Tool and language:

Antlr with Java on IntelliJ & Python programming language

AIM:

The aim for this assignment is to design and implement a source language and return a correct parse-tree. The base visitor class is extended in a class called 'SecondBaseVisitor' which helps print the out-put. The code has to be able to run on Python and give the intended result.

Implementation

Specification of basic language

GRAMMAR

The WHILE language is a simple imperative language, with assignment to local variables, if statements, while loops, and simple integer and Boolean expressions.

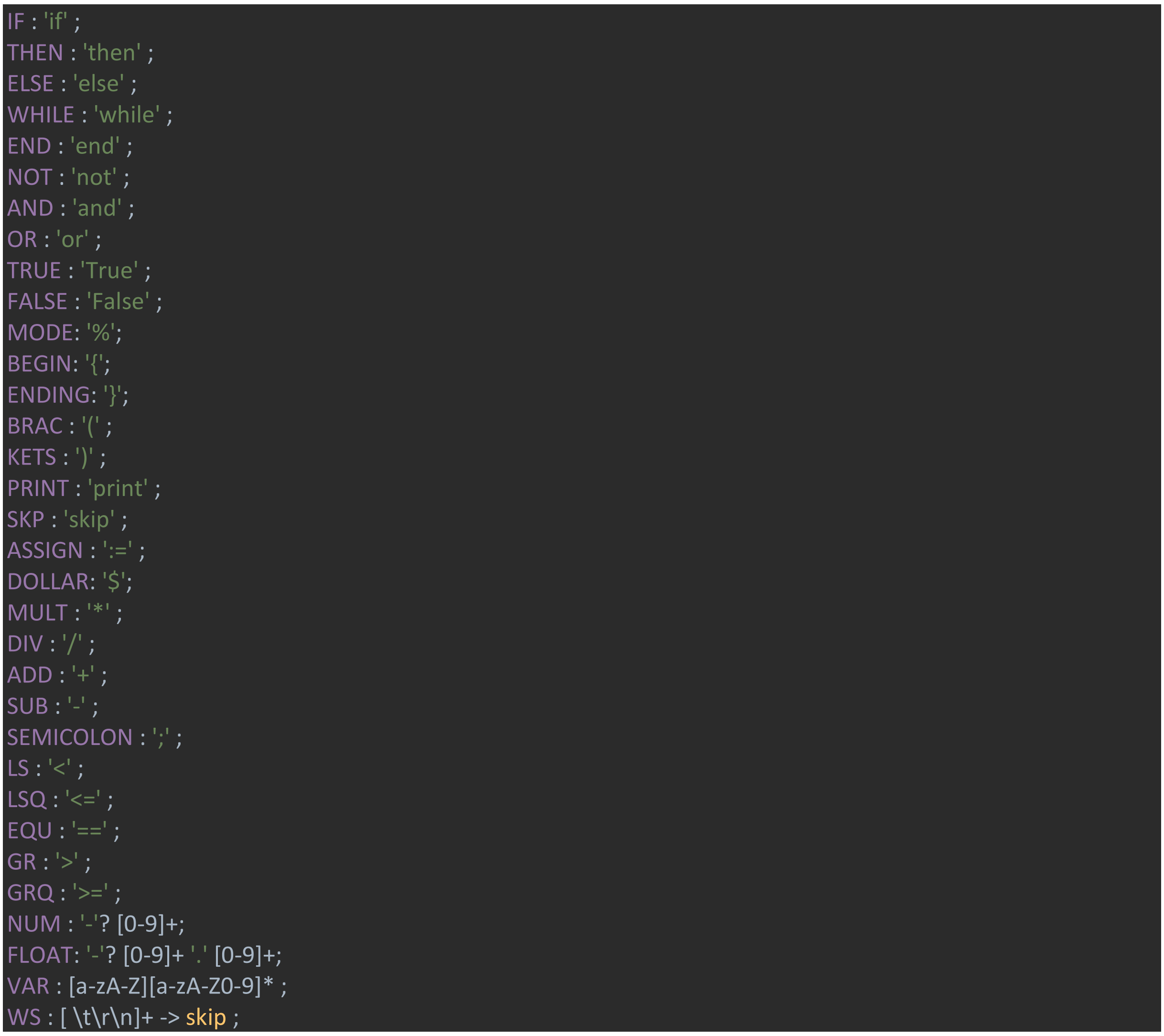
```
grammar assignment2;

start:  stmt ;
//Statmemets
stmt : VAR ASSIGN expr                          #assign
    | SKP                               #skip
    | IF boolExp THEN stmt ELSE stmt               #ifelse
    | WHILE boolExp stmt                       #while
    | PRINT BRAC expr KETS                      #printExp
    | PRINT BRAC boolExp KETS                     #printBool
    | BEGIN stmt+ ENDING                        #CompStmt
    ;

//Expressions
expr  : left = expr op = (MULT | DIV | MODE) right = expr #multiplicative
     | left = expr op = (ADD | SUB) right = expr  #additive
     | figure                         #number
     | BRAC expr KETS                      #brackets
     | VAR                           #variable
     | SUB expr                          #negExp
     ;
//Boolean Expression
boolExp : TRUE                                   #true
       | FALSE                                #false
       | NOT boolExp                              #negBool
       | left=boolExp op=(AND | OR) right=boolExp           #boolOp
       | left=expr op=(LS | LSQ | EQU | GR | GRQ) right=expr    #relational
       ;
//Numbers
figure  : FLOAT            #float
       |  NUM            #integer
       ;
```

Definition of Tokens:

Tokens holds valid values (i.e. strings, characters, integers etc.), they are stepping stones towards creating a coherent structure called **expressions**.



```
IF : 'if' ;
THEN : 'then' ;
ELSE : 'else' ;
WHILE : 'while' ;
END : 'end' ;
NOT : 'not' ;
AND : 'and' ;
OR : 'or' ;
TRUE : 'True' ;
FALSE : 'False' ;
MODE: '%';
BEGIN: '{';
ENDING: '}';
BRAC : '(' ;
KETS : ')' ;
PRINT : 'print' ;
SKP : 'skip' ;
ASSIGN : ':=' ;
DOLLAR: '$';
MULT : '*' ;
DIV : '/' ;
ADD : '+' ;
SUB : '-' ;
SEMICOLON : ';' ;
LS : '<' ;
LSQ : '<=' ;
EQU : '==' ;
GR : '>' ;
GRQ : '>=' ;
NUM : '-'? [0-9]+;
FLOAT: '-'? [0-9]+ '.' [0-9]+;
VAR : [a-zA-Z][a-zA-Z0-9]* ;
WS : [ \t\r\n]+ -> skip ;
```

2. Implementation of basic language

Basic notions:

The code below shows that my program allows variable declarations (e.g. int a)

```
@Override
public String visitVariable(assignment2Parser.VariableContext ctx) {
   return ctx.VAR().getText();
}
```

Another feature is variable assignment statements (e.g. a:=b+1), in other words the Variable = expression.

```
@Override
public String visitAssign(assignment2Parser.AssignContext ctx) {
   return ctx.VAR().getText() + " = " + this.visit(ctx.expr());
}
```

My program accepts expression (e.g. a:=(2+b*3)/(5+d))

```
@Override
public String visitPrintExp(assignment2Parser.PrintExpContext ctx) {
   return "print(" + this.visit(ctx.expr()) + ')';
}
```

Sequencing of statements(e.g. b:=2; a:=b+1) is also a feature. The code below shows this, indents are also available so the python code can run.

```
@Override
 public String visitCompStmt(assignment2Parser.CompStmtContext ctx) {
   String t = "";

   for (assignment2Parser.StmtContext e: ctx.stmt())
      t += empties(indent) + this.visit(e) + "\n";
   return t;
}
```

My BaseVisitor can perform operations including addition, subtraction, division and multiplication.

```
@Override
//addition and subtraction
public String visitAdditive(assignment2Parser.AdditiveContext ctx) {
   String left = visit(ctx.left);
   String op = ctx.op.getText();
   String right = visit(ctx.right);
   if (op.equals("+"))
      return left + " + " + right;
   else
      return left + " - " + right;
}
```

```
//multiplication and division
@Override
public String visitMultiplicative(assignment2Parser.MultiplicativeContext ctx) {
    String left = visit(ctx.left);
    String op = ctx.op.getText();
    String right = visit(ctx.right);
    if (op.equals("*"))
        return left + " * " + right;
    else if (op.equals("/"))
        return left + " / " + right;
    else
        return left + "%" + right;
}
```

My program can perform tasks involving Boolean expressions. The code below shows the necessary implementations.

```
//boolean operation

    @Override
    public String visitBoolOp(assignment2Parser.BoolOpContext ctx) {
        String left = visit(ctx.left);
        String op = ctx.op.getText();
        String right = visit(ctx.right);
        if (op.equals("and"))
            return left + " and " + right;
        else
            return left + " or " + right;
    }

    @Override
    public String visitTrue(assignment2Parser.TrueContext ctx) {
        return"True";
    }

    @Override
    public String visitFalse(assignment2Parser.FalseContext ctx) {
        return "False";
    }

    @Override
    public String visitNegBool(assignment2Parser.NegBoolContext ctx) {
        return ctx.NOT().getText() + this.visit(ctx.boolExp());
    }
//Greater than, Greater or equal too, Less than, Less than or equal too
    @Override
    public String visitRelational(assignment2Parser.RelationalContext ctx) {
        String left = visit(ctx.left);
        String op = ctx.op.getText();
        String right = visit(ctx.right);
        if (op.equals("<"))
            return left + " < " + right;
        else if (op.equals("<="))
            return left + " <= " + right;
    else if (op.equals("=="))
      return left + " == " + right;
    else if (op.equals(">"))
      return left + " > " + right;
    else
      return left + " >= " + right;
  }
```

The Python files

The python code is generated by the code I have created. The files Factorial, Fibonacci and Odd_even have proven to be successful by running no errors and copying all contents of the text files individually to a python interpreter.

Control flow structures:

This program has been tested for Conditional statements (e.g. if e=2 then f: =2, and if-then-else statements). The code begins with 'if' because that is the beginning of the if statement, it is then followed by a Boolean expression. Idents and statements follow until the end of the if-else statement and an answer is returned.

```
@Override
public String visitIfelse(assignment2Parser.IfelseContext ctx) {
  String ans = "if " + this.visit(ctx.boolExp()) + ":\n";
  indent +=  5;
  ans += this.visit(ctx.stmt(0));
  indent -=  5;
  ans += empties(indent);
  indent +=  5;
  ans += this.visit(ctx.stmt(1));
  indent -=  5;
  return ans ;
}
```

Unbounded iteration(e.g. while x<1 do x=x+1)

My program contains a while loop which works alongside Boolean expressions.

```
@Override
public String visitWhile(assignment2Parser.WhileContext ctx) {
  String ans = "while " + this.visit(ctx.boolExp()) + ":\n";
  indent += 5;
  ans += this.visit(ctx.stmt()) + "\n";
  indent -= 5;
  return ans;
}
```

Lastly my program accepts Statement blocks and Control flow statement (e.g. using begin, end, or {...}). This can be seen in my grammar

```
| BEGIN stmt+ ENDING                                        #CompStmt
```

```
@Override
 public String visitCompStmt(assignment2Parser.CompStmtContext ctx) {
   String t = "";

   for (assignment2Parser.StmtContext e: ctx.stmt())
      t += empties(indent) + this.visit(e) + "\n";
   return t;
}
```

Additional Features:

This program declares and use floats and integers,

```
   @Override
   public String visitInteger(assignment2Parser.IntegerContext ctx)  {
      return ctx.NUM().getText();
   }

   @Override public String visitFloat(assignment2Parser.FloatContext ctx) {
      return ctx.FLOAT().getText(); }



}
```

I created a Java class called 'ASTvisitor' which produces an abstract syntax tree that go beyond the built-in features of the development environment

<u>Ouput (In text file)</u>

```
val = 1
counter = 1
while counter <= n:
    val = val * counter
    counter = counter + 1


print(val)
```

<u>Abstract Syntax Tree</u>

```
((CompStmt (assign n 8)
(assign val 1)
(assign counter 1)
(while counter <= n
(CompStmt (assign val val * counter)
(assign counter (Add counter 1))
))
(print val)))
```

My program produces error identification that goes beyond the built-in features of the development environment

```
catch(Exception e){
   System.out.println("....Warning: Some Other exception!");
}
```

Improvements

I had troubles implementing the type checking which involved building the symbol table.