PA1 - ANTLR4

2019312601
이찬구

How to describe grammar

Lexer Rule

Allowed tokens will be NUMBER(includes real number and interger), VARIABLE, FUNCTION NAME(or Reserved Word). Also, by adding white space and newline rule, parser can ignore them. (Technically, parenthesis[()], equal sign[=], arithmetic operator[+-*/] should be included in lexer rule as well, but Antlr4 allows to write them in parse rule.)

```
NEWLINE: [\r\n]+ ;
INT: [0-9]+ ;
REAL: [0-9]+'.'[0-9]* ;
FUNC0: 's''q''r''t' ;
FUNC1: ( 'm''i''n' | 'm''a''x' | 'p''o''w' ) ;
WS: [ \t\r\n]+ -> skip ;
VAR: [a-zA-Z][a-zA-Z]* ;
```

The reason why seperate rules of function name `sqrt` and the rest (`min`, `max`, `pow`) is because their cardinality of arguments differs so that parser should apply different rules to them.

Parser Rule

Below four expressions should be accepted

- four arithmetic operations (ADD, SUB, MUL, DIV)
- 4 functions in java.lang.Math (max, min, pow, sqrt)
- variable assignment

First of all, four arithmetic operations are described as below.

```
expr  = expr ('*'|'/') expr   # infixExpr
      |  expr ('+'|'-') expr   # infixExpr
```

Operand of arithmetic operations can be not only numbers but expression which consist of arithmetic operation, math function, or variable.

There can be ambiguous grammar in this rule. However, Antlr4 parser chooses preceded operand in a OR(|) operation, so that multiplication and division can be caculated before addition and substraction.

Second, four math functions and parenthesis are described as below.

```
| FUNC0 '(' expr ')'          # func0Expr
| FUNC1 '(' expr ',' expr ')' # func1Expr
| '(' expr ')'                # parensExpr
```

FUNC0 token is function sqrt which has one parameter, and FUNC1 tokens are functions max, min, pow which has two parameters. (Although assignment specification says that only numbers are considered as parameter of math function, I allowed expressions as parameters.)

Third, I put number and variable as terminal values of this parser rule.

```
| num        # numberExpr
| VAR        # varExpr
```

To handle negative values of number, I seperated numberExpr parsing rule as below

```
num  : INT    #posNum
     | REAL    #posNum
     | '-' INT    #negNum
     | '-' REAL   #negNum
```

Fourth, variable assignment is described as below.

```
assign : VAR '=' num
```

arithmetic operations and math functions can be combined, and variable assignment is considered as a single line. Our first parser rule will be below.

```
prog : ( (expr | assign) ';' NEWLINE?)*;
```

How to code AST builder

The process flow of Program.java is below.

1. make parse tree → 2. translate to AST (ExprTree) → 3. print AST → 4. evaluate AST

The parse tree is fully made by Antlr, so let's take a look at how to translate it to AST.

1. translate to AST (BuildAstVisitor.java)

Nodes of AST will be instance of one of the following table's elements. Therefore, generic of visitor class should be <ExprNode> which is parent class of all of the below classes.

	Cardinality of children	data type
ExprTree (Root Node)	as many as input lines	-
infixNode	2 (left operand, right operand)	"+" | "-" | "*" | "/"
func0Node (sqrt)	1 (parameter1)	"sqrt"
func1Node (min, max, pow)	2 (parameter1, parameter2)	"min" | "max" | "pow"
varNode	0 (TERMINAL)	variable value
numNode	0 (TERMINAL)	operand value
assignNode	2 (variable name, variable value)	"0"

Like upper table says, `ExprNode` class has member variable `ExprNode left`, `ExprNode right` (which are children) and `String data`. In addition, `ExprTree` which is a root node, can have multiple children, so that it uses `List<Expr>` children instead of `ExprNode left` and `ExprNode right`.

Now we traverse the parse tree from the root of it recursively , and push result to ExprTree which is a root node of AST. Let's have an example of situation when meeting infix expression during traverse.

```
@Override public ExprNode visitInfixExpr(ExprParser.InfixExprContext ctx) {
    InfixNode infix = new InfixNode();
    infix.left = visit(ctx.getChild(0));
    infix.data = ctx.getChild(1).getText();
    infix.right = visit(ctx.getChild(2));
    return infix;
  }
```

According to the visitor pattern, `visit()` invokes the accept method, and the accept method executes the appropriate visiting method(like `visitInfixExpr()`) depends on the type of node to visit. Therefore, the InfixNode obtains left and right child in recursive way (**base condition with NumNode or VarNode**), and when both are obtained, the infixNode is returned.

Let's have an another example of situation when meeting parenthesis.

```
@Override public ExprNode visitParensExpr(ExprParser.ParensExprContext ctx) {
    return visit(ctx.getChild(1));
  }
```

Just pass expression inside the parenthesis to hide them in AST.

2. print AST (AstCall.java)

Now the nodes of the AST have operator (or function name) as data, and operand(or parameter) as children. Therefore we should use **preorder traversal** of binary tree in order to print tree as specified form.

Every time traverse mehod recursively called, the depth variable is passed with adding 1 (Following is the code). Then we can exploit it when printing indent.

```
public void Call(ExprNode node, int depth){
    for(int i=0; i<depth; i++){
      System.out.print("\t");
    }
    printz(node);
    if(node.left!=null) Call(node.left, depth+1);
```

```
    if(node.right!=null) Call(node.right, depth+1);
  }
```

3. calculate AST (Evaluate.java)

Calculating AST is almost same procedure with printing it, but one more step will be needed, which is variable memorization. It can be done by simply define a `list<Variable>`. Variables will be memorized in list when the recursion meeting AssignNode and will be read when recursion meeting VarNode.

```
....
else if(node instanceof VarNode){
    for(Variable var:vars){
      if(node.data.equals(var.name)){
        return var.value;
    }
}else if(node instanceof AssignNode){
    Variable newVar = new Variable((node.left).data, evaluate(node.right));
    vars.add(newVar);
    return 0.0;
}
....
```

Reference

https://github.com/antlr/antlr4/blob/master/doc/index.md

https://www.youtube.com/watch?v=-FdD_xzNFL4&list=PL5dxAmCmjv_4FGYtGzcvBeoS-BobRTJLq

https://www.antlr.org/api/Java/org/antlr/v4/runtime/