CA4003 ASSIGNMENT 2

STUDENT DETAILS

NAME: CATHAL HUGHES

STUDENT NUMBER: 15417922

PROGRAMME: CASE

MODULE CODE: CA4003

SUBMISSION DATE: 17TH DECEMEBER 2018

MODULE COORDINATOR: DAVID SINCLAIR

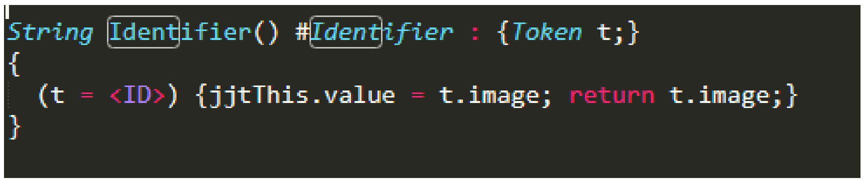
PLAGIARISM DECLARATION

INTRODUCTION

In this document, I will discuss my implementation of assignment two. Each section of the assignment will be discussed and the problems that arose while completing each section. I will explain how I overcame some of these problems and will provide code snippets along with console output.

ABSTRACT SYNTAX TREE

To generate the Abstract Syntax Tree for the given grammar was the first challenge of this assignment. This involved using our implementation for the first assignment adding it to a jjt file and then building the corresponding tree from the grammar specified in the file. In order to get the value of some nodes such as declarations, I implemented some terminals as non-terminals such as Identifier() and Number().

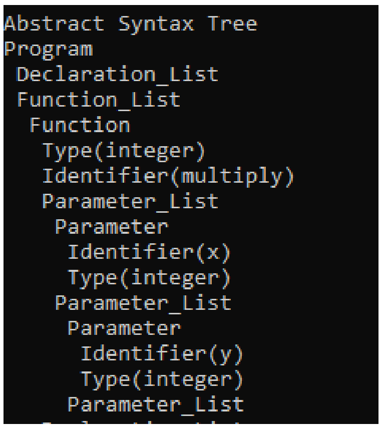


```
String Identifier() #Identifier : {Token t;}
{
  (t = <ID>) {jjtThis.value = t.image; return t.image;}
}
```

Using this meant it would far easier to keep track of IDs and Numbers which would make my semantic analysis a simpler task.

When I initially built the AST for my first assignment I realised there were some issues with regard to precedence. Although my previous grammar was not too far off being correct, it still took some tweaking in order for me to be happy with the layout of my AST. In order to implement some precedence new functions were created.

After overcoming the precedence issues with some rearrangement and creation of new functions, my AST was purely recursive. This was one of the most challenging aspects of the whole assignment but the most rewarding as I knew if my AST was strictly recursive, it would make the implementation of my Semantic Visitor far easier and also allow for easy implementation of the required semantic checks. An example issue I had to overcome involved the Paramater_List function. If I entered two Parameters into a function initially the Parameter_List node had two Paramater nodes as children. After implementing precedence, the Parameter_List node had two children, a Parameter node and a Parameter_List node. The second Parameter_List node had a Parameter node and Parameter_List node and the final Parameter_List node had no children indicating we had reached the end of the Parameters. Here is a snippet of my AST for the multiply function test which highlights how the precedence works for the parameters of the function:



```
Abstract Syntax Tree
Program
 Declaration_List
 Function_List
  Function
   Type(integer)
   Identifier(multiply)
   Parameter_List
    Parameter
     Identifier(x)
     Type(integer)
    Parameter_List
     Parameter
      Identifier(y)
      Type(integer)
     Parameter_List
```

To print the AST, I altered the dump() method in the SimpleNode class so I could see more information about the tokens. When I did this, I was able to see their values. When choosing nodes to be added to the AST, anything that was deemed relevant was added.

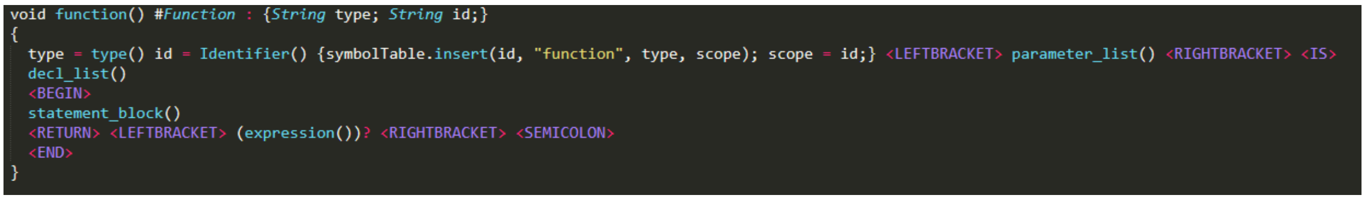
SYMBOL TABLE

The next step of the assignment involved creating a Symbol Table that was able to handle the concept of scope. I.e. when a variable is created inside a function, its scope is not global, rather its scope is the body of the function.

After much research online it seemed that the use of a HashTable would be the best approach to take. I decided that my Symbol Table would contain three hash tables. One hash table would use scope as the key and a LinkedList as the value which would track vraiables/IDs inside that particular scope. The other two hash tables would be essential for semantic checking later. Type Checking is something that would needed to be performed and so a hashtable was required to track the various types as well as a hash table to track the values of the different identifiers.

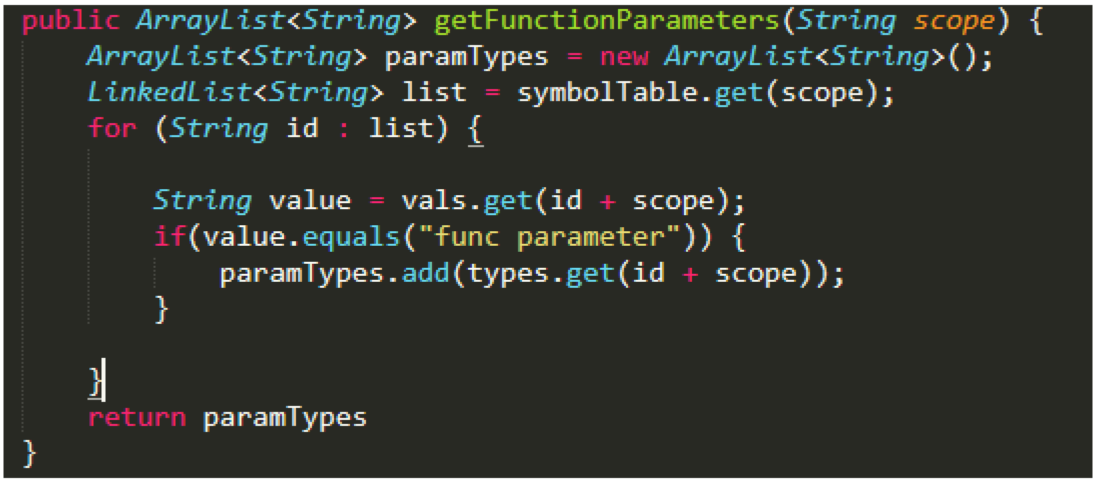
Some of the methods of the Symbol Table included a print method that would print out the symbol table. This works by printing each scope and for each scope it prints out the ID, type and value.

Also, an insert method which would add elements to the table. The insert method was used inside my .jjt file. For declarations (constant or variable) or functions the insert method would add the relevant information to the symbol table. Importantly here, is that every time a function was added to the symbol table the scope was updated to the id of that function, indicating that the scope has now changed to the body of that function.



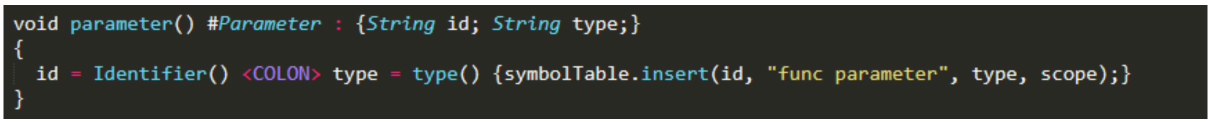
```
void function() #Function : {String type; String id;}
{
  type = type() id = Identifier() {symbolTable.insert(id, "function", type, scope); scope = id;} <LEFTBRACKET> parameter_list() <RIGHTBRACKET> <IS>
  decl_list()
  <BEGIN>
  statement_block()
  <RETURN> <LEFTBRACKET> (expression())? <RIGHTBRACKET> <SEMICOLON>
  <END>
}
```

I also implemented some getters that would be used during the semantic checking phase. Some of these included getFunctions() (used to help check if every function is called) and getFunctionParameters() (used in the semantic check to see if a function is called with the correct number of arguments and that the arguments are of the correct types).



```
public ArrayList<String> getFunctionParameters(String scope) {
    ArrayList<String> paramTypes = new ArrayList<String>();
    LinkedList<String> list = symbolTable.get(scope);
    for (String id : list) {

        String value = vals.get(id + scope);
        if(value.equals("func parameter")) {
            paramTypes.add(types.get(id + scope));
        }

    }
    return paramTypes
}
```

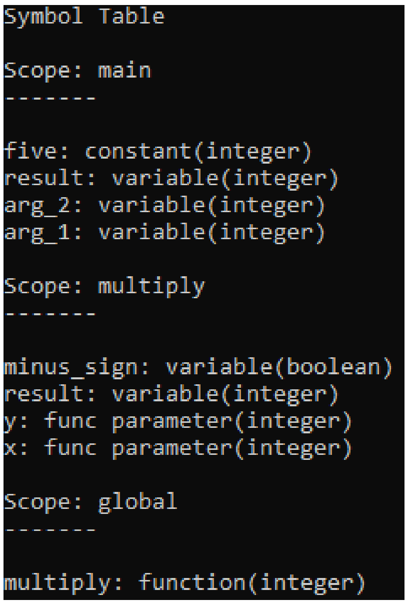
As you can see from the above code snippet this function returns all the types for values that are equal to “func parameter ”. The value “func parameter” is added to the symbol table by the parser specified in my .jjt file. The line of code where this is performed can be seen below.



```
void parameter() #Parameter : {String id; String type;}
{
  id = Identifier() <COLON> type = type() {symbolTable.insert(id, "func parameter", type, scope);}
}
```

The link between my parser and SymbolTable was essential when I needed to perform the semantic checks. Without an effectively written SymbolTable the semantic checks would have been very difficult to complete.

Here is what was printed by the symbol table when the multiply function from the assignment specification was used as a test:



```
Symbol Table

Scope: main
-------

five: constant(integer)
result: variable(integer)
arg_2: variable(integer)
arg_1: variable(integer)

Scope: multiply
-------

minus_sign: variable(boolean)
result: variable(integer)
y: func parameter(integer)
x: func parameter(integer)

Scope: global
-------

multiply: function(integer)
```

SEMANTIC CHECKS

The next aspect of this assignment was to perform a series of semantic checks. During this part of the assignment there were a number of checks carried out. Some of these semantic checks have been provided in the assignment specification whilst others were added in by myself. These semantic checks make the compiler useful.

To perform these semantic checks, a visitor class had to be created. I called my visitor class "PrintVisitor". This class must implement an interface that has been created by jjtree. It creates all the methods that must be implemented by the class. Each method pertains to a different Node that could be used in my Abstract Syntax Tree.

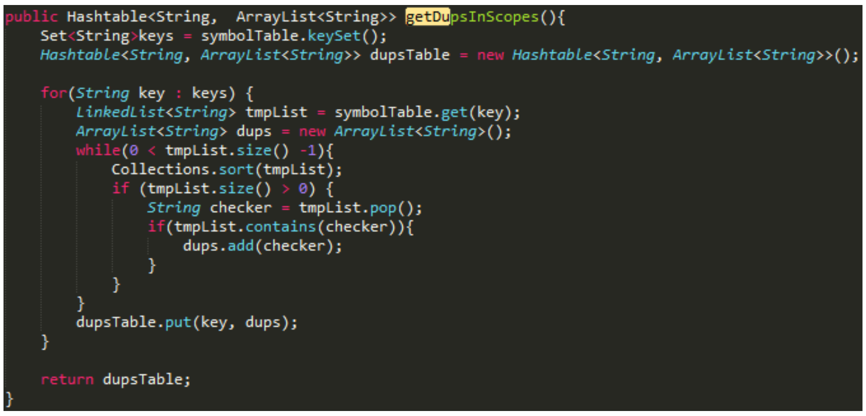
The semantic checks/errors are printed in a way that no error for a variable will be printed more than once. Also, what semantic checks that passed will be printed.

The following checks were implemented:

IS NO IDENTIFIER DECLARED MORE THAN ONCE IN THE SAME SCOPE?

A check was created to ensure that duplicate identifiers had not been declared inside the same scope. This task purely involved the use of the Symbol Table. Inside the symbol table class, a function was created called getDupsInScopes().

The function is called which produces a key for every scope in the symbol table. For every scope a LinkedList is returned with all the declarations inside that scope. The Linked List is sorted and the first element is popped off. This function returns a hashtable and adds all the duplicates in a particular scope to an ArrayList. If there are duplicates in the scope. The identifier and the scope is printed out to console. Here is the code that produces this hashtable:



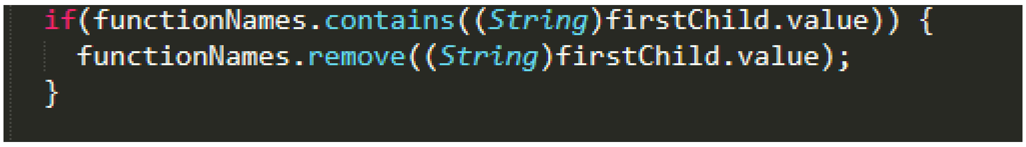
```java
public Hashtable<String,  ArrayList<String>> getDupsInScopes(){
    Set<String>keys = symbolTable.keySet();
    Hashtable<String, ArrayList<String>> dupsTable = new Hashtable<String, ArrayList<String>>();

    for(String key : keys) {
        LinkedList<String> tmpList = symbolTable.get(key);
        ArrayList<String> dups = new ArrayList<String>();
        while(0 < tmpList.size() -1){
            Collections.sort(tmpList);
            if (tmpList.size() > 0) {
                String checker = tmpList.pop();
                if(tmpList.contains(checker)){
                    dups.add(checker);
                }
            }
        }
        dupsTable.put(key, dups);
    }

    return dupsTable;
}
```

EVERY FUNCTION HAS BEEN CALLED

This check is run immediately after the visitor class executes. It checks to see that if every declared function in the program has been called. If a function called "multiply" is created then it may be called as so "result := multiply(arg_1, arg_2)".

To complete this semantic check a function was again created in the symbol table to return an ArrayList of type string of all the functions that have been declared in the program.

This function is called at beginning of the visitor class and the functions are passed into a variable called "functionNames". Upon entering the visit function for the ASTFunctionCall the function name that is being called is removed from the functionNames list like so:



```
if(functionNames.contains((String)firstChild.value)) {
  functionNames.remove((String)firstChild.value);
}
```

At the end of the visitor execution the size of the "functionNames" ArrayList is checked and if it is greater than zero each function that has not been called is printed.

ARE THE ARGUMENTS OF AN ARITHMETIC OPERATOR THE INTEGER VARIABLES OR INTEGER CONSTANTS?

The next check is for the operands of arithmetic operators. Inside the add and minus operation visit functions, the data types of the two children are checked if one of them does not equal to Num, then it does not pass, and an error is displayed and an unknown_type is returned. If they both are of type Num, the check passes.

ARE THE ARGUMENTS OF A BOOLEAN OPERATOR BOOLEAN VARIABLES OR BOOLEAN CONSTANTS?

A Boolean check also exists to ensure that no Boolean comparison operation uses the wrong type. This code was implemented in the visitor class and did not require the use of the symbol table and checks that both children are of type num. If they are not off the correct type an error message is displayed.

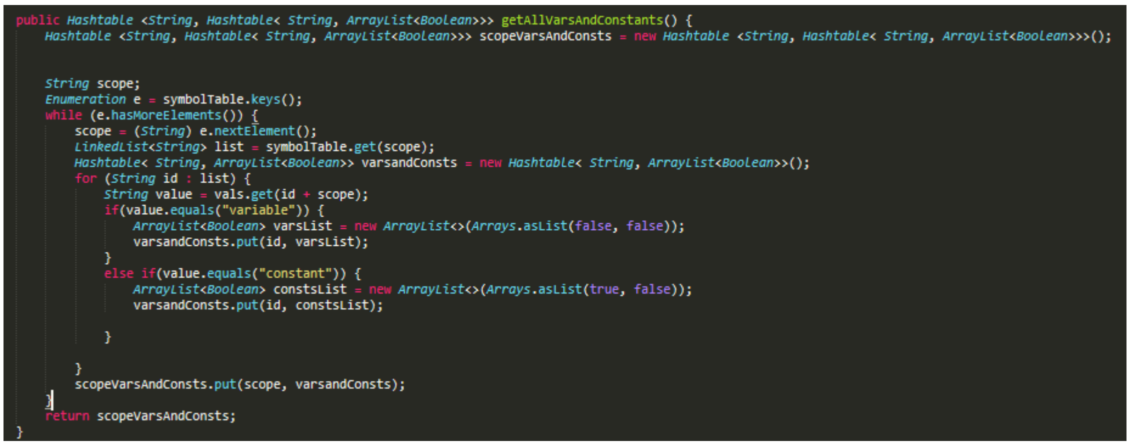
The check for this was implemented in every comparison op and so a function was created to perform the check rather than writing the same code for each comparison operator.

EVERY IDENTIFIER DECLARED WITHIN SCOPE BEFORE IT IS USED

The next check is to check if an identifier has been declared in the scope before it has been used. This is checked in the visit function for the ASTAssignment node. If the variable was of "type_unknown" when being processed by the Semantic Visitor then an error message would be printed stating that the variable had not been declared inside the scope. What is essential here is to check if the variable has not also been declared in "global" scope. For example, I believe it is valid that a variable can be declared in "global" scope and then given a value in "main". That example should pass this semantic check.

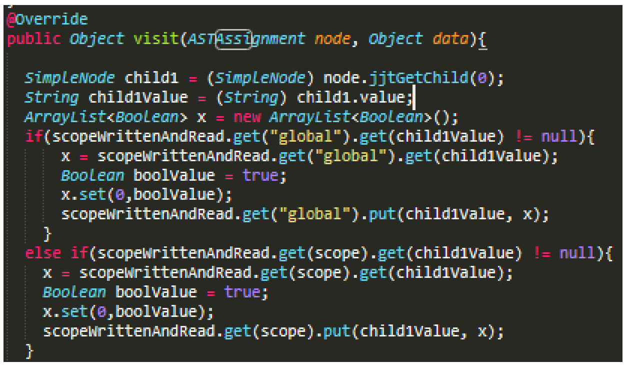
EVERY VARIABLE WRITTEN TO AND READ FROM

This was perhaps the most difficult semantic check to implement from the suggested semantic checks provide by the assignment description. For this semantic check I created a Hashtable that had a string key for the scope and the value was another hashtable. This second hashtable's keys were Id's of variables and its value was a Boolean ArrayList that has two values, true or false for written to and true or false for read from. This hashtable is initialised in the SymbolTable class like so:



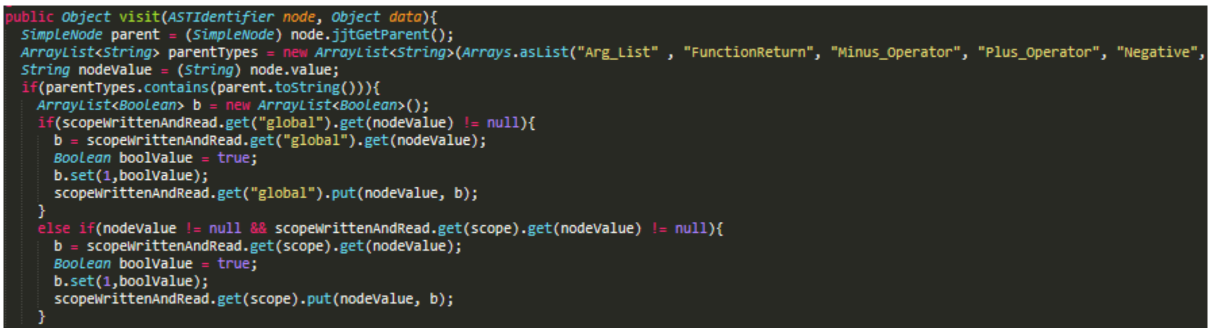
```java
public Hashtable <String, Hashtable< String, ArrayList<Boolean>>> getAllVarsAndConstants() {
    Hashtable <String, Hashtable< String, ArrayList<Boolean>>> scopeVarsAndConsts = new Hashtable <String, Hashtable< String, ArrayList<Boolean>>>();


    String scope;
    Enumeration e = symbolTable.keys();
    while (e.hasMoreElements()) {
        scope = (String) e.nextElement();
        LinkedList<String> list = symbolTable.get(scope);
        Hashtable< String, ArrayList<Boolean>> varsandConsts = new Hashtable< String, ArrayList<Boolean>>();
        for (String id : list) {
            String value = vals.get(id + scope);
            if(value.equals("variable")) {
                ArrayList<Boolean> varsList = new ArrayList<>(Arrays.asList(false, false));
                varsandConsts.put(id, varsList);
            }
            else if(value.equals("constant")) {
                ArrayList<Boolean> constsList = new ArrayList<>(Arrays.asList(true, false));
                varsandConsts.put(id, constsList);

            }

        }
        scopeVarsAndConsts.put(scope, varsandConsts);
    }
    return scopeVarsAndConsts;
}
```

As we meet variables that occur in assignments, if the variable occurs on the left hand side of an assignment, it is written to, whilst if it occurs on the right hand side of an assignment it is read from. It was also important to be able to handle global variables which can be written to and read from in different scopes. When this occurs, this symbol table is updated like so:



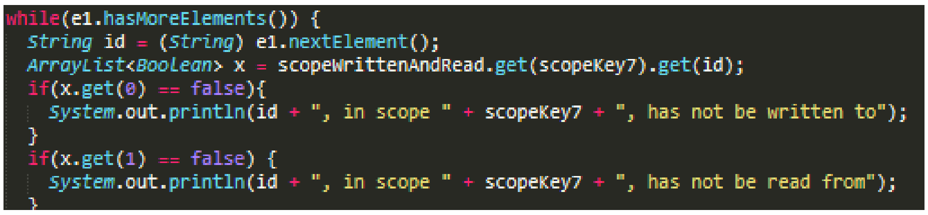
```java
@Override
public Object visit(ASTAssignment node, Object data){

  SimpleNode child1 = (SimpleNode) node.jjtGetChild(0);
  String child1Value = (String) child1.value;
  ArrayList<Boolean> x = new ArrayList<Boolean>();
  if(scopeWrittenAndRead.get("global").get(child1Value) != null){
      x = scopeWrittenAndRead.get("global").get(child1Value);
      Boolean boolValue = true;
      x.set(0,boolValue);
      scopeWrittenAndRead.get("global").put(child1Value, x);
    }
  else if(scopeWrittenAndRead.get(scope).get(child1Value) != null){
    x = scopeWrittenAndRead.get(scope).get(child1Value);
    Boolean boolValue = true;
    x.set(0,boolValue);
    scopeWrittenAndRead.get(scope).put(child1Value, x);
  }
```

As a variable is read from if it is in a condition or if it is passed in as an argument to a function, it was imperative to make sure the table was updated when this occurred also.

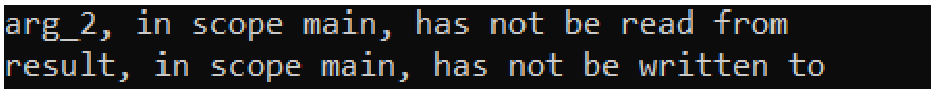


```java
public Object visit(ASTIdentifier node, Object data){
  SimpleNode parent = (SimpleNode) node.jjtGetParent();
  ArrayList<String> parentTypes = new ArrayList<String>(Arrays.asList("Arg_List" , "FunctionReturn", "Minus_Operator", "Plus_Operator", "Negative",
  String nodeValue = (String) node.value;
  if(parentTypes.contains(parent.toString())){
    ArrayList<Boolean> b = new ArrayList<Boolean>();
    if(scopeWrittenAndRead.get("global").get(nodeValue) != null){
      b = scopeWrittenAndRead.get("global").get(nodeValue);
      Boolean boolValue = true;
      b.set(1,boolValue);
      scopeWrittenAndRead.get("global").put(nodeValue, b);
    }
    else if(nodeValue != null && scopeWrittenAndRead.get(scope).get(nodeValue) != null){
      b = scopeWrittenAndRead.get(scope).get(nodeValue);
      Boolean boolValue = true;
      b.set(1,boolValue);
      scopeWrittenAndRead.get(scope).put(nodeValue, b);
    }
```

When the program is finished executing, the hashtable is checked to see if every variable is written to and read from. An output like so is displayed if a variable does not fully satisfy these conditions:

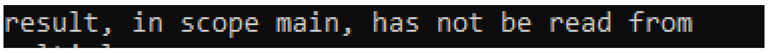


```
while(e1.hasMoreElements()) {
  String id = (String) e1.nextElement();
  ArrayList<Boolean> x = scopeWrittenAndRead.get(scopeKey7).get(id);
  if(x.get(0) == false){
    System.out.println(id + ", in scope " + scopeKey7 + ", has not be written to");
  }
  if(x.get(1) == false) {
    System.out.println(id + ", in scope " + scopeKey7 + ", has not be read from");
```



```
arg_2, in scope main, has not be read from
result, in scope main, has not be written to
```

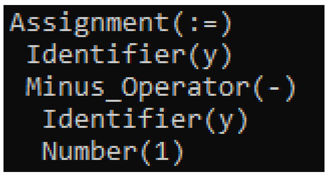
This semantic check even has an issue with the final variable “result” that is written to in the multiply example, as after result is written to in main, it is never read from and so the following error is printed:



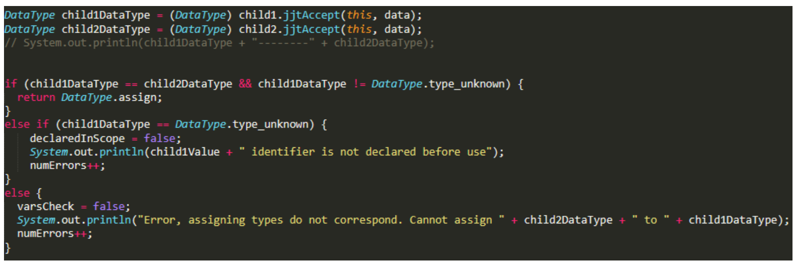
```
result, in scope main, has not be read from
```

IS THE LEFT-HAND SIDE OF AN ASSIGNMENT VARIABLE OF THE CORRECT TYPE?

This was a check that was made easy by the way I had created my Abstract Syntax Tree (AST). Although and Identifier is met first in an assignment, in my jjt file I force the Identifier to be a child of the Assignment Node. As a result, both the left hand side and the right-hand side of the assignment are a child of the Assignment Node. The first child will always be an identifier whilst the second child could be one of a Minus or Plus Operator or a function call. This means I can easily get their datatype using the created DataType class. If these two datatypes do not match the error is displayed. The layout of my AST for an Assignment is shown below as well as the simple check to either side of the assignment:



```
Assignment(:=)
 Identifier(y)
 Minus_Operator(-)
  Identifier(y)
  Number(1)
```



```
DataType child1DataType = (DataType) child1.jjtAccept(this, data);
DataType child2DataType = (DataType) child2.jjtAccept(this, data);
// System.out.println(child1DataType + "--------" + child2DataType);


if (child1DataType == child2DataType && child1DataType != DataType.type_unknown) {
  return DataType.assign;
}
else if (child1DataType == DataType.type_unknown) {
    declaredInScope = false;
    System.out.println(child1Value + " identifier is not declared before use");
    numErrors++;
}
else {
  varsCheck = false;
  System.out.println("Error, assigning types do not correspond. Cannot assign " + child2DataType + " to " + child1DataType);
  numErrors++;
}
```

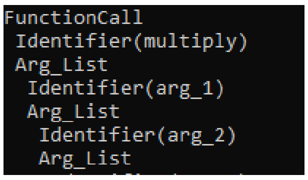
Of course, when declaring a constant both sides of the assignment had to be checked to make sure that the types of the two sides matched. This check is performed in the visit function for the Constant_Declaration Node.

IS THERE A FUNCTION FOR EVERY INVOKED IDENTIFIER?

If this semantic check does not pass, a message like so is printed: “Error, function: multiply does not exist.” When this error message is printed to console it means a function has been called that does not exist. A function can be called in an assignment or it can be called on its own. For this check the name of the invoked function is passed into a getType() function declared in the symbol table. As functions are declared in global scope, we basically check for the type of the function in the symbol table and if there is no type returned then the function does not exist. The getType() is a function that is used in multiple places throughout the code and was a very useful function which handles multiple of the semantic checks.

DOES EVERY FUNCTION CALL HAVE THE CORRECT NUMBER OF ARGUMENTS?

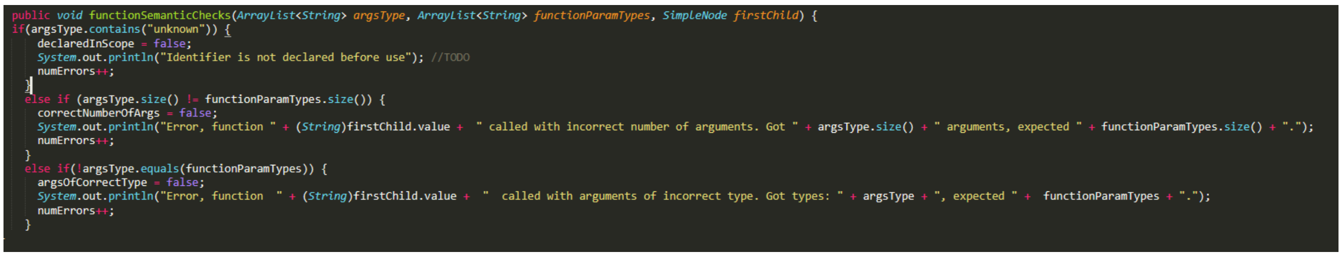
This semantic check inspired me to create some of my own extra semantic checks which will be discussed in further detail later on in the report. Performing this semantic check required being able to access all the arguments passed into an invoke function as well as being able to get the number of parameters that are passed in when the function is declared. Getting the arguments from the function call was a quite simple due to the way my Syntax Tree was set up. Here is the output of my Syntax Tree for a function call:



```
FunctionCall
 Identifier(multiply)
 Arg_List
  Identifier(arg_1)
  Arg_List
   Identifier(arg_2)
   Arg_List
```

As you can see an Arg_List node has two children an Identifier and an Arg_List. This means I could use a recursive function to retrieve the number of arguments and their type. As I was able to retrieve their type as well as the number of arguments, this meant I could create some extra semantic checks. As mentioned I had to be able to see how many parameters are passed into the function. To do this another function was written in the symbol table which returned the number and type of the parameters used in the function. As I now had these two values I could compare them to check if they matched. If they didn't, an error would be printed.

I could now check that the function call had the correct number or arguments and an extra check that was not mentioned in the assignment specification. I could now check if the **function is called with arguments of the correct type** as well as checking if it had the correct number of arguments.



```
public void functionSemanticChecks(ArrayList<String> argsType, ArrayList<String> functionParamTypes, SimpleNode firstChild) {
if(argsType.contains("unknown")) {
    declaredInScope = false;
    System.out.println("Identifier is not declared before use"); //TODO
    numErrors++;
  }
  else if (argsType.size() != functionParamTypes.size()) {
    correctNumberOfArgs = false;
    System.out.println("Error, function " + (String)firstChild.value +  " called with incorrect number of arguments. Got " + argsType.size() + " arguments, expected " + functionParamTypes.size() + ".");
    numErrors++;
  }
  else if(!argsType.equals(functionParamTypes)) {
    argsOfCorrectType = false;
    System.out.println("Error, function  " + (String)firstChild.value +  "  called with arguments of incorrect type. Got types: " + argsType + ", expected " +  functionParamTypes + ".");
    numErrors++;
  }
```

EXTRA SEMANTIC CHECKS

As well as the aforementioned check highlighted in bold above, I created some extra semantic checks which I will briefly discuss.

FUNCTION TYPE MATCHES RETURN TYPE

This function involved getting evaluating the children nodes of the Function Node and comparing them to see if they matched. The Function Node has 6 children nodes. The first child is the Type of the function Integer, Boolean, Void etc. The 6th child is a FunctionReturn which has a child stating the return Type. Effectively the Type child of the function had to be compared to the 6th child's (FunctionReturn Node) first child to perform this this semantic check.

The next two semantic checks follow on from this check.

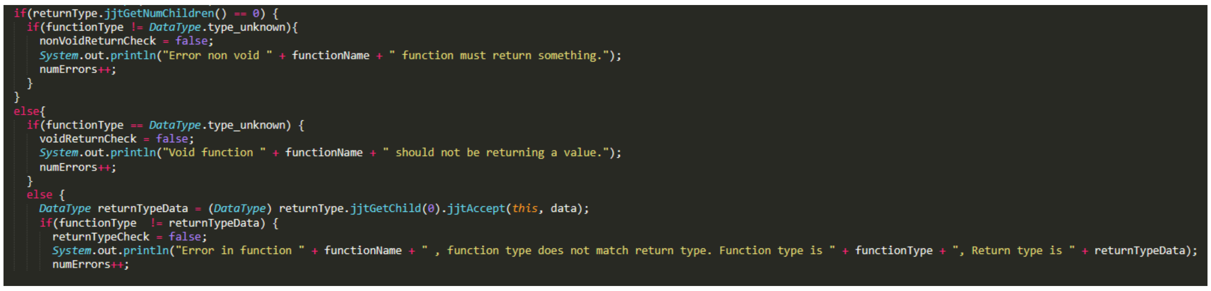
CHECK IF NON-VOID FUNCTION MAKES A RETURN

This semantic check makes sure that no void function makes a return. It checks if the return type of the function and what is returned both equal “type_unknown” which is a DataType in the DataType class.

CHECK IF VOID FUNCTION MAKES A RETURN

This check ensures that if there is something being returned by the function, then the function should not be of type void.

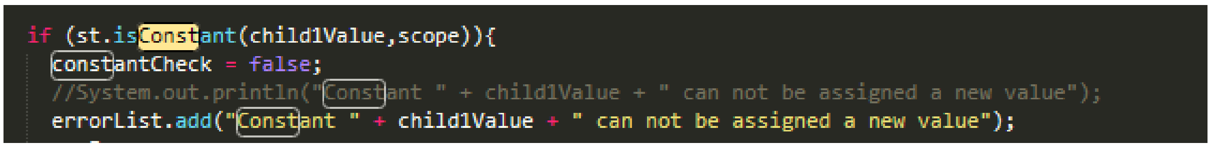
Here all the three semantic checks that are being performed:



```
if(returnType.jjtGetNumChildren() == 0) {
  if(functionType != DataType.type_unknown){
    nonVoidReturnCheck = false;
    System.out.println("Error non void " + functionName + " function must return something.");
    numErrors++;
  }
}
else{
  if(functionType == DataType.type_unknown) {
    voidReturnCheck = false;
    System.out.println("Void function " + functionName + " should not be returning a value.");
    numErrors++;
  }
  else {
    DataType returnTypeData = (DataType) returnType.jjtGetChild(0).jjtAccept(this, data);
    if(functionType  != returnTypeData) {
      returnTypeCheck = false;
      System.out.println("Error in function " + functionName + " , function type does not match return type. Function type is " + functionType + ", Return type is " + returnTypeData);
      numErrors++;
```

CONSTANT CANNOT BE REASSIGNED A VALUE

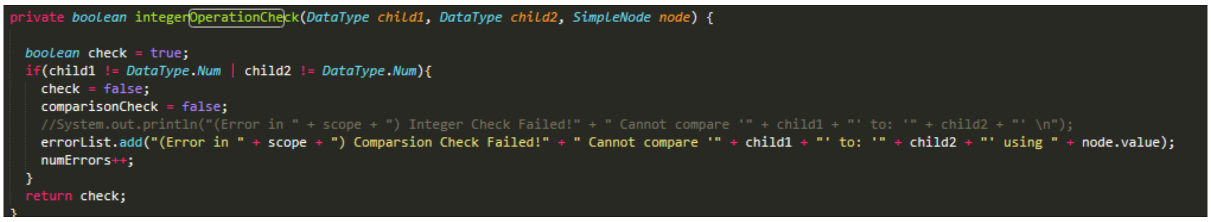
This check makes sure that a constant cannot be reassigned a new value. When an Assignment occurs a function called isConstant() is called on the left hand side of the assignment. This function checks the symbol table to see if the identifier is a constant and if it returns True an error is generated.



```
if (st.isConstant(child1Value,scope)){
  constantCheck = false;
  //System.out.println("Constant " + child1Value + " can not be assigned a new value");
  errorList.add("Constant " + child1Value + " can not be assigned a new value");
```

COMPARISON OPERATOR CHECK

Another semantic check I created makes sure that only Numbers can be compared using the comparison operators. This check means it is not possible to have a condition like so “true > true” but it is possible to have condition like so “true = true”.

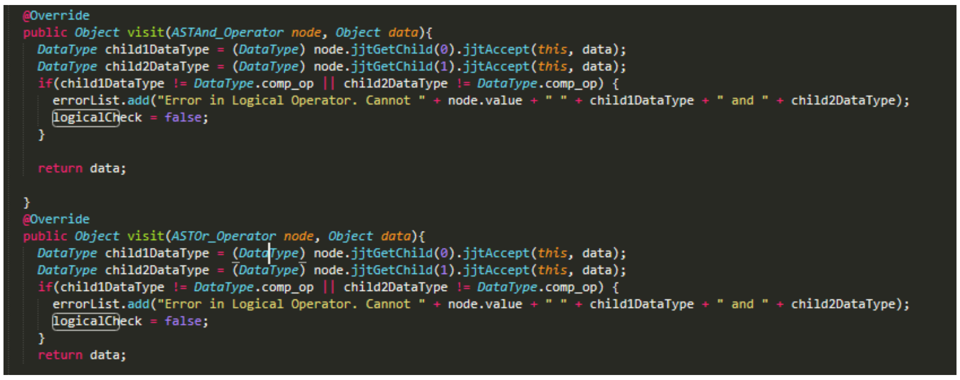


```
private boolean integerOperationCheck(DataType child1, DataType child2, SimpleNode node) {

  boolean check = true;
  if(child1 != DataType.Num | child2 != DataType.Num){
    check = false;
    comparisonCheck = false;
    //System.out.println("(Error in " + scope + ") Integer Check Failed!" + " Cannot compare '" + child1 + "' to: '" + child2 + "' \n");
    errorList.add("(Error in " + scope + ") Comparsion Check Failed!" + " Cannot compare '" + child1 + "' to: '" + child2 + "' using " + node.value);
    numErrors++;
  }
  return check;
}
```

This check is called in the relevant nodes.

LOGICAL OPERATOR CHECK

This check is similar to the one above and checks if valid conditions are found either side of a logical operator. This is useful as it ensures all logically combined conditions are valid. Here is the code implemented in both visit functions for the ASTAnd_Operator (&) node and the ASTOr_Operator (||) node to perform this semantic check:



```
@Override
public Object visit(ASTAnd_Operator node, Object data){
  DataType child1DataType = (DataType) node.jjtGetChild(0).jjtAccept(this, data);
  DataType child2DataType = (DataType) node.jjtGetChild(1).jjtAccept(this, data);
  if(child1DataType != DataType.comp_op || child2DataType != DataType.comp_op) {
    errorList.add("Error in Logical Operator. Cannot " + node.value + " " + child1DataType + " and " + child2DataType);
    logicalCheck = false;
  }

  return data;

}
@Override
public Object visit(ASTOr_Operator node, Object data){
  DataType child1DataType = (DataType) node.jjtGetChild(0).jjtAccept(this, data);
  DataType child2DataType = (DataType) node.jjtGetChild(1).jjtAccept(this, data);
  if(child1DataType != DataType.comp_op || child2DataType != DataType.comp_op) {
    errorList.add("Error in Logical Operator. Cannot " + node.value + " " + child1DataType + " and " + child2DataType);
    logicalCheck = false;
  }
  return data;
```

Overall, I implemented 16 semantic checks. Implementing them, whilst tough, was an enjoyable task as it really made you think of different errors that can potentially occur when writing code in a particular language.

THREE ADDRESS CODE VISITOR

This was perhaps the most enjoyable part of the assignment. Effectively, this part of the assignment involved printing the code passed into to the parser in three address code. Three address code is a type of intermediate code which is easy to generate and can be easily converted to machine code. It makes use of at most three addresses and one operator to represent an expression and the value computed at each instruction is stored in temporary variable generated by compiler.

The general representation for three address code is:

```
x = y op z
```

Consider this example from the course notes written normally and then in three address code:

a + a ∗ (b − c) + (b − c) ∗ d

t1 = b − c

t2 = a ∗ t1

t3 = a + t2

t4 = d ∗ t1

t5 = t3 + t4

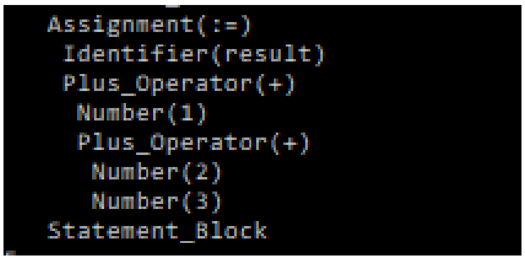
Providing this type of representation for expressions was the goal for this part of the assignment. I found this exercise beneficial as it really improved my knowledge of three address code and its benefits.

To generate this code we had to create a class very similar to that of the semantic visitor class. The class implemented the same interface and thus it had the same methods. The main difference between that of the semantic visitor class and the three address code visitor was that in the semantic visitor class, data types were returned whilst in the three address code visitor class the relevant information was printed in three address code format and then null was returned.

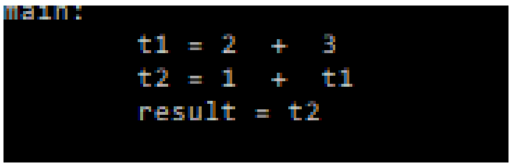
Provided with the assignment was a jar file which interpreted a version of Three Address code named TACi. I followed the layout of the TACi language when creating my three address code.

CREATING TEMPORARY VARIABLES

A function was created to handle the creation of temporary values. This function is called in all operation where variables, numbers, Booleans etc. are used. As Three Address Code can only handle at most two values and an operator, this function must be able to handle logically combined conditions like "a > 0 & b < 1" or a statement like so "result = 1 + 2 + 3". As my AST implemented precedence and as a result, was recursive, implementing a solution to handle the previous two statements was quite simple. Here is the output for my AST and Three Address code for the example "result" mentioned above:

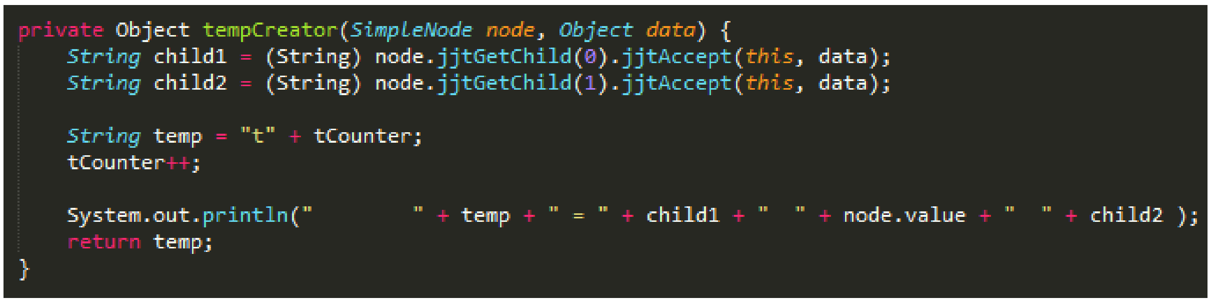


```
 Assignment(:=)
  Identifier(result)
  Plus_Operator(+)
   Number(1)
   Plus_Operator(+)
    Number(2)
    Number(3)
 Statement_Block
```



```
main:
        t1 = 2  +  3
        t2 = 1  +  t1
        result = t2
```

As you can see the "2 + 3" is evaluated first and added to a t1. Then the remaining 1 is added to t1 and this is stored in t2. Finally "result" is set equal to t2 which is useful as it makes it easy to track the identifiers. The code below handles what the Three Address code being produced above and you can see it is a very simple function simply because of how my AST is implemented.

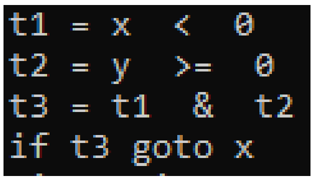


```java
private Object tempCreator(SimpleNode node, Object data) {
    String child1 = (String) node.jjtGetChild(0).jjtAccept(this, data);
    String child2 = (String) node.jjtGetChild(1).jjtAccept(this, data);

    String temp = "t" + tCounter;
    tCounter++;

    System.out.println("        " + temp + " = " + child1 + "  " + node.value + "  " + child2 );
    return temp;
}
```

CONDITIONS

Conditions handled in three address code were important to get right as they could easily be logically combined using AND or OR with another condition. If two conditions are used together, the two conditions are stored in temporary variables and then in a final third temporary the logical operator is put between them. For the first if statement in the multiply function provided in the assignment specification, this is how it appears in my three address code.

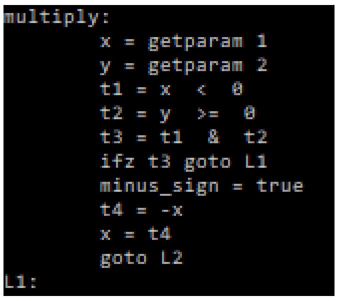
The condition is x < 0 & y >= 0



```
t1 = x  <  0
t2 = y  >=  0
t3 = t1  &  t2
if t3 goto x
```

FUNCTIONS

When a function node is encountered the function name is printed. We proceed to print out the body of the function which are all tabbed in under the function name. Also similar to the TACI language, when a function is declared with parameters these parameters are passed into the program via "getparam" statements. If there are two parameters named x & y belonging to the function the first two statements inside the function will be "x = getparam 1", "y = getparam 2". This is coincided by how functions are invoked in that the arguments are pushed onto a stack before the function is called and this is discussed in further detail below. Here is the start of the multiply function in three address code.

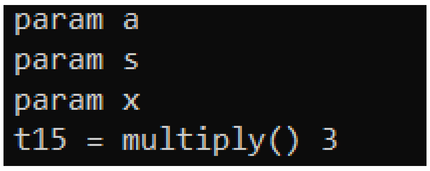


```
multiply:
        x = getparam 1
        y = getparam 2
        t1 = x  <  0
        t2 = y  >=  0
        t3 = t1  &  t2
        ifz t3 goto L1
        minus_sign = true
        t4 = -x
        x = t4
        goto L2
L1:
```

As you can see the two parameters passed into multiply are initialised at the top of the function before the rest of the body executes.

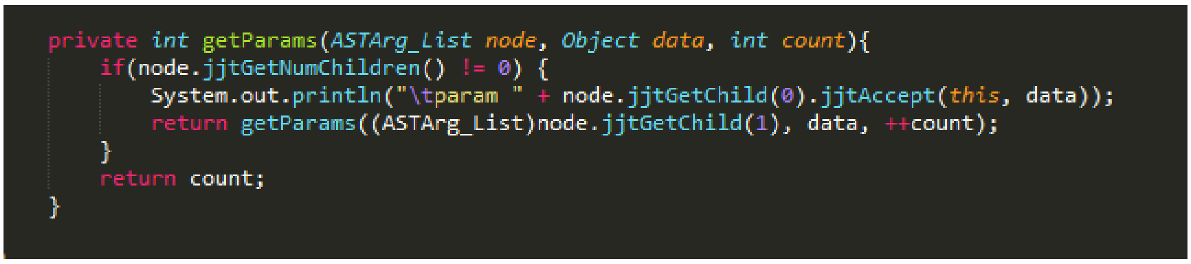
FUNCTION CALLS

When a function is invoked with arguments, I try to follow the layout of the TACI language as much as possible. As a result the arguments are declared as "params" before the function is invoked. When the function is then called it is called with the number of arguments and these are subsequently popped off the stack and passed into the program. If a multiple function is called with three arguments called "a", "s" and "x" this would be the output from my three address code visitor.



```
param a
param s
param x
t15 = multiply() 3
```

Here is the code to generate the “param” code:

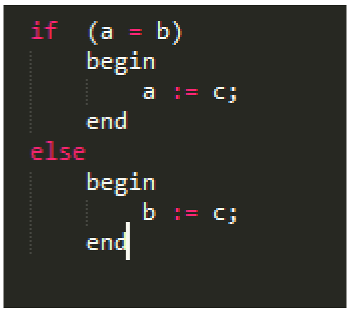


```
private int getParams(ASTArg_List node, Object data, int count){
    if(node.jjtGetNumChildren() != 0) {
        System.out.println("\tparam " + node.jjtGetChild(0).jjtAccept(this, data));
        return getParams((ASTArg_List)node.jjtGetChild(1), data, ++count);
    }
    return count;
}
```

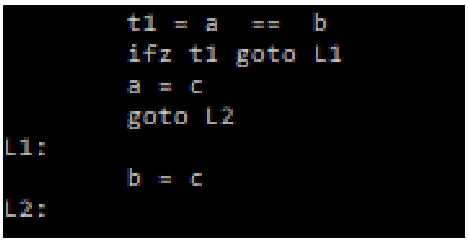
IF ELSE STATEMENTS

When the three address code generator handled if statement it used the following syntax “ifz tn goto Ln”. Effectively this describes “if false goto else statement”. This was the method employed in the TACi language and felt the most logical way to handle an if-else statements. Here is how an if-else statement is handled

Here is the input, and the Three Address code:



```
if  (a = b)
    begin
        a := c;
    end
else
    begin
        b := c;
    end
```



```
        t1 = a  ==  b
        ifz t1 goto L1
        a = c
        goto L2
L1:
        b = c
L2:
```

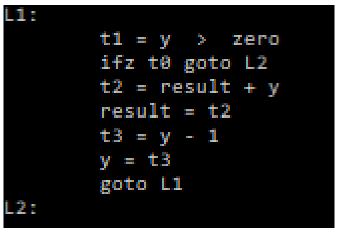
From this you can see the condition being added to a temporary variable. If it is false you move to L1 which is the else condition, otherwise you just execute the block of code and jump to L2.

WHILE STATEMENTS

The while statements are written in the correct format for three address code. The flow of control returns to the same block if a condition has not been satisfied and when it is it leaves the loop. Here is the output for the following while loop: while (y > 0)

result = result + y

y = y – 1

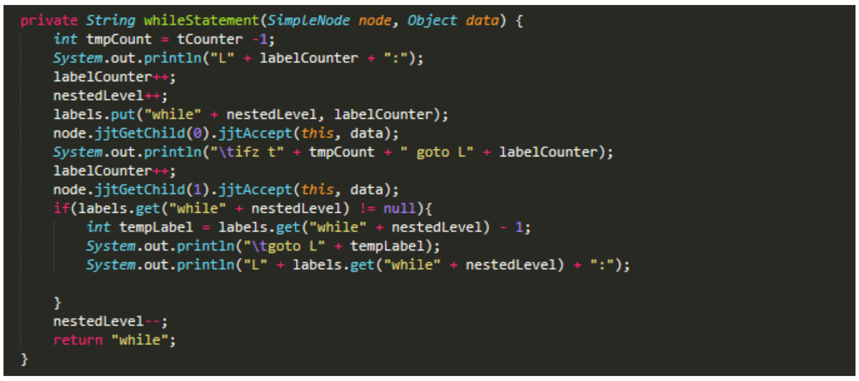


```
L1:
        t1 = y  >  zero
        ifz t0 goto L2
        t2 = result + y
        result = t2
        t3 = y - 1
        y = t3
        goto L1
L2:
```

From this snippet you can see how the code returns to L1, the start of the while whilst y > 0. Once this condition is not satisfied, the code moves to L2.

My three address code allows for nested while loops. This is done by storing how nested a while loop is in a hashtable. This was quite tricky to implement but worthwhile as all the labels for blocks of code are in the correct position. Allowing for nested while loops and if statements (discussed below) really improves the standard of the three address code being generated.

Here is the code to handle the nested while loops:



```java
private String whileStatement(SimpleNode node, Object data) {
    int tmpCount = tCounter -1;
    System.out.println("L" + labelCounter + ":");
    labelCounter++;
    nestedLevel++;
    labels.put("while" + nestedLevel, labelCounter);
    node.jjtGetChild(0).jjtAccept(this, data);
    System.out.println("\tifz t" + tmpCount + " goto L" + labelCounter);
    labelCounter++;
    node.jjtGetChild(1).jjtAccept(this, data);
    if(labels.get("while" + nestedLevel) != null){
        int tempLabel = labels.get("while" + nestedLevel) - 1;
        System.out.println("\tgoto L" + tempLabel);
        System.out.println("L" + labels.get("while" + nestedLevel) + ":");

    }
    nestedLevel--;
    return "while";
}
```

HANDLING IF-ELSE STATEMENTS AND WHILE STATEMENTS WITH LABELS

This is the aspect of the Three address code that I am most proud of. I have instantiated the correct labels for the correct blocks of code. Doing this required a hashtable which allowed me track how nested an if or else statement was. How effective this is requires the use of an example which will be shown below. Some of the labels will appear to not have any code when using if statements, but this is the correct behaviour and not a defect of the code. Empty labels may show that if an if passes, no changes may be made to the code and you move to the next block.

This was by far the most challenging aspect of creating the three address code as keeping tracking of what the "goto" commands was quite tricky. This was eventually solved using a hashtable which was mentioned above.

OPTIMISATION OF THREE ADDRESS CODE

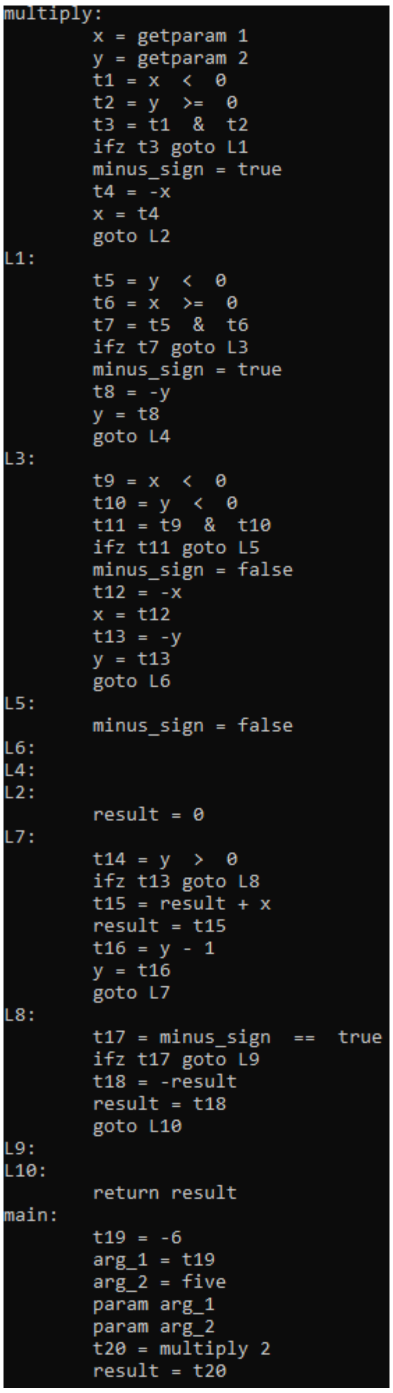
When I initially wrote my Three Address code generator, I created a temporary variable for every assignment apart from a simple statement like "a = b". This meant there were many unnecessary temporary variables being created. A statement such as "result = 1 + 2" would be printed like so:

t1 = 1 + 2

result = t1

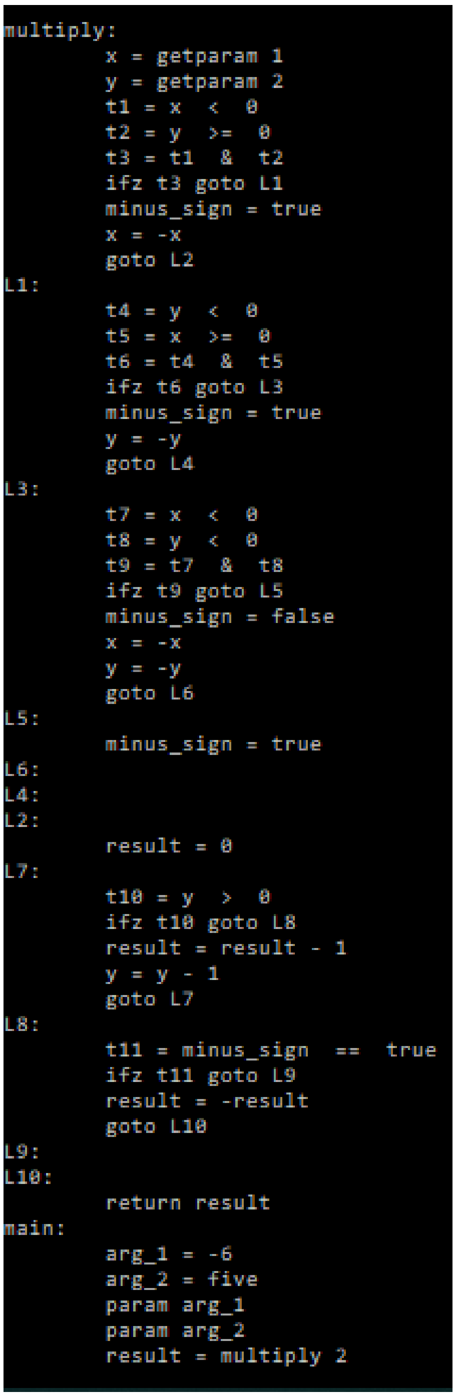
This is an inefficient use of temporary variables as “result = 1 + 2” is an acceptable statement in Three Address Code. My aim was to eliminate these inefficiencies and in turn reduce the number of temporaries being created. After implementing some optimisation in my Three Address Code here is the before and after of the output for the multiply example:

Before:



```
multiply:
        x = getparam 1
        y = getparam 2
        t1 = x  <  0
        t2 = y  >=  0
        t3 = t1  &  t2
        ifz t3 goto L1
        minus_sign = true
        t4 = -x
        x = t4
        goto L2
L1:
        t5 = y  <  0
        t6 = x  >=  0
        t7 = t5  &  t6
        ifz t7 goto L3
        minus_sign = true
        t8 = -y
        y = t8
        goto L4
L3:
        t9 = x  <  0
        t10 = y  <  0
        t11 = t9  &  t10
        ifz t11 goto L5
        minus_sign = false
        t12 = -x
        x = t12
        t13 = -y
        y = t13
        goto L6
L5:
        minus_sign = false
L6:
L4:
L2:
        result = 0
L7:
        t14 = y  >  0
        ifz t13 goto L8
        t15 = result + x
        result = t15
        t16 = y - 1
        y = t16
        goto L7
L8:
        t17 = minus_sign  ==  true
        ifz t17 goto L9
        t18 = -result
        result = t18
        goto L10
L9:
L10:
        return result
main:
        t19 = -6
        arg_1 = t19
        arg_2 = five
        param arg_1
        param arg_2
        t20 = multiply 2
        result = t20
```

After:



```
multiply:
        x = getparam 1
        y = getparam 2
        t1 = x  <  0
        t2 = y  >=  0
        t3 = t1  &  t2
        ifz t3 goto L1
        minus_sign = true
        x = -x
        goto L2
L1:
        t4 = y  <  0
        t5 = x  >=  0
        t6 = t4  &  t5
        ifz t6 goto L3
        minus_sign = true
        y = -y
        goto L4
L3:
        t7 = x  <  0
        t8 = y  <  0
        t9 = t7  &  t8
        ifz t9 goto L5
        minus_sign = false
        x = -x
        y = -y
        goto L6
L5:
        minus_sign = true
L6:
L4:
L2:
        result = 0
L7:
        t10 = y  >  0
        ifz t10 goto L8
        result = result - 1
        y = y - 1
        goto L7
L8:
        t11 = minus_sign  ==  true
        ifz t11 goto L9
        result = -result
        goto L10
L9:
L10:
        return result
main:
        arg_1 = -6
        arg_2 = five
        param arg_1
        param arg_2
        result = multiply 2
```

As you can see there is a clear reduction in the number of temporaries being used. Following the TACi layout for the Three Address Code was really helpful and enabled me to produce what I believe to be really clean Three Address Code.

CONCLUSION/THOUGHTS ON THE ASSIGNMENT

I thoroughly enjoyed this assignment and thought it was a great challenge. The integration of the two assignments was really important as it felt as if you were really building something substantial. Also, as the second assignment progressed, it opened my eyes to important changes that you make to your first assignment in order to generate an Abstract Syntax Tree. After the second assignment especially, I have acquired a far greater appreciation of the complexities that lie behind compiler construction.

The toughest aspect of this assignment was creating and performing the semantic checks. Although, seeing them work was quite rewarding, some of them were quite tedious to get perfect. I feel I have made a really good attempt at this assignment and I tried to implement all that was asked of me to the best of my ability.

Notable are my efforts to create some extra semantic checks and also being able to handle nested if and while statements when creating the three address code.

If there was more time allotted for this assignment my next step would be to optimise the code further in both the Semantic Visitor class and the Three Address Code Visitor class. Optimisation and efficiency are such important factors of a compiler and I'm sure implementing them would be quite interesting.

EXTERNAL RESOURCES

https://www.geeksforgeeks.org/three-address-code-compiler/