## Demo – Simple bracketed list recognizer

- navigate to /Users/Robert/Documents/ANTLR/code/parsing/recursive-descent

- show grammar for NameList.g

- show ant build script

- explain ant build

- NameList.g => \*.java

- \*.java => \*.class

- java Test

- ‘ant clean’

- ‘ant run’

# ANTLRWorks

## Language - Simple Expression Translator

Steps:

1. Build a parse grammar to recognize valid/invalid arithmetic expressions

2. Add actions to evaluate and print results for the expressions

3. Augment the parse grammar to build intermediate tree data structure. Build a tree grammar to walk the tree adding actions to evaluate and print results

## 1. Build a parse grammar to recognize valid/invalid arithmetic expressions

I've pre-entered the parse grammar for the arithmetic expression.

prog will recognize a list of one or more statements.

stat can be an expr followed by a NEWLINE

an assignment of a n expr to an ID

or just a NEWLINE

expr can be a multExpr followed by zero or more adds or subtracts of a multExpr's

multExpr can be an atom followed by zero or more multiplies of atom

atom can be a INT (whole number) or an ID (identifier) or and parenthetical expr

Note the recursion

Select a rule and demonstrate its railroad syntax diagram.

Why have expr rule and multExpr rule? Ans. To show multiplication has higher precedence than addition or subtraction. Notice that the higher the precedence the lower the rule appears in the grammar

Are decimal numbers accepted? Ans. No only integers.

ANTLR will generate a parser and lexer in java that can be compiled. Show generated files.

.../Documents/PSGV-DSL/ANTLR-projects/SimpleExpr/output.

Expr1.tokens list of token types

Expr1Lexer.java recursive descent lexer

Expr1Parser.java recursive descent parser

recursive method for every grammar rule  
 token references are translated to match(TOKEN) calls

pushFollow has to do with error recovery

Activate Interpreter pane and input

2+3 (followed by CR)

then click run icon and behold the syntax tree exposed

Next enter 2&3 and click run icon and see error report on console window

Now try more complex ‘program’:

a = 1

b = 2

c = 3

a + b \* c

(a + b) \* c

## 2. Add actions to evaluate and print results for the expressions

Now we're going to add actions to the grammar. The actions are going to do things during the parse.

1. Define a hashtable (Fowler's Symbol Table) to store a variable to value mapping

2. Upon expr recognition print the result of evaluating it.

3. Upon assignment evaluate the right hand side and map the variable on the left to the result storing the results in Symbol Table.

etc.

These actions are going to be implemented in Java and there are ANTLR rules for how to intermix the grammar rules and the actions.

Open Expr2.g grammar.

* Point out the import for hash map.
* Point out the action for expr recognition (System.out.println($expr.value)); $expr.value is requesting the value of the expression from ANTLR.
* Point out the action for ID '=' expr rule.

Note that the actions are enclosed between curly braces. Are these closures? Ans. No

Also rules have a return value.

Activate the Debugger. Enter **CMD-D to enter input.**

a=3

b=5

2+a\*b

Hit the Step icon and step through the parsing of the program and its actions. Fast forward by hitting the FF icon.

See the result displayed in the output window.

## 3. Build an AST and then walk the tree with a tree grammar

Reuse the parser grammar to build an intermediate data structure replacing the embedded actions with tree construction rules. Write a tree parser to walk the tree and execute embedded actions.

Big difference here is the construction of the AST (Abstract Syntax Tree).

What we had before was interpretation of the parse tree. This doesn't scale well to sophisticated languages that have method calls and while statements. Why doesn't it scale? Ans. Because of the need to retranslate the statements over and over again.

Walking an intermediate form tree is much faster than reparsing an input program.

Show a tree for 3 + 4.

Let's agree on a text representation for the tree (prefix notation here): (+ 3 4).

Show a tree for 3+4\*5.

The text representation will be (+3 (\* 4 5))

An AST is to be distinguished from a parse tree which represents the sequence of rule invocations used to match an input stream. The leaves of a parse tree are input symbols and the non-leaves are rule names.

Now we will instruct ANTLR to generate a tree grammar as it parses the input. The tree grammar is more compact and can be manipulated to make it even more efficient.

Open Expr3.g. Point out 'output=AST;' which instructs ANTLR to generate an AST for the entire program. 'CommonTree' indicates a Java tree.

^ indicates generate a tree node.

! indicates skip generation.

-> points to a rule for generating a tree node

Point out stat parsing and tree generation rules

* expr NEWLINE -> expr
* ID '=' expr NEWLINE -> ^('=' ID expr)
* NEWLINE ->

Start the debugger and input the same ‘program’ and let it run to completion. Compare and contrast the parse tree with the grammar tree. Note the output window contents.

This is where ANTLR gets tricky. We’ve generated a tree and we want to traverse the tree to do useful things like constant-folding and other optimizations.

Now the ANTLR guy has recognized that tree traversal is much like tree parsing so why not re-use ANTLR’s parser generation capability to parse the generated tree and do some generation stuff in the process.

ANTLR reduces tree parsing to conventional one-dimensional token stream parsing.

AST for tree (+ 3 4) is represented by + DOWN 3 4 UP. Let’s see a tree grammar that recognizes this pattern. Open eval.g and point out the tree parsing with actions.

What is the net effect of the tree grammar? Ans. compute the value of each statement.

Now to process the tree grammar we have to leave ANTLRWorks and write our own code.

Switch to command line and cd to /Users/Robert/Documents/PSGV-DSL/ANTLR-projects/SimpleExpr/

List java programs:

Cd to cli and run: java TestEval.g

Why multiple tree walkers?

- constant folding

- common subexpression elimination

- code hoisting

- dead code elimination