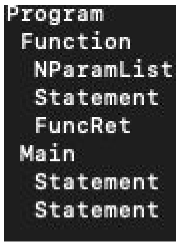
Compilers Assignment 2

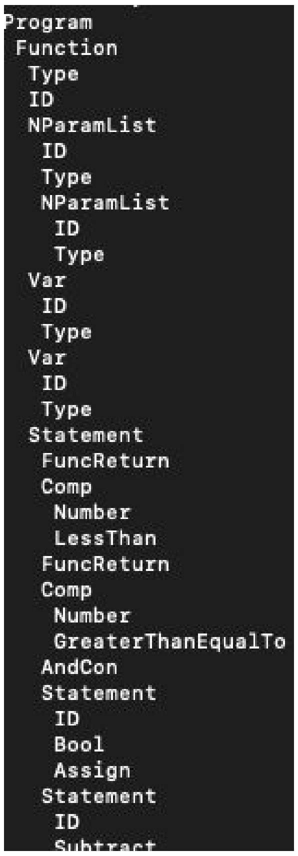
1. Abstract Syntax Tree

For my second assignment I had to start off by doing research in the best way to implement an abstract syntax tree. As described in this tutorial https://tomassetti.me/parse-tree-abstract-syntax-tree/ and the videos provided, I started off by defining our Node. This was my program function, then I decided to trace an example program and decide what nodes I needed as decorators. I firstly created a very basic AST with a few nodes as I did not know what nodes I needed for the semantic analyser for later in the program. My first AST looked like this after I used root.dump to print it:



```
Program
 Function
  NParamList
  Statement
  FuncRet
 Main
  Statement
  Statement
```

This was a very simple implementation before I needed to refactor it for my semantic analyser. I focused on the main functions to add decorators, such as the function, the statement and the main. This gave me a general understanding of how the tree should work. After all my semantic checks were completed, my final tree looks like this:

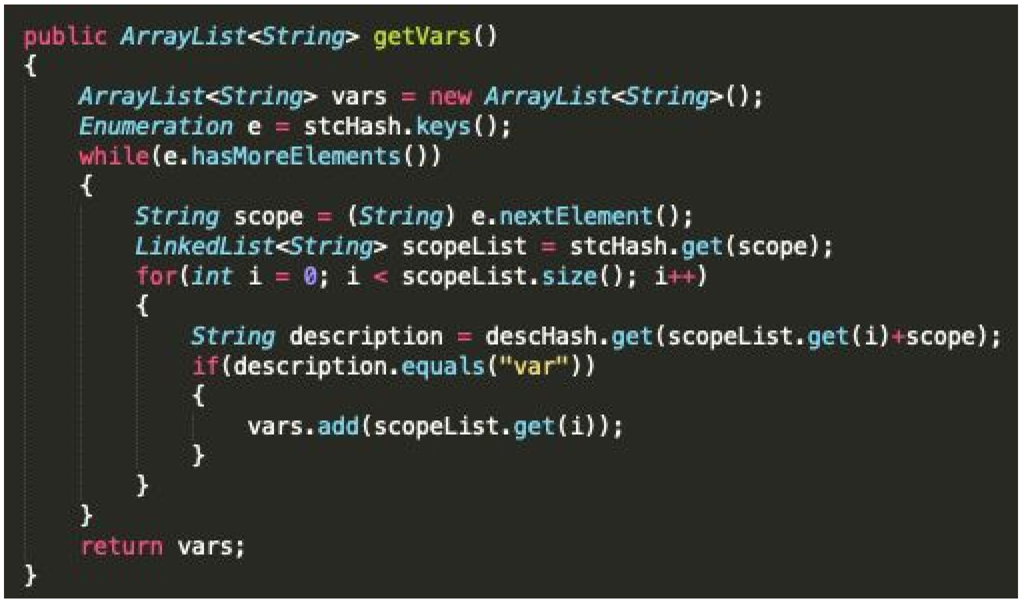


```
Program
 Function
  Type
  ID
  NParamList
   ID
   Type
   NParamList
    ID
    Type
  Var
   ID
   Type
  Var
   ID
   Type
  Statement
   FuncReturn
   Comp
    Number
    LessThan
   FuncReturn
   Comp
    Number
    GreaterThanEqualTo
   AndCon
   Statement
    ID
    Bool
    Assign
   Statement
    ID
```

The final AST was a lot more complex than the beginning, as I was working on the semantic checks, I was able to figure out the exact nodes and decorators that I needed. We see the structure here for the function node, it has a id and type, and also has its parameters and its ids and types and all of the variables in the function body. I also needed to add non terminals such as id() so that I could get some values of nodes such as function and parameter list. This allowed me to access id's and types for the semantic checks.

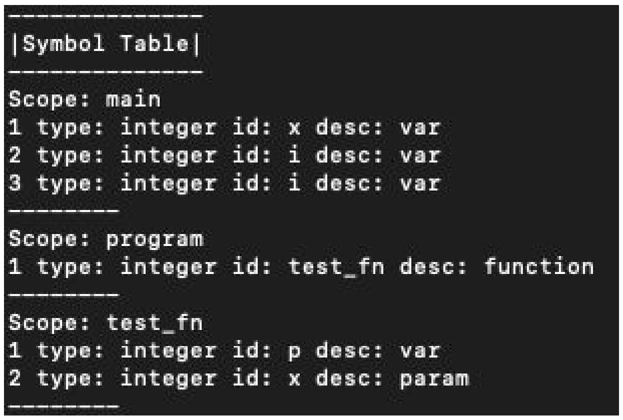
2. Symbol Table

To implement the symbol table I had a look at online resources for the best way to do this. Using the tutorial here https://www.tutorialspoint.com/compiler_design/compiler_design_symbol_table.htm it describes a symbol tableand the best ways to implement it. The article explains that hashtables are the most efficient way to handle a symbol table as they are O(1) for insertion and lookup if correctly implemented. The lecture notes advise using chaining so I have used 3 hash table, using them for the scope, the type, and to determine if we have a function, or parameter etc. I also added get and helper methods so that our visitor could retrieve information from the symbol table for the semantic checks. We have functions like getVars() which fetches all the variables and their scopes from the symbol table.



```
public ArrayList<String> getVars()
{
    ArrayList<String> vars = new ArrayList<String>();
    Enumeration e = stcHash.keys();
    while(e.hasMoreElements())
    {
        String scope = (String) e.nextElement();
        LinkedList<String> scopeList = stcHash.get(scope);
        for(int i = 0; i < scopeList.size(); i++)
        {
            String description = descHash.get(scopeList.get(i)+scope);
            if(description.equals("var"))
            {
                vars.add(scopeList.get(i));
            }
        }
    }
    return vars;
}
```

Here we go through all of the items in our scopes list, we make sure that we make a new arraylist for all the variables. We add all the variables with their scopes to the list and return to. This allows us to do semantic checks in the visitor class. I also created the printing class there to print the symbol table with the correct scopes. The symbol table looked like this for a basic function:

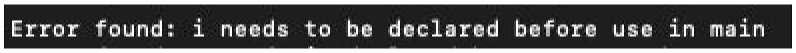

```
--------------
|Symbol Table|
--------------
Scope: main
1 type: integer id: x desc: var
2 type: integer id: i desc: var
3 type: integer id: i desc: var
--------
Scope: program
1 type: integer id: test_fn desc: function
--------
Scope: test_fn
1 type: integer id: p desc: var
2 type: integer id: x desc: param
--------
```

We see that in the main scope we have 1 id 'x' of type integer and we are told that it is a variable. We also see all the other scopes such as program and the test_fn function.

3. Semantic Checks

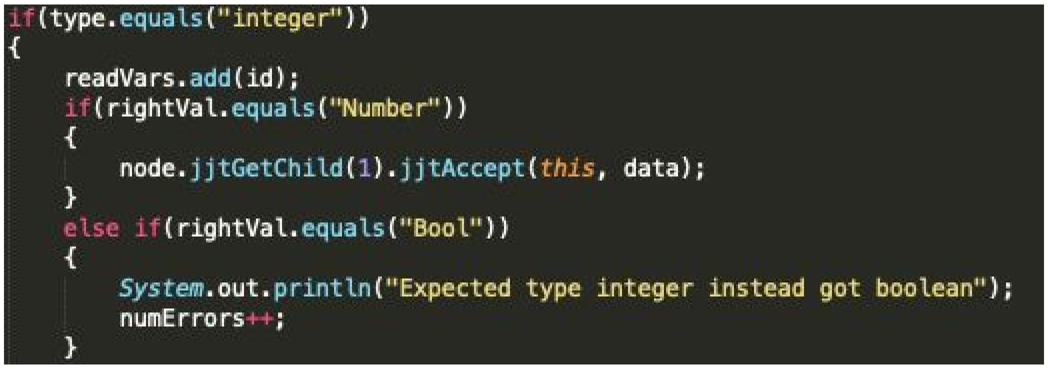
The next part was to implement all of the semantic checks that were described in the assignment documentation. This was done in the semantic check visitor class with java. The program visits every node recursively and return data and perform semantic checks.

The first check was to if all the variables were declared before use. This was done using the declared function which was invoked when we are in the statement node, this checks that the id is declared with the correct scope or the program scope so that it can be used anywhere. If the function returns false, that means that the variable is not declared in the correct scope and then we get this error.


```
Error found: i needs to be declared before use in main
```

The next semantic check was to check if there was no identifier declared more than once in scope. This is done by the multiple declarations function which runs at the end of the program after we finish traversing the tree. Then we check if there are two or more declarations for the same variable in the same scope.

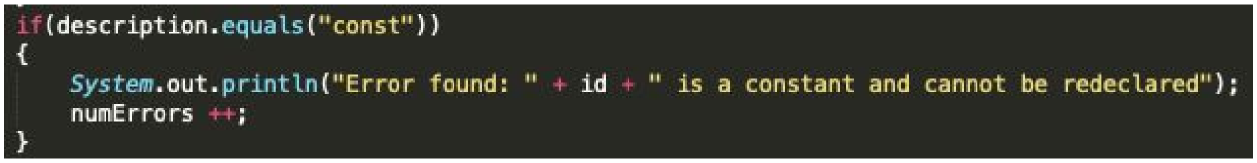
After this the next semantic check was to check if the left hand side of an assignment variable the correct type. This was done in the statement when we have a ID and if the variable is declared we check the type of the right hand side, and this must be equal to the type of the ID.



```
if(type.equals("integer"))
{
    readVars.add(id);
    if(rightVal.equals("Number"))
    {
        node.jjtGetChild(1).jjtAccept(this, data);
    }
    else if(rightVal.equals("Bool"))
    {
        System.out.println("Expected type integer instead got boolean");
        numErrors++;
    }
```

This check is if we have an integer on the left hand side, if we have, we check if the right hand side is a Number or returns a number. If we get a boolean we print an error and increase the number of errors count.

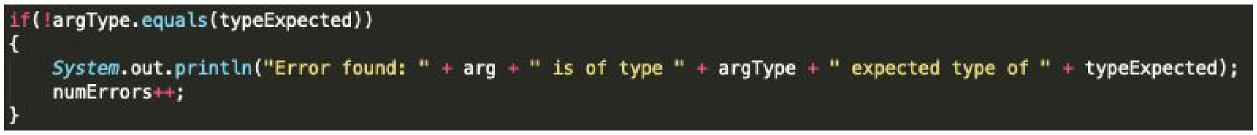
The next checks were to check if the the arguments of an arithmetic operator the integer variables or integer constants and are the arguments of a boolean operator boolean variables or boolean constants. If it is a constant it must be declared with a value and the value cannot change, and a variable must be initialised before being used. This is done in the statement node, this is done by using the symbol table by assigning a description to the id's. This is easily done as all ids have a description in the symbol table so to check if the id is a variable or a constant.



```
if(description.equals("const"))
{
    System.out.println("Error found: " + id + " is a constant and cannot be redeclared");
    numErrors ++;
}
```

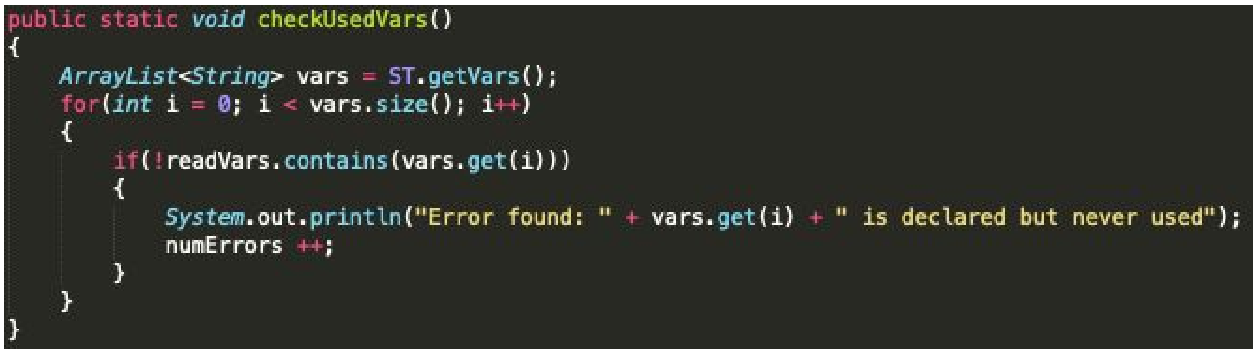
This checks the description if it is a constant, if it is it print an error message and increases the number of errors.

For the next semantic checks there needed to be checks if there is a function for every invoked identifier and does every function call have the correct number of arguments. We first check if the function has the correct number of arguments. We get this from the arguments list that has the number of parameters passed. Then I can check how much parameters the function has in the symbol table and compare it to the number of children of arguments list. Then it is possible to check if the parameter is the correct type. We get the type the parameters should be from the symbol table and compare it to the passed parameters type. If the types do not match we output an error message and increase the error count.



```
if(!argType.equals(typeExpected))
{
    System.out.println("Error found: " + arg + " is of type " + argType + " expected type of " + typeExpected);
    numErrors++;
}
```

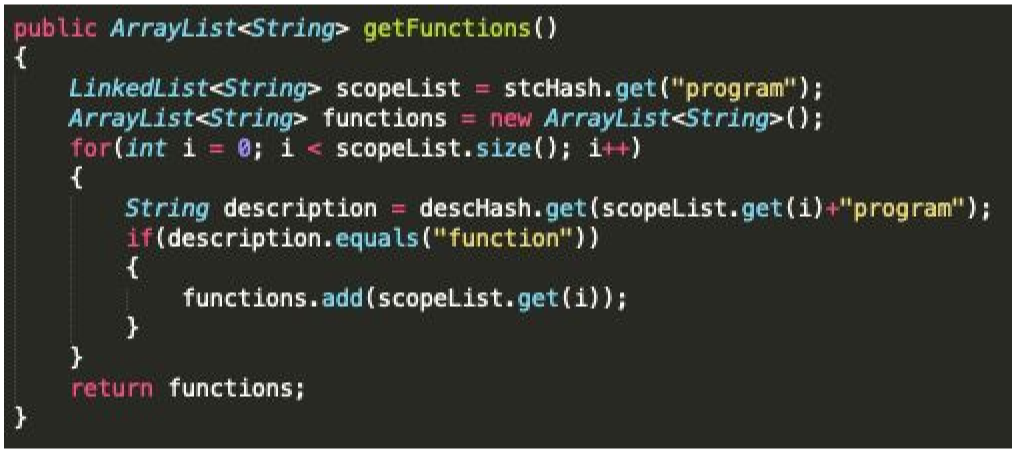
The next semantic check is to see if every variable written to and read from. To do this, I created an list to store all the variables that are read. Once a variable is read, it is added to the list and then once the program completes, the list is then checked against all the variables declared.



```java
public static void checkUsedVars()
{
    ArrayList<String> vars = ST.getVars();
    for(int i = 0; i < vars.size(); i++)
    {
        if(!readVars.contains(vars.get(i)))
        {
            System.out.println("Error found: " + vars.get(i) + " is declared but never used");
            numErrors ++;
        }
    }
}
```

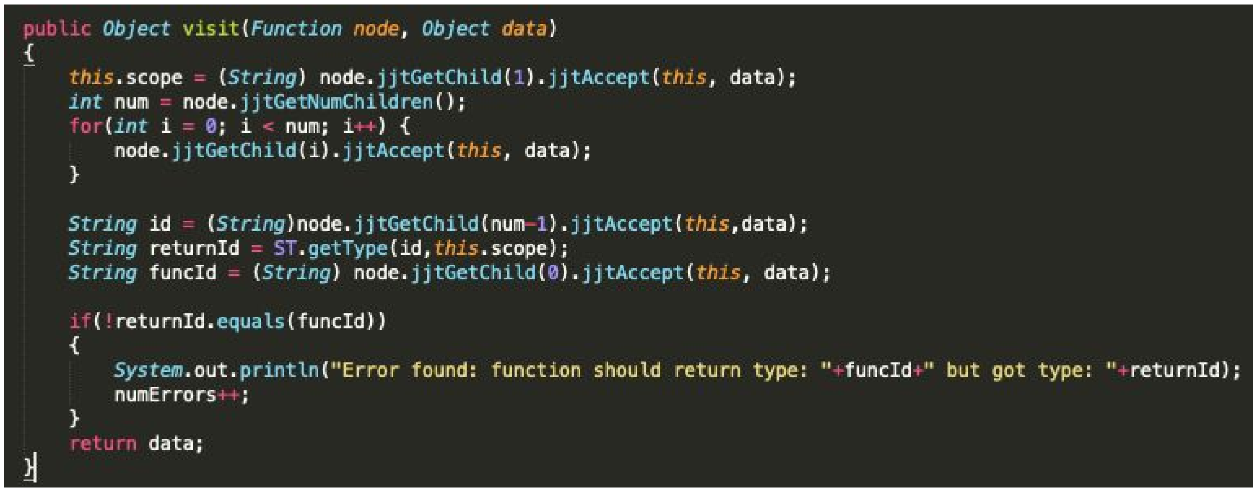
This function runs at the end of our program and checks if all the variables have been used. It sends an error for every declared but not used variable in the program.

The final semantic check is to see if every function called. Similar to the previous semantic check everytime there is a function called, we add it to a list. At the end of the program we call checkInvokedFunctions() which checks all the functions in the symbol table and compares them to the list of invoked function. If a function is not in the list of invoked functions, we send an error message and increment the count. The get functions method gets all the functions in the program so that we can compare them to the invoked functions list. It is described below.



```java
public ArrayList<String> getFunctions()
{
    LinkedList<String> scopeList = stcHash.get("program");
    ArrayList<String> functions = new ArrayList<String>();
    for(int i = 0; i < scopeList.size(); i++)
    {
        String description = descHash.get(scopeList.get(i)+"program");
        if(description.equals("function"))
        {
            functions.add(scopeList.get(i));
        }
    }
    return functions;
}
```

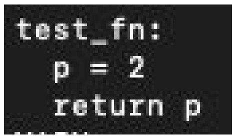
I added one extra semantic check to check if the return type matched the type of the function. This was done in the function class, by getting the type of the function by getting the correct child node and comparing to the type that is after the return child. The function is shown here.



```java
public Object visit(Function node, Object data)
{
    this.scope = (String) node.jjtGetChild(1).jjtAccept(this, data);
    int num = node.jjtGetNumChildren();
    for(int i = 0; i < num; i++) {
        node.jjtGetChild(i).jjtAccept(this, data);
    }

    String id = (String)node.jjtGetChild(num-1).jjtAccept(this,data);
    String returnId = ST.getType(id,this.scope);
    String funcId = (String) node.jjtGetChild(0).jjtAccept(this, data);

    if(!returnId.equals(funcId))
    {
        System.out.println("Error found: function should return type: "+funcId+" but got type: "+returnId);
        numErrors++;
    }
    return data;
}
```

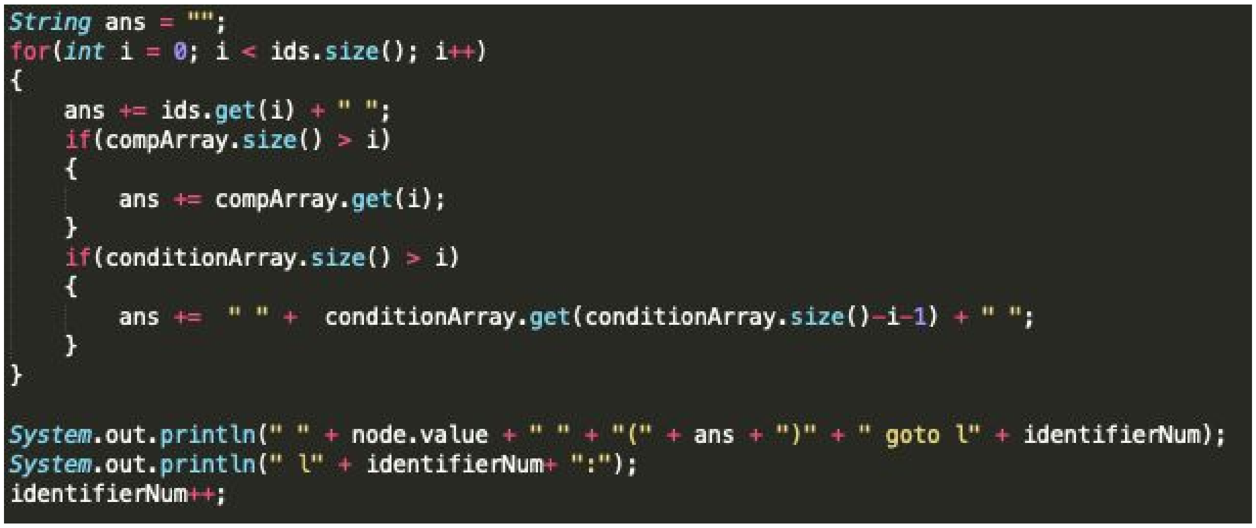
4. IR code generation

The ir code generation is similar to the semantic visitor class as it will visit each node of the program recursively and perform the operation for the code generation. This involves turning our parser cal language into machine readable TACI code. Variables are initialized in function and main scopes automatically when they get assigned a value. Functions are labeled when the function node is encountered. We print the function name, then we check for any variables declared and used and add those to the function body and finally look if there is anything returned by the function, if there is we add the ret value at the end. An example function is described here:

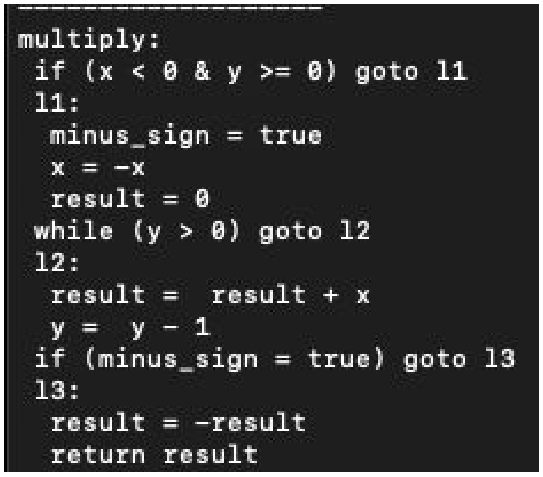


```
test_fn:
  p = 2
  return p
```

We then have to deal with the if/while statements in the program. We get the information we need from the ‘comp’ node, this gives us the id’s and the comparison operator and the conditional operators. As we encounter these nodes, we add the information to an array for each id,comparison or conditional, and later use the arrays to output the results. As we print the result we increment the label number counter which tells our conditional statements where to goto if the condition is true. The code for printing is shown here:

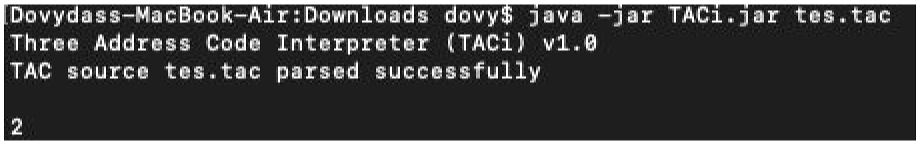

```java
String ans = "";
for(int i = 0; i < ids.size(); i++)
{
    ans += ids.get(i) + " ";
    if(compArray.size() > i)
    {
        ans += compArray.get(i);
    }
    if(conditionArray.size() > i)
    {
        ans +=  " " +  conditionArray.get(conditionArray.size()-i-1) + " ";
    }
}

System.out.println(" " + node.value + " " + "(" + ans + ")" + " goto l" + identifierNum);
System.out.println(" l" + identifierNum+ ":");
identifierNum++;
```

An example generated ir code for an if/while statements looked like this:


```
--------------------
multiply:
 if (x < 0 & y >= 0) goto l1
 l1:
  minus_sign = true
  x = -x
  result = 0
 while (y > 0) goto l2
 l2:
  result =  result + x
  y =  y - 1
 if (minus_sign = true) goto l3
 l3:
  result = -result
  return result
```

We see the label of the function at the top followed by the body of the function. The if condition goes to l1 if the condition is true. We also have while that goes to l2 as long as y is greater than zero. The IR code is generated and then added to a file with the name of the original file plus the .tac extension.

We then attempt to run the IR code using the TACi interpreter to see if everything is correctly generated.


```
[Dovydass-MacBook-Air:Downloads dovy$ java -jar TACi.jar tes.tac
Three Address Code Interpreter (TACi) v1.0
TAC source tes.tac parsed successfully

2
```

We see the TACi generator correctly parses our file and accepts the new generated machine code and is able to run the interpreter and we are able to get the simple output which print the value returned by the function.

Running the program

1. jjtree calparser.jjt
2. javacc calparser.jj
3. javac *.java
4. java CALParser [Name of test file]