Sep 6th Report

By Xiang Chen (Echo)

Sep 6, 2019

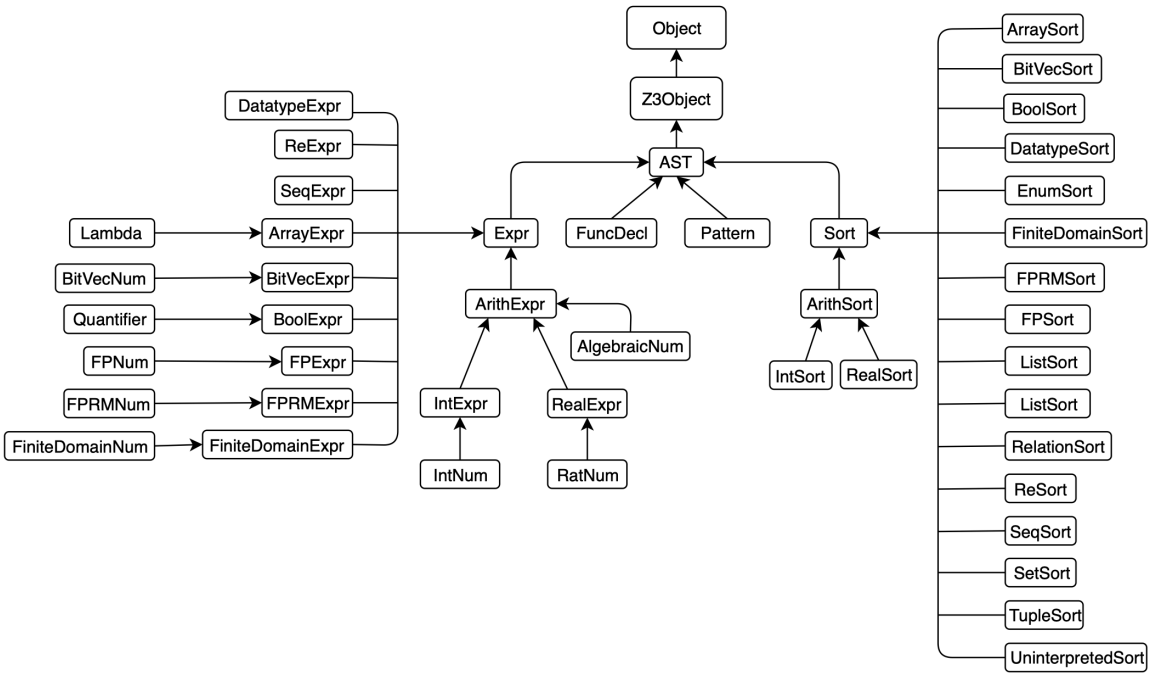
Contents

Part I

Finished Report Before Meeting on Sep 3

1 Z3 Class Hierarchy

I read the z3 library source code and I have drawn out the basic class hierarchy without any details:



Among these classes, I think the most important classes is **Expr** class and its descendant classes, including **ArithExpr** which handles Arithmetic expressions and operations, **BoolExpr** which handles Proposition logic and Predicate logic expressions and operations.

Therefore I use **Expr** as the return type for my original program that uses the Antlr built-in visitor pattern.

---

2 Antlr Grammar and Generated Class Hierarchy

2.1 Antlr Grammar file

Below is my original grammar:

```
grammar Logic;


stat : line+ ;

line : boolExpr NEWLINE;

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


NOT : 'not';
AND : 'and';
OR : 'or';
IMPLIES : '=>';
IFF : '<=>';

VAR : [a-z][a-zA-Z0-9]*;
NEWLINE : '\r'? '\n' ;
COMMENT : '--' .*? '\n' -> skip;
WS  :   [ \t]+ -> skip ;
```

I have modified it so that it accepts multiple lines of input, each line contains a boolean expression.

I also add the line that will skip the COMMENT:

```
COMMENT : '--' .*? '\n' -> skip;
```

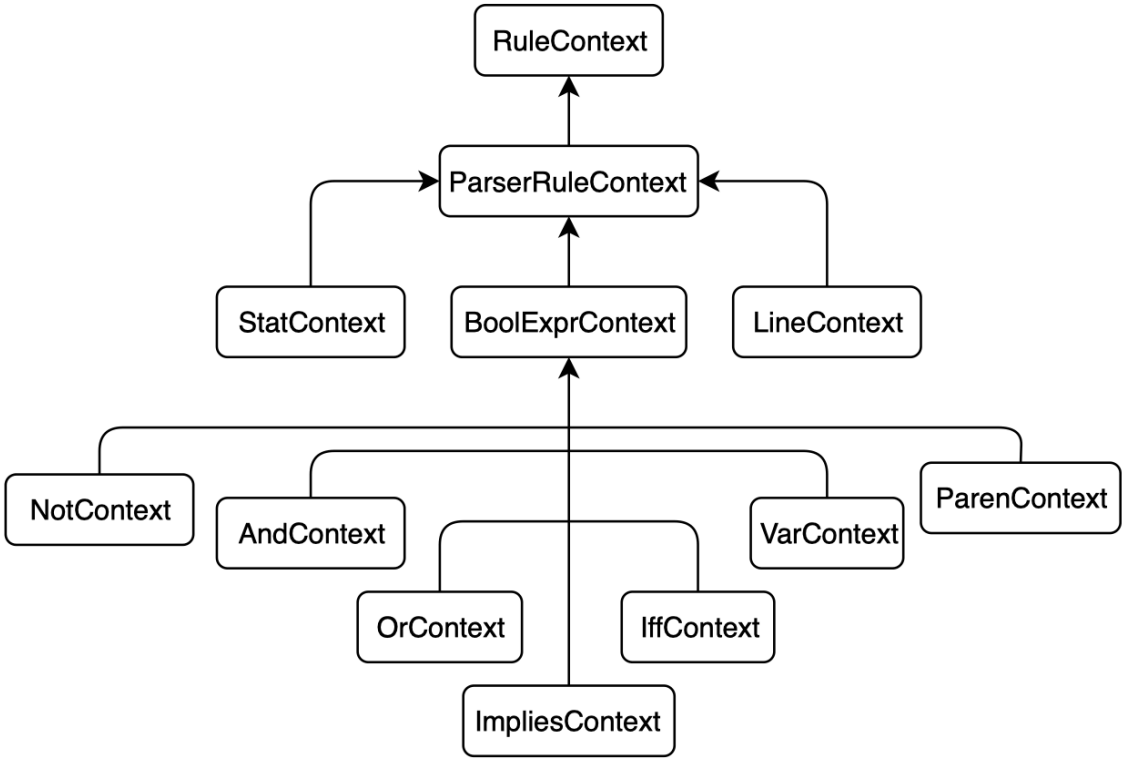
It means that anything between '–' and a newline symbol will be ignored, which will skip a one-line comment.

Base on this grammar, after you compile this grammar, Antlr will automatically generate the specific classes according to the my grammar name, the rules and labels, and they are all so-called Contexts.

Also, because I use the option **- visitor** when I compile my grammar, Antlr will generate the visitor interface for me as well.

2.2 Antlr Generated Classes Based on My Grammar

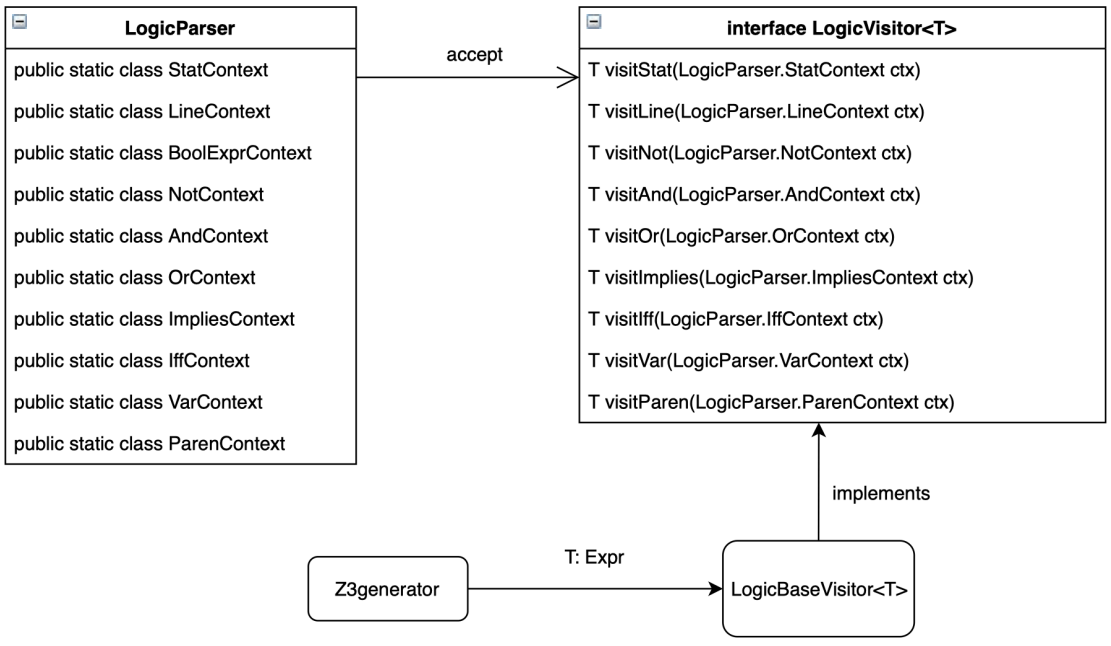
Below is a diagram that shows the class hierarchy that Antlr generates based on my grammar:



The inheritance structure is very clear here.

2.3 Antlr Visitor Pattern

Below is a diagram that shows the visitor pattern that generated by Antlr base on my grammar, and **Z3generator** is the class that I use this pattern, and let the return type to be **Expr**. It could solve multiple lines of propositional logic formula now.

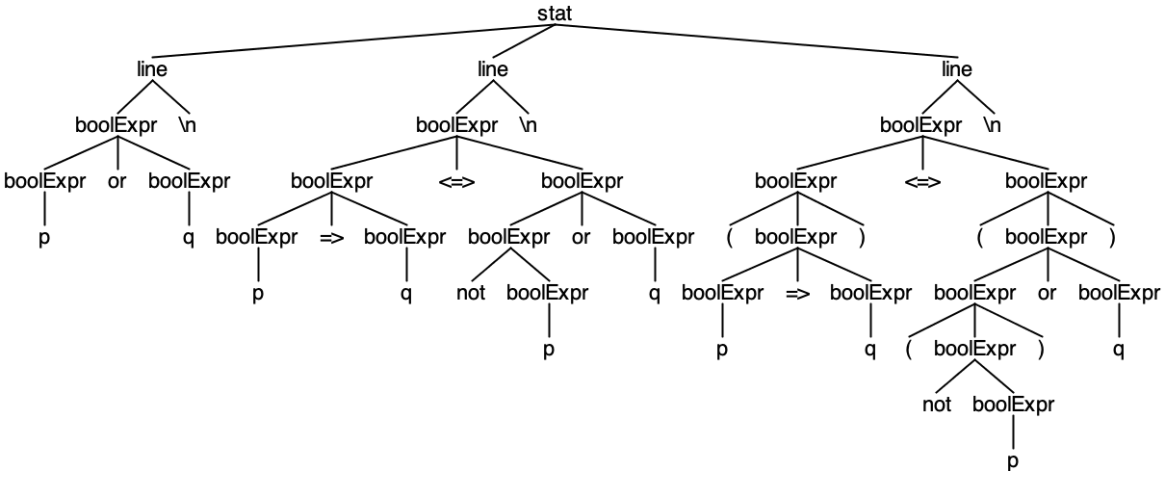


3 Running Examples

Below is a sample txt file that contains several lines of propositional logic expressions:

```
p or q
-- comment
p => q <=> not p or q
(p => q) <=> ((not p) or q)
```

Below is the parse tree that Antlr generated base on this sample txt file:



As we could see, the comment line is successfully ignored by Antlr.

Also, there are three subtrees that representing the three input propositional logic expressions.

Below is the result when running my **Z3generator**:

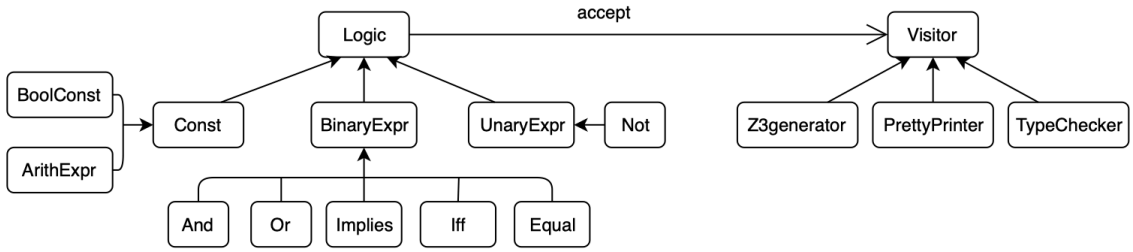
```
The formula is not tautology.

Witness:
p = false
q = false

The formula is tautology.
The formula is tautology.
```

4 My New Visitor Pattern

Below is the ideal visitor pattern for the design, but it doesn't perform as expected, and I'm still struggling with the details.



---

5 Achievement and Problems Left

5.1 Achievement

1. I have transform the to-do.txt to README.md.
2. Antlr parser will successfully ignore the comment now.
3. User can now enter multiple lines of propositional logic expressions.
4. User could choose to optionally add parentheses, it will not affect the test result because Antlr will choose alternatives based on the order of the subrule. The first alternative has the highest precedence.
5. My program will output a witness if the formula is not tautology.

5.2 Problems

1. The customized visitor pattern does not work.
2. Based on the previous example, when visiting the parse tree, it will finally reach the leaf, and when it reach the leaf, the method **visitVar()** will be called. Therefore, my program will definitely create the boolean constant. So user may not need to declare the constant at all.
3. Becaue later on arithmetic expression might be added to the program, then variables could be both boolean or arithmetic variable, I'm stuck at the part that distinguish the type of the variables.

Part II
Extended Report After Meeting on Sep 3

6 Modified Grammar

6.1 Grammar

I modified my grammar again as follows:

```
grammar Logic;


stat : line+ ;

line
  : BOOL VAR NEWLINE      # SingleBool
  | INT VAR NEWLINE       # SingleInt
  | 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
  ;



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

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

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

NOT : 'not';
AND : 'and';
OR : 'or';
IMPLIES : '=>';
IFF : '<=>';

VAR : [a-z][a-zA-Z0-9]*;
NUM : [0-9]+;
NEWLINE : '\r'? '\n' ;
```

I changed the expression of **COMMENT** as follows since the previous version may cause unexpected error.

```
COMMENT : '--' ~[\r\n]* -> skip;
```

The modified version works as expected.

Every line that starts with '- -' will be ignored by the parser.

6.2 Test class

I also make my program interactive (like using '-i' in Eiffel).

By default, **Antlr** could only generate the parse tree and traverse it after the user quit the program (e.g. press ctrl + D in Mac or ctrl + Z in Windows). **Antlr** will consume the whole input all together and output the result once the user quit the program.

However, this makes it impossible to get the output immediately. I enhance the test class so that when the user hit Return, my program will immediately responds with an specific output if it's possible.

Below is the code for the Test class:

```
1  public class TestExpr {
2    public static void main(String[] args) {
3      try {
4        // test to see if there is a input file
5        String inputFile = null;
6        if ( args.length>0 ) inputFile = args[0];
7        InputStream is = System.in;
8        if ( inputFile!=null ) {
9        is = new FileInputStream(inputFile);
10       }
11
12       BufferedReader br = new BufferedReader(new InputStreamReader(is));
13
14       // get first expression
15       String expr = br.readLine();
16       // track input expr line numbers
17       int line = 1;
18
19       Z3generator z3 = new Z3generator();
20
21       while ( expr!=null ) {  // while we have more expressions
22         // create new lexer and token stream for each line (expression)
23         ANTLRInputStream input = new ANTLRInputStream(expr + "\n");
24         LogicLexer lexer = new LogicLexer(input);
25
26         // notify lexer of input position
27         lexer.setLine(line);
28         lexer.setCharPositionInLine(0);
29
30         CommonTokenStream tokens = new CommonTokenStream(lexer);
31         LogicParser parser = new LogicParser(tokens);
32
33         // tell ANTLR to build a parse tree for current line
34         parser.setBuildParseTree(true);
35         // parse
36         ParseTree tree = parser.stat();
37
38         // visit the generated tree
39         z3.visit(tree);
```

```
40
41          // see if there's another line
42          expr = br.readLine();
43          line++;
44          }
45        }catch(Exception e){
46          System.out.println("z3 exception");
47          e.printStackTrace();
48          }
49        }
50      }
```

6.3 Z3generator Class

Below is the Z3generator class that inherits the **Visitor** class:

```
1  public class Z3generator extends LogicBaseVisitor<Expr>{
2
3    public Context context = new Context();
4
5    // map that stores variable-type pair
6    Map<String, String> varType = new HashMap<String, String>();
7
8    // map that stores variable-value pair
9    Map<String, String> varValue = new HashMap<String, String>();
10
11   // single boolean declaration
12   public Expr visitSingleBool(LogicParser.SingleBoolContext ctx) {
13     // if the variable has not been declared before, add it to the map
14     if (!varType.containsKey(ctx.VAR().getText())) {
15       varType.put(ctx.VAR().getText(), "boolean");
16     // else return the error msg
17     }else {
18       System.out.println("Variable " +ctx.VAR().getText() + " is already
             declared as "
19           + varType.get(ctx.VAR().getText())
20           + ", you cannot declare it twice.");
21     }
22     return null;
23   }
24
25   // single int declaration
26   public Expr visitSingleInt(LogicParser.SingleIntContext ctx) {
27     // if the variable has not been declared before, add it to the map
28     if (!varType.containsKey(ctx.VAR().getText())) {
29       varType.put(ctx.VAR().getText(), "int");
30     // else return the error msg
31     }else {
32       System.out.println("Variable " +ctx.VAR().getText() + " is already
             declared as "
33           + varType.get(ctx.VAR().getText())
34           + ", you cannot declare it twice.");
35     }
36     return null;
```

```
37    }
38
39    // evaluate the boolean expression
40    public Expr visitEvalBoolExpr(LogicParser.EvalBoolExprContext ctx) {
41      // create a new solver for the formula each time
42      Solver solver = context.mkSolver();
43
44
45      if (visit(ctx.boolExpr()) != null) {
46        BoolExpr expr = (BoolExpr) visit(ctx.boolExpr());
47        // negate the output expr
48        BoolExpr formula = context.mkNot(expr);
49
50        // add the formula
51        solver.add(formula);
52
53        // check to see of the formula is tautology
54        Status result = solver.check();
55
56        // show the checked result:
57            if (result == Status.SATISFIABLE){
58              System.out.println("The formula is not tautology.");
59
60
61              // get the model
62              Model m = solver.getModel();
63
64              System.out.println(m.toString());
65               }
66            else if(result == Status.UNSATISFIABLE)
67                System.out.println("The formula is tautology.");
68            else
69                System.out.println("Unknow formula");
70        }
71        //System.out.println(expr.toString());
72        return null;
73      }
74
75
76    // define the atom
77    public Expr visitBoolVar(LogicParser.BoolVarContext ctx) {
78      // check is the variable has been declared
79      if (varType.containsKey(ctx.getText())) {
80        // check if the variable is the right type
81        if (varType.get(ctx.getText()) == "boolean") {
82          return context.mkBoolConst(ctx.getText());
83        // if the variable is not the right type, set up the error msg
84        }else {
85          System.out.println("Variable " + ctx.getText() + " is declared as
                " + varType.get(ctx.getText()) + ".");
86          return null;
87        }
88      // if the variable is not declared, set up the error msg
89      }else {
90        System.out.println("Variable " + ctx.getText() + " is not declared.
             ");
91        return null;
92      }
93    }
94
```

```
95
96    // set up the sub-formula of NOT
97    public Expr visitNot(LogicParser.NotContext ctx) {
98      if (visit(ctx.boolExpr()) != null) {
99        return context.mkNot((BoolExpr) visit(ctx.boolExpr()));
100     }
101     return null;
102   }
103
104   // set up the sub-formula for OR
105   public Expr visitOr(LogicParser.OrContext ctx) {
106     if ((visit(ctx.boolExpr(0)) != null) && (visit(ctx.boolExpr(1))) !=
          null) {
107       return context.mkOr((BoolExpr) visit(ctx.boolExpr(0)), (BoolExpr)
            visit(ctx.boolExpr(1)));
108     }
109     return null;
110   }
111
112
113   // set up the sub-formula for IMPLIES
114   public Expr visitImplies(LogicParser.ImpliesContext ctx) {
115     if ((visit(ctx.boolExpr(0)) != null) && (visit(ctx.boolExpr(1))) !=
          null) {
116       return context.mkImplies((BoolExpr) visit(ctx.boolExpr(0)), (
            BoolExpr)visit(ctx.boolExpr(1)));
117     }
118     return null;
119   }
120
121
122   // set up the sub-formula for IFF
123   public Expr visitIff(LogicParser.IffContext ctx) {
124     if ((visit(ctx.boolExpr(0)) != null) && (visit(ctx.boolExpr(1))) !=
          null) {
125       return context.mkIff((BoolExpr) visit(ctx.boolExpr(0)), (BoolExpr)
            visit(ctx.boolExpr(1)));
126     }
127     return null;
128   }
129
130   // set up the sub-formula for AND
131   public Expr visitAnd(LogicParser.AndContext ctx) {
132     if ((visit(ctx.boolExpr(0)) != null) && (visit(ctx.boolExpr(1))) !=
          null) {
133       return context.mkAnd((BoolExpr) visit(ctx.boolExpr(0)), (BoolExpr)
            visit(ctx.boolExpr(1)));
134     }
135     return null;
136   }
137
138   // Set up the formula with parentheses
139   public Expr visitParen(LogicParser.ParenContext ctx) {
140     return visit(ctx.boolExpr());
141   }
142 }
```

I have not yet properly formatted the output for witness, right now it will just simply

return the default **z3** typed output. I will try to finish it in the following days.

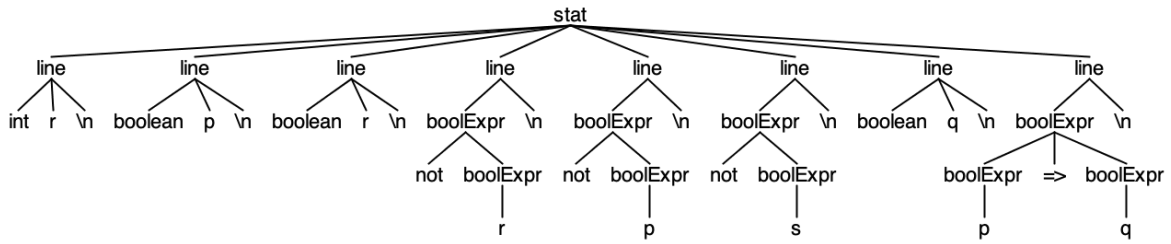
---

7 Running Examples

Below is a sample txt file that contains several lines of propositional logic expressions:

```
int r
-- comment
boolean p
boolean r
not r
not p
not s
boolean q
p => q
```

Below is the parse tree that Antlr generated base on this sample txt file:



Below is the output from my program when use this sample file as input:

```
Variable r is already declared as int, you cannot declare it twice.

Variable r is declared as int.

The formula is not tautology.
(define-fun p () Bool
  true)

Variable s is not declared.

The formula is not tautology.
(define-fun p () Bool
  true)
(define-fun q () Bool
  false)
```

It worked as expected.

---

8 Achievement and Problems Left

8.1 Achievement

1. I have modified the lexer rule for COMMENT so that it worked without any error.
2. I have successfully made my program interactive.
3. I have solved the problem that distinguish the type of the variables by using Symbol Table.

8.2 Problems

1. I still didn't finish the customized Visitor Pattern.
2. For the working **z3** version, I haven't extend it with arithmetic expressions (e.g. +, -, *, /).

I will continue to extend my grammar and my program, and try to finish the z3 version and customized Visitor version by next Monday.