ANTLR 3

Mark Volkmann
mark@ociweb.com
Object Computing, Inc.
2008

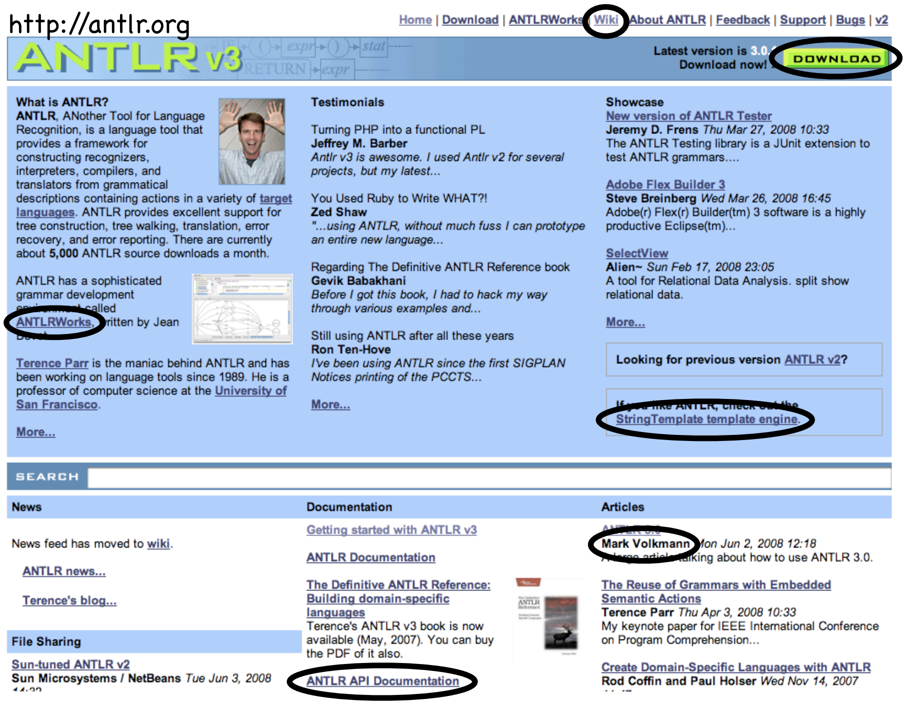
ANTLR Overview

- ANother Tool for Language Recognition
  - written by Terence Parr in Java
- Easier to use than most/all similar tools
- Supported by ANTLRWorks
  - graphical grammar editor and debugger
  - written by Jean Bovet using Swing
- Used to implement
  - "real" programming languages
  - domain-specific languages (DSLs)
- http://www.antlr.org
  - download ANTLR and ANTLRWorks here
  - both are free and open source
  - docs, articles, wiki, mailing list, examples

I'm a professor at the University of San Francisco.
Ter
I worked with Ter as a masters student there.
Jean

ANTLR Documentation

http://antlr.org



Home | Download | ANTLRWorks | Wiki | About ANTLR | Feedback | Support | Bugs | v2

ANTLR v3

Latest version is 3.0.
Download now!

DOWNLOAD

What is ANTLR?
ANTLR, ANother Tool for Language Recognition, is a language tool that provides a framework for constructing recognizers, interpreters, compilers, and translators from grammatical descriptions containing actions in a variety of target languages. ANTLR provides excellent support for tree construction, tree walking, translation, error recovery, and error reporting. There are currently about **5,000** ANTLR source downloads a month.

ANTLR has a sophisticated grammar development environment called ANTLRWorks, written by Jean Bovet.

Terence Parr is the maniac behind ANTLR and has been working on language tools since 1989. He is a professor of computer science at the University of San Francisco.

More...

Testimonials

Turning PHP into a functional PL
Jeffrey M. Barber
Antlr v3 is awesome. I used Antlr v2 for several projects, but my latest...

You Used Ruby to Write WHAT?!
Zed Shaw
"...using ANTLR, without much fuss I can prototype an entire new language...

Regarding The Definitive ANTLR Reference book
Gevik Babakhani
Before I got this book, I had to hack my way through various examples and...

Still using ANTLR after all these years
Ron Ten-Hove
I've been using ANTLR since the first SIGPLAN Notices printing of the PCCTS...

More...

Showcase
New version of ANTLR Tester
Jeremy D. Frens *Thu Mar 27, 2008 10:33*
The ANTLR Testing library is a JUnit extension to test ANTLR grammars....

Adobe Flex Builder 3
Steve Breinberg *Wed Mar 26, 2008 16:45*
Adobe(r) Flex(r) Builder(tm) 3 software is a highly productive Eclipse(tm)...

SelectView
Alien~ *Sun Feb 17, 2008 23:05*
A tool for Relational Data Analysis. split show relational data.

More...

Looking for previous version ANTLR v2?

If you like ANTLR, check out the StringTemplate template engine.

SEARCH

News

News feed has moved to wiki.

ANTLR news...

Terence's blog...

File Sharing

Sun-tuned ANTLR v2
Sun Microsystems / NetBeans *Tue Jun 3, 2008 14:22*

Documentation

Getting started with ANTLR v3

ANTLR Documentation

The Definitive ANTLR Reference: Building domain-specific languages
Terence's ANTLR v3 book is now available (May, 2007). You can buy the PDF of it also.

ANTLR API Documentation

Articles

ANTLR 3.0
Mark Volkmann *Mon Jun 2, 2008 12:18*
A large article talking about how to use ANTLR 3.0.

The Reuse of Grammars with Embedded Semantic Actions
Terence Parr *Thu Apr 3, 2008 10:33*
My keynote paper for IEEE International Conference on Program Comprehension...

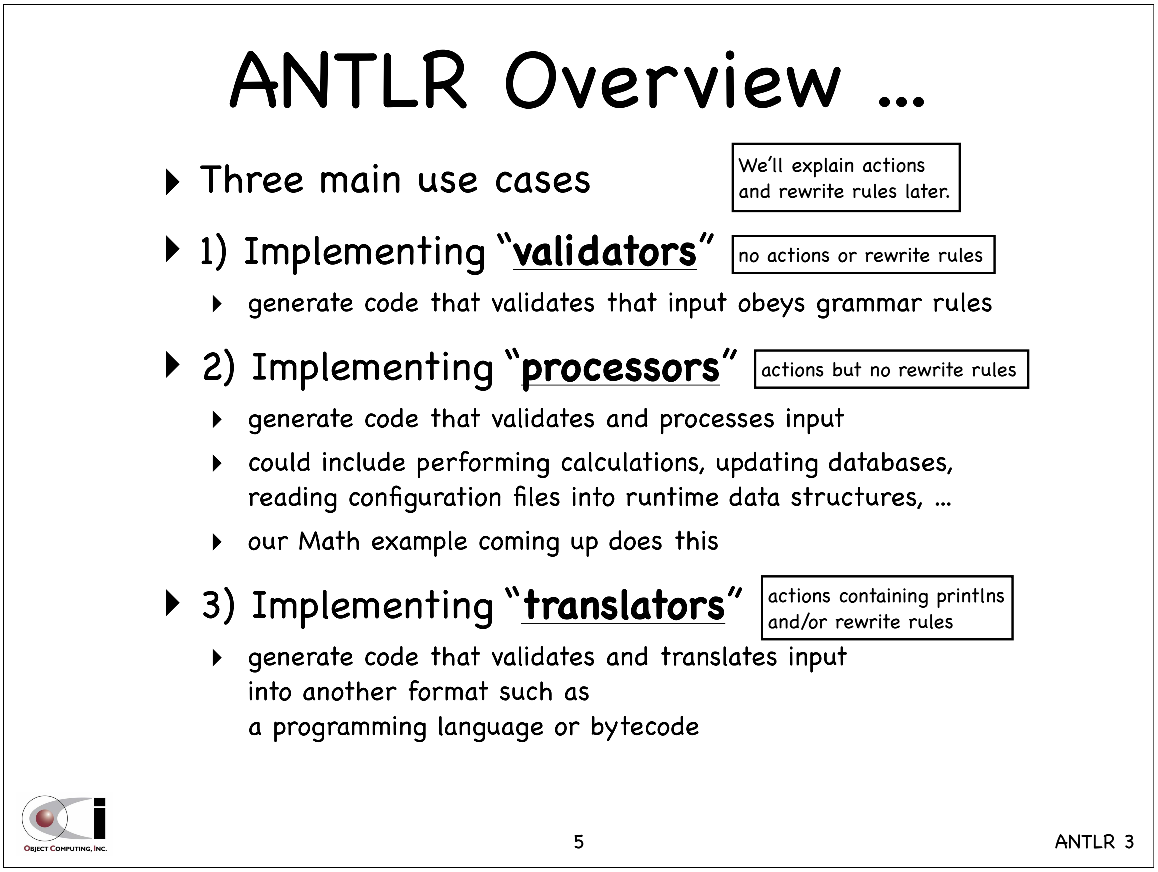
Create Domain-Specific Languages with ANTLR
Rod Coffin and Paul Holser *Wed Nov 14, 2007 11:47*

ANTLR Overview ...

- Uses EBNF grammars
  - Extended Backus-Naur Form
  - can directly express optional and repeated elements
  - supports subrules (parenthesized groups of elements)

BNF grammars require more verbose syntax to express these.

- Supports many target languages for generated code
  - Java, Ruby, Python, Objective-C, C, C++ and C#
- Provides infinite lookahead
  - most parser generators don't
  - used to choose between rule alternatives
- Plug-ins available for IDEA and Eclipse



ANTLR Overview ...

- Three main use cases

We'll explain actions and rewrite rules later.

- 1) Implementing "**validators**"

no actions or rewrite rules

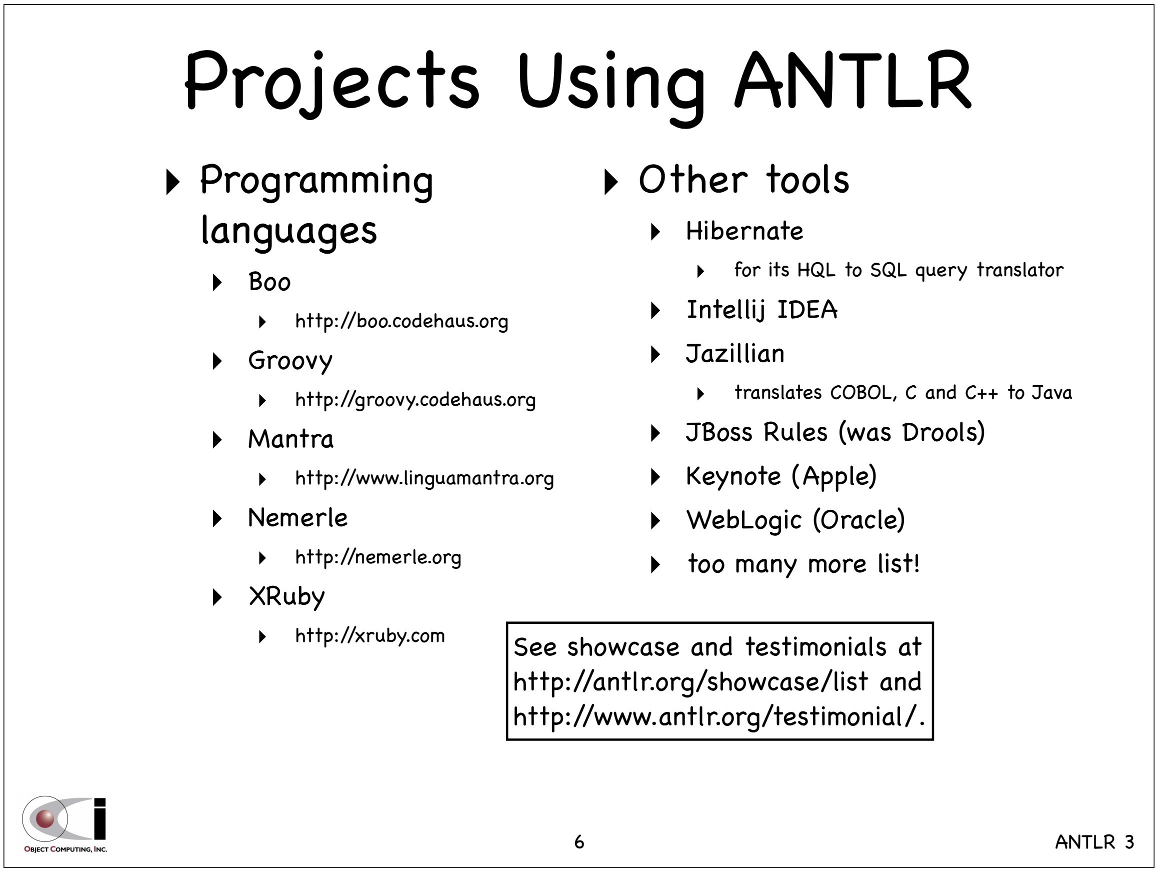
  - generate code that validates that input obeys grammar rules
- 2) Implementing "**processors**"

actions but no rewrite rules

  - generate code that validates and processes input
  - could include performing calculations, updating databases, reading configuration files into runtime data structures, ...
  - our Math example coming up does this
- 3) Implementing "**translators**"

actions containing printlns and/or rewrite rules

  - generate code that validates and translates input into another format such as a programming language or bytecode

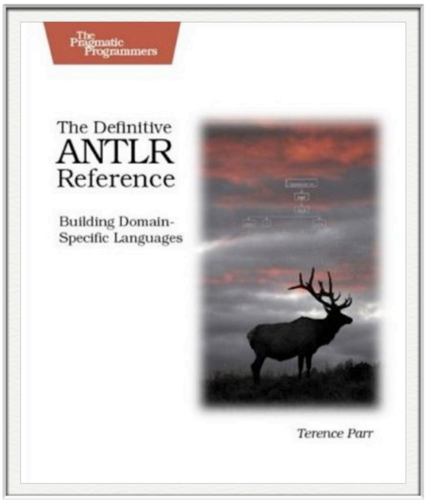


Projects Using ANTLR

- Programming languages
  - Boo
    - http://boo.codehaus.org
  - Groovy
    - http://groovy.codehaus.org
  - Mantra
    - http://www.linguamantra.org
  - Nemerle
    - http://nemerle.org
  - XRuby
    - http://xruby.com
- Other tools
  - Hibernate
    - for its HQL to SQL query translator
  - Intellij IDEA
  - Jazillian
    - translates COBOL, C and C++ to Java
  - JBoss Rules (was Drools)
  - Keynote (Apple)
  - WebLogic (Oracle)
  - too many more list!

See showcase and testimonials at http://antlr.org/showcase/list and http://www.antlr.org/testimonial/.

Books


The Pragmatic Programmers
The Definitive ANTLR Reference
Building Domain-Specific Languages
Terence Parr

- "ANTLR Recipes"? in the works
  - another Pragmatic Programmers book from Terence Parr

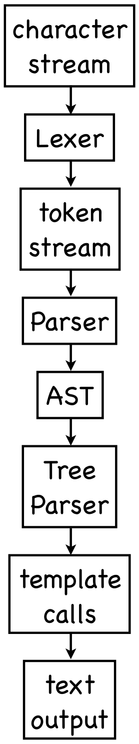
Other DSL Approaches

- Languages like Ruby and Groovy are good at implementing DSLs, but ...
- The DSLs have to live within the syntax rules of the language
- For example
  - dots between object references and method names
  - parameters separated by commas
  - blocks of code surrounded by { ... } or do ... end
- What if you don't want these in your language?

Conventions

- ANTLR grammar syntax makes frequent use of the characters [ ] and { }
- In these slides
  - when describing a placeholder, I'll use italics
  - when describing something that's optional, I'll use **`item`**?

Some Definitions



- Lexer
  - converts a stream of characters to a stream of tokens
- Parser
  - processes a stream of tokens, possibly creating an AST

Token objects know their start/stop character stream index, line number, index within the line, and more.

- Abstract Syntax Tree (AST)
  - an intermediate tree representation of the parsed input that
    - is simpler to process than the stream of tokens
    - can be efficiently processed multiple times
- Tree Parser
  - processes an AST
- StringTemplate
  - a library that supports using templates with placeholders for outputting text (for example, Java source code)

General Steps

- Write grammar
  - can be in one or more files
- Optionally write StringTemplate templates
- Debug grammar with ANTLRWorks
- Generate classes from grammar
  - these validate that text input conforms to the grammar and execute target language "actions" specified in the grammar
- Write application that uses generated classes
- Feed the application text that conforms to the grammar

Let's Create A Language!

- Features
  - run on a file or interactively
  - get help - **`? or help`**
  - one data type, double
  - assign values to variables - **`a = 3.14`**
  - define polynomial functions - **`f(x) = 3x^2 - 4x + 2`**
  - print strings, numbers, variables and function evaluations - **`print "The value of f for " a " is " f(a)`**
  - print the definition of a function and its derivative - **`print "The derivative of " f() " is " f'()`**
  - list variables and functions - **`list variables`** and **`list functions`**
  - add/subtract functions - **`h = f - g`**
    - the function variables don't have to match
  - exit - **`exit`** or **`quit`**

```
Input:
f(x) = 3x^2 - 4
g(y) = y^2 - 2y + 1
h = f - g
print h()

Output:
h(x) = 2x^2 + 2x - 5
```

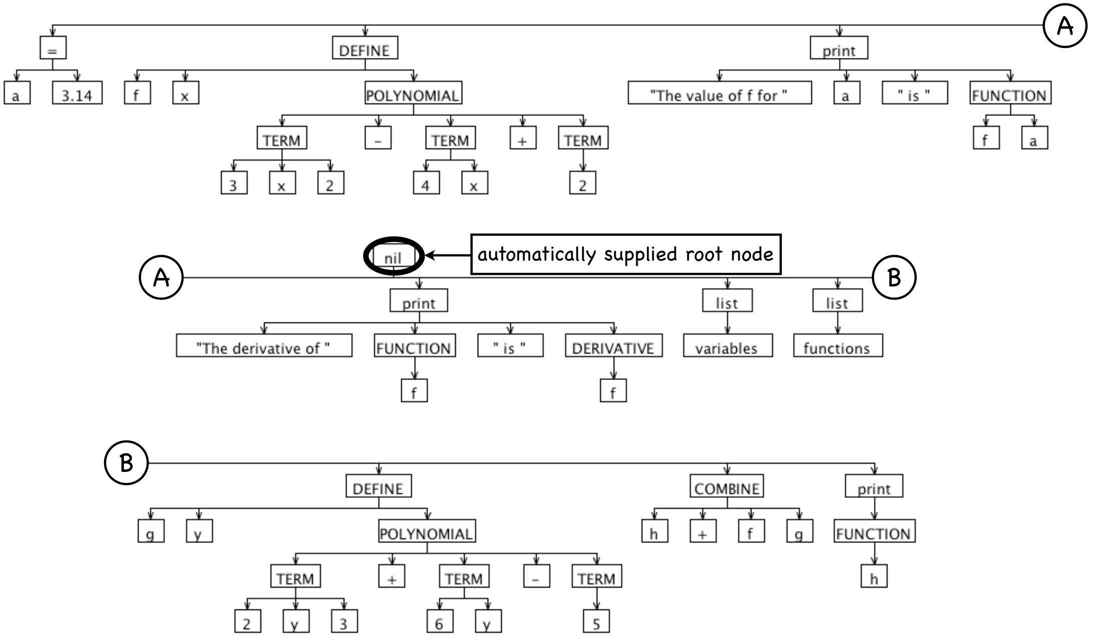
Example Input/Output

```
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()
```

```
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
```

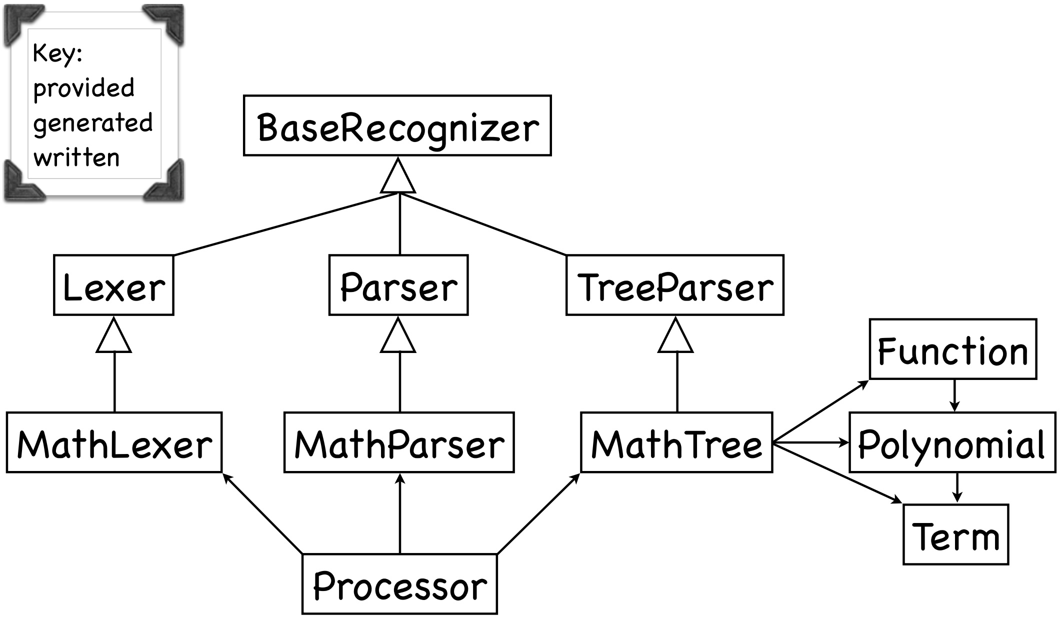
Example AST

drawn by ANTLRWorks



automatically supplied root node

Important Classes


Key:
provided
generated
written

BaseRecognizer

Lexer

Parser

TreeParser

MathLexer

MathParser

MathTree

Function

Polynomial

Term

Processor

Grammar Actions

- Add to the generated code
- **@*grammar-type*::header { ... }**
  - inserts contained code before the class definition
    - commonly used to specify a package name and import classes in other packages
- **@*grammar-type*::members { ... }**
  - inserts field declarations and methods inside the class definition
  - commonly used to
    - define constants and attributes accessible to all rule methods in the generated class
    - define methods used by multiple rule actions
    - override methods in the superclasses of the generated classes
      - useful for customizing error reporting and handling

grammar-type must be lexer, parser (the default) or treeparser

Lexer Rules

- Need one for every kind of token to be processed in parser grammar
- Name must start uppercase
  - typically all uppercase
- Assign a token name to
  - a single literal string found in input
  - a selection of literal strings found in input
  - one or more characters and ranges of characters
    - can use cardinality indicators ?, * and +
- Can refer to other lexer rules
- "fragment" lexer rules
  - do not result in tokens
  - are only referenced by other lexer rules

Regular expressions aren't supported.

The next lexer rule used is the one that matches the most characters. If there is a tie, the one listed first is used, so order matters!

See LETTER and DIGIT rules in the upcoming example.

Whitespace & Comments

- Handled in lexer rules
- Two common options
  - throw away - **`skip();`**
  - write to a different "channel" - **`$channel = HIDDEN;`**

The ANTLRWorks debugger input panel doesn't display skipped characters, but does display hidden ones.

constant defined in BaseRecognizer

- Examples

```
WHITESPACE: (' ' | '\t')+ { $channel = HIDDEN; };
NEWLINE: ('\r'? '\n')+;
SINGLE_COMMENT: '//' ~('\r' | '\n')* NEWLINE { skip(); };
MULTI_COMMENT
options { greedy = false; }
  : '/*' .* '*/' NEWLINE? { skip(); };
```

Don't skip or hide NEWLINEs if they are used as statement terminators.

The greedy option defaults to true, except for the patterns .* and .+, so it doesn't need to be specified here. When true, the lexer matches as much input as possible. When false, it stops when input matches the next element.

Our Lexer Grammar

```
lexer grammar MathLexer;

@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'*;
```

We want the generated lexer class to be in this package.

Our Lexer Grammar ...

```
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 omits the double-quote character,
// digits, uppercase letters and lowercase letters.
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; };
```

Token Specification

- The lexer creates tokens for all input character sequences that match lexer rules
- It can be useful to create other tokens that
  - don’t exist in the input (imaginary)
    - often serve to group other tokens
  - have a better name than is found in the input
- Do this with a token specification in the parser grammar

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

See all the uppercase token names in the AST diagram on slide 14.

We need this for the imaginary tokens DEFINE, POLYNOMIAL, TERM, FUNCTION, DERIVATIVE and COMBINE.

Rule Syntax

only for lexer rules

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

add code before and/or after code in the generated method for this rule

Each element in these alternative sequences can be followed by an action which is target language code in curly braces. The code is executed immediately after a preceding element is matched by input.

to customize exception handling for this rule

Creating ASTs

- Requires grammar option **`output = AST;`**
- Approach #1 - Rewrite rules
  - appear after a rule alternative
  - the recommended approach in most cases
  - **`-> ^(parent child-1 child-2 ... child-n)`**

can't use both approaches in the same rule alternative!

- Approach #2 - AST operators
  - appear in a rule alternative, immediately after tokens
  - works best for sequences like mathematical expressions
  - operators
    - ^ - make new root node for all child nodes at the same level
    - none - make a child node of current root node
    - ! - don't create a node ←
  - parent^ '('! child-1 child-2 ... child-n ')'!

often used for bits of syntax that aren't needed in the AST such as parentheses, commas and semicolons

Declaring Rule Arguments and Return Values

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

arguments

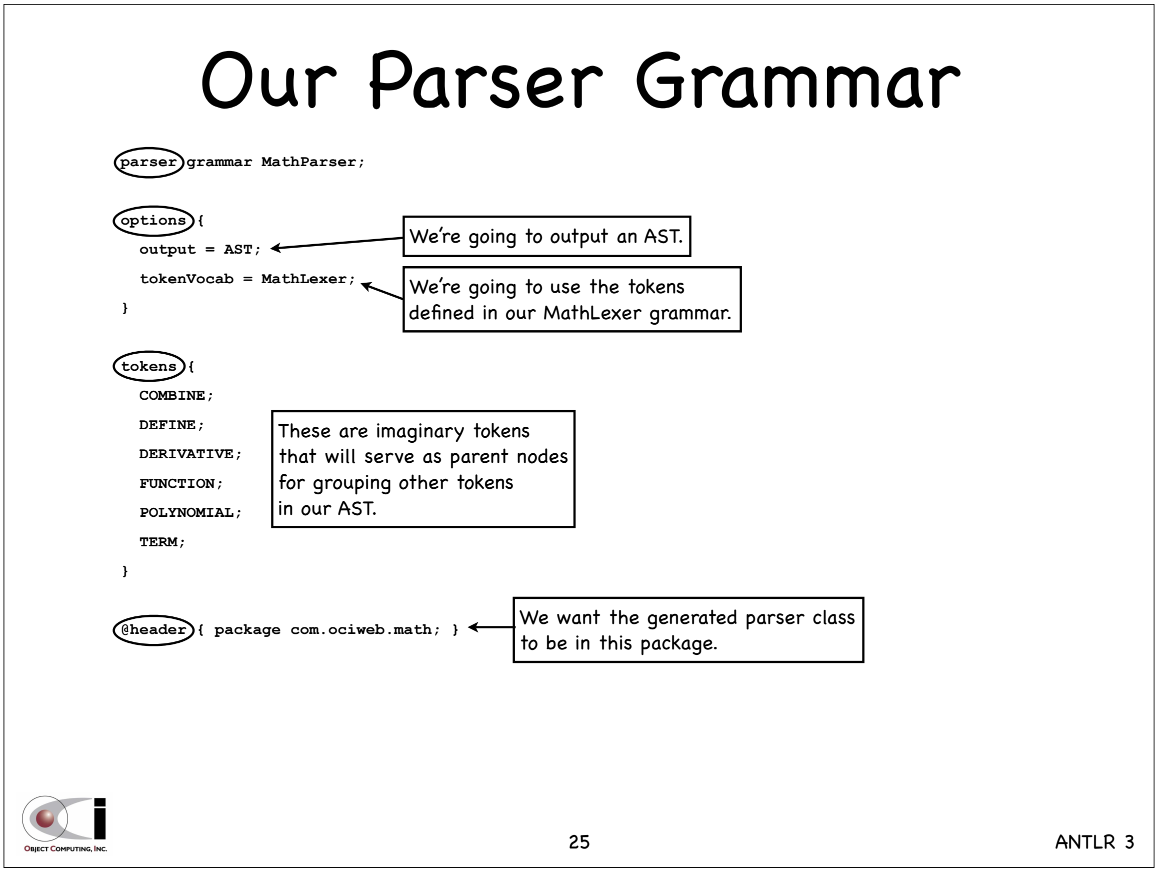
return values;
can have more than one

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

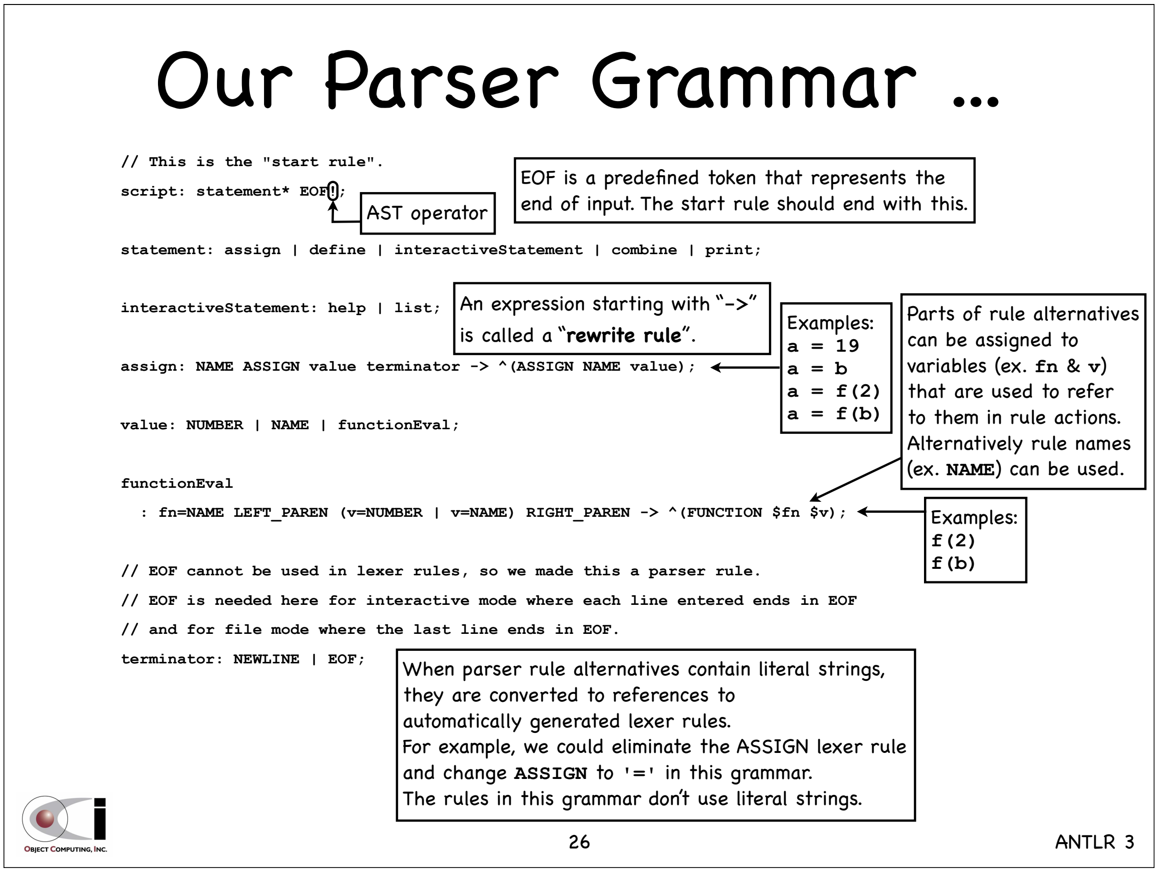
```
parser grammar MathParser;

options {
  output = AST;
  tokenVocab = MathLexer;
}

tokens {
  COMBINE;
  DEFINE;
  DERIVATIVE;
  FUNCTION;
  POLYNOMIAL;
  TERM;
}

@header { package com.ociweb.math; }
```

We're going to output an AST.

We're going to use the tokens defined in our MathLexer grammar.

These are imaginary tokens that will serve as parent nodes for grouping other tokens in our AST.

We want the generated parser class to be in this package.



Our Parser Grammar ...

```
// This is the "start rule".
script: statement* EOF!;

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

interactiveStatement: help | list;

assign: NAME ASSIGN value terminator -> ^(ASSIGN NAME value);

value: NUMBER | NAME | functionEval;

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;
```

AST operator

EOF is a predefined token that represents the end of input. The start rule should end with this.

An expression starting with "->" is called a "**rewrite rule**".

Examples:
a = 19
a = b
a = f(2)
a = f(b)

Parts of rule alternatives can be assigned to variables (ex. **fn** & **v**) that are used to refer to them in rule actions. Alternatively rule names (ex. **NAME**) can be used.

Examples:
f(2)
f(b)

When parser rule alternatives contain literal strings, they are converted to references to automatically generated lexer rules.
For example, we could eliminate the ASSIGN lexer rule and change **ASSIGN** to **'='** in this grammar.
The rules in this grammar don't use literal strings.

Our Parser Grammar ...

```
define
  : fn=NAME LEFT_PAREN fv=NAME RIGHT_PAREN ASSIGN
    polynomial[$fn.text, $fv.text] terminator
    -> ^(DEFINE $fn $fv polynomial);
```

Examples:
f(x) = 3x^2 - 4
g(y) = y^2 - 2y + 1

To get the text value from a variable that refers to a Token object, use "**$var.text**".

```
// fnt = function name text; fvt = function variable text
polynomial[String fnt, String fvt]
  : term[$fnt, $fvt] (SIGN term[$fnt, $fvt])*
    -> ^(POLYNOMIAL term (SIGN term)*);
```

Examples:
3x^2 - 4
y^2 - 2y + 1

Our Parser Grammar ...

```
// 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.
    { tv == null ? true : ($tv.text).equals($fvt) }?
    -> ^(TERM $c? $tv? $e?)
  ;
  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;

exponent: CARET NUMBER -> NUMBER;
```

Examples:
4
4x
x^2
4x^2

Term variables must match their function variable. This catches bad function definitions such as `f(x) = 2y`.

Example:
^2

Our Parser Grammar ...

```
help: HELP terminator -> HELP;
```

Examples:
```
?
help
```

```
list
  : LIST listOption terminator -> ^(LIST listOption);
```

Examples:
```
list functions
list variables
```

```
listOption: FUNCTIONS | VARIABLES;
```

Examples:
```
functions
variables
```

```
combine
  : fn1=NAME ASSIGN fn2=NAME op=SIGN fn3=NAME terminator
    -> ^(COMBINE $fn1 $op $fn2 $fn3);
```

Examples:
```
h = f + g
h = f - g
```

Our Parser Grammar ...

```
print
  : PRINT printTarget* terminator -> ^(PRINT printTarget*);
```

Example:
```
print "f(" a ") = " f(a)
```

```
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
  ;
```

Examples:
```
19
3.14
"my text"
a
f()
f(2)
f(a)
f'()
```

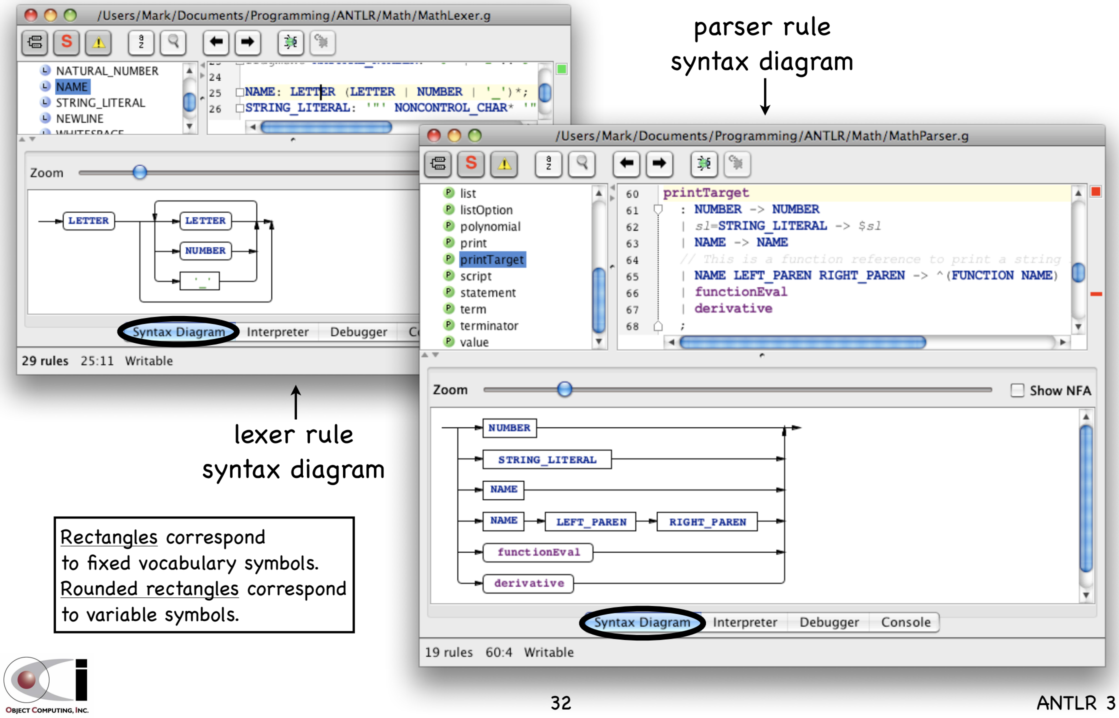
```
derivative
  : NAME APOSTROPHE LEFT_PAREN RIGHT_PAREN -> ^(DERIVATIVE NAME);
```

Example:
```
f'()
```

ANTLRWorks

- A graphical grammar editor and debugger
- Features
  - highlights grammar syntax errors
  - checks for grammar errors beyond the syntax variety
    - such as conflicting rule alternatives
  - displays a syntax diagram for the selected rule
  - debugger can step through creation of parse trees and ASTs

ANTLRWorks ...



```
NAME: LETTER (LETTER | NUMBER | '_')*;
STRING_LITERAL: '"' NONCONTROL_CHAR* '"
```

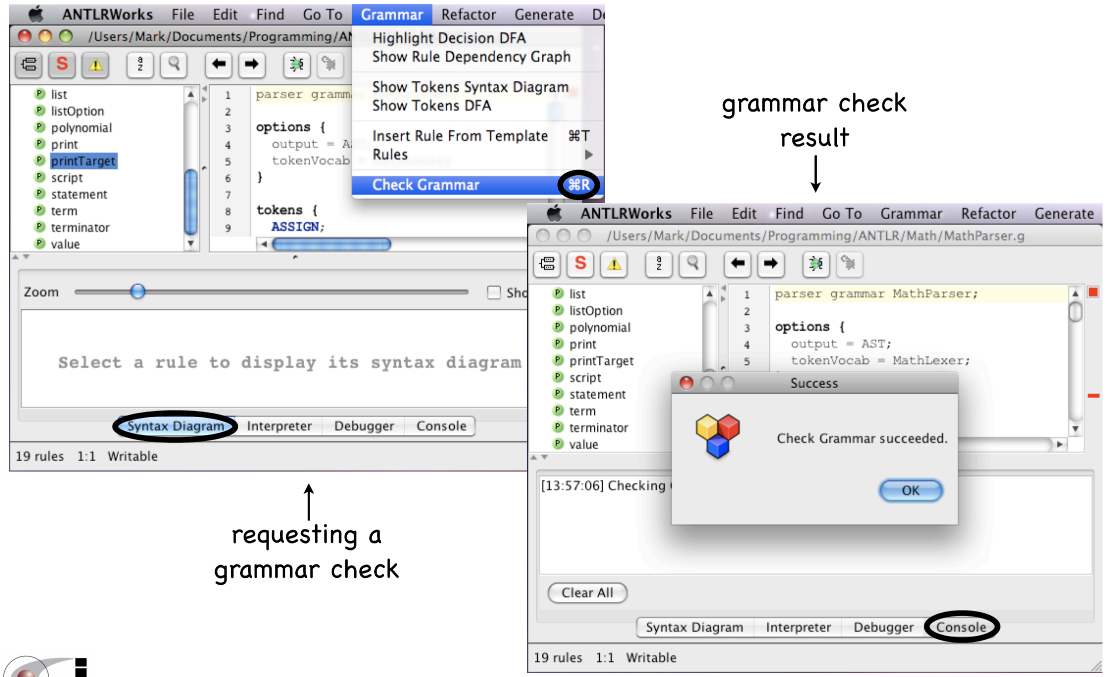
```
printTarget
  : NUMBER -> NUMBER
  | s1=STRING_LITERAL -> $s1
  | NAME -> NAME
  // This is a function reference to print a string
  | NAME LEFT_PAREN RIGHT_PAREN -> ^(FUNCTION NAME)
  | functionEval
  | derivative
  ;
```

parser rule
syntax diagram

lexer rule
syntax diagram

Rectangles correspond to fixed vocabulary symbols. Rounded rectangles correspond to variable symbols.

ANTLRWorks …


ANTLRWorks File Edit Find Go To Grammar Refactor Generate
Highlight Decision DFA
Show Rule Dependency Graph
Show Tokens Syntax Diagram
Show Tokens DFA
Insert Rule From Template ⌘T
Rules
Check Grammar ⌘R
/Users/Mark/Documents/Programming/ANTLR/Math/MathParser.g
list
listOption
polynomial
print
printTarget
script
statement
term
terminator
value
Zoom
Select a rule to display its syntax diagram
Syntax Diagram
Interpreter
Debugger
Console
19 rules 1:1 Writable
Success
OK
Clear All

```
parser grammar MathParser;

options {
  output = AST;
  tokenVocab = MathLexer;
```

Check Grammar succeeded.

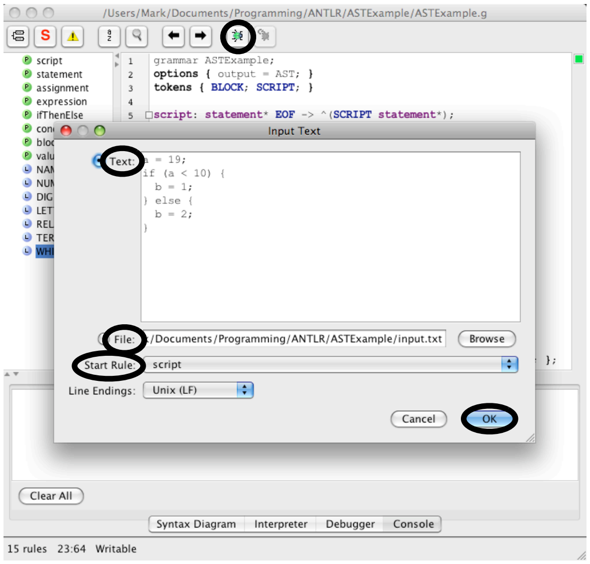
[13:57:06] Checking

grammar check result

requesting a grammar check

ANTLRWorks Debugger

- Simple when lexer and parser rules are combined in a single grammar file
  - press Debug toolbar button
  - enter input text or select an input file
  - select start rule
    - allows debugging a subset of grammar
  - press OK button



```
grammar ASTExample;
options { output = AST; }
tokens { BLOCK; SCRIPT; }

script: statement* EOF -> ^(SCRIPT statement*);
```

```
a = 19;
if (a < 10) {
  b = 1;
} else {
  b = 2;
}
```

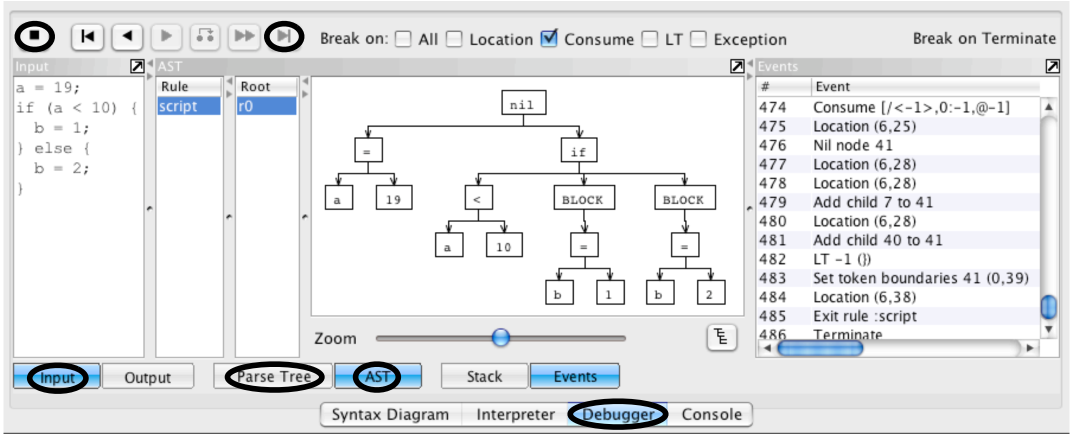
File: /Documents/Programming/ANTLR/ASTExample/input.txt

Start Rule: script

Line Endings: Unix (LF)

ANTLRWorks Debugger ...

- At the bottom of the ANTLRWorks window



Break on: ☐ All ☐ Location ☑ Consume ☐ LT ☐ Exception Break on Terminate

```
a = 19;
if (a < 10) {
  b = 1;
} else {
  b = 2;
}
```

Rule	Root
script	r0

#	Event
474	Consume [/<-1>,0:-1,@-1]
475	Location (6,25)
476	Nil node 41
477	Location (6,28)
478	Location (6,28)
479	Add child 7 to 41
480	Location (6,28)
481	Add child 40 to 41
482	LT -1 (})
483	Set token boundaries 41 (0,39)
484	Location (6,38)
485	Exit rule :script
486	Terminate

ASTExample
Start Rule: script

ANTLRWorks Debugger ...

- A bit more complicated when lexer and parser rules are in separate files

See the ANTLR Wiki page
"When do I need to use remote debugging?" at
http://www.antlr.org/wiki/pages/viewpage.action?pageId=5832732

- We'll demonstrate this after we see the Java code that ties all the generated classes together
  - see slides 53-56

Using Rule Return Values

These are examples from our tree grammar.

The code in curly braces is a rule "action" written in the target language, in this case Java.

```
printTarget
  : NUMBER { out($NUMBER); }
  | STRING_LITERAL {
      String s = unescape($STRING_LITERAL.text);
      out(s.substring(1, s.length() - 1)); // remove quotes
    }
  | NAME { out(getVariable($NAME)); }
  | ^(FUNCTION NAME) { out(getFunction(NAME)); }
  | functionEval { out($functionEval.result); }
  | derivative // handles own output
  ;
```

"unescape", "out", "getFunction", "getVariable", "evalFunction" and "toDouble" are methods we wrote in the tree grammar coming up.

```
functionEval returns [double result]
  : ^(FUNCTION fn=NAME v=NUMBER) {
      $result = evalFunction($fn, toDouble($v));
    }
  | ^(FUNCTION fn=NAME v=NAME) {
      $result = evalFunction($fn, getVariable($v));
    }
  ;
```

Rule Actions

- Add code before and/or after the generated code in the method generated for a rule
  - can be used for AOP-like wrapping of methods
- **@init { ... }**
  - inserts contained code before generated code
    - can be used to declare local variables used in actions of rule alternatives
  - used in our tree parser **polynomial** and **term** rules ahead
- **@after { ... }**
  - inserts contained code after generated code

Attribute Scopes

- Data is shared between rules in two ways
  - passing parameters and/or returning values
  - using attributes

same as options to share 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
  - all rules that request the named global scope of the attributes
- Attribute scopes
  - define collections of attributes that can be accessed by multiple rules
  - two kinds, global and rule scopes

Attribute Scopes ...

- Global scopes
  - named scopes defined outside any rule
  - define with

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

Use an @init rule action to initialize attributes.

  - request access to the scope in a rule with `scope name;`

To access multiple scopes, list them separated by spaces.

  - rule actions access variables in the scope with `$name::variable`

- Rule scopes
  - unnamed scopes defined inside a rule
  - define with

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

  - rule actions in the defining rule and rules invoked by it access attributes in the scope with `$rule-name::variable`

Our Tree Grammar

```
tree grammar MathTree;

options {
  ASTLabelType = CommonTree;
  tokenVocab = MathParser;
}

@header {
  package com.ociweb.math;

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

@members {
  private Map<String, Function> functionMap = new TreeMap<String, Function>();
  private Map<String, Double> variableMap = new TreeMap<String, Double>();
```

We're going to process an AST whose nodes are of type 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.

We want the generated parser class to be in this package.

We're using TreeMaps so the entries are sorted on their keys which is desired when listing them.

Our Tree Grammar ...

```
  private void define(Function function) {
    functionMap.put(function.getName(), function);
  }

  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;
  }

  private double evalFunction(CommonTree nameNode, double value) {
    return getFunction(nameNode).getValue(value);
  }
```

This adds a Function to our function Map.

This retrieves a Function from our function Map whose name matches the text of a given AST tree node.

This evaluates a function whose name matches the text of a given AST tree node for a given value.

Our Tree Grammar ...

```
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;
}

private static void out(Object obj) {
  System.out.print(obj);
}

private static void outln(Object obj) {
  System.out.println(obj);
}
```

This retrieves the value of a variable from our variable Map whose name matches the text of a given AST tree node.

These just shorten the code for print and println calls.

Our Tree Grammar ...

```
  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;
  }

  private static String unescape(String text) {
    return text.replaceAll("\\\\n", "\n");
  }

} // @members
```

This converts the text of a given AST node to a double.

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.

Our Tree Grammar ...

```
script: statement*;

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

interactiveStatement: help | list;

assign: ^(ASSIGN NAME v=value) { variableMap.put($NAME.text, $v.result); };

value returns [double result]
  : NUMBER { $result = toDouble($NUMBER); }
  | NAME { $result = getVariable($NAME); }
  | functionEval { $result = $functionEval.result; }
  ;

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 adds a variable to the variable map.

could also use $value here

This returns a value as a double. The value can be a number, a variable name or a function evaluation.

This returns the result of a function evaluation as a double.

Our Tree Grammar ...

```
define
  : ^(DEFINE name=NAME variable=NAME polynomial) {
      define(new Function($name.text, $variable.text, $polynomial.result));
    }
  ;

polynomial returns [Polynomial result]
scope { Polynomial current; }
@init { $polynomial::current = new Polynomial(); }
  : ^(POLYNOMIAL term[""] (s=SIGN term[$s.text])*) {
      $result = $polynomial::current;
    }
  ;
```

This builds a Function object and adds it to the function map.

This builds a Polynomial object and returns it.

The "current" attribute in this rule scope is visible to rules invoked by this one, such as term.

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.

Our Tree Grammar ...

```
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 builds a Term object and adds it to the current Polynomial.

Our Tree Grammar ...

```
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");
      // some lines omitted for space
      outln("To exit: exit or quit");
    }
  ;
```

This outputs help on our language which is useful in interactive mode.

Our Tree Grammar ...

```
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 lists all the functions or variables that are currently defined.

Our Tree Grammar ...

```
combine
  : ^(COMBINE fn1=NAME op=SIGN fn2=NAME fn3=NAME) {
      Function f2 = getFunction(fn2);
      Function f3 = getFunction(fn3);
      if ("+".equals($op.text)) {
        define(f2.add($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 adds or subtracts two functions to create a new one.

"`$fn1.text`" is the name of the new function to create.

Our Tree Grammar ...

```
print
  : ^(PRINT printTarget*)
    { System.out.println(); };

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)); }
  | functionEval { out($functionEval.result); }
  | derivative
  ;

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

This prints a list of printTargets then prints a newline.

This prints a single printTarget without a newline.

on slide 46

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

Using Generated Classes

- Our manually written Processor class
  - uses the generated classes
    - MathLexer extends Lexer
    - MathParser extends Parser
    - MathTree extends TreeParser
  - uses other manually written classes
    - Function
    - Polynomial
    - Term
  - supports two modes
    - batch - see processFile method
    - interactive - see processInteractive method

Lexer, Parser and TreeParser extend BaseRecognizer

Processor.java

```
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]");
        }
    }
```

Processor.java ...

```
    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
    }
```

Processor.java ...

```
    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);
        }
    }
```

Processor.java ...

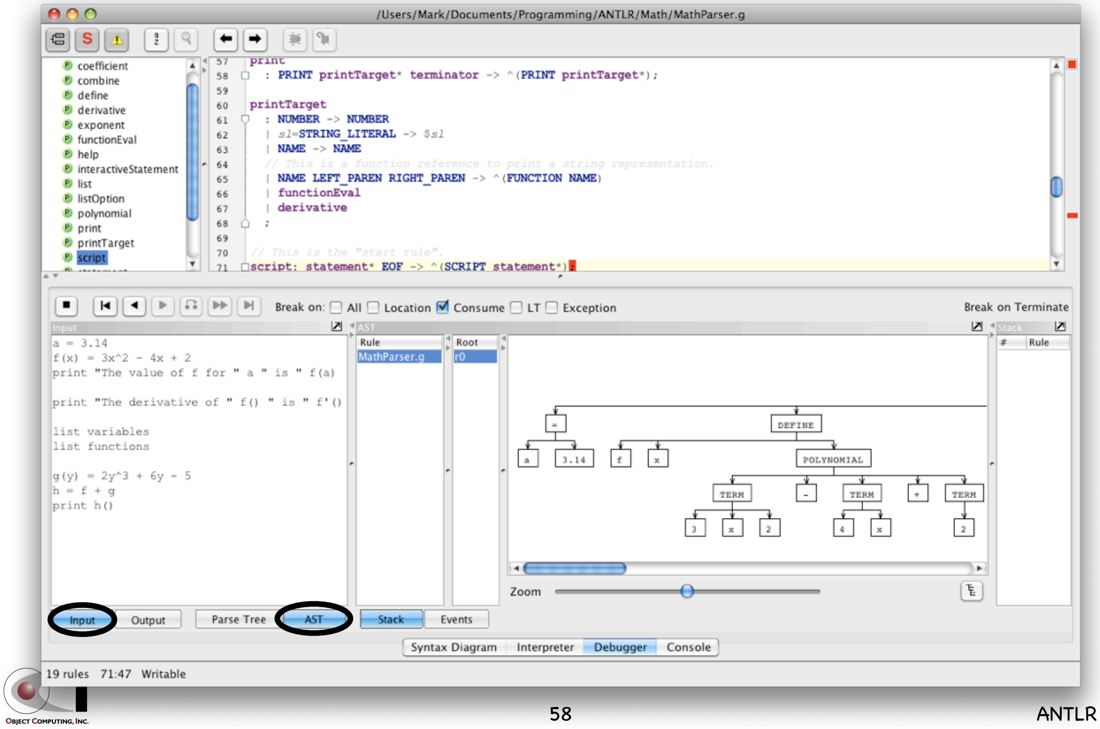
```
    private void processLine(MathTree treeParser, String line) throws RecognitionException {
        // Run the lexer and token parser on the line.
        MathLexer lexer = new MathLexer(new ANTLRStringStream(line));
        MathParser tokenParser = new MathParser(new CommonTokenStream(lexer));
        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
```

We can't create a new instance of MathTree
for each line processed because
it maintains the variable and function Maps.

ANTLRWorks Debugger

- Let's demonstrate using remote debugging which is necessary when lexer and parser rules are in separate grammar files
  - edit build.properties to include `-debug` in `tool.options`
  - `ant clean run`
    - the run target in build.xml tells it to parse the file "simple.math"
  - start ANTLRWorks
  - open the parser grammar file
  - select Debugger … Debug Remote…
  - press "Connect" button
  - debug as usual

ANTLRWorks Debugger …



/Users/Mark/Documents/Programming/ANTLR/Math/MathParser.g

```
print
  : PRINT printTarget* terminator -> ^(PRINT printTarget*);

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
  ;

// This is the "start rule".
script: statement* EOF -> ^(SCRIPT statement*);
```

Break on: □ All □ Location ☑ Consume □ LT □ Exception

```
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()
```

Rule	Root
MathParser.g	r0

19 rules 71:47 Writable

References

- ANTLR
  - http://www.antlr.org
- ANTLRWorks
  - http://www.antlr.org/works
- StringTemplate
  - http://www.stringtemplate.org
  - http://www.codegeneration.net/tiki-read_article.php?articleId=65 and 77
- My slides and code examples
  - http://www.ociweb.com/mark - look for "ANTLR 3"