



J-Spring

16 april 2008 Spant! - Bussum



Building a Domain Specific language with ANTLR v3

Jeroen Leenarts
InfoSupport



About InfoSupport and me

- About InfoSupport
 - www.infosupport.com
 - Come visit our conference booth
- About Jeroen Leenarts
 - blog.leenarts.net



Agenda

- Domain specific languages
- Steps in parsing a language
- Features of ANTLR v3
- Writing a Grammar
- What we did not cover today
- Questions

Some materials in these slides have been derived from the book “The Definitive ANTLR Reference” by Terence Parr.



Domain Specific languages

- Validator
 - Validate input against the defined grammar
- Processor
 - Validate and process input
 - perhaps some calculations, database updating, convert config data to runtime data structures
- Translator
 - Validate and translate input
 - other language, bytecode
 - XML/Other structured format

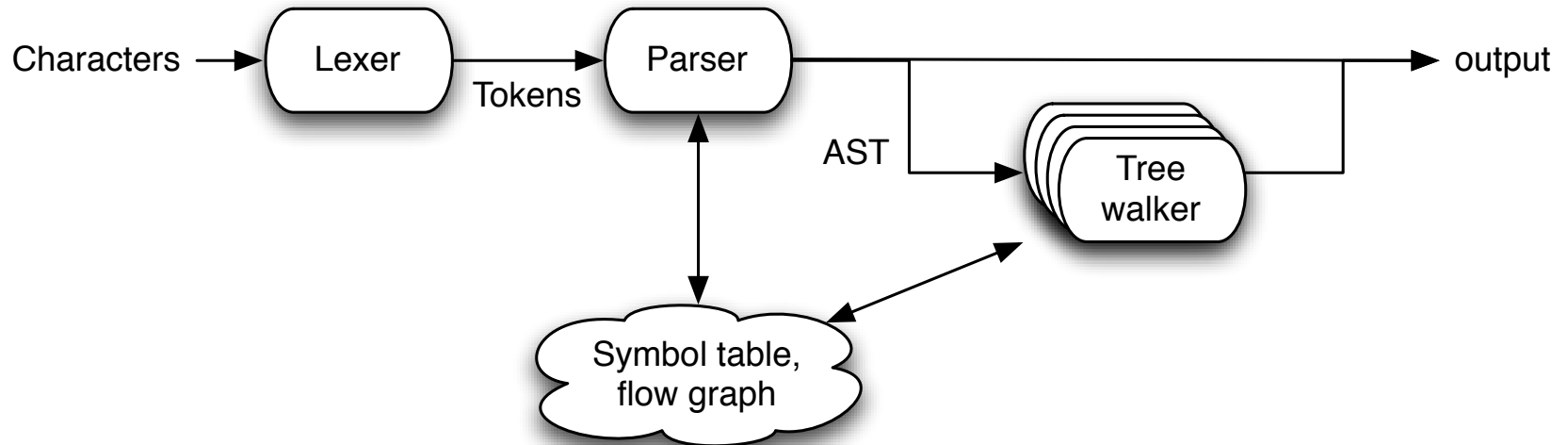


Other DSL related sessions

- xText: DSL's made easy
 - Jeroen Benckhuijsen, Meinte Boersma
- Pragmatic MDA: Domain Specific Languages with Eclipse (EMF + GMF + oAW)
 - Jos Warmer



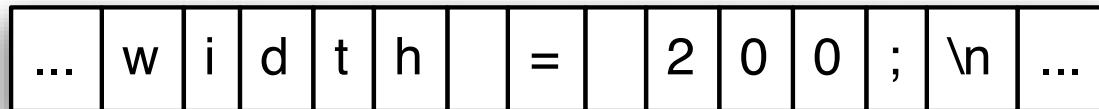
Steps in parsing a language



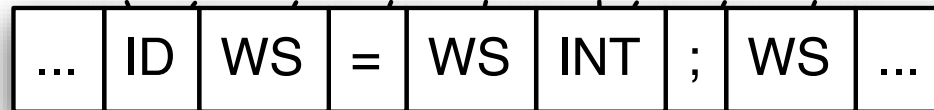


Steps in parsing a language

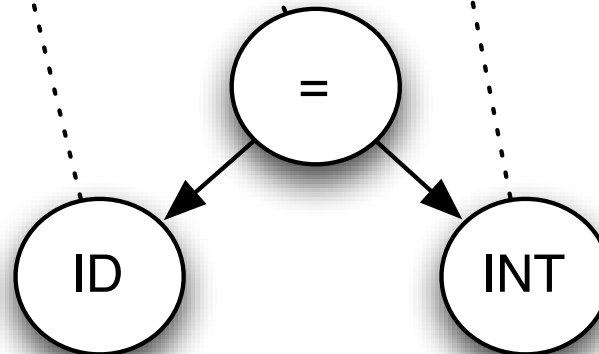
Characters



Tokens



AST



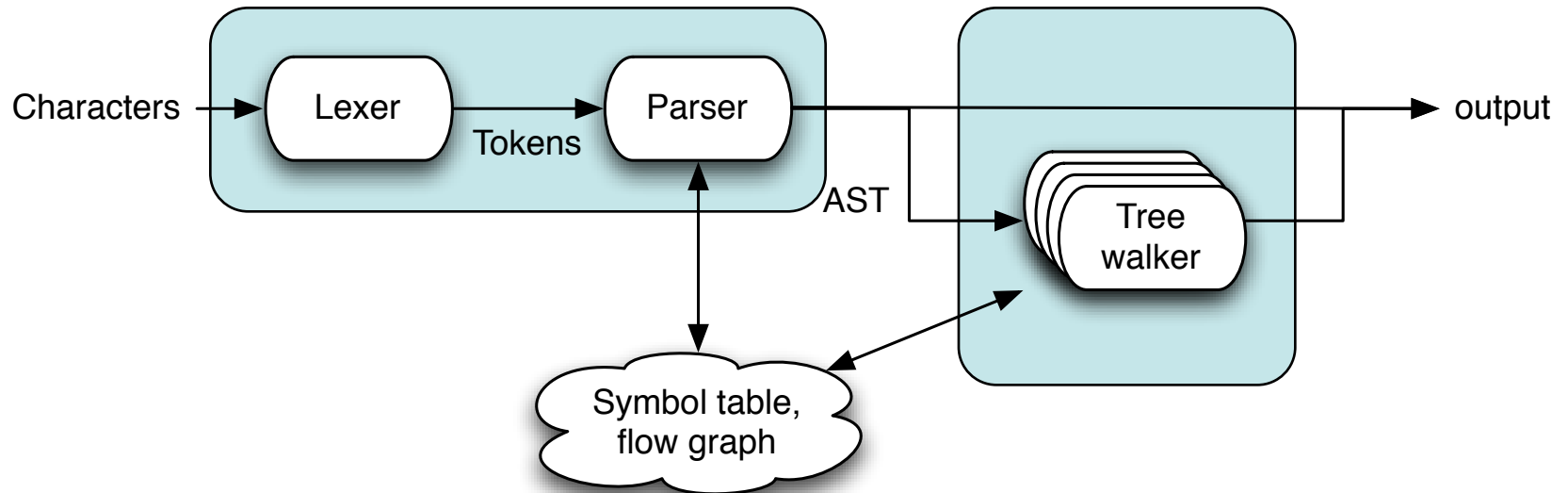


Some terminology

- Lexer
 - Process character stream to token stream
- Parser
 - Process a stream of tokens
 - Like creating an AST
- Abstract Syntax Tree
 - An intermediate representation of the input
 - Tree based datastructure
- Tree parser
 - Processes an AST



Steps in parsing a language





ANTLR v3

- **AN**other **T**ool for **L**anguage **R**ecognition
 - Takes a grammar description, generates a lexer and parser
- Easier to use, compared to similar tools
- EBNF Grammar
- Infinite lookahead (LL*)
 - Allows complexer grammar definitions
 - Top down parser



ANTLR v3

- Detailed recognition-error reporting
- History dates back to 1988
- Used by industry leaders
 - Apple: Keynote 4
 - Adobe: Flex Builder 3
 - BEA Weblogic: Server (JSP parsing)
 - JBoss: Drools
- Written in Java, language independent
 - Java, C#, Objective-C, C, Python, Ruby



Writing a Grammar

- Simple arithmetic expressions
 - $a=3$
 - $b=4$
 - $2+a*b$
- Three stages in the demo
 - Grammar definition
 - Action based processing
 - Tree based processing



The arithmetic grammar

Our program consists of statements

```
prog: stat+ ;
```

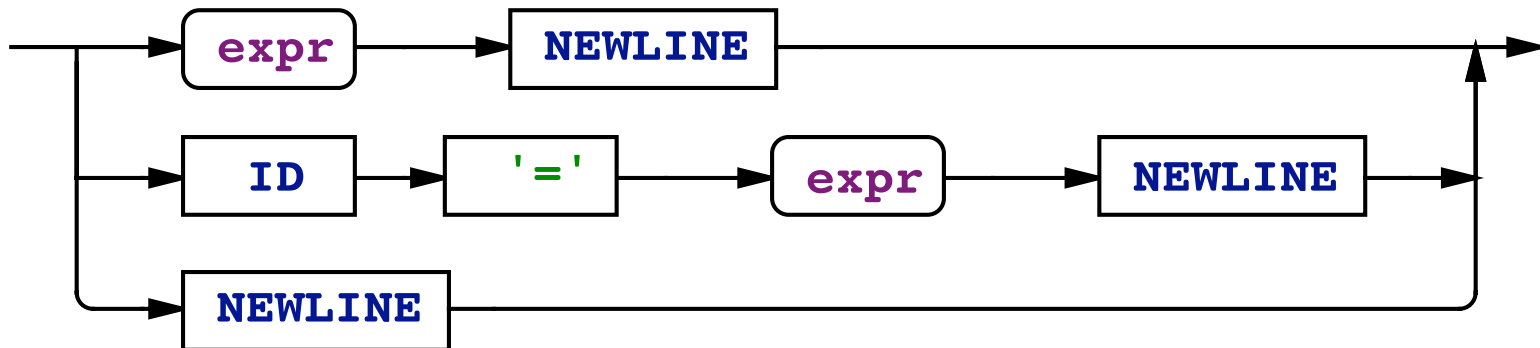




The arithmetic grammar

Let's start with a statement

```
stat:  expr NEWLINE  
      | ID '=' expr NEWLINE  
      | NEWLINE  
      ;
```

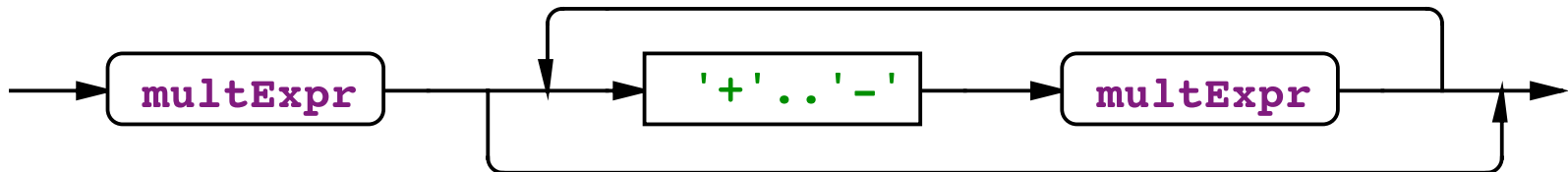




The arithmetic grammar

A statement consists of expressions

```
expr: multExpr (('+'|'-') multExpr)* ;
```



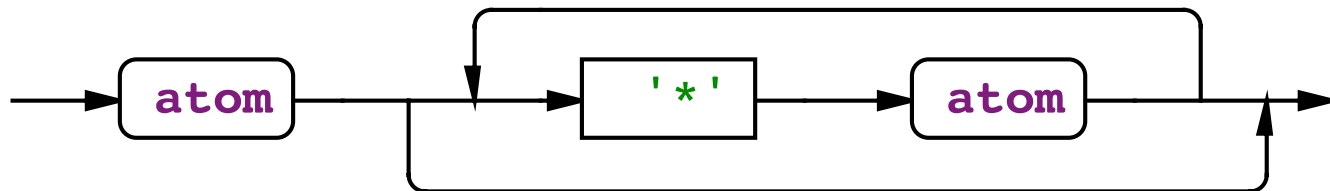


The arithmetic grammar

Let's multiply, when needed:

multExpr

: atom ('*' atom)* ;





The arithmetic grammar

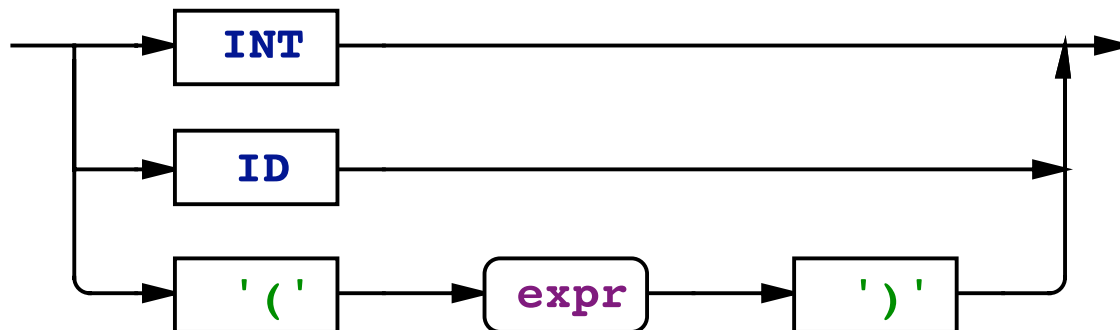
The base components are either an:

atom: INT

| ID

| '(' expr ')'

;

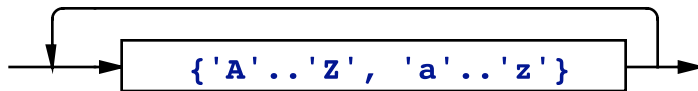




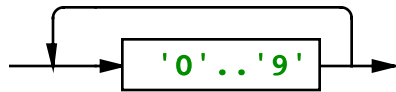
The arithmetic grammar

And in the end everything is a token:

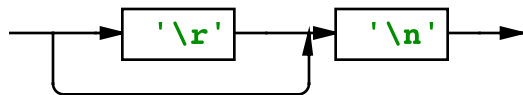
ID : ('a'..'z' | 'A'..'Z')+ ;



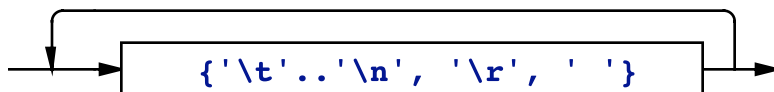
INT : '0'..'9'+ ;



NEWLINE: '\r'? '\n' ;



WS : (' |\t|\n|\r')+ {skip();} ;





How about whitespace?

- Note the skip() syntax in our example.
 - The skip() action is a pre-made action that will make your resulting parser ignore the current token.
- You can also redirect tokens to a “hidden” stream
 - WS : (' |\t|\n|\r')+ {\$channel = HIDDEN;};
 - The hidden channel will retain the tokens, but they will not show up in the default token stream to your parser.



The arithmetic grammar

- Now let's see this in action
 - Write the grammar to a file
 - Compile the grammar with ANTLR
 - Write a bit of code to run our grammar
 - Compile the java code
 - Execute
- Note the error handling ANTLR provides out of the box.



The arithmetic grammar: Actions

- The previous example only validated the input.
- Now we'll add some functionality with actions
- I'll need to do a couple of things
 - Define some storage mechanism
 - Add action clauses to our current grammar



The arithmetic grammar: Actions

Let's add our storage mechanism

```
@header {  
import java.util.HashMap;  
}
```

```
@members {  
HashMap memory = new HashMap();  
}
```



The arithmetic grammar: Actions

The stat statement

```
stat:  expr NEWLINE
      {System.out.println($expr.value);}
      |  ID '=' expr NEWLINE
      {memory.put($ID.text, new
                  Integer($expr.value));}
      |  NEWLINE ;
```



The arithmetic grammar: Actions

The expression statement

expr returns [int value]

: e=multExpr {\$value = \$e.value;}

('+' e=multExpr {\$value += \$e.value;}

| '-' e=multExpr {\$value -= \$e.value;}

)*;

Was:

expr: multExpr (('+'|'-') multExpr)* ;



The arithmetic grammar: Actions

The multExpr statement

multExpr returns [int value]

: e=atom {\$value = \$e.value;}

('*' e=atom {\$value *= \$e.value;})*;

Was:

multExpr

: atom ('*' atom)* ;



The arithmetic grammar: Actions

- And finally our atom statement

atom **returns [int value]**

```
: INT {$value = integer.parseInt($INT.text);}
```

```
| ID {Integer v = (Integer)
```

```
memory.get($ID.text);
```

```
if ( v!=null ) $value = v.intValue();
```

```
else System.err.println("undefined “
```

```
+”variable” +$ID.text);}
```

```
| '(' expr ')' {$value = $expr.value;};
```



The arithmetic grammar: Actions

- Let's see the actions in action
 - Recompile the grammar
 - Execute
- Note how the grammar now actually does something

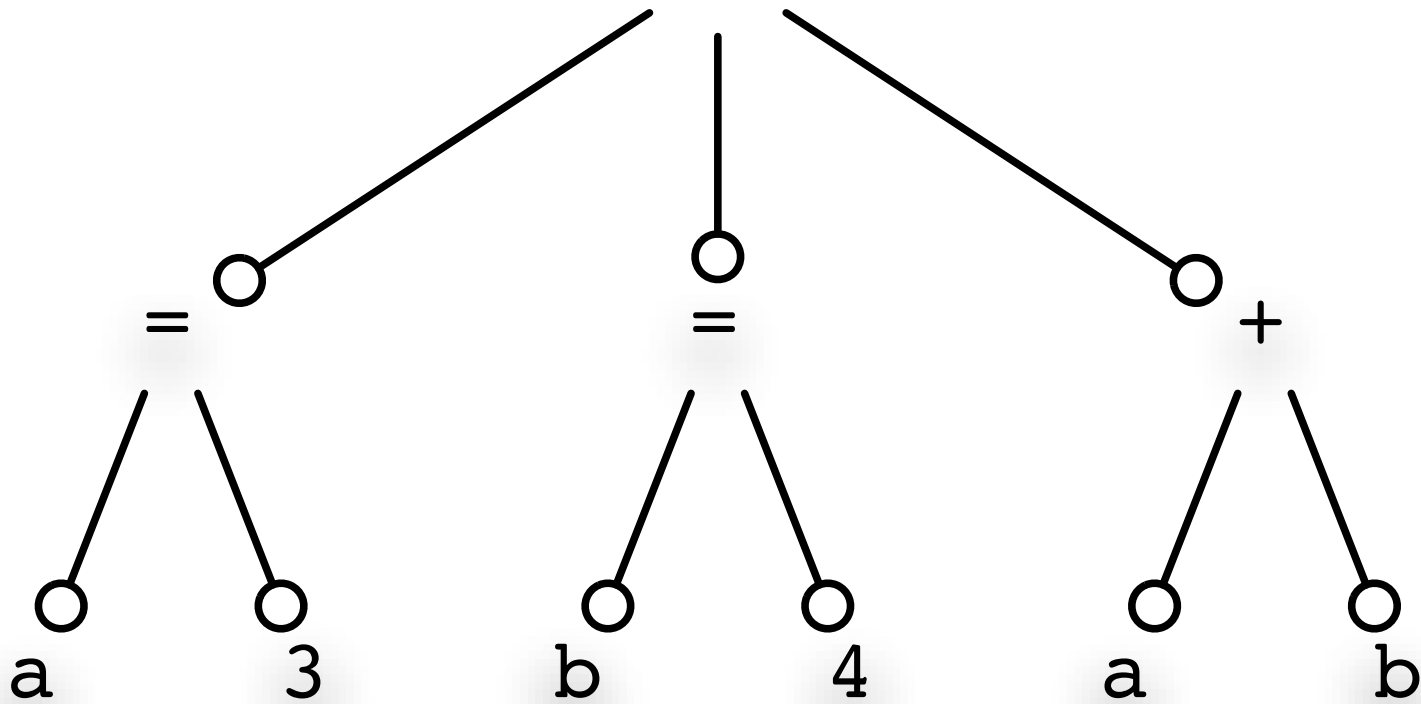


The arithmetic grammar: AST

- Next we will do the same with AST based processing
- While our example is trivial, using an AST can have great benefits on more complex problems.
- We need to add AST specific processing instructions.
- We also need to create an AST grammar that will help us convert the generated tree into output.



The arithmetic grammar: AST





The arithmetic grammar: AST

First we need to add some processing instructions

```
options {  
    output=AST;  
    ASTLabelType=CommonTree; // type of  
    $stat.tree ref etc...  
}
```



The arithmetic grammar: AST

The prog statement contains an action for debugging:

```
prog: ( stat  
      {System.out.println($stat.tree.toStringTree());}  
      )+ ;
```



The arithmetic grammar: AST

The stat statement with AST syntax:

```
stat:  expr NEWLINE      -> expr
      |  ID '=' expr NEWLINE -> ^('=' ID expr)
      |  NEWLINE         -> ;
```



The arithmetic grammar: AST

Add the AST syntax here:

```
expr: multExpr (('+'^|'-'^) multExpr)*
```

```
;
```

```
multExpr
```

```
: atom ('*' ^ atom)*
```

```
;
```

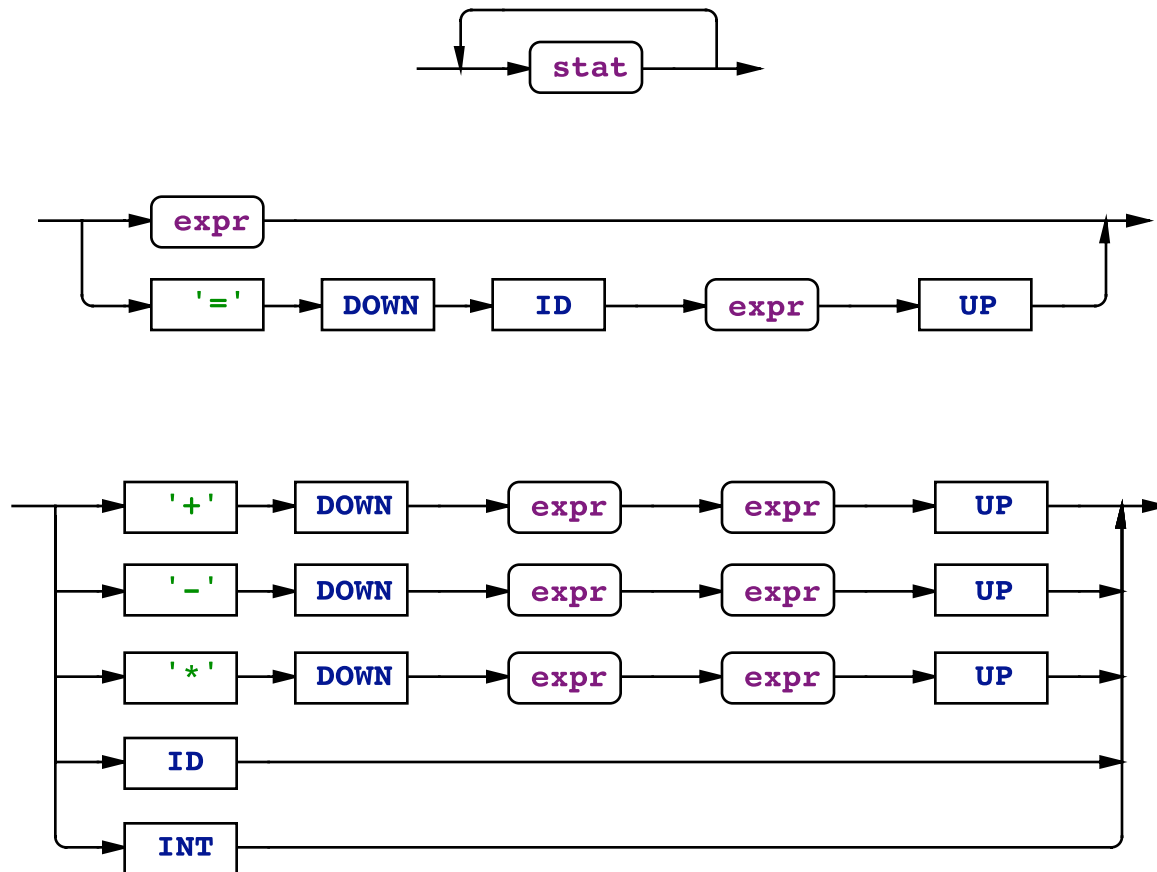


The arithmetic grammar: AST

- Now we need to create a tree grammar to convert our AST into something usefull (output in our case)
- I'll walk us through that grammar in my editor. It's a bit more involved.



The arithmetic grammar: AST





How to: execute a parser

- Command-line invocation
 - Wrap generated parser in a main method, as shown in presentation
- Call from Application code
 - Call generated parser from code, similar approach as calling from the main method.
- For performance
 - No pre-caching required
 - It's all just code



How to: javax.scripting

- Java provides scripting interface since version
- JavaScript Rhino is default implementation
- Any parser you create, can be used
- Implement
 - **AbstractScriptEngine**
 - **ScriptEngineFactory**



What we did not cover today

- Error handling
- AST Operators, allows more behaviour when modifying AST trees.
- ANLRWorks (I'll demo if time permits)
- StringTemplate
 - Template engine written in Java
 - Alternative to printlns
 - Cleaner separation between logic and output



What we did not cover today

- Lookahead
 - backtrack = true, k = look-ahead-distance
- gUnit



The Pragmatic Programmers

The Definitive ANTLR Reference

Building Domain-Specific Languages



Terence Parr

DEFINITE READ

.nl.
jug



Questions