Oct 7th Report

By Xiang Chen (Echo)

Oct 07, 2019

Contents

1 Regression Test

I successfully modified the Python script from previous 3311 course (e.g. Eiffel ETF) and let it worked for my program.

For now, I have 150 regression tests from simple formulas to extremely complicated formulas. Below is the table showing the content of my test cases:

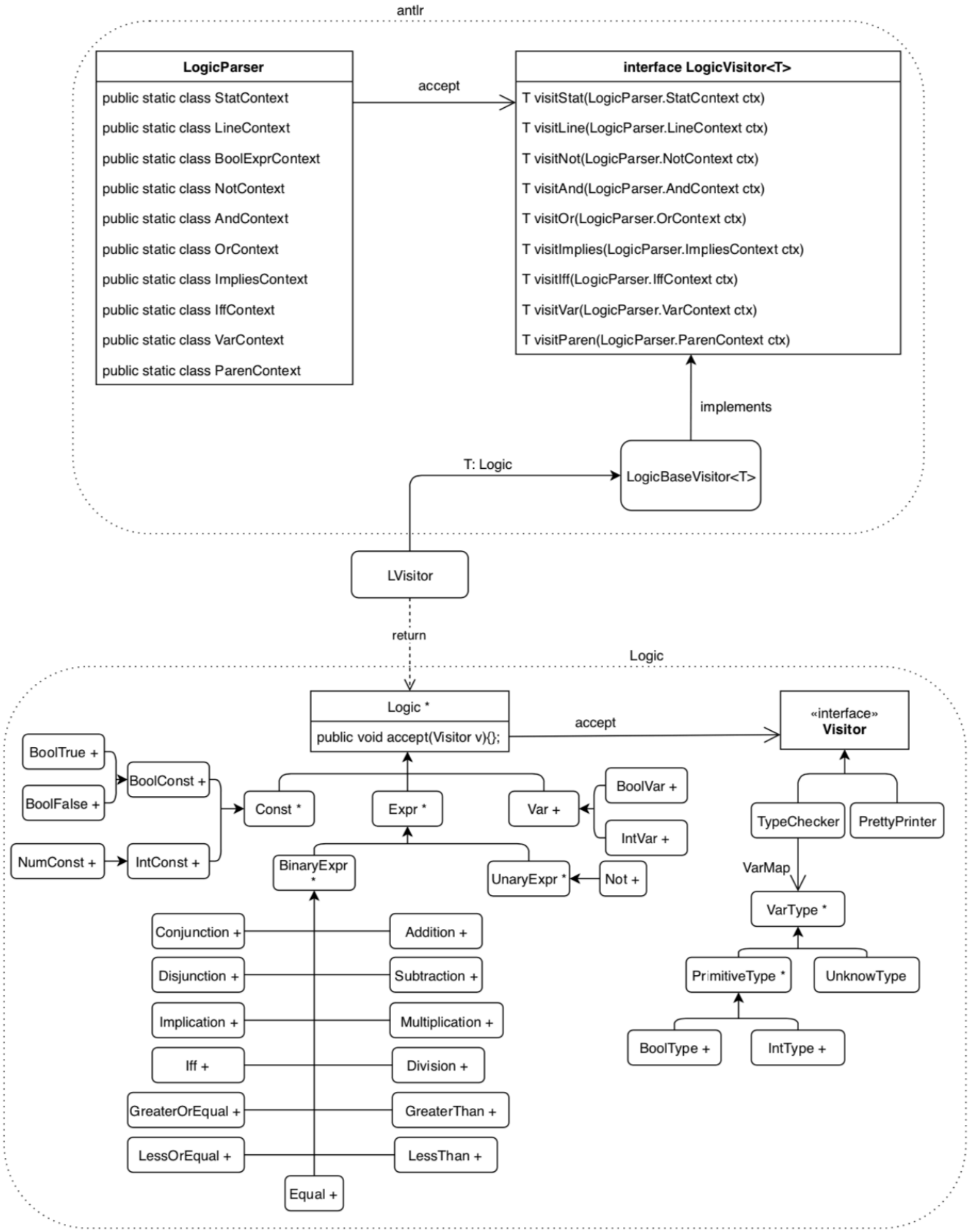
Tests	**Content**
01 - 16	Variable declaration
17 - 26	Negation
27 - 36	Conjunction
37 - 46	Disjunction
47 - 56	Implication
57 - 66	Iff
67 - 72	Equal
73 - 78	Greater than
79 - 84	Less than
85 - 90	Greater or equal
91 - 96	Less or equal
97 - 106	Combination of propositional logic (not, and, or, =>, <=>)
107 - 112	Combination of relational logic (=, >, <, >=, <=)
113 - 118	Combination of propositional logic and greater than
119 - 124	Combination of propositional logic and less than
125 - 130	Combination of propositional logic and greater or equal than
131 - 136	Combination of propositional logic and less or equal than
137 - 142	Combination of propositional logic and equal
143 - 150	Combination of propositional logic and all relational logic

I also have uploaded the whole directory of my regression test as well as my program to **GitHub**.

2 Program Structure

2.1 BON Diagram

Below is the BON diagram of my program structure:



The intermediate class **LVisitor** will transform the Antlr generated AST into my Logic structure AST, then I could let my own Visitor Pattern to have different actions based on my Logic structure AST.

3 LVisitor Class

3.1 Code

Below is the code of the **LVisitor** class:

```
1  package version.logic;
2
3  import org.antlr.v4.runtime.*;
4  import org.antlr.v4.runtime.misc.Pair;
5  import org.antlr.v4.runtime.tree.*;
6
7  import java.io.*;
8  import java.util.*;
9
10 import com.microsoft.z3.*;
11
12 import antlr.*;
13 import antlr.LogicParser.*;
14 import types.*;
15 import values.*;
16 import version.logic.composite.*;
17
18 public class LVisitor extends LogicBaseVisitor<Logic>{
19
20   // uninitialized boolean variable declaration
21   @Override
22   public Logic visitSingleBool(SingleBoolContext ctx) {
23     return new BoolVar(ctx.ID().getText(), 0);
24   }
25
26   // uninitialized int variable declaration
27   @Override
28   public Logic visitSingleInt(SingleIntContext ctx) {
29     return new IntVar(ctx.ID().getText(), 0);
30   }
31
32
33
34   // initialized boolean variable declaration
35   @Override
36   public Logic visitBoolValueDecl(BoolValueDeclContext ctx) {
37     return new BoolVar(ctx.ID().getText(), visit(ctx.boolExpr()), 2);
38   }
39
40   // initialized int variable declaration
41   @Override
42   public Logic visitIntValueDecl(IntValueDeclContext ctx) {
43     return new IntVar(ctx.ID().getText(), visit(ctx.arithmetic()), 2);
44   }
45
46   // verify the formula
47   @Override
48   public Logic visitEvalBoolExpr(EvalBoolExprContext ctx) {
49     return visit(ctx.boolExpr());
50   }
51
52
53
54   // boolean variable verification
55   @Override
```

```
56    public Logic visitBoolVar(BoolVarContext ctx) {
57      return new BoolVar(ctx.ID().getText(), 1);
58    }
59
60    // int variable verification
61    @Override
62    public Logic visitIntVar(IntVarContext ctx) {
63      return new IntVar(ctx.ID().getText(), 1);
64    }
65
66    // boolean true declaration
67    @Override
68    public Logic visitBoolTrue(BoolTrueContext ctx) {
69      return new BoolTrue(ctx.TRUE().getText());
70    }
71
72    // boolean false declaration
73    @Override
74    public Logic visitBoolFalse(BoolFalseContext ctx) {
75      return new BoolFalse(ctx.FALSE().getText());
76    }
77
78    // number declaration
79    @Override
80    public Logic visitNum(NumContext ctx) {
81      return new NumConst(ctx.NUM().getText());
82    }
83
84
85
86
87    // Negation
88    @Override
89    public Logic visitNot(NotContext ctx) {
90      return new Negation(visit(ctx.boolExpr()));
91    }
92
93    // Conjunction
94    @Override
95    public Logic visitAnd(AndContext ctx) {
96      return new Conjunction(visit(ctx.boolExpr(0)), visit(ctx.boolExpr(1)));
97    }
98
99    // Disjunction
100   @Override
101   public Logic visitOr(OrContext ctx) {
102     return new Disjunction(visit(ctx.boolExpr(0)), visit(ctx.boolExpr(1)));
103   }
104
105   // Implication
106   @Override
107   public Logic visitImplies(ImpliesContext ctx) {
108     return new Implication(visit(ctx.boolExpr(0)), visit(ctx.boolExpr(1)));
109   }
110
111   // Iff
112   @Override
113   public Logic visitIff(IffContext ctx) {
114     return new Iff(visit(ctx.boolExpr(0)), visit(ctx.boolExpr(1)));
115   }
```

```
116
117   // parentheses
118   @Override
119   public Logic visitParen(ParenContext ctx) {
120     return visit(ctx.boolExpr());
121   }
122
123   @Override
124   public Logic visitRelate(RelateContext ctx) {
125     return visit(ctx.relation());
126   }
127
128
129
130   // arithmetic equal
131   @Override
132   public Logic visitEqual(EqualContext ctx) {
133     return new Equal(visit(ctx.arithmetic(0)), visit(ctx.arithmetic(1)));
134   }
135
136   // arithmetic greater than
137   @Override
138   public Logic visitGreaterThan(GreaterThanContext ctx) {
139     return new GreaterThan(visit(ctx.arithmetic(0)), visit(ctx.arithmetic(1)));
140   }
141
142   // arithmetic less than
143   @Override
144   public Logic visitLessThan(LessThanContext ctx) {
145     return new LessThan(visit(ctx.arithmetic(0)), visit(ctx.arithmetic(1)));
146   }
147
148   // arithmetic greater or equal
149   @Override
150   public Logic visitGreaterOrEqual(GreaterOrEqualContext ctx) {
151     return new GreaterOrEqual(visit(ctx.arithmetic(0)), visit(ctx.arithmetic(1)));
152   }
153
154   // arithmetic less or equal
155   @Override
156   public Logic visitLessOrEqual(LessOrEqualContext ctx) {
157     return new LessOrEqual(visit(ctx.arithmetic(0)), visit(ctx.arithmetic(1)));
158   }
159
160   // multiply or division
161   @Override
162   public Logic visitMulDiv(MulDivContext ctx) {
163     if (ctx.op.getType() == LogicParser.MUL) {
164       return new Multiplication(visit(ctx.arithmetic(0)), visit(ctx.arithmetic(1)));
165     }
166     else {
167       return new Division(visit(ctx.arithmetic(0)), visit(ctx.arithmetic(1)));
168     }
169   }
170
171   // add or subtract
172   @Override
173   public Logic visitAddSub(AddSubContext ctx) {
174     if (ctx.op.getType() == LogicParser.ADD) {
175       return new Addition(visit(ctx.arithmetic(0)), visit(ctx.arithmetic(1)));
```

```
176     }
177     else {
178       return new Subtraction(visit(ctx.arithmetic(0)), visit(ctx.arithmetic(1)));
179     }
180   }
181
182   // arithmetic parentheses
183   @Override
184   public Logic visitArithParen(ArithParenContext ctx) {
185     return visit(ctx.arithmetic());
186   }
187 }
```

3.2 Important Methods in LVisitor Class

Among these methods, the important ones are:

Methods	**Return**
visitSingleBool(SingleBoolContext ctx)	returns a **BoolVar** Object in mode **0**
visitSingleInt(SingleIntContext ctx)	returns a **IntVar** Object in mode **0**
visitBoolValueDecl(BoolValueDeclContext ctx)	returns a **BoolVar** Object in mode **2**
visitIntValueDecl(IntValueDeclContext ctx)	returns a **IntVar** Object in mode **2**
visitBoolVar(BoolVarContext ctx)	returns a **BoolVar** Object in mode **1**
visitIntVar(IntVarContext ctx)	returns a **IntVar** Object in mode **1**

3.3 Modes

And for the mode number:

Mode	**Meaning**
mode 0	Uninitialized variable declaration (e.g. boolean p)
mode 1	Variable verification (e.g. not p)
mode 2	Initialized variable declaration (e.g. boolean q = true)

Then based on the different mode of the creation of my variables, my visitors could have different actions based on these mode.

4 TypeChecker

4.1 Code

Below is the code of my TypeChecker Visitor Class:

```
1 package version.logic.visitor;
2
3 import org.antlr.v4.runtime.*;
```

```
4  import org.antlr.v4.runtime.misc.Pair;
5  import org.antlr.v4.runtime.tree.*;
6  import org.stringtemplate.v4.compiler.CodeGenerator.list_return;
7
8  import java.io.*;
9  import java.util.*;
10
11 import com.microsoft.z3.*;
12
13 import antlr.*;
14 import types.*;
15 import values.*;
16 import version.logic.composite.*;
17
18 public class TypeChecker implements Visitor{
19   // hashmap for type checking
20   public static Map<String, Pair<VarType, Logic>> varMap = new HashMap<String, Pair<
         VarType, Logic>>();
21
22   // error message
23   public List<String> errormsg;
24
25
26   // constructor
27   public TypeChecker() {
28     errormsg = new ArrayList<String>();
29   }
30
31   // helper method for checking binary expression
32   public void BinaryChecker(BinaryExpr b) {
33     TypeChecker checker1 = new TypeChecker();
34     TypeChecker checker2 = new TypeChecker();
35
36     b.left().accept(checker1);
37     b.right().accept(checker2);
38
39     errormsg.addAll(checker1.errormsg);
40     errormsg.addAll(checker2.errormsg);
41   }
42
43   // helper method for checking unary expression
44   public void UnaryChecker(UnaryExpr u) {
45
46     TypeChecker checker = new TypeChecker();
47
48     u.child.accept(checker);
49
50     errormsg.addAll(checker.errormsg);
51   }
52
53   // not
54   @Override
55   public void visitNot(Negation e) {
56     UnaryChecker(e);
57   }
58
59   // and
60   @Override
61   public void visitAnd(Conjunction e) {
62     BinaryChecker(e);
```

```
63    }
64
65    // or
66    @Override
67    public void visitOr(Disjunction e) {
68      BinaryChecker(e);
69    }
70
71    // impies
72    @Override
73    public void visitImplies(Implication e) {
74      BinaryChecker(e);
75    }
76
77    // if and only if
78    @Override
79    public void visitIff(Iff e) {
80      BinaryChecker(e);
81    }
82
83    // arithmetic equal (=)
84    @Override
85    public void visitEqual(Equal e) {
86      BinaryChecker(e);
87    }
88
89    // arithmetic greater than (>)
90    @Override
91    public void visitGreaterThan(GreaterThan e) {
92      BinaryChecker(e);
93    }
94
95    // arithmetic less than (<)
96    @Override
97    public void visitLessThan(LessThan e) {
98      BinaryChecker(e);
99    }
100
101   // arithmetic greater than or equal (>=)
102   @Override
103   public void visitGreaterOrEqual(GreaterOrEqual e) {
104     BinaryChecker(e);
105   }
106
107   // arithmetic less or equal (<=)
108   @Override
109   public void visitLessOrEqual(LessOrEqual e) {
110     BinaryChecker(e);
111   }
112
113   // arithmetic add (+)
114   @Override
115   public void visitAddition(Addition e) {
116     BinaryChecker(e);
117   }
118
119   // arithmetic subtract (-)
120   @Override
121   public void visitSubtraction(Subtraction e) {
122     BinaryChecker(e);
```

```
123   }
124
125   // arithmetic multiply (*)
126   @Override
127   public void visitMultiplication(Multiplication e) {
128     BinaryChecker(e);
129   }
130
131   // arithmetic divide (/)
132   @Override
133   public void visitDivision(Division e) {
134     BinaryChecker(e);
135   }
136
137   // boolean variable
138   @Override
139   public void visitBoolVar(BoolVar v) {
140     // mode 0: uninitialized declaration
141     // e.g. boolean q
142     if (v.mode == 0) {
143       // if this variable is declared for the first time, simply add it to the map
144       if (!varMap.containsKey(v.name)) {
145         varMap.put(v.name, new Pair<VarType, Logic>(new BoolType(), null));
146       }
147       // if this variable is not declared for the first time, change its type to
              unknown type
148       // and add the error message
149       else {
150         varMap.replace(v.name, new Pair<VarType, Logic>(new UnknowType(), null));
151         errormsg.add("Error: Type declaration of variable " + v.name + " is ambigous
              . "
152             + "Please make sure each variable is declared exactly once.");
153       }
154     }
155     // mode 1: verification
156     // e.g. p => q
157     else if (v.mode == 1) {
158       if (!varMap.containsKey(v.name)) {
159         errormsg.add("Error: Variable " + v.name + " has not been declared.");
160       }
161       // if it has unknown type
162       else if (varMap.containsKey(v.name) && (varMap.get(v.name).a instanceof types.
             UnknowType)) {
163         errormsg.add("Error: Type of variable " + v.name + " in this expression is
              ambigous. "
164       + "Please make sure each variable is declared exactly once.");
165       }
166       // if it's not boolean type
167       else if (varMap.containsKey(v.name) && !(varMap.get(v.name).a instanceof types
             .BoolType)) {
168         errormsg.add("Error: Variable " + v.name + " is not declared as boolean type
              .");
169       }
170     }
171     // mode 2: initialized declaration
172     // e.g. boolean p = not q
173     else if (v.mode == 2) {
174       // type check this boolean variable's value first
175       TypeChecker checker = new TypeChecker();
176       v.value.accept(checker);
```

```
177
178       // if there is no error, add it to the map
179       if (checker.errormsg.isEmpty()) {
180         if (!varMap.containsKey(v.name)) {
181           varMap.put(v.name, new Pair<VarType, Logic>(new BoolType(), v.value));
182         }
183         else {
184           varMap.replace(v.name, new Pair<VarType, Logic>(new UnknowType(), null));
185           errormsg.add("Error: Type declaration of variable " + v.name + " is
                ambigous. "
186               + "Please make sure each variable is declared exactly once.");
187         }
188       }else {
189         errormsg.addAll(checker.errormsg);
190       }
191     }
192   }
193
194   // int variable
195   @Override
196   public void visitIntVar(IntVar v) {
197     // mode 0: uninitialized declaration
198     // e.g. int i
199     if (v.mode == 0) {
200       // if this variable is declared for the first time, simply add it to the map
201       if (!varMap.containsKey(v.name)) {
202         varMap.put(v.name, new Pair<VarType, Logic>(new IntType(), null));
203       }
204       // if this variable is not declared for the first time, change its type to
             unknown type
205       // and add the error message
206       else {
207         varMap.replace(v.name, new Pair<VarType, Logic>(new UnknowType(), null));
208         errormsg.add("Error: Type declaration of variable " + v.name + " is ambigous
              . "
209             + "Please make sure each variable is declared exactly once.");
210       }
211     }
212     // mode 1: verification
213     else if (v.mode == 1) {
214       if (!varMap.containsKey(v.name)) {
215         errormsg.add("Error: Variable " + v.name + " has not been declared.");
216       }
217       // if it has unknown type
218       else if (varMap.containsKey(v.name) && (varMap.get(v.name).a instanceof types.
             UnknowType)) {
219         errormsg.add("Error: Type of variable " + v.name + " in this expression is
              ambigous. "
220             + "Please make sure each variable is declared exactly once.");
221       }
222       // if it's not declared as int type
223       else if (varMap.containsKey(v.name) && !(varMap.get(v.name).a instanceof types
             .IntType)) {
224         errormsg.add("Error: Variable " + v.name + " is not declared as integer type
              .");
225       }
226     }
227     // mode 2: initialized declaration
228     // int j = 1 + 2
229     else if (v.mode == 2) {
```

```
230       // type check this arithmetic variable's value first
231       TypeChecker checker = new TypeChecker();
232       v.value.accept(checker);
233
234       // if there is no error, check the map first
235       if (checker.errormsg.isEmpty()) {
236         if (!varMap.containsKey(v.name)) {
237           varMap.put(v.name, new Pair<VarType, Logic>(new IntType(), v.value));
238         }
239         else {
240           varMap.replace(v.name, new Pair<VarType, Logic>(new UnknowType(), null));
241           errormsg.add("Error: Type declaration of variable " + v.name + " is
                ambigous. "
242               + "Please make sure each variable is declared exactly once.");
243         }
244       }else {
245         errormsg.addAll(checker.errormsg);
246       }
247     }
248   }
249
250   // boolean true
251   @Override
252   public void visitBoolTrue(BoolTrue c) {
253     // automatically type correct
254   }
255
256   // boolean false
257   @Override
258   public void visitBoolFalse(BoolFalse c) {
259     // automatically type correct
260   }
261
262   // number constant
263   @Override
264   public void visitNumConst(NumConst l) {
265     // automatically type correct
266   }
267 }
```

Among all these methods, the important ones are:

Methods	**Return**
visitBoolVar(BoolVar v)	Do the type checking on boolean variable based on different mode
visitIntVar(IntVar v)	Do the type checking on int variable based on different mode
visitBoolTrue(BoolTrue c)	For boolean constant **true**, it's automatically type correct
visitBoolFalse(BoolFalse c)	For boolean constant **false**, it's automatically type correct
visitNumConst(NumConst l)	For number constant (e.g. 2, 0, etc.), it's automatically type correct

These are the base case for type checking, other methods will recursively do the type checking use the TypeChecker Visitor.

4.2 Variable Type Checking

For **visitBoolVar(BoolVar v)** and **visitIntVar(IntVar v)**, the base logic is the same:

- **For mode 0**: Uninitialized variable declaration (e.g. boolean p)
  - **Case 1**: the Hashmap does not contain the variable, which means that this variable is declared for the first time, then just simply add it to the Hashmap.
  - **Case 2**: the Hashmap already contains the variable, which means the user has declared the variable more than once, then change the type of this variable to unknow type, and add the error message.
- **For mode 1**: Variable verification (e.g. p => q)
  - **Case 1**: the Hashmap does not contain the variable, which means the user wants to verify the truth value of this variable without declaring it, then add the error message to indicates that this variable has not been declared.
  - **Case 2**: the Hashmap contains the variable, but the type of this variable is unknown type, which means the user has declared this variable more than once, then add the error message to indicate that the type of this variable is ambigous.
  - **Case 3**: the Hashmap contains the variable, but the type of this variable is not correct (e.g. want to verify the truth value of a int variable, or want to do arithmetic operation on a boolean variable), then add the error message to indicate that the type of this variable is not correct.
- **For mode 2**: Initialized variable declaration (e.g. boolean q = true)
  - First type check the value assigned to this variable, if there is no type error on the assigned value:
    - **Case 1**: the Hashmap does not contain the variable, which means that this variable is declared for the first time, then just simply add both the variable and its value to the Hashmap.
    - **Case 2**: the Hashmap already contains the variable, which means the user has declared the variable more than once, then change the type of this variable to unknow type, and add the error message.
  - If after the type check, there is type error on the assigned value, simply append all the error message.

5 PrettyPrinter

Use the argument "-p" to use PrettyPrinter

For my PrettyPrinter class, I will transform the user input into a string version that can be recognized by the z3 online tool. Below is an example:

User Input:

```
1 -- test propositional logic and equal
2 -- with initialized boolean and int variable declaration with expression value(no
     error)
3 int i = 5 * 6 - 9
4 int j = 1 + 9
```

```
int m = 35 - 12 * 2
boolean p = true
boolean q = not p
int n = i + m * j - 25
verify p => q and (i >= j <=> not m = 0) or n <= i + j or (not q and not j > n - m)
boolean r = not p => q and p
boolean s = r and p <=> q and p
verify not r and (s or q) <=> (p or m = i * n) and not i >= m => n > m * i + j
boolean s1 = i >= j - 3
boolean r0 = m = i * (j + 5)
boolean k = i = m - 7
verify not r0 <=> (s1 => (not k or r)) and i = j - m or k and not m >= n
boolean p0 = i + j * m < n and not q or n >= i + j => m = 11
verify  i >= m - j and not p => q and s1 or k <=> n > m * i - 2 or not m = j - i * 9
```

After calling my PrettyPrinter, four files will be generated since there are four formulas to be verified. At the end of the file, I also added the comment to indicate the user that if the result is "sat", then they could use **(get-model)** to get the result value for the variables.

First file:

```
(declare-const i Int)
(assert (= i (- (* 5 6) 9)))
(declare-const j Int)
(assert (= j (+ 1 9)))
(declare-const m Int)
(assert (= m (- 35 (* 12 2))))
(declare-const p Bool)
(assert (= p true))
(declare-const q Bool)
(assert (= q (not p)))
(declare-const n Int)
(assert (= n (- (+ i (* m j)) 25)))
(assert (not (=> p (or (or (and q (= (>= i j) (not (= m 0)))) (<= n (+ i j))) (and (
   not q) (not (> j (- n m))))))))
(check-sat)
;Remove the comment if the result of z3 online tool returns "sat"
;(get-model)
```

Second file:

```
(declare-const i Int)
(assert (= i (- (* 5 6) 9)))
(declare-const j Int)
(assert (= j (+ 1 9)))
(declare-const m Int)
(assert (= m (- 35 (* 12 2))))
(declare-const p Bool)
(assert (= p true))
(declare-const q Bool)
(assert (= q (not p)))
(declare-const n Int)
(assert (= n (- (+ i (* m j)) 25)))
(declare-const r Bool)
(assert (= r (=> (not p) (and q p))))
(declare-const s Bool)
(assert (= s (= (and r p) (and q p))))
(assert (not (= (and (not r) (or s q)) (=> (and (or p (= m (* i n))) (not (>= i m)))
    (> n (+ (* m i) j))))))
```

```
(check-sat)
;Remove the comment if the result of z3 online tool returns "sat"
;(get-model)
```

Third file:

```
(declare-const i Int)
(assert (= i (- (* 5 6) 9)))
(declare-const j Int)
(assert (= j (+ 1 9)))
(declare-const m Int)
(assert (= m (- 35 (* 12 2))))
(declare-const p Bool)
(assert (= p true))
(declare-const q Bool)
(assert (= q (not p)))
(declare-const n Int)
(assert (= n (- (+ i (* m j)) 25)))
(declare-const r Bool)
(assert (= r (=> (not p) (and q p))))
(declare-const s Bool)
(assert (= s (= (and r p) (and q p))))
(declare-const s1 Bool)
(assert (= s1 (>= i (- j 3))))
(declare-const r0 Bool)
(assert (= r0 (= m (* i (+ j 5)))))
(declare-const k Bool)
(assert (= k (= i (- m 7))))
(assert (not (= (not r0) (or (and (=> s1 (or (not k) r)) (= i (- j m))) (and k (not
   (>= m n)))))))
(check-sat)
;Remove the comment if the result of z3 online tool returns "sat"
;(get-model)
```

Fourth file:

```
(declare-const i Int)
(assert (= i (- (* 5 6) 9)))
(declare-const j Int)
(assert (= j (+ 1 9)))
(declare-const m Int)
(assert (= m (- 35 (* 12 2))))
(declare-const p Bool)
(assert (= p true))
(declare-const q Bool)
(assert (= q (not p)))
(declare-const n Int)
(assert (= n (- (+ i (* m j)) 25)))
(declare-const r Bool)
(assert (= r (=> (not p) (and q p))))
(declare-const s Bool)
(assert (= s (= (and r p) (and q p))))
(declare-const s1 Bool)
(assert (= s1 (>= i (- j 3))))
(declare-const r0 Bool)
(assert (= r0 (= m (* i (+ j 5)))))
(declare-const k Bool)
(assert (= k (= i (- m 7))))
(declare-const p0 Bool)
```

```
(assert (= p0 (=> (or (and (< (+ i (* j m)) n) (not q)) (>= n (+ i j))) (= m 11))))
(assert (not (= (=> (and (>= i (- m j)) (not p)) (or (and q s1) k)) (or (> n (- (* m
    i) 2)) (not (= m (- j (* i 9))))))))
(check-sat)
;Remove the comment if the result of z3 online tool returns "sat"
;(get-model)
```

6 Verifier

Use the argument "-v" to use Verifier For the same example from previous section, after calling my verifier, there are also four files been generated. **First file**:

```
This formula is tautology.
```

Second file:

```
This formula is not tautology.
(define-fun p () Bool
  true)
(define-fun r () Bool
  true)
(define-fun s () Bool
  false)
(define-fun n () Int
  106)
(define-fun q () Bool
  false)
(define-fun m () Int
  11)
(define-fun j () Int
  10)
(define-fun i () Int
  21)
```

Third file:

```
This formula is not tautology.
(define-fun p () Bool
  true)
(define-fun r () Bool
  true)
(define-fun s1 () Bool
  true)
(define-fun k () Bool
  false)
(define-fun r0 () Bool
  false)
(define-fun s () Bool
  false)
(define-fun n () Int
  106)
(define-fun q () Bool
  false)
(define-fun m () Int
  11)
(define-fun j () Int
  10)
```

```
(define-fun i () Int
  21)
```

Fourth file:

```
This formula is tautology.
```

I have not yet formatted the output value for the variables returned from z3 because I'm not sure what's the best way to show the witness, I will discuss this with my supervisor, then I'll split the output string and formatted it in the prettier way.