Description of 4302 Example Program

By Xiang Chen (Echo)

Feb 7, 2020

Contents

1 Alias

In this description, I will use two alias:

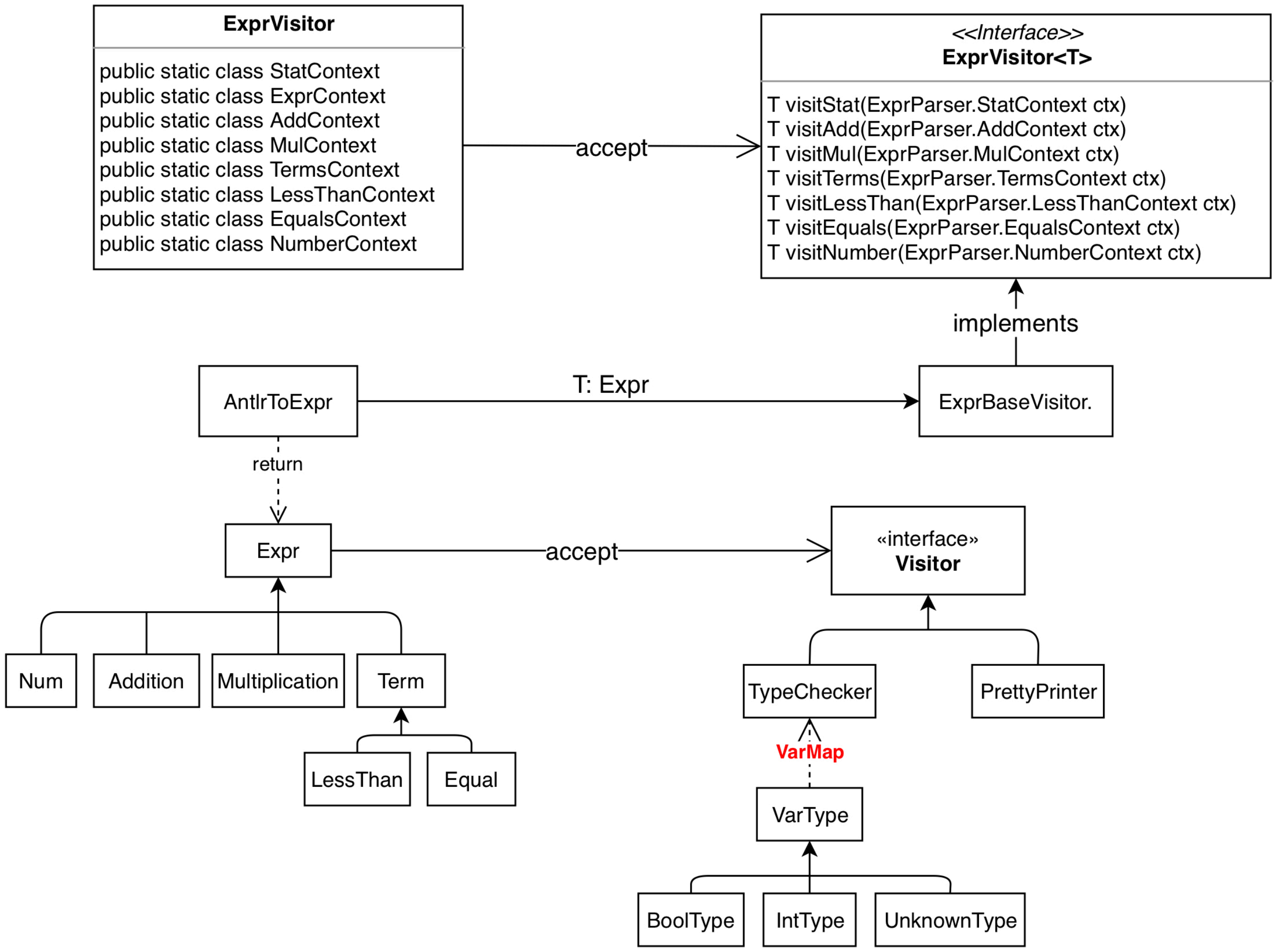
```
alias antlr4='java -jar /usr/local/lib/antlr-4.7.2-complete.jar'

alias grun='java org.antlr.v4.gui.TestRig'
```

Where **antlr4** will be used to run the Antlr complete JAR file (the path will be the location where you put your Antlr JAR file), and **grun** will be used to run the Antlr build-in TestRig for testing the grammar.

2 Visitor Version

Antlr 4 has a build-in visitor pattern for transforming Antlr AST into whatever structure you want. Here I create my own structure for the expression hierarchy.

Below is the overall structure of the whole program:



ExprVisitor
public static class StatContext
public static class ExprContext
public static class AddContext
public static class MulContext
public static class TermsContext
public static class LessThanContext
public static class EqualsContext
public static class NumberContext
accept
<<Interface>>
ExprVisitor<T>
T visitStat(ExprParser.StatContext ctx)
T visitAdd(ExprParser.AddContext ctx)
T visitMul(ExprParser.MulContext ctx)
T visitTerms(ExprParser.TermsContext ctx)
T visitLessThan(ExprParser.LessThanContext ctx)
T visitEquals(ExprParser.EqualsContext ctx)
T visitNumber(ExprParser.NumberContext ctx)
implements
AntlrToExpr
T: Expr
ExprBaseVisitor.
return
Expr
accept
«interface»
Visitor
Num
Addition
Multiplication
Term
LessThan
Equal
TypeChecker
PrettyPrinter
VarMap
VarType
BoolType
IntType
UnknownType

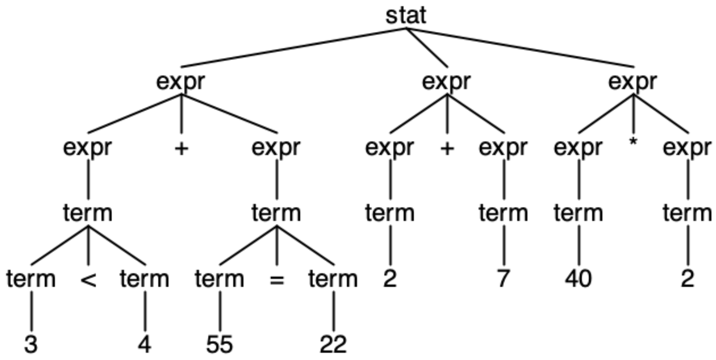
2.1 Steps of Compiling the Grammar File

Below are the steps of compiling the grammar file for future modification, such as creating our own Expr hierarchy:

1. `antlr4 -visitor Expr.g4`
   This step will compile the grammar file. Remember to add the `-visitor` argument, it will automatically generate the necessary classes for the Antlr build-in visitor pattern.

2. `javac Expr*.java`
   This step will compile all the Java files that generated by Antlr.
   After this step, you could feel free to import everything into Eclipse and create your own Expr hierarchy and other necessary classes.

3. `grun Expr stat -gui test01.txt`
   This extra step could be used to test if the input file is consistent with the grammar. It will call the Antlr build-in TestRig for testing. The -gui argument will generate a figure for the Antlr parse tree of user input. Below is an example:



2.2 Grammar

The grammar is called `Expr`, and the grammar file will have the extension of **.g4**, also the file name should have the same name as the grammar name.

Below is the grammar:

```
grammar Expr;
// this is the start symbol
stat : expr+;


expr
  : expr MUL expr       # Mul
  | expr ADD expr       # Add
  | term                # Terms
  ;


term
  : term LESSTHAN term    # LessThan
  | term EQUAL term       # Equals
  | NUM                   # Number
  ;


ADD : '+';
MUL : '*';
EQUAL : '=';
LESSTHAN : '<';


NUM : '0'|'-'?[1-9][0-9]*;


COMMENT : '--' ~[\r\n]* -> skip;
WS  :   [ \t\n]+ -> skip ;
```

2.3 Explaination

- Those **labels** are used for generating Antlr build-in visitor methods.
- For example, if you use the label `#Mul`, Antlr will automatically generate the method called `visitMul()` in the class `ExprBaseVisitor<T>` and interface `ExprVisitor<T>`.
- Then you can create your own class, let it inherit the `ExprBaseVisitor<T>` class, and set the generic parameter as what you want, ane rewrite those methods.
- Here I create my own class called `AntlrToExpr`, set up the generic parameter as `Expr` and rewrite the necessary methods.

2.4 Intermediate Class: AntlrToExpr

This intermediate class will transform the Antlr AST into my Expr object.

Below is the code for my `AntlrToExpr` class:

```
package expr;

import antlr.*;
import antlr.ExprParser.*;
import expr.composite.*;

public class AntlrToExpr extends ExprBaseVisitor<Expr>{


  @Override
  public Expr visitAdd(AddContext ctx) {
    return new Addition(visit(ctx.expr(0)), visit(ctx.expr(1)));
  }

  @Override
  public Expr visitMul(MulContext ctx) {
    return new Multiplication(visit(ctx.expr(0)), visit(ctx.expr(1)));
```

```java
  }

  @Override
  public Expr visitTerms(TermsContext ctx) {
    return visit(ctx.term());
  }

  @Override
  public Expr visitLessThan(LessThanContext ctx) {
    return new LessThan(visit(ctx.term(0)), visit(ctx.term(1)));
  }

  @Override
  public Expr visitEquals(EqualsContext ctx) {
    return new Equal(visit(ctx.term(0)), visit(ctx.term(1)));
  }

  @Override
  public Expr visitNumber(NumberContext ctx) {
    return new Num(ctx.NUM().getText());
  }
}
```

- Here I set up the generic parameter as `Expr`, so every method in the class will return an Expr object, then I'll call my `TypeChecker` class to traverse the Expr object.
- In the grammar, each of `expr MUL expr, expr ADD expr` contains two `expr` rules, we could refer to them as `expr(0)` and `expr(1)`.
- Similarly, each of `term LESSTHAN term, term EQUAL term` contains two `term` rules, we could refer to them as `term0` and `term(1)`.
- For the method `visitNumber`, this rule contains one Lexer rule called `NUM`, then we could call this rule by `ctx.NUM()` directly, and `getText()` method will return the String of the Lexer rule NUM.

2.5 TypeCheker and PrettyPrinter

2.5.1 TypeCheker

The most important part in the Typecheker class is the symbol table called `varMap`, which helps to check the type for every small expression and constant.

Below is the code for my `TypeCheker` class:

```java
package expr.visitor;

import expr.composite.*;
import java.util.*;
import types.*;

public class TypeChecker implements Visitor{
  // hashmap for type checking
  public static Map<String, VarType> varMap =  new HashMap<String, VarType>();

  // error message
  public List<String> errormsg;

  // constructor
  public TypeChecker() {
    errormsg = new ArrayList<String>();
  }

  // type checker for binary relational expr
  // e.g. =, >, <, >=, <=
  public void relationalBinaryChecker(Expr e) {
    TypeChecker leftChecker = new TypeChecker();
    TypeChecker rightChecker = new TypeChecker();

    e.left().accept(leftChecker);
    e.right().accept(rightChecker);

    PrettyPrinter leftPrinter = new PrettyPrinter();
    PrettyPrinter rightPrinter = new PrettyPrinter();

    e.left().accept(leftPrinter);
    e.right().accept(rightPrinter);

    PrettyPrinter printer = new PrettyPrinter();
    e.accept(printer);

    errormsg.addAll(leftChecker.errormsg);
    errormsg.addAll(rightChecker.errormsg);

    // if left child is not int type of real type
    if (!(varMap.get(leftPrinter.output) instanceof types.IntType)) {
      varMap.put(leftPrinter.output, new UnknowType());
      errormsg.add(leftPrinter.output + " is not integer type. "
          + printer.output + " is not type correct.");

    }

    // if right child is not int type of real type
    if (!(varMap.get(rightPrinter.output) instanceof types.IntType)) {
      varMap.put(rightPrinter.output, new UnknowType());
      errormsg.add(rightPrinter.output + " is not integer type. "
          + printer.output + " is not type correct.");
    }

    // if left and right child are both arithmetic type (no type error)
    // add this expr to the varmap
    varMap.put(printer.output, new BoolType());
  }

  // type checker for binary arithmetic expr
  // e.g. +, −, ∗, /
  public void arithmeticBinaryChecker(Expr e) {
    TypeChecker leftChecker = new TypeChecker();
    TypeChecker rightChecker = new TypeChecker();

    e.left().accept(leftChecker);
    e.right().accept(rightChecker);

    PrettyPrinter leftPrinter = new PrettyPrinter();
    PrettyPrinter rightPrinter = new PrettyPrinter();

    e.left().accept(leftPrinter);
    e.right().accept(rightPrinter);

    PrettyPrinter printer = new PrettyPrinter();
    e.accept(printer);

    errormsg.addAll(leftChecker.errormsg);
    errormsg.addAll(rightChecker.errormsg);

    // if left child is not int type
    if (!(varMap.get(leftPrinter.output) instanceof types.IntType)) {

      varMap.put(leftPrinter.output, new UnknowType());
      errormsg.add(leftPrinter.output + " is not integer type. "
          + printer.output + " is not type correct.");
    }

    // if right child is not int type or real type
    if (!(varMap.get(rightPrinter.output) instanceof types.IntType)) {

      varMap.put(rightPrinter.output, new UnknowType());
      errormsg.add(rightPrinter.output + " is not integer type. "
          + printer.output + " is not type correct.");
    }

    // if both left and right child are int type, the whole expr will be int type
      varMap.put(printer.output, new IntType());
  }

  // arithmetic equal (==)
  @Override
  public void visitEqual(Equal e) {
    relationalBinaryChecker(e);
  }

  // arithmetic greater than (>)
  @Override
  public void visitLessThan(LessThan e) {
    relationalBinaryChecker(e);
  }

  // arithmetic add (+)
  @Override
  public void visitAddition(Addition e) {
    arithmeticBinaryChecker(e);
  }

  // arithmetic multiply (*)
  @Override
  public void visitMultiplication(Multiplication e) {
    arithmeticBinaryChecker(e);
  }

  // int number constant
  @Override
  public void visitNum(Num c) {
    if (!varMap.containsKey(c.name)) {
      varMap.put(c.name, new IntType());
    }
  }
}
```

- This class is very simple, for binary expression, just recursively check if its left child and right child are type correct. If anyone of them is not type correct, then output the error message.
- For the base case, which is constant number, it is always type correct.

2.5.2 PrettyPrinter

The `PrettyPrinter` class will format the user input, then we could use it to output the error message in the TypeChecker class.

Below is the code for my `PrettyPrinter` class:

```
package expr.visitor;

import expr.composite.*;

public class PrettyPrinter implements Visitor{

  public String output;

  public PrettyPrinter() {
    output = "";
  }

  public void visitBinaryExpr (Expr b, String op) {
    PrettyPrinter leftPrinter = new PrettyPrinter();
    PrettyPrinter rightPrinter = new PrettyPrinter();

    b.left().accept(leftPrinter);
    b.right().accept(rightPrinter);
    output = output.concat("(" + leftPrinter.output
      + " " + op + " " + rightPrinter.output + ")");
  }

  @Override
  public void visitEqual(Equal e) {
    visitBinaryExpr(e, "=");
  }

  @Override
  public void visitLessThan(LessThan e) {
    visitBinaryExpr(e, "<");
  }

  @Override
  public void visitAddition(Addition e) {
    visitBinaryExpr(e, "+");
  }

  @Override
  public void visitMultiplication(Multiplication e) {
    visitBinaryExpr(e, "*");
  }

  @Override
  public void visitNum(Num c) {
    output = output.concat(c.name);
  }
}
```

2.6 Main Test Class: TestExpr

This class contains the Main class, and we could use it to parse the user input and run the program.

Below is the code for my `TestExpr` class:

```
package root;

import org.antlr.v4.runtime.*;
import org.antlr.v4.runtime.tree.*;
import java.io.*;
import java.util.*;
import antlr.*;
import expr.*;
import expr.composite.*;
import expr.visitor.*;

public class TestExpr {

  public static void main(String[] args) {

    try {
      String inputFile = null;
          if ( args.length>0 )
            inputFile = args[0];

          InputStream is = System.in;

          if ( inputFile!=null ) {
              is = new FileInputStream(inputFile);
          }

          // parse the input
      ANTLRInputStream input = new ANTLRInputStream(is);
      ExprLexer lexer = new ExprLexer(input);
      CommonTokenStream tokens = new CommonTokenStream(lexer);
      ExprParser parser = new ExprParser(tokens);

      // tell ANTLR to build a parse tree
      parser.setBuildParseTree(true);
      // parse
      ParseTree tree = parser.stat();

      // Call the intermediate class AntlrToExpr
      AntlrToExpr antlrToExpr = new AntlrToExpr();

      // list that stores the subtree separately
      List<Expr> expr = new ArrayList<Expr>();
      // add user input to the list line by line
       for (int i = 0; i < tree.getChildCount(); i++) {
        expr.add(antlrToExpr.visit(tree.getChild(i)));
      }

       // create new TypeChecker
       TypeChecker checker = new TypeChecker();

       // Recursively accept TypeChecker
       for (int i = 0; i < expr.size(); i++) {
        expr.get(i).accept(checker);
      }

      // check if error msg is empty
      // only when it's empty, call the pretty printer
      if (checker.errormsg.isEmpty()) {
        // Recursively accept TypeChecker
        for (int i = 0; i < expr.size(); i++) {
          PrettyPrinter printer = new PrettyPrinter();
          expr.get(i).accept(printer);
          System.out.println(printer.output + " is type correct.");
        }
      }
      else {
      // print the error message
      for (int i = 0; i < checker.errormsg.size(); i++) {
        System.out.println(checker.errormsg.get(i));
      }
    }
  } catch (Exception e) {
    System.out.println("Exception");
    e.printStackTrace();
    }
  }
}
```

- This Main class will first parse the user input to make sure the user input is consistent with the grammar.
- Then it will call the `TypeChecker` class to do the type checking line by line. Only when there is no error message, it will call the `PrettyPrinter` class, and indicate that the specific line is type correct.

3 Actions version

For this version, we don't need to use the Antlr build-in visitor pattern anymore. Instead, we could write `Embedded Actions` into the grammar directly.

3.1 Grammar

Below is the grammar that contains the embedded actions directly:

```
grammar Expr;
// this is the start symbol

@header {
  //package Action;
  import org.antlr.v4.runtime.*;
  import org.antlr.v4.runtime.tree.*;
  import java.io.*;
  import java.util.*;
  import types.*;
}

stat
  : expr+
    {
      if ($expr.t instanceof types.UnknowType) {
        System.out.println($expr.text + " is not type correct.");
      }
      else {
        System.out.println($expr.text + " is type correct.");
      }
    }
  ;


expr returns [types.VarType t]
  : a=expr MUL b=expr
    {
      if (!($a.t instanceof types.IntType)) {
        $t = new UnknowType();
      }

      if (!($b.t instanceof types.IntType)) {
        $t = new UnknowType();
      }

      else {
        $t = new IntType();
      }
    }
  | a=expr ADD b=expr
    {
      if (!($a.t instanceof types.IntType)) {
        $t = new UnknowType();
      }

      if (!($b.t instanceof types.IntType)) {
        $t = new UnknowType();
      }

      else {
        $t = new IntType();
      }
    }
  | term
    {
      $t = $term.t;
    }
  ;


term returns [types.VarType t]
  : a=term LESSTHAN b=term
    {
      if (!($a.t instanceof types.IntType)) {
        $t = new UnknowType();
      }

      if (!($b.t instanceof types.IntType)) {
        $t = new UnknowType();
      }

      else {
        $t = new BoolType();
      }
    }
  | a=term EQUAL b=term
    {
      if (!($a.t instanceof types.IntType)) {
        $t = new UnknowType();
      }

      if (!($b.t instanceof types.IntType)) {
        $t = new UnknowType();
      }

      else {
        $t = new BoolType();
      }
    }
  | NUM
    {
      $t = new IntType();
    }
  ;


ADD : '+';
MUL : '*';
EQUAL : '=';
LESSTHAN : '<';


NUM : '0'|'-'?[1-9][0-9]*;


COMMENT : '--' ~[\r\n]* -> skip;
WS  :   [ \t\n]+ -> skip ;
```

- The header section will contain the package name, import libraries, etc.
- For each rule, we could use ”{ }” after it to indicate the ”actions” for each rule.
- For this version, I only create the type hierarchy to indicate the type for each sub-rule for type checking.

3.2 TypeChecker Class

This time I call my main class `TypeChecker`, and I also modify the code so that it supports `Interactive Mode`.

Below is the code for my `TypeChecker` class:

```
import org.antlr.v4.runtime.*;
import java.io.*;

public class TypeChecker {
    public static void main(String[] args) throws Exception {
        String inputFile = null;
        if ( args.length>0 ) inputFile = args[0];
        InputStream is = System.in;
        if ( inputFile!=null ) {
            is = new FileInputStream(inputFile);
        }

        BufferedReader br = new BufferedReader(new InputStreamReader(is));
        // get first expression
        String expr = br.readLine();
        // track input expr line numbers
        int line = 1;

        // share single parser instance
        ExprParser parser = new ExprParser(null);
        // don't need trees
        parser.setBuildParseTree(false);

        // while we have more expressions
        while ( expr!=null ) {
            // create new lexer and token stream for each line (expression)
            ANTLRInputStream input = new ANTLRInputStream(expr+"\n");
            ExprLexer lexer = new ExprLexer(input);
            // notify lexer of input position
            lexer.setLine(line);
            lexer.setCharPositionInLine(0);
            CommonTokenStream tokens = new CommonTokenStream(lexer);
            // notify parser of new token stream
            parser.setInputStream(tokens);
            // start the parser
            parser.stat();
            expr = br.readLine();
            // see if there's another line
            line++;
        }
    }
}
```

3.3 Steps of Compiling and Running the Program

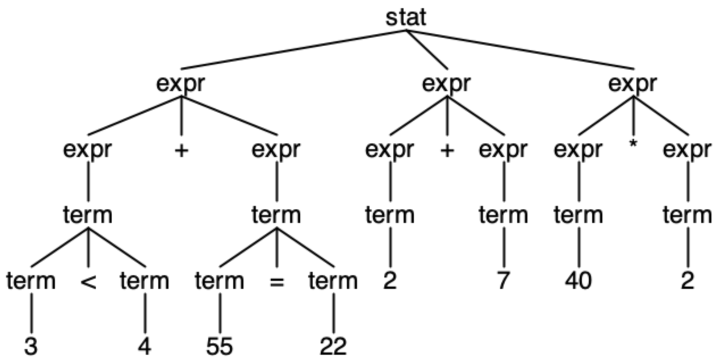
Below are the steps of compiling the grammar file for future modification, such as creating our own Expr hierarchy:

1. `antlr4 -no-listener Expr.g4`
   This step will compile the grammar file. We don't need the build-in visitor pattern this time. Remember to add the `--no-listener` argument, it means that we don't need the build-in Listener pattern as well.

2. `javac Expr*.java TypeChecker.java`
   This step will compile all the Java files that generated by Antlr as well out main class TypeChecker.java.

3. `java TypeChecker test01.txt`
   This step allows you use test01.txt as the input file, and run the main class TypeChecker.java. If you wish to use the interactive mode, simple use `java TypeChecker` instead. The program will output the result when you hit the **return** button.

4. `grun Expr stat -gui test01.txt`
   This extra step could be used to test if the input file is consistent with the grammar. It will call the Antlr build-in TestRig for testing. It's the same as the visitor pattern. The -gui argument will generate a figure for the Antlr parse tree of user input. Below is an example:



4 Comparison

- For small programs, using actions will be much simpler than using the visitor pattern.
- Actions might be more efficient because it won't generate a parse tree. In resource-critical applications, we might not want to waste the time or memory needed to build a parse tree.
- The visitor pattern might be more complicated, but if the program has a large structure and you want to build a cleaner hierarchy, then it might be better to use the Antlr build-in visitor pattern, and let the intermediate class returns the object you want for future modification.