDEs demands asynchronous operations, and expressing locations and versions of documents has become necessary but not always clean.

Swing

- The `Document` interface expresses a document, but operations on it are not thread-safe and it doesn't support the concepts of snapshots and versions.
- The `Position` interface expresses a position, but without snapshots the object is not thread safe and the implementation eagerly updates the offset as the document changes, up until the garbage collector collects the position.
- The `Position.Bias` enumeration expresses a position bias, but there are no methods provided by Swing interfaces for actually creating a position with a specified bias.

NetBeans

- The `PositionRegion` class holds a pair of `Position` objects, but suffers from the same drawbacks of `Position` mentioned above.
- The `BaseDocument` interface provides a `createPosition` method which takes a specified bias as an argument. The created positions suffer from the same drawbacks listed above.
- The NetBeans implementation of `BaseDocument` increments an internal version number when the document is edited, but the number is not publicly exposed so code operating on documents must implement their own versioning in response to `DocumentListener` events.

Visual Studio SDK

- The `ITextSnapshot` and `ITextVersion` interfaces provide features similar to `DocumentSnapshot` and `DocumentVersion`.
- The `SnapshotPoint` and `SnapshotSpan` structures provide features similar to `SnapshotPosition` and `SnapshotPositionRegion`.

Lines and columns

Methods for mapping between document offsets and line/column numbers are provided by the `DocumentSnapshot` interface and `SnapshotPosition` class. Line and column numbers are 0-based, and column numbers treat all characters (including "hard" tab characters) as taking up exactly one column. The `DocumentSnapshotLine` interface provides access to information about a line including the line number, text, and position region. An instance of this interface is returned by `DocumentSnapshot.findLineFromLineNumber`, `DocumentSnapshot.findLineFromOffset`, and `SnapshotPosition.getContainingLine`.

VersionedDocumentUtilities

The `VersionedDocumentUtilities` class provides overloaded `getVersionedDocument` methods which return a `VersionedDocument` from a `Document` or `FileObject`.

Implementation details

TODO

LineTextCache

TODO

Lazily tracked positions and regions

TODO

2. Parsing API

ANTLRWorks 2 uses a custom API for parsing due to a number of major issues which severely undermine the performance of the [NetBeans Parsing API](#).

Evaluating the NetBeans Parsing API

Advantages

- Cleanly integrated with the [Lexer API](#) which allows a shared lexer to be used for both the syntax highlighting and parsing features.
- Provides full-text indexing support backed by [Apache Lucene](#).
- Uses a common representation for opened and unopened project files ([Source](#) and [Snapshot](#) objects provide immutable views of either [Document](#) or [FileObject](#) documents).
- Encourages asynchronous parsing operations for parsing both opened documents (indexing and features like semantic highlighting) and unopened documents (project indexing).
- Cleanly and easily supports custom source embeddings which can be used for anything from special parsing for documentation comments to mixed HTML/PHP source documents. This is supported by both the Lexer and Parsing APIs, so it works for syntax highlighting, indexing, and more.

Disadvantages

- Only supports one [Parser](#) for a mime type. Different IDE operations require different types/amounts of information as input, and the restriction to a single [Parser.Result](#) restricts the ability to meet the performance potential of "quick" tasks when "slow" tasks are also supported.
- Uses a [scheduler](#) with a severely restricted threading policy which is shared among foreground (code completion) and background (project indexing) tasks. It's also shared between languages, so the (very slow) project indexing process for Java source code prevents code completion within ANTLR grammars from working.
- Uses Lucene for built-in indexing support, which is brutally slow for standard IDE operations.

Conclusion

Due to the fundamental goal of having an IDE (like ANTLRWorks 2) operate at least as fast as the developer works (types), it is immediately clear that the task scheduler and indexing support provided by the Parsing API cannot be used. While it would be theoretically possible to extend the [Task](#) class to declare data dependencies which are handled/dispatched within a special [Parser](#) implementation, such a technique has already diverged from the original implementation in the Parsing API that little is gained from reusing those base classes/interfaces.

Requirements for the ANTLRWorks Parsing API

Snapshots

- The API should operate on the [1. Document Snapshot API](#) to allow mapping between the parsed snapshot and the current version of the document.

Concurrency and task scheduling

- The API should support multithreaded parsing to take advantage of modern multi-core processors.
- The API should be able to distinguish between low- and high-priority tasks in addition to basic support for asynchronous operations. High-priority tasks might include operations like semantic highlighting and code completion, where low-priority tasks might include background indexing and/or compilation.
- The API should support adjustable (and potentially adaptive) reparse delays to offer "instant" UI updates for features like code completion and semantic highlighting without causing excessive overhead or irritating UI updates for tasks like background compilation and error reporting.

Declarative data dependencies

- The API should allow different parsing strategies to be used by different tasks/features to allow optimization of low-latency features like code completion without restricting the ability to provide higher-latency features like refactoring or Find Usages.

Stale data

- Tasks should be able to request "stale" data rather than a reparse if results from a previous `DocumentSnapshot` are available. Several low-latency features in ANTLRWorks 2 rely on [dynamic anchors](#), which use a blend of stale and updated data to provide near-instantaneous responses even in extremely large source files.

The ANTLRWorks 2 Parsing API

TODO

Implementation details

TODO

3. Syntax Highlighting

4. Code Completion

5. Refactoring

Home

Welcome

This ANTLR 4 wiki complements and supports the [ANTLR website](#). Inside the wiki you will find documentation, tutorials and a FAQ.

We hope that you find what you need to learn more about ANTLR 4 on this wiki. If this is not the case, you may register and help.

To get started with ANTLR, jump to [Getting Started with ANTLR v4](#). If you run into trouble, ANTLR has a [mailing list](#) full of helpful ANTLR fans and check out [Stack overflow ANTLR4 tag](#). You should also consider reading the following books:



See [information on ANTLR releases](#).

Error rendering macro 'recently-updated-dashboard' : null

Blog Posts

[Matching parse tree patterns, paths](#) created by Terence Parr [Administrator]
Terence Parr Sep 01, 2013

[The real story on null vs empty](#) created by Terence Parr [Administrator]
Terence Parr Dec 27, 2012

[Tree rewriting in ANTLR v4](#) created by Terence Parr [Administrator]
Terence Parr Dec 08, 2012

[ANTLR and Shroedinger's Tokens](#) created by Terence Parr [Administrator]

Wiki Contents

ANTLR 4 Documentation

- [Getting Started with ANTLR v4](#)
- [Grammar Lexicon](#)
- [Grammar Structure](#)
- [Parser Rules](#)
- [Left-recursive rules](#)
- [Actions and Attributes](#)
- [Lexer Rules](#)
- [Wildcard Operator and Nongreedy Subrules](#)
- [Parse Tree Listeners](#)
- [Parse Tree Matching and XPath](#)
- [Semantic Predicates](#)
- [Options](#)
- [ANTLR Tool Command Line Options](#)
- [Runtime Libraries and Code Generation Targets](#)
 - [C# Target](#)
 - [C++ Target](#)
 - [JavaScript Target](#)
 - [Java Target](#)
 - [Python Target](#)
- [Parser and lexer interpreters](#)
- [Integrating ANTLR into Development Systems](#)
 - [Java IDE Integration](#)
 - [C# IDE Integration](#)
 - [C++ IDE Integration](#)

ANTLR v4 FAQ

- [FAQ - Getting Started](#)
 - [How to I install and run a simple grammar?](#)
 - [Why does my parser test program hang?](#)
- [FAQ - Installation](#)
 - [Why can't ANTLR \(grun\) find my lexer or parser?](#)
 - [Why can't I run the ANTLR tool?](#)
 - [Why doesn't my parser compile?](#)
- [FAQ - General](#)
 - [Why do we need ANTLR v4?](#)
 - [What is the difference between ANTLR 3 and 4?](#)
 - [Why is my expression parser slow?](#)
- [FAQ - Grammar syntax](#)
- [FAQ - Lexical analysis](#)
 - [How can I parse non-ASCII text and use characters in token rules?](#)
 - [How do I replace escape characters in string tokens?](#)
 - [Why are my keywords treated as identifiers?](#)
 - [Why are there no whitespace tokens in the token stream?](#)
- [FAQ - Parse Trees](#)
 - [What if I need ASTs not parse trees for a compiler, for example?](#)
 - [When do I use listener/visitor vs XPath vs Tree pattern matching?](#)
- [FAQ - Translation](#)
 - [ASTs vs parse trees](#)
 - [Decoupling input walking from output generation](#)
- [FAQ - Actions and semantic predicates](#)

- How do I test if an optional rule was matched?
- FAQ - Error handling
 - How do I perform semantic checking with ANTLR?

Articles and Resources

Stale

- ANTLRWorks 2
 - 1. Overview
 - 2. Grammar Editing Features
 - 3. StringTemplate 4 Editing Features
 - 4. Lexer Debugging
 - 5. Parser Debugging
- Building ANTLR 4 with Maven

ANTLR 4 Documentation

Please check [Frequently asked questions \(FAQ\)](#) before asking questions. The [main wiki page](#) is also useful.

The latest release is ANTLR 4.4 as of July 16, 2014 (4.5 release candidate 2 available for testing)

Copyright © 2012, The Pragmatic Bookshelf. Pragmatic Bookshelf grants a nonexclusive, irrevocable, royalty-free, worldwide license to reproduce, distribute, prepare derivative works, and otherwise use this contribution as part of the ANTLR project and associated documentation.

This text was copied with permission from the [The Definitive ANTLR 4 Reference](#), though it is being morphed over time as the tool changes. Links in the documentation refer to various sections of the book but have been redirected to the general book page on the publisher's site. There are two excerpts on the publisher's website that might be useful to you without having to purchase the book: [Let's get Meta](#) and [Building a Translator with a Listener](#). You should also consider reading the following books (the vid describes the reference book):



This documentation is a reference and summarizes grammar syntax and the key semantics of ANTLR grammars. The [source code](#) for all examples in the book, not just this chapter, are free at the publisher's website. The following video is a general tour of ANTLR 4 and includes a description of how to use parse tree listeners to process Java files easily:



Sections

You can also [build ANTLR itself](#).

Getting Started with ANTLR v4

Introduction

Hi and welcome to the version 4 release of ANTLR! It's named after the fearless hero of the [Crazy Nasty-Ass Honey Badger](#) since ANTLR v4

takes whatever you give it--it just doesn't give a crap! See [Why do we need ANTLR v4?](#) and the [preface of the ANTLR v4 book](#).

Installation

ANTLR is really two things: a tool that translates your grammar to a parser/lexer in Java (or other [target language](#)) and the runtime needed by the generated parsers/lexers. Even if you are using the [ANTLR IntelliJ plug-in](#) or [ANTLRWorks](#) to run the ANTLR tool, the generated code will still need the runtime library.

The first thing you should do is probably download and install a [development tool plug-in](#). Even if you only use such tools for editing, they are great. Then, follow the instructions below to get the runtime environment available to your system to run generated parsers/lexers. In what follows, I talk about `antlr-4.5-complete.jar`, which has the tool and the runtime and any other support libraries (e.g., ANTLR v4 is written in v3).

If you are going to integrate ANTLR into your existing build system using `mvn`, `ant`, or want to get ANTLR into your IDE such as `eclipse` or `intellij`, see [Integrating ANTLR into Development Systems](#).

UNIX

0. Install Java (version 1.6 or higher)

1. Download

```
$ cd /usr/local/lib
$ curl -O http://www.antlr.org/download/antlr-4.5-complete.jar
```

Or just download in browser from website:

<http://www.antlr.org/download.html>

and put it somewhere rational like `/usr/local/lib`.

2. Add `antlr-4.5-complete.jar` to your CLASSPATH:

```
$ export CLASSPATH=".:/usr/local/lib/antlr-4.5-complete.jar:$CLASSPATH"
```

It's also a good idea to put this in your `.bash_profile` or whatever your startup script is.

3. Create aliases for the ANTLR Tool, and TestRig.

```
$ alias antlr4='java -Xmx500M -cp "/usr/local/lib/antlr-4.5-complete.jar:$CLASSPATH"
org.antlr.v4.Tool'
$ alias grun='java org.antlr.v4.runtime.misc.TestRig'
```

WINDOWS

(Thanks to Graham Wideman)

0. Install Java (version 1.6 or higher)

1. **Download** <http://www.antlr.org/download/antlr-4.5-complete.jar>
Save to your directory for 3rd party Java libraries, say `C:\Javalib`

2. Add `antlr-4.5-complete.jar` to CLASSPATH, either:

- Permanently: Using System Properties dialog > Environment variables > Create or append to CLASSPATH variable
- Temporarily, at command line:

```
SET CLASSPATH=.;C:\Javalib\antlr-4.5-complete.jar;%CLASSPATH%
```

3. Create short convenient commands for the ANTLR Tool, and TestRig, using batch files or doskey commands:

Batch files (in directory in system PATH)

antlr4.bat

```
java org.antlr.v4.Tool %*
```

grun.bat

```
java org.antlr.v4.runtime.misc.TestRig %*
```

Or, use doskey commands:

```
doskey antlr4=java org.antlr.v4.Tool $*
doskey grun =java org.antlr.v4.runtime.misc.TestRig $*
```

Testing the installation

Either launch `org.antlr.v4.Tool` directly:

```
$ java org.antlr.v4.Tool
ANTLR Parser Generator Version 4.5
-o ___ specify output directory where all output is generated
-lib ___ specify location of .tokens files
...
```

or use `-jar` option on java:

```
$ java -jar /usr/local/lib/antlr-4.5-complete.jar
ANTLR Parser Generator Version 4.5
-o ___ specify output directory where all output is generated
-lib ___ specify location of .tokens files
...
```

A First Example

In a temporary directory, put the following grammar inside file `Hello.g4`:

Hello.g4

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

Then run ANTLR the tool on it:

```
$ cd /tmp
$ antlr4 Hello.g4
$ javac Hello*.java
```

Now test it:

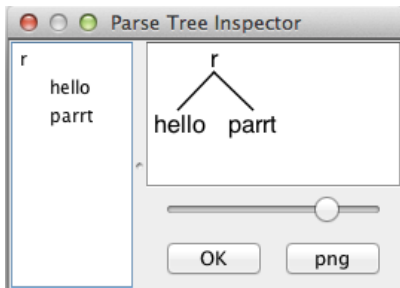
```
$ grun Hello r -tree
hello parrt
^D
(r hello parrt)
```

(That ^D means EOF on unix; it's ^Z in Windows.) The -tree option prints the parse tree in LISP notation.

It's nicer to look at parse trees visually.

```
$ grun Hello r -gui
hello parrt
^D
```

That pops up a dialog box showing that rule `r` matched keyword `hello` followed by identifier `parrt`.



Book source code

The book has lots and lots of examples that should be useful to. You can download them here for free:

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

Also, there is a collection of grammars for v4 at github:

<https://github.com/antlr/grammars-v4>

Grammar Lexicon

[ANTLR 4 Documentation Home](#)

- [Comments](#)
- [Identifiers](#)
- [Literals](#)
- [Actions](#)
- [Keywords](#)

The lexicon of ANTLR is familiar to most programmers because it follows the syntax of C and its derivatives with some extensions for grammatical descriptions.

Comments

There are single-line, multiline, and Javadoc-style comments:

```

/** This grammar is an example illustrating the three kinds
 * of comments.
 */
grammar T;
/* a multi-line
   comment
 */
/** This rule matches a declarator for my language */
decl : ID ; // match a variable name

```

The Javadoc comments are hidden from the parser and are ignored at the moment. They are intended to be used only at the start of the grammar and any rule.

Identifiers

Token names always start with a capital letter and so do lexer rules as defined by Java's `Character.isUpperCase` method. Parser rule names always start with a lowercase letter (those that fail `Character.isUpperCase`). The initial character can be followed by uppercase and lowercase letters, digits, and underscores. Here are some sample names:

```

ID, LPAREN, RIGHT_CURLY // token names/rules
expr, simpleDeclarator, d2, header_file // rule names

```

Like Java, ANTLR accepts Unicode characters in ANTLR names:

```

grammar 外;
a : '外';

```

To support Unicode parser and lexer rule names, ANTLR uses the following rule:

```

ID : a=NameStartChar NameChar*
    {
        if ( Character.isUpperCase(getText().charAt(0)) ) setType(TOKEN_REF);
        else setType(RULE_REF);
    }
;

```

`NameChar` identifies the valid identifier characters:

```

fragment
NameChar
    : NameStartChar
    | '0'..'9'
    | '_'
    | '\u00B7'
    | '\u0300'..' \u036F'
    | '\u203F'..' \u2040'
    ;

fragment
NameStartChar
    : 'A'..'Z' | 'a'..'z'
    | '\u00C0'..' \u00D6'
    | '\u00D8'..' \u00F6'
    | '\u00F8'..' \u02FF'
    | '\u0370'..' \u037D'
    | '\u037F'..' \u1FFF'
    | '\u200C'..' \u200D'
    | '\u2070'..' \u218F'
    | '\u2C00'..' \u2FEF'
    | '\u3001'..' \uD7FF'
    | '\uF900'..' \uFDCF'
    | '\uFDF0'..' \uFFFD'
    ;

```

NameStartChar is the list of characters that can start an identifier (rule, token, or label name):

These more or less correspond to `isJavaIdentifierPart` and `isJavaIdentifierStart` in Java's `Character` class. Make sure to use the `-encoding` option on the ANTLR tool if your grammar file is not in UTF-8 format, so that ANTLR reads characters properly.

Literals

ANTLR does not distinguish between character and string literals as most languages do. All literal strings one or more characters in length are enclosed in single quotes such as `' ; '`, `' if '`, `'>= '`, and `'\ ' ' (refers to the one-character string containing the single quote character). Literals never contain regular expressions.`

Literals can contain Unicode escape sequences of the form `\uXXXX`, where `XXXX` is the hexadecimal Unicode character value. For example, `'\u00E8'` is the French letter with a grave accent: `'è'`. ANTLR also understands the usual special escape sequences: `'\n'` (newline), `'\r'` (carriage return), `'\t'` (tab), `'\b'` (backspace), and `'\f'` (form feed). You can use Unicode characters directly within literals or use the Unicode escape sequences. See `code/reference/Foreign.g4`:

```

grammar Foreign;
a: '外';

```

The recognizers that ANTLR generates assume a character vocabulary containing all Unicode characters. The input file encoding assumed by the runtime library depends on the target language. For the Java target, the runtime library assumes files are in UTF-8. Using the constructors, you can specify a different encoding. See, for example, ANTLR's `ANTLRFileStream`.

Actions

Actions are code blocks written in the target language. You can use actions in a number of places within a grammar, but the syntax is always the same: arbitrary text surrounded by curly braces. You don't need to escape a closing curly character if it's in a string or comment: `{ " } "` or `{ /* } *` `/ ; }`. If the curly braces are balanced, you also don't need to escape `}`: `{ { . . }`. Otherwise, escape extra curly braces with a backslash: `{ \ }` or `{ \ }`. The action text should conform to the target language as specified with the `language` option.

Embedded code can appear in: `@header` and `@members` named actions, parser and lexer rules, exception catching specifications, attribute sections for parser rules (return values, arguments, and locals), and some rule element options (currently predicates).

The only interpretation ANTLR does inside actions relates to grammar attributes; see [Token Attributes](#) and Chapter 10, [Attributes and Actions](#). Actions embedded within lexer rules are emitted without any interpretation or translation into generated lexers.

Keywords

Here's a list of the reserved words in ANTLR grammars:

```
import, fragment, lexer, parser, grammar, returns, locals, throws, catch, finally, mode, options, tokens.
```

Also, although it is not a keyword, do not use the word `rule` as a rule name. Further, do not use any keyword of the target language as a token, label, or rule name. For example, rule `if` would result in a generated function called `if`.

Grammar Structure

[ANTLR 4 Documentation Home](#)

- [Grammar Imports](#)
- [Tokens Section](#)
- [Actions at the Grammar Level](#)

A grammar is essentially a grammar declaration followed by a list of rules, but has the general form:

	<i>/** Optional javadoc style comment */</i>
	grammar Name ;
	options {...}
	import ... ;
	tokens {...}
	channels {...} // lexer only
	@actionName {...}
	<i>rule1 // parser and lexer rules, possibly intermingled</i>
	...
	<i>ruleN</i>

The file name containing grammar `X` must be called `X.g4`. You can specify options, imports, token specifications, and actions in any order. There can be at most one each of options, imports, and token specifications. All of those elements are optional except for the header and at least one rule. Rules take the basic form:

<code>ruleName : alternative1 ... alternativeN ;</code>

Parser rule names must start with a lowercase letter and lexer rules must start with a capital letter.

Grammars defined without a prefix on the `grammar` header are combined grammars that can contain both lexical and parser rules. To make a parser grammar that only allows parser rules, use the following header.

<code>parser grammar Name ;</code>
...

And, naturally, a pure lexer grammar looks like this:

<code>lexer grammar Name ;</code>
...

Only lexer grammars can contain `mode` specifications.

Only lexer grammars can contain custom channels specifications:

```

channels {
    WHITESPACE_CHANNEL,
    COMMENTS_CHANNEL
}

```

Those channels can then be used like enums within lexer rules:

```
WS : [ \r\t\n]+ -> channel(WHITESPACE_CHANNEL) ;
```

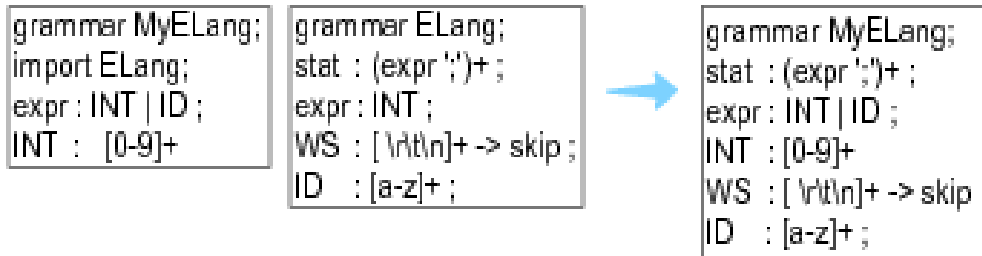
Sections Section 15.5, [Lexer Rules](#) and Section 15.3, [Parser Rules](#) contain details on rule syntax. Section Section 15.8, [Options](#) describes grammar options and Section 15.4, [Actions and Attributes](#) has information on grammar-level actions. We'll look at grammar imports, token specifications, and named actions next.

Grammar Imports

Grammar `imports` let you break up a grammar into logical and reusable chunks, as we saw in [Importing Grammars](#). ANTLR treats imported grammars very much like object-oriented programming languages treat superclasses. A grammar inherits all of the rules, `tokens` specifications, and named actions from the imported grammar. Rules in the “main grammar” override rules from imported grammars to implement inheritance.

Think of `import` as more like a smart include statement (which does not include rules that are already defined). The result of all imports is a single combined grammar; the ANTLR code generator sees a complete grammar and has no idea there were imported grammars.

To process a main grammar, the ANTLR tool loads all of the imported grammars into subordinate grammar objects. It then merges the rules, token types, and named actions from the imported grammars into the main grammar. In the diagram below, the grammar on the right illustrates the effect of grammar `MyELang` importing grammar `ELang`.

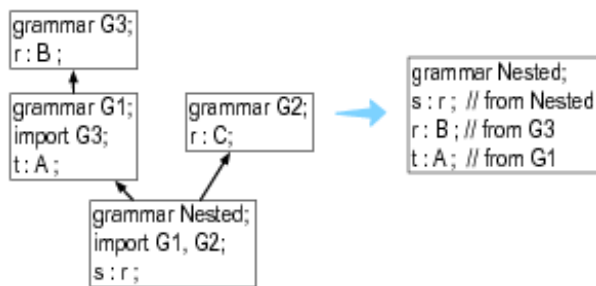


`MyELang` inherits rules `stat`, `WS`, and `ID`, but overrides rule `expr` and adds `INT`. Here's a sample build and test run that shows `MyELang` can recognize integer expressions whereas the original `ELang` can't. The third, erroneous input statement triggers an error message that also demonstrates the parser was looking for `MyELang`'s `expr` not `ELang`'s.

=>	\$ antlr4 MyELang.g4
=>	\$ javac MyELang*.java
=>	\$ grun MyELang stat
=>	34;
=>	a;
=>	;
=>	E _O F
<=	line 3:0 extraneous input ';' expecting {INT, ID}

If there were any `tokens` specifications, the main grammar would merge the token sets. Any named actions such as `@members` would be merged. In general, you should avoid named actions and actions within rules in imported grammars since that limits their reuse. ANTLR also ignores any options in imported grammars.

Imported grammars can also import other grammars. ANTLR pursues all imported grammars in a depth-first fashion. If two or more imported grammars define ruler, ANTLR chooses the first version of `r` it finds. In the following diagram, ANTLR examines grammars in the following order: Nested, G1, G3, G2.



Nested includes the `r` rule from G3 because it sees that version before the `r` in G2.

Not every kind of grammar can import every other kind of grammar:

- Lexer grammars can import lexers.
- Parsers can import parsers.
- Combined grammars can import lexers or parsers.

ANTLR adds imported rules to the end of the rule list in a main lexer grammar. That means lexer rules in the main grammar get precedence over imported rules. For example, if a main grammar defines rule `IF` : `'if'` ; and an imported grammar defines rule `ID` : `[a-z]+` ; (which also recognizes `if`), the imported `ID` won't hide the main grammar's `IF` token definition.

Tokens Section

The purpose of the `tokens` section is to define token types needed by a grammar for which there is no associated lexical rule. The basic syntax is:

```
tokens { Token1, ..., TokenN }
```

Most of the time, the `tokens` section is used to define token types needed by actions in the grammar (as we did in Section 10.3, [Recognizing Languages whose Keywords Aren't Fixed](#)):

```
// explicitly define keyword token types to avoid implicit definition warnings
tokens { BEGIN, END, IF, THEN, WHILE }

@lexer::members { // keywords map used in lexer to assign token types
Map<String,Integer> keywords = new HashMap<String,Integer>() {{
put("begin", KeywordsParser.BEGIN);
put("end", KeywordsParser.END);
...
}};
}
```

The `tokens` section really just defines a set of tokens to add to the overall set.

```
$ cat Tok.g4
grammar Tok;

tokens { A, B, C }

a : X ;

$ antlr4 Tok.g4
warning(125): Tok.g4:3:4: implicit definition of token X in parser
```

\$ cat Tok.tokens
A=1
B=2
C=3
X=4

Actions at the Grammar Level

Using Actions Outside of Grammar Rules illustrates the use of named actions at the top level of the grammar file. Currently there are only two defined actions (for the Java target): `header` and `members`. The former injects code into the generated recognizer class file, before the recognizer class definition, and the latter injects code into the recognizer class definition, as fields and methods.

For combined grammars, ANTLR injects the actions into both the parser and the lexer. To restrict an action to the generated parser or lexer, use `@parser:: name` or `@lexer:: name`.

Here's an example where the grammar specifies a package for the generated code:

reference/foo/Count.g4
grammar Count;
@header {
package foo;
}
@members {
int count = 0;
}
list
@after {System.out.println(count+" ints");}
: INT {count++;} (',' INT {count++;})*
;
INT : [0-9]+ ;
WS : [\r\t\n]+ -> skip ;

The grammar itself then should be in directory `foo` so that ANTLR generates code in that same `foo` directory (at least when not using the `-o ANTLR tool option`):

=>	\$ cd foo
=>	\$ antlr4 Count.g4 # generates code in the current directory (foo)
=>	\$ ls
<=	Count.g4 CountLexer.java CountParser.java
	Count.tokens CountLexer.tokens
	CountBaseListener.java CountListener.java

=>	\$ javac *.java
=>	\$ cd ..
=>	\$ grun foo.Count list
=>	9, 10, 11
=>	$\text{E}_\text{O}_\text{F}$
<=	3 ints

The Java compiler expects classes in package `foo` to be in directory `foo`.

Now that we've seen the overall structure of a grammar, let's dig into the parser and lexer rules.

Parser Rules

[ANTLR 4 Documentation Home](#)

- [Alternative Labels](#)
- [Rule Context Objects](#)
- [Rule Element Labels](#)
- [Rule Elements](#)
- [Subrules](#)
- [Catching Exceptions](#)
- [Rule Attribute Definitions](#)
- [Start Rules and EOF](#)

Parsers consist of a set of parser rules either in a `parser` or a combined grammar. A Java application launches a parser by invoking the rule function, generated by ANTLR, associated with the desired start rule. The most basic rule is just a rule name followed by a single alternative terminated with a semicolon:

	<i>/** Javadoc comment can precede rule */</i>
	<code>retstat : 'return' expr ';' ;</code>

Rules can also have alternatives separated by the `|` operator:

	<code>stat: retstat</code>
	<code> 'break' ';' ;</code>
	<code> 'continue' ';' ;</code>
	<code>;</code>

Alternatives are either a list of rule elements or empty. For example, here's a rule with an empty alternative that makes the entire rule optional:

	<code>superClass</code>
	<code>: 'extends' ID</code>
	<code> // empty means other alternative(s) are optional</code>
	<code>;</code>

Alternative Labels

As we saw in Section 7.4, [Labeling Rule Alternatives for Precise Event Methods](#), we can get more precise parse-tree listener events by labeling the outermost alternatives of a rule using the `#` operator. All alternatives within a rule must be labeled, or none of them. Here are two rules with labeled alternatives.

	grammar <code>T;</code>
	<code>stat: 'return' e ';' # Return</code>
	<code> 'break' ';' # Break</code>

	<code>;</code>
	<code>e : e '*' e # Mult</code>
	<code> e '+' e # Add</code>
	<code> INT # Int</code>
	<code>;</code>

Alternative labels do not have to be at the end of the line and there does not have to be a space after the # symbol.

ANTLR generates a rule context class definition for each label. For example, here is the listener that ANTLR generates:

<code>public interface AListener extends ParseTreeListener {</code>
<code>void enterReturn(AParser.ReturnContext ctx);</code>
<code>void exitReturn(AParser.ReturnContext ctx);</code>
<code>void enterBreak(AParser.BreakContext ctx);</code>
<code>void exitBreak(AParser.BreakContext ctx);</code>
<code>void enterMult(AParser.MultContext ctx);</code>
<code>void exitMult(AParser.MultContext ctx);</code>
<code>void enterAdd(AParser.AddContext ctx);</code>
<code>void exitAdd(AParser.AddContext ctx);</code>
<code>void enterInt(AParser.IntContext ctx);</code>
<code>void exitInt(AParser.IntContext ctx);</code>
<code>}</code>

There are enter and exit methods associated with each labeled alternative. The parameters to those methods are specific to alternatives.

You can reuse the same label on multiple alternatives to indicate that the parse tree walker should trigger the same event for those alternatives. For example, here's a variation on rule `e` from grammar A above:

<code>e : e '*' e # BinaryOp</code>
<code> e '+' e # BinaryOp</code>
<code> INT # Int</code>
<code>;</code>

ANTLR would generate the following listener methods for `e`:

<code>void enterBinaryOp(AParser.BinaryOpContext ctx);</code>
<code>void exitBinaryOp(AParser.BinaryOpContext ctx);</code>
<code>void enterInt(AParser.IntContext ctx);</code>
<code>void exitInt(AParser.IntContext ctx);</code>

ANTLR gives errors if an alternative name conflicts with a rule name. Here's another rewrite of rule `e` where two alternative labels conflict with rule names:

<code>e : e '*' e # e</code>
<code> e '+' e # Stat</code>
<code> INT # Int</code>
<code>;</code>

The context objects generated from rule names and labels get capitalized and so label `Stat` conflicts with rule `stat`:

\$ antlr4 A.g4
error(124): A.g4:5:23: rule alt label e conflicts with rule e
error(124): A.g4:6:23: rule alt label Stat conflicts with rule stat
warning(125): A.g4:2:13: implicit definition of token INT in parser

Rule Context Objects

ANTLR generates methods to access the rule context objects (parse tree nodes) associated with each rule reference. For rules with a single rule reference, ANTLR generates a method with no arguments. Consider the following rule.

inc : e '++' ;

ANTLR generates this context class:

public static class IncContext extends ParserRuleContext {
public EContext e() { ... } <i>// return context object associated with e</i>
...
}

ANTLR also provide support to access context objects when there is more than a single reference to a rule:

field : e '.' e ;

ANTLR generates a method with an index to access the *i*th element as well as a method to get context for all references to that rule:

public static class FieldContext extends ParserRuleContext {
public EContext e(int i) { ... } <i>// get ith e context</i>
public List<EContext> e() { ... } <i>// return ALL e contexts</i>
...
}

If we had another rule, `s`, that references `field`, an embedded action could access the list of `e` rule matches performed by `field`:

s : field
{
List<EContext> x = \$field.ctx.e();
...
}
;

A listener or visitor could do the same thing. Given a pointer to a `FieldContext` object, `f`, `f.e()` would return `List<EContext>`.

Rule Element Labels

You can label rule elements using the `=` operator to add fields to the rule context objects:

stat: 'return' value=e ';' # Return
'break' ';' # Break

```
;
```

Here `value` is the label for the return value of rule `e`, which is defined elsewhere.

Labels become fields in the appropriate parse tree node class. In this case, label `value` becomes a field in `ReturnContext` because of the `Return` alternative label:

```
public static class ReturnContext extends StatContext {
    public EContext value;
    ...
}
```

It's often handy to track a number of tokens, which you can do with the `+=` "list label" operator. For example, the following rule creates a list of the `Token` objects matched for a simple array construct:

```
array : '{' el+=INT (',' el+=INT)* '}' ;
```

ANTLR generates a `List` field in the appropriate rule context class:

```
public static class ArrayContext extends ParserRuleContext {
    public List<Token> el = new ArrayList<Token>();
    ...
}
```

These list labels also work for rule references:

```
elist : exprs+=e (',' exprs+=e)* ;
```

ANTLR generates a field holding the list of context objects:

```
public static class ElistContext extends ParserRuleContext {
    public List<EContext> exprs = new ArrayList<EContext>();
    ...
}
```

Rule Elements

Rule elements specify what the parser should do at a given moment just like statements in a programming language. The elements can be rule, token, string literal like `expression`, `ID`, and `'return'`. Here's a complete list of the rule elements (we'll look at actions and predicates in more detail later):

Syntax	Description
T	Match token T at the current input position. Tokens always begin with a capital letter.
'literal'	Match the string literal at the current input position. A string literal is simply a token with a fixed string.
r	Match rule r at current input position, which amounts to invoking the rule just like a function call. Parser rule names always begin with a lowercase letter.
r [«args»]	Match rule r at current input position, passing in a list of arguments just like a function call. The arguments inside the square brackets are in the syntax of the target language and are usually a comma-separated list of expressions.
{«action»}	Execute an action immediately after the preceding alternative element and immediately before the following alternative element. The action conforms to the syntax of the target language. ANTLR copies the action code to the generated class verbatim, except for substituting attribute and token references such as <code>\$x</code> and <code>\$x.y</code> .

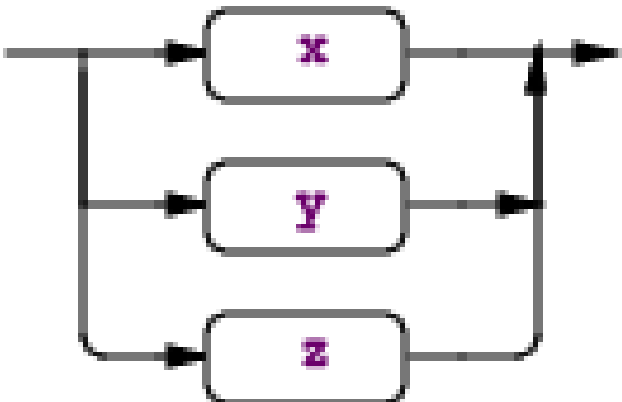
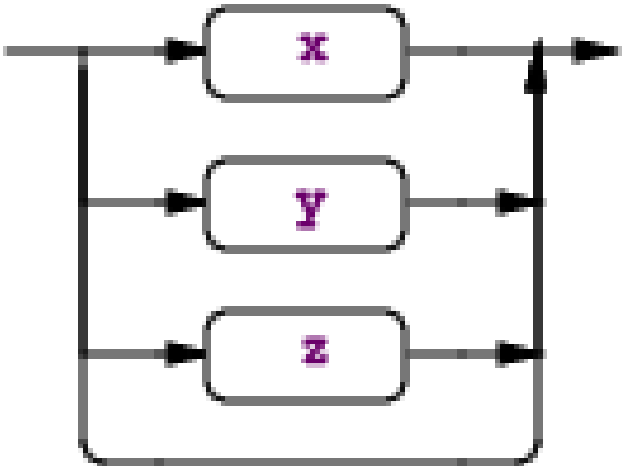
{«p»?}	Evaluate semantic predicate «p». Do not continue parsing past a predicate if «p» evaluates to false at runtime. Predicates encountered during prediction, when ANTLR distinguishes between alternatives, enable or disable the alternative(s) surrounding the predicate(s).
.	Match any single token except for the end of file token. The “dot” operator is called the wildcard.

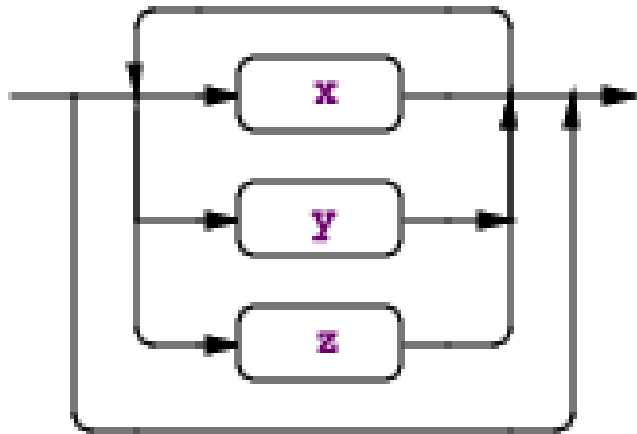
When you want to match everything but a particular token or set of tokens, use the ~ “not” operator. This operator is rarely used in the parser but is available. ~INT matches any token except the INT token. ~', ' matches any token except the comma. ~(INT | ID) matches any token except an INT or an ID.

Token, string literal, and semantic predicate rule elements can take options. See [Rule Element Options](#).

Subrules

A rule can contain alternative blocks called subrules (as allowed in Extended BNF Notation: EBNF). A subrule is like a rule that lacks a name and is enclosed in parentheses. Subrules can have one or more alternatives inside the parentheses. Subrules cannot define attributes with locals and returns like rules can. There are four kinds of subrules (x, y, and z represent grammar fragments):

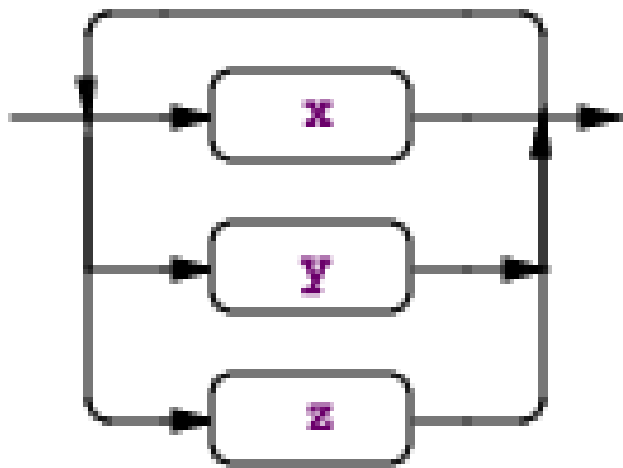
	<div>(x y z)</div> <div>Match any alternative within the subrule exactly once. Example:</div> <table><tr><td>returnType</td><td>:</td><td>(type 'void')</td><td>;</td></tr></table>	returnType	:	(type 'void')	;							
returnType	:	(type 'void')	;									
	<div>(x y z)?</div> <div>Match nothing or any alternative within subrule. Example:</div> <table><tr><td>classDeclaration</td></tr><tr><td>:</td><td>'class'</td><td>ID</td><td>(typeParameters)?</td><td>('extends' type)?</td></tr><tr><td></td><td>('implements' typeList)?</td></tr><tr><td>classBody</td></tr><tr><td>:</td><td>;</td></tr></table>	classDeclaration	:	'class'	ID	(typeParameters)?	('extends' type)?		('implements' typeList)?	classBody	:	;
classDeclaration												
:	'class'	ID	(typeParameters)?	('extends' type)?								
	('implements' typeList)?											
classBody												
:	;											



$(x|y|z)^*$

Match an alternative within subrule zero or more times. Example:

```
annotationName : ID ( '.' ID ) * ;
```



$(x|y|z)^+$

Match an alternative within subrule one or more times. Example:

```
annotations : (annotation) + ;
```

You can suffix the `?`, `*`, and `+` subrule operators with the nongreedy operator, which is also a question mark: `??`, `?*`, and `?+`. See Section 15.6, [Wildcard Operator and Nongreedy Subrules](#).

As a shorthand, you can omit the parentheses for subrules composed of a single alternative with a single rule element reference. For example, `notation+` is the same as `(notation)+` and `ID+` is the same as `(ID)+`. Labels also work with the shorthand. `ids+=INT+` make a list of `INT` token objects.

Catching Exceptions

When a syntax error occurs within a rule, ANTLR catches the exception, reports the error, attempts to recover (possibly by consuming more tokens), and then returns from the rule. Every rule is wrapped in a `try/catch/finally` statement:

```
void r() throws RecognitionException {
    try {
        rule-body
    }
    catch (RecognitionException re) {
        _errHandler.reportError(this, re);
        _errHandler.recover(this, re);
    }
}
```

finally {
exitRule();
}
}

In Section 9.5, [Altering ANTLR's Error Handling Strategy](#), we saw how to use a strategy object to alter ANTLR's error handling. Replacing the strategy changes the strategy for all rules, however. To alter the exception handling for a single rule, specify an exception after the rule definition:

r : ...
;
catch [RecognitionException e] { throw e; }

That example shows how to avoid default error reporting and recovery. `r` rethrows the exception, which is useful when it makes more sense for a higher-level rule to report the error. Specifying any exception clause, prevents ANTLR from generating a clause to handle `RecognitionException`.

You can specify other exceptions as well:

r : ...
;
catch [FailedPredicateException fpe] { ... }
catch [RecognitionException e] { ... }

The code snippets inside curly braces and the exception “argument” actions must be written in the target language; Java, in this case.

When you need to execute an action even if an exception occurs, put it into the `finally` clause:

r : ...
;
<i>// catch blocks go first</i>
finally { System.out.println("exit rule r"); }

The `finally` clause executes right before the rule triggers `exitRule` before returning. If you want to execute an action after the rule finishes matching the alternatives but before it does its cleanup work, use an `after` action.

Here's a complete list of exceptions:

Exception name	Description
<code>RecognitionException</code>	The superclass of all exceptions thrown by an ANTLR-generated recognizer. It's a subclass of <code>RuntimeException</code> to avoid the hassles of checked exceptions. This exception records where the recognizer (lexer or parser) was in the input, where it was in the ATN (internal graph data structure representing the grammar), the rule invocation stack, and what kind of problem occurred.
<code>NoViableAltException</code>	Indicates that the parser could not decide which of two or more paths to take by looking at the remaining input. This exception tracks the starting token of the offending input and also knows where the parser was in the various paths when the error occurred.
<code>LexerNoViableAltException</code>	The equivalent of <code>NoViableAltException</code> but for lexers only.
<code>InputMismatchException</code>	The current input <code>Token</code> does not match what the parser expected.
<code>FailedPredicateException</code>	A semantic predicate that evaluates to false during prediction renders the surrounding alternative nonviable. Prediction occurs when a rule is predicting which alternative to take. If all viable paths disappear, parser will throw <code>NoViableAltException</code> . This predicate gets thrown by the parser when a semantic predicate evaluates to false outside of prediction, during the normal parsing process of matching tokens and calling rules.

Rule Attribute Definitions

There are a number of action-related syntax elements associated with rules to be aware of. Rules can have arguments, return values, and local variables just like functions in a programming language. (Rules can have actions embedded among the rule elements, as we'll see in Section 15.4, [Actions and Attributes](#).) ANTLR collects all of the variables you define and stores them in the rule context object. These variables are usually called attributes. Here's the general syntax showing all possible attribute definition locations:

```
rulename[args] returns [retvals] locals [localvars] : ... ;
```

The attributes defined within those [...] can be used like any other variable. Here is a sample rule that copies parameters to return values:

```
// Return the argument plus the integer value of the INT token  
add[int x] returns [int result] : '+' INT {$result = $x + $INT.int;} ;
```

As with the grammar level, you can specify rule-level named actions. For rules, the valid names are `init` and `after`. As the names imply, parsers execute `init` actions immediately before trying to match the associated rule and execute `after` actions immediately after matching the rule. ANTLR `after` actions do not execute as part of the `finally` code block of the generated rule function. Use the ANTLR `finally` action to place code in the generated rule function `finally` code block.

The actions come after any argument, return value, or local attribute definition actions. The `row` rule preamble from Section 10.2, [Accessing Token and Rule Attributes](#) illustrates the syntax nicely:

actions/CSV.g4	
	<i>/** Derived from rule "row : field (',' field)* '\r'? '\n' ;" */</i>
	<code>row[String[] columns] returns [Map<String,String> values]</code>
	<code>locals [int col=0]</code>
	<code>@init {</code>
	<code> \$values = new HashMap<String,String>();</code>
	<code>}</code>
	<code>@after {</code>
	<code> if (\$values!=null && \$values.size()>0) {</code>
	<code> System.out.println("values = "+\$values);</code>
	<code> }</code>
	<code>}</code>

Rule `row` takes argument `columns`, returns values, and defines local variable `col`. The “actions” in square brackets are copied directly into the generated code:

<code>public class CSVParser extends Parser {</code>
<code> ...</code>
<code> public static class RowContext extends ParserRuleContext {</code>
<code> public String [] columns;</code>
<code> public Map<String,String> values;</code>
<code> public int col=0;</code>
<code> ...</code>
<code> }</code>
<code> ...</code>
<code>}</code>

The generated rule functions also specify the rule arguments as function arguments, but they are quickly copied into the local `RowContext` object

:

<code>public class CSVParser extends Parser {</code>
<code>...</code>
<code>public final RowContext row(String [] columns) throws RecognitionException {</code>
<code>RowContext _localctx = new RowContext(_ctx, 4, columns);</code>
<code>enterRule(_localctx, RULE_row);</code>
<code>...</code>
<code>}</code>
<code>...</code>
<code>}</code>

ANTLR tracks nested [...] within the action so that `String[] columns` is parsed properly. It also tracks angle brackets so that commas within generic type parameters do not signify the start of another attribute. `Map<String,String> values` is one attribute definition.

There can be multiple attributes in each action, even for return values. Use a comma to separate attributes within the same action:

<code>a[Map<String,String> x, int y] : ... ;</code>

ANTLR interprets that action to define two arguments, `x` and `y`:

<code>public final AContext a(Map<String,String> x, int y)</code>
<code>throws RecognitionException</code>
<code>{</code>
<code>AContext _localctx = new AContext(_ctx, 0, x, y);</code>
<code>enterRule(_localctx, RULE_a);</code>
<code>...</code>
<code>}</code>

Start Rules and EOF

A start rule is the rule engaged first by the parser; it's the rule function called by the language application. For example, a language application that parsed to Java code might call `parser.compilationUnit()` for a `JavaParser` object called `parser`. Any rule in the grammar can act as a start rule.

Start rules don't necessarily consume all of the input. They consume only as much input as needed to match an alternative of the rule. For example, consider the following rule that matches one, two, or three tokens, depending on the input.

<code>s : ID</code>
<code> ID '+'</code>
<code> ID '+' INT</code>
<code>;</code>

Upon `a+3`, rule `s` matches the third alternative. Upon `a+b`, it matches the second alternative and ignores the final `b` token. Upon `a b`, it matches the first alternative, ignoring the `b` token. The parser does not consume the complete input in the latter two cases because rule `s` doesn't explicitly say that end of file must occur after matching an alternative of the rule.

This default functionality is very useful for building things like IDEs. Imagine the IDE wanting to parse a method somewhere in the middle of a big Java file. Calling rule `methodDeclaration` should try to match just a method and ignore whatever comes next.

On the other hand, rules that describe entire input files should reference special predefined-token `EOF`. If they don't, you might scratch your head for a while wondering why the start rule doesn't report errors for any input no matter what you give it. Here's a rule that's part of a grammar for reading configuration files:

```
config : element*; // can "match" even with invalid input.
```

Invalid input would cause `config` to return immediately without matching any input and without reporting an error. Here's the proper specification:

```
file : element* EOF; // don't stop early. must match all input
```

Left-recursive rules

The most natural expression of some common language constructs is left recursive. For example C declarators and arithmetic expressions. Unfortunately, left recursive specifications of arithmetic expressions are typically ambiguous but much easier to write out than the multiple levels required in a typical top-down grammar. Here is a sample ANTLR 4 grammar with a left recursive expression rule:

```
stat: expr '=' expr ';' // e.g., x=y; or x=f(x);
    | expr ';'          // e.g., f(x); or f(g(x));
    ;
expr: expr '*' expr
    | expr '+' expr
    | expr '(' expr ')' // f(x)
    | id
    ;
```

In straight context free grammars, such a rule is ambiguous because $1+2*3$ it can interpret either operator as occurring first, but ANTLR rewrites that to be non-left recursive and unambiguous using semantic predicates:

```
expr[int pr] : id
              ( {4 >= $pr}? '*' expr[5]
                | {3 >= $pr}? '+' expr[4]
                | {2 >= $pr}? '(' expr[0] ')'
                )*
              ;
```

The predicates resolve ambiguities by comparing the precedence of the current operator against the precedence of the previous operator. An expansion of `expr[pr]` can match only those subexpressions whose precedence meets or exceeds `pr`.

Formal rules

The formal 4.0, 4.1 ANTLR left-recursion elimination rules are laid out in the [ALL\(*\) tech report](#), but have been simplified for 4.2:

- **Binary expressions** are expressions which contain a recursive invocation of the rule as the first and last element of the alternative.
- **Suffix expressions** contain a recursive invocation of the rule as the first element of the alternative, but not as the last element.
- **Prefix expressions** contain a recursive invocation of the rule as the last element of the alternative, but not as the first element.

There is no such thing as a "ternary" expression--they are just binary expressions in disguise.

The right associativity specifiers used to be on the individual tokens but it's done on alternative basis anyway so the option is now on the individual alternative; e.g.,

```

e : e '*' e
  | e '+' e
  | <assoc=right> e '?' e ':' e
  | <assoc=right> e '=' e
  | INT
  ;

```

If your 4.0 or 4.1 grammar uses a right-associative ternary operator, you will need to update your grammar to include `<assoc=right>` on the alternative operator. To smooth the transition, `<assoc=right>` is still allowed on token references but it is ignored.

Actions and Attributes

[ANTLR 4 Documentation Home](#)

- [Token Attributes](#)
- [Parser Rule Attributes](#)
- [Dynamically-Scoped Attributes](#)

In Chapter 10, *Attributes and Actions*, we learned how to embed actions within grammars and looked at the most common token and rule attributes. This section summarizes the important syntax and semantics from that chapter and provides a complete list of all available attributes. (You can learn more about actions in the grammar from the free [excerpt on listeners and actions](#).)

Actions are blocks of text written in the target language and enclosed in curly braces. The recognizer triggers them according to their locations within the grammar. For example, the following rule emits `found a decl` after the parser has seen a valid declaration:

decl: type ID ';' {System.out.println("found a decl");} ;
type: 'int' 'float' ;

Most often, actions access the attributes of tokens and rule references:

decl: type ID ';' {System.out.println("var "+\$ID.text+":"+\$type.text+";");}
t=ID id=ID ';' {System.out.println("var "+\$id.text+":"+\$t.text+";");}
;

Token Attributes

All tokens have a collection of predefined, read-only attributes. The attributes include useful token properties such as the token type and text matched for a token. Actions can access these attributes via `$ label.attribute` where `label` labels a particular instance of a token reference (`a` and `b` in the example below are used in the action code as `$a` and `$b`). Often, a particular token is only referenced once in the rule, in which case the token name itself can be used unambiguously in the action code (token `INT` can be used as `$INT` in the action). The following example illustrates token attribute expression syntax:

r : INT {int x = \$INT.line;}
(ID {if (\$INT.line == \$ID.line) ...;})?
a=FLOAT b=FLOAT {if (\$a.line == \$b.line) ...;}
;

The action within the `(...)?` subrule can see the `INT` token matched before it in the outer level.

Because there are two references to the `FLOAT` token, a reference to `$FLOAT` in an action is not unique; you must use labels to specify which token reference you're interested in.

Token references within different alternatives are unique because only one of them can be matched for any invocation of the rule. For example, in the following rule, actions in both alternatives can reference `$ID` directly without using a label:

```
r : ... ID {System.out.println($ID.text);}
  | ... ID {System.out.println($ID.text);}
  ;
```

To access the tokens matched for literals, you must use a label:

```
stat: r='return' expr ';' {System.out.println("line="+$r.line);} ;
```

Most of the time you access the attributes of the token, but sometimes it is useful to access the `Token` object itself because it aggregates all the attributes. Further, you can use it to test whether an optional subrule matched a token:

```
stat: 'if' expr 'then' stat (el='else' stat)?
    {if ( $el!=null ) System.out.println("found an else");}
    | ...
    ;
```

`$T` and `$l` evaluate to `Token` objects for token name `T` and token label `l`. `$ll` evaluates to `List<Token>` for list label `ll`. `$T.attr` evaluates to the type and value specified in the following table for attribute `attr`:

Attribute	Type	Description
text	String	The text matched for the token; translates to a call to <code>getText</code> . Example: <code>\$ID.text</code> .
type	int	The token type (nonzero positive integer) of the token such as <code>INT</code> ; translates to a call to <code>getType</code> . Example: <code>\$ID.type</code> .
line	int	The line number on which the token occurs, counting from 1; translates to a call to <code>getLine</code> . Example: <code>\$ID.line</code> .
pos	int	The character position within the line at which the token's first character occurs counting from zero; translates to a call to <code>getCharPositionInLine</code> . Example: <code>\$ID.pos</code> .
index	int	The overall index of this token in the token stream, counting from zero; translates to a call to <code>getTokenIndex</code> . Example: <code>\$ID.index</code> .
channel	int	The token's channel number. The parser tunes to only one channel, effectively ignoring off-channel tokens. The default channel is 0 (<code>Token.DEFAULT_CHANNEL</code>), and the default hidden channel is <code>Token.HIDDEN_CHANNEL</code> . Translates to a call to <code>getChannel</code> . Example: <code>\$ID.channel</code> .
int	int	The integer value of the text held by this token; it assumes that the text is a valid numeric string. Handy for building calculators and so on. Translates to <code>Integer.valueOf(text-of-token)</code> . Example: <code>\$INT.int</code> .

Parser Rule Attributes

ANTLR predefines a number of read-only attributes associated with parser rule references that are available to actions. Actions can access rule attributes only for references that precede the action. The syntax is `$r.attr` for rule name `r` or a label assigned to a rule reference. For example, `$expr.text` returns the complete text matched by a preceding invocation of rule `expr`:

```
returnStat : 'return' expr {System.out.println("matched "+$expr.text);} ;
```

Using a rule label looks like this:

```
returnStat : 'return' e=expr {System.out.println("matched "+$e.text);} ;
```

You can also use `$` followed by the name of the attribute to access the value associated with the currently executing rule. For example, `$start` is the starting token of the current rule.

```
returnStat : 'return' expr {System.out.println("first token "+$start.getText());} ;
```

`$r` and `$rl` evaluate to `ParserRuleContext` objects of type `R Context` for rule name `r` and rule label `rl`. `$rll` evaluates to `List< R Context >`

for rule list label `rll`. `$ r . attr` evaluates to the type and value specified in the following table for attribute `attr`:

Attribute	Type	Description
text	String	The text matched for a rule or the text matched from the start of the rule up until the point of the <code>\$text</code> expression evaluation. Note that this includes the text for all tokens including those on hidden channels, which is what you want because usually that has all the whitespace and comments. When referring to the current rule, this attribute is available in any action including any exception actions.
start	Token	The first token to be potentially matched by the rule that is on the main token channel; in other words, this attribute is never a hidden token. For rules that end up matching no tokens, this attribute points at the first token that could have been matched by this rule. When referring to the current rule, this attribute is available to any action within the rule.
stop	Token	The last nonhidden channel token to be matched by the rule. When referring to the current rule, this attribute is available only to the <code>after</code> and <code>finally</code> actions.
ctx	ParserRuleContext	The rule context object associated with a rule invocation. All of the other attributes are available through this attribute. For example, <code>\$ctx.start</code> accesses the <code>start</code> field within the current rules context object. It's the same as <code>\$start</code> .

Dynamically-Scoped Attributes

You can pass information to and from rules using parameters and return values, just like functions in a general-purpose programming language. Programming languages don't allow functions to access the local variables or parameters of invoking functions, however. For example, the following reference to local variable `x` from a nested method call is illegal in Java:

<code>void f() {</code>
<code>int x = 0;</code>
<code>g();</code>
<code>}</code>
<code>void g() {</code>
<code>h();</code>
<code>}</code>
<code>void h() {</code>
<code>int y = x; // INVALID reference to f's local variable x</code>
<code>}</code>

Variable `x` is available only within the scope of `f`, which is the text lexically delimited by curly brackets. For this reason, Java is said to use lexical scoping. Lexical scoping is the norm for most programming languages. Languages that allow methods further down in the call chain to access local variables defined earlier are said to use dynamic scoping. The term dynamic refers to the fact that a compiler cannot statically determine the set of visible variables. This is because the set of variables visible to a method changes depending on who calls that method.

It turns out that, in the grammar realm, distant rules sometimes need to communicate with each other, mostly to provide context information to rules matched below in the rule invocation chain. (Naturally, this assumes that you are using actions directly in the grammar instead of the parse-tree listener event mechanism.) ANTLR allows dynamic scoping in that actions can access attributes from invoking rules using syntax `$r:x` where `r` is a rule name and `x` is an attribute within that rule. It is up to the programmer to ensure that `r` is in fact an invoking rule of the current rule. A runtime exception occurs if `r` is not in the current call chain when you access `$r:x`.

To illustrate the use of dynamic scoping, consider the real problem of defining variables and ensuring that variables in expressions are defined. The following grammar defines the `symbols` attribute where it belongs in the `block` rule but adds variable names to it in rule `decl`. Rule `stat` then consults the list to see whether variables have been defined.

reference/DynScope.g4	
<code>grammar DynScope;</code>	
<code>prog: block</code>	

;
block
<i>/* List of symbols defined within this block */</i>
locals [
List<String> symbols = new ArrayList<String>()
]
: '{' decl* stat+ '}'
<i>// print out all symbols found in block</i>
<i>// \$block::symbols evaluates to a List as defined in scope</i>
{System.out.println("symbols="+\$symbols);}
;
<i>/** Match a declaration and add identifier name to list of symbols */</i>
decl: 'int' ID {\$block::symbols.add(\$ID.text);} ';' ;
;
<i>/** Match an assignment then test list of symbols to verify</i>
<i>* that it contains the variable on the left side of the assignment.</i>
<i>* Method contains() is List.contains() because \$block::symbols</i>
<i>* is a List.</i>
<i>*/</i>
stat: ID '=' INT ';' ;
{
if (!\$block::symbols.contains(\$ID.text)) {
System.err.println("undefined variable: "+\$ID.text);
}
}
block
;
ID : [a-z]+ ;
INT : [0-9]+ ;
WS : [\t\r\n]+ -> skip ;

Here's a simple build and test sequence:

=>	\$ antlr4 DynScope.g4
=>	\$ javac DynScope*.java

=>	\$ grun DynScope prog
=>	{
=>	int i;
=>	i = 0;
=>	j = 3;
=>	}
=>	$\mathbb{E}_{\mathbb{O}_F}$
<=	undefined variable: j
	symbols=[i]

There's an important difference between a simple field declaration in a `@members` action and dynamic scoping. `symbols` is a local variable and so there is a copy for each invocation of rule `block`. That's exactly what we want for nested blocks so that we can reuse the same input variable name in an inner block. For example, the following nested code block redefines `i` in the inner scope. This new definition must hide the definition in the outer scope.

reference/nested-input	
	{
	int i;
	int j;
	i = 0;
	{
	int i;
	int x;
	x = 5;
	}
	x = 3;
	}

Here's the output generated for that input by DynScope:

	\$ grun DynScope prog nested-input
	symbols=[i, x]
	undefined variable: x
	symbols=[i, j]

Referencing `$block::symbols` accesses the `symbols` field of the most recently invoked `block`'s rule context object. If you need access to a `symbols` instance from a rule invocation farther up the call chain, you can walk backwards starting at the current context, `$ctx`. Use `getParent` to walk up the chain.

Lexer Rules

[ANTLR 4 Documentation Home](#)

- [Lexical Modes](#)
- [Lexer Rule Elements](#)
- [Recursive Lexer Rules](#)
- [Redundant String Literals](#)
- [Lexer Rule Actions](#)
- [Lexer Commands](#)
 - [skip](#)

- `mode()`, `pushMode()`, `popMode()`, and more
- `type()`
- `channel()`

A lexer grammar is composed of lexer rules, optionally broken into multiple `modes`, as we saw in [Issuing Context-Sensitive Tokens with Lexical Modes](#). Lexical modes allow us to split a single lexer grammar into multiple sublexers. The lexer can only return tokens matched by rules from the current mode.

Lexer rules specify token definitions and more or less follow the syntax of parser rules except that lexer rules cannot have arguments, return values, or local variables. Lexer rule names must begin with an uppercase letter, which distinguishes them from parser rule names:

	<code>/** Optional document comment */</code>
	<code>TokenName : alternative1 ... alternativeN ;</code>

You can also define rules that are not tokens but rather aid in the recognition of tokens. These `fragment` rules do not result in tokens visible to the parser:

	<code>fragment HelperTokenRule : alternative1 ... alternativeN ;</code>
--	--

For example, `DIGIT` is a pretty common `fragment` rule:

	<code>INT : DIGIT+ ; // references the DIGIT helper rule</code>
	<code>fragment DIGIT : [0-9] ; // not a token by itself</code>

Lexical Modes

Modes allow you to group lexical rules by context, such as inside and outside of XML tags. It's like having multiple sublexers, one for context. The lexer can only return tokens matched by entering a rule in the current mode. Lexers start out in the so-called default mode. All rules are considered to be within the default mode unless you specify a `mode` command. Modes are not allowed within combined grammars, just `lexer` grammars. (See grammar `XMLLexer` from [Tokenizing XML](#).)

	<code>rules in default mode</code>
	<code>...</code>
	<code>mode MODE1;</code>
	<code>rules in MODE1</code>
	<code>...</code>
	<code>mode MODEN;</code>
	<code>rules in MODEN</code>
	<code>...</code>

Lexer Rule Elements

Lexer rules allow two constructs that are unavailable to parser rules: the `..` range operator and the character set notation enclosed in square brackets, `[characters]`. Don't confuse character sets with arguments to parser rules. `[characters]` only means character set in a lexer. Here's a summary of all lexer rule elements:

Syntax	Description
'literal'	Match that character or sequence of characters. E.g., <code>'while'</code> or <code>'='</code> .

[char set]	<p>Match one of the characters specified in the character set. Interpret x-y as set of characters between range x and y, inclusively. The following escaped characters are interpreted as single special characters: \n, \r, \b, \t, and \f. To get], \, or - you must escape them with \. You can also use Unicode character specifications: \uXXXX. Here are a few examples:</p> <table><tr><td>WS</td><td>: [\n\u000D] -> skip ; // same as [\n\r]</td></tr><tr><td>ID</td><td>: [a-zA-Z] [a-zA-Z0-9]* ; // match usual identifier spec</td></tr><tr><td>DASHBRACK</td><td>: [\-\]]+ ; // match - or] one or more times</td></tr></table>	WS	: [\n\u000D] -> skip ; // same as [\n\r]	ID	: [a-zA-Z] [a-zA-Z0-9]* ; // match usual identifier spec	DASHBRACK	: [\-\]]+ ; // match - or] one or more times
WS	: [\n\u000D] -> skip ; // same as [\n\r]						
ID	: [a-zA-Z] [a-zA-Z0-9]* ; // match usual identifier spec						
DASHBRACK	: [\-\]]+ ; // match - or] one or more times						
'x'..'y'	<p>Match any single character between range x and y, inclusively. E.g., 'a'..'z'. 'a'..'z' is identical to [a-z].</p>						
T	<p>Invoke lexer rule T; recursion is allowed in general, but not left recursion. T can be a regular token or fragment rule.</p> <table><tr><td>ID</td><td>: LETTER (LETTER '0'..'9')* ;</td></tr><tr><td>fragment</td><td></td></tr><tr><td>LETTER</td><td>: [a-zA-Z\u0080-\u00FF_] ;</td></tr></table>	ID	: LETTER (LETTER '0'..'9')* ;	fragment		LETTER	: [a-zA-Z\u0080-\u00FF_] ;
ID	: LETTER (LETTER '0'..'9')* ;						
fragment							
LETTER	: [a-zA-Z\u0080-\u00FF_] ;						
.	<p>The dot is a single-character wildcard that matches any single character. Example:</p> <table><tr><td>ESC</td><td>: '\\\ ' . ; // match any escaped \x character</td></tr></table>	ESC	: '\\\ ' . ; // match any escaped \x character				
ESC	: '\\\ ' . ; // match any escaped \x character						
{«action»}	<p>Lexer actions can appear anywhere as of 4.2, not just at the end of the outermost alternative. The lexer executes the actions at the appropriate input position, according to the placement of the action within the rule. To execute a single action for a rule that has multiple alternatives, you can enclose the alts in parentheses and put the action afterwards:</p> <table><tr><td>END</td><td>: ('endif' 'end') {System.out.println("found an end");} ;</td></tr></table> <p>The action conforms to the syntax of the target language. ANTLR copies the action's contents into the generated code verbatim; there is no translation of expressions like \$x.y as there is in parser actions.</p> <p>Only actions within the outermost token rule are executed. In other words, if STRING calls ESC_CHAR and ESC_CHAR has an action, that action is not executed when the lexer starts matching in STRING.</p>	END	: ('endif' 'end') {System.out.println("found an end");} ;				
END	: ('endif' 'end') {System.out.println("found an end");} ;						
{«p»?}	<p>Evaluate semantic predicate «p». If «p» evaluates to false at runtime, the surrounding rule becomes “invisible” (nonviable). Expression «p» conforms to the target language syntax. While semantic predicates can appear anywhere within a lexer rule, it is most efficient to have them at the end of the rule. The one caveat is that semantic predicates must precede lexer actions. See Predicates in Lexer Rules.</p>						
~x	<p>Match any single character not in the set described by x. Set x can be a single character literal, a range, or a subrule set like ~('x' 'y' 'z') or ~[xyz]. Here is a rule that uses ~ to match any character other than characters using ~[\r\n]*:</p> <table><tr><td>COMMENT</td><td>: '#' ~[\r\n]* '\r'? '\n' -> skip ;</td></tr></table>	COMMENT	: '#' ~[\r\n]* '\r'? '\n' -> skip ;				
COMMENT	: '#' ~[\r\n]* '\r'? '\n' -> skip ;						

Just as with parser rules, lexer rules allow subrules in parentheses and EBNF operators: ?, *, +. The COMMENT rule illustrates the * and ? operators. A common use of + is [0-9]+ to match integers. Lexer subrules can also use the nongreedy ? suffix on those EBNF operators.

Recursive Lexer Rules

ANTLR lexer rules can be recursive, unlike most lexical grammar tools. This comes in really handy when you want to match nested tokens like nested action blocks: { ... { ... } ... }.

reference/Recur.g4	
	lexer grammar Recur;
	ACTION : '{' (ACTION ~[{}]* '}')* '}' ;

	WS : [\r\t\n]+ -> skip ;
--	---------------------------

Redundant String Literals

Be careful that you don't specify the same string literal on the right-hand side of multiple lexer rules. Such literals are ambiguous and could match multiple token types. ANTLR makes this literal unavailable to the parser. The same is true for rules across modes. For example, the following lexer grammar defines two tokens with the same character sequence:

reference/L.g4	
	lexer grammar L;
	AND : '&' ;
	mode STR;
	MASK : '&' ;

A parser grammar cannot reference literal '&', but it can reference the name of the tokens:

reference/P.g4	
	parser grammar P;
	options { tokenVocab=L; }
	a : '&' <i>// results in a tool error: no such token</i>
	AND <i>// no problem</i>
	MASK <i>// no problem</i>
	;

Here's a build and test sequence:

=>	\$ antlr4 L.g4 # yields L.tokens file needed by tokenVocab option in P.g4
=>	\$ antlr4 P.g4
<=	error(126): P.g4:3:4: cannot create implicit token for string literal '&'
	in non-combined grammar

Lexer Rule Actions

An ANTLR lexer creates a Token object after matching a lexical rule. Each request for a token starts in `Lexer.nextToken`, which calls `emit` once it has identified a token. `emit` collects information from the current state of the lexer to build the token. It accesses fields `_type`, `_text`, `_channel`, `_tokenStartCharIndex`, `_tokenStartLine`, and `_tokenStartCharPositionInLine`. You can set the state of these with the various setter methods such as `setType`. For example, the following rule turns `enum` into an identifier if `enumIsKeyword` is false.

ENUM : 'enum' {if (!enumIsKeyword) setType(Identifier);} ;
--

ANTLR does no special \$ x attribute translations in lexer actions (unlike v3).

There can be at most a single action for a lexical rule, regardless of how many alternatives there are in that rule.

Lexer Commands

To avoid tying a grammar to a particular target language, ANTLR supports lexer commands. Unlike arbitrary embedded actions, these commands follow specific syntax and are limited to a few common commands. Lexer commands appear at the end of the outermost alternative of a lexer rule definition. Like arbitrary actions, there can only be one per token rule. A lexer command consists of the `->` operator followed by one or more command names that can optionally take parameters:

TokenName : «alternative» -> command-name

TokenName : «alternative» -> command-name («identifier or integer»)

An alternative can have more than one command separated by commas. Here are the valid command names:

- skip
- more
- popMode
- mode(x)
- pushMode(x)
- type(x)
- channel(x)

See the [book source code](#) for usage, some examples of which are shown here:

skip

A 'skip' command tells the lexer to get another token and throw out the current text.

From actions/tool/Expr.g4

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

mode(), pushMode(), popMode, and more

The mode commands alter the mode stack and hence the mode of the lexer. The 'more' command forces the lexer to get another token but without throwing out the current text. The token type will be that of the "final" rule matched (i.e., the one without a more or skip command).

From lemagic/XMLLexer.g4

```
// Default "mode": Everything OUTSIDE of a tag
COMMENT : '<!--' .*? '-->' ;
CDATA : '<![CDATA[' .*? ']]>' ; OPEN : '<' -> pushMode(INSIDE) ;

...

XMLDeclOpen : '<?xml' S -> pushMode(INSIDE) ;
SPECIAL_OPEN : '<?' Name -> more, pushMode(PROC_INSTR) ;

// ----- Everything INSIDE of a tag -----
mode INSIDE;

CLOSE : '>' -> popMode ;
SPECIAL_CLOSE : '?>' -> popMode ; // close <?xml...?>
SLASH_CLOSE : '/>' -> popMode ;
```

From reference/Strings.g4

```
lexer grammar Strings;
LQUOTE : '"' -> more, mode(STR) ;
WS : [ \t\n]+ -> skip ;

mode STR;

STRING : '"' -> mode(DEFAULT_MODE) ; // token we want parser to see
TEXT : . -> more ; // collect more text for string
```

Popping the bottom layer of a mode stack will result in an exception. Switching modes with 'mode' changes the current stack top. More than one 'more' is the same as just one and the position does not matter.

type()

From reference/SetType.g4

```
lexer grammar SetType;

tokens { STRING }

DOUBLE : '"' .*? '"' -> type(STRING) ;
SINGLE : "'" .*? "'" -> type(STRING) ;
WS : [ \t\n]+ -> skip ;
```

For multiple 'type()' commands, only the rightmost has an effect.

channel()

From reference/ANTLRv4Lexer.g4

```
BLOCK_COMMENT
: '/' .*? '/' -> channel(HIDDEN)
;

LINE_COMMENT
: '/' ~[\r\n]* -> channel(HIDDEN)
;

...

// -----
// Whitespace
//
// Characters and character constructs that are of no import
// to the parser and are used to make the grammar easier to read
// for humans.
//
WS
: [ \t\r\n\f ]+
-> channel(HIDDEN)
;
```

As of 4.5, you can also define channel names like you enumerations with the following construct above the lexer rules:

```
channels { WSCHANNEL, MYHIDDEN }
```

Wildcard Operator and Nongreedy Subrules

[ANTLR 4 Documentation Home](#)

- [Nongreedy Lexer Subrules](#)
- [Nongreedy Parser Subrules](#)

EBNF subrules like $(\dots)?$, $(\dots)^*$ and $(\dots)^+$ are greedy—They consume as much input as possible, but sometimes that's not what's needed. Constructs like \dots^* consume until the end of the input in the lexer and sometimes in the parser. We want that loop to be nongreedy so we need to use different syntax: $\dots^*?$ borrowed from regular expression notation. We can make any subrule that has a $?$, $*$, or $+$ suffix nongreedy by adding another $?$ suffix. Such nongreedy subrules are allowed in both the parser and the lexer, but they are used much more frequently in the lexer.

Nongreedy Lexer Subrules

Here's the very common C-style comment lexer rule that consumes any characters until it sees the trailing `*/`:

```
COMMENT : '/' .*? '/' -> skip ; // .*? matches anything until the first */
```

Here's another example that matches strings that allow `\` as an escaped quote character:

[reference/Nongreedy.g4](#)

```
grammar Nongreedy;

s : STRING+ ;

STRING : '"' ( '\\' | . ) *? '"' ; // match "foo", "\"", "x\"y", ...

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

```
=> $ antlr4 Nongreedy.g4
=> $ javac Nongreedy*.java
=> $ grun Nongreedy s -tokens
=> "quote:\"
=> E_O_F
```

<=	[@0,0:9='\"quote:\",<1>,1:0]
	[@1,11:10='<EOF>',<-1>,2:0]

Nongreedy subrules should be used sparingly because they complicate the recognition problem and sometimes make it tricky to decipher how the lexer will match text. Here is how the lexer chooses token rules:

- The primary goal is to match the lexer rule that recognizes the most input characters.

	INT : [0-9]+ ;
	DOT : '.' ; // match period
	FLOAT : [0-9]+ '.' ; // match FLOAT upon '34.' not INT then DOT

- If more than one lexer rule matches the same input sequence, the priority goes to the rule occurring first in the grammar file.

	DOC : '/*' .*? '*/' ; // both rules match /* foo */, resolve to DOC
	CMT : '/*' .*? '*/' ;

- Nongreedy subrules match the fewest number of characters that still allows the surrounding lexical rule to match.

	/** Match anything except \n inside of double angle brackets */
	STRING : '<<' ~'\n'*? '>>' ; // Input '<<foo>>>' matches STRING then END
	END : '>>' ;

- After crossing through a nongreedy subrule within a lexical rule, all decision-making from then on is "first match wins."

For example, alternative 'ab' in rule right-hand side .*? ('a' | 'ab') is dead code and can never be matched. If the input is ab, the first alternative, 'a', matches the first character and therefore succeeds. ('a' | 'ab') by itself on the right-hand side of a rule properly matches the second alternative for input ab. This quirk arises from a nongreedy design decision that's too complicated to go into here.

To illustrate the different ways to use loops within lexer rules, consider the following grammar, which has three different action-like tokens (using different delimiters so that they all fit within one example grammar).

reference/Actions.g4	
	ACTION1 : '{' (STRING .) *? '}' ; // Allows {"foo}
	ACTION2 : '[' (STRING ~'"') *? ']' ; // Doesn't allow ["foo]; nongreedy *?
	ACTION3 : '<' (STRING ~"[>]") * '>' ; // Doesn't allow <"foo>; greedy *
	STRING : '"' ('\\' .) *? ' ' ;

Rule ACTION1 allows unterminated strings, such as {"foo}, because input "foo matches to the wildcard part of the loop. It doesn't have to go into rule STRING to match a quote. To fix that, rule ACTION2 uses ~'"' to match any character but the quote. Expression ~'"' is still ambiguous with the ']' that ends the rule, but the fact that the subrule is nongreedy means that the lexer will exit the loop upon a right square bracket. To avoid a nongreedy subrule, make the alternatives explicit. Expression ~"[>]" matches anything but the quote and right angle bracket. Here's a sample run:

=>	\$ antlr4 Actions.g4
=>	\$ javac Actions*.java
=>	\$ grun Actions tokens -tokens
=>	{"foo}
=>	E _{O_F}
<=	[@0,0:5='{ "foo}',<1>,1:0]
	[@1,7:6='<EOF>',<-1>,2:0]
=>	\$ grun Actions tokens -tokens

=>	<code>["foo]</code>
=>	<code>E_{O_F}</code>
<=	line 1:0 token recognition error at: <code>["foo]</code>
	<code>,</code>
	<code>[@0,7:6='<EOF>',<-1>,2:0]</code>
=>	<code>\$ grun Actions tokens -tokens</code>
=>	<code><"foo></code>
=>	<code>E_{O_F}</code>
<=	line 1:0 token recognition error at: <code><"foo></code>
	<code>,</code>
	<code>[@0,7:6='<EOF>',<-1>,2:0]</code>

Nongreedy Parser Subrules

Nongreedy subrules and wildcard are also useful within parsers to do “fuzzy parsing” where the goal is to extract information from an input file without having to specify the full grammar. In contrast to nongreedy lexer decision-making, parsers always make globally correct decisions. A parser never makes a decision that will ultimately cause valid input to fail later on during the parse. Here is the central idea: Nongreedy parser subrules match the shortest sequence of tokens that preserves a successful parse for a valid input sentence.

For example, here are the key rules that demonstrate how to pull integer constants out of an arbitrary Java file:

reference/FuzzyJava.g4	
grammar	FuzzyJava;
<i>/** Match anything in between constant rule matches */</i>	
file : .*? (constant .*?)+ ;	
<i>/** Faster alternate version (Gets an ANTLR tool warning about</i>	
<i>* a subrule like .* in parser that you can ignore.)</i>	
<i>*/</i>	
altfile : (constant .)* ; <i>// match a constant or any token, 0-or-more times</i>	
<i>/** Match things like "public static final SIZE" followed by anything */</i>	
constant	
: 'public' 'static' 'final' 'int' Identifier	
{System.out.println("constant: "+\$Identifier.text);}	
;	
Identifier : [a-zA-Z_\$] [a-zA-Z_\$0-9]* ; <i>// simplified</i>	

The grammar contains a greatly simplified set of lexer rules from a real Java lexer; the whole file about 60 lines. The recognizer still needs to handle string and character constants as well as comments so it doesn’t get out of sync, trying to match a constant inside of the string for example. The only unusual lexer rule performs “match any character not matched by another lexer rule” functionality:

reference/FuzzyJava.g4
OTHER : . -> skip ;

This catchall lexer rule and the `. * ?` subrule in the parser are the critical ingredients for fuzzy parsing.

Here's a sample file that we can run into the fuzzy parser:

reference/C.java
<code>import java.util.*;</code>
<code>public class C {</code>
<code>public static final int A = 1;</code>
<code>public static final int B = 1;</code>
<code>public void foo() { }</code>
<code>public static final int C = 1;</code>
<code>}</code>

And here's the build and test sequence:

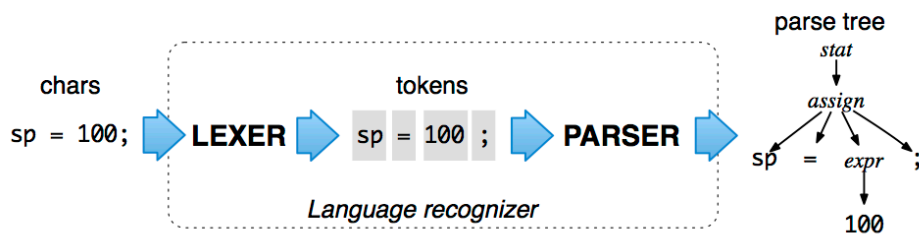
<code>\$ antlr4 FuzzyJava.g4</code>
<code>\$ javac FuzzyJava*.java</code>
<code>\$ grun FuzzyJava file C.java</code>
constant: A
constant: B
constant: C

Notice that it totally ignores everything except for the `public static final int` declarations. This all happens with only two parser rules.

Parse Tree Listeners

[Partially taken from publically visible [excerpt from ANTLR 4 book](#)]

By default, ANTLR-generated parsers build a data structure called a parse tree or syntax tree that records how the parser recognized the structure of the input sentence and component phrases.



The interior nodes of the parse tree are phrase names that group and identify their children. The root node is the most abstract phrase name, in this case *stat* (short for *statement*). The leaves of a parse tree are always the input tokens. Parse trees sit between a language recognizer and an interpreter or translator implementation. They are extremely effective data structures because they contain all of the input and complete knowledge of how the parser grouped the symbols into phrases. Better yet, they are easy to understand and the parser generates them automatically (unless you turn them off with `parser.setBuildParseTree(false)`).

Because we specify phrase structure with a set of rules, parse tree subtree roots correspond to grammar rule names. ANTLR has

a `ParseTreeWalker` that knows how to walk these parse trees and trigger events in listener implementation objects that you can create. ANTLR the tool generates listener interfaces for you also unless you, unless you turn that off with a commandline option. You can also have it generate visitors. For example from a `Java.g4` grammar, ANTLR generates:

```
public interface JavaListener extends ParseTreeListener<Token> {
    void enterClassDeclaration(JavaParser.ClassDeclarationContext ctx);
    void exitClassDeclaration(JavaParser.ClassDeclarationContext ctx);
    void enterMethodDeclaration(JavaParser.MethodDeclarationContext ctx);
    ...
}
```

where there is an enter and exit method for each rule in the parser grammar. ANTLR also generates a base listener with the fall empty implementations of all listener interface methods, in this case called `JavaBaseListener`. You can build your listener by subclassing this base and overriding the methods of interest.

Assuming you've created a listener object called `MyListener`, here is how to call the Java parser and walk the parse tree:

```
JavaLexer lexer = new JavaLexer(input);
CommonTokenStream tokens = new CommonTokenStream(lexer);
JavaParser parser = new JavaParser(tokens);
ParserRuleContext<Token> tree = parser.compilationUnit(); // parse

ParseTreeWalker walker = new ParseTreeWalker(); // create standard walker
MyListener extractor = new MyListener(parser);
walker.walk(extractor, tree); // initiate walk of tree with listener
```

Listeners and visitors are great because they keep application-specific code out of grammars, making grammars easier to read and preventing them from getting entangled with a particular application.

See the book for more information on listeners and to learn how to use visitors. (The biggest difference between the listener and visitor mechanisms is that listener methods are called independently by an ANTLR-provided walker object, whereas visitor methods must walk their children with explicit visit calls. Forgetting to invoke visitor methods on a node's children, means those subtrees don't get visited.)

Parse Tree Matching and XPath

Since ANTLR 4.2

ANTLR 4 introduced a visitor and listener mechanism that lets you implement DOM visiting or SAX-analogous event processing of tree nodes. This works great. For example, if all you care about is looking at Java method declarations, grab the `Java.g4` file and then override `methodDeclaration` in `JavaBaseListener`. From there, a `ParseTreeWalker` can trigger calls to your overridden method as it walks the tree. Easy things are easy.

This mechanism works more or less on a node-level basis. In other words, for every method declaration subtree root, your `methodDeclaration()` would get called. There are many situations where we care more about subtrees not just nodes. We might want to:

- Collect method declarations within a particular context (i.e., nested within another method) or methods with specific structure or specific types (e.g., "`void <ID>() { }`"). We'll combine XPath and tree pattern matching for this.
- Group translation operations by patterns in the tree rather than spreading operations across listener event methods.
- Get a list of all assignments anywhere in the tree. It's much easier to say "go find me all "... = ...;" subtrees" rather than creating a class just to get a listener method for rule assignment and then passing the listener to the parse tree walker.

The other important idea here is that, since we're talking about parse trees not abstract syntax trees, we can use concrete patterns instead of tree syntax. For example, we can say "`x = 0;`" instead of AST "`(= x 0)`" where the `';`' would probably be stripped before it went into the AST.

Parse tree patterns

To test a subtree to see if it has a particular structure, we use a tree pattern. We also often want to extract descendents from the subtree based upon the structure. A very simple example is checking to see if a subtree matches an assignment statement. The pattern might look like the following in your language:


```
<ID> = <expr>;
```

where ``tags" in angle brackets represent either token or rule references in the associated grammar. ANTLR converts that string to a parse tree with special nodes that represent any token ID and rule expr subtree. To create this parse tree, the pattern matching compiler needs to know which rule in the grammar the pattern conforms to. In this case it might be statement. Here is how we could test a tree, t, to see if it matches that pattern:

```
ParseTree t = ...; // assume t is a statement
ParseTreePattern p = parser.compileParseTreePattern("<ID> = <expr>;",
MyParser.RULE_statement);
ParseTreeMatch m = p.match(t);
if ( m.succeeded() ) {...}
```

We can also test for specific expressions or token values. For example, the following checks to see if t is an expression consisting of an identifier added to 0:

```
ParseTree t = ...; // assume t is an expression
ParseTreePattern p = parser.compileParseTreePattern("<ID>+0", MyParser.RULE_expr);
ParseTreeMatch m = p.match(t);
```

We can also ask the ParseTreeMatch result to pull out the token matched to the <ID> tag:

```
String id = m.get("ID");
```

You can change the tag delimiters using a method on the pattern matcher:

```
ParseTreePatternMatcher m = new ParseTreePatternMatcher();
m.setDelimiters("<<", ">>", "$"); // $ is the escape character
```

This would allow pattern "<<ID>> = <<expr>> ;\$<< ick \$>>" to be interpreted as elements: ID, ' = ', expr, and ' ;<< ick >> '.

```
String xpath = "//blockStatement/*";
String treePattern = "int <Identifier> = <expression>;";
ParseTreePattern p =
parser.compileParseTreePattern(treePattern,
JavaParser.RULE_localVariableDeclarationStatement);
List<ParseTreeMatch> matches = p.findAll(tree, xpath);
```

Pattern labels

The tree pattern matcher tracks the nodes in the tree at matches against the tags in a tree pattern. That way we can use the get() and getAll() methods to retrieve components of the matched subtree. For example, for pattern <ID>, get(" ID") returns the node matched for that ID. If more than one node matched the specified token or rule tag, only the first match is returned. If there is no node associated with the label, this returns null.

You can also label the tags with identifiers. If the label is the name of a parser rule or token in the grammar, the resulting list from getAll() (or node from get()) will contain both the parse trees matching rule or tags explicitly labeled with the label and the complete set of parse trees matching the labeled and unlabeled tags in the pattern for the parser rule or token. For example, if label is "foo", the result will contain *all* of the following.

- Parse tree nodes matching tags of the form <foo:anyRuleName> and <foo:AnyTokenName>.
- Parse tree nodes matching tags of the form <anyLabel:foo>.

- Parse tree nodes matching tags of the form `<foo>`.

Creating parse trees with the pattern matcher

You can use the parse tree pattern compiler to create parse trees for partial input fragments. Just use method `ParseTreePattern.getPatternTree()`.

See [TestParseTreeMatch.java](#).

Using XPath to identify parse tree node sets

XPath paths are strings representing nodes or subtrees you would like to select within a parse tree. It's useful to collect subsets of the parse tree to process. For example you might want to know where all assignments are in a method or all variable declarations that are initialized.

A path is a series of node names with the following separators.

Expression	Description
<i>nodename</i>	Nodes with the token or rule name <i>nodename</i>
/	The root node but <code>/X</code> is the same as <code>X</code> since the tree you pass to xpath is assumed to be the root. Because it looks better, start all of your patterns with <code>/</code> (or <code>//</code> below).
//	All nodes in the tree that match the next element in the path. E.g., <code>//ID</code> finds all ID token nodes in the tree.
!	Any node except for the next element in the path. E.g., <code>/classdef!field</code> should find all children of classdef root node that are not field subtrees.

Examples:

```
/prog/func, -> all funcs under prog at root
/prog/*, -> all children of prog at root
/*/func, -> all func kids of any root node
prog, -> prog must be root node
/prog, -> prog must be root node
/*, -> any root
*, -> any root
//ID, -> any ID in tree
//expr/primary/ID, -> any ID child of a primary under any expr
//body//ID, -> any ID under a body
//'return', -> any 'return' literal in tree
//primary/*, -> all kids of any primary
//func/*/stat, -> all stat nodes grandkids of any func node
/prog/func/'def', -> all def literal kids of func kid of prog
//stat/';', -> all ';' under any stat node
//expr/primary/!ID, -> anything but ID under primary under any expr node
//expr/!primary, -> anything but primary under any expr node
/!*, -> nothing anywhere
/!*, -> nothing at root
```

Given a parse tree (tree), the typical mechanism for visiting those nodes is the following loop:

```
for (ParseTree t : XPath.findAll(tree, xpath, parser) ) {
    ... process t ...
}
```

E.g., here is a general formula for making a list of the text associated with every node identified by a path specification:

```

List<String> nodes = new ArrayList<String>();
for (ParseTree t : XPath.findAll(tree, xpath, parser) ) {
    if ( t instanceof RuleContext) {
        RuleContext r = (RuleContext)t;
        nodes.add(parser.getRuleNames()[r.getRuleIndex()]);    }
    else {
        TerminalNode token = (TerminalNode)t;
        nodes.add(token.getText());
    }
}

```

Combining XPath and tree pattern matching

Naturally you can combine the use of XPath to find a set of root nodes and then use tree pattern matching to identify a certain subset of those and extract component nodes.

```

// assume we are parsing Java
ParserRuleContext tree = parser.compilationUnit();
String xpath = "//blockStatement/*"; // get children of blockStatement
String treePattern = "int <Identifier> = <expression>";
ParseTreePattern p =
    parser.compileParseTreePattern(treePattern,
        ExprParser.RULE_localVariableDeclarationStatement);
List<ParseTreeMatch> matches = p.findAll(tree, xpath);
System.out.println(matches);

```

See [TestXPath.java](#).

Semantic Predicates

[ANTLR 4 Documentation Home](#)

- [Making Predicated Parsing Decisions](#)
- [Finding Visible Predicates](#)
- [Using Context-Dependent Predicates](#)
- [Predicates in Lexer Rules](#)

Semantic predicates, `{...}?`, are Boolean expressions written in the target language that indicate the validity of continuing the parse along the path “guarded” by the predicate. Predicates can appear anywhere within a parser rule just like actions can, but only those appearing on the left edge of alternatives can affect prediction (choosing between alternatives). We discussed predicates in detail in Chapter 11, [Altering the Parse with Semantic Predicates](#). This section provides all of the fine print regarding the use of semantic predicates in parser and lexer rules. Let’s start out by digging deeper into how the parser incorporates predicates into parsing decisions.

Making Predicated Parsing Decisions

ANTLR’s general decision-making strategy is to find all viable alternatives and then ignore the alternatives guarded with predicates that currently evaluate to false. (A viable alternative is one that matches the current input.) If more than one viable alternative remains, the parser chooses the alternative specified first in the decision.

Consider a variant of C++ where array references also use parentheses instead of square brackets. If we only predicate one of the alternatives, we still have an ambiguous decision in `expr`:

	<code>expr: ID '(' expr ')' // array reference (ANTLR picks this one)</code>
	<code> {istype()}? ID '(' expr ')' // ctor-style typecast</code>
	<code> ID '(' expr ')' // function call</code>
	<code>;</code>

In this case, all three alternatives are viable for input `x(i)`. When `x` is not a type name, the predicate evaluates to false, leaving only the first and third alternatives as possible matches for `expr`. ANTLR automatically chooses the first alternative matching the array reference to resolve the ambiguity. Leaving ANTLR with more than one viable alternative because of too few predicates is probably not a good idea. It's best to cover `n` viable alternatives with at least `n-1` predicates. In other words, don't build rules like `expr` with too few predicates.

Sometimes, the parser finds multiple visible predicates associated with a single choice. No worries. ANTLR just combines the predicates with appropriate logical operators to conjure up a single meta-predicate on-the-fly.

For example, the decision in rule `stat` joins the predicates from both alternatives of `expr` with the `||` operator to guard the second `stat` alternative:

<code>stat: decl expr ;</code>
<code>decl: ID ID ;</code>
<code>expr: {istype()}? ID '(' expr ')' // ctor-style typecast</code>
<code> {isfunc()}? ID '(' expr ')' // function call</code>
<code>;</code>

The parser will only predict an `expr` from `stat` when `istype() || isfunc()` evaluates to true. This makes sense because the parser should only choose to match an expression if the upcoming `ID` is a type name or function name. It wouldn't make sense to just test one of the predicates in this case. Note that, when the parser gets to `expr` itself, the parsing decision tests the predicates individually, one for each alternative.

If multiple predicates occur in a sequence, the parser joins them with the `&&` operator. For example, consider changing `stat` to include a predicate before the call to `expr`:

<code>stat: decl {java5}? expr ;</code>

Now, the parser would only predict the second alternative if `java5&&(istype() || isfunc())` evaluated to true.

Turning to the code inside the predicates themselves now, keep in mind the following guidelines.

Even when the parser isn't making decisions, predicates can deactivate alternatives, causing rules to fail. This happens when a rule only has a single alternative. There is no choice to make, but ANTLR evaluates the predicate as part of the normal parsing process, just like it does for actions. That means that the following rule always fails to match.

<code>prog: {false}? 'return' INT ; // throws FailedPredicateException</code>

ANTLR converts `{false}? in the grammar to a conditional in the generated parser:`

<code>if (!false) throw new FailedPredicateException(...);</code>

So far, all of the predicates we've seen have been visible and available to the prediction process, but that's not always the case.

Finding Visible Predicates

The parser will not evaluate predicates during prediction that occur after an action or token reference. Let's think about the relationship between actions and predicates first.

ANTLR has no idea what's inside the raw code of an action and so it must assume any predicate could depend on side effects of that action. Imagine an action that computed value `x` and a predicate that tested `x`. Evaluating that predicate before the action executed to create `x` would violate the implied order of operations within the grammar.

More importantly, the parser can't execute actions until it has decided which alternative to match. That's because actions have side effects and we can't undo things like print statements. For example, in the following rule, the parser can't execute the action in front of the `{java5}? predicate` before committing to that alternative.

<code>@members {boolean allowgoto=false;}</code>
<code>stat: {System.out.println("goto"); allowgoto=true;} {java5}? 'goto' ID ';' ;</code>
<code> ...</code>
<code>;</code>

If we can't execute the action during prediction, we shouldn't evaluate the `{java5}? predicate` because it depends on that action.

The prediction process also can't see through token references. Token references have the side effect of advancing the input one symbol. A

predicate that tested the current input symbol would find itself out of sync if the parser shifted it over the token reference. For example, in the following grammar, the predicates expect `getCurrentToken` to return an `ID` token.

<code>stat: '{' decl '}'</code>
<code> '{' stat '}'</code>
<code>;</code>
<code>decl: {istype(getCurrentToken().getText())? ID ID ';' ;</code>
<code>expr: {isvar(getCurrentToken().getText())? ID ;</code>

The decision in `stat` can't test those predicates because, at the start of `stat`, the current token is a left curly. To preserve the semantics, ANTLR won't test the predicates in that decision.

Visible predicates are those that prediction encounters before encountering an action or token. The prediction process ignores nonvisible predicates, treating them as if they don't exist.

In rare cases, the parser won't be able to use a predicate, even if it's visible to a particular decision. That brings us to our next fine print topic.

Using Context-Dependent Predicates

A predicate that depends on a parameter or local variable of the surrounding rule, is considered a context-dependent predicate. Clearly, we can only evaluate such predicates within the rules in which they're defined. For example, it makes no sense for the decision in `prog` below to test context-dependent predicate `{ $i <= 5 }?`. That `$i` local variable is not even defined in `prog`.

<code>prog: vec5</code>
<code> ...</code>
<code>;</code>
<code>vec5</code>
<code>locals [int i=1]</code>
<code>: ({ \$i <= 5 }? INT { \$i ++; })* // match 5 INTs</code>
<code>;</code>

ANTLR ignores context-dependent predicates that it can't evaluate in the proper context. Normally the proper context is simply the rule defining the predicate, but sometimes the parser can't even evaluate a context-dependent predicate from within the same rule! Detecting these cases is done on-the-fly at runtime during adaptive LL(*) prediction.

For example, prediction for the optional branch of the `else` subrule in `stat` below "falls off" the end of `stat` and continues looking for symbols in the invoking `prog` rule.

<code>prog: stat+ ; // stat can follow stat</code>
<code>stat</code>
<code>locals [int i=0]</code>
<code>: { \$i == 0 }? 'if' expr 'then' stat { \$i = 5; } ('else' stat)?</code>
<code> 'break' ';' ;</code>

The prediction process is trying to figure out what can follow an `if` statement other than an `else` clause. Since the input can have multiple `stats` in a row, the prediction for the optional branch of the `else` subrule reenters `stat`. This time, of course, it gets a new copy of `$i` with a value of 0, not 5. ANTLR ignores context-dependent predicate `{ $i == 0 }?` because it knows that the parser isn't in the original `stat` call. The predicate would test a different version of `$i` so the parser can't evaluate it.

The fine print for predicates in the lexer more or less follow these same guidelines, except of course lexer rules can't have parameters and local variables. Let's look at all of the lexer-specific guidelines in the next section.

Predicates in Lexer Rules

In parser rules, predicates must appear on the left edge of alternatives to aid in alternative prediction. Lexers, on the other hand, prefer predicates on the right edge of lexer rules because they choose rules after seeing a token's entire text. Predicates in lexer rules can technically be anywhere within the rule. Some positions might be more or less efficient than others; ANTLR makes no guarantees about the optimal spot. A predicate in a lexer rule might be executed multiple times even during a single token match. You can embed multiple predicates per lexer rule and they are evaluated as the lexer reaches them during matching.

Loosely speaking, the lexer's goal is to choose the rule that matches the most input characters. At each character, the lexer decides which rules are still viable. Eventually, only a single rule will be still viable. At that point, the lexer creates a token object according to the rule's token type and matched text.

Sometimes the lexer is faced with more than a single viable matching rule. For example, input `enum` would match an `ENUM` rule and an `ID` rule. If the next character after `enum` is a space, neither rule can continue. The lexer resolves the ambiguity by choosing the viable rule specified first in the grammar. That's why we have to place keyword rules before an identifier rule like this:

	<code>ENUM : 'enum' ;</code>
	<code>ID : [a-z]+ ;</code>

If, on the other hand, the next character after input `enum` is a letter, then only `ID` is viable.

Predicates come into play by pruning the set of viable lexer rules. When the lexer encounters a false predicate, it deactivates that rule just like parsers deactivate alternatives with false predicates.

Like parser predicates, lexer predicates can't depend on side effects from lexer actions. That's because actions can only execute after the lexer positively identifies the rule to match. Since predicates are part of the rule selection process, they can't rely on action side effects. Lexer actions must appear after predicates in lexer rules. As an example, here's another way to match `enum` as a keyword in the lexer:

reference/Enum3.g4	
	<code>ENUM: [a-z]+ {getText().equals("enum")}?</code>
	<code>{System.out.println("enum!");}</code>
	<code>;</code>
	<code>ID : [a-z]+ {System.out.println("ID "+getText());} ;</code>

The print action in `ENUM` appears last and executes only if the current input matches `[a-z]+` and the predicate is true. Let's build and test `Enum3` to see if it distinguishes between `enum` and an identifier:

=>	<code>\$ antlr4 Enum3.g4</code>
=>	<code>\$ javac Enum3.java</code>
=>	<code>\$ grun Enum3 tokens</code>
=>	<code>enum abc</code>
=>	<code>E_O_F</code>
<=	<code>enum!</code>
	<code>ID abc</code>

That works great, but it's really just for instructional purposes. It's easier to understand and more efficient to match `enum` keywords with a simple rule like this:

	<code>ENUM : 'enum' ;</code>
--	------------------------------

Options

[ANTLR 4 Documentation Home](#)

- [Grammar Options](#)
- [Rule Options](#)
- [Rule Element Options](#)

There are a number of options that you can specify at the grammar and rule element level. (There are currently no rule options.) These change how ANTLR generates code from your grammar. The general syntax is:

options { name1=value1; ... nameN=valueN; } // <i>ANTLR not target language syntax</i>

where a value can be an identifier, a qualified identifier (for example, a.b.c), a string, a multi-line string in curly braces { ... }, and an integer.

Grammar Options

All grammars can use the following options. In combined grammars, all options except `language` pertain only to the generated parser. Options may be set either within the grammar file using the options syntax (described above) or when invoking ANTLR on the command line, using the `-D` option. (see Section 15.9, [ANTLR Tool Command Line Options](#).) The following examples demonstrate both mechanisms; note that `-D` overrides options within the grammar.

Rule Options

There are currently no valid rule-level options, but the tool still supports the following syntax for future use:

rulename
options { ... }
: ...
;

Rule Element Options

Token options have the form `T < name=value >` as we saw in Section 5.4, [Dealing with Precedence, Left Recursion, and Associativity](#). The only token option is `assoc` and it accepts values `left` and `right`. Here's a sample grammar with a left-recursive expression rule that specifies a token option on the `^` exponent operator token:

reference/ExprLR.g4
grammar ExprLR;
expr : expr '^' <assoc=right> expr
expr '*' expr // match subexpressions joined with '*' operator
expr '+' expr // match subexpressions joined with '+' operator
INT // matches simple integer atom
;
INT : '0'..'9'+ ;
WS : [\n]+ -> skip ;

Semantic predicates also accept an option, per [Catching failed semantic predicates](#). The only valid option is the `fail` option, which takes either a string literal in double-quotes or an action that evaluates to a string. The string literal or string result from the action should be the message to emit upon predicate failure.

errors/VecMsg.g4
ints[int max]
locals [int i=1]
: INT (',' { \$i++; } { \$i <= \$max } ? <fail={ "exceeded max " + \$max } > INT) *
;

The action can execute a function as well as compute a string when a predicate fails: `{ ... } ? <fail={ doSomethingAndReturnAString() } >`

ANTLR Tool Command Line Options

ANTLR 4 Documentation Home

If you invoke the ANTLR tool without command line arguments, you'll get a help message:

\$ antlr4
ANTLR Parser Generator Version 4.0b3
-o ____ specify output directory where all output is generated
-lib ____ specify location of grammars, tokens files
-atn generate rule augmented transition network diagrams
-encoding ____ specify grammar file encoding; e.g., euc-jp
-message-format ____ specify output style for messages in antlr, gnu, vs2005
-long-messages show exception details when available for errors and warnings
-listener generate parse tree listener (default)
-no-listener don't generate parse tree listener
-visitor generate parse tree visitor
-no-visitor don't generate parse tree visitor (default)
-package ____ specify a package/namespace for the generated code
-depend generate file dependencies
-D<option>=value set/override a grammar-level option
-Werror treat warnings as errors
-Xsave-lexer save temp lexer file created for combined grammars
-XdbgST launch StringTemplate visualizer on generated code
-Xforce-atn use the ATN simulator for all predictions
-Xlog dump lots of logging info to antlr-timestamp.log
-XdbgSTWait wait for STViz to close before continuing

Here are more details on the options:

-o outdir

ANTLR generates output files in the current directory by default. This option specifies the output directory where ANTLR should generate parsers, listeners, visitors, and tokens files.

\$ antlr4 -o /tmp T.g4
\$ ls /tmp/T*
/tmp/T.tokens /tmp/TListener.java
/tmp/TBaseListener.java /tmp/TParser.java

-lib libdir

When looking for tokens files and imported grammars, ANTLR normally looks in the current directory. This option specifies which directory to look in instead. It is only used for resolving grammar references for the import statement and the tokenVocab option. The path to the primary grammar must always be fully specified.

\$ cat /tmp/B.g4

parser grammar B;
x : ID ;
\$ cat A.g4
grammar A;
import B;
s : x ;
ID : [a-z]+ ;
\$ antlr4 -lib /tmp A.g4

-atn

Generate DOT graph files that represent the internal ATN (augmented transition network) data structures that ANTLR uses to represent grammars. The files come out as Grammar.rule .dot. If the grammar is a combined grammar, the lexer rules are named Grammar.Lexer.rule .dot.

\$ cat A.g4
grammar A;
s : b ;
b : ID ;
ID : [a-z]+ ;
\$ antlr4 -atn A.g4
\$ ls *.dot
A.b.dot A.s.dot A.Lexer.ID.dot

-encoding encodingname

By default ANTLR loads grammar files using the UTF-8 encoding, which is a very common character file encoding that degenerates to ASCII for characters that fit in one byte. There are many character file encodings from around the world. If that grammar file is not the default encoding for your locale, you need this option so that ANTLR can properly interpret grammar files. This does not affect the input to the generated parsers, just the encoding of the grammars themselves.

```
# my locale is en_US on a Mac OS X box
# I saved this file with a UTF-8 encoding to handle grammar name 外 (uCDE2)
# inside the grammar file
$ cat 外.g4
grammar 外;
a: 'foreign';
$ antlr4 -encoding UTF-8 外.g4
$ ls 外*.java
外BaseListener.java 外Listener.java
外Lexer.java 外Parser.java
$ javac -encoding UTF-8 外*.java
```

-message-format format

ANTLR generates warning and error messages using templates from directory tool/resources/org/antlr/v4/tool/templates/messages/formats. By default, ANTLR uses the antlr.stg (StringTemplate group) file. You can change this to gnu or vs2005 to have ANTLR generate messages appropriate for Emacs or Visual Studio. To make your own called X, create resource org/antlr/v4/tool/templates/messages/formats/ X and place it in the CLASSPATH.

-listener

This option tells ANTLR to generate a parse tree listener and is the default.

-no-listener

This option tells ANTLR not to generate a parse tree listener.

-visitor

ANTLR does not generate parse tree visitors by default. This option turns that feature on. ANTLR can generate both parse tree listeners and visitors; this option and `-listener` aren't mutually exclusive.

-no-visitor

Tell ANTLR not to generate a parse tree visitor; this is the default.

-package

Use this option to specify a package or namespace for ANTLR-generated files. Alternatively, you can add a `@header { . . . }` action but that ties the grammar to a specific language. If you use this option and `@header`, make sure that the header action does not contain a package specification otherwise the generated code will have two of them.

-depend

Instead of generating a parser and/or lexer, generate a list of file dependencies, one per line. The output shows what each grammar depends on and what it generates. This is useful for build tools that need to know ANTLR grammar dependencies. Here's an example:

\$ antlr4 -depend T.g
T.g: A.tokens
TParser.java : T.g
T.tokens : T.g
TLexer.java : T.g
TListener.java : T.g
TBaseListener.java : T.g

If you use `-lib libdir` with `-depend` and grammar option `tokenVocab=A`, then the dependencies include the library path as well: `T.g: libdir/A.tokens`. The output is also sensitive to the `-o outdir` option: `outdir/TParser.java : T.g`.

-D<option>=value

Use this option to override or set a grammar-level option in the specified grammar or grammars. This option is useful for generating parsers in different languages without altering the grammar itself. (I expect to have other targets in the near future.)

\$ antlr4 -Dlanguage=Java T.g4 # default
\$ antlr4 -Dlanguage=C T.g4
error(31): ANTLR cannot generate C code as of version 4.0b3

-Werror

As part of a large build, ANTLR warning messages could go unnoticed. Turn on this option to have warnings treated as errors, causing the ANTLR tool to report failure back to the invoking commandline shell.

There are also some extended options that are useful mainly for debugging ANTLR itself:

-Xsave-lexer

ANTLR generates both a parser and a lexer from a combined grammar. To create the lexer, ANTLR extracts a lexer grammar from the combined grammar. Sometimes it's useful to see what that looks like if it's not clear what token rules ANTLR is creating. This does not affect the generated parsers or lexers.

-XdbgST

For those building a code generation target, this option brings up a window showing the generated code and the templates used to generate that code. It invokes the StringTemplate inspector window.

-Xforce-atn

ANTLR normally builds traditional "switch on token type" decisions where possible (one token of lookahead is sufficient to distinguish between all alternatives in a decision). To force even these simple decisions into the adaptive LL(*) mechanism, use this option.

-Xlog

This option creates a log file containing lots of information messages from ANTLR as it processes your grammar. If you would like to see how ANTLR translates your left-recursive rules, turn on this option and look in the resulting log file.

\$	<code>antlr4 -Xlog T.g4</code>
wrote	<code>./antlr-2012-09-06-17.56.19.log</code>

Runtime Libraries and Code Generation Targets

This page lists the available and upcoming ANTLR runtimes. Please note that you won't find here language specific code generators. This is because there is only one tool, written in Java, which is able to generate lexer and parser code for all targets, through command line options. The tool can be invoked from the command line, or any integration plugin to popular IDEs and build systems: Eclipse, IntelliJ, Visual Studio, Maven. So whatever your environment and target is, you should be able to run the tool and produce the code in the targeted language. As of writing, the available targets are the following:

Java

C# and also Sam Harwell's [Alternative C# target](#)

Python (2 and 3)

JavaScript

In the near future, we hope to have the following targets:

C++

C# Target

Which frameworks are supported?

The C# runtime is CLS compliant, and only requires a corresponding 2.0 .Net framework.

In practice, this has been extensively tested against:

- Microsoft .Net 2.0 framework
- Microsoft .Net 4.0 framework
- Mono .Net 2.0 framework
- Mono .Net 4.0 framework

No issue was found, so you should find that the runtime works pretty much against any recent .Net framework.

How do I get started?

You will find full instructions on the [Git web page for ANTLR C# runtime](#).

How do I use the runtime from my project?

How do I run the generated lexer and/or parser?

Let's suppose that your grammar is named, as above, "MyGrammar".

Let's suppose this parser comprises a rule named "StartRule"

The tool will have generated for you the following files:

MyGrammarLexer.cs

MyGrammarParser.cs

MyGrammarListener.cs (if you have not activated the -no-listener option)

MyGrammarBaseListener.js (if you have not activated the -no-listener option)

MyGrammarVisitor.js (if you have activated the -visitor option)

MyGrammarBaseVisitor.js (if you have activated the -visitor option)

Now a fully functioning code might look like the following:

```
using Antlr4.Runtime;

public void MyParseMethod() {
    String input = "your text to parse here";
    AntlrInputStream stream = new InputStream(input);
    ITokenSource lexer = new MyGrammarLexer(stream);
    ITokenStream tokens = new CommonTokenStream(lexer);
    MyGrammarParser parser = new MyGrammarParser(tokens);
    parser.buildParseTrees = true;
    IParseTree tree = parser.StartRule();
}
```

This program will work. But it won't be useful unless you do one of the following:

- you visit the parse tree using a custom listener
 - you visit the parse tree using a custom visitor
 - your grammar comprises production code (like AntLR3)
- (please note that production code is target specific, so you can't have multi target grammars that include production code)

How do I create and run a custom listener?

Let's suppose your MyGrammar grammar comprises 2 rules: "key" and "value".

The antlr4 tool will have generated the following listener (only partial code shown here):

```
interface IMyGrammarParserListener : IParseTreeListener {
    void EnterKey (MyGrammarParser.KeyContext context);
    void ExitKey (MyGrammarParser.KeyContext context);
    void EnterValue (MyGrammarParser.ValueContext context);
    void ExitValue (MyGrammarParser.ValueContext context);
}
```

In order to provide custom behavior, you might want to create the following class:

```
class KeyPrinter : MyGrammarBaseListener {
    // override default listener behavior
    void ExitKey (MyGrammarParser.KeyContext context) {
        Console.WriteLine("Oh, a key!");
    }
}
```

In order to execute this listener, you would simply add the following lines to the above code:

```
...
IParseTree tree = parser.StartRule() - only repeated here for reference
KeyPrinter printer = new KeyPrinter();
ParseTreeWalker.DEFAULT.walk(printer, tree);
```

Further information can be found from The Definitive ANTLR Reference book.

The C# implementation of ANTLR is as close as possible to the Java one, so you shouldn't find it difficult to adapt the examples for C#.

C++ Target

We have actually started on this target (12/15/2013) using an automated Java to C++ translator, which seems to do a very good job.

JavaScript Target

Which browsers are supported?

In theory, all browsers supporting ECMAScript 5.1.

In practice, this has been extensively tested against:

- Firefox 34.0.5
- Safari 8.0.2
- Chrome 39.0.2171
- Explorer 11.0.3

The tests were conducted using Selenium. No issue was found, so should find that the runtime works pretty much against any recent JavaScript engine.

The runtime has also been extensively tested against Node.js 0.10.33. No issue was found.

How to create a JavaScript lexer or parser?

This is pretty much the same as creating a Java lexer or parser, except you need to specify the language target, for example:

```
antlr4 -Dlanguage=JavaScript MyGrammar.g4
```

For a full list of antlr4 tool options, please visit the [tool documentation page](#).

Where can I get the runtime?

Once you've generated the lexer and/or parser code, you need to download the runtime.

The JavaScript runtime is available from [the ANTLR web site download section](#):

The runtime is provided in the form of source code, so no additional installation is required.

We will not document here how to refer to the runtime from your project, since this would differ a lot depending on your project type and IDE.

How do I get the runtime in my browser?

The runtime is quite big and is currently maintained in the form of around 50 scripts, which follow the same structure as the runtimes for other targets (Java, C#, Python...).

This structure is key in keeping code maintainable and consistent across targets.

However, it would be a bit of a problem when it comes to get it into a browser. Nobody wants to write 50 times:

```
<script src='lib/myscript.js'>
```

In order to avoid having to do this, and also to have the exact same code for browsers and Node.js, we rely on a script which provides the equivalent of the Node.js 'require' function.

This script is provided by Torben Haase, and is NOT part of ANTLR JavaScript runtime, although the runtime heavily relies on it.

So in short, assuming you have at the root of your web site, both the 'antlr4' directory and a 'lib' directory with 'require.js' inside it, all you need to put in your HTML header is the following:

```
<script src='lib/require.js'>
<script>
    var antlr4 = require('antlr4/index');
</script>
```

This will load the runtime asynchronously.

How do I get the runtime in Node.js?

Right now, there is no npm package available, so you need to register a link instead.

This can be done by running from the antlr4 directory:

```
npm link antlr4
```

This will install antlr4 using the package.json descriptor that comes with the script.

How do I run the generated lexer and/or parser?

Let's suppose that your grammar is named, as above, "MyGrammar".

Let's suppose this parser comprises a rule named "StartRule"

The tool will have generated for you the following files:

MyGrammarLexer.js

MyGrammarParser.js

MyGrammarListener.js (if you have not activated the -no-listener option)

MyGrammarVisitor.js (if you have activated the -visitor option)

(Developers used to Java/C# AntLR will notice that there is no base listener or visitor generated, this is because JavaScript having no support for interfaces, the generated listener and visitor are fully fledged classes)

Now a fully functioning script might look like the following:

```
var input = "your text to parse here"
var chars = new antlr4.InputStream(input);
var lexer = new MyGrammarLexer.MyGrammarLexer(chars);
var tokens = new antlr4.CommonTokenStream(lexer);
var parser = new MyGrammarParser.MyGrammarParser(tokens);
parser.buildParseTrees = true;
var tree = parser.StartRule();
```

This program will work. But it won't be useful unless you do one of the following:

- you visit the parse tree using a custom listener
 - you visit the parse tree using a custom visitor
 - your grammar comprises production code (like AntLR3)
- (please note that production code is target specific, so you can't have multi target grammars that include production code)

How do I create and run a custom listener?

Let's suppose your MyGrammar grammar comprises 2 rules: "key" and "value".

The antlr4 tool will have generated the following listener:

```
MyGrammarListener = function(ParseTreeListener) {  
    // some code here  
}  
  
// some code here  
MyGrammarListener.prototype.enterKey = function(ctx) {};  
MyGrammarListener.prototype.exitKey = function(ctx) {};  
MyGrammarListener.prototype.enterValue = function(ctx) {};  
MyGrammarListener.prototype.exitValue = function(ctx) {};
```

In order to provide custom behavior, you might want to create the following class:

```
KeyPrinter = function() {  
    MyGrammarListener.call(this); // inherit default listener  
    return this;  
};
```

// inherit default listener

```
KeyPrinter.prototype = Object.create(MyGrammarListener.prototype);
```

```
KeyPrinter.prototype.constructor = KeyPrinter;
```

// override default listener behavior

```
KeyPrinter.prototype.exitKey = function(ctx) {  
    console.log("Oh, a key!");  
};
```

In order to execute this listener, you would simply add the following lines to the above code:

```
...  
  
tree = parser.StartRule() - only repeated here for reference  
  
var printer = new KeyPrinter();  
  
antlr4.tree.ParseTreeWalker.DEFAULT.walk(printer, tree);
```

Further information can be found from [The definitive ANTLR 4 reference book](#).

The JavaScript implementation of ANTLR is as close as possible to the Java one, so you shouldn't find it difficult to adapt the examples for JavaScript.

Java Target

The **ANTLR v4 book** has a decent summary of the runtime library. We have added a useful **XPath feature** since the book was printed that lets you select bits of parse trees.

Runtime API

See [Getting Started with ANTLR v4](#)

Python Target

Actually there are 2 Python targets: Python2 and Python3.

This is because there is only limited compatibility between those 2 versions of the language, please refer to the [Python documentation](#) for full details.

How to create a Python lexer or parser?

This is pretty much the same as creating a Java lexer or parser, except you need to specify the language target, for example:

```
antlr4 -Dlanguage=Python2 MyGrammar.g4
```

or

```
antlr4 -Dlanguage=Python3 MyGrammar.g4
```

For a full list of antlr4 tool options, please visit the [tool documentation page](#).

Where can I get the runtime?

Once you've generated the lexer and/or parser code, you need to download the runtime.

The Python runtimes are available from PyPI:

<https://pypi.python.org/pypi/antlr4-python2-runtime/>

<https://pypi.python.org/pypi/antlr4-python3-runtime/>

The runtimes are provided in the form of source code, so no additional installation is required.

We will not document here how to refer to the runtime from your Python project, since this would differ a lot depending on your project type and IDE.

How do I run the generated lexer and/or parser?

Let's suppose that your grammar is named, as above, "MyGrammar".

Let's suppose this parser comprises a rule named "StartRule"

The tool will have generated for you the following files:

MyGrammarLexer.py

MyGrammarParser.py

MyGrammarListener.py (if you have not activated the -no-listener option)

MyGrammarVisitor.py (if you have activated the -visitor option)

(Developers used to Java/C# AntLR will notice that there is no base listener or visitor generated, this is because Python having no support for interfaces, the generated listener and visitor are fully fledged classes)

Now a fully functioning script might look like the following:

```
from antlr4 import *

from MyGrammarLexer import MyGrammarLexer
from MyGrammarParser import MyGrammarParser

def main(argv):
```



```

input = FileStream(argv[1])
lexer = MyGrammarLexer(input)
stream = CommonTokenStream(lexer)
parser = MyGrammarParser(stream)
tree = parser.StartRule()

if __name__ == '__main__':
    main(sys.argv)

```

This program will work. But it won't be useful unless you do one of the following:

- you visit the parse tree using a custom listener
 - you visit the parse tree using a custom visitor
 - your grammar comprises production code (like AntLR3)
- (please note that production code is target specific, so you can't have multi target grammars that include production code, except for very limited use cases, see below)

How do I create and run a custom listener?

Let's suppose your MyGrammar grammar comprises 2 rules: "key" and "value".

The antlr4 tool will have generated the following listener:

```

class MyGrammarListener(ParseTreeListener):
    def enterKey(self, ctx):
        pass
    def exitKey(self, ctx):
        pass
    def enterValue(self, ctx):
        pass
    def exitValue(self, ctx):
        pass

```

In order to provide custom behavior, you might want to create the following class:

```

class KeyPrinter(MyGrammarListener):
    def exitKey(self, ctx):
        print("Oh, a key!")

```

In order to execute this listener, you would simply add the following lines to the above code:

```

...
tree = parser.StartRule() - only repeated here for reference
printer = KeyPrinter()

```

```
walker = ParseTreeWalker()

walker.walk(printer, tree)
```

Further information can be found from the AntLR4 definitive guide.

The Python implementation of AntLR is as close as possible to the Java one, so you shouldn't find it difficult to adapt the examples for Python.

Target agnostic grammars

If your grammar is targeted to Python only, you may ignore the following.

But if your goal is to get your Java parser to also run in Python, then you might find it useful.

- 1) Do not embed production code inside your grammar. This is not portable and will not be. Move all your code to listeners or visitors.
- 2) The only production code absolutely required to sit with the grammar should be semantic predicates, like:

```
ID {$text.equals("test")}? 
```

Unfortunately, this is not portable, but you can work around it. The trick involves:

- deriving your parser from a parser you provide, such as `BaseParser`
- implementing utility methods in this `BaseParser`, such as `isEqualText`
- adding a `"self"` field to the Java/C# `BaseParser`, and initialize it with `"this"`

Thanks to the above, you should be able to rewrite the above semantic predicate as follows:

```
ID {$self.isEqualText($text,"test")}? 
```

Parser and lexer interpreters

Since ANTLR 4.2

For small parsing tasks it is sometimes convenient to use ANTLR in interpreted mode, rather than generating a parser in a particular target, compiling it and running it as part of your application. Here's some sample code that creates lexer and parser Grammar objects and then creates interpreters. Once we have a `ParserInterpreter`, we can use it to parse starting in any rule we like, given a rule index (which the Grammar can provide).

```
LexerGrammar lg = new LexerGrammar(
    "lexer grammar L;\n" +
    "A : 'a' ;\n" +
    "B : 'b' ;\n" +
    "C : 'c' ;\n");
Grammar g = new Grammar(
    "parser grammar T;\n" +
    "s : (A|B)* C ;\n",
    lg);
LexerInterpreter lexEngine =
    lg.createLexerInterpreter(new ANTLRInputStream(input));
CommonTokenStream tokens = new CommonTokenStream(lexEngine);
ParserInterpreter parser = g.createParserInterpreter(tokens);
ParseTree t = parser.parse(g.rules.get(startRule).index);
```

You can also load combined grammars from a file:

```

public static ParseTree parse(String fileName,
                             String combinedGrammarFileName,
                             String startRule)
    throws IOException
{
    final Grammar g = Grammar.load(combinedGrammarFileName);
    LexerInterpreter lexEngine = g.createLexerInterpreter(new
ANTLRFileStream(fileName));
    CommonTokenStream tokens = new CommonTokenStream(lexEngine);
    ParserInterpreter parser = g.createParserInterpreter(tokens);
    ParseTree t = parser.parse(g.getRule(startRule).index);
    System.out.println("parse tree: "+t.toStringTree(parser));
    return t;
}

```

Then:

```

ParseTree t = parse("T.om",
    MantraGrammar,
    "compilationUnit");

```

To load separate lexer/parser grammars, do this:

```

public static ParseTree parse(String fileNameToParse,
                             String lexerGrammarFileName,
                             String parserGrammarFileName,
                             String startRule)
    throws IOException
{
    final LexerGrammar lg = (LexerGrammar) Grammar.load(lexerGrammarFileName);
    final Grammar pg = Grammar.load(parserGrammarFileName, lg);
    ANTLRFileStream input = new ANTLRFileStream(fileNameToParse);
    LexerInterpreter lexEngine = lg.createLexerInterpreter(input);
    CommonTokenStream tokens = new CommonTokenStream(lexEngine);
    ParserInterpreter parser = pg.createParserInterpreter(tokens);
    ParseTree t = parser.parse(pg.getRule(startRule).index);
    System.out.println("parse tree: " + t.toStringTree(parser));
    return t;
}

```

Then:

```

ParseTree t = parse(fileName, XMLLexerGrammar, XMLParserGrammar, "document");

```

This is also how we will integrate instantaneous parsing into ANTLRWorks2 and development environment plug-ins.

See [TestParserInterpreter.java](#).

Integrating ANTLR into Development Systems

The Java target is the reference implementation mirrored by other targets. The following pages help you integrate ANTLR into development

environments and build systems appropriate for your target language. As of January 2015, we have Java, C#, Python 2, Python 3, and JavaScript targets. C++ is in progress.

The easiest thing is probably just to use an [ANTLR plug-in for your favorite development environment](#).

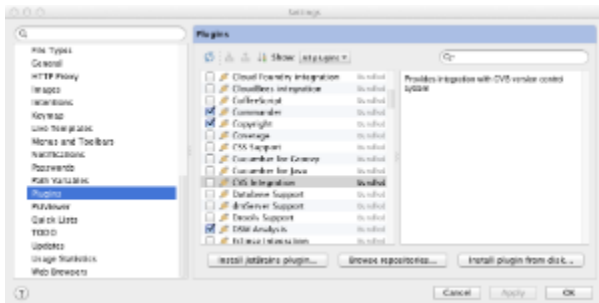
- [Java IDE Integration](#)
- [C# IDE Integration](#)
- [C++ IDE Integration](#)

Java IDE Integration

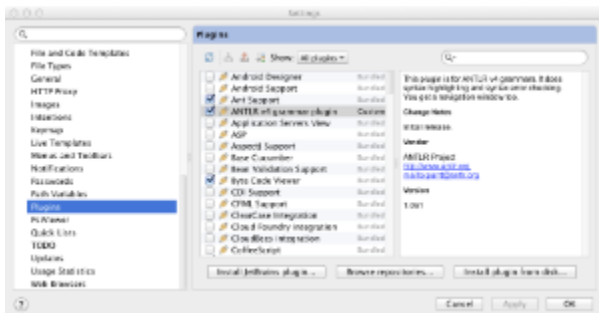
Development environments

IntelliJ

There is a very complete and useful plug-in for IntelliJ 12-14, you can grab at the [download page](#). Check the [plugin readme](#) for feature set. Just go to the preferences and click on the "Install plug-in from disk..." button from this dialog box:



Select the intellij-plugin-1.x.zip (or whatever version) file and hit okay or apply. It will ask you to restart the IDE. If you look at the plug-ins again, you will see:



Also, I have prepared a video that will help you generate grammars and so on using ANTLR v4 in IntelliJ (w/o the plugin).

Eclipse

Edgar Espina has created an [eclipse plugin for ANTLR v4](#). Features: Advanced Syntax Highlighting, Automatic Code Generation (on save), Manual Code Generation (through External Tools menu), Code Formatter (Ctrl+Shift+F), Syntax Diagrams, Advanced Rule Navigation between files (F3), Quick fixes.

NetBeans

Sam Harwell's [ANTLRWorks2](#) works also as a plug-in, not just a stand-alone tool built on top of NetBeans.

Build systems

ant

mvn

Maven Plugin Reference

The reference pages for the latest version of the Maven plugin for ANTLR 4 can be found here:

<http://www.antlr.org/api/maven-plugin/latest/index.html>

Walkthrough

This section describes how to create a simple Antlr 4 project and build it using maven. We are going to use the ArrayInit.g4 example from chapter 3 of the book, and bring it under maven. We will need to rename files and modify them. We will conclude by building a portable stand alone application.

Generate the skeleton

To generate the maven skeleton, type these commands:

```
mkdir SimpleAntlrMavenProject
cd SimpleAntlrMavenProject
mvn archetype:generate -DgroupId=org.abcd.examples -DartifactId=array-example
-Dpackage=org.abcd.examples.ArrayInit -Dversion=1.0
# Accept all the default values
cd array-example
```

Maven will ask a series of questions, simply accept the default answers by hitting enter.

Move into the directory created by maven:

```
cd array-example
```

We can use the find command to see the files created by maven:

```
$ find . -type f
./pom.xml
./src/test/java/org/abcd/examples/ArrayInit/AppTest.java
./src/main/java/org/abcd/examples/ArrayInit/App.java
```

We need to edit the pom.xml file extensively. The App.java will be renamed to ArrayInit.java and will contain the main ANTLR java program which we will download from the book examples. The AppTest.java file will be renamed ArrayInitTest.java but will remain the empty test as created by maven. We will also be adding the grammar file ArrayInit.g4 from the book examples in there.

Get the examples for the book and put them in the Downloads folder

To obtain the ArrayInit.g4 grammar from the book, simply download it:

```
pushd ~/Downloads
wget http://media.pragprog.com/titles/tpantlr2/code/tpantlr2-code.tgz
tar xvfz tpantlr2-code.tgz
popd
```

Copy the grammar to the maven project

The grammar file goes into a special folder under the src/ directory. The folder name must match the maven package name org.abcd.examples.ArrayInit.

```
mkdir -p src/main/antlr4/org/abcd/examples/ArrayInit
cp ~/Downloads/code/starter/ArrayInit.g4 src/main/antlr4/org/abcd/examples/ArrayInit
```

Copy the main program to the maven project

We replace the maven App.java file with the main java program from the book. In the book, that main program is called Test.java, we rename it to ArrayInit.java:

```
# Remove the maven file
rm ./src/main/java/org/abcd/examples/ArrayInit/App.java
# Copy and rename the example from the book
cp ~/Downloads/code/starter/Test.java
./src/main/java/org/abcd/examples/ArrayInit/ArrayInit.java
```

Spend a few minutes to read the main program. Notice that it reads the standard input stream. We need to remember this when we run the application.

Edit the ArrayInit.java file

We need to add a package declaration and to rename the class. Edit the file ./src/main/java/org/abcd/examples/ArrayInit/ArrayInit.java in your favorite editor. The head of the file should look like this when you are done:

```
package org.abcd.examples.ArrayInit;
import org.antlr.v4.runtime.*;
import org.antlr.v4.runtime.tree.*;

public class ArrayInit {
    ...
}
```

Edit the ArrayInitTest.java file

Maven creates a test file called AppTest.java, we need to rename it to match the name of our application:

```
pushd src/test/java/org/abcd/examples/ArrayInit
mv AppTest.java ArrayInitTest.java
sed 's/App/ArrayInit/g' ArrayInitTest.java >ArrayInitTest.java.tmp
mv ArrayInitTest.java.tmp ArrayInitTest.java
popd
```

Edit the pom.xml file

Now we need to extensively modify the pom.xml file. The final product looks like this:

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>org.abcd.examples</groupId>
  <artifactId>array-init</artifactId>
  <version>1.0</version>
  <packaging>jar</packaging>
  <name>array-init</name>
```

```

<url>http://maven.apache.org</url>
<properties>
  <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
</properties>
<dependencies>
  <dependency>
    <groupId>org.antlr</groupId>
    <artifactId>antlr4-runtime</artifactId>
    <version>4.1</version>
  </dependency>
  <dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>3.8.1</version>
  </dependency>
</dependencies>
<build>
  <plugins>
    <!-- This plugin sets up maven to use Java 7 -->
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-compiler-plugin</artifactId>
      <version>3.1</version>
      <configuration>
        <source>1.7</source>
        <target>1.7</target>
      </configuration>
    </plugin>
    <!-- Plugin to compile the g4 files ahead of the java files
        See
https://github.com/antlr/antlr4/blob/master/antlr4-maven-plugin/src/site/apt/examples/
        simple.apt.vm
        Except that the grammar does not need to contain the package declaration as
        stated in the documentation (I do not know why)
        To use this plugin, type:
            mvn antlr4:antlr4
        In any case, Maven will invoke this plugin before the Java source is
        compiled
    -->
    <plugin>
      <groupId>org.antlr</groupId>
      <artifactId>antlr4-maven-plugin</artifactId>
      <version>4.2</version>
      <executions>
        <execution>
          <goals>
            <goal>antlr4</goal>
          </goals>
        </execution>
      </executions>
    </plugin>
    <!-- plugin to create a self-contained portable package
        This allows us to execute our application like this:
            java -cp target/array-init-1.0-jar-with-dependencies.jar
            org.abcd.examples.ArrayInit.ArrayInit
    -->
    <plugin>
      <artifactId>maven-assembly-plugin</artifactId>
      <configuration>

```

```
    <descriptorRefs>
      <descriptorRef>jar-with-dependencies</descriptorRef>
    </descriptorRefs>
  </configuration>
  <executions>
    <execution>
      <id>simple-command</id>
      <phase>package</phase>
      <goals>
        <goal>attached</goal>
      </goals>
    </execution>
  </executions>
</plugin>
```



```
</plugins>
</build>
</project>
```

This concludes the changes we had to make. We can look at the list of files we have with the find command:

```
$ find . -type f
./pom.xml
./src/test/java/org/abcd/examples/ArrayInit/ArrayInitTest.java
./src/main/antlr4/org/abcd/examples/ArrayInit/ArrayInit.g4
./src/main/java/org/abcd/examples/ArrayInit/ArrayInit.java
```

Building a stand alone application

With all the files now in place, we can ask maven to create a standalone application. The following command does this:

```
mvn package
```

Maven creates a self-contained jar file called target/array-init-1.0-jar-with-dependencies.jar. We can execute the jar file, but remember that it expects some input on the command line, which means the command will hang on the command line until we feed it some input:

```
java -cp target/array-init-1.0-jar-with-dependencies.jar
org.abcd.examples.ArrayInit.ArrayInit
```

And let's feed it the following input:

```
{1,2,3}
^D
```

The ^D signals the end of the input to the standard input stream and gets the rest of the application going. You should see the following output:

```
(init { (value 1) , (value 2) , (value 3) })
```

You can also build a jar file without the dependencies, and execute it with a maven command instead:

```
mvn install
mvn exec:java -Dexec.mainClass=org.abcd.examples.ArrayInit.ArrayInit
{1,2,3}
^D
```

C# IDE Integration

C++ IDE Integration

ANTLR v4 FAQ

FAQ Contents

FAQ - Getting Started

- How to I install and run a simple grammar?
- Why does my parser test program hang?

FAQ - Installation

- Why can't ANTLR (grun) find my lexer or parser?
- Why can't I run the ANTLR tool?
- Why doesn't my parser compile?

FAQ - General

- Why do we need ANTLR v4?
- What is the difference between ANTLR 3 and 4?
- Why is my expression parser slow?

FAQ - Grammar syntax

FAQ - Lexical analysis

- How can I parse non-ASCII text and use characters in token rules?
- How do I replace escape characters in string tokens?
- Why are my keywords treated as identifiers?
- Why are there no whitespace tokens in the token stream?

FAQ - Parse Trees

- What if I need ASTs not parse trees for a compiler, for example?
- When do I use listener/visitor vs XPath vs Tree pattern matching?

FAQ - Translation

- ASTs vs parse trees
- Decoupling input walking from output generation

FAQ - Actions and semantic predicates

- How do I test if an optional rule was matched?

FAQ - Error handling

- How do I perform semantic checking with ANTLR?

FAQ - Getting Started

How to I install and run a simple grammar?

Why does my parser test program hang?

How to I install and run a simple grammar?

See [Getting Started with ANTLR v4](#).

Why does my parser test program hang?

Your test program is likely not hanging but simply waiting for you to type some input for standard input. Don't forget that you need to type the end of file character, generally on a line by itself, at the end of the input. On a Mac or Linux machine it is ctrl-D, as gawd intended, or ctrl-Z on a Windows machine.

See [Getting Started with ANTLR v4](#).

FAQ - Installation

Why can't ANTLR (grun) find my lexer or parser?

Why can't I run the ANTLR tool?

Why doesn't my parser compile?

Why can't ANTLR (grun) find my lexer or parser?

If you see "Can't load Hello as lexer or parser", it's because you don't have '.' (current directory) in your CLASSPATH.

```
$ alias antlr4='java -jar /usr/local/lib/antlr-4.2.2-complete.jar'
$ alias grun='java org.antlr.v4.runtime.misc.TestRig'
$ export CLASSPATH="/usr/local/lib/antlr-4.2.2-complete.jar"
$ antlr4 Hello.g4
$ javac Hello*.java
$ grun Hello r -tree
Can't load Hello as lexer or parser
$
```

For mac/linux, use:

```
export CLASSPATH=".:usr/local/lib/antlr-4.2.2-complete.jar:$CLASSPATH"
```

or for Windows:

```
SET CLASSPATH=.;C:\Javalib\antlr4-complete.jar;%CLASSPATH%
```

See the dot at the beginning?

Please read carefully: [Getting Started with ANTLR v4](#).

Why can't I run the ANTLR tool?

If you get a no class definition found error, you are missing the ANTLR jar in your CLASSPATH (or you might only have the runtime jar):

```
/tmp $ java org.antlr.v4.Tool Hello.g4
Exception in thread "main" java.lang.NoClassDefFoundError: org/antlr/v4/Tool
Caused by: java.lang.ClassNotFoundException: org.antlr.v4.Tool
    at java.net.URLClassLoader$1.run(URLClassLoader.java:202)
    at java.security.AccessController.doPrivileged(Native Method)
    at java.net.URLClassLoader.findClass(URLClassLoader.java:190)
    at java.lang.ClassLoader.loadClass(ClassLoader.java:306)
    at sun.misc.Launcher$AppClassLoader.loadClass(Launcher.java:301)
    at java.lang.ClassLoader.loadClass(ClassLoader.java:247)
```

See [Getting Started with ANTLR v4](#).

Why doesn't my parser compile?

If you see these kinds of errors, it's because you don't have the runtime or complete ANTLR library in your CLASSPATH.

```
/tmp $ javac Hello*.java
HelloBaseListener.java:3: package org.antlr.v4.runtime does not exist
import org.antlr.v4.runtime.ParserRuleContext;
                        ^
...
```

See [Getting Started with ANTLR v4](#).

FAQ - General

Why do we need ANTLR v4?

What is the difference between ANTLR 3 and 4?

Why is my expression parser slow?

Why do we need ANTLR v4?

Oliver Zeigermann asked me some questions about v4. Here is our conversation.

See the [preface](#) from the upcoming ANTLR v4 book.

Q: Why is the new version of ANTLR also called “honey badger”?

A: ANTLR v4 is called the honey badger release after the fearless hero of the YouTube sensation, [The Crazy Nastyass Honey Badger](#).

Q: Why did you create a new version of ANTLR?

Well, I start creating a new version because v3 had gotten very messy on the inside and also relied on grammars written in ANTLR v2. Unfortunately, v2's open-source license was unclear and so projects such as Eclipse could not include v3 because of its dependency on v2. In the end, Sam Harwell converted all of the v2 grammars into v3 so that v3 was written in itself. Because v3 has a very clean BSD license, the Eclipse project okayed for inclusion in that project in the summer of 2011.

As I was rewriting ANTLR, I wanted to experiment with a new variation of the LL(*) parsing algorithm. As luck would have it, I came up with a cool new version called adaptive LL(*) that pushes all of the grammar analysis effort to runtime. The parser warms up like Java does with its JIT on-the-fly compiler; the code gets faster and faster the longer it runs. The benefit is that the adaptive algorithm is much stronger than the static LL(*) grammar analysis algorithm in v3. Honey Badger takes any grammar that you give it; it just doesn't give a damn. (v4 accepts even left recursive grammars, except for indirectly left recursive grammars where x calls y which calls x).

v4 is the culmination of 25 years of research into parsers and parser generators. I think I finally know what I want to build. 😊

Q: What makes you excited about ANTLR4?

A: The biggest thing is the new adaptive parsing strategy, which lets us accept any grammar we care to write. That gives us a huge productivity boost because we can now write much more natural expression rules (which occur in almost every grammar). For example, bottom-up parser generators such as yacc let you write very natural grammars like this:

```
e : e '*' e
  | e '+' e
  | INT
  ;
```

ANTLR v4 will also take that grammar now, translating it secretly to a non-left recursive version.

Another big thing with v4 is that my goal has shifted from performance to ease-of-use. For example, ANTLR automatically can build parse trees for you and generate listeners and visitors. This is not only a huge productivity win, but also an important step forward in building grammars that don't depend on embedded actions. Those embedded actions (raw Java code or whatever) locked the grammar into use with only one language. If we keep all of the actions out of the grammar and put them into external visitors, we can reuse the same grammar to generate code in any language for which we have an ANTLR target.

Q: What do you think are the things people had problems with in ANTLR3?

A: The biggest problem was figuring out why ANTLR did not like their grammar. The static analysis often could not figure out how to generate a parser for the grammar. This problem totally goes away with the honey badger because it will take just about anything you give it without a whimper.

Q: And what with other compiler generator tools?

A: The biggest problem for the average practitioner is that most parser generators do not produce code you can load into a debugger and step through. This immediately removes bottom-up parser generators and the really powerful GLR parser generators from consideration by the average programmer. There are a few other tools that generate source code like ANTLR does, but they don't have v4's adaptive LL(*) parsers. You will be stuck with contorting your grammar to fit the needs of the tool's weaker, say, LL(k) parsing strategy. PEG-based tools have a number of weaknesses, but to mention one, they have essentially no error recovery because they cannot report an error and until they have parsed the entire input.

Q: What are the main design decisions in ANTLR4?

A: Ease-of-use over performance. I will worry about performance later. Simplicity over complexity. For example, I have taken out explicit/manual AST construction facilities and the tree grammar facilities. For 20 years I've been trying to get people to go that direction, but I've since decided that it was a mistake. It's much better to give people a parser generator that can automatically build trees and then let them use pure code to do whatever tree walking they want. People are extremely familiar and comfortable with visitors, for example.

Q: What do you think people will like most on ANTLR4?

A: The lack of errors when you run your grammar through ANTLR. The automatic tree construction and listener/visitor generation.

Q: What do you think are the problems people will try to solve with ANTLR4?

A: In my experience, almost no one uses parser generators to build commercial compilers. So, people are using ANTLR for their everyday work, building everything from configuration files to little scripting languages.

In response to a question about this entry from stackoverflow.com: I believe that compiler developers are very concerned with parsing speed, error reporting, and error recovery. For that, they want absolute control over their parser. Also, some languages are so complicated, such as C++, that parser generators might build parsers slower than compiler developers want. The compiler developers also like the control of a recursive-descent parser for predicated the parse to handle context-sensitive constructs such as T(i) in C++.

There is also likely a sense that parsing is the easy part of building a compiler so they don't immediately jump automatically to parser generators. I think this is also a function of previous generation parser generators. McPeak's Elkhound GLR-based parser generator is powerful enough and fast enough, in the hands of someone that knows what they're doing, to be suitable for compilers. I can also attest to the fact that ANTLR v4 is now powerful enough and fast enough to compete well with handbuilt parsers. E.g., after warm-up, it's now taking just 1s to parse the entire JDK java/* library.

What is the difference between ANTLR 3 and 4?

First, see [Why do we need ANTLR v4?](#)

The biggest difference between ANTLR 3 and 4 is that ANTLR 4 takes any grammar you give it unless the grammar had indirect left recursion. That means we don't need syntactic predicates or backtracking so ANTLR 4 does not support that syntax; you will get a warning for using it. ANTLR 4 allows direct left recursion so that expressing things like arithmetic expression syntax is very easy and natural:

```
expr : expr '*' expr
     | expr '+' expr
     | INT
     ;
```

ANTLR 4 automatically constructs parse trees for you and abstract syntax tree (AST) construction is no longer an option. See also [What if I need ASTs not parse trees for a compiler, for example?](#)

Another big difference is that we discourage the use of actions directly within the grammar because ANTLR 4 automatically generates [listeners](#) and [visitors](#) for you to use that trigger method calls when some phrases of interest are recognized during a tree walk after parsing. See also [Parse Tree Matching and XPath](#).

Semantic predicates are still allowed in both the parser and lexer rules as our actions. For efficiency sake keep semantic predicates to the right edge of lexical rules.

There are no tree grammars because we use listeners and visitors instead.

Why is my expression parser slow?

Hi. Make sure to use two-stage parsing. See example in [bug report](#).

```
CharStream input = new ANTLRFileStream(args[0]);
ExprLexer lexer = new ExprLexer(input);
CommonTokenStream tokens = new CommonTokenStream(lexer);
ExprParser parser = new ExprParser(tokens);
parser.getInterpreter().setPredictionMode(PredictionMode.SLL);
try {
    parser.stat(); // STAGE 1
}
catch (Exception ex) {
    tokens.reset(); // rewind input stream
    parser.reset();
    parser.getInterpreter().setPredictionMode(PredictionMode.LL);
    parser.stat(); // STAGE 2
    // if we parse ok, it's LL not SLL
}
```

FAQ - Grammar syntax

FAQ - Lexical analysis

How can I parse non-ASCII text and use characters in token rules?

How do I replace escape characters in string tokens?

Why are my keywords treated as identifiers?

Why are there no whitespace tokens in the token stream?

How can I parse non-ASCII text and use characters in token rules?

See [Using non-ASCII characters in token rules](#).

How do I replace escape characters in string tokens?

Unfortunately, manipulating the text of the token matched by a lexical rule is cumbersome (as of 4.2). You have to build up a buffer and then set the text at the end. Actions in the lexer execute at the associated position in the input just like they do in the parser. Here's an example that does escape character replacement in strings. It's not pretty but it works.

```

grammar Foo;

@members {
StringBuffer buf = new StringBuffer(); // can't make locals in lexer rules
}

STR : '''
    ( '\\\
      ( 'r'  {buf.append('\r');}
      | 'n'  {buf.append('\n');}
      | 't'  {buf.append('\t');}
      | '\\\ {buf.append('\\\\');}
      | '\"' {buf.append('\"');}
    )
    | ~('\\\\'|'\"') {buf.append((char)_input.LA(-1));}
    )*
    ''';

    {setText(buf.toString()); buf.setLength(0); System.out.println(getText());}
;

```

It's easier and more efficient to return original input string and then use a small function to rewrite the string later during a parse tree walk or whatever. But, here's how to do it from within the lexer.

Lexer actions don't work in the interpreter, which includes xpath and tree patterns.

For more on the argument against doing complicated things in the lexer, see the [related lexer-action issue at github](#).

Why are my keywords treated as identifiers?

Keywords such as "begin" are also valid identifiers lexically and so that input is ambiguous. To resolve ambiguities, ANTLR gives precedence to the lexical rules specified first. That implies that you must put the identifier rule after all of your keywords:

```

grammar T;

decl : DEF 'int' ID ';'

DEF : 'def' ; // ambiguous with ID as is 'int'
ID  : [a-z]+ ;

```

Notice that literal 'int' is also physically before the ID rule and will also get precedence.

Why are there no whitespace tokens in the token stream?

The lexer is not sending white space to the parser, which means that the rewrite stream doesn't have access to the tokens either. It is because of the skip lexer command:

```

WS : [ \t\r\n\u000C]+ -> skip
;

```

You have to change all those to `-> channel(HIDDEN)` which will send them to the parser on a different channel, making them available in the token stream, but invisible to the parser.

FAQ - Parse Trees

[What if I need ASTs not parse trees for a compiler, for example?](#)

[When do I use listener/visitor vs XPath vs Tree pattern matching?](#)

What if I need ASTs not parse trees for a compiler, for example?

For writing a compiler, either generate [LLVM-type static-single-assignment form](#) or construct an AST from the parse tree or using actions in grammar, turning off auto-parse-tree construction.

When do I use listener/visitor vs XPath vs Tree pattern matching?

XPath

XPath works great when you need to find specific nodes, possibly in certain contexts. The context is limited to the parents on the way to the root of the tree. For example, if you want to find all ID nodes, use path `"//ID"`. If you want all variable declarations, you might use path `"//vardecl"`. If you only want fields declarations, then you can use some context information via path `"/classdef/vardecl"`, which would only find vardecls that our children of class definitions. You can merge the results of multiple XPath `findAll()`s simulating a set union for XPath. The only caveat is that the order from the original tree is not preserved when you union multiple `findAll()` sets.

Tree pattern matching

Use tree pattern matching when you want to find specific subtree structures such as all assignments to 0 using pattern `"x = 0;"`. (Recall that these are very convenient because you specify the tree structure in the concrete syntax of the language described by the grammar.) If you want to find all assignments of any kind, you can use pattern `"x = <expr>;"` where `<expr>` will find any expression. This works great for matching particular substructures and therefore gives you a bit more ability to specify context. I.e., instead of just finding all identifiers, you can find all identifiers on the left hand side of an expression.

Listeners/Visitors

Using the listener or visitor interfaces give you the most power but require implementing more methods. It might be more challenging to discover the emergent behavior of the listener than a simple tree pattern matcher that says "go find me X under node Y".

Listeners are great when you want to visit many nodes in a tree.

Listeners allow you to compute and save context information necessary for processing at various nodes. For example, when building a symbol table manager for a compiler or translator, you need to compute symbol scopes such as globals, class, function, and code block. When you enter a class or function, you push a new scope and then pop it when you exit that class or function. When you see a symbol, you need to define it or look it up in the proper scope. By having enter/exit listener functions push and pop scopes, listener functions for defining variables simply say something like:

```
scopeStack.peek().define(new VariableSymbol("foo"))
```

That way each listener function does not have to compute its appropriate scope.

Examples: [DefScopesAndSymbols.java](#) and [SetScopeListener.java](#) and [VerifyListener.java](#)

FAQ - Translation

ASTs vs parse trees

Decoupling input walking from output generation

ASTs vs parse trees

I used to do specialized AST (**abstract** syntax tree) nodes rather than (**concrete**) parse trees because I used to think more about compilation and generating bytecode/assembly code. When I started thinking more about translation, I started thinking more about parse trees. For v4, I realized that I did mostly translation. I guess what I'm saying is that maybe parse trees are not as good as ASTs for generating byte codes. Personally, I would rather see `(+ 3 4)` rather than `(expr 3 + 4)` for generating byte codes, but it's not the end of the world. (Can someone fill this in?)

Decoupling input walking from output generation

I suggest creating an intermediate model that represents your output. You walk the parse tree to collect information and create your model. Then, you could almost certainly automatically walk this internal model to generate output based upon string templates that match the class names of the internal model. In other words, define a special `IFStatement` object that has all of the fields you want and then create them as you walk the parse tree. This decoupling of the input from the output is very powerful. Just because we have a parse tree listener doesn't mean that the parse tree

itself is necessarily the best data structure to hold all information necessary to generate code. Imagine a situation where the output is the exact reverse of the input. In that case, you really want to walk the input just to collect data. Generating output should be driven by the internal model not the way it was represented in the input.

FAQ - Actions and semantic predicates

[How do I test if an optional rule was matched?](#)

How do I test if an optional rule was matched?

For optional rule references such as the initialization clause in the following

```
decl : 'var' ID (EQUALS expr)? ;
```

testing to see if that clause was matched can be done using `$EQUALS!=null` or `$expr.ctx!=null` where `$expr.ctx` points to the context or parse tree created for that reference to rule `expr`.

FAQ - Error handling

[How do I perform semantic checking with ANTLR?](#)

How do I perform semantic checking with ANTLR?

See [How to implement error handling in ANTLR4](#)

Articles and Resources

Books



- [Amazon reviews: US, UK, DE, CA, IN](#)
- [Tom Harwood's ANTLR 4 Book Review](#)

Articles

- [Playing with ANTLR4, Primefaces extensions for Code Mirror and web-based DSLs](#)
- [A Tale of Two Grammars](#)
- [ANTLR 4: using the lexer, parser and listener with example grammar](#)
- [Creating External DSLs using ANTLR and Java](#)

Presentations

- [Introduction to ANTLR 4 by Oliver Zeigermann](#)

Videos



Resources

- [Stack overflow ANTLR4 tag](#)
- [Antlr 4 with C# and Visual Studio 2012](#)
- [ANTLR Language Support in VisualStudio](#)
- [Upgrading to ANTLR 4 with C#](#)
- [Generate parsers with Antlr4 via Maven](#)
- [Exploring ANTLR v4](#)
- [antlr4dart](#)

Stale

ANTLRWorks 2

For the latest information about the ANTLRWorks2 antlr development environment, please see [Sam Harwell's page](#). I have moved this docs subtree to a stale area as information is woefully out of date.

1. Overview

Early access releases

These builds of ANTLRWorks 2 are the earliest early builds available for some level of testing and feedback. They are pre-alpha quality, which means many features will be incomplete and/or missing, and it's likely to include bugs which could result in loss of work (crashes, etc.).

Latest release: Sep. 1, 2012 (early access preview 13)

[Download from Tunnel Vision Labs \(zip\)](#) (also available in [7z format](#))

- Ability to specify tabs & whitespace settings for ANTLR and StringTemplate files specifically in Tools > Options > Editor > Formatting
- Initial attempt at "smart" auto-indent in grammar files
- Improved code completion accuracy in a few areas
- Various performance improvements, many of which come from updates to the core ANTLR 4 runtime

Previous releases

Aug. 9, 2012 (early access preview 12)

- Implement the "Go to Declaration" command
- Syntax highlighter support for mode names
- A few bug fixes in code completion and code folding
- Add the "Factor label for set" hint
- Add the "Group set elements" hint
- Various performance improvements

Jul. 12, 2012 (early access preview 11)

- Improved support for lexer char sets and lexer commands
- Syntax diagram supports lexer char sets and negated sets
- Update parser to reflect the change from -> to # for labeled alternatives
- Add the "Open Containing Folder" action to document tabs

Mar. 23, 2012 (early access preview 10)

- Add tooltips for rule references
- Finally fixed the navigator update bug
- Major improvements to code completion
- Fix legacy mode
- Many performance and stability updates

Jan. 31, 2012 (early access preview 9)

- Major performance and stability updates

Jan. 30, 2012 (early access preview 8)

- Add Mark Occurrences feature
- Fix background color of nodes in the syntax diagram

- More performance and stability updates
- Note: Preview 7 earlier in the day was a broken build

Jan. 28, 2012 (early access preview 6)

- Lots of performance and stability updates

Jan. 26, 2012 (early access preview 5)

- Lots of performance and stability updates

Jan. 17, 2012 (early access preview 4)

- Fix syntax highlighting of list labels
- Add basic (trivial) editor indentation support (uses value from previous line)
- Fix several performance and stability issues in the editor
- Fix editor whitespace highlighting when "Show Non-printable Characters" option is enabled

Jan. 9, 2010 (early access preview 3)

- Fix a few exceptions in editor, navigator, and syntax diagram
- Syntax diagram properly shows range literals (previously showed ???)

Jan. 8, 2012 (early access preview 2)

- Syntax highlighting for ANTLR grammars uses an ANTLR 4 lexer (was using ANTLR 3 lexers)
- Performance improvements for code completion
- Automatically reparse the document when legacy mode is enabled
- Fix several unhandled exception messages and an out of memory error

Jan. 6, 2012 (early access preview 1)

- Initial public release

2. Grammar Editing Features

Note: Items in **gray** are intended features but not yet implemented.

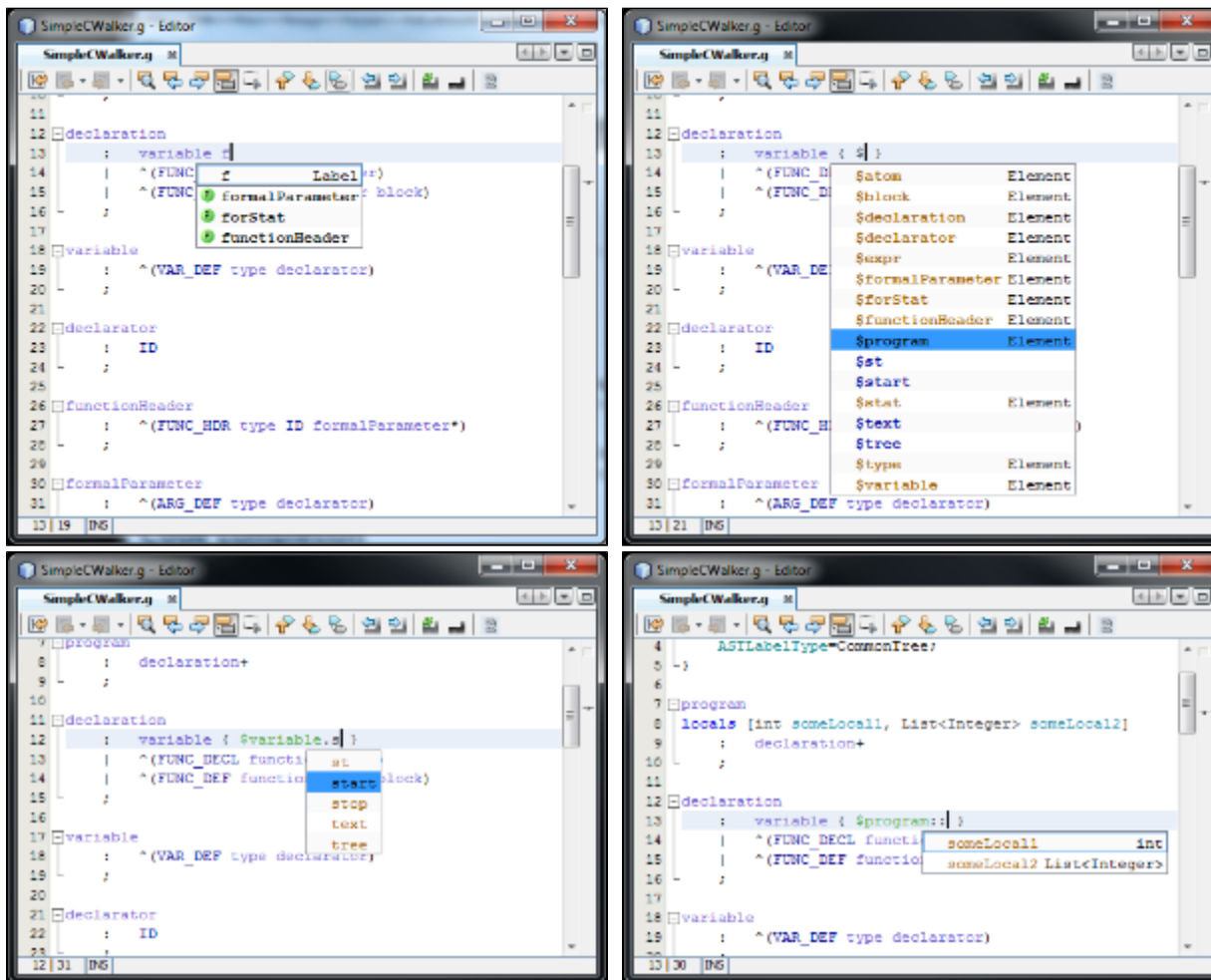
Syntax highlighting

ANTLRWorks 2 uses a fast, synchronous lexer for "primary" syntax highlighting. Eventually, a hybrid synchronous/asynchronous lexer will be used to improve performance in large documents. On-screen text is updated synchronously so the syntax highlighter always appears up-to-date. Off-screen text is updated asynchronously to prevent cascading changes in large documents from incurring editor latencies (such as entering `/ *` which could affect the rest of the document).

Additional semantic highlighting such as parameter, return value, and local declarations and references are performed asynchronously following the reference anchor update task. Eventually these will also be incrementally updated following the dynamic anchor update task to improve latency, and priority will be given to on-screen elements.

Code completion

ANTLRWorks 2 provides *extremely* fast and accurate code completion. Unlike other editors making similar claims, ANTLRWorks 2 provides these features without sacrificing usability - it's always available but won't replace something unless you actually meant it to.

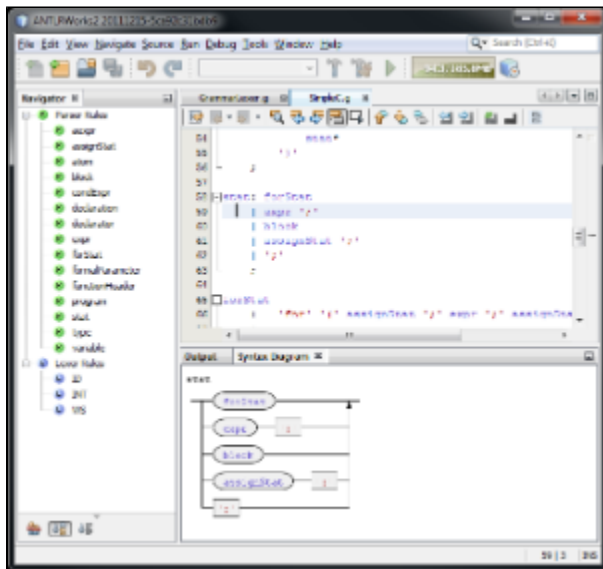


Users interested in implementation details can see the following for reference:

- ANTLRWorks 2 uses the selection, filtering, and completion algorithms described on [Sam Harwell's blog](#). These algorithms provide consistency and a high degree of accuracy in predicting what the user is attempting to complete at any given point.
- ANTLRWorks 2 uses [dynamic anchors](#) to ensure low-latency performance in large documents.
- ANTLRWorks 2 uses [parallel parse trees](#) to ensure automatic completion popups never attempt to replace a declaration of a new item with a reference to an existing item.

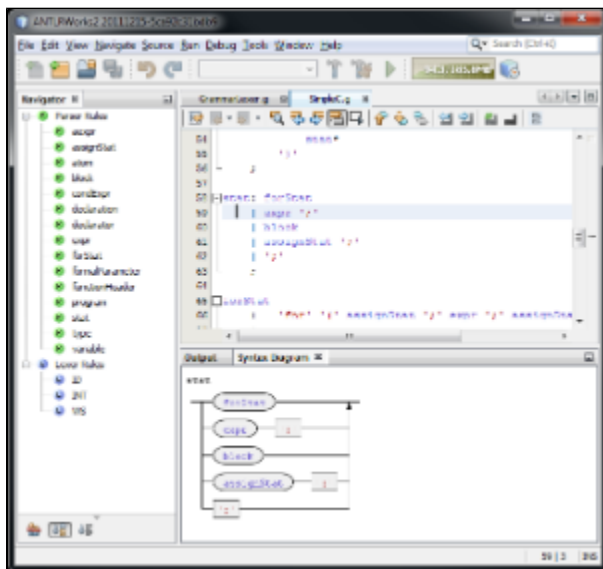
Navigator

ANTLRWorks 2 supports NetBeans' Navigator Window to give an overview of the rules in the current grammar.



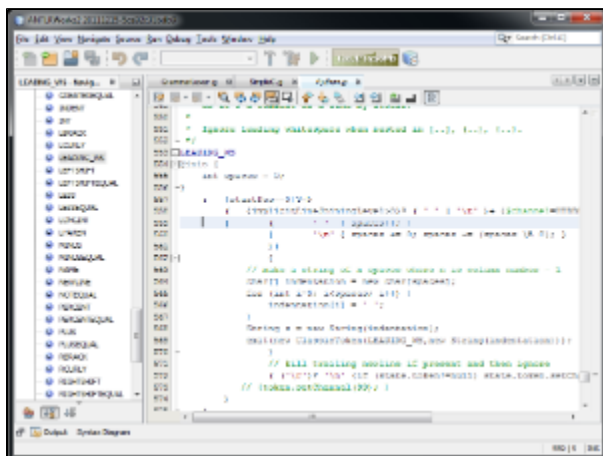
Syntax diagram view

ANTLRWorks 2 provides a syntax diagram window to show a [railroad diagram](#) of the current rule. The window uses a version of Terence Parr's Syntax Diagram control which has been modified to construct itself from ANTLR v4 parse trees.



Legacy mode

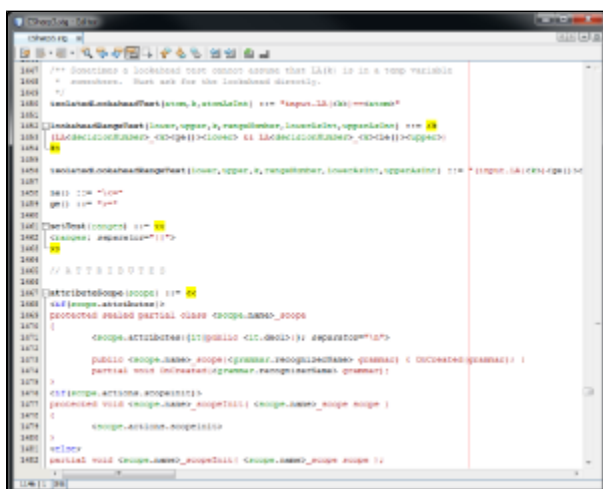
ANTLRWorks 2 includes a legacy mode provides limited support for grammars which target ANTLR v3. Legacy mode is toggled via a button on the document toolbar. When legacy mode is enabled, several features including code completion and the syntax diagram view will be unavailable.



3. StringTemplate 4 Editing Features

Syntax highlighting

The StringTemplate 4 editor includes semantic syntax highlighting. As a quick teaser, the following screenshot shows the ability to distinguish between inherited attribute references (green, italic) and named parameter references (green, plain). It can also handle expression options such as *separator* and *wrap*, shown here in italics.

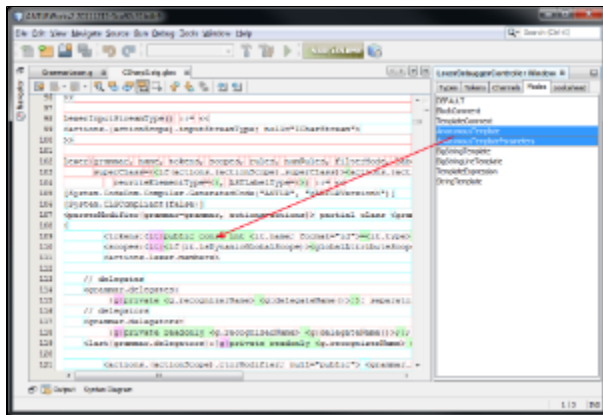


4. Lexer Debugging

Debugger Controller

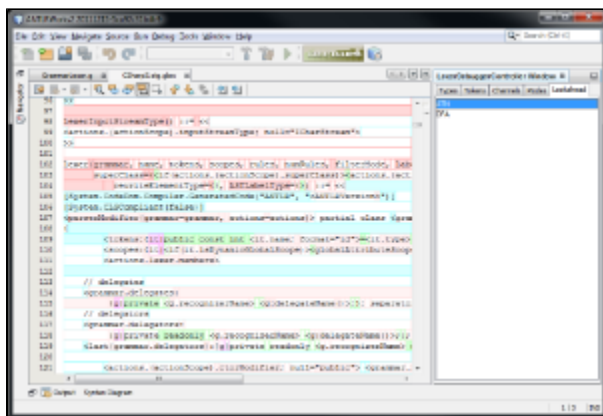
Token types

Lists all token types for the lexer, and allows selection of any number of types to highlight in the input view.



Lookahead

This mode allows highlighting "DFA" tokens which were fully parsed using the cached DFA versus "ATN" tokens which required the ATN interpreter to handle one or more input symbols for the token. In the future this debug mode will be greatly extended to provide more useful information.



5. Parser Debugging

Building ANTLR 4 with Maven

The tool and runtime for ANTLR 4 themselves are no longer built with maven, though of course you can build software that USES ANTLR via the antlr4-maven-plugin. This page used to be about how to build ANTLR itself. Instead we use the build.py Python scripts sitting in the main directory.

Sam Harwell - Dec. 23, 2012

Several phases of the Maven build exist for preparing proper releases. Each of the following variables must be configured or explicitly overridden/ignored.

1. Initial Setup

Before the first time Maven is used to build ANTLR 4 from source, the following command should be executed in the top-level directory of the source tree.

```
mvn -N install
```

2. Quick Start Examples

Each of these examples is written as a "quick start" case, and assumes the following:

- You are building only the minimal set of artifacts required for using ANTLR 4 (i.e. not using the release profile described below)
- You are building the project with Maven 3+ and JDK 6+

2.1. For All Platforms

```
mvn -Duser.name="Your Name" -DskipTests=true install
```

3. Unit Test Configuration

3.1. Skipping Unit Tests

Unit tests may be skipped by passing the following flag to Maven. Note that the unit test classes are still *compiled* when this flag is specified.

```
-DskipTests=true
```

3.2. JDK_SOURCE_ROOT for TestPerformance

One of the unit tests (`TestPerformance`) requires configuring the `JDK_SOURCE_ROOT` for Maven. You should extract the contents of `src.zip` included in the JDK distribution of Java 6, and pass the following flag to Maven. The `JDK_SOURCE_ROOT` folder should contain the subfolder `java/lang` with the Java files for the Java 6 package `java.lang`.

```
-DJDK_SOURCE_ROOT=/path/to/jdk_src
```

3.3. In-Process Testing

The following flag configures the unit tests to execute in-process (via a new `ClassLoader`) rather than creating new Java processes. This results in a 50+% reduction in the time it takes to execute the unit tests, and allows for fully-functional debugging for all of the unit tests (normally code executed in the forked Java processes cannot be debugged directly).

```
-Dantlr4.testinprocess=true
```

4. Complete Build (Release Profile)

To improve development time while working on ANTLR 4, by default Maven builds only the minimal set of Jar files necessary for using ANTLR 4. By passing `-Psonatype-oss-release` to Maven, the complete set of files are built. However, this configuration requires additional configuration as shown in the subsections below. The following table summarizes the files built with and without this option.

	Default Profile	Release Profile <code>-Psonatype-oss-release</code>
antlr4	X	X
antlr4-runtime	X	X
antlr4-maven-plugin	X	X
antlr4-complete		X
Java 6 API usage verification		X
GPG signed		X
Javadoc jars		X
Source jars		X

4.1. GraphViz for Generated Diagrams

The release profile uses [GraphViz](#) to generate SVG graphics from dot files for the generated Javadocs. The `dot` executable should be in your system path for this. Windows users can simply use the [Windows installer for GraphViz 2.28](#) or newer and the path will be automatically

configured.

4.2. Artifact Signing (GPG)

By default Maven will sign the build artifacts with GPG, and assumes you have your system configured according to Sonatype's Blog entry "[How to Generate PGP Signatures with Maven](#)". One of the following should be used each time Maven is used to build the project.

1. Skip the phase by passing `-Dgpg.skip=true` to Maven (required for users without GPG installed or configured).
2. Configure the GPG passphrase by passing `-Dgpg.passphrase=passphrase` to Maven (or simply `-Dgpg.passphrase` if your private key does not require a passphrase).
3. Run the Maven build without specifying one of the above, in which case signing will be enabled and Maven will ask you to enter the passphrase during the signing phase.

4.3. Bootstrap Classpath

The release profile is configured to pass the `-bootclasspath` option to `javac` to ensure that the compiled artifacts and test suites will run with Java 6, even when compiled with Java 7.

Maven needs the path to a copy of `rt.jar` (Windows, Linux) or `Classes.jar` (OSX) from Java 6. This may be configured by any of the following methods:

1. (Windows, Linux) Set the `JAVA6_HOME` environment variable to the home folder of a Java 6 runtime installation. When using this option, the library `rt.jar` should be available in `%JAVA6_HOME%\lib\rt.jar` (Windows) or `$JAVA6_HOME/lib/rt.jar` (Linux).
2. (Windows, Linux) Pass the option `-Djava6.home=path` to Maven. The path should be in the same form as listed for the `JAVA6_HOME` environment variable, and takes precedence over the environment variable if both are specified.
3. (Windows, Linux, OSX) Pass the option `-Dbootclasspath.java6=path` to Maven. This should be the full path to `rt.jar` (Windows, Linux) or `Classes.jar` (OSX).
4. For users without a copy of Java 6 installed, the Java 6 validation may be skipped by passing the option `-Dbootclasspath.compile=path -Dbootclasspath.testCompile=path` to Maven. This should be the full path to `rt.jar` (Windows, Linux) or `Classes.jar` (OSX) for some version of Java newer than Java 6.

4.4. Additional Build Options

These options are not required for building ANTLR 3, but can be useful in certain scenarios. This is not a complete list of options which affect the build, but should include the most commonly used (or "interesting") options.

Option	Result
<code>"-Duser.name=Your Name"</code>	This option controls the value for <code>Built-By</code> contained in the generated <code>MANIFEST.MF</code> . The default value is the username of the user launching the build.
<code>-Dgpg.skip=true</code>	When this option is specified, the signing phase of the build will be skipped. Users without GPG installed must specify this option. This flag only applies to the release profile.
<code>-Dgpg.useagent=true</code>	For users with GPG 2 installed, passing this option to Maven will prevent the build from producing messages about <code>--use-agent</code> being an obsolete option.
<code>-Dmaven.javadoc.skip=true</code>	Skips the generation of Javadoc archive for each artifact. This flag only applies to the release profile.
<code>-Dsource.skip=true</code>	Skips the generation of a source archive for each artifact. This flag only applies to the release profile.
<code>-DskipTests=true</code>	Skips the execution of unit tests as part of the build.
<code>-Dmaven.test.skip=true</code>	Skips both the compilation and execution of unit tests as part of the build.

TODO list

4.0

- 3.5 release
- api doc for 4.0

LL(1) Optimization

It's not correct to avoid full LL for k=1 SLL conflict; we found a counterexample. something to do with the fact that, while the class of LL(1)=SLL(1), that doesn't mean that the parser decisions are equally powerful. Anyway, Sam reports that it's a big optimization to put this back in. Then, if we get a syntax error, we fail over to full LL. That leaves us with 3 stages SLL -> LL+k=1 -> LL.

I said "LL(1) == SLL(1)", Sam says "this results in a faulty reportAmbiguity on the else in "{ if a then foo else bar }", when in fact it's only a conflict for "{ if a then if a then foo else bar }"

Semantics and error checking

- warn about non-unique refs to elements from headers like \$ID
- warn if \$e used for rule ref in rule e. e : ... | '(' e ')' {print \$e.v;} // doesn't translate to rule ref

parse trees

- xpath or jquery like feature to do find nodes
- tree pattern matching? e.g., find all (e 1 1) trees.
- Create method to create new parse tree with concrete syntax.
return parse("stat", "while (i>3) {...}");
- Find a way to have some nodes not appear in parse tree or at least in listener. just skip creating new _localctx.
- (I believe this is corrected now) ~~Take this example: e : ID | e ' ' ID~~
~~With the input "a.a.a" you get (select (select a a) a) the .stop value for the outer ctx is correct, but for the inner (select a a) it's null~~
~~correction, i'm not sure it's correct for the outer due to a syntax error occuring in mine, but it's definitely null for the inner: ID | e ' ' ID~~
~~With the input "a.a.a" you get (select (select a a) a) the .stop value for the outer ctx is correct, but for the inner (select a a) it's null~~
~~correction, i'm not sure it's correct for the outer due to a syntax error occuring in mine, but it's definitely null for the inner. yep,~~
~~_localctx.stop is set at the very end of the rule, but it needs to be inside the postfix expr loop. right after "_prevctx = _localctx;" add~~
~~"_prevctx.stop = input.LT(-1);" Should be last real token; not conjured.~~
- need to know if an error occurred in rule.

syntax

- ID*[';'] comma-separated list of ID
- &foo syn preds like pegs; 0 width. where are they allowed?

Code generation

- move all ctx objects to bottom?
- be able to split big grammar into chunks using inheritance
- split serialized atn

analysis

- [Sam's LL\(1\) optimization](#)
- Sam's optimized ATN transition thingie. optimize during deserialization.
- tail call optimization. x : a b; closure can jump to b not pushing frame since if we ever fall out of entry rule, x, it'll compute FOLLOW in SLL mode. Slightly weaker since we might do FOLLOW when we could have specific call stack. but drops mem like 50%. this grammar would try to compute FOLLOW at end of x (stack is s calls a) since we didn't push calls site in a.

```
s : a INT;
s2 : b ID;
a : x;
b : x;
x : ID;
```

s calls a then a pretends to make decision but doesn't push call. at end of x let's say we look further but have no stack in SLL mode so

we do FOLLOW. It'll see ID in there but in real SLL, we'd see call site in a on stack.

Sam now reports that you can't skip the push when calling a rule from the decision entry rule in the lexer since we stop at stop state when stack empty in match(). This gives the wrong token type. Works to resolve weakness in SLL if we push tail call frame from dec entry rule since it doesn't fall back to FOLLOW too early.

- tail call elimination has a big impact.
- DFAs mostly have one edge. optimize to avoid array for this case
- testNotSetRuleRootInLoop. ~set in LL1Analyzer doesn't compute ~
- big expressions still are slow due to full LL
- turn on predctx cache for lexer

Errors

- add sync()-like functionality to prediction so that, even during prediction, we can do single token insertion or deletion.

Visitors/ event listeners

-

Options

Runtime

- `CommonToken.getText()` points at the current input stream for the lexer, but if somebody resets it, it will point at the wrong stream. added new pointer or replace the token source in the token object; Sam suggests sharing a 2 ptr object that has the lexer and the input stream pointer.
- Make an efficient token object

lexers

- `[\]]` should be just `]` on inside not `\\` and `]`
- `tokenVocab '++'=33` imported to lexer doesn't define a literal; should it? nah
- don't allow actions in fragment rules; they don't exec
- don't allow labels or parameters or return values on lexer elements
- should we allow same token name in multiple modes? seems useful.
- Using the first `ANY_GENERAL` rule, it consumes everything. Swapping for the second `ANY_GENERAL` rule, it works as intended.

```
fragment START_TAG : '<';

fragment WORDCHAR : 'a'..'z' | 'A'..'Z' | '0'..'9' | '_';

TAG_START : START_TAG WORDCHAR+ { pushMode(IN_TAG); };

ANY_GENERAL : ~START_TAG+;

// ANY_GENERAL : ~'<'+;
```

I don't understand why they are not doing the same thing?

- import mode pulls rules into another mode; shares common stuff like WS, ID, etc...

Misc

- `@ANTLR(...)` to compile grammars in package
- `@api` to signify stuff to use from antlr api vs public; Sam points out that this really should be a Java interface.
- Consider this example:

- `expr` returns `[int r]`
 `: '-' expr { $r = - $expr.r; }`

In this example `$expr` should bind to the sub-expression in my opinion. However, it does not. Since the rule is also named `expr`, `$expr` refers to the rule context instead of the context of the sub-expression. I think most of the time this is not what the user wants.