CIS2380-1819 LANGUAGE TRANSLATORS ASSIGNMENT

Question 2

Adam Birch
U1761249

Table of Contents

You must hand in for question 2:

i) The two files parser.cup and scanner.java defining your final language. This must allow me to generate an interpreter using JavaCup and validate your tests.
ii) Several test programs and screen shots of their execution showing the use of ALL the new features in your language and showing the correctness of your code. If you do not include tests with associated screen shots of the new features of MYPL, they will be assumed not to work,
iii) A written report including the following: a statement of WHAT you have achieved in Question 2; and documentation describing HOW you have upgraded TLM for Question 2. Here you must explain in your own words how the code you have added to TLM works, any deficiencies of it, and any future extensions you could achieve given more time

Required files

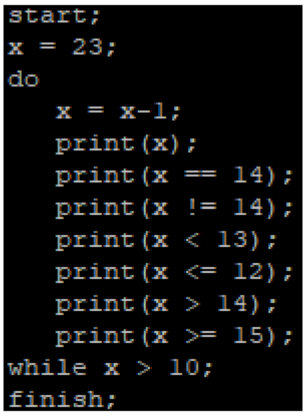
The required files can be found inside the zip directory provided which includes this documentation.

Test Programs and Verification of test results

Boolean Operators:

I have implemented many new Boolean operators, these being ‘<’, ‘<=’, ‘>=’, ‘!=’ and ‘==’.

To test these I wrote a new inputprogram4 file to be used within the parser. The function is as follows:

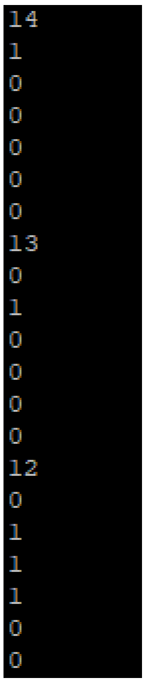


```
start;
x = 23;
do
   x = x-1;
   print(x);
   print(x == 14);
   print(x != 14);
   print(x < 13);
   print(x <= 12);
   print(x > 14);
   print(x >= 15);
while x > 10;
finish;
```

The lines are in pairs. After the initial assign and print of x within the loop x == 14 and x != 14 should always be opposites of each other. X < 13 and X <= 12 will always display the same, as will X > 14 and X >= 15. Here is the output provided (put into a Table):

X	*X == 14*	*X != 14*	*X < 13*	*X <= 12*	*X > 14*	*X >= 15*
22	0	1	0	0	1	1
21	0	1	0	0	1	1
20	0	1	0	0	1	1
19	0	1	0	0	1	1
18	0	1	0	0	1	1
17	0	1	0	0	1	1
16	0	1	0	0	1	1
15	0	1	0	0	1	1
14	1	0	0	0	0	0
13	0	1	0	0	0	0
12	0	1	1	1	0	0
11	0	1	1	1	0	0
10	0	1	1	1	0	0

Here is an example of the output (and why I put it into the table)


```
14
1
0
0
0
0
0
13
0
1
0
0
0
0
12
0
1
1
1
0
0
```

Within this test you can see that column 1 is only true when the value of x is 14 and when it is, column 2 is false. As columns 3 / 4 and 5 / 6 are the same condition but use different syntax for it, both pairs are true and false with the same value as it's partner. As 13 and 14 are not within the bounds of either sets of columns, both are false.

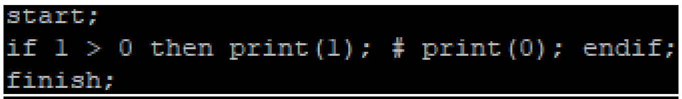
This test also proves that both Assigns and Equals work as intended as x can be assigned a new value and can be compared to a value.

If Else statement.

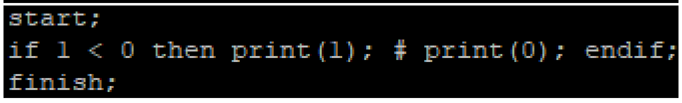
For this I wanted to use standard syntax (If condition then statement; else statement; endif;) However as Else and Endif both start with e, it caused the compiler to get confused between the two. To combat this, I replaced Else with #.

I created two test programs, inputprogram2 and inputprogram3.

These check whether 1 > 0 and 1 < 0 respectively, outputting a 1 if the condition is true, and 0 if it is false. Here is the code I used for these programs;



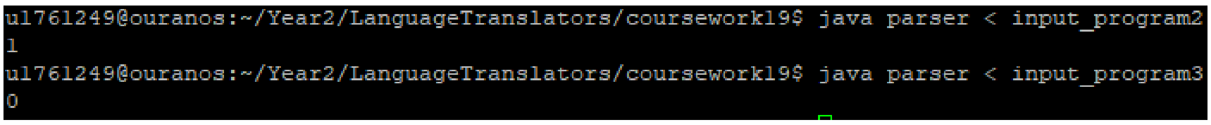
```
start;
if 1 > 0 then print(1); # print(0); endif;
finish;
```



```
start;
if 1 < 0 then print(1); # print(0); endif;
finish;
```

Input 2 and 3 respectively.

After running input 2 then input 3, this is the console output:



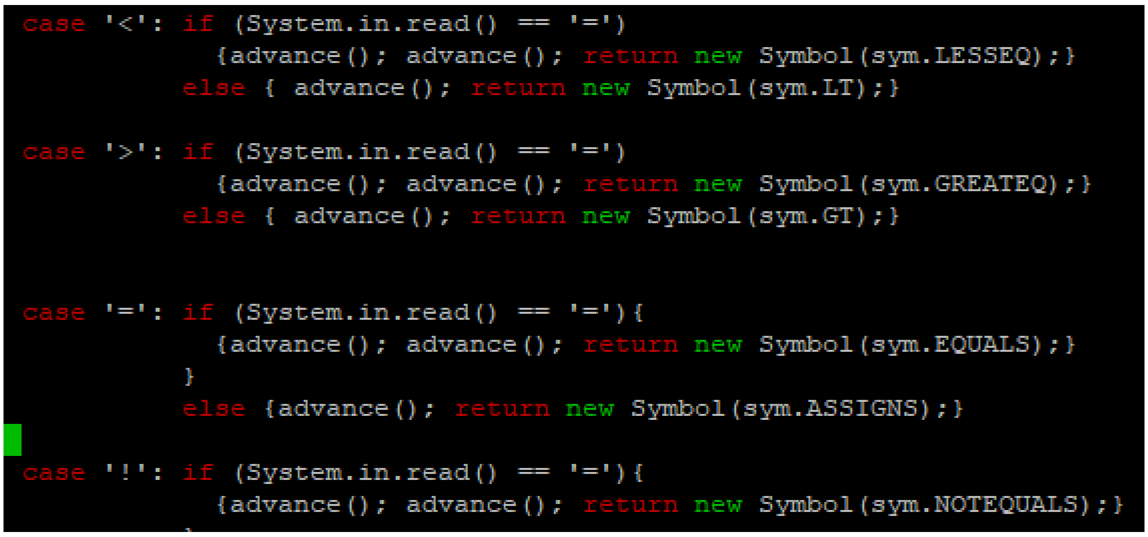
```
u1761249@ouranos:~/Year2/LanguageTranslators/coursework19$ java parser < input_program2
1
u1761249@ouranos:~/Year2/LanguageTranslators/coursework19$ java parser < input_program3
0
```

As the initial condition is true in input 2, the output is 1 – while the opposite is true for input 3.

Final Report

I have been able to achieve several extensions to the TLM to create MYPL. The first of which was to create more Boolean operators to work within the language.

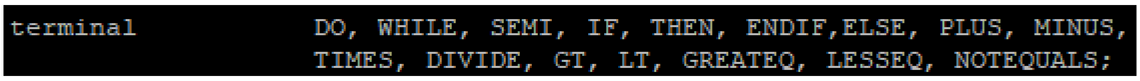
I added the following code to scanner.java to be able to recognize the symbols for the syntax:



```
 case '<': if (System.in.read() == '=')
              {advance(); advance(); return new Symbol(sym.LESSEQ);}
           else { advance(); return new Symbol(sym.LT);}

 case '>': if (System.in.read() == '=')
              {advance(); advance(); return new Symbol(sym.GREATEQ);}
           else { advance(); return new Symbol(sym.GT);}


 case '=': if (System.in.read() == '='){
              {advance(); advance(); return new Symbol(sym.EQUALS);}
           }
           else {advance(); return new Symbol(sym.ASSIGNS);}

 case '!': if (System.in.read() == '='){
              {advance(); advance(); return new Symbol(sym.NOTEQUALS);}
```

In each case, the first symbol may have a trailing symbol that alters the meaning. For this I included a java If statement to read the next character (but not store it) and compare it to an expected value. For example, '<' can be less than or the start of less than or greater than. If the next char was 'x' then it is less than. In the case of '!' I check for a trailing '=' as it is required for the symbol, else a syntax error will be thrown.

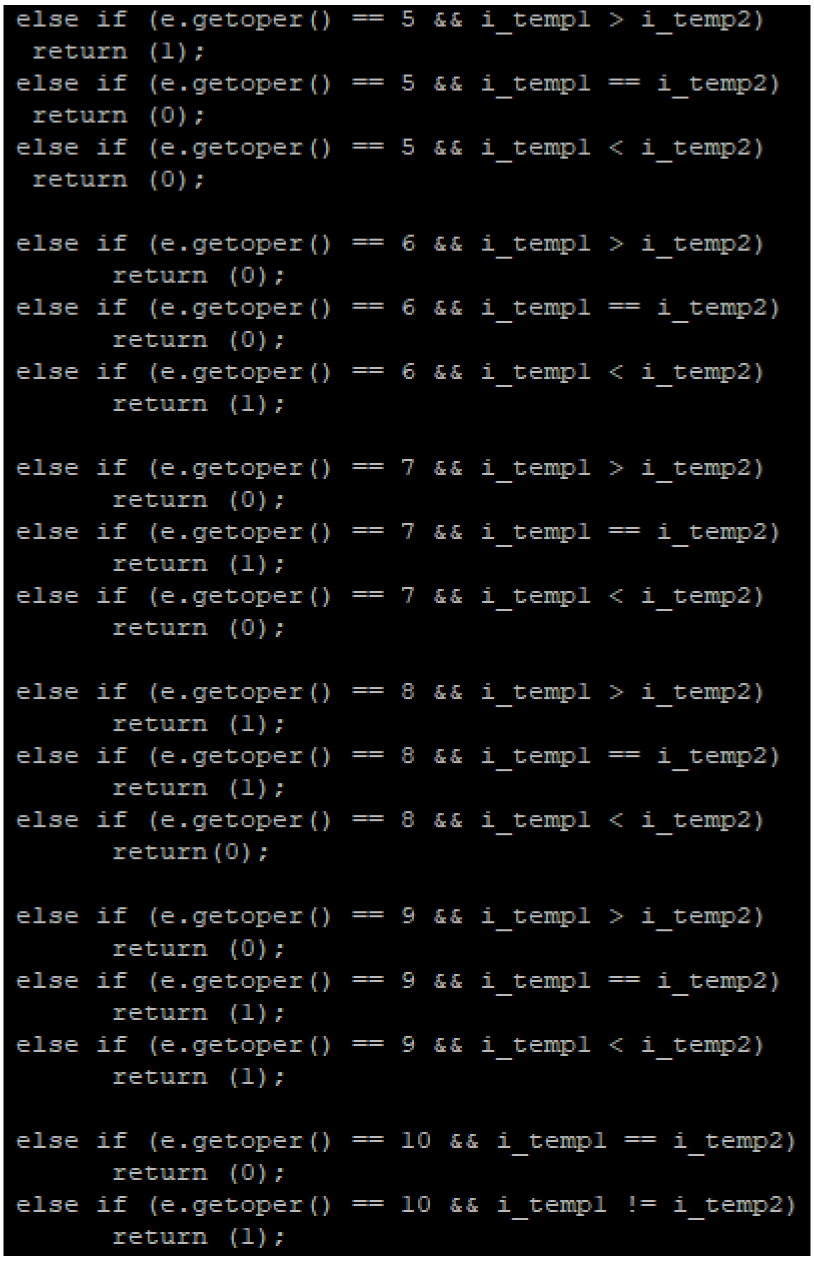
I then needed to add the functions to the parser.cup file. I started by adding each symbol to the list of terminals.



```
terminal            DO, WHILE, SEMI, IF, THEN, ENDIF,ELSE, PLUS, MINUS,
                    TIMES, DIVIDE, GT, LT, GREATEQ, LESSEQ, NOTEQUALS;
```

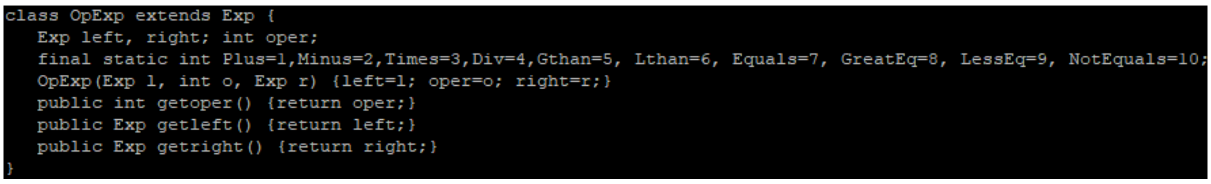
(Split to fit better into an image)

With the new terminal symbols, I added the logic to each symbol so that each would have the correct functionality for what symbol it was.



```
else if (e.getoper() == 5 && i_temp1 > i_temp2)
 return (1);
else if (e.getoper() == 5 && i_temp1 == i_temp2)
 return (0);
else if (e.getoper() == 5 && i_temp1 < i_temp2)
 return (0);

else if (e.getoper() == 6 && i_temp1 > i_temp2)
      return (0);
else if (e.getoper() == 6 && i_temp1 == i_temp2)
      return (0);
else if (e.getoper() == 6 && i_temp1 < i_temp2)
      return (1);

else if (e.getoper() == 7 && i_temp1 > i_temp2)
      return (0);
else if (e.getoper() == 7 && i_temp1 == i_temp2)
      return (1);
else if (e.getoper() == 7 && i_temp1 < i_temp2)
      return (0);

else if (e.getoper() == 8 && i_temp1 > i_temp2)
      return (1);
else if (e.getoper() == 8 && i_temp1 == i_temp2)
      return (1);
else if (e.getoper() == 8 && i_temp1 < i_temp2)
      return(0);

else if (e.getoper() == 9 && i_temp1 > i_temp2)
      return (0);
else if (e.getoper() == 9 && i_temp1 == i_temp2)
      return (1);
else if (e.getoper() == 9 && i_temp1 < i_temp2)
      return (1);

else if (e.getoper() == 10 && i_temp1 == i_temp2)
      return (0);
else if (e.getoper() == 10 && i_temp1 != i_temp2)
      return (1);
```

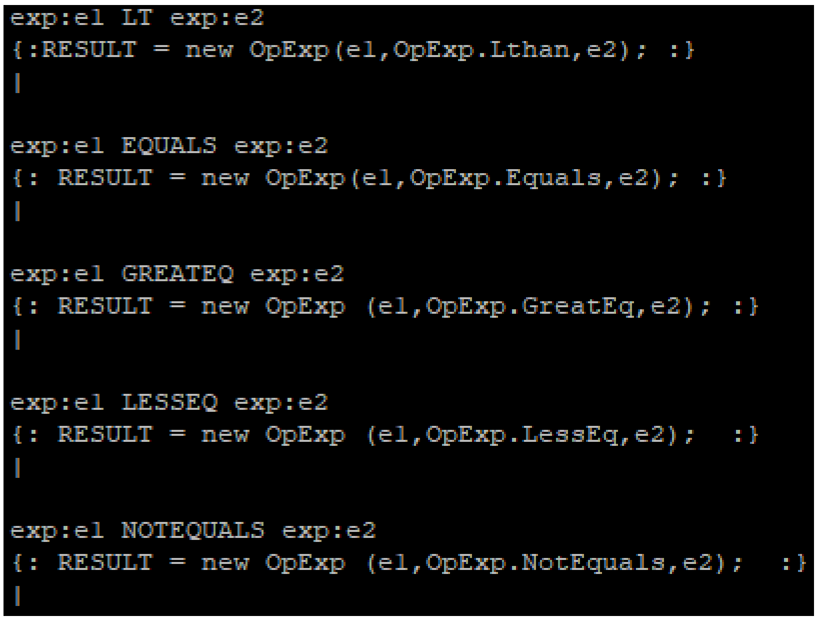
As each operator is referred to by an integer value, I needed to assign each new operator a value.



```
class OpExp extends Exp {
   Exp left, right; int oper;
   final static int Plus=1,Minus=2,Times=3,Div=4,Gthan=5, Lthan=6, Equals=7, GreatEq=8, LessEq=9, NotEquals=10;
   OpExp(Exp l, int o, Exp r) {left=l; oper=o; right=r;}
   public int getoper() {return oper;}
   public Exp getleft() {return left;}
   public Exp getright() {return right;}
}
```

This turns the terminal symbol into an integer value that can be used above.

Now that the functionality is working, I needed to add the syntax to the parser so an input can become a terminal, become an integer, become a Boolean, and be used however it needs to be.



```
exp:e1 LT exp:e2
{:RESULT = new OpExp(e1,OpExp.Lthan,e2); :}
|

exp:e1 EQUALS exp:e2
{: RESULT = new OpExp(e1,OpExp.Equals,e2); :}
|

exp:e1 GREATEQ exp:e2
{: RESULT = new OpExp (e1,OpExp.GreatEq,e2); :}
|

exp:e1 LESSEQ exp:e2
{: RESULT = new OpExp (e1,OpExp.LessEq,e2);  :}
|

exp:e1 NOTEQUALS exp:e2
{: RESULT = new OpExp (e1,OpExp.NotEquals,e2);  :}
|
```

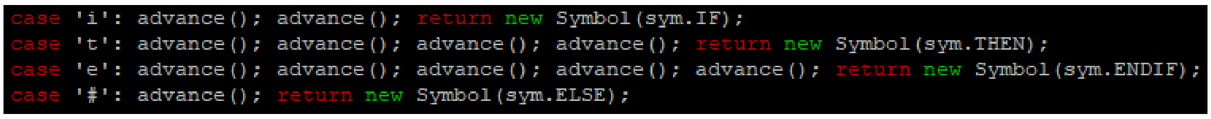
As GT was already implemented, there is no need to include it here.

With all of these steps implemented, an input string can be parsed into each of these new Boolean conditions and produce the correct outputs as seen in the test results.

Given more time, I would have included compound statements, including || and && functionality – such that (0 == 1||1 == 1) and (x == 1 && y == x) would have the expected result.

Another success I had within this project was to be able to utilise an if else within the syntax – using the syntax “start; If 1 > 0 then print(1); # print(0); endif; end;” where # is the symbol used for else.

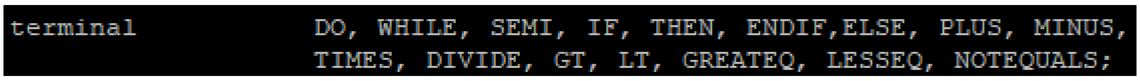
I first needed to add the relevant code to the scanner.java to be able to take this input and convert it into a usable set of terminal and non-terminal symbols. To do this, I added the following code:



```
case 'i': advance(); advance(); return new Symbol(sym.IF);
case 't': advance(); advance(); advance(); advance(); return new Symbol(sym.THEN);
case 'e': advance(); advance(); advance(); advance(); advance(); return new Symbol(sym.ENDIF);
case '#': advance(); return new Symbol(sym.ELSE);
```

As mentioned previously, I was forced to use a symbol for either Else or Endif as having both start with E was confusing the parser.

Next I needed to add the new terminals to the existing list of terminals. The new list is as follows:

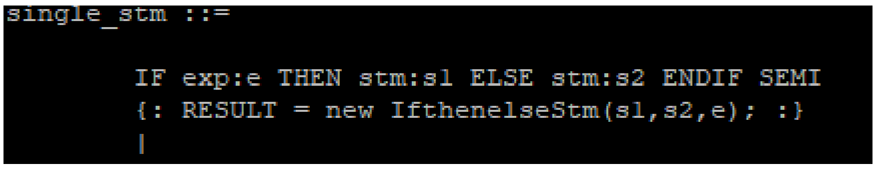


```
terminal            DO, WHILE, SEMI, IF, THEN, ENDIF,ELSE, PLUS, MINUS,
                    TIMES, DIVIDE, GT, LT, GREATEQ, LESSEQ, NOTEQUALS;
```

(Split to fit better into an image)

This is the same image used for the Boolean operators as they were both done before this documentation was created.

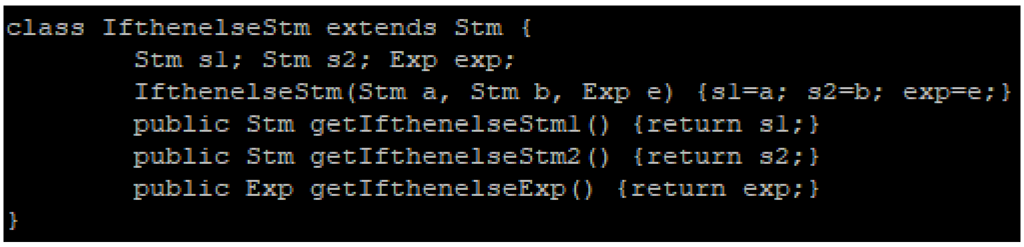
After the new terminals were created I needed to add the syntax to the parser so that the parser can use the new terminals. This was done by declaring the new syntax in the single_stm of the grammar:



```
single_stm ::=

        IF exp:e THEN stm:s1 ELSE stm:s2 ENDIF SEMI
        {: RESULT = new IfthenelseStm(s1,s2,e); :}
        |
```

Within this, exp is the condition that is being checked. Stm:s1 is the code that will execute if the condition is true, and Stm:s2 is the code that will execute if the condition is false.

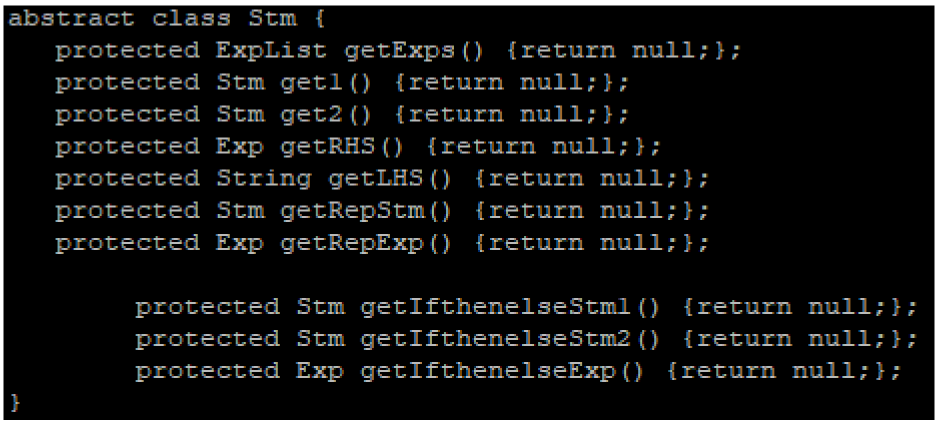
Now that the syntax has been added, I needed to create the IfthenelseStm class to extend the abstract Stm class with the new functionality. This was done by creating the new class and creating it as an extension to Stm. The implementation is as follows:



```
class IfthenelseStm extends Stm {
        Stm s1; Stm s2; Exp exp;
        IfthenelseStm(Stm a, Stm b, Exp e) {s1=a; s2=b; exp=e;}
        public Stm getIfthenelseStm1() {return s1;}
        public Stm getIfthenelseStm2() {return s2;}
        public Exp getIfthenelseExp() {return exp;}
}
```

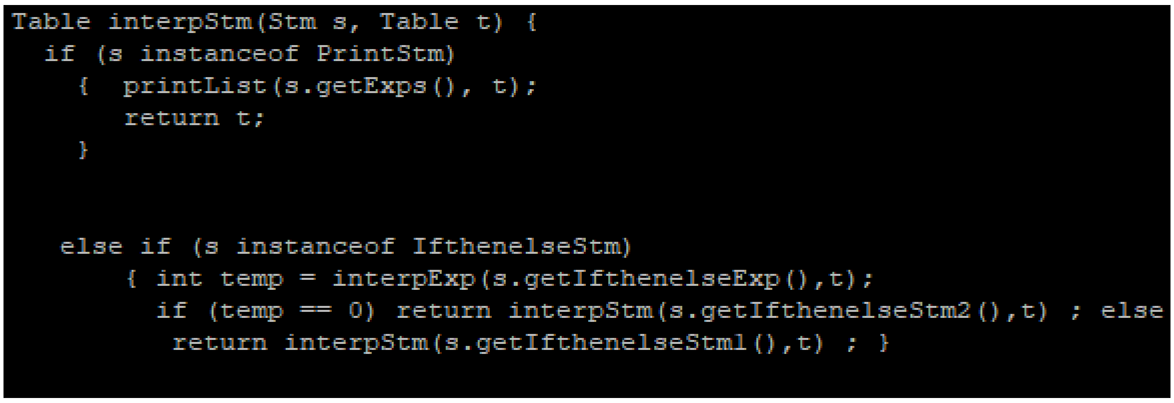
As there are two statements that can be called, there needs to be two get methods for statement, as they aren't the same statement each time, I chose to create two separate functions – one for each statement.

To use these new functions, I needed to add them to the initial abstract Stm class which return null be default. This was easily done by defining them and the return:



```
abstract class Stm {
   protected ExpList getExps() {return null;};
   protected Stm get1() {return null;};
   protected Stm get2() {return null;};
   protected Exp getRHS() {return null;};
   protected String getLHS() {return null;};
   protected Stm getRepStm() {return null;};
   protected Exp getRepExp() {return null;};

        protected Stm getIfthenelseStm1() {return null;};
        protected Stm getIfthenelseStm2() {return null;};
        protected Exp getIfthenelseExp() {return null;};
}
```

There is just one more thing that needs to be done to allow the functionality to work as intended. The logic of the statement must be implemented. As it is currently it could be completely ignored, run both statements, or just outright crash. To prevent this, logic for instances of IfElse statements need to be added within the interpStm table as follows:



```
Table interpStm(Stm s, Table t) {
  if (s instanceof PrintStm)
    {  printList(s.getExps(), t);
       return t;
    }


   else if (s instanceof IfthenelseStm)
       { int temp = interpExp(s.getIfthenelseExp(),t);
         if (temp == 0) return interpStm(s.getIfthenelseStm2(),t) ; else
          return interpStm(s.getIfthenelseStm1(),t) ; }
```

And with that, the implementation of an If Else statement is complete. Given more time I would try to implement statements both with and without an else tied to a statement to give functionality. This could be a separate set of functions similar to above, or treat missing else information as null and run no code if the condition is failed, until the endif is found.

Due to a lack of time and understanding of the development language, I was unable to implement any more functions, including a Call() function or an extension of such a function.