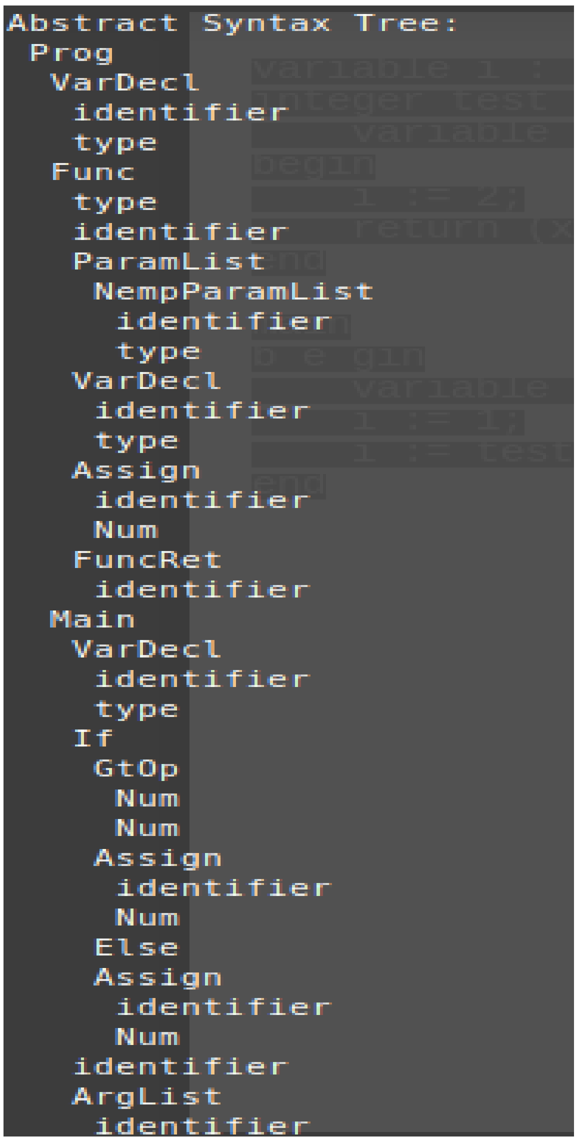
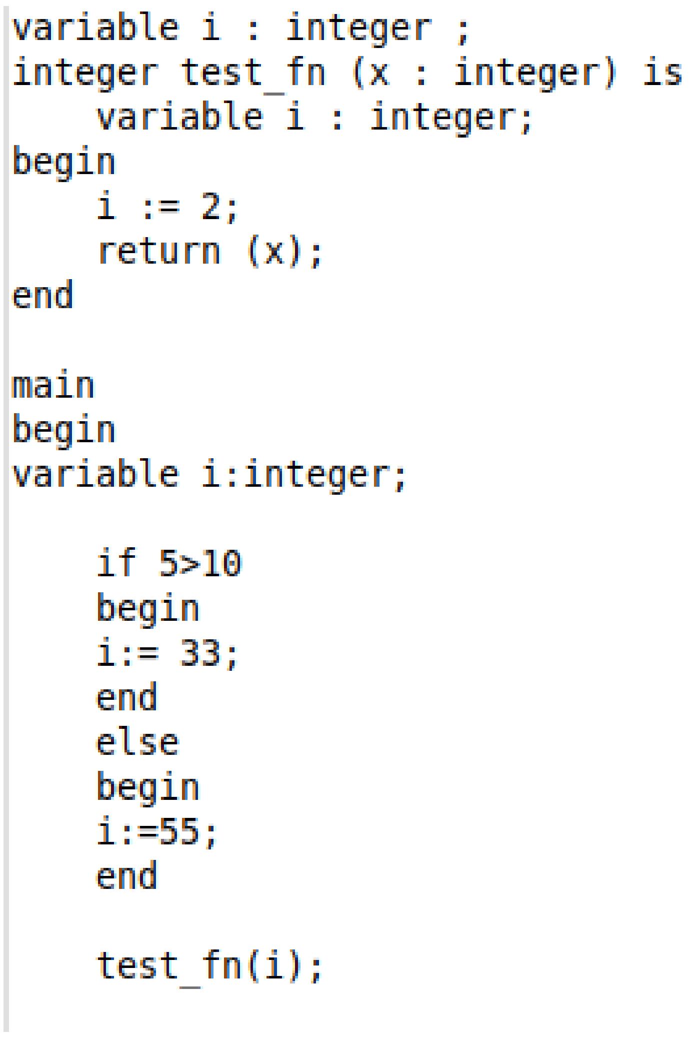
CA4010 Assignment 2 Report - Alex Randles (15302766)

Abstract Syntax Tree



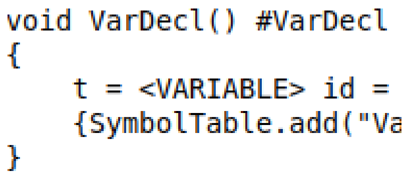
```
Abstract Syntax Tree:
 Prog
  VarDecl
   identifier
   type
  Func
   type
   identifier
   ParamList
    NempParamList
     identifier
     type
   VarDecl
    identifier
    type
   Assign
    identifier
    Num
   FuncRet
    identifier
  Main
   VarDecl
    identifier
    type
   If
    GtOp
     Num
     Num
    Assign
     identifier
     Num
    Else
    Assign
     identifier
     Num
   identifier
   ArgList
    identifier
```



```
variable i : integer ;
integer test_fn (x : integer) is
    variable i : integer;
begin
    i := 2;
    return (x);
end

main
begin
variable i:integer;

    if 5>10
    begin
    i:= 33;
    end
    else
    begin
    i:=55;
    end

    test_fn(i);
```

- The abstract syntax tree was implemented by declaring nodes at whichever point I felt it was relevant to inform the user or store information about a certain node in order to do my semantic checks in the later part of this assignment. I will break down this abstract syntax tree above and explain how it was created.
- A node is created at the start of each Program run, #Prog, A node would also be created when the program enters Main, everything within would be its children nodes.
- Line 1 of the code declares a variable of type of integer. For each variable

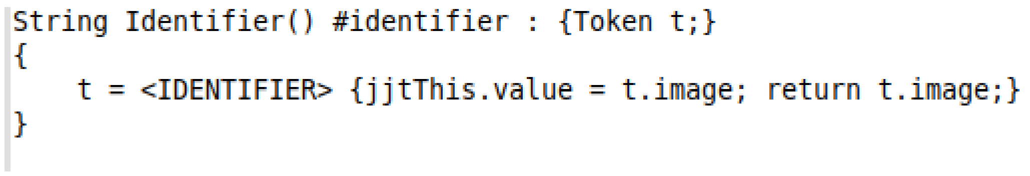
declaration I created a VarDecl node on the tree. This was done by the following piece of code.



```
void VarDecl() #VarDecl
{
    t = <VARIABLE> id =
    {SymbolTable.add("Va
}
```

The #VarDecl will create a node on the tree each time a variable declaration call is made.

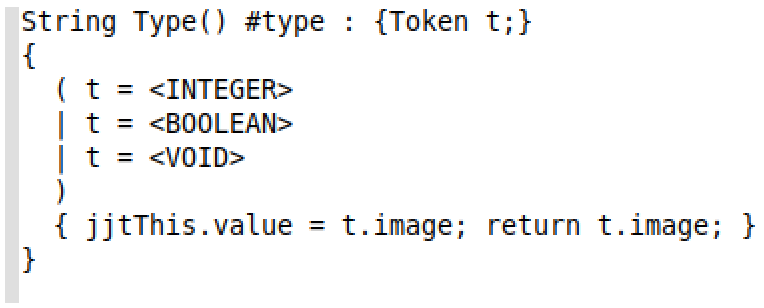
After this I thought it would be necessary to have a node to store the identifier of that declaration. This was done by done by creating an string function that created a child node if an identifier was matched.



```
String Identifier() #identifier : {Token t;}
{
    t = <IDENTIFIER> {jjtThis.value = t.image; return t.image;}
}
```

If an <IDENTIFIER> token is matched, it creates an identifier node and sets the value of the identifier to the image of the token, it must return it as this is a string function.

The variable declaration must also have a type e.g boolean, integer
This is done by a similar method as above. This will create a type node.
Below the identifier node which are both children of the Variable declaration node.

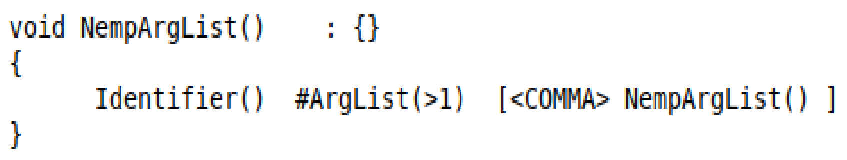


```
String Type() #type : {Token t;}
{
  ( t = <INTEGER>
  | t = <BOOLEAN>
  | t = <VOID>
  )
  { jjtThis.value = t.image; return t.image; }
}
```

- For a function declaration a ParamList node is created which will have a number of NempParamList nodes depending on how many paramaters are declared within that function. The Parent node will “Func” and all the contents included in that function will be children of the Func node.

- For Assignments i.e i:=33; , I made an Assign node with two children, the first child being the identifier (i) in this case and the second child being the value which the identifier is assigned (33).

- For If statements I created a parent If node with a number of children nodes but mainly the else node. The first child of the If statement node will be the comparison operation being used. In this example Greater then operation (>).
- For operations like this, I created two children nodes which were the two values being compared. In this example Num and Num.
- Finally the Else statement node which will have all the information within the else statement as its children.
- The last line relates to a function call. This is done by creating an identifier note. In this example the function test_fn is the identifier. It will have an ArgList node as it child which will contain all the identifiers that the function call has been passed. The number of children the ArgList node has depends on the number of arguments passed. How the number is increased is shown below.

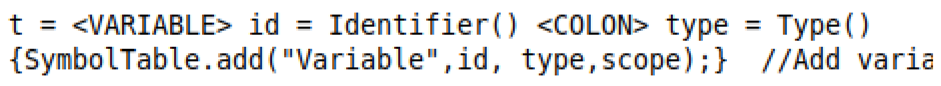


```
void NempArgList()    : {}
{
      Identifier()  #ArgList(>1)  [<COMMA> NempArgList() ]
}
```

The ArgList(>1) creates an extra depth each time an arguments is passed.

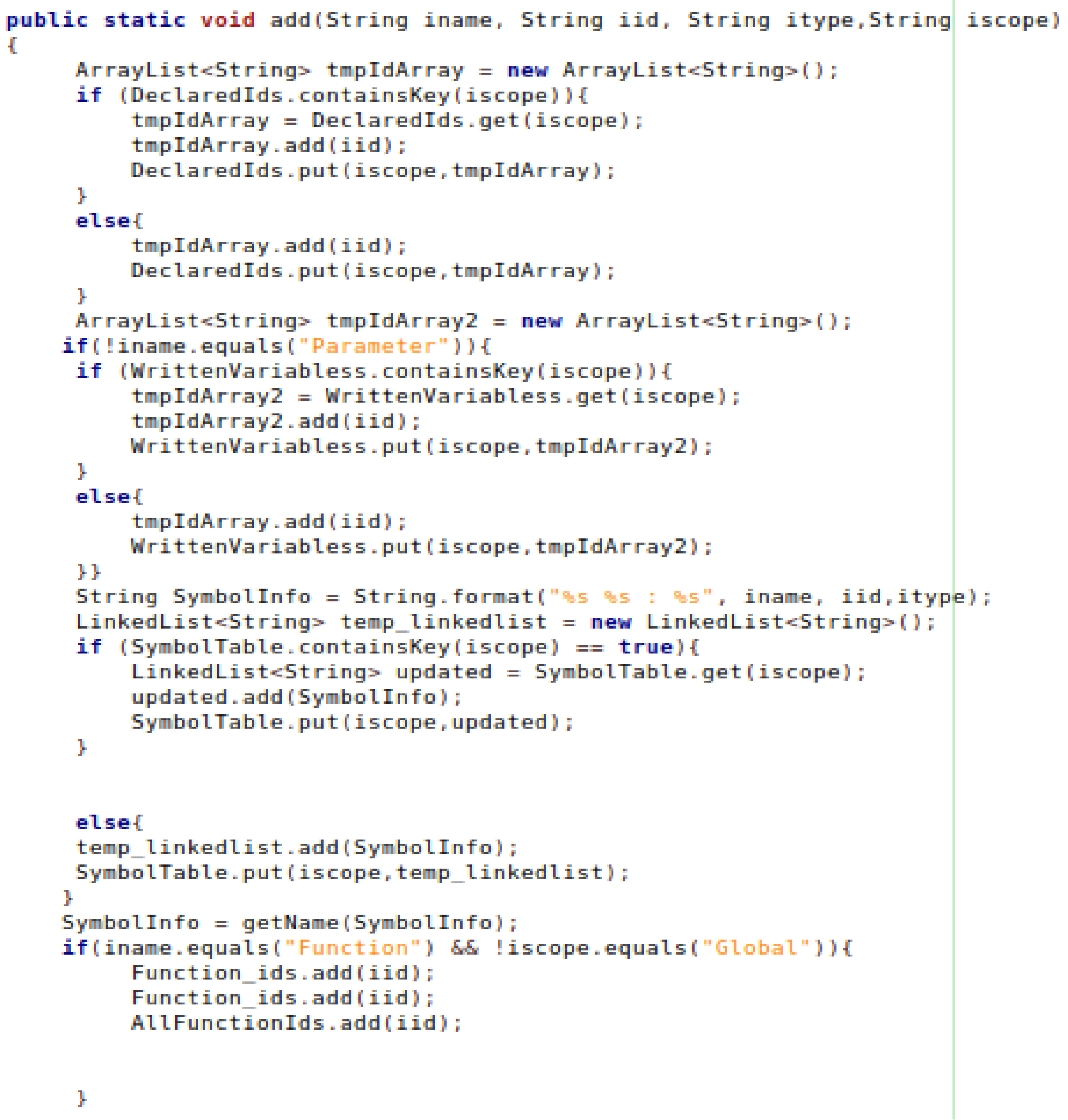
Symbol Table

- The symbol table was created by firstly initialising a global scope variable which had to be changed when the scope of the program changed e.g function, global, main. These were insert into a symbol table class as shown below.



```
t = <VARIABLE> id = Identifier() <COLON> type = Type()
{SymbolTable.add("Variable",id, type,scope);}  //Add varia
```

I created an add operation in the symbol table which added the symbols to a HashTable which had string keys and a linked list as its values. If the scope would be used as the key. If a symbol from that scope had already been declared then it be added to the end of the linkedlist for that scope.

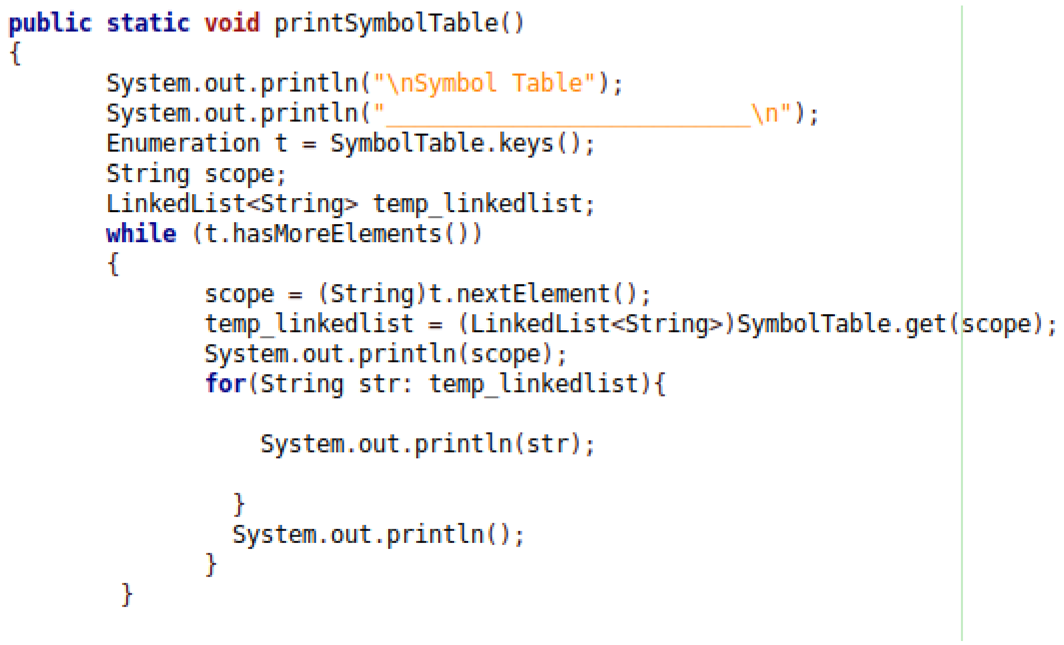


```java
public static void add(String iname, String iid, String itype,String iscope)
{
     ArrayList<String> tmpIdArray = new ArrayList<String>();
     if (DeclaredIds.containsKey(iscope)){
         tmpIdArray = DeclaredIds.get(iscope);
         tmpIdArray.add(iid);
         DeclaredIds.put(iscope,tmpIdArray);
     }
     else{
         tmpIdArray.add(iid);
         DeclaredIds.put(iscope,tmpIdArray);
     }
     ArrayList<String> tmpIdArray2 = new ArrayList<String>();
    if(!iname.equals("Parameter")){
     if (WrittenVariabless.containsKey(iscope)){
         tmpIdArray2 = WrittenVariabless.get(iscope);
         tmpIdArray2.add(iid);
         WrittenVariabless.put(iscope,tmpIdArray2);
     }
     else{
         tmpIdArray.add(iid);
         WrittenVariabless.put(iscope,tmpIdArray2);
     }}
     String SymbolInfo = String.format("%s %s : %s", iname, iid,itype);
     LinkedList<String> temp_linkedlist = new LinkedList<String>();
     if (SymbolTable.containsKey(iscope) == true){
         LinkedList<String> updated = SymbolTable.get(iscope);
         updated.add(SymbolInfo);
         SymbolTable.put(iscope,updated);
     }


     else{
     temp_linkedlist.add(SymbolInfo);
     SymbolTable.put(iscope,temp_linkedlist);
    }
    SymbolInfo = getName(SymbolInfo);
    if(iname.equals("Function") && !iscope.equals("Global")){
         Function_ids.add(iid);
         Function_ids.add(iid);
         AllFunctionIds.add(iid);


     }
```

It adds the Symbol Information in a String format which constist of the name (variable,constant), id(i,j), type (boolean, integer) and finally the scope which the Symbol is in. It adds certain parts of the symbol to other ArrayList and HashMaps To be used later in the semantic checker class.

Printing the Symbol Table

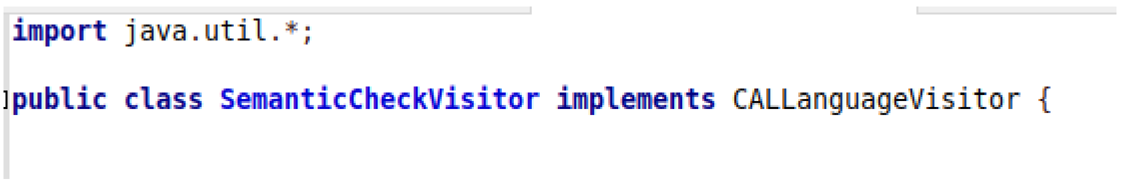
This is done by first Printing the key of the Symbol table which is the scope, then creating a tmp linkedlist variable to hold the values of that scope and iterating through that linkedlist and keep repeating till the Symbol table has no more keys left.



```
public static void printSymbolTable()
{
        System.out.println("\nSymbol Table");
        System.out.println("_____________________________\n");
        Enumeration t = SymbolTable.keys();
        String scope;
        LinkedList<String> temp_linkedlist;
        while (t.hasMoreElements())
        {
                scope = (String)t.nextElement();
                temp_linkedlist = (LinkedList<String>)SymbolTable.get(scope);
                System.out.println(scope);
                for(String str: temp_linkedlist){

                    System.out.println(str);

                  }
                  System.out.println();
                }
         }
```

Semantic Check

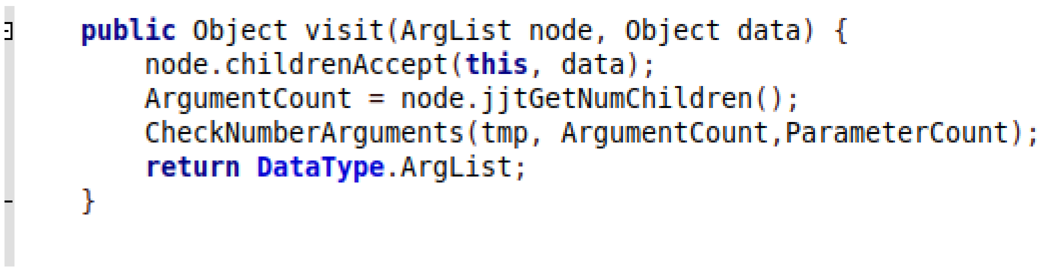
This is was done by a huge number of checks on each node on the abstract tree. The Semantic checks were done by implenting a class with the visitor helper. This helped me to visit each node and get the information about them I needed.



```
import java.util.*;

public class SemanticCheckVisitor implements CALLanguageVisitor {
```

An example of one of the semantic checks was “Does every function call have the correct number of arguments?”

I did this by checking the number of children the ArgList node had and check this against the Symbol Table parameter count for that function. If they didnt match I would then print a semantic check error message. The jjtGetNumChildren() function came in very handy for this as it found the number of children which I could then check the number of parameters declared for that function in the symbol Table.



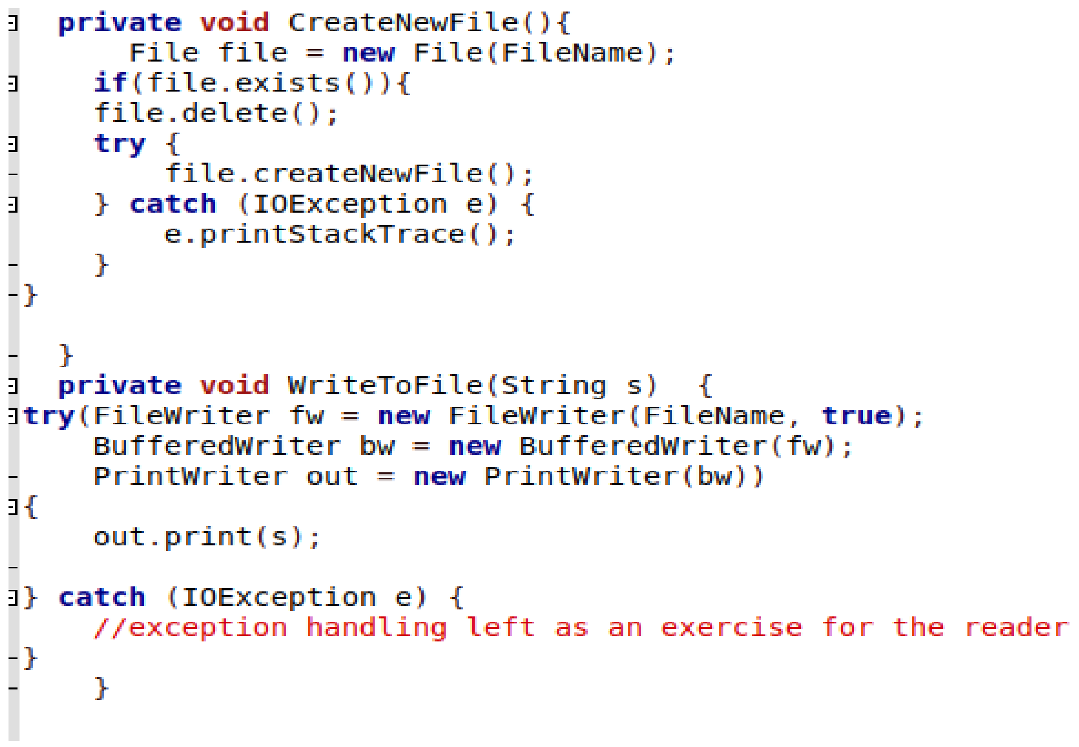
```
    public Object visit(ArgList node, Object data) {
        node.childrenAccept(this, data);
        ArgumentCount = node.jjtGetNumChildren();
        CheckNumberArguments(tmp, ArgumentCount,ParameterCount);
        return DataType.ArgList;
    }
```

Intermediate code generation

I followed the syntax directed definition approach which we covered when doing exam revision in class.

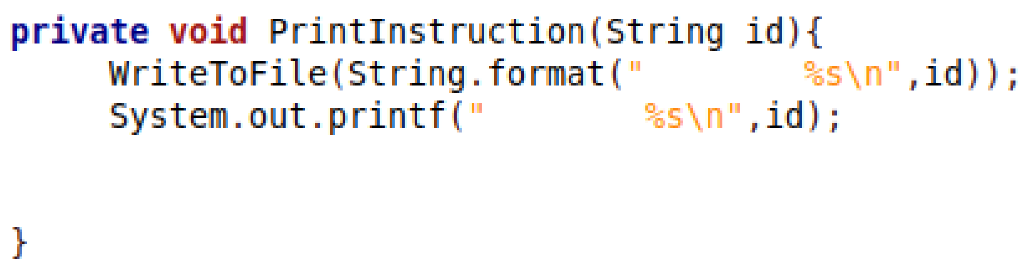
This was done by a similar method to the semantic checks but instead of checking the nodes I printed out to the command line and to a file named. “ThreeAddressCode.ir” the representation of that piece of code. I used the visitor class to help me visit each node and decide what to print for that node.

First I had to create the file to store in the 3 address code, every time the program is ran the file must be overwritten so we only have the representation for the latest file that was given to the CALParser.



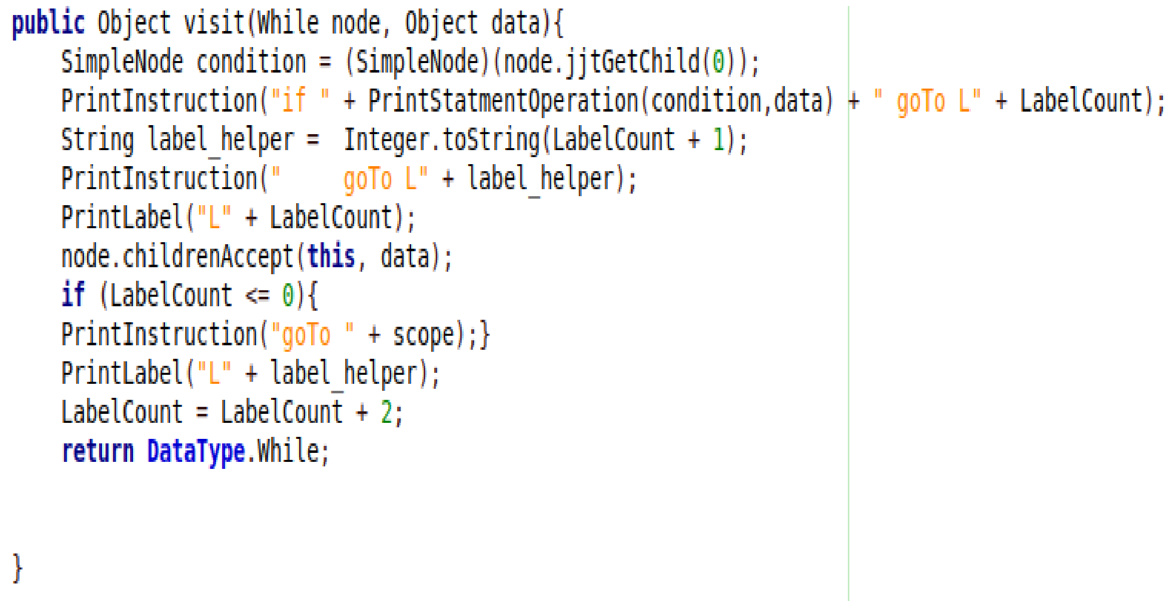
```
  private void CreateNewFile(){
        File file = new File(FileName);
     if(file.exists()){
     file.delete();
     try {
         file.createNewFile();
     } catch (IOException e) {
         e.printStackTrace();
     }
}

  }
  private void WriteToFile(String s)  {
try(FileWriter fw = new FileWriter(FileName, true);
     BufferedWriter bw = new BufferedWriter(fw);
     PrintWriter out = new PrintWriter(bw))
{
     out.print(s);

} catch (IOException e) {
     //exception handling left as an exercise for the reader
}
     }
```

created a number of printing functions at the top of the class which would help me to indent the instruction the program was printing correctly. These functions printing to the command line as well as to the file.



```java
private void PrintInstruction(String id){
    WriteToFile(String.format("        %s\n",id));
    System.out.printf("        %s\n",id);


}
```

The while loop was one of the most difficult parts to implement as I had to create a goTo label for when the condition was true and when it was false. Also if i was true I would have to go back to that label to check the condition again. I first printed the if statement followed by a goTo label which would be the one we'd go to if it was true. Then I had to create that label with a call back to the label with the if condition in it. Below the if statement I had another goTo Label which is where the program would jump to if the condition became false. I kept a global label count which was incremented each time and if the label count was 0 this means I would have to jump back to the previous label which was the scope instead of one which I created.



```java
public Object visit(While node, Object data){
    SimpleNode condition = (SimpleNode)(node.jjtGetChild(0));
    PrintInstruction("if " + PrintStatmentOperation(condition,data) + " goTo L" + LabelCount);
    String label_helper =  Integer.toString(LabelCount + 1);
    PrintInstruction("     goTo L" + label_helper);
    PrintLabel("L" + LabelCount);
    node.childrenAccept(this, data);
    if (LabelCount <= 0){
    PrintInstruction("goTo " + scope);}
    PrintLabel("L" + label_helper);
    LabelCount = LabelCount + 2;
    return DataType.While;


}
```