Sep 13th Report

By Xiang Chen (Echo)

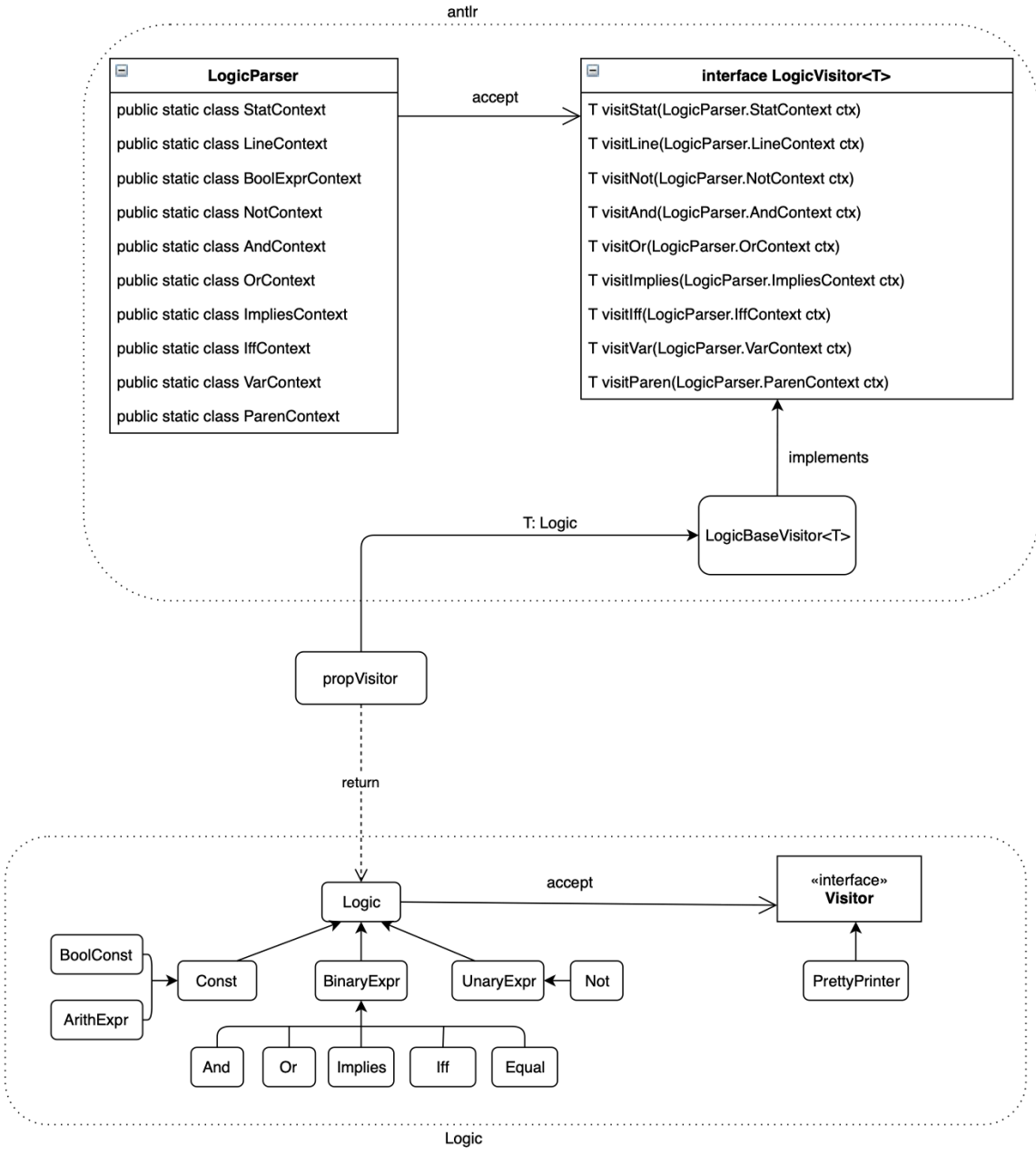
Sep 13, 2019

Contents

1 My Logic Structure

1.1 Customized Structure Diagram

Below is the whole structure of my program, including **Antlr** generated classes and my customized Logic structure:



LogicParser, **LogicVisitor<T>**, and **LogicBaseVisitor<T>** are Antlr generated classes that implement its built-in visitor pattern.

I create **propVisitor** to traverse the Antlr generated parse tree, and transform it into my Logic object.

Below is the code of my **propVisitor** class:

```
1 public class PropVisitor extends LogicBaseVisitor<Logic>{
```

```
2
3   // map that stores variable's name, type, and value
4   public Map<String, Pair<VarType, VarValue>> varMap = new HashMap<String
        , Pair<VarType, VarValue>>();
5
6   // single boolean declaration
7   public Logic visitSingleBool(LogicParser.SingleBoolContext ctx) {
8     // if the variable has not been declared before, add it to the map
9     if (!varMap.containsKey(ctx.VAR().getText())) {
10      varMap.put(ctx.VAR().getText(), new Pair<VarType, VarValue>(new
            BoolType(), new Unspecified()));
11    // else return the error msg
12    }else {
13      System.out.println("Variable " +ctx.VAR().getText() + " is already
            declared as "
14           + ((IntType) varMap.get(ctx.VAR().getText()).a).getStr()
15           + ", you cannot declare it twice.");
16    }
17    return null;
18  }
19
20  // single int declaration
21  public Logic visitSingleInt(LogicParser.SingleIntContext ctx) {
22    // if the variable has not been declared before, add it to the map
23    if (!varMap.containsKey(ctx.VAR().getText())) {
24      varMap.put(ctx.VAR().getText(), new Pair<VarType, VarValue>(new
            IntType(), new Unspecified()));
25    // else return the error msg
26    }else {
27      System.out.println("Variable " +ctx.VAR().getText() + " is already
            declared as "
28           + ((BoolType) varMap.get(ctx.VAR().getText()).a).getStr()
29           + ", you cannot declare it twice.");
30    }
31    return null;
32  }
33
34  // evaluate the boolean expression
35  public Logic visitEvalBoolExpr(LogicParser.EvalBoolExprContext ctx) {
36    if (visit(ctx.boolExpr()) != null) {
37    return visit(ctx.boolExpr());
38    }
39    return null;
40  }
41
42  // define the atom
43  public Logic visitBoolVar(LogicParser.BoolVarContext ctx) {
44    // check is the variable has been declared
45    if (varMap.containsKey(ctx.getText())) {
46      // check if the variable is the right type
47      if (varMap.get(ctx.getText()).a.getClass().getName() == "types.
            BoolType") {
48        return new BoolConst(ctx.getText());
49      // if the variable is not the right type, set up the error msg
50      }else {
51        System.out.println("Variable " + ctx.getText() + " is not boolean
              type.");
52        return null;
53      }
54    // if the variable is not declared, set up the error msg
```

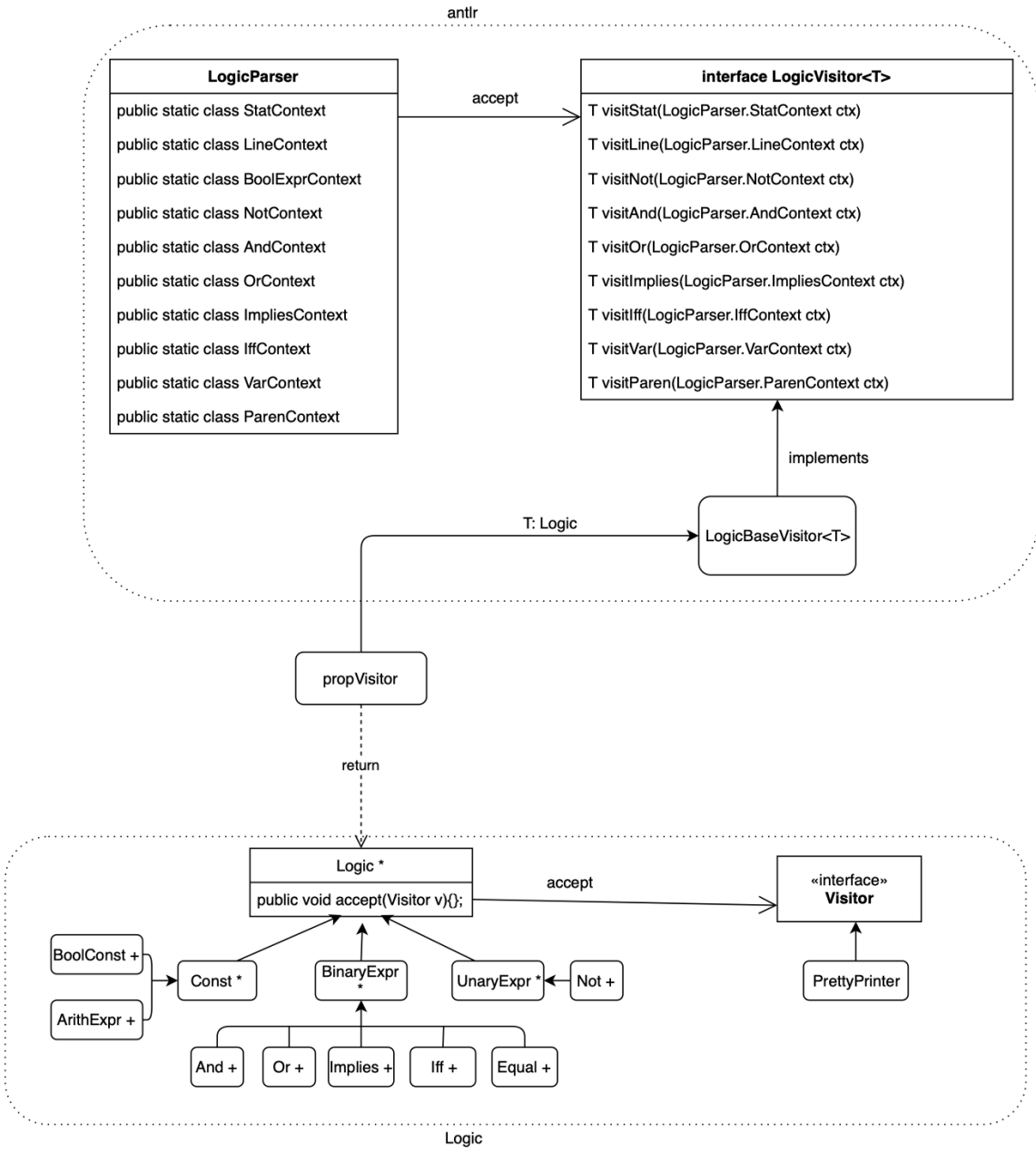
```
55      }else {
56        System.out.println("Variable " + ctx.getText() + " is not declared.
              ");
57        return null;
58      }
59    }
60
61
62    // set up the sub-formula of NOT
63    public Logic visitNot(LogicParser.NotContext ctx) {
64      if (visit(ctx.boolExpr()) != null) {
65        return new Negation(visit(ctx.boolExpr()));
66        }
67      return null;
68    }
69
70    // set up the sub-formula for OR
71    public Logic visitOr(LogicParser.OrContext ctx) {
72      if ((visit(ctx.boolExpr(0)) != null) && (visit(ctx.boolExpr(1))) !=
            null) {
73        return new Disjunction(visit(ctx.boolExpr(0)), visit(ctx.boolExpr
              (1)));
74        }
75      return null;
76    }
77
78    // set up the sub-formula for IMPLIES
79    public Logic visitImplies(LogicParser.ImpliesContext ctx) {
80      if ((visit(ctx.boolExpr(0)) != null) && (visit(ctx.boolExpr(1))) !=
            null) {
81        return new Implication(visit(ctx.boolExpr(0)), visit(ctx.boolExpr
              (1)));
82      }
83      return null;
84    }
85
86    // set up the sub-formula for IFF
87    public Logic visitIff(LogicParser.IffContext ctx) {
88      if ((visit(ctx.boolExpr(0)) != null) && (visit(ctx.boolExpr(1))) !=
            null) {
89        return new Iff(visit(ctx.boolExpr(0)), visit(ctx.boolExpr(1)));
90      }
91      return null;
92    }
93
94    // set up the sub-formula for AND
95    public Logic visitAnd(LogicParser.AndContext ctx) {
96      if ((visit(ctx.boolExpr(0)) != null) && (visit(ctx.boolExpr(1))) !=
            null) {
97        return new Conjunction(visit(ctx.boolExpr(0)), visit(ctx.boolExpr
              (1)));
98      }
99      return null;
100   }
101
102   // Set up the formula with parentheses
103   public Logic visitParen(LogicParser.ParenContext ctx) {
104     return visit(ctx.boolExpr());
105   }
106 }
```

1.2 Example

For example, assume that there is no syntax error or type error, if the user type in:

```
1 boolean p
2 not p
```

Antlr will generate a parse tree as follows:



For the first line that declares the variable, if there is no type error, I will add it to my symbol table.

For the second line that declares the formula, inside my **propVisitor** class, there are two methods that are related to handle this formula, which are **visitNot** and **visitBoolVar**.

visitNot will return a new Negation Object, **visitBoolVar** will return a new BoolConst Object and link it to the Negation Object's child.

1.3 Pretty Printing

As for my customized Visitor pattern, below is the code for the interface **Visitor**:

```
public interface Visitor {

  void visitNot(Negation l);

  void visitOr(Disjunction l);

  void visitImplies(Implication l);

  void visitBoolConst(BoolConst l);

  void visitIntConst(IntConst l);

  void visitAnd(Conjunction l);

  void visitIff(Iff l);
}
```

Class **PrettyPrinter** implements **Visitor** Class and make those methods all effective:

```
public class PrettyPrinter implements Visitor{

  public String varDecl;
  public String formula;

  public PrettyPrinter() {
    varDecl = "";
    formula = "";
  }

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

    b.left().accept(leftPrinter);
    b.right().accept(rightPrinter);
    varDecl = leftPrinter.varDecl + rightPrinter.varDecl;

    formula = "(assert (" + op + " " + leftPrinter.formula
        + " " + rightPrinter.formula + "))\n" + "(check-sat)";

  }

  public void visitUnaryExpr(UnaryExpr u, String op) {

    PrettyPrinter p = new PrettyPrinter();

    u.child.accept(p);


```

```
    varDecl = p.varDecl;
    formula = "(assert (" + op + " " + p.formula + "))\n" + "(check-sat)"
        ;

  }


  @Override
  public void visitBoolConst(BoolConst l) {

    varDecl = "(declare-const " + l.name + " Bool)\n";

    formula = formula.concat(l.name);

  }

  @Override
  public void visitNot(Negation l) {
    visitUnaryExpr(l, "not");
  }

  @Override
  public void visitOr(Disjunction l) {
    visitBinaryExpr(l, "or");
  }

  @Override
  public void visitImplies(Implication l) {
    visitBinaryExpr(l, "=>");
  }



  @Override
  public void visitIntConst(IntConst l) {
    varDecl = "(declare-const " + l.name + " Int)\n";

    formula = formula.concat(l.name);
  }

  @Override
  public void visitAnd(Conjunction l) {
    visitBinaryExpr(l, "and");
  }

  @Override
  public void visitIff(Iff l) {
    visitBinaryExpr(l, "=");
  }
}
```

Below is the output of my Pretty Printer class based on the previous user input:

```
(declare-const p Bool)
(assert (not p))
(check-sat)
```

User could copy and paste the output into rise4fun to get the result from z3 directly.

2 Modified Grammar

Below is my modified Grammar that could also accept variable declaration with values, such as :

```
boolean p = true
int i = 4
int j = 3 * 4 + 1
```

```
grammar Logic;


stat : line+ ;

line
  : BOOL VAR NEWLINE                # SingleBool
  | INT VAR NEWLINE                 # SingleInt
  | BOOL VAR '=' TRUE NEWLINE       # BoolTrue
  | BOOL VAR '=' FALSE NEWLINE      # BoolFalse
  | INT VAR '=' arithmetic NEWLINE  # IntValueDecl
  | boolExpr NEWLINE                # EvalBoolExpr
  | NEWLINE                         # Blank
  ;


boolExpr
  : NOT boolExpr                # Not
  | boolExpr AND boolExpr       # And
  | boolExpr OR boolExpr        # Or
  | boolExpr IMPLIES boolExpr   # Implies
  | boolExpr IFF boolExpr       # Iff
  | VAR                         # BoolVar
  | '(' boolExpr ')'            # Paren
  | relation                    # Relate
  ;

relation
  : arithmetic EQUAL arithmetic           # Equal
  | arithmetic GREATERTHAN arithmetic     # GreaterThan
  | arithmetic LESSTHAN arithmetic        # LessThan
  | arithmetic GREATEROREQUAL arithmetic  # GreaterOrEqual
  | arithmetic LESSOREQUAL arithmetic     # LessOrEqual
  ;

arithmetic
  : arithmetic (MUL|DIV) arithmetic   # MulDiv
  | arithmetic (ADD|SUB) arithmetic   # AddSub
  | VAR                               # IntVar
  | NUM                               # Num
  | '(' arithmetic ')'                # ArithParen
  ;


BOOL : 'boolean';
INT : 'int';

TRUE : 'true' | 'True';
FALSE : 'false' | 'False';
```

```
50
51 NOT : 'not';
52 AND : 'and';
53 OR : 'or';
54 IMPLIES : '=>';
55 IFF : '<=>';
56
57 EQUAL : '=';
58 GREATERTHAN : '>';
59 LESSTHAN : '<';
60 GREATEROREQUAL : '>=';
61 LESSOREQUAL : '<=';
62
63 MUL : '*';
64 DIV : '/';
65 ADD : '+';
66 SUB : '-';
67
68 COMMENT : '--' ~[\r\n]* -> skip;
69 WS  :   [ \t]+ -> skip ;
70
71 VAR : [a-z][a-zA-Z0-9]*;
72 NUM : [1-9][0-9]*;
73 NEWLINE : '\r'? '\n' ;
```

3 Achivements and Problems

3.1 Achivements

1. I have successfully implement my customized Logic structure and make it working.
2. I also successfully implement the customized Visitor Pattern, and create the PrettyPrinter that outputs the String that could be recognized by z3 directly.
3. I successfully extend my Grammar such that it could also accept variable declaration with values.
4. I found some papers and uploaded to GitHub. (But i'm not sure if it's really good).

3.2 Problems

1. I could't make the Regression Test work
2. Because of the nature of z3, I haven't finish modifying the PrettyPrinter such that it could output the separated files for each formula declaration.

I will keep working and try to make the Regression Test work, and keep extending my Grammar and my Logic structure.