ANTLR 3

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

Key:
Fundamental Topics
Our Example
Advanced Topics

Outline

- ANTLR Overview
- Math Language Description
- Grammar Syntax
- Lexer Rules
- Whitespace & Comments
- Hidden Tokens
- Math Lexer Grammar
- Token Specification
- Rule Syntax
- ASTs
- Math Parser Grammar
- ANTLRWorks
- Rule Actions
- Attribute Scopes
- Math Tree Parser Grammar
- Using Generated Classes
- Ant Tips
- ANTLRWorks
  Remote Debugging
- StringTemplate
- Lookahead
- Semantic Predicates
- Syntactic Predicates
- Error Handling
- gUnit
- References

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.

I worked with Ter as a masters student there.

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 ...

- Supports LL(*)
  - **LL**(**k**) parsers are top-down parsers that
    - parse from **L**eft to right
    - construct a **L**eftmost derivation of the input
    - look ahead **k** tokens
  - **LR**(**k**) parsers are bottom-up parsers that
    - parse from **L**eft to right
    - construct a **R**ightmost derivation of the input
    - look ahead **k** tokens
  - LL parsers can't handle left-recursive rules
  - most people find LL grammars easier to understand than LR
- Supports predicates
  - aid in resolving ambiguities (non-syntactic rules)

Wikipedia has good descriptions of LL and LR.

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

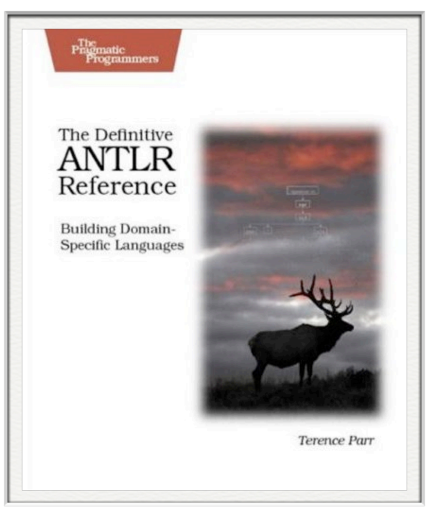
  - generate code that validates and translates input into another format such as a programming language or bytecode
  - covered when we discuss "StringTemplate" later

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



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

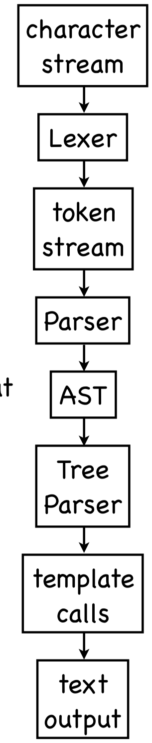
Other DSL Approaches

- 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

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

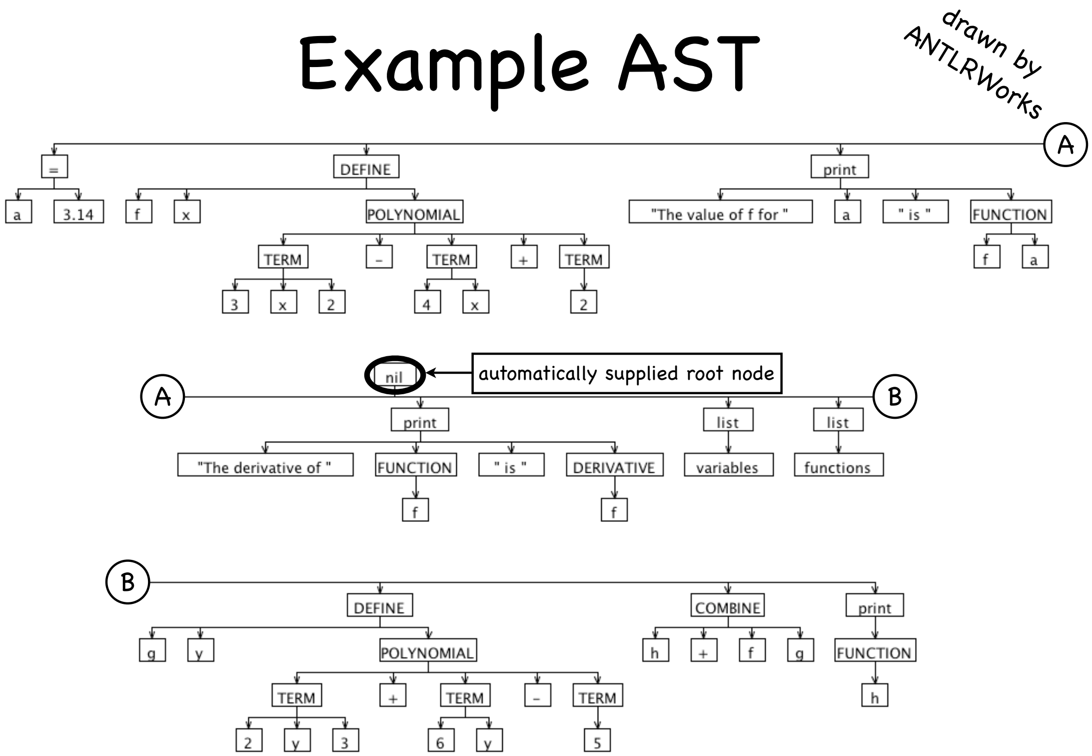
Output:
h(x) = 2x^2 + 2x - 5

Example Input/Output

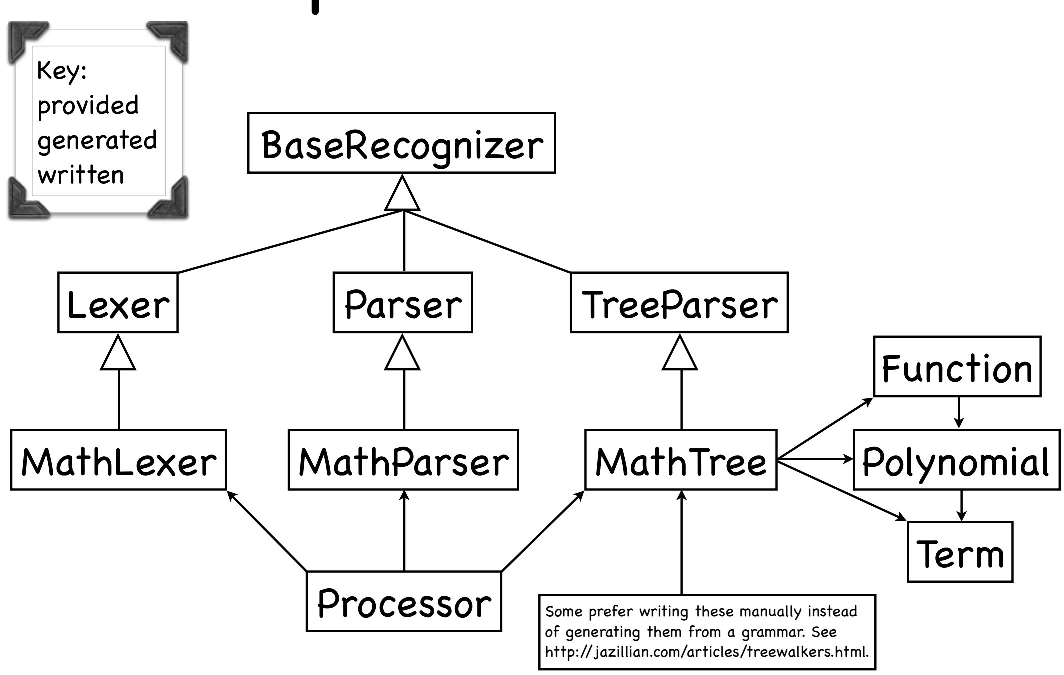
```
a = 3.14
f(x) = 3x^2 - 4x + 2
print "The value of f for " a " is " f(a)

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



Important Classes



ANTLR Documentation

http://antlr.org

Grammar Syntax

Comments use the same syntax as Java.

3 types: lexer, parser and tree; defaults to combined lexer and parser

grammar-type? **grammar** ***grammar-name***;

must match the filename with a ".g" extension

grammar-options?

token-spec?

Syntax elements will be discussed in the order they are needed in our Math language.

attribute-scopes?

grammar-actions?

rule+

The classes generated by ANTLR will contain a method for each rule.

Grammar Options

- These include
  - AST node type - **`ASTLabelType = CommonTree`**
    - used in grammars that create or parse ASTs

    can use your own class to represent AST nodes
  - infinite lookahead - **`backtrack = true`**
    - provides infinite lookahead for all rules; parsing is slower with this on
  - limited lookahead - **`k =`** ***`integer`***
  - output type - **`output = AST | template`**
    - choose **`template`** when using the StringTemplate library
    - don't set if not producing output or doing so with printlns in actions
  - token vocabulary - **`tokenVocab =`** ***`grammar-name`***
    - allows one grammar file to use tokens defined in another (with lexer rules or a token spec.); reads generated .tokens files
- Specified with

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

don't need quotes around single word values

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` must be lexer, parser (the default) or treeparser
- **`@`*`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
- @rulecatch

  see discussion on "Error Handling" later

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; same value as Token.HIDDEN_CHANNEL

- 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.

Hidden Tokens

- By default the parser only processes tokens from the default channel
- Can request tokens from other channels
  - tokens are assigned unique, sequential indexes regardless of the channel to which they are written
  - Token constants and methods
    - `public static final int DEFAULT_CHANNEL`
    - `public static final int HIDDEN_CHANNEL`
    - `public int getChannel() // where this Token was written`
    - `public int getTokenIndex() // index of this Token`
  - CommonTokenStream methods
    - `public Token get(int index)`
    - `public List getTokens(int start, int stop)`
    - `public int index() // returns index of the last Token read`

CommonTokenStream class implements TokenStream interface

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 15.

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

Rule Syntax

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

only for lexer rules (→ fragment?)

include backtrack and k (→ rule-options?)

```
options {
  ...
}
```

add code before and/or after code in the generated method for this rule (→ rule-actions?)

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 (→ exceptions-spec?)

Creating ASTs

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

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

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

a combined lexer/parser grammar

Parse Trees and ASTs

Grammar with AST operators and rewrite rules

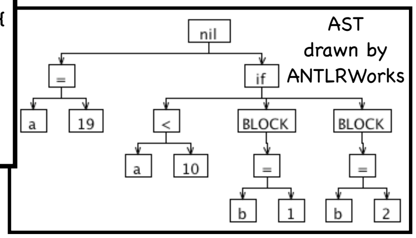
```
grammar ASTExample;
options { output = AST; }
tokens { BLOCK; }

script: statement* EOF -> statement*;
statement: assignment | ifThenElse;
assignment: NAME '=' expression TERMINATOR
  -> ^('=' NAME expression);
expression: value (('+' | '-')^ value)*; // no '*' or '/'
ifThenElse:
  'if' '(' condition ')' b1=block ('else' b2=block)?
  -> ^('if' condition $b1 $b2?);
condition: value RELATION^ value;
block: '{' statement* '}' -> ^(BLOCK statement*);
value: NAME | NUMBER;

NAME: LETTER (LETTER | DIGIT | '_')*;
NUMBER: '-'? DIGIT+; // just integers
fragment DIGIT: '0'..'9';
fragment LETTER: 'A'..'Z' | 'a'..'z';
RELATION: '<' | '<=' | '==' | '>=' | '>';
TERMINATOR: ';';
WHITESPACE: (' ' | '\t' | '\r' | '\n')+ { $channel = HIDDEN; };
```

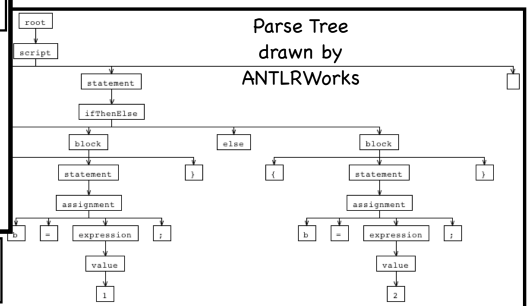
"Labels" like b1 and b2 are used to refer to non-unique elements.

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

Input

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





Parse trees show the depth-first order of rules that are matched.

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.

Using Rule Arguments

These are examples from our parser grammar.

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

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

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

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

```
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);
  }
```

term variables must match their function variable

This catches bad function definitions such as `f(x) = 2y`.

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

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.

a "subrule"

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);

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

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 (for `c=coefficient? (tv=NAME e=exponent?)?`):
`4`
`4x`
`x^2`
`4x^2`

Example (for `exponent`):
`^2`

Our Parser Grammar ...

```
help: HELP terminator -> HELP;

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

listOption: FUNCTIONS | VARIABLES;

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

Examples (for `help`):
`?`
`help`

Examples (for `list`):
`list functions`
`list variables`

Examples (for `listOption`):
`functions`
`variables`

Examples (for `combine`):
`h = f + g`
`h = f - g`

Our Parser Grammar ...

```
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
  ;

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

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

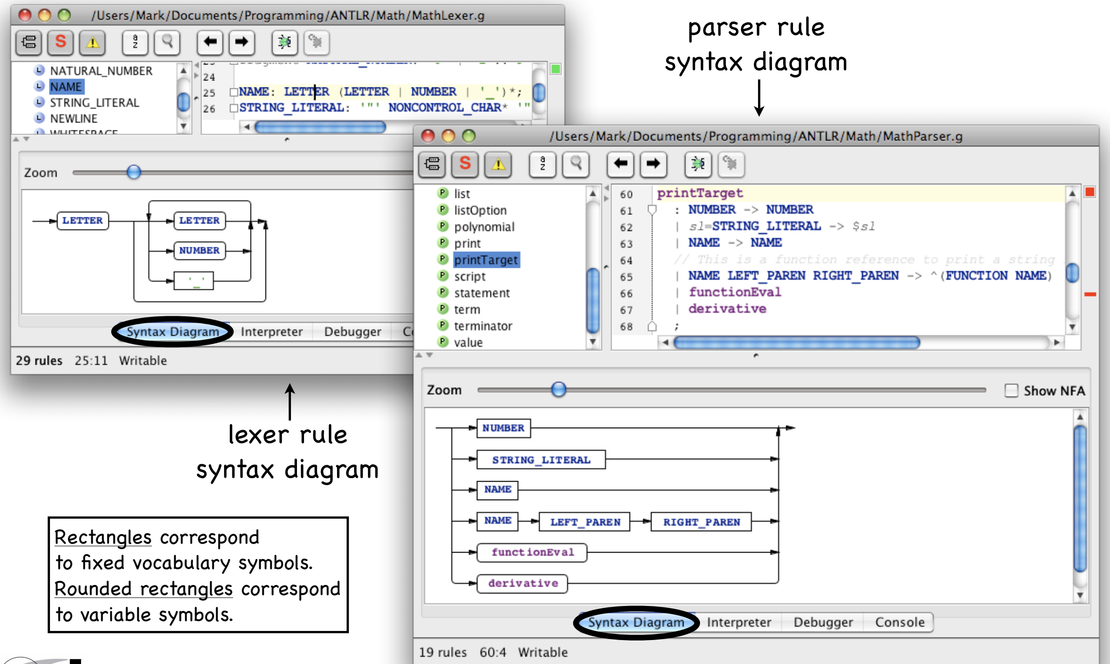
Examples (for printTarget):
```
19
3.14
"my text"
a
f()
f(2)
f(a)
f'()
```

Example (for derivative):
`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 ...

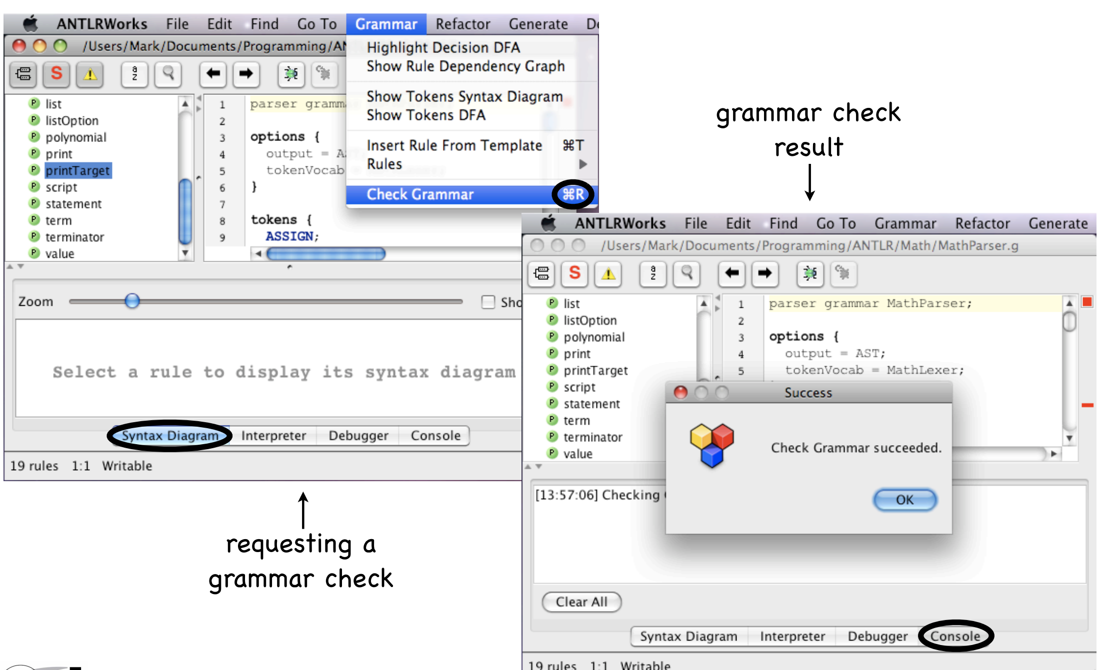


parser rule
syntax diagram

lexer rule
syntax diagram

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

ANTLRWorks ...



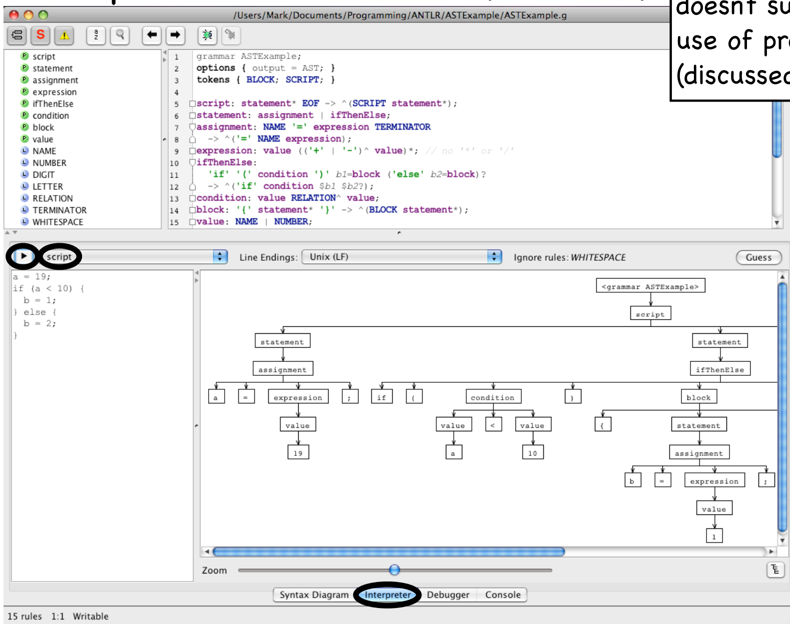
grammar check
result

requesting a
grammar check

ANTLRWorks Interpreter

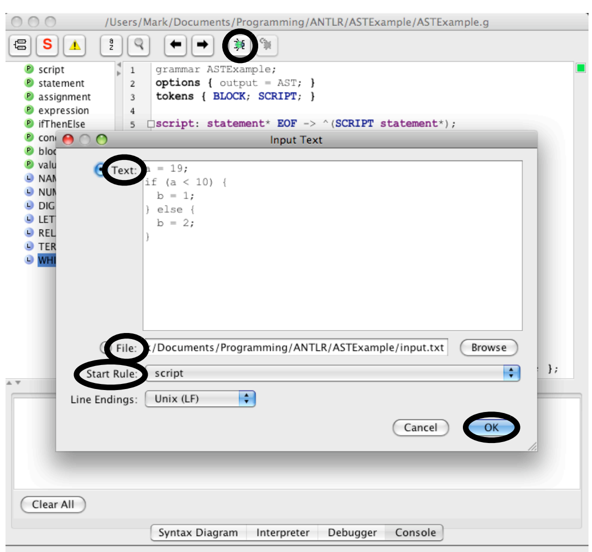
- Tests parse tree creation (not AST)

doesn't support use of predicates (discussed later)



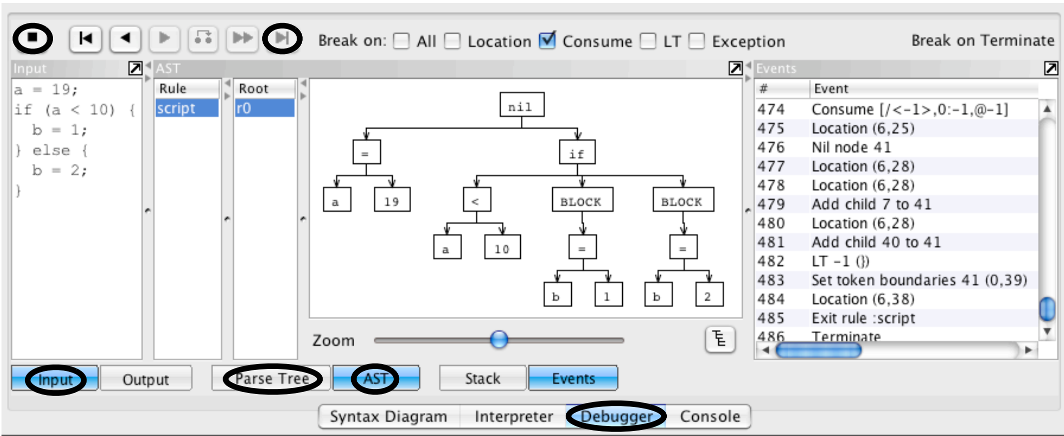
ANTLRWorks Debugger

- 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



ANTLRWorks Debugger ...

- At the bottom of the ANTLRWorks window



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 61-64

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
  ;
```

"out" is a method we wrote. See slide 51.

"unescape" is a method we wrote. See slide 52.

"getVariable" is a method we wrote. See slide 51.

"getFunction" is a method we wrote. See slide 50.

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

"evalFunction" is a method we wrote. See slide 50.

"toDouble" is a method we wrote. See slide 52.

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 53

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.

Ant Tips

- Ant is a great tool for automating tasks used to develop and test grammars
  - generate Java classes and .tokens files from each grammar file

```
<java classname="org.antlr.Tool"
  classpathref="classpath" fork="true">
  <arg line="-lib gen -o gen ${grammar.name}.g"/>
</java>
```

.tokens files assign integer constants to token names; used by org.antlr.Tool

    - -o option specifies directory where generated files should be written
    - -lib option specifies directory where .tokens files can be found

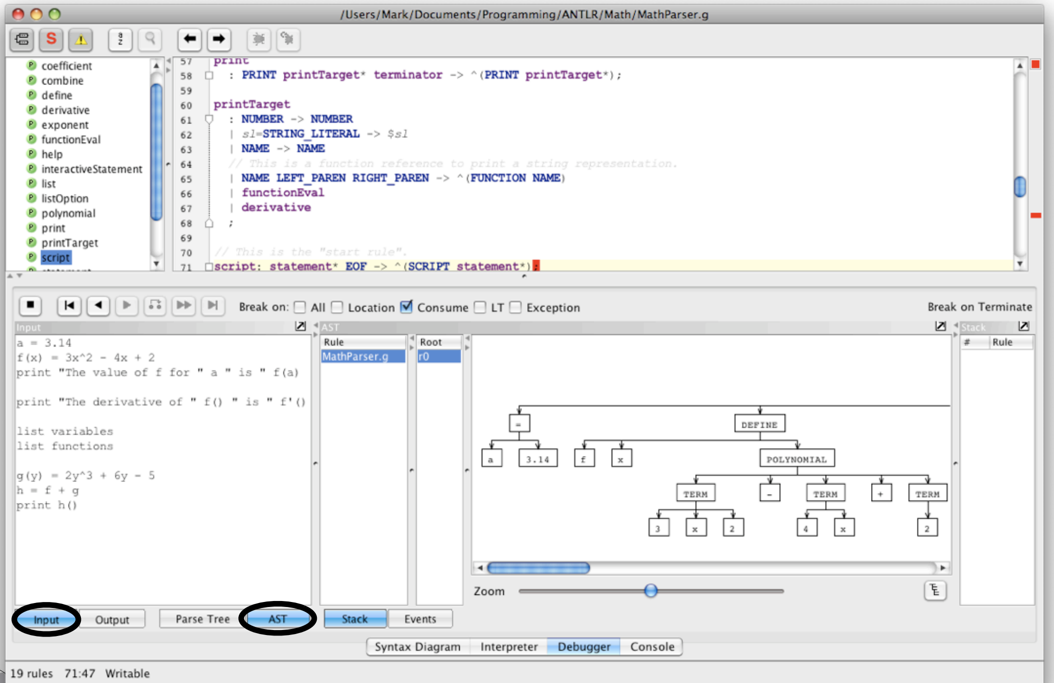
-o and -lib values may differ if lexer, parser, and tree grammars are in different directories.

  - only run org.antlr.Tool if the grammar has changed since the last build
    - using the "uptodate" task and the "unless" target attribute
  - compile Java source files
  - run automated tests
  - run the application using a specific file as input
  - delete generated files (clean)

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`**
  - start ANTLRWorks
  - open the parser grammar file
  - select Debugger ... Debug Remote...
  - press "Connect" button
  - debug as usual

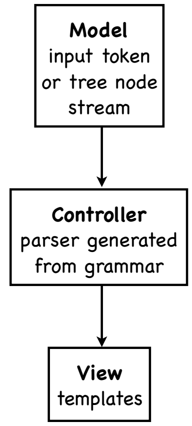
ANTLRWorks Debugger ...



Some More Advanced Topics

StringTemplate

- Great for implementing translators
  - see slide 6
- A template engine implemented in Java
  - alternative to producing output with printlns
    - set grammar option "`output = template;`"
  - separates output from the logic that produces it
    - like the view portion of MVC
  - supports output "retargeting"
    - for example, translating input into multiple programming languages, Java bytecode and XML
    - can automatically indent output source code
- http://www.stringtemplate.org



StringTemplate ...

- Output is specified by templates
  - typically stored in separate text files, not in grammar files
    - .stg file extension stands for StringTemplate Group
  - each template can accept "attributes"
    and produce single or multi-line output
- Example
  - our Math scripts could be translated
    to equivalent Java source code
- Usage strategy
  - every rule returns a StringTemplate object
    that represents the accumulated output
    from that rule and all rules it invokes
  - the start rule parser method returns its StringTemplate object
    and the code that invoked it prints that object

StringTemplate ...

- StringTemplate Group file syntax

```
group name

template-name(attribute-list) ::= "content"

template-name(attribute-list) ::= <<
content
>>
```

The value of ***name*** is only important when group inheritance or interfaces are used.
See http://www.antlr.org/wiki/display/ST/Group+Files

can include \n characters

Use << >> for multi-line output and when content contains double quotes.

  - attribute lists are comma-separated attribute names without types
  - there are some keywords that cannot be used for template or attribute names

default first group if implements interface last length optional rest strip super trunc else endif elseif

StringTemplate ...

- Expression elements in template content
  - single-valued attributes
    - <***attribute***>
  - multi-valued attributes
    - <***attribute***; **separator="*text*"**>
      concatenates the toString value of each with separator
    - <**first(*attribute*)**>
    - <**last(*attribute*)**>
    - <**rest(*attribute*)**> - all but first
    - <**length(*attribute*)**> - number of values
  - conditional logic
    - <**if(*attr1*)**>***content***<**elseif(*attr2*)**>***content***<**else**>***content***<**endif**>
      outputs content if attribute has a value and isn't the boolean false
    - <**if(!attr)**>***content***<**endif**>
      outputs content if attribute doesn't have a value or is the boolean false
  - include output of another template
    - <***template-name*(*attr-list*)**>
  - and much more

not XML!

StringTemplate Example

Referring to templates in rewrite rules (MathTree.g)

See full example in MathWithStringTemplate.zip.

```
assign
  : ^(ASSIGN NAME value)
    -> assign(name={$NAME}, value={$value.st})
  ;

value
  : NUMBER -> number(text={$NUMBER})
  | NAME -> variable(name={$NAME})
  | functionEval -> {$functionEval.st}
  ;
```

note use of named parameters in template calls

using the StringTemplate result from another rule

a rule not shown here

Templates (MathTree.stg)

```
assign(name, value) ::=
  <<assign("<name>", <value>);>>
number(text) ::= "<text>"
variable(name) ::= <<getVariable("<name>")>>
```

a Java method we wrote that stores the variable in a HashMap

a Java method we wrote that retrieves the value of a variable from a Map

Java application code required (Processor.java)

```
Reader reader = new FileReader("MathTree.stg");
treeParser.setTemplateLib(new StringTemplateGroup(reader));
reader.close();
MathTree.script_return result = treeParser.script();
System.out.println(result.getTemplate());
```

the start rule method

Lookahead

- Necessary when more than one token must be examined to choose between rule alternatives
  - example - Java field or method declaration?

```
accessSpecifier 'static'? type name ('=' value)? ';'
accessSpecifier 'static'? type name '(' parameterList? ')' ...
```

- 3 ways to get lookahead
  - option "**backtrack = true;**" for infinite lookahead
  - option "**k = *look-ahead-distance*;**" for finite lookahead
    - more efficient
  - syntactic predicates to prioritize alternatives - discussed ahead

also see memoize option

- 2 places to specify options
  - globally as a grammar option
  - locally as a rule option so lookahead in each rule can differ

Semantic Predicates

- Alter parsing based on a boolean expression written in the target language
- Three types

ANTLR was the first parser generator tool to support grammar predicates.

- validating semantic predicate
  - throws FailedPredicateException if it evaluates to false
  - syntax: `{ target-language-expression }?`
  - location: end of a rule alternative
- gated semantic predicate
  - disables a rule alternative if it evaluates to false
  - if no other alternative matches, the input is considered a syntax error
  - could refer to command-line option values
  - syntax: `{ target-language-expression }?=>`
  - location: beginning of a rule alternative
- disambiguating semantic predicate
  - like a gated sem. pred., but only used when syntax is insufficient to choose
  - syntax: `{ target-language-expression }?`
  - location: beginning of a rule alternative

Validating Sem. Pred.

- Example
  - don't allow defining a function that is already defined
  - modify the "define" rule in our tree grammar

```
define
  : ^(DEFINE name=NAME variable=NAME polynomial)
    { !functionMap.containsKey($name.text) }?
    { define(new Function(
        $name.text, $variable.text, $polynomial.result)); }
  ;
  catch[FailedPredicateException e] {
    System.err.println("cannot redefine function " + $name);
  }
```

Gated Sem. Pred

- Example
  - only allow use of the interactive statements "help" and "list" when in interactive mode
  - add this near top of our parser grammar

```
@members { public boolean interactiveMode; }
```

  - add this to the processLine method in Processor.java after creating the MathParser instance

```
tokenParser.interactiveMode = true;
```

  - modify the "statement" rule in our parser grammar

```
statement
  : assign | combine | define | print
  | { interactiveMode }?=> interactiveStatement
  ;
```

defined on slide 33

When not in interactive mode and a list statement is encountered, the error message "mismatched input 'list' expecting EOF" is output.

Disambiguating Sem. Pred.

- Example
  - support printing function definitions without following name with ()

```
print f
instead of
print f()
```

    - requires checking whether the name is a variable or function
  - remove the following unneeded alternative from the parser grammar "printTarget" rule (37)

```
| NAME LEFT_PAREN RIGHT_PAREN -> ^(FUNCTION NAME)
```

The rule used to match variable names will also be used to match function names.

  - modify two alternatives in tree grammar "printTarget" rule (59)
    - old alternatives

```
| NAME { out(getVariable($NAME)); }
| ^(FUNCTION fn=NAME) { out(getFunction($fn)); }
```

    - new alternatives

```
| { variableMap.containsKey(((Tree) input.LT(1)).getText()) }?
  NAME { out(getVariable($NAME)); }
| NAME { out(getFunction($NAME)); }
```

ambiguous alternatives with different actions

We'll assume that name is a function if it's not a variable.

The Parser class has an attribute named "**input**" that is a TokenStream.
The TreeParser class has an attribute named "**input**" that is a TreeNodeStream.
Both TokenStream and TreeNodeStream have a method named "**LT**" that returns the ith **L**ookahead **T**oken or tree node.

Syntactic Predicates

- Examine upcoming tokens in the stream to determine whether a rule alternative should be considered
  - if the upcoming tokens match a given sequence then consider this alternative
    - rewinds the input stream and processes the alternative
  - syntax: **(*sequence*)=>**
  - location: beginning of a rule alternative
  - implemented as a gated semantic predicate
- Two uses
  - to specify precedence of ambiguous rule alternatives
  - when a fixed amount of lookahead won't work
    - recursive, nested structures such as parenthesized groups
    - otherwise can use "k" option instead

Syntactic Predicates ...

- Example – C function declarations/definitions
  - function <u>declarations</u> look like `type ID '(' arg* ')' ';'`
  - function <u>definitions</u> look like `type ID '(' arg* ')' '{' body '}'`
  - can't recognize them by examining a fixed number of tokens because **arg** can consist of nested parentheses
    - for example "**int (*ptr)(double)**" describes an argument named **ptr** that is a pointer to a function that takes a **double** parameter and returns an **int**
    - could have a pointer to a function that takes a parameter that is a pointer to a function
- Three ways to resolve
  - **backtrack** option
  - left-factoring alternatives
  - syntactic predicates

Example input:

```
int add(int m, int n);
int add(int m, int n) {
  return m + n;
}
int myCallback(double x);
void registerCallback(int (*callbackPtr)(double));

int main() {
  registerCallback(myCallback);
  int sum = add(1, 2);
}
```

Syntactic Predicates ...

```
grammar CFunctions;

source: topLevelStmt* EOF; // start rule

topLevelStmt
  // options { backtrack = true; }
  : (funcDecl)=> funcDecl
  | funcDef
  | COMMENT;

funcDecl: type ID '(' args? ')' ';';
funcDef:  type ID '(' args? ')' '{' body '}';

args: arg (',' arg)*;
arg: type ID? | funcPtr;

funcPtr: type '(' '*' ID ')' '(' args ')';
                                 ↑ recursion

body: (COMMENT | bodyStmt)*;
bodyStmt: (assignment | funcCall | returnStmt) ';';
```

The "**backtrack = true;**" grammar/rule option adds a syntactic predicate to every rule alternative.

This is less efficient than only adding them where needed and only checking as many tokens as necessary to select an alternative.

Use of the "backtrack" option is recommended only during grammar prototyping. It can be eliminated by adding syntactic predicates or by "left-factoring" alternatives.

Alternately, **funcDecl** and **funcDef** can be left-factored like this.

```
topLevelStmt: funcDeclOrDef | COMMENT;
funcDeclOrDef: funcPrefix (';' | '{' body '}');
funcPrefix: type ID '(' args? ')';
```

A disadvantage of left-factoring is that it makes writing actions and rewrite rules more difficult since what were distinct alternatives are now combined.

Syntactic Predicates ...

```
assignment: type? ID '=' expression;
type: BUILTIN; // not supporting arrays, structs, pointers or references
expression: value (('+' | '-') value)*;
value: NUMBER | string | ID | funcCall;
funcCall: ID '(' params? ')';
params: expression (',' expression)*;
returnStmt: 'return' expression;
string: '"' ~('"')* '"';

BUILTIN: 'void' | 'bool' | 'char' | 'double' | 'int' | 'float' | 'long' | 'short';
ID: LOWERCASE LETTER*;
fragment LETTER: LOWERCASE | UPPERCASE;
fragment LOWERCASE: 'a'..'z';
fragment UPPERCASE: 'A'..'Z';
NUMBER: '0' | '1'..'9' '0'..'9'*;
NEWLINE: ('\r'? '\n')+ { $channel = HIDDEN; };
WHITESPACE: (' ' | '\t')+ { $channel = HIDDEN; };
```

Error Handling

- Default error handling
  - code in the generated method for each rule
    is in a try block with the following catch

```
catch (RecognitionException re) {
    reportError(re);
    recover(input, re);
    ...
}
```

RecognitionException is the base class of most exceptions thrown by generated code. It provides access to the related input characters or token.

  - **reportError** and **recover** are methods in BaseRecognizer
    - the superclass of generated lexer, parser and tree parser classes
  - override in all rules to stop processing after first error

```
@rulecatch {
  catch (RecognitionException re) {
    reportError(re);
  }
}
```

@rulecatch is a grammar action

Error Handling ...

- **reportError** method
  - calls **displayRecognitionError**
    - concatenates an error header generated by **getErrorHeader**
      with an error message generated by **getErrorMessage**
      and passes the result to **emitErrorMessage**
    - calls **getErrorHeader**
      - returns "line {line-#}:{column-#}"
      - override to change or eliminate error message headers
    - calls **getErrorMessage**
      - returns a string that is specific to each RecognitionException subclass
      - override to customize messages
      - calls **getTokenErrorDisplay**
        - if the token has text, returns that in singles quotes
        - otherwise returns the token type in angle brackets
    - calls **emitErrorMessage**
      - writes the message to stderr
      - override to write elsewhere such as a log file

These are all BaseRecognizer methods. The easiest way to override these is to use the @members grammar action.

Error Handling ...

- Methods generated for each rule
  - make multiple calls to the BaseRecognizer **match** method in a try block
- BaseRecognizer **match** method
  - calls **mismatch** when the next token isn't what is expected
  - **mismatch** throws one of three kinds of exception based on details of the mismatch
    - **UnwantedTokenException**
    - **MissingTokenException**
    - **MismatchedTokenException**
  - can override **mismatch** and call **mismatchRecover** to attempt to recover and continue parsing
    - if an expected token was missing, it will insert a single token
    - if an unexpected token was found, it will delete a single token

gUnit

- Grammar unit testing framework
  - at http://www.antlr.org/wiki/display/ANTLR3/gUnit+-+Grammar+Unit+Testing
  - download gunit-1.0.2.jar
- Verifies that grammar produces expected outputs from specified inputs
  - input can be a single line (delimited by " "), multiple lines (delimited by << >>) or file content

    " " can contain \n characters.

    similar to StringTemplate syntax
  - output can be a single line, multiple lines or an AST
  - can test rule return values
  - can test that an error message is emitted or no error message is emitted

gUnit ...

- To run
  - CLASSPATH must contain ...
    - antlr-3.0.jar, stringtemplate-3.0.jar and gunit-1.0.2.jar
  - **java org.antlr.gunit.Interp *filename*.testsuite**
- Example MathParser.testsuite file
  - tests AST construction

To try, download Math.zip and run "**ant gunit**".

```
gunit MathParser;

assign:
"a = 3.14" -> (= a 3.14)

combine:
"f = g + h" -> (COMBINE f + g h)

define:
"f(x) = 3x^2 - 2x + 4" ->
(DEFINE f x (POLYNOMIAL (TERM 3 x 2) - (TERM 2 x) + (TERM 4)))
```

Note that right sides look like AST construction rewrite rules, but don't start with "^".

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"

Thanks

- Thank you for attending my talk!
- Feel free to email me questions about ANTLR

mark@ociweb.com