Submission Liam McAweeney 14415152

Programme: Computer Applications Module Code:

CA4003 Assignment Title: Semantic Analysis and Intermediate Representation.

Submission Date: 17/12/2018

Module Coordinator: David Sinclair

Name(s): Liam McAweeney Date: 17/12/2018

CA4003 - Compiler Construction Assignment 2

Student Name: Liam McAweeney
Student Number: 14415152
Submission Date: 14/12/18

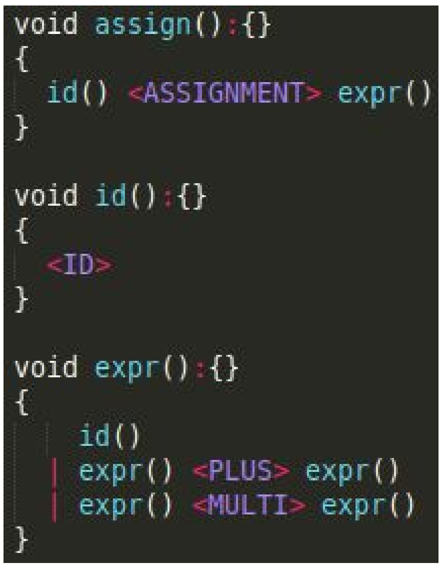
Introduction

The assignment is a continuation of assignment 1 where I created a lexical and syntax analyser for the defined language. This document is split into four main sections.

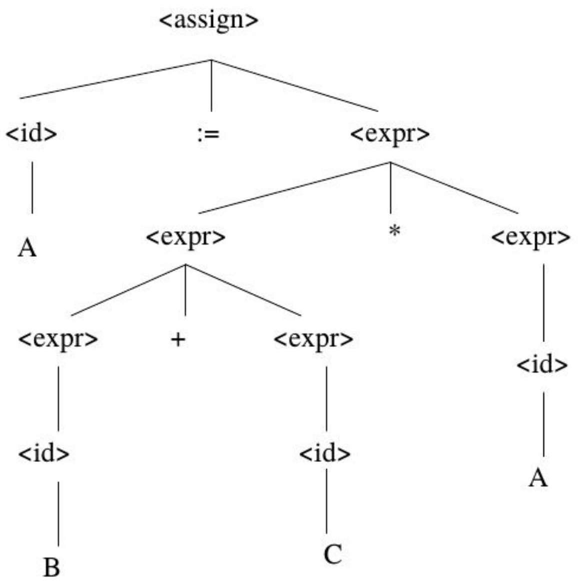
Abstract Syntax Tree

An abstract syntax tree is a clean interface between parsing and semantic analysis (and other later phases of a compiler). The abstract syntax tree captures the phrase structure of the input.

The reason I am creating a Abstract Syntax Tree (AST) is to make implementation of the semantic checks and generation of the Three Address Code easier. The AST allows for a clear path to be to be produced when iterating over production rules. Here is an example of an input: A := B + C * A

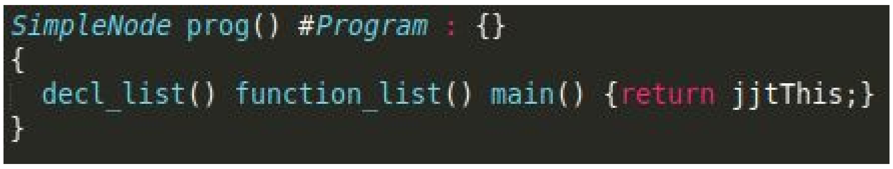


```
void assign():{}
{
  id() <ASSIGNMENT> expr()
}

void id():{}
{
  <ID>
}

void expr():{}
{
    id()
  | expr() <PLUS> expr()
  | expr() <MULTI> expr()
}
```



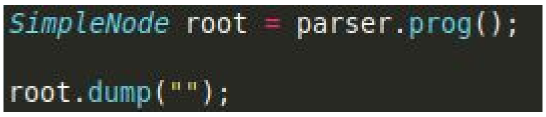
Implementation

The first task was to change the .jj file to .jjt, it was to allow the “jjtree” function to process the file. When I first ran “jjtee calParser.jjt” is created SimpleNode.java. SimpleNode is a class which implements the Node class. It allows each leaf on the tree to be connected. The next task was to change the first production rule to return type of SimpleNode and create a return to return the SimpleNode. SimpleNode generates a node, leaf on the tree, each time a production rule is entered.



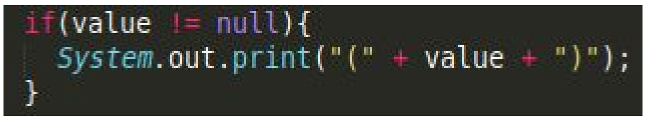
```
SimpleNode prog() #Program : {}
{
  decl_list() function_list() main() {return jjtThis;}
}
```

This rule was previously a void it was not assigned to anything, but now it has a return type of SimpleNode I must assign it to a SimpleNode, which I called root. SimpleNode has a dump function which prints out the name of each node on the AST.



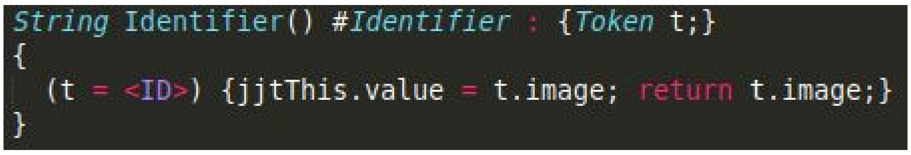
```
SimpleNode root = parser.prog();

root.dump("");
```

This function can be modified to allow the value inside a node to be printed out. The dump function can then be called to print out the AST.



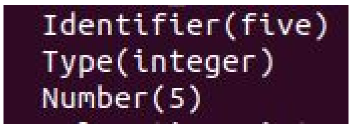
```
 if(value != null){
   System.out.print("(" + value + ")");
 }
```

When designing and implemented the first assignment, I strongly considered the structure of the production rules for design of the AST for assignment 2. Because of this I had very little changes to makes to the production rules. The main changes I had to make were to abstract the identifier and type token into separate rules. I did this to allow the image, value, of the token to be inserted into the node so it would appear in the AST.



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

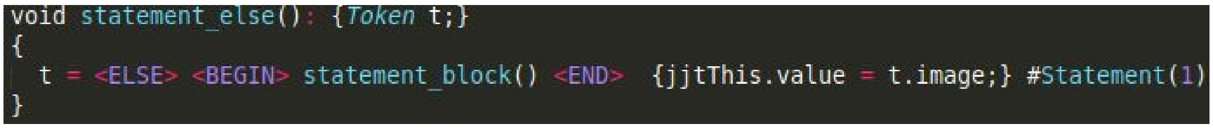
Each production rule by default creates a node on the AST. Each production rule is vital in ensuring that the input is syntactically correct. But once the syntax is correct, not all of the nodes are needed to evaluate whether the semantics of the input is correct. If these nodes were left on the tree it would create a very large tree which would become more computationally expensive for the compiler. I configured the compiler to not create a node by default. Nodes are only created when specified in the production rule. This is completed by adding a ‘#’ followed by the name of the node you want to create. I image above you can see I created a node for *Identifier* by appending “#Identifier” to the definition. In this example I also return the image of the *ID* token which allows the value to be passed into the node.



```
  Identifier(five)
  Type(integer)
  Number(5)
```

An example of node I do not need on the AST is fragment.

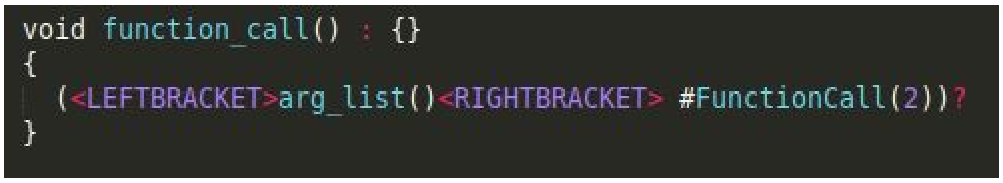
When creating a node I can specify the number of child nodes that child would have. Here is an example:



```
void statement_else(): {Token t;}
{
  t = <ELSE> <BEGIN> statement_block() <END>  {jjtThis.value = t.image;} #Statement(1)
}
```

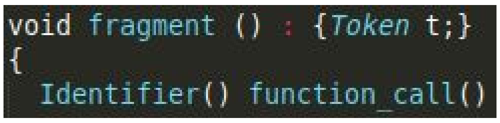
As you can see there statement will have one child which will be statemen_block. This is trivial as this is the default number of children this rule will have as it only have one non-terminal.

In cases the compiler may want force the node to have a different number of children nodes. An example of this is:



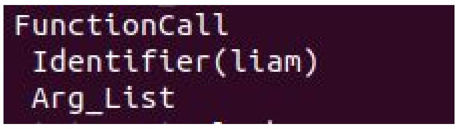
```
void function_call() : {}
{
  (<LEFTBRACKET>arg_list()<RIGHTBRACKET> #FunctionCall(2))?
}
```

As you can see this rule only contains one non terminal but the node created specifies that it needs two children. This will force this node up the tree one position in search of another non terminal as a child.



```
void fragment () : {Token t;}
{
  Identifier() function_call()
```

As you can see above function call has a sibling non terminal, this will allow function call to now have 2 children, Identifier and arg_list.



```
FunctionCall
 Identifier(liam)
 Arg_List
```

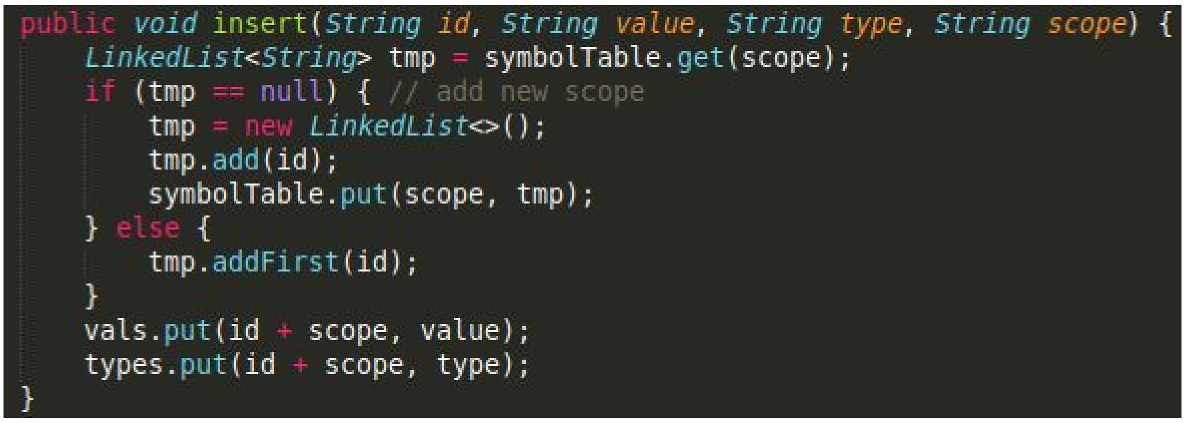
While this developing the AST I started at a low level non terminal which evaluated to a terminal. I then incrementally moved up the tree to higher level non terminals. This greatly helped in reducing time spent on debugging.

Symbol Table

A Simple Table is used to store data on variables and constants. The purpose of this is to add in the development of semantics checks.

Implementation

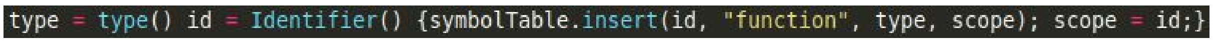
In a program variables can generally only be accessed from within their scope. Two functions may have two variables with the same id, which is legal. A variable may also be declared globally so it can be accessed from anywhere inside the program. Since a function can only be declared within the global scope, no two functions will have the same id. Because of this I used the function id as the scope. This in turn means that every scope will be unique. And every scope can have variables and their metadata. Therefore I will create a hashtable with the scope as the keys and a linkedlist of the variables. I will then create a hashtable with the scope and variable id as the key and a string as variable type. I will also create another hashtable with the same key and a string with “constant” , “variable”, “function” or “function parameter”. This will be used for the semantic checks.



```java
public void insert(String id, String value, String type, String scope) {
    LinkedList<String> tmp = symbolTable.get(scope);
    if (tmp == null) { // add new scope
        tmp = new LinkedList<>();
        tmp.add(id);
        symbolTable.put(scope, tmp);
    } else {
        tmp.addFirst(id);
    }
    vals.put(id + scope, value);
    types.put(id + scope, type);
}
```

Above is the function which is used to insert the values into the respective hashtables.

Next I set the value of the scope to global inside the parser. The scope must the be change to main once it enters the main non terminal. When a function is created the scope must be changed to the function name and allows back to global once the function body is complete.



```java
type = type() id = Identifier() {symbolTable.insert(id, "function", type, scope); scope = id;}
```

Above you can see I assign the value return from the type production rule to a variable type, and the same for id. I then insert this into these values into the symbol table and change the scope to the function id for the body of the function. The scope is then changed back to global after the function body is complete.

I create a print function to print the symbol table to the terminal for debugging. I also create other function which are used to access the symbol table from outside the class for semantic checks. I will describe these functions in further detail in the next section.

Semantic Analyser

The semantic analyser performs semantics checks. It will use the symbol table and AST tree to perform the checks. The semantic analyser will output each error in the input out to the console for the user to correct.

After running “jjtee calParser.jjt” a visitor interface is created. It will contain method signatures for each node type created. I will implement each of these methods to perform the semantic checks. Each of the implementations of the node visitors will be ran as each node is traversed. I created the “PrintVisitor” class which implements the interface. Each method created will have two parameters, the specific node class and data. Each node is implements the simpleNode class which has methods for accessing child nodes and parent nodes.

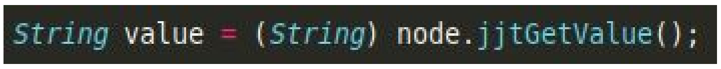
Implementation

In this section I will do through each semantic check specified in the assignment specification and also addition semantic checks which I added in.

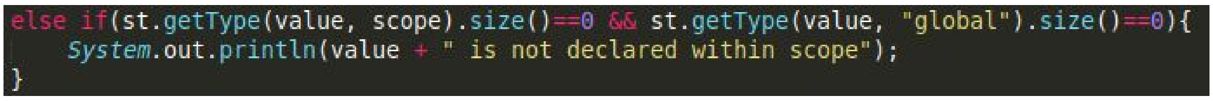
Is every identifier declared within scope before its is used?

To perform this check I two pieces of information, the current scope and the identifier’s name. This check must be done from inside the visitor method for the identifier function.

An issue here is that every time an identifier visited it make perform this check, this check is not needed to be performed if the identifier is vistored in a declaration of a function, variable or a constant. Therefore the first thing I check is it the if its parent, node previously visited is a function or variable etc. The is completed by specifying the return of the parent nodes to be the function or variable etc. To this I created a class DataType which has a list of defined types. I will then check if the identifiers parent is of function DataType or variable declaration type. After checking that the identifier’s parent is not any of the three mentioned above I check the symbol table to see if the contains an entry at the specified scope with that variable name. I must also check the symbol table for an entry with the scope of global.



```
String value = (String) node.jjtGetValue();
```



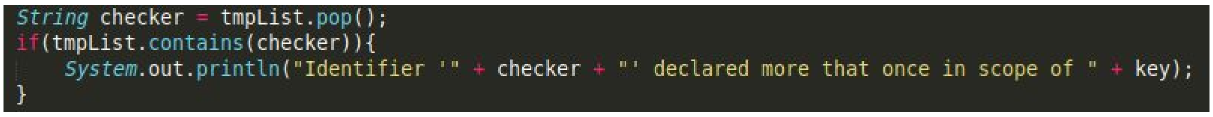
```
else if(st.getType(value, scope).size()==0 && st.getType(value, "global").size()==0){
    System.out.println(value + " is not declared within scope");
}
```

Is no identifier declared more than once in the same scope?

Every time an identifier is declared it is added to into the symbol table. To perform this check I iterate through the symbol table to see if any identifier appears twice with the same scope.

Since this check does not need to visit any of the node I can create a function in the symbol table to perform this check.

The symbol table has a hashtable with the scope as its key and a linkedlist as it's value and the declarations are stored in the linkedlist. The function will check duplicate identifiers in each linkedlist in the hashtable. The function will print out a each duplicate identifier and its respective scope.



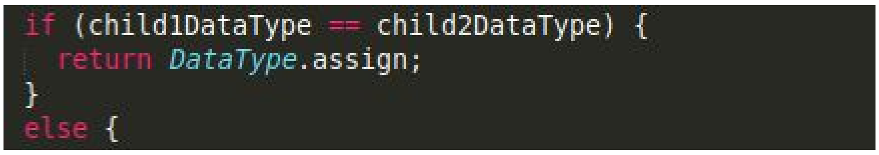
```
String checker = tmpList.pop();
if(tmpList.contains(checker)){
    System.out.println("Identifier '" + checker + "' declared more that once in scope of " + key);
}
```

The tmpList is the linkedList for each scope. An identifier is popped of tmpList and then the tmpList is checked to see if it contains the popped identifier.

This will detect constants, variables and functions which have duplicates.

Is the left-hand side of an assignment a variable of the correct type?

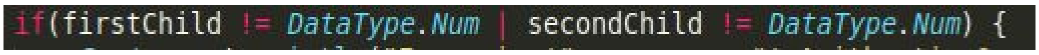
To perform this check I must check the type of identifier on the left side of an assignment and everything on the right side. When designing the AST I took this into consideration and when an assignment occurs its node on the AST tree has two children. The first child is the identifier being written too, the second child can be an identifier, arithmetic operation or a function call. An arithmetic operation will be of type number, a function call will by of the type of its identifier. Every time an identifier is visited it will return its type, when a arithmetic operation is valid it will return type integer. A simple check in assignment visitor function will check that both its children are of the same type, and if not it will result in an error message.



```
if (child1DataType == child2DataType) {
  return DataType.assign;
}
else {
```

Are the arguments of an arithmetic operator the integer variables or integer constants?

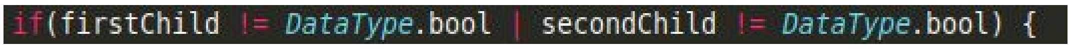
To perform this check I must check that both arguments in the operation are of type integer. This was taken into consideration when developing the AST. When a '+' or '-' is reach by the parser it creates an arithmetic operation node. Since some token is reached before either operation I forced the operations node up the tree to make its two children the arguments to its operation. Now a simple check to see if both its children are of type integer.



```
if(firstChild != DataType.Num | secondChild != DataType.Num) {
```

Are the arguments of a boolean operator boolean variables or boolean constants?

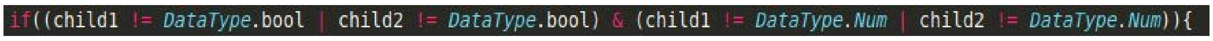
To perform this check I must check that both arguments in the operation are of type integer. This check is very simpler to the check above. A boolean operation has two children both of which must be comparison operations as per the grammar provided. Once the two comparison operations are checked to be correct, as I well go through next, they should both return a boolean datatype. If they fail an error will be displayed.



```
if(firstChild != DataType.bool | secondChild != DataType.bool) {
```

Are the arguments of a comparison operator both of the same type?

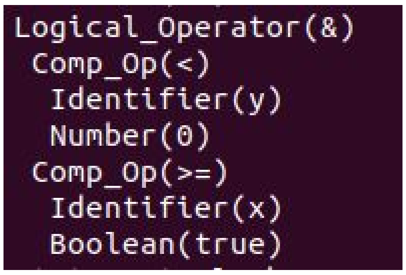
To perform this check I must check that both arguments are of the same type. Like above I check that both of the children of the operation are of the same type.



```
if((child1 != DataType.bool | child2 != DataType.bool) & (child1 != DataType.Num | child2 != DataType.Num)){
```

Are the arguments of a Logical operator both of the same type?

A logical operation is performed on two comparison operation nodes.

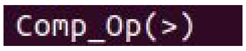


```
Logical_Operator(&)
 Comp_Op(<)
  Identifier(y)
  Number(0)
 Comp_Op(>=)
  Identifier(x)
  Boolean(true)
```

When both of logical_operations children are accepted if either comparison operation fail it will cause an error message for the logical operation. I use the node value to enter the correct operation into the error message.

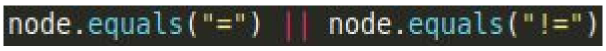
Additional comparison operation check

There is 6 comparison operations in this grammar. When i was designing the AST I only created one node for them, comp_op and passed the value of the token into the comp_op node.



```
Comp_Op(>)
```

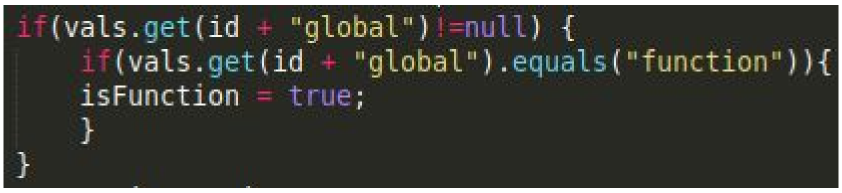
A boolean comparison can not be perform with a ‘<’ or ‘>=’, only with ‘=’ and ‘!=’. A integer comparison can be perform on each of the operations. To perform this check I check the value of the node comp_op and if it equals ‘=’ or ‘!=’ the arguments can be both of type integer or boolean, else only of type integer.



```
node.equals("=") || node.equals("!=")
```

Is there a function for every invoked identifier?

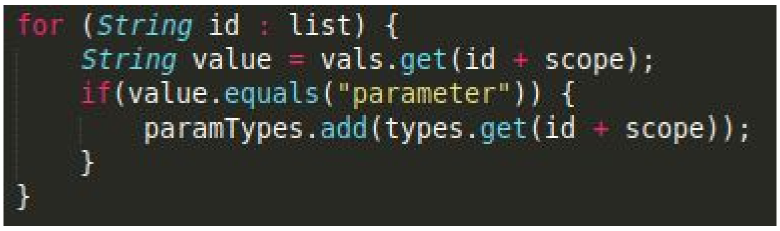
To perform this check I must check that every time the function call node is visited that its first child and scope are the key to the vals hashtable with a value of function. This is because functions can only be defined in the scope of global. To do this I use the isFunction function I created in the symbol table. I then only allow the function call node to evaluate if it is true, else i will print an error.



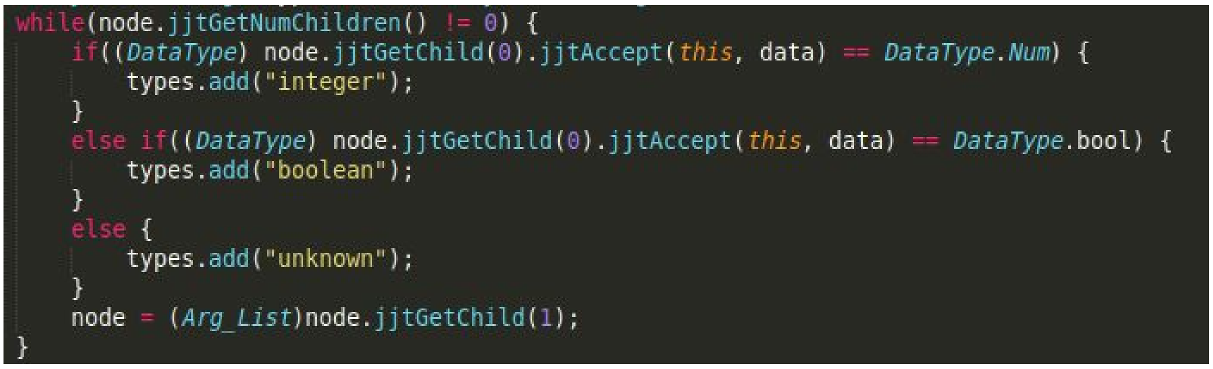
```
if(vals.get(id + "global")!=null) {
    if(vals.get(id + "global").equals("function")){
    isFunction = true;
    }
}
```

Does every function call have the correct number of arguments?

To perform this I must check if the number of arguments in the function call matches the number of parameters the function has. Every time time the parser reaches a parameter it adds it to the symbol table with the scope as the id of the function. I created a function in the symbol table to return the list of parameters for each function when the function call node is visited. I then created a private function in the semantic check class to return a list of arguments which are being called for that function. Because I sent up the AST to make arg_list to only have 2 children, argument and arg_list, this function could keep adding arguments to a list while arglist has children. Every time I add an argument to the list I check if the child, arglist, has children and if it does I continue adding to the list. Ithen check if the two lists are of the same length.



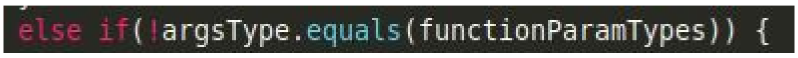
```
for (String id : list) {
    String value = vals.get(id + scope);
    if(value.equals("parameter")) {
        paramTypes.add(types.get(id + scope));
    }
}
```



```
while(node.jjtGetNumChildren() != 0) {
    if((DataType) node.jjtGetChild(0).jjtAccept(this, data) == DataType.Num) {
        types.add("integer");
    }
    else if((DataType) node.jjtGetChild(0).jjtAccept(this, data) == DataType.bool) {
        types.add("boolean");
    }
    else {
        types.add("unknown");
    }
    node = (Arg_List)node.jjtGetChild(1);
}
```

Does every argument of a function have the correct type?

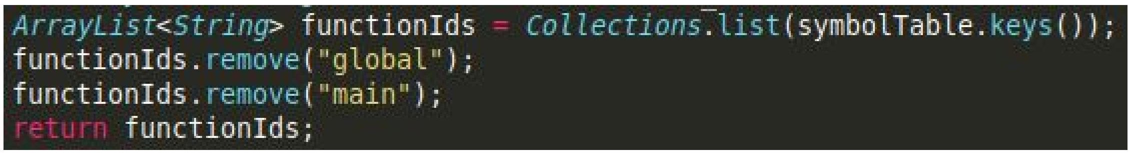
To perform this check I examine the two lists returned from the above check to see that each entry in each list is of the same type. The list of arguments first has to reversed. Arraylist has an equals function which can be used to check that two arraylist are the exact same.



```
else if(!argsType.equals(functionParamTypes)) {
```

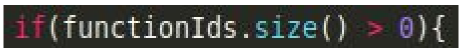
Is every function called?

To perform I need a list of functions. To get the list of functions I created a function in the symbol table which returns all the scopes in a list except main and global. This works as every function will have its own scope, and each functions scope is the same as the functions identifier.



```
ArrayList<String> functionIds = Collections.list(symbolTable.keys());
functionIds.remove("global");
functionIds.remove("main");
return functionIds;
```

In the AST I create a node for a function call. This has multiple children but the first is its identifier. When the function call node is visited I remove its identifier from the list. If after all the nodes are visited the list is greater than zero, at least one function as not been called. I then print out the name of each function.



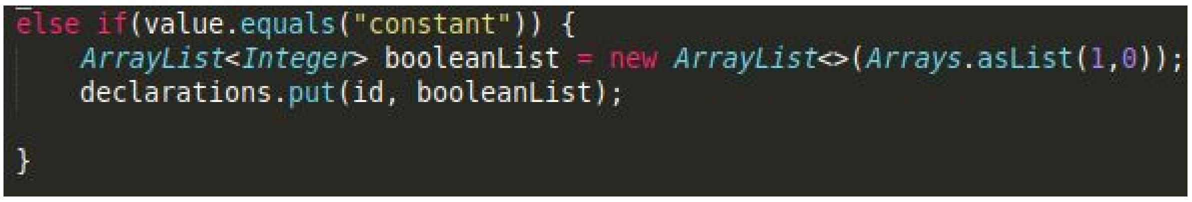
```
if(functionIds.size() > 0){
```

Does every function return the correct type?

To perform this check I must check if the return function type in the function header is equal to the return type in the function body. The parser creates a node for the return call inside the function. If there the return call doesn't return anything I check that the function is of type void. If it is not it will return in an error. If the return call returns a type I check if that type is the same type as specified in the function header. If not it results in an error. If the return call returns a type and the function is of type void this also returns in an error.

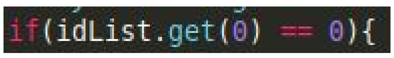
Is every variable both written to and read from?

To perform this check i must check if every identifier in the symbol table is both written to and read from. To do this I created a function in the symbol table which creates a hashtable it the scope as its key and another hashtable as its value. The inner hashtable is populated with the identifiers as its keys and a linkedlist as its value. The linkedlist is populated with two integers 1 and 0, true and false. When a variable written to the first value in the list is set to 1, true. When it is read from the last value in the list is set to 1, true. A variable is initially set with a linkedlist of [0,0] and a constant is set with [1,0] as it has already been written to.



```
else if(value.equals("constant")) {
    ArrayList<Integer> booleanList = new ArrayList<>(Arrays.asList(1,0));
    declarations.put(id, booleanList);

}
```

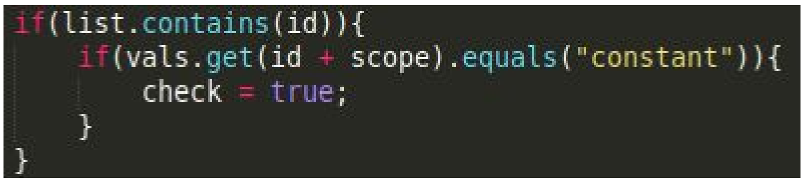
When an identifier is visited it each value in the linkedlist is updated accordingly. After each node is visited I iterate through the hashtable and print and error message when a 0 is encountered in a linkedlist.



```
if(idList.get(0) == 0){
```

Is any constants reassigned?

To perform this check I must check if the first child of an assignment, the identifier, is a constant.If it is than it is an error. I created a function in the symbol table to check if a identifier is a constant. The function will check with scope of global and scope of the location of the identifier.



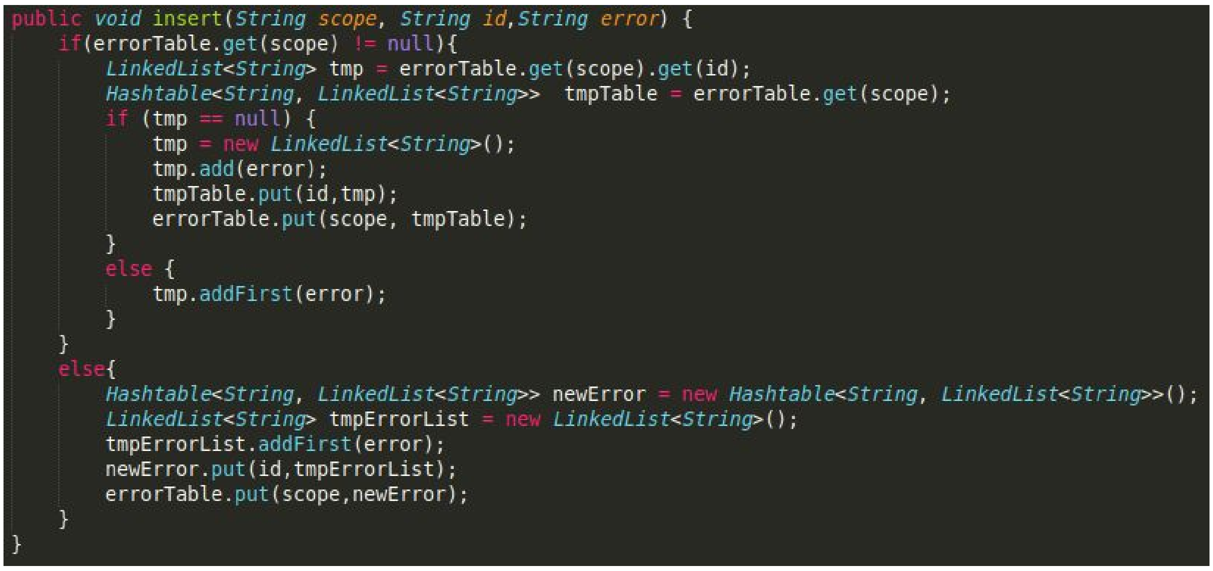
```java
if(list.contains(id)){
    if(vals.get(id + scope).equals("constant")){
        check = true;
    }
}
```

Aside

After implementing all semantic checks which were indicated in the assignment specification I implemented 4 addition checks. Through revision of the class I abstracted code to make the class cleaner and reduce code duplication.

Through various iterations of testing each semantic check individually and together, unit and integration testing, I notice error messages seemed disorganised. I then decided to add the errors to a linkedlist and store them in a hashtable with the identifier as the key. I could then print each error for a specific identifier together. Through further testing I notice that errors for the same identifier in main and function printed out together. I decided to add the previously described hashtable into another hashtable with the identifier's scope as the key. I also notice duplicate errors in the console. I decided to cast the linkedlist to a set to remove duplicates before printing the errors.

I created a ErrorTable class to store the error messages and initiate it when the first node is visited.



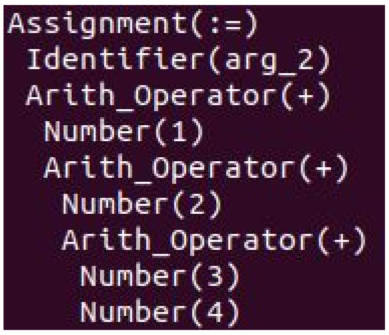
```java
public void insert(String scope, String id,String error) {
    if(errorTable.get(scope) != null){
        LinkedList<String> tmp = errorTable.get(scope).get(id);
        Hashtable<String, LinkedList<String>>  tmpTable = errorTable.get(scope);
        if (tmp == null) {
            tmp = new LinkedList<String>();
            tmp.add(error);
            tmpTable.put(id,tmp);
            errorTable.put(scope, tmpTable);
        }
        else {
            tmp.addFirst(error);
        }
    }
    else{
        Hashtable<String, LinkedList<String>> newError = new Hashtable<String, LinkedList<String>>();
        LinkedList<String> tmpErrorList = new LinkedList<String>();
        tmpErrorList.addFirst(error);
        newError.put(id,tmpErrorList);
        errorTable.put(scope,newError);
    }
}
```

I call the print error method after every node is visited. The print error method then iterates through both of the hash tables and linkedlist to print each error.

Intermediate Representation

The intermediate representation is achieve in a similar way to the semantic checks. I created a class which implements the visitor class. This Three Address Code class is initiated after the semantic checks in the calparser.jjt. We designing my interpretation of intermediate representation I followed Taci's design. The majority of the nodes visited

Assignment
In 3-address code there is at most one operator on the right-hand of an instruction. Therefore an assignment such as, "arg_2 := 1 + 2 + 3 + 4;" is not valid. This is represented on the AST as:



```
Assignment(:=)
 Identifier(arg_2)
 Arith_Operator(+)
  Number(1)
  Arith_Operator(+)
   Number(2)
   Arith_Operator(+)
    Number(3)
    Number(4)
```

Assignment always has two children, the first is the identifier being written to and the second is the value being read from. To make this assignment valid for Three Address Code (TAC) I employ the use of temporaries. A temporary is a compiler generated variable. The second child of an assignment can be an arithmetic operation, function call, negative identifier, identifier and number.

When the second child is an identifier it simply print as: `arg_2 = t25`

When the second child is a number it will print as: `arg_2 = 1`

When the second child is a negative identifier it will print as: `arg_2 = -multiply`

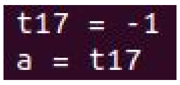
When the second child is a function call it will print as: `arg_2 = cathal() 0`

An arithmetic operation has two child, if both are an identifier, function call or a number it will print as : `arg_2 = 1 + multiply` , `arg_1 = multiply() 2 + 2`

An arithmetic operation can have the same children as assignment, when any of the children are an arithmetic operation, a negative identifier or a negative number, causing the assignment to have more that one operation, such as : `arg_2 := 1 + -arg_1;` , `arg_2 := 1 + 2 + 3;` compiler generated variables, temporaries must be employed.

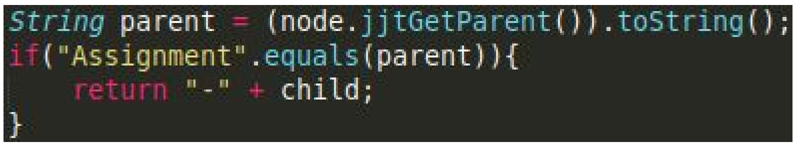
To do this I check it see if an arithmetic operation's parent is not an assignment, therefore its parent is an arithmetic operation. If so I create a temporary variable which will be assignment the two children with the arithmetic operation between them. I then return the temporary variable to the parent arithmetic operation, it's second child is now a temporary. This process limits each line of code to only have a single operation, such to a valid TAC line of code.

Original I assigned every arithmetic operation, negative identifier and negative number to a temporary and then assigned that temporary to the first child of an assignment, the left hand side of an assignment.
This resulted in:



```
t17 = -1
a = t17
```

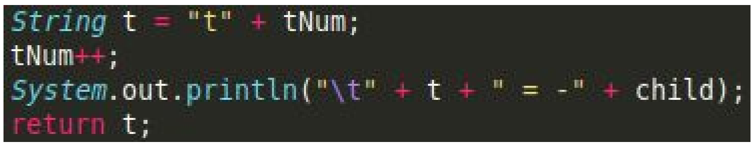
I optimised the number of temporaries created by checking the parent of an arithmetic operation, negative number and negative identifier, if their parents were an assignment this meant they would be valid TAC lines of code. I then would print them normal assignment.



```
String parent = (node.jjtGetParent()).toString();
if("Assignment".equals(parent)){
    return "-" + child;
}
```

When testing both approaches on the multiply function test provided in the assignment specification it reduced the number of temporaries created from 17 to 9, that is a 53% reduction of temporaries created.

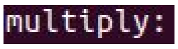
I initiated a temporary count and incremented every time a value was assigned to a temporary, this is how I generated the temporaries:



```
String t = "t" + tNum;
tNum++;
System.out.println("\t" + t + " = -" + child);
return t;
```

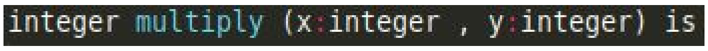
Function

When creating the TAC for a function the first step is creating the label for access to the function from a function call. This is completed by printing out the function's identifier followed by a colon. If the function is called multiply the label generated would be:



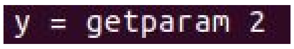
```
multiply:
```

The next task is to get the parameters. This is completed by printing the parameter's id followed by "= getparam " and its position on the param stack. The position on the stack correlates with the position of the parameter in the list of parameters.



```
integer multiply (x:integer , y:integer) is
```

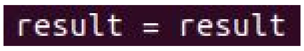
,



```
y = getparam 2
```

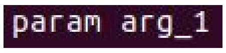
I will cover the body of a function in each of the other sections as they are what a function's body is comprised of.

The last task is to handle the return statement of a function. If the function is void it will have no return statement. To check this I check if the return statement node has no children. If the return statement has a child it is the identifier to be returned. The TAC code is print as so:



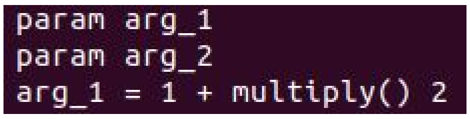
```
result = result
```

Function Call

The first task in creating TAC for a function call is to push the arguments to the param stack. I created a function to handle this. The function i created takes the function calls parameter list as an argument. Each parameter list's second child is a parameter list. The first child is a parameter which is print out as: 

```
param arg_1
```

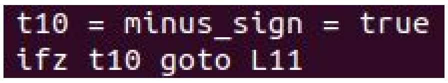
I then child to see it the parameter list's second child parameter list has any children, if it has a repeated the process. The function the returns the number of parameters in the printed out. The function call is print out will the number of parameters which have been added to the stack:



```
param arg_1
param arg_2
arg_1 = 1 + multiply() 2
```

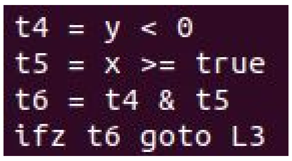
Conditions

Conditions can contain comparisons operations and logical operations. If a condition consists of just a single comparison such as arg < 2, it is assigned to a temporary and then that temporary variable is placed in the if statement, which I will further explain in the next section.



```
t10 = minus_sign = true
ifz t10 goto L11
```

The condition could consist of two comparison operations joined by a logical operation. In this case each of the two comparisons are assigned to temporaries and then joined by the logical operation in another temporary.

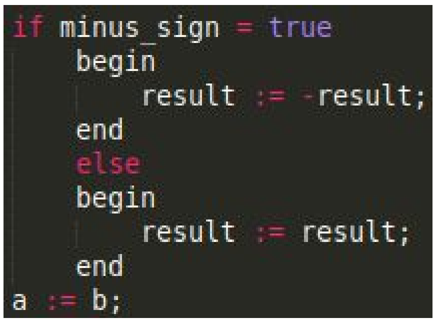


```
t4 = y < 0
t5 = x >= true
t6 = t4 & t5
ifz t6 goto L3
```

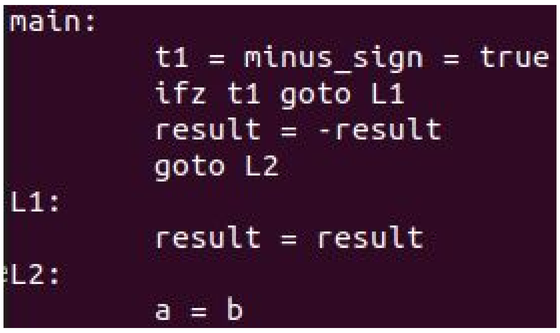
If Statements

To check if the statement is a if statement I check that the value of the statement node is equal to "if". The first task is to allow the conditions temporaries to be printed.

The approach I took when designing the TAC for a if statement was to check if the condition evaluates to false and if so goto the label associated with the else. If the condition evaluates to be true it will contain through all of the statements until it reaches a goto which is located before the else label. The goto will point to a label which is directly after the else block.



```
if minus_sign = true
    begin
        result := -result;
    end
    else
    begin
        result := result;
    end
a := b;
```

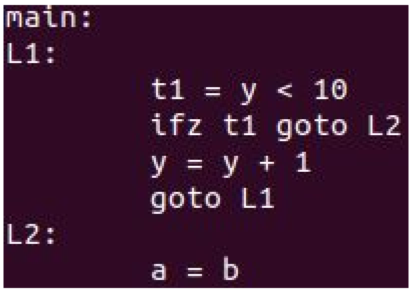


```
main:
        t1 = minus_sign = true
        ifz t1 goto L1
        result = -result
        goto L2
L1:
        result = result
L2:
        a = b
```

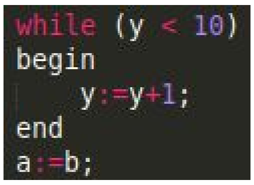
While Statements

To check if the statement is a while statement I check that the value of the statement node is equal to “while”. The first task is to allow the conditions temporaries to be printed.

The approach I took when designing the TAC for a while statement was to check if the condition evaluates to false and if so goto the label directly after the while block of code. If the condition evaluates to be true it will contain through all of the statements until it reaches a goto which is located at the end of the block of code. This goto will point to a label which is above the while statement, the condition can then be evaluated again and will continue until the condition evaluates to false.



```
main:
L1:
        t1 = y < 10
        ifz t1 goto L2
        y = y + 1
        goto L1
L2:
        a = b
```

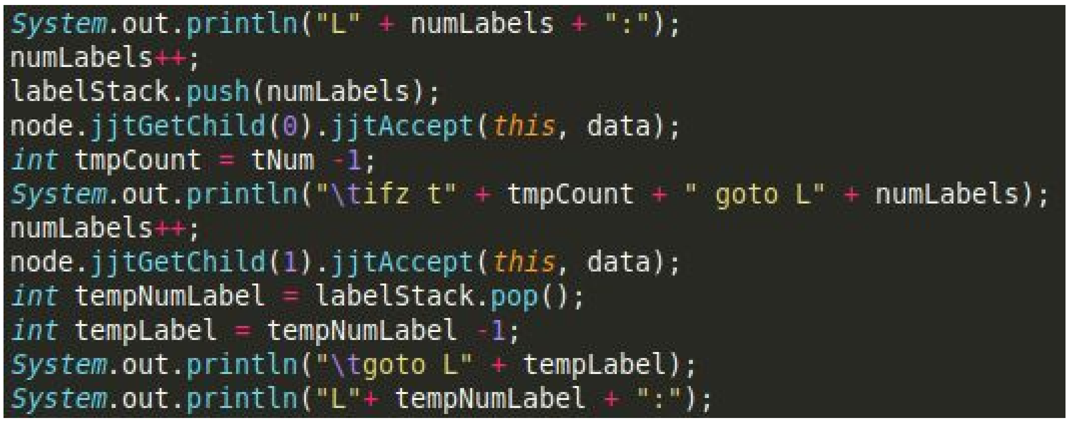


```
while (y < 10)
begin
    y:=y+1;
end
a:=b;
```

Handling nested

After testing I realised that the compiler was not printing the labels correctly for nested conditions. This is be for example it would create the TAC for a while loop inside an if block before creating the TAC for the else block.

To combat this I initiated a stack which stored the label associated with that level of nesting. As more nested conditions are created the labels are all stored on the stack. Then when the highest level of nesting is reach and the labels are popped of the stacked and printed in the appropriated position. This process is basically the same for both the if and while statements. Below is the code used to handle while statements:



```
System.out.println("L" + numLabels + ":");
numLabels++;
labelStack.push(numLabels);
node.jjtGetChild(0).jjtAccept(this, data);
int tmpCount = tNum -1;
System.out.println("\tifz t" + tmpCount + " goto L" + numLabels);
numLabels++;
node.jjtGetChild(1).jjtAccept(this, data);
int tempNumLabel = labelStack.pop();
int tempLabel = tempNumLabel -1;
System.out.println("\tgoto L" + tempLabel);
System.out.println("L"+ tempNumLabel + ":");
```

Conclusion

I thoroughly enjoyed working on this assignment as it gave me a great insight and appreciation for what a compiler actually is. As this assignment was a continuation of the previous assignment it provided me with a sense of completion as I feel as I have created a product.

I found building the AST rewarding as my first submission of the parser was designed with it in mind. When I came to building the AST I found it not too difficult as I had a good understand has of how the tree would look.

The most difficult task of this designing this compiler was the creation of semantic checks. What reduced this burden this was my approach of working from the lower level node first such as statement and identifier to perform lower level semantic checks. Through the implementation of semantic checks i gained a better understand of methods such as jjtappect and jjtgetparent. This reduced the workload further.

As I gained a better understanding of visitor class I was able to reduce the complexity of my check, but also can 4 more checks which I feel enhance my compiler.

I also reduced the complexity of the errors outputted to optimise the user’s chances of successful debugging.

The generation of the IR code was my favourite aspect of the assignment. This is because I had a great understand of the visitor class and how node interacted.

The aspect I am most proud of is the compilers ability to handle nested conditions. This was achieved through the use of a stack which would store the appropriate labels.

From research of various implementations of TAC I found the need to optimise the use of compiler generated temporaries. The next step in the development of the compiler is optimisation and I feel like I have reduced the workload need to do so.

If I had work time and scope while developing this compiler I would have optimised the temporaries generated in handle conditions. I would have further reduced the complexity of the semantic checks through greater use of my datatype class.

As a concluding note I would say that I found this assignment, as well as the previous assignment, to have been greatly beneficial in terms of understand of a compiler but also in terms of implementation of real world product.