Java News Brief
www.ociweb.com

Java Technical Insight of the Month

ANTLR 3

by
R. Mark Volkmann, Partner/Software Engineer
Object Computing, Inc. (OCI)

Preface

ANTLR is a big topic, so this is a big article. The table of contents that follows contains hyperlinks to allow easy navigation to the many topics discussed. Topics are introduced in the order in which understanding them is essential to the example code that follows. Your questions and feedback are welcomed at mark@ociweb.com.

Table of Contents

Part I - Overview

Introduction to ANTLR

ANTLR is a free, open source parser generator tool that is used to implement "real" programming languages and domain-specific languages (DSLs). The name stands for ANother Tool for Language Recognition. Terence Parr, a professor at the University of San Francisco, implemented (in Java) and maintains it. It can be downloaded from http://www.antlr.org. This site also contains documentation, articles, examples, a Wiki and information about mailing lists.

Many people feel that ANTLR is easier to use than other, similar tools. One reason for this is the syntax it uses to express grammars. Another is the existence of a graphical grammar editor and debugger called ANTLRWorks. Jean Bovet, a former masters student at the University of San Francisco who worked with Terence, implemented (using Java Swing) and maintains it.

A brief word about conventions in this article... ANTLR grammar syntax makes frequent use of the characters [ ] and { }. When describing a placeholder we will use italics rather than surrounding it with { }. When describing something that's optional, we'll follow it with a question mark rather than surrounding it with [ ].

ANTLR Overview

ANTLR uses Extended Backus-Naur (EBNF) grammars which can directly express optional and repeated elements. BNF grammars require a more verbose syntax to express these. EBNF grammars also support "subrules" which are parenthesized groups of elements.

ANTLR supports infinite lookahead for selecting the rule alternative that matches the portion of the input stream being evaluated. The technical way of stating this is that ANTLR supports LL(*). An LL(k) parser is a top-down parser that parses from left to right, constructs a leftmost derivation of the input and looks ahead k tokens when selecting between rule alternatives. The * means any number of lookahead tokens. Another type of parser, LR(k), is a bottom-up parser that parses from left to right and constructs a rightmost derivation of the input. LL parsers can't handle left-recursive rules so those must be avoided when writing ANTLR grammars. Most people find LL grammars easier to understand than LR grammars. See Wikipedia for a more detailed descriptions of LL and LR parsers.

ANTLR supports three kinds of predicates that aid in resolving ambiguities. These allow rules that are not based strictly on input syntax.

While ANTLR is implemented in Java, it generates code in many target languages including Java, Ruby, Python, C, C++, C# and Objective C.

There are IDE plug-ins available for working with ANTLR inside IDEA and Eclipse, but not yet for NetBeans or other IDEs.

Use Cases

There are three primary use cases for ANTLR.

The first is implementing "validators." These generate code that validates that input obeys grammar rules.

The second is implementing "processors." These generate code that validates and processes input. They can perform calculations, update databases, read configuration files into runtime data structures, etc. Our Math

example coming up is an example of a processor.

The third is implementing "translators." These generate code that validates and translates input into another format such as a programming language or bytecode.

Later we'll discuss "actions" and "rewrite rules." It's useful to point out where these are used in the three use cases above. Grammars for validators don't use actions or rewrite rules. Grammars for processors use actions, but not rewrite rules. Grammars for translators use actions (containing `println`s) and/or rewrite rules.

Other DSL Approaches

Dynamic languages like Ruby and Groovy can be used to implement many DSLs. However, when they are used, the DSLs have to live within the syntax rules of the language. For example, such DSLs often require dots between object references and method names, parameters separated by commas, and blocks of code surrounded by curly braces or do/end keywords. Using a tool like ANTLR to implement a DSL provides maximum control over the syntax of the DSL.

Definitions

Lexer
: converts a stream of characters to a stream of tokens (ANTLR token objects know their start/stop character stream index, line number, index within the line, and more)

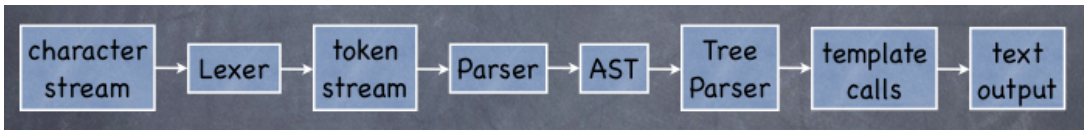
Parser
: processes a stream of tokens, possibly creating an AST

Abstract Syntax Tree (AST)
: an intermediate tree representation of the parsed input that is simpler to process than the stream of tokens and can be efficiently processed multiple times

Tree Parser
: processes an AST

StringTemplate
: a library that supports using templates with placeholders for outputting text (ex. Java source code)

An input character stream is fed into the lexer. The lexer converts this to a stream of tokens that is fed to the parser. The parser often constructs an AST which is fed to the tree parser. The tree parser processes the AST and optionally produces text output, possibly using StringTemplate.



General Steps

The general steps involved in using ANTLR include the following.

1. Write the grammar using one or more files.
   A common approach is to use three grammar files, each focusing on a specific aspect of the processing. The first is the lexer grammar, which creates tokens from text input. The second is the parser grammar, which creates an AST from tokens. The third is the tree parser grammar, which processes an AST. This results in three relatively simple grammar files as opposed to one complex grammar file.
2. Optionally write StringTemplate templates for producing output.
3. Debug the grammar using ANTLRWorks.
4. Generate classes from the grammar. These validate that text input conforms to the grammar and execute target language "actions" specified in the grammar.
5. Write an application that uses the the generated classes.
6. Feed the application text that conforms to the grammar.

Part II - Jumping In

Example Description

Enough background information, let's create a language!

Here's a list of features we want our language to have:

- run on a file or interactively
- get help using `?` or `help`
- support a single data type: double
- assign values to variables using syntax like `a = 3.14`
- define polynomial functions using syntax like `f(x) = 3x^2 - 4x + 2`
- print strings, numbers, variables and function evaluations using syntax like `print "The value of f for " a " is " f(a)`
- print the definition of a function and its derivative using syntax like `print "The derivative of " f() " is " f'()`
- list variables and functions using `list variables` and `list functions`
- add and subtract functions using syntax like `h = f - g` (note that the variables used in the functions do not have to match)
- exit using `exit` or `quit`

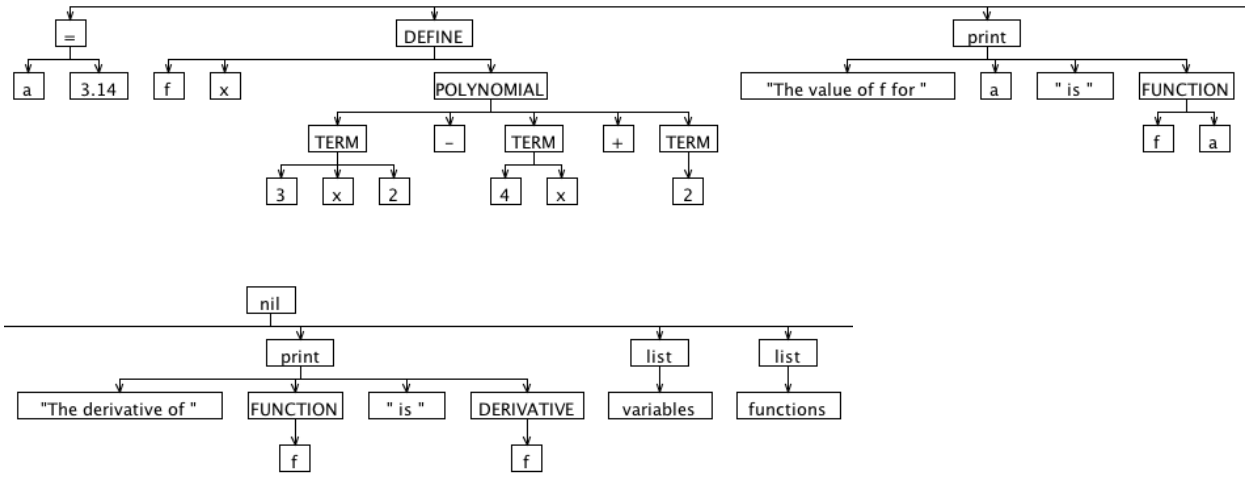
Here's some example input.

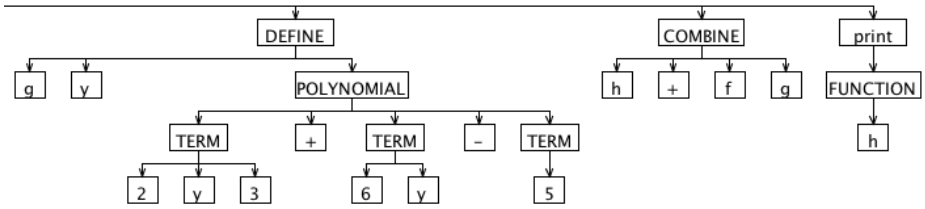
```
a = 3.14
f(x) = 3x^2 - 4x + 2
print "The value of f for " a " is " f(a)

print "The derivative of " f() " is " f'()

list variables
list functions

g(y) = 2y^3 + 6y - 5
h = f + g
print h()
```

Here's the output that would be produced.

```
The value of f for 3.14 is 19.0188
The derivative of f(x) = 3x^2 - 4x + 2 is f'(x) = 6x - 4
# of variables defined: 1
a = 3.14
# of functions defined: 1
f(x) = 3x^2 - 4x + 2
h(x) = 2x^3 + 3x^2 + 2x - 3
```

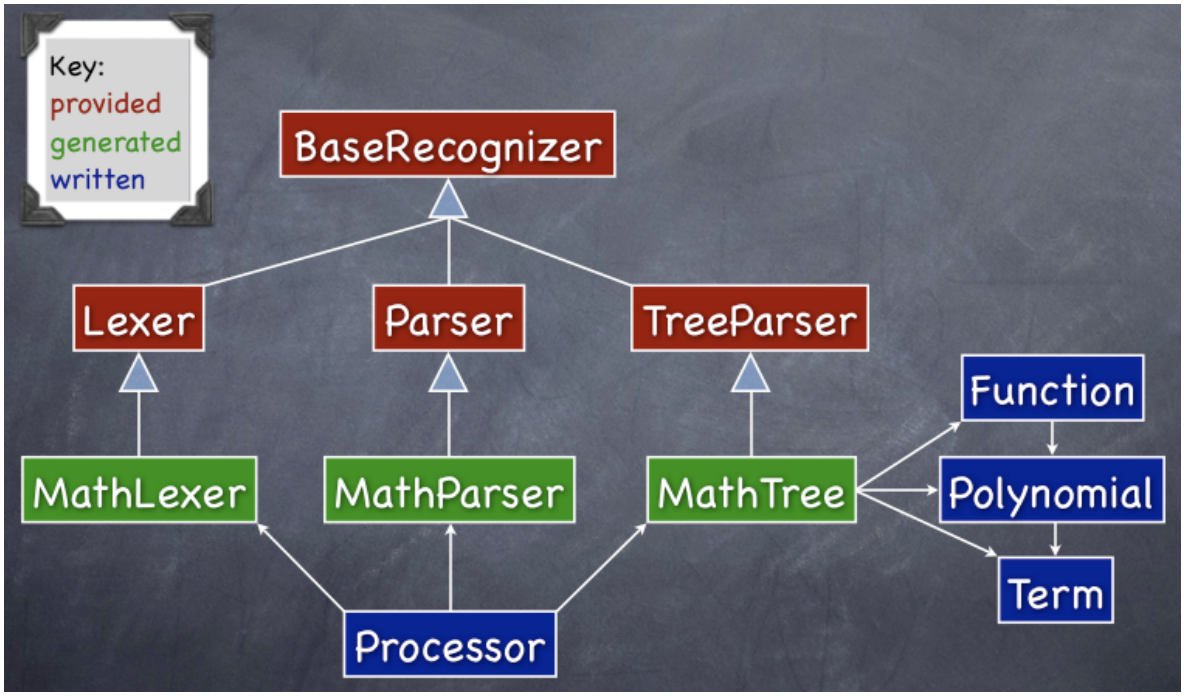
Here's the AST we'd like to produce for the input above, drawn by ANTLRWorks. It's split into three parts because the image is really wide. The "nil" root node is automatically supplied by ANTLR. Note the horizontal line under the "nil" root node that connects the three graphics. Nodes with uppercase names are "imaginary nodes" added for the purpose of grouping other nodes. We'll discuss those in more detail later.





Important Classes

The diagram below shows the relationships between the most important classes used in this example.



Note the key in the upper-left corner of the diagram that distinguishes between classes provided by ANTLR, classes generated from our grammar by ANTLR, and classes we wrote manually.

Grammar Syntax

The syntax of an ANTLR grammar is described below.

```
grammar-type? grammar grammar-name;
grammar-options?
token-spec?
attribute-scopes?
grammar-actions?
rule+
```

Comments in an ANTLR grammar use the same syntax as Java. There are three types of grammars: lexer, parser and tree. If a *grammar-type* isn't specified, it defaults to a combined lexer and parser. The name of the file containing the grammar must match the *grammar-name* and have a ".g" extension. The classes generated by ANTLR will contain a method for each rule in the grammar. Each of the elements of grammar syntax above will be discussed in the order they are needed to implement our Math language.

Grammar Options

Grammar options include the following:

AST node type - `ASTLabelType = CommonTree`
This is used in grammars that create or parse ASTs. CommonTree is a provided class. It is also possible

to use your own class to represent AST nodes.

infinite lookahead - `backtrack = true`
This provides infinite lookahead for all rules. Parsing is slower with this on.

limited lookahead - `k = integer`
This limits lookahead to a given number of tokens.

output type - `output = AST | template`
Choose `template` when using the StringTemplate library.
Don't set this when not producing output or doing so with printlns in actions.

token vocabulary - `tokenVocab = grammar-name`
This allows one grammar file to use tokens defined in another (with lexer rules or a token spec.). It reads generated `.token` files.

Grammar options are specified using the following syntax. Note that quotes aren't needed around single word values.

```
options {
  name = 'value';
  . . .
}
```

Grammar Actions

Grammar actions add code to the generated code. There are three places where code can be added by a grammar action.

1. Before the generated class definition:
   This is commonly used to specify a Java package name and import classes in other packages. The syntax for adding code here is `@header { ... }`. In a combined lexer/parser grammar, this only affects the generated parser class. To affect the generated lexer class, use `@lexer::header { ... }`.

2. Inside the generated class definition:
   This is commonly used to define constants, attributes and methods accessible to all rule methods in the generated classes. It can also be used to override methods in the superclasses of the generated classes. The syntax for adding code here is `@members { ... }`. In a combined lexer/parser grammar, this only affects the generated parser class. To affect the generated lexer class, use `@lexer::members { ... }`.

3. Inside generated methods:
   The catch blocks for the try block in the methods generated for each rule can be customized. One use for this is to stop processing after the first error is encountered rathering than attempting to recover by skipping unrecognized tokens.
   The syntax for adding catch blocks is `@rulecatch { catch-blocks }`

Part III - Lexers

Lexer Rules

A lexer rule or token specification is needed for every kind of token to be processed by the parser grammar. The names of lexer rules must start with an uppercase letter and are typically all uppercase. A lexer rule can be associated with:

- a single literal string expected in the input
- a selection of literal strings that may be found
- a sequence of specific characters and ranges of characters using the cardinality indicators ?, * and +

A lexer rule cannot be associated with a regular expression.

When the lexer chooses the next lexer rule to apply, it chooses the one that matches the most characters. If there is a tie then the one listed first is used, so order matters.

A lexer rule can refer to other lexer rules. Often they reference "fragment" lexer rules. These do not result in creation of tokens and are only present to simplify the definition of other lexer rules. In the example ahead, LETTER and DIGIT are fragment lexer rules.

Whitespace and Comments

Whitespace and comments in the input are handled in lexer rules. There are two common options for handling these: either throw them away or write them to a different "channel" that is not automatically inspected by the parser. To throw them away, use "skip();". To write them to the special "hidden" channel, use "$channel = HIDDEN;".

Here are examples of lexer rules that handle whitespace and comments.

```
// Send runs of space and tab characters to the hidden channel.
WHITESPACE: (' ' | '\t')+ { $channel = HIDDEN; };

// Treat runs of newline characters as a single NEWLINE token.
// On some platforms, newlines are represented by a \n character.
// On others they are represented by a \r and a \n character.
NEWLINE: ('\r'? '\n')+;

// Single-line comments begin with //, are followed by any characters
// other than those in a newline, and are terminated by newline characters.
SINGLE_COMMENT: '//' ~('\r' | '\n')* NEWLINE { skip(); };

// Multi-line comments are delimited by /* and */
// and are optionally followed by newline characters.
MULTI_COMMENT options { greedy = false; }
  : '/*' .* '*/' NEWLINE? { skip(); };
```

When the greedy option is set to true, the lexer matches as much input as possible. When false, it stops when input matches the next element in the lexer rule. The greedy option defaults to true except when the patterns ".*" and ".+" are used. For this reason, it didn't need to be specified in the example above.

If newline characters are to be used as statement terminators then they shouldn't be skipped or hidden since the parser needs to see them.

Our Lexer Grammar

```
lexer grammar MathLexer;

// We want the generated lexer class to be in this package.
@header { package com.ociweb.math; }

APOSTROPHE: '\''; // for derivative
ASSIGN: '=';
CARET: '^'; // for exponentiation
FUNCTIONS: 'functions'; // for list command
HELP: '?' | 'help';
LEFT_PAREN: '(';
LIST: 'list';
PRINT: 'print';
RIGHT_PAREN: ')';
SIGN: '+' | '-';
VARIABLES: 'variables'; // for list command

NUMBER: INTEGER | FLOAT;
fragment FLOAT: INTEGER '.' '0'..'9'+;
fragment INTEGER: '0' | SIGN? '1'..'9' '0'..'9'*;
NAME: LETTER (LETTER | DIGIT | '_')*;
STRING_LITERAL: '"' NONCONTROL_CHAR* '"';

fragment NONCONTROL_CHAR: LETTER | DIGIT | SYMBOL | SPACE;
fragment LETTER: LOWER | UPPER;
fragment LOWER: 'a'..'z';
fragment UPPER: 'A'..'Z';
fragment DIGIT: '0'..'9';
fragment SPACE: ' ' | '\t';

// Note that SYMBOL does not include the double-quote character.
fragment SYMBOL: '!' | '#'..'/' | ':'..'@' | '['..'`' | '{'..'~';
```

```
// Windows uses \r\n. UNIX and Mac OS X use \n.
// To use newlines as a terminator,
// they can't be written to the hidden channel!
NEWLINE: ('\r'? '\n')+;
WHITESPACE: SPACE+ { $channel = HIDDEN; };
```

We'll be looking at the parser grammar soon. When parser rule alternatives contain literal strings, they are converted into references to automatically generated lexer rules. For example, we could eliminate the ASSIGN lexer rule above and change `ASSIGN` to `'='` in the parser grammar.

Part IV - Parsers

Token Specifications

The lexer creates tokens for all input character sequences that match lexer rules. It can be useful to create other tokens that either don't exist in the input (imaginary) or have a better name than what is found in the input. Imaginary tokens are often used to group other tokens. In the parser grammar ahead, the tokens that play this role are DEFINE, POLYNOMIAL, TERM, FUNCTION, DERIVATIVE and COMBINE.

The syntax for specifying these kinds of tokens in a parser grammar is:

```
tokens {
  imaginary-name;
  better-name = 'input-name';
}
```

Rule Syntax

The syntax for defining rules is

```
fragment? rule-name arguments?
(returns return-values)?
throws-spec?
rule-options?
rule-attribute-scopes?
rule-actions?
  : token-sequence-1
  | token-sequence-2
  ...
;
exceptions-spec?
```

The `fragment` keyword only appears at the beginning of lexer rules that are used as fragments (described earlier).

Rule options include `backtrack` and `k` which customize those options for a specific rule instead of using the grammar-wide values specified as grammar options. They are specified using the syntax `options { ... }`.

The token sequences are alternatives that can be selected by the rule. Each element in the sequences can be followed by an action which is target language code (such as Java) in curly braces. The code is executed immediately after a preceding element is matched by input.

The optional *exceptions-spec* customizes exception handling for this rule.

Elements in a token sequence can be assigned to variables so they can be accessed in actions. To obtain the text value of a token that is referred to by a variable, use `$variable.text`. There are several examples of this in the parser grammar that follows.

Creating ASTs

Parser grammars often create ASTs. To do this, the grammar option `output` must be set to `AST`.

There are two approaches for creating ASTs. The first is to use "rewrite rules". These appear after a rule alternative. This is the recommended approach in most cases. The syntax of a rewrite rule is
`-> ^(parent child-1 child-2 ... child-n)`

The second approach for creating ASTs is to use AST operators. These appear in a rule alternative, immediately after tokens. They work best for sequences like mathematical expressions. There are two AST operators. When a `^` is used, a new root node is created for all child nodes at the same level. When a `!` is used, no node is created. This is often used for bits of syntax that aren't needed in the AST such as parentheses, commas and semicolons. When a token isn't followed by one of them, a new child node is created for that token using the current root node as its parent.

A rule can use both of these approaches, but each rule alternative can only use one approach.

Rule Arguments and Return Values

The following syntax is used to declare rule arguments and return types.

```
rule-name[type1 name1, type2 name2, ...]
returns [type1 name1, type2 name2, ...] :
  ...
;
```

The names after the rule name are arguments and the names after the `returns` keyword are return values.

Note that rules can return more than one value. ANTLR generates a class to use as the return type of the generated method for the rule. Instances of this class hold all the return values. The generated method name matches the rule name. The name of the generated return type class is the rule name with "_return" appended.

Our Parser Grammar

```
parser grammar MathParser;

options {
  // We're going to output an AST.
  output = AST;

  // We're going to use the tokens defined in our MathLexer grammar.
  tokenVocab = MathLexer;
}

// These are imaginary tokens that will serve as parent nodes
// for grouping other tokens in our AST.
tokens {
  COMBINE;
  DEFINE;
  DERIVATIVE;
  FUNCTION;
  POLYNOMIAL;
  TERM;
}

// We want the generated parser class to be in this package.
@header { package com.ociweb.math; }

// This is the "start rule".
// EOF is a predefined token that represents the end of input.
// The "start rule" should end with this.
// Note the use of the ! AST operator
// to avoid adding the EOF token to the AST.
script: statement* EOF!;

statement: assign | define | interactiveStatement | combine | print;

// These kinds of statements only need to be supported
```

```
// when reading input from the keyboard.
interactiveStatement: help | list;

// Examples of input that match this rule include
// "a = 19", "a = b", "a = f(2)" and "a = f(b)".
assign: NAME ASSIGN value terminator -> ^(ASSIGN NAME value);

value: NUMBER | NAME | functionEval;

// A parenthesized group in a rule alternative is called a "subrule".
// Examples of input that match this rule include "f(2)" and "f(b)".
functionEval
  : fn=NAME LEFT_PAREN (v=NUMBER | v=NAME) RIGHT_PAREN -> ^(FUNCTION $fn $v);

// EOF cannot be used in lexer rules, so we made this a parser rule.
// EOF is needed here for interactive mode where each line entered ends in EOF
// and for file mode where the last line ends in EOF.
terminator: NEWLINE | EOF;

// Examples of input that match this rule include
// "f(x) = 3x^2 - 4" and "g(x) = y^2 - 2y + 1".
// Note that two parameters are passed to the polynomial rule.
define
  : fn=NAME LEFT_PAREN fv=NAME RIGHT_PAREN ASSIGN
    polynomial[$fn.text, $fv.text] terminator
    -> ^(DEFINE $fn $fv polynomial);

// Examples of input that match this rule include
// "3x2 - 4" and "y^2 - 2y + 1".
// fnt = function name text; fvt = function variable text
// Note that two parameters are passed in each invocation of the term rule.
polynomial[String fnt, String fvt]
  : term[$fnt, $fvt] (SIGN term[$fnt, $fvt])*
    -> ^(POLYNOMIAL term (SIGN term)*);

// Examples of input that match this rule include
// "4", "4x", "x^2" and "4x^2".
// fnt = function name text; fvt = function variable text
term[String fnt, String fvt]
  // tv = term variable
  : c=coefficient? (tv=NAME e=exponent?)?
    // What follows is a validating semantic predicate.
    // If it evaluates to false, a FailedPredicateException will be thrown.
    // It is testing whether the term variable matches the function variable.
    { tv == null ? true : ($tv.text).equals($fvt) }?
    -> ^(TERM $c? $tv? $e?)
  ;
  // This catches bad function definitions such as
  // f(x) = 2y
  catch [FailedPredicateException fpe] {
    String tvt = $tv.text;
    String msg = "In function \"" + fnt +
      "\" the term variable \"" + tvt +
      "\" doesn't match function variable \"" + fvt + "\".";
    throw new RuntimeException(msg);
  }

coefficient: NUMBER;

// An example of input that matches this rule is "^2".
exponent: CARET NUMBER -> NUMBER;

// Inputs that match this rule are "?" and "help".
help: HELP terminator -> HELP;

// Inputs that match this rule include
// "list functions" and "list variables".
list
  : LIST listOption terminator -> ^(LIST listOption);
```

```
// Inputs that match this rule are "functions" and "variables".
listOption: FUNCTIONS | VARIABLES;

// Examples of input that match this rule include
// "h = f + g" and "h = f - g".
combine
  : fn1=NAME ASSIGN fn2=NAME op=SIGN fn3=NAME terminator
    -> ^(COMBINE $fn1 $op $fn2 $fn3);

// An example of input that matches this rule is
// print "f(" a ") = " f(a)
print
  : PRINT printTarget* terminator -> ^(PRINT printTarget*);

// Examples of input that match this rule include
// 19, 3.14, "my text", a, f(), f(2), f(a) and f'().
printTarget
  : NUMBER -> NUMBER
  | sl=STRING_LITERAL -> $sl
  | NAME -> NAME
  // This is a function reference to print a string representation.
  | NAME LEFT_PAREN RIGHT_PAREN -> ^(FUNCTION NAME)
  | functionEval
  | derivative
  ;

// An example of input that matches this rule is "f'()".
derivative
  : NAME APOSTROPHE LEFT_PAREN RIGHT_PAREN -> ^(DERIVATIVE NAME);
```

Part V - Tree Parsers

Rule Actions

Rule actions add code before and/or after the generated code in the method generated for a rule. They can be used for AOP-like wrapping of methods. The syntax `@init { ...code... }` inserts the contained code before the generated code. The syntax `@after { ...code... }` inserts the contained code after the generated code. The tree grammar rules `polynomial` and `term` ahead demonstrate using `@init`.

Attribute Scopes

Data is shared between rules in two ways: by passing parameters and/or returning values, or by using attributes. These are the same as the options for sharing data between Java methods in the same class. Attributes can be accessible to a single rule (using `@init` to declare them), a rule and all rules invoked by it (rule scope), or by all rules that request the named global scope of the attributes.

Attribute scopes define collections of attributes that can be accessed by multiple rules. There are two kinds, global and rule scopes.

Global scopes are named scopes that are defined outside any rule. To request access to a global scope within a rule, add `scope name;` to the rule. To access multiple global scopes, list their names separated by spaces. The following syntax is used to define a global scope.

```
scope name {
  type variable;
  ...
}
```

Rule scopes are unnamed scopes that are defined inside a rule. Rule actions in the defining rule and rules invoked by it access attributes in the scope with `$rule-name::variable` . The following syntax is used to define a rule scope.

```
scope {
  type variable;
  ...
}
```

To initialize an attribute, use an `@init` rule action.

Our Tree Grammar

```
tree grammar MathTree;

options {
  // We're going to process an AST whose nodes are of type CommonTree.
  ASTLabelType = CommonTree;

  // We're going to use the tokens defined in
  // both our MathLexer and MathParser grammars.
  // The MathParser grammar already includes
  // the tokens defined in the MathLexer grammar.
  tokenVocab = MathParser;
}

@header {
  // We want the generated parser class to be in this package.
  package com.ociweb.math;

  import java.util.Map;
  import java.util.TreeMap;
}

// We want to add some fields and methods to the generated class.
@members {
  // We're using TreeMaps so the entries are sorted on their keys
  // which is desired when listing them.
  private Map<String, Function> functionMap = new TreeMap<String, Function>();
  private Map<String, Double> variableMap = new TreeMap<String, Double>();

  // This adds a Function to our function Map.
  private void define(Function function) {
    functionMap.put(function.getName(), function);
  }

  // This retrieves a Function from our function Map
  // whose name matches the text of a given AST tree node.
  private Function getFunction(CommonTree nameNode) {
    String name = nameNode.getText();
    Function function = functionMap.get(name);
    if (function == null) {
      String msg = "The function \"" + name + "\" is not defined.";
      throw new RuntimeException(msg);
    }
    return function;
  }

  // This evaluates a function whose name matches the text
  // of a given AST tree node for a given value.
  private double evalFunction(CommonTree nameNode, double value) {
    return getFunction(nameNode).getValue(value);
  }

  // This retrieves the value of a variable from our variable Map
  // whose name matches the text of a given AST tree node.
  private double getVariable(CommonTree nameNode) {
    String name = nameNode.getText();
    Double value = variableMap.get(name);
```

```
    if (value == null) {
      String msg = "The variable \"" + name + "\" is not set.";
      throw new RuntimeException(msg);
    }
    return value;
  }

  // This just shortens the code for print calls.
  private static void out(Object obj) {
    System.out.print(obj);
  }

  // This just shortens the code for println calls.
  private static void outln(Object obj) {
    System.out.println(obj);
  }

  // This converts the text of a given AST node to a double.
  private double toDouble(CommonTree node) {
    double value = 0.0;
    String text = node.getText();
    try {
      value = Double.parseDouble(text);
    } catch (NumberFormatException e) {
      throw new RuntimeException("Cannot convert \"" + text + "\" to a double.");
    }
    return value;
  }

  // This replaces all escaped newline characters in a String
  // with unescaped newline characters.
  // It is used to allow newline characters to be placed in
  // literal Strings that are passed to the print command.
  private static String unescape(String text) {
    return text.replaceAll("\\\\n", "\n");
  }

} // @members

script: statement*;

statement: assign | combine | define | interactiveStatement | print;

// These kinds of statements only need to be supported
// when reading input from the keyboard.
interactiveStatement: help | list;

// This adds a variable to the map.
// Parts of rule alternatives can be assigned to variables (ex. v)
// that are used to refer to them in rule actions.
// Alternatively rule names (ex. NAME) can be used.
// We could have used $value in place of $v below.
assign: ^(ASSIGN NAME v=value) { variableMap.put($NAME.text, $v.result); };

// This returns a value as a double.
// The value can be a number, a variable name or a function evaluation.
value returns [double result]
  : NUMBER { $result = toDouble($NUMBER); }
  | NAME { $result = getVariable($NAME); }
  | functionEval { $result = $functionEval.result; }
  ;

// This returns the result of a function evaluation as a double.
functionEval returns [double result]
  : ^(FUNCTION fn=NAME v=NUMBER) {
      $result = evalFunction($fn, toDouble($v));
    }
  | ^(FUNCTION fn=NAME v=NAME) {
      $result = evalFunction($fn, getVariable($v));
```

```
    }
  ;

// This builds a Function object and adds it to the function map.
define
  : ^(DEFINE name=NAME variable=NAME polynomial) {
      define(new Function($name.text, $variable.text, $polynomial.result));
    }
  ;

// This builds a Polynomial object and returns it.
polynomial returns [Polynomial result]
// The "current" attribute in this rule scope is visible to
// rules invoked by this one, such as term.
scope { Polynomial current; }
@init { $polynomial::current = new Polynomial(); }
  // There can be no sign in front of the first term,
  // so "" is passed to the term rule.
  // The coefficient of the first term can be negative.
  // The sign between terms is passed to
  // subsequent invocations of the term rule.
  : ^(POLYNOMIAL term[""] (s=SIGN term[$s.text])*) {
      $result = $polynomial::current;
    }
  ;

// This builds a Term object and adds it to the current Polynomial.
term[String sign]
@init { boolean negate = "-".equals(sign); }
  : ^(TERM coefficient=NUMBER) {
      double c = toDouble($coefficient);
      if (negate) c = -c; // applies sign to coefficient
      $polynomial::current.addTerm(new Term(c));
    }
  | ^(TERM coefficient=NUMBER? variable=NAME exponent=NUMBER?) {
      double c = coefficient == null ? 1.0 : toDouble($coefficient);
      if (negate) c = -c; // applies sign to coefficient
      double exp = exponent == null ? 1.0 : toDouble($exponent);
      $polynomial::current.addTerm(new Term(c, $variable.text, exp));
    }
  ;

// This outputs help on our language which is useful in interactive mode.
help
  : HELP {
      outln("In the help below");
      outln("* fn stands for function name");
      outln("* n stands for a number");
      outln("* v stands for variable");
      outln("");
      outln("To define");
      outln("* a variable: v = n");
      outln("* a function from a polynomial: fn(v) = polynomial-terms");
      outln("  (for example, f(x) = 3x^2 - 4x + 1)");
      outln("* a function from adding or subtracting two others: " +
        "fn3 = fn1 +|- fn2");
      outln("  (for example, h = f + g)");
      outln("");
      outln("To print");
      outln("* a literal string: print \"text\"");
      outln("* a number: print n");
      outln("* the evaluation of a function: print fn(n | v)");
      outln("* the defintion of a function: print fn()");
      outln("* the derivative of a function: print fn'()");
      outln("* multiple items on the same line: print i1 i2 ... in");
      outln("");
      outln("To list");
      outln("* variables defined: list variables");
      outln("* functions defined: list functions");
```

```
      outln("");
      outln("To get help: help or ?");
      outln("");
      outln("To exit: exit or quit");
    }
  ;

// This lists all the functions or variables that are currently defined.
list
  : ^(LIST FUNCTIONS) {
      outln("# of functions defined: " + functionMap.size());
      for (Function function : functionMap.values()) {
        outln(function);
      }
    }
  | ^(LIST VARIABLES) {
      outln("# of variables defined: " + variableMap.size());
      for (String name : variableMap.keySet()) {
        double value = variableMap.get(name);
        outln(name + " = " + value);
      }
    }
  ;

// This adds or substracts two functions to create a new one.
combine
  : ^(COMBINE fn1=NAME op=SIGN fn2=NAME fn3=NAME) {
      Function f2 = getFunction(fn2);
      Function f3 = getFunction(fn3);
      if ("+".equals($op.text)) {
        // "$fn1.text" is the name of the new function to create.
        define(f2.add($fn1.text, f3));
      } else if ("-".equals($op.text)) {
        define(f2.subtract($fn1.text, f3));
      } else {
        // This should never happen since SIGN is defined to be either "+" or "-".
        throw new RuntimeException(
          "The operator \"" + $op +
          " cannot be used for combining functions.");
      }
    }
  ;

// This prints a list of printTargets then prints a newline.
print
  : ^(PRINT printTarget*)
    { System.out.println(); };

// This prints a single printTarget without a newline.
// "out", "unescape", "getVariable", "getFunction", "evalFunction"
// and "toDouble" are methods we wrote that were defined
// in the @members block earlier.
printTarget
  : NUMBER { out($NUMBER); }
  | STRING_LITERAL {
      String s = unescape($STRING_LITERAL.text);
      out(s.substring(1, s.length() - 1)); // removes quotes
    }
  | NAME { out(getVariable($NAME)); }
  | ^(FUNCTION NAME) { out(getFunction($NAME)); }
    // The next line uses the return value named "result"
    // from the earlier rule named "functionEval".
  | functionEval { out($functionEval.result); }
  | derivative
  ;

// This prints the derivative of a function.
// This also could have been done in place in the printTarget rule.
derivative
```

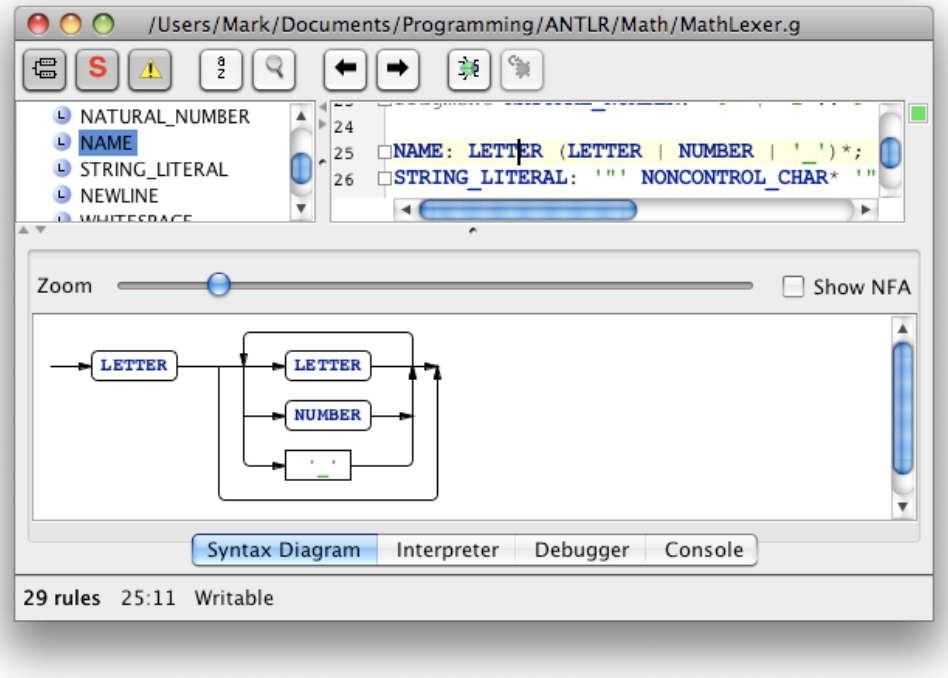
```
  : ^(DERIVATIVE NAME) {
      out(getFunction($NAME).getDerivative());
    }
  ;
```

Part VI - ANTLRWorks

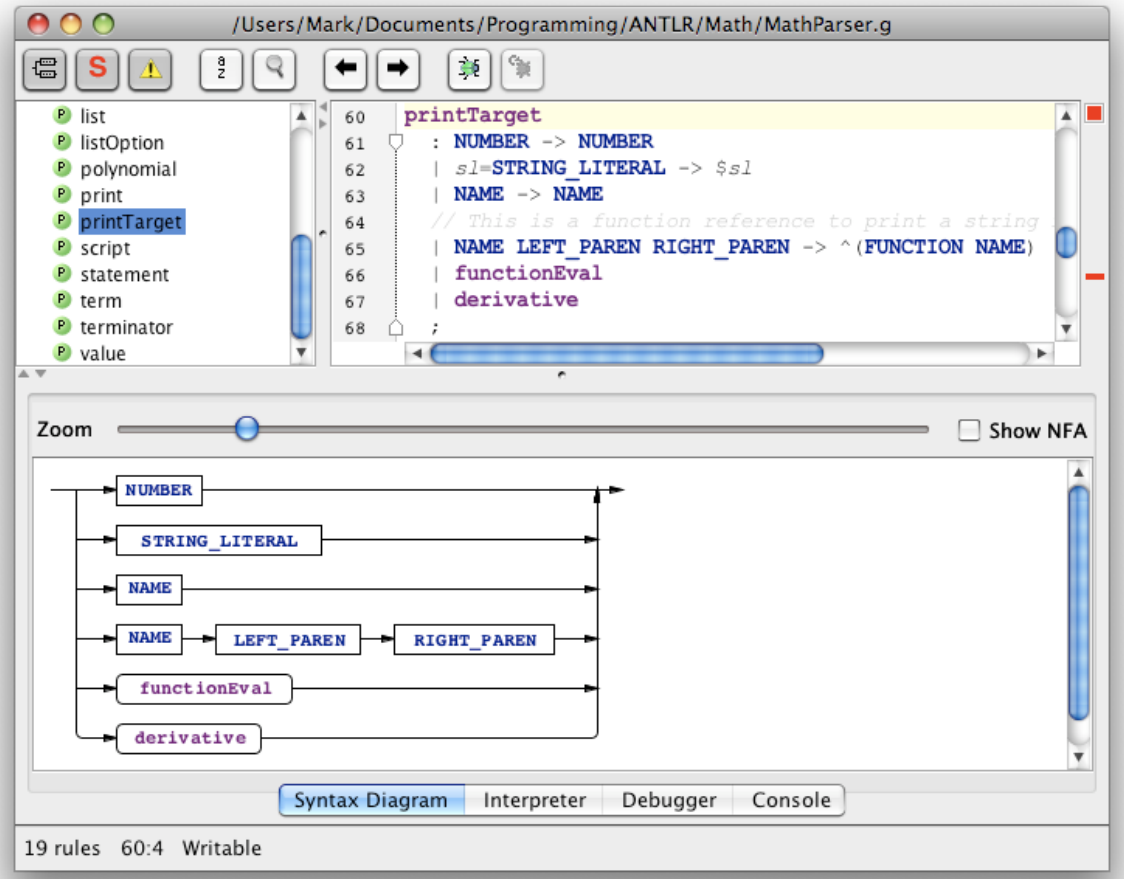
ANTLRWorks is a graphical grammar editor and debugger. It checks for grammar errors, including those beyond the syntax variety such as conflicting rule alternatives, and highlights them. It can display a syntax diagram for a selected rule. It provides a debugger that can step through creation of parse trees and ASTs.

Rectangles in syntax diagrams correspond to fixed vocabulary symbols. Rounded rectangles correspond to variable symbols.

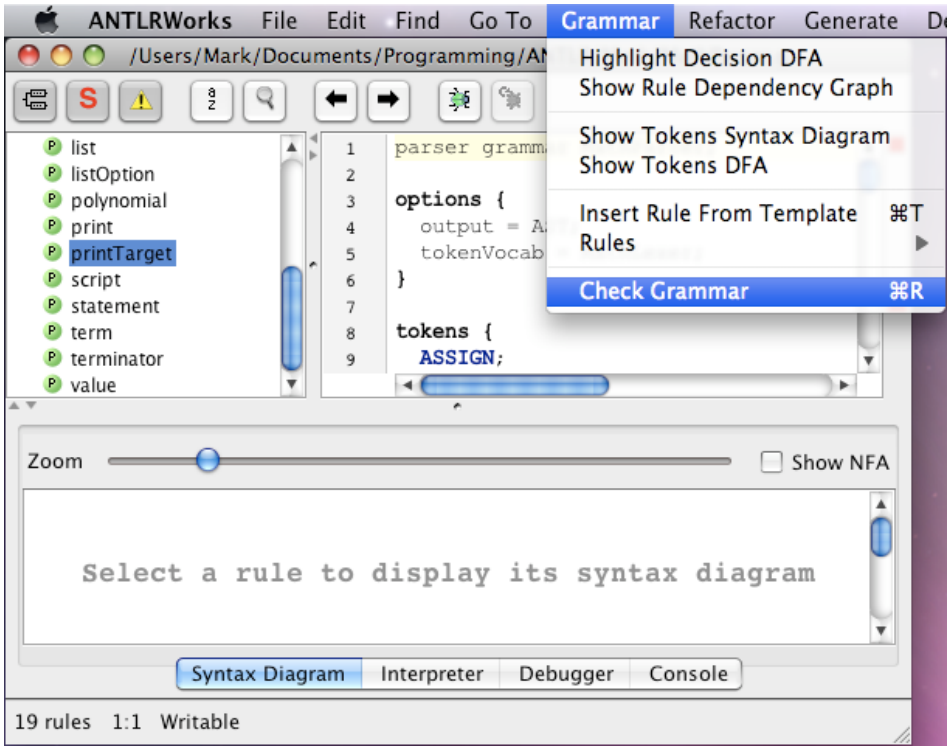
Here's an example of a syntax diagram for a selected lexer rule.

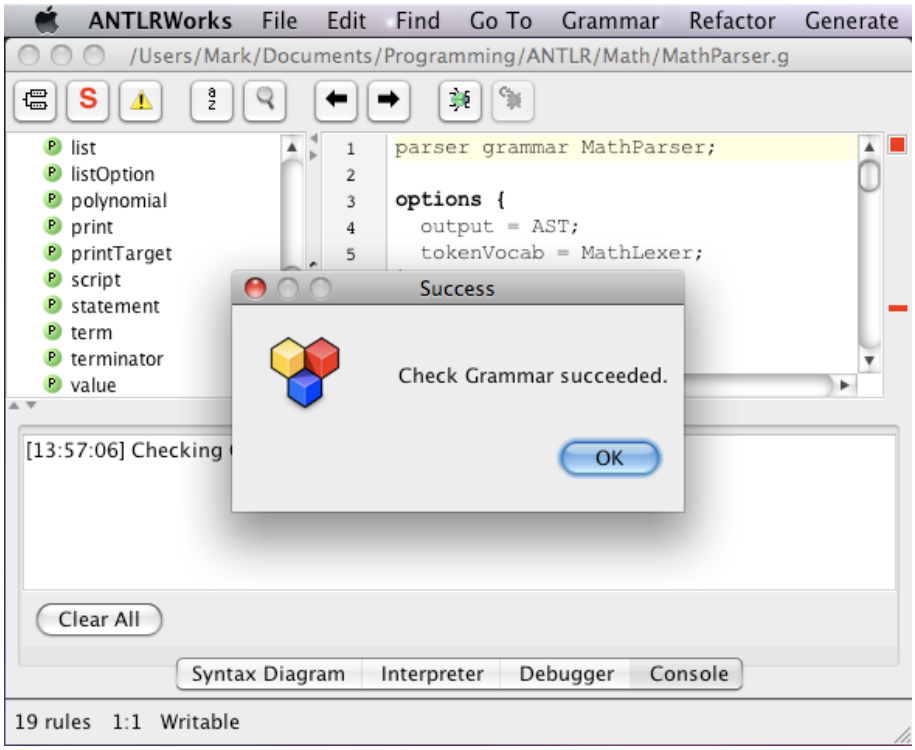


Here's an example of a syntax diagram for a selected parser rule.

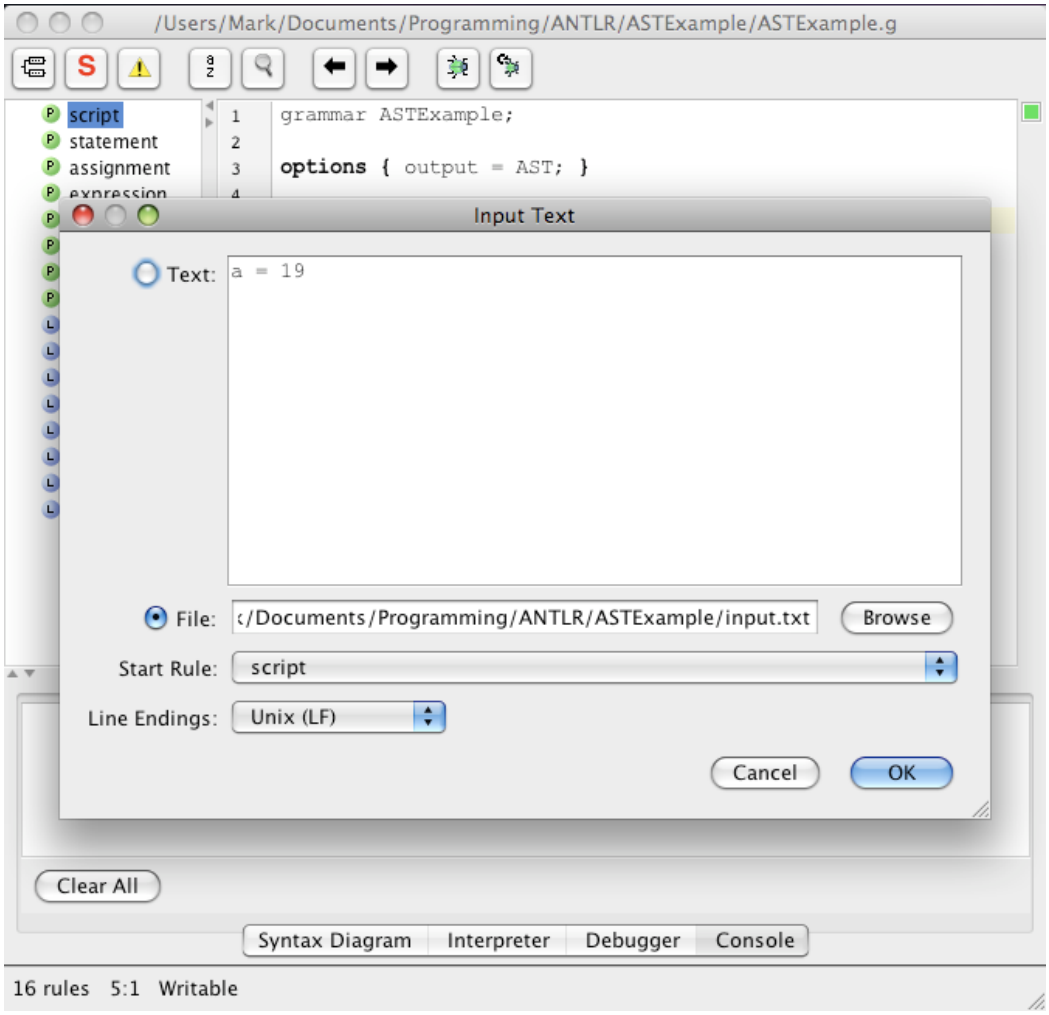


Here's an example of requesting a grammar check, followed by a successful result.

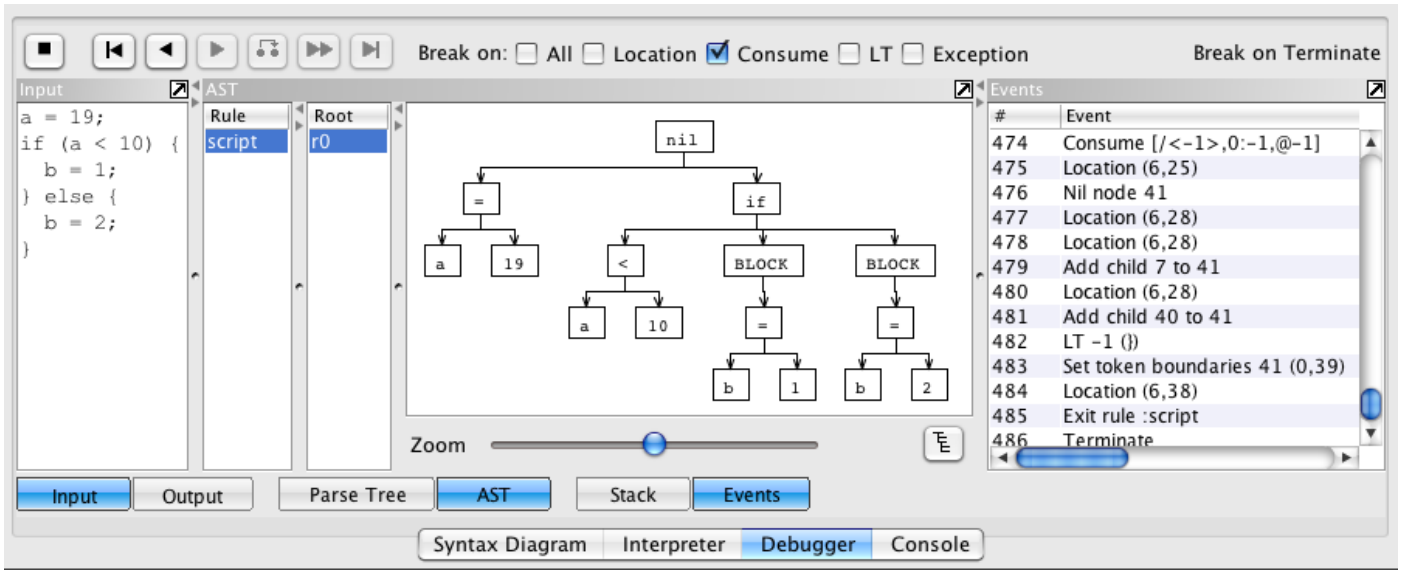




Using the ANTLRWorks debugger is simple when the lexer and parser rules are combined in a single grammar file, unlike our example. Press the Debug toolbar button (with a bug on it), enter input text or select an input file, select the start rule (allows debugging a subset of the grammar) and press the OK button. Here's an example of entering the input for a different, simpler grammar that defines the lexer and parser rules in a single file:



The debugger controls and output are displayed at the bottom of the ANTLRWorks window. Here's an example using that same, simpler grammar:



Using the debugger when the lexer and parser rules are in separate files, like in our example, is a bit more complicated. See the ANTLR Wiki page titled "When do I need to use remote debugging."

Part VII - Putting It All Together

Using Generated Classes

Next we need to write a class to utilize the classes generated by ANTLR. We'll call ours Processor. This class will use MathLexer (extends Lexer), MathParser (extends Parser) and MathTree (extends TreeParser). Note that the classes Lexer, Parser and TreeParser all extend the class BaseRecognizer. Our Processor class will also use other classes we wrote to model our domain. These classes are named Term, Function and Polynomial. We'll support two modes of operation, batch and interactive.

Here's our Processor class.

```
package com.ociweb.math;

import java.io.*;
import java.util.Scanner;
import org.antlr.runtime.*;
import org.antlr.runtime.tree.*;

public class Processor {

    public static void main(String[] args) throws IOException, RecognitionException {
        if (args.length == 0) {
            new Processor().processInteractive();
        } else if (args.length == 1) { // name of file to process was passed in
            new Processor().processFile(args[0]);
        } else { // more than one command-line argument
            System.err.println("usage: java com.ociweb.math.Processor [file-name]");
        }
    }

    private void processFile(String filePath) throws IOException, RecognitionException {
        CommonTree ast = getAST(new FileReader(filePath));
        //System.out.println(ast.toStringTree()); // for debugging
        processAST(ast);
    }

    private CommonTree getAST(Reader reader) throws IOException, RecognitionException {
        MathParser tokenParser = new MathParser(getTokenStream(reader));
        MathParser.script_return parserResult =
            tokenParser.script(); // start rule method
        reader.close();
        return (CommonTree) parserResult.getTree();
    }

    private CommonTokenStream getTokenStream(Reader reader) throws IOException {
        MathLexer lexer = new MathLexer(new ANTLRReaderStream(reader));
        return new CommonTokenStream(lexer);
    }

    private void processAST(CommonTree ast) throws RecognitionException {
        MathTree treeParser = new MathTree(new CommonTreeNodeStream(ast));
        treeParser.script(); // start rule method
    }

    private void processInteractive() throws IOException, RecognitionException {
        MathTree treeParser = new MathTree(null); // a TreeNodeStream will be assigned later
        Scanner scanner = new Scanner(System.in);

        while (true) {
            System.out.print("math> ");
            String line = scanner.nextLine().trim();
            if ("quit".equals(line) || "exit".equals(line)) break;
            processLine(treeParser, line);
        }
    }

    // Note that we can't create a new instance of MathTree for each
    // line processed because it maintains the variable and function Maps.
    private void processLine(MathTree treeParser, String line) throws RecognitionException {
```

```
        // Run the lexer and token parser on the line.
        MathLexer lexer = new MathLexer(new ANTLRStringStream(line));
        MathParser tokenParser = new MathParser(new CommonTokenStream(lexer));
        MathParser.statement_return parserResult =
            tokenParser.statement(); // start rule method

        // Use the token parser to retrieve the AST.
        CommonTree ast = (CommonTree) parserResult.getTree();
        if (ast == null) return; // line is empty

        // Use the tree parser to process the AST.
        treeParser.setTreeNodeStream(new CommonTreeNodeStream(ast));
        treeParser.statement(); // start rule method
    }

} // end of Processor class
```

Ant Tips

Ant is a great tool for automating tasks used to develop and test grammars. Suggested independent "targets" include the following.

- Use `org.antlr.Tool` to generate Java classes and ".tokens" files from each grammar file.
  - ".tokens" files assign integer constants to token names and are used by `org.antlr.Tool` when processing subsequent grammar files.
  - The "uptodate" task can be used to determine whether the grammar has changed since the last build.
  - The "unless" target attribute can be used to avoid running `org.antlr.Tool` if the grammar hasn't changed since the last build.
- Compile Java source files.
- Run automated tests.
- Run the application using a specific file as input.
- Delete all generated files (clean target).

For examples of all of these, download the source code from the URL listed at the end of this article and see the build.xml file.

Part VIII - Wrap Up

Hidden Tokens

By default the parser only processes tokens from the default channel. It can however request tokens from other channels such as the hidden channel. Tokens are assigned unique, sequential indexes regardless of the channel to which they are written. This allows parser code to determine the order in which the tokens were encountered, regardless of the channel to which they were written.

Here are some related public constants and methods from the Token class.

- `static final int DEFAULT_CHANNEL`
- `static final int HIDDEN_CHANNEL`
- `int getChannel()` This gets the number of the channel where this Token was written.
- `int getTokenIndex()` This gets the index of this Token.

Here are some related public methods from the CommonTokenStream class, which implements the TokenStream interface.

- `Token get(int index)` This gets the Token found at a given position in the input.
- `List getTokens(int start, int stop)` This gets a List of Tokens found between given positions in the input.
- `int index()` This gets the index of the last Token that was read.

Advanced Topics

We have demonstrated the basics of using ANTLR. For information on advanced topics, see the slides from the presentation on which this article was based at http://www.ociweb.com/mark/programming/ANTLR3.html. This

web page contains links to the slides and the code presented in this article. The advanced topics covered in these slides include the following.

- remote debugging
- using the StringTemplate library
- details on the use of lookahead in grammars
- three kinds semantic predicates: validating, gated and disambiguating
- syntactic predicates
- customizing error handling
- gUnit grammar unit testing framework

Projects Using ANTLR

Many programming languages have been implemented using ANTLR. These include Boo, Groovy, Mantra, Nemerle and XRuby.

Many other kinds of tools use ANTLR in their implementation. These include Hibernate (for its HQL to SQL query translator), Intellij IDEA, Jazillian (translates COBOL, C and C++ to Java), JBoss Rules (was Drools), Keynote (from Apple), WebLogic (from Oracle), and many more.

Books

Currrently only one book on ANTLR is available. Terence Parr, creator of ANTLR, wrote "The Definitive ANTLR Reference" It is published by "The Pragmatic Programmers." Terence is working on a second book for the same publisher that may be titled "ANTLR Recipes."

Summary

There you have it! ANTLR is a great tool for generating custom language parsers. We hope this article will make it easier to get started creating validators, processors and translators.

References

- ANTLR - http://www.anltr.org/
- ANTLRWorks - http://www.antlr.org/works/
- StringTemplate - http://www.stringtemplate.org/
- Our slides and code examples - http://www.ociweb.com/mark/
  (look for "ANTLR 3")

OCI Educational Services

OCI is the leading provider of Object Oriented technology training in the Midwest. More than 3,000 students participated in our training program over the last 12 months. Targeted toward Software Engineers and the development community, our extensive program of over 50 hands-on workshops is delivered to corporations and individuals throughout the U.S. and internationally. OCI's Educational Services include Group Training events and Open Enrollment classes.

Course Catalog | Open Enrollment Schedule

For further information regarding OCI's Educational Services programs, please visit our Educational Services section on the web or contact us at training@ociweb.com.

Object Computing, Inc. is a Sun Authorized Java Center in St. Louis, MO and a Member of the Object Management Group, OMG. OCI specializes in distributed computing using object-oriented and web-enabled technologies and provides Consulting, Education and Product Development services to clients nation-wide. For more information contact us in St. Louis, MO (314)579-0066, Tempe, AZ (480)752-0042 or email info@ociweb.com.

Inquiries regarding **Career Opportunities** can be directed to: hr@ociweb.com.

The ***Java News Brief*** is a monthly newsletter. The purpose and intent of this publication is to advance Java, provide technical value and announce available OCI educational services. To subscribe or unsubscribe

Top