Assignment 2: Semantic Analysis and Intermediate Representation for the CCAL Language

Declaration on Plagiarism

This form must be filled in and completed by the student(s) submitting an assignment

Name: Emily McGivern
Programme: CASE 4
Module Code: CA4003
Assignment Title: Assignment 1: Semantic Analysis and Intermediate Representation for the CCAL Language
Submission Date:
Module Coordinator: David Sinclair

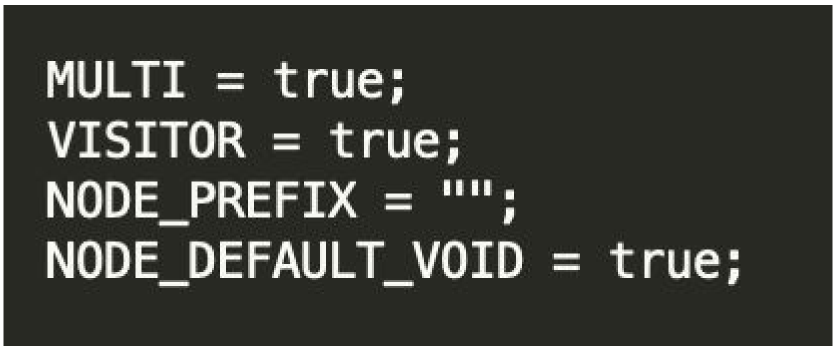
Name: Emily McGivern **Date: 16/12/2019**

Introduction

The aim of this assignment is to add semantic analysis checks and intermediate representation generation to the lexical and syntax analyser you have implemented for the CCAL language as Assignment 1. Unfortunately, I found this assignment extremely difficult and did not anticipate the time it would take to complete the semantic checks and intermediate representation. I completed as much of the assignment as possible.

Options

JJTree introduces some additional options along with the options in JavaCC. I implemented some of these new options into my .jjt file. The MULTI option set to true means that a decoration will generate a node derived from a class. The VISITOR option set to true will insert a jjtAccept method in each node class and generate a Visitor interface. The NODE_PREFIX option set to “” overwrites the default prefix of AST to “”. Usually, to suppress the creation of a node for a production rule you must set the decorator #void on that rule. However, I added the option NODE_DEFAULT_VOID = true to my options to ensure that any non-decorated production rules would not have nodes created. To understand these options fully, I used the official JAVACC documentation on JJTree, https://javacc.org/jjtree.



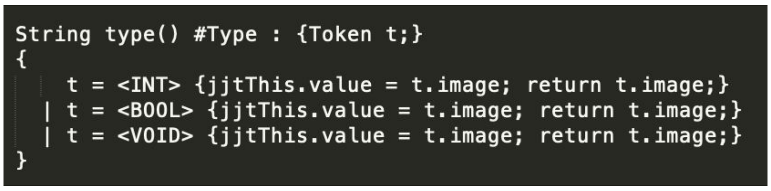
```
MULTI = true;
VISITOR = true;
NODE_PREFIX = "";
NODE_DEFAULT_VOID = true;
```

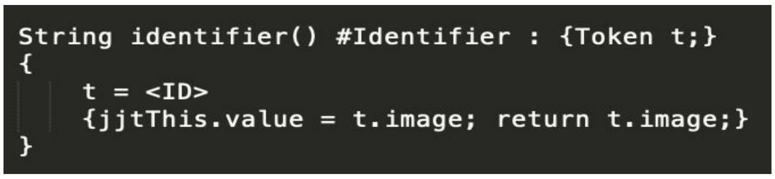
AST

I began by doing research on Abstract Syntax Trees and the best way to implement them. I also used the code supplied on the course page as a guide for beginning my implementation. Generating the AST involved using the code from the first assignment, namely the CCALParser.jj file. I began by converting my CCALParser.jj file to a .jjt file as it contains annotations which will help build the AST. I then began working on my grammar to ensure that the correct AST structure is created. I added decorations to my production rules to define the type of node created. I only added decorations to rules which I thought would be needed for semantic analysis later on.

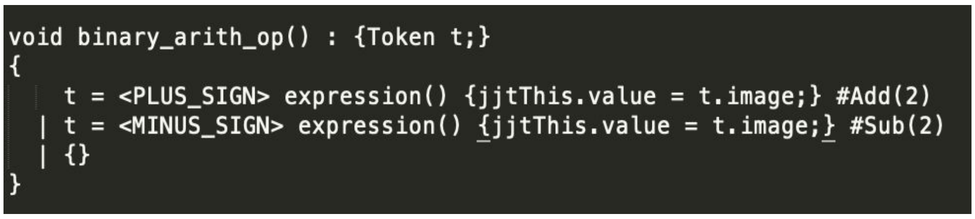
Some of the values of tokens are disregarded in the process of converting a parse tree to a syntax tree. However, some of them need to be included. To handle this, I had to implement some terminals as non-terminals. Examples of this can be seen in type() and identifier(). As I needed to return the values of these tokens, I changed the return type from *void* to *String*.

After I had carried this out, I then replaced occurrences of <ID> with a reference to my identifier() production rule.

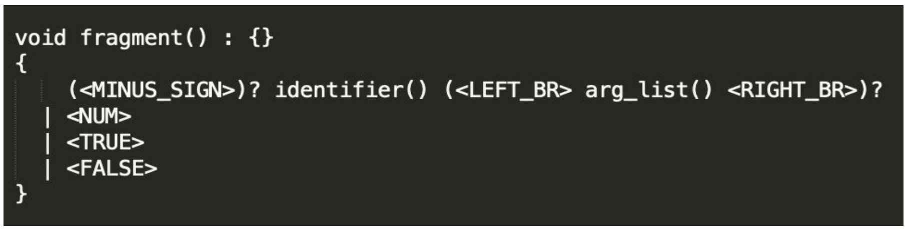

```
String type() #Type : {Token t;}
{
    t = <INT> {jjtThis.value = t.image; return t.image;}
  | t = <BOOL> {jjtThis.value = t.image; return t.image;}
  | t = <VOID> {jjtThis.value = t.image; return t.image;}
}
```

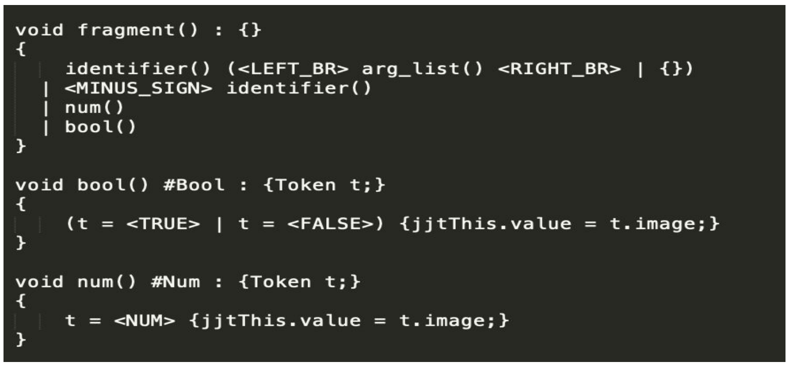

```
String identifier() #Identifier : {Token t;}
{
    t = <ID>
    {jjtThis.value = t.image; return t.image;}
}
```

After this I began looking again at the nodes I had created to see could I add any further which would help me create my AST. I added decorations to part of some production rules, which can be seen in binary_arith_op(). This creates an add and sub node, both with 2 children.

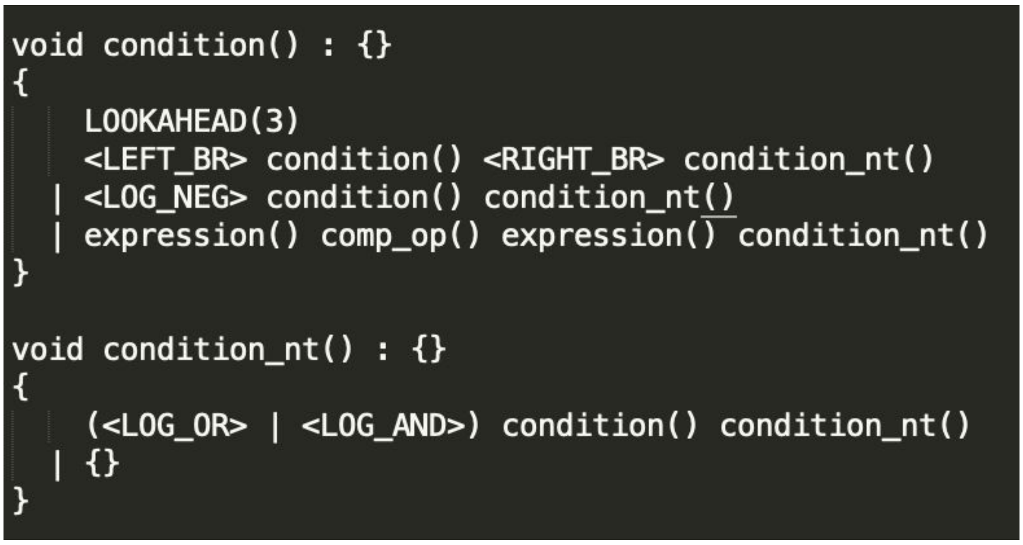

```
void binary_arith_op() : {Token t;}
{
    t = <PLUS_SIGN> expression() {jjtThis.value = t.image;} #Add(2)
  | t = <MINUS_SIGN> expression() {jjtThis.value = t.image;} #Sub(2)
  | {}
}
```

I made some further changes in some other production rules by creating new production rules. I did this in fragment() by creating production rules for num() and bool(). This meant that a node is created for each and they can be used as a DataType for Semantic Checks later on.

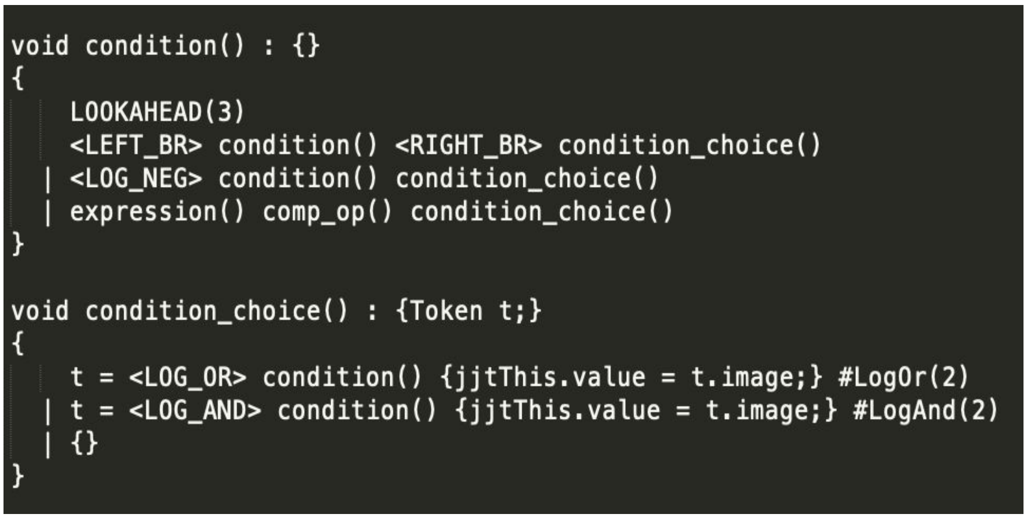

```
void fragment() : {}
{
    (<MINUS_SIGN>)? identifier() (<LEFT_BR> arg_list() <RIGHT_BR>)?
  | <NUM>
  | <TRUE>
  | <FALSE>
}
```


```
void fragment() : {}
{
    identifier() (<LEFT_BR> arg_list() <RIGHT_BR> | {})
  | <MINUS_SIGN> identifier()
  | num()
  | bool()
}

void bool() #Bool : {Token t;}
{
    (t = <TRUE> | t = <FALSE>) {jjtThis.value = t.image;}
}

void num() #Num : {Token t;}
{
    t = <NUM> {jjtThis.value = t.image;}
}
```

As I worked through my production rules, I could see some changes I had to make to ensure I had all the decorators in place for the nodes that I needed. I started with small test programs and worked up to ensure that I had covered all bases with my AST. I had to make some changes to condition() and condition_nt() so i could have nodes with 2 children being created for OR or AND operations.



```
void condition() : {}
{
    LOOKAHEAD(3)
    <LEFT_BR> condition() <RIGHT_BR> condition_nt()
  | <LOG_NEG> condition() condition_nt()
  | expression() comp_op() expression() condition_nt()
}

void condition_nt() : {}
{
    (<LOG_OR> | <LOG_AND>) condition() condition_nt()
  | {}
}
```



```
void condition() : {}
{
    LOOKAHEAD(3)
    <LEFT_BR> condition() <RIGHT_BR> condition_choice()
  | <LOG_NEG> condition() condition_choice()
  | expression() comp_op() condition_choice()
}

void condition_choice() : {Token t;}
{
    t = <LOG_OR> condition() {jjtThis.value = t.image;} #LogOr(2)
  | t = <LOG_AND> condition() {jjtThis.value = t.image;} #LogAnd(2)
  | {}
}
```

After I was happy with the changes to my production rules for building my AST I could then move on to working on my Symbol Table implementation.

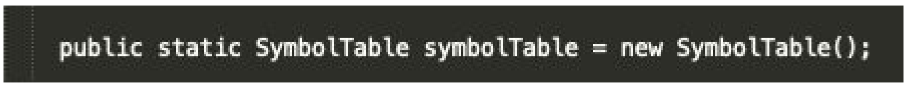
Symbol Table

The next step in the assignment was constructing a symbol table which could handle scope. I read through the notes to decide how to go about implementing my symbol table. The suggested way was to use a hash table with external chaining, which is what I chose to implement. I then did some research on what I should be keeping track of in my symbol table. Some of the resources I consulted were:

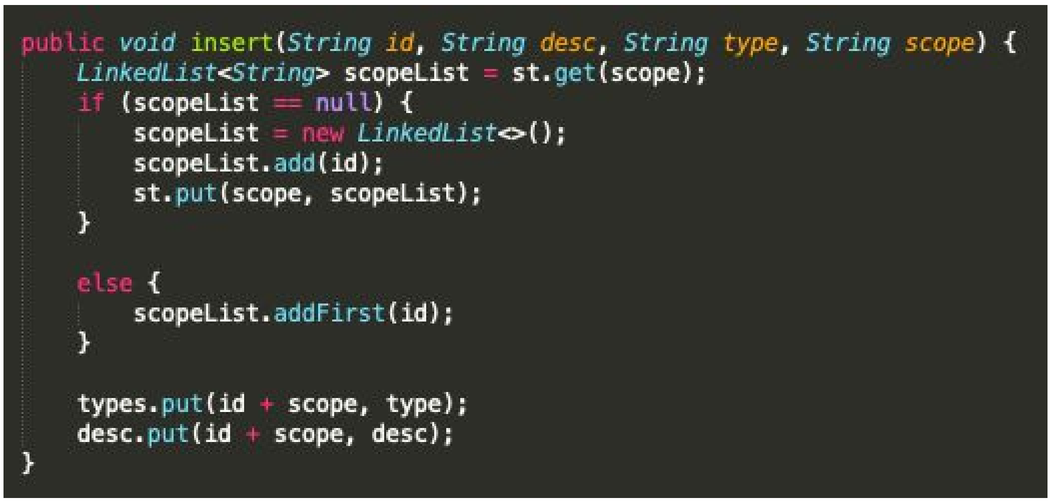
https://www.tutorialspoint.com/compiler_design/compiler_design_symbol_table.htm https://introcs.cs.princeton.edu/java/44st/

I created a symbol table object with 3 separate hash tables. The hashtables track the scope, the type and whether we have a function, variable, constant etc. To begin, I used the basic example from the notes STC.java, but I named mine SymbolTable.java. This gave me a skeleton of how to begin writing the symbol table.

The first step was instantiating the symbol table in my .jjt file. I then had to implement some basic methods to construct the Symbol Table. I implemented an insert method to insert items into my symbol table. The insert method was called in my production rules to add the relevant information.

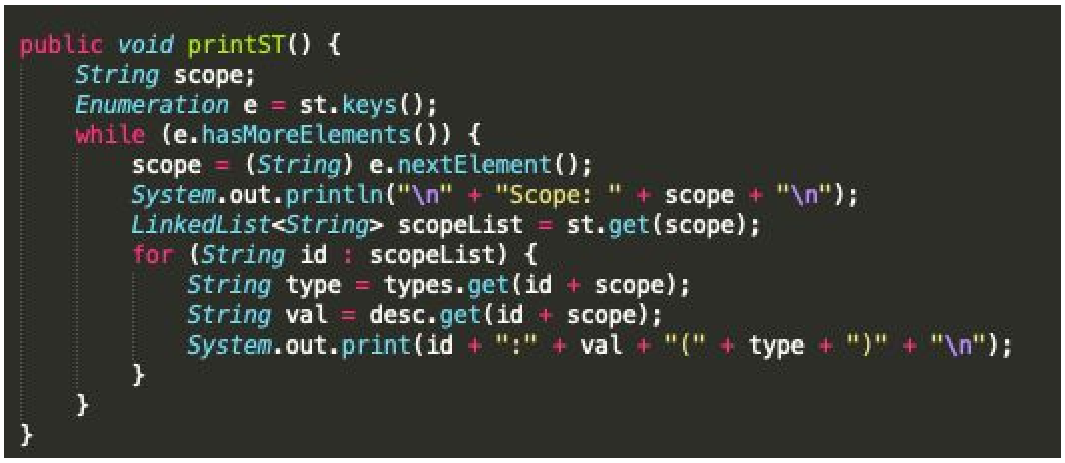


```
public static SymbolTable symbolTable = new SymbolTable();
```



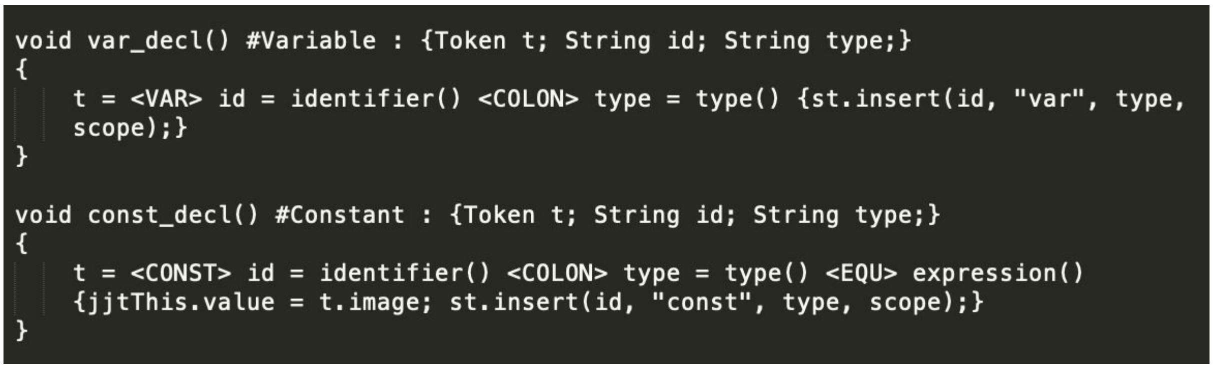
```
public void insert(String id, String desc, String type, String scope) {
    LinkedList<String> scopeList = st.get(scope);
    if (scopeList == null) {
        scopeList = new LinkedList<>();
        scopeList.add(id);
        st.put(scope, scopeList);
    }

    else {
        scopeList.addFirst(id);
    }

    types.put(id + scope, type);
    desc.put(id + scope, desc);
}
```

I then implemented a print method which printed the symbol table to the terminal. This was essential for testing and ensuring I was on the right track.



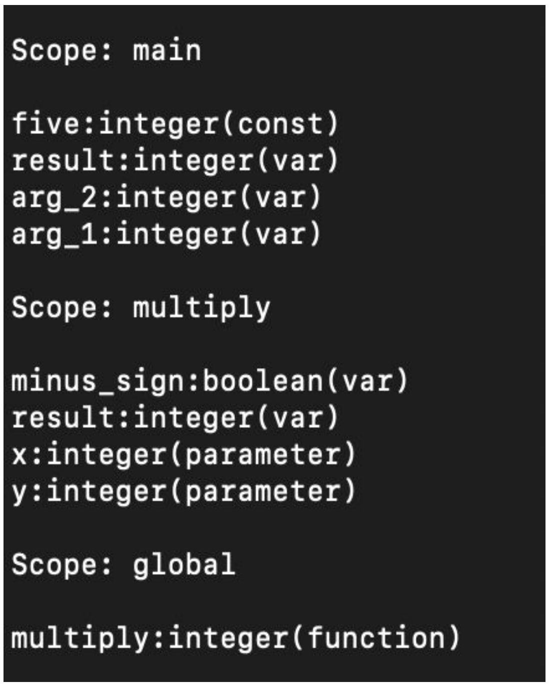
```
public void printST() {
    String scope;
    Enumeration e = st.keys();
    while (e.hasMoreElements()) {
        scope = (String) e.nextElement();
        System.out.println("\n" + "Scope: " + scope + "\n");
        LinkedList<String> scopeList = st.get(scope);
        for (String id : scopeList) {
            String type = types.get(id + scope);
            String val = desc.get(id + scope);
            System.out.print(id + ":" + val + "(" + type + ")" + "\n");
        }
    }
}
```

The following shows how the insert method was called and how items were inserted into the Symbol Table.



```
void var_decl() #Variable : {Token t; String id; String type;}
{
    t = <VAR> id = identifier() <COLON> type = type() {st.insert(id, "var", type,
    scope);}
}

void const_decl() #Constant : {Token t; String id; String type;}
{
    t = <CONST> id = identifier() <COLON> type = type() <EQU> expression()
    {jjtThis.value = t.image; st.insert(id, "const", type, scope);}
}
```

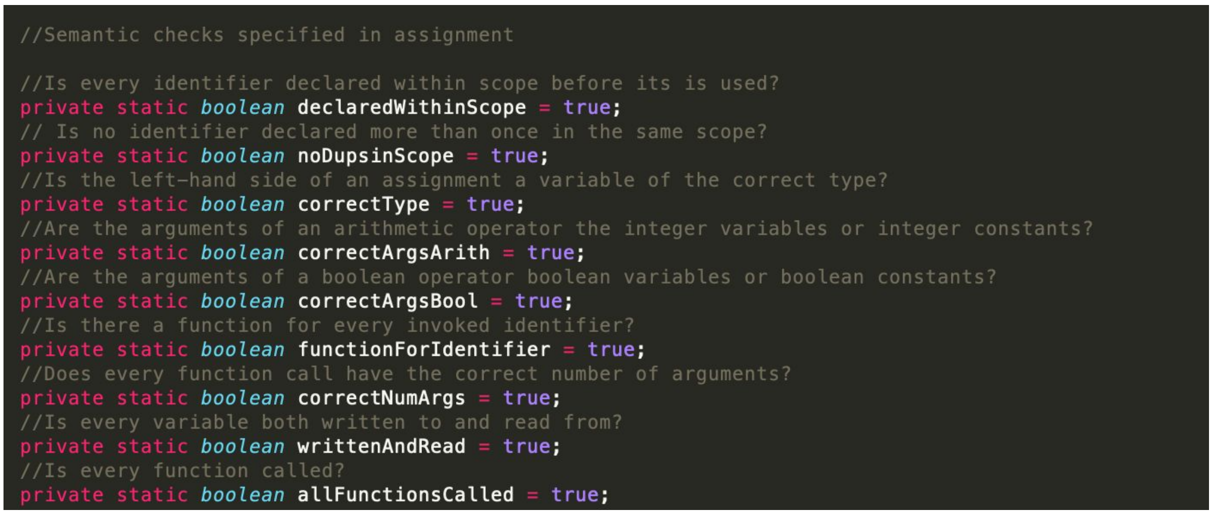
To test symbol table, I ran test from language spec and the following was outputted to terminal. When I was happy with the results, I began working on my semantic checks.



```
Scope: main

five:integer(const)
result:integer(var)
arg_2:integer(var)
arg_1:integer(var)

Scope: multiply

minus_sign:boolean(var)
result:integer(var)
x:integer(parameter)
y:integer(parameter)

Scope: global

multiply:integer(function)
```

Semantic Checks

The next task was to implement a number of semantic checks. We were supplied with a list of semantic checks to implement and we could also add some further ones that were not listed. Firstly, I had a look at the example files on the course page to get an idea of how to start. To begin, I created a visitor class which would carry out the semantic analysis. I followed the notes to carry this out, and created a class called SemanticChecks.java. As per the notes, each visitor must implement the CCALParserVisitor interface which was generated by JJTree. I followed the sample file on the module page, TypeCheckVisitor.java, to get started with my code. As I worked through my semantic checks, I realised that there were many methods which I could have implemented in my symbol table to help me with my semantic checks. I implemented methods in both my SymbolTable and SemanticChecks classes. I also made a DataType.java file. I implemented a number of booleans to keep track of whether my checks were passing and failing as I thought this was the easiest way to return whether they passed or failed by setting the value to true or false depending on the outcome.

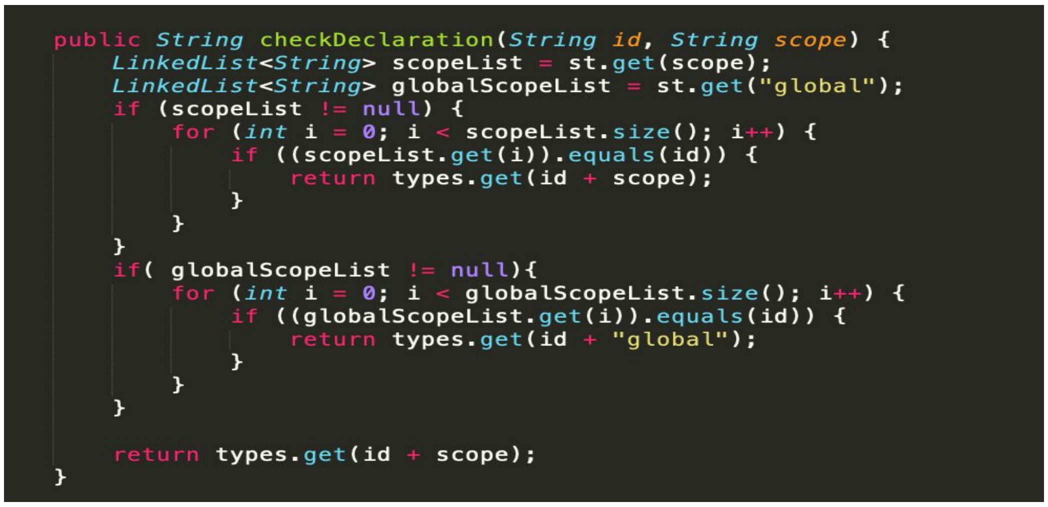


```java
//Semantic checks specified in assignment

//Is every identifier declared within scope before its is used?
private static boolean declaredWithinScope = true;
// Is no identifier declared more than once in the same scope?
private static boolean noDupsinScope = true;
//Is the left-hand side of an assignment a variable of the correct type?
private static boolean correctType = true;
//Are the arguments of an arithmetic operator the integer variables or integer constants?
private static boolean correctArgsArith = true;
//Are the arguments of a boolean operator boolean variables or boolean constants?
private static boolean correctArgsBool = true;
//Is there a function for every invoked identifier?
private static boolean functionForIdentifier = true;
//Does every function call have the correct number of arguments?
private static boolean correctNumArgs = true;
//Is every variable both written to and read from?
private static boolean writtenAndRead = true;
//Is every function called?
private static boolean allFunctionsCalled = true;
```

I attempted to implement all the semantic checks, but unfortunately I was not able to. I spent a very long time working on my AST and Symbol Table which meant that I did not have the time to understand and implement the semantic checks properly and correctly. I will discuss what I did manage to implement below.

1. **Is every identifier declared within scope before its is used?**

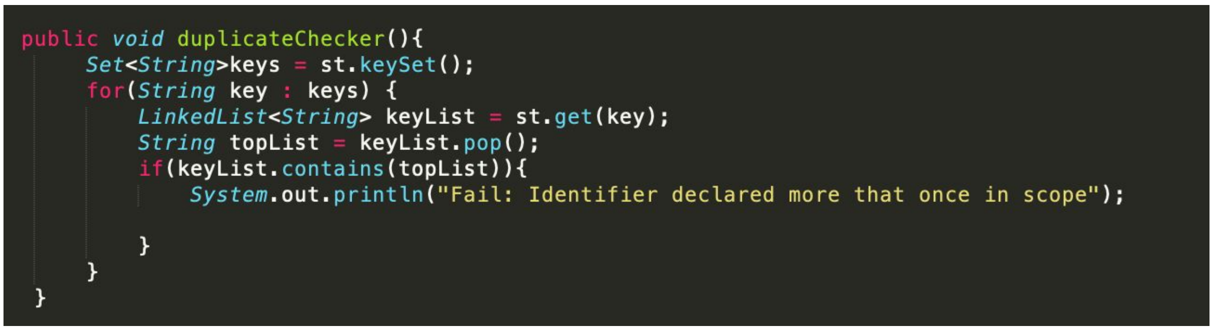
To implement this check, I needed to be able to check if an identifier has been declared already. To do this, I implemented a checkDeclaration() function in my SymbolTable class. The function checks both the current scope and global scope to see if the identifier has been declared. If it has, it returns the type so that a DataType can be assigned to the identifier. If it hasn't, an error message is returned. I call this function in the identifier node where it can check each identifier as they are passed in.



```java
public String checkDeclaration(String id, String scope) {
    LinkedList<String> scopeList = st.get(scope);
    LinkedList<String> globalScopeList = st.get("global");
    if (scopeList != null) {
        for (int i = 0; i < scopeList.size(); i++) {
            if ((scopeList.get(i)).equals(id)) {
                return types.get(id + scope);
            }
        }
    }
    if( globalScopeList != null){
        for (int i = 0; i < globalScopeList.size(); i++) {
            if ((globalScopeList.get(i)).equals(id)) {
                return types.get(id + "global");
            }
        }
    }

    return types.get(id + scope);
}
```

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

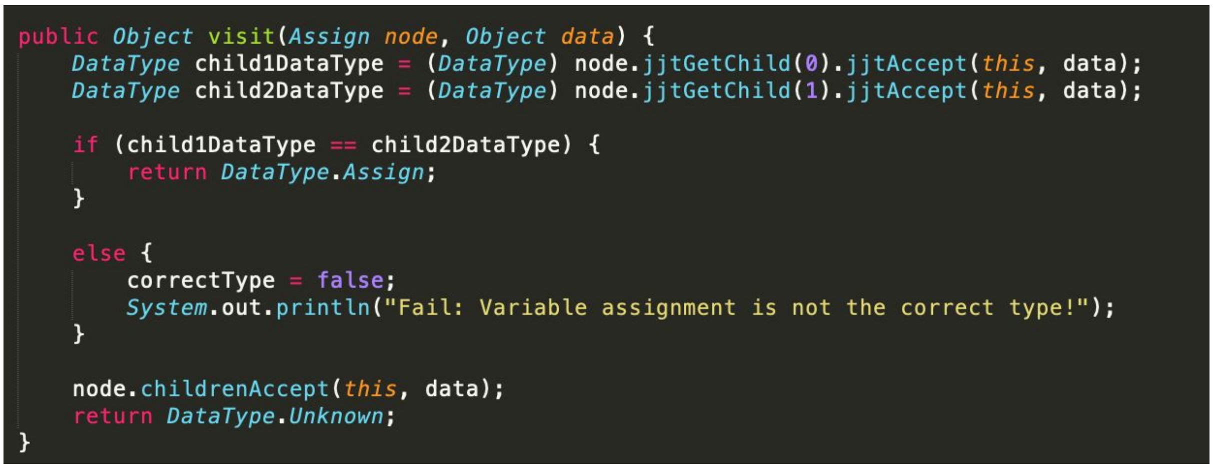
To perform this check, I needed to implement a function which checked if there are duplicate identifiers in the same scope. I implemented this function in my Symbol Table. This function gets a set of keys from the main symbol table. It then iterates through the set and creates a list. The item at the top of the list is popped off and then checked to see if it is in the list. If it is, it means that there is a duplicate value and an error message is returned. If it is not in the list, the check passes.



```
public void duplicateChecker(){
     Set<String>keys = st.keySet();
     for(String key : keys) {
         LinkedList<String> keyList = st.get(key);
         String topList = keyList.pop();
         if(keyList.contains(topList)){
             System.out.println("Fail: Identifier declared more that once in scope");

         }
     }
 }
```

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

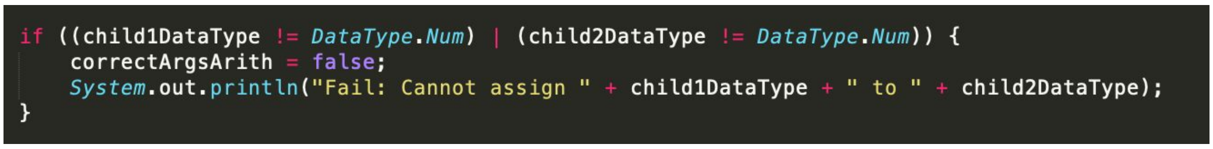
I implemented this check in the assigned node as this check involves assignment. In this check, I compared the assigned DataType of two child nodes in an assignment to see if they are they same type. If they are the same the check will pass as this means that the variable is the correct type. If they do not match the check will fail.



```
public Object visit(Assign node, Object data) {
    DataType child1DataType = (DataType) node.jjtGetChild(0).jjtAccept(this, data);
    DataType child2DataType = (DataType) node.jjtGetChild(1).jjtAccept(this, data);

    if (child1DataType == child2DataType) {
        return DataType.Assign;
    }

    else {
        correctType = false;
        System.out.println("Fail: Variable assignment is not the correct type!");
    }

    node.childrenAccept(this, data);
    return DataType.Unknown;
}
```

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

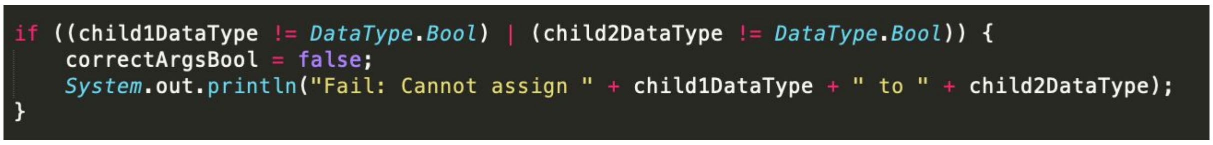
I performed a basic check in the add and sub nodes to check whether two arguments involved in a sum were the same type. I was unsure on how to carry out this check but I decided that a check like this could also be useful and also checks the validity of arithmetic operations



```java
if ((child1DataType != DataType.Num) | (child2DataType != DataType.Num)) {
    correctArgsArith = false;
    System.out.println("Fail: Cannot assign " + child1DataType + " to " + child2DataType);
}
```

5. **Are the arguments of a boolean operator boolean variables or boolean constants**

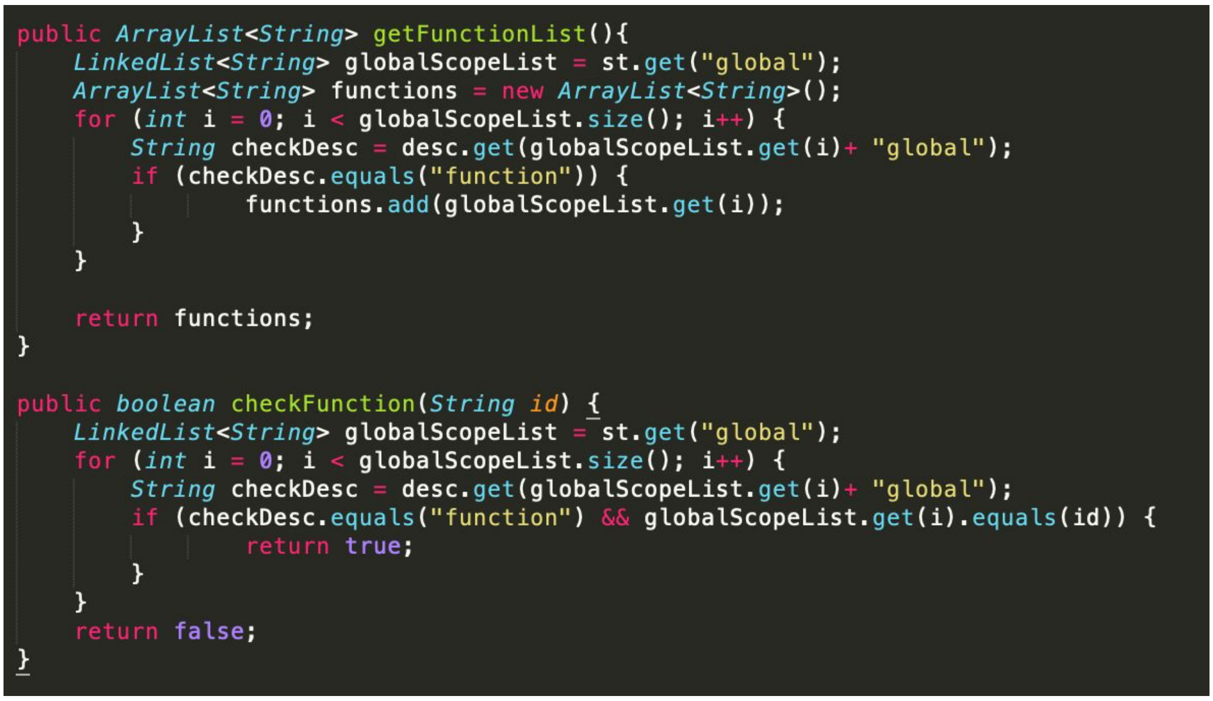
I carried out this check in a similar way to the previous check by checking whether the arguments involved were boolean.



```java
if ((child1DataType != DataType.Bool) | (child2DataType != DataType.Bool)) {
    correctArgsBool = false;
    System.out.println("Fail: Cannot assign " + child1DataType + " to " + child2DataType);
}
```

6. **Is every function called?**

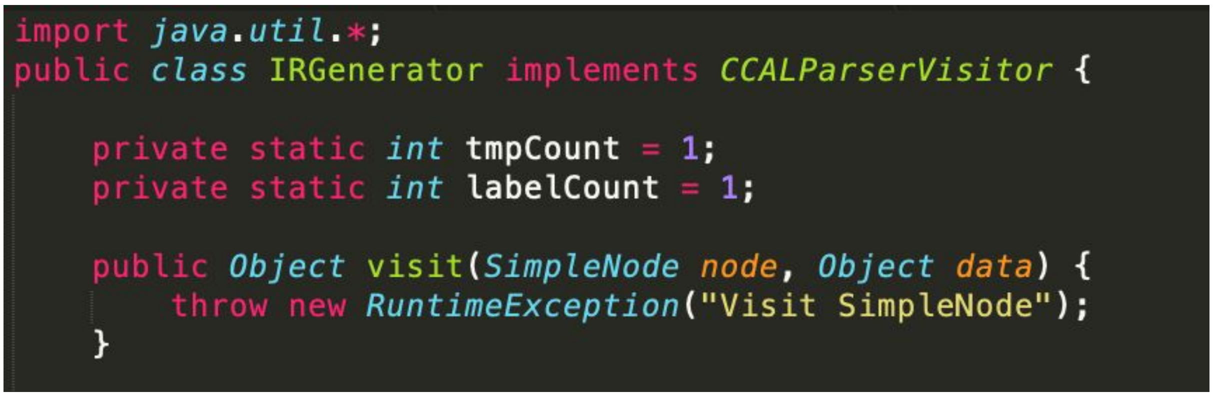
To check if every function was called, I decided that I needed some way to compare a list of all the functions in a program with a list of functions which are actually called. I implemented the method getFunctionList() in my symbol table which created a list of all items which were described as functions in the symbol table. Then I created an empty list in my SemanticChecks. In my function node, I check to make sure it is a function when it is invoked and I add it to the list. I then checked the list against each other to make sure that every function was called. If there was a function missing it means that not all functions were invoked.



```java
public ArrayList<String> getFunctionList(){
    LinkedList<String> globalScopeList = st.get("global");
    ArrayList<String> functions = new ArrayList<String>();
    for (int i = 0; i < globalScopeList.size(); i++) {
        String checkDesc = desc.get(globalScopeList.get(i)+ "global");
        if (checkDesc.equals("function")) {
                functions.add(globalScopeList.get(i));
        }
    }

    return functions;
}

public boolean checkFunction(String id) {
    LinkedList<String> globalScopeList = st.get("global");
    for (int i = 0; i < globalScopeList.size(); i++) {
        String checkDesc = desc.get(globalScopeList.get(i)+ "global");
        if (checkDesc.equals("function") && globalScopeList.get(i).equals(id)) {
                return true;
        }
    }
    return false;
}
```

Intermediate Representation using 3-address code

To implement the Intermediate Representation using 3-address code, I began by looking at the sample file on the course page. I used this as a guide to begin my implementation. I was not able to complete this part of the assignment but I made a basic attempt which I could build on to complete the assignment in the future. From what I did learn about IR code generation, I created a class called IRGenerator. It behaves like SemanticChecks by visiting each node recursively. I added 2 variables at the start of my class, tmpCount and labelCount. I initialised all the nodes in my IRGenerator class but this is as far as I got.



```java
import java.util.*;
public class IRGenerator implements CCALParserVisitor {

    private static int tmpCount = 1;
    private static int labelCount = 1;

    public Object visit(SimpleNode node, Object data) {
        throw new RuntimeException("Visit SimpleNode");
    }
```

Conclusion

I found this assignment extremely difficult and I am very disappointed I didn't manage to finish it entirely. I feel like I did not manage my time correctly and I spent too long implementing my AST and Symbol Table while underestimating how long everything else would take. I also did not account for any changes I could have to make to my AST to that the semantic checks executed correctly. Throughout the report, I have highlighted the work I did manage to carry out.

How to run the program

I have included a bash script which will generate the compiler when it is ran. Run *"bash build.sh"* to execute it.