CA4003 - Compiler Construction Assignment 2

Matthew Nolan - 16425716

Declaration on Plagiarism

Assignment Submission Form

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

Name(s): **Matthew Nolan**
Programme: **Compiler Construction Assignment**
Module Code: **CA4003**
Assignment Title: **Semantic Analysis and Intermediate Representation**
Submission Date: **16/12/2019**
Module Coordinator: **David Sinclair**

I/We declare that this material, which I/We now submit for assessment, is entirely my/our own work and has not been taken from the work of others, save and to the extent that such work has been cited and acknowledged within the text of my/our work. I/We understand that plagiarism, collusion, and copying are grave and serious offences in the university and accept the penalties that would be imposed should I engage in plagiarism, collusion or copying. I/We have read and understood the Assignment Regulations. I/We have identified and included the source of all predicates, ideas, opinions, and viewpoints of others in the assignment references. Direct quotations from books, journal articles, internet sources, module text, or any other source whatsoever are acknowledged, and the source cited are identified in

the assignment references. This assignment, or any part of it, has not been previously submitted by me/us or any other person for assessment on this or any other course of study.

I/We have read and understood the referencing guidelines found at

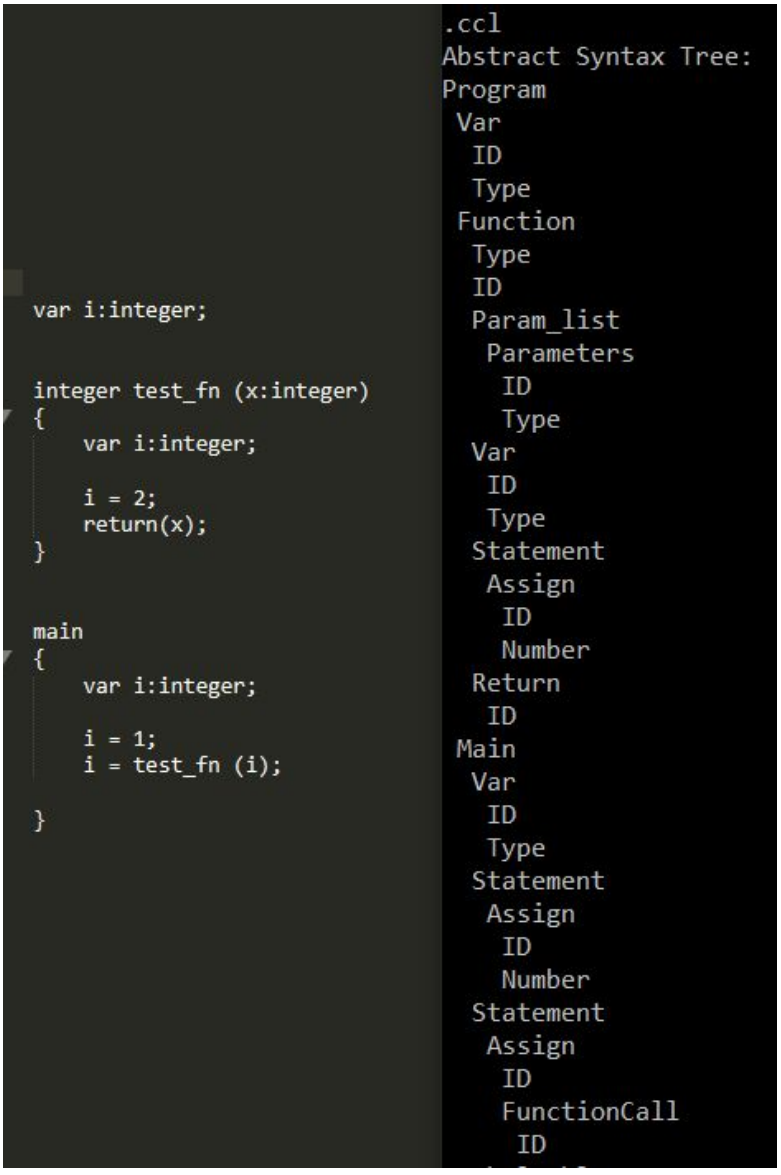
http://www.dcu.ie/info/regulations/plagiarism.shtml ,

https://www4.dcu.ie/students/az/plagiarism and/or recommended in the assignment guidelines.

Name(s): Matthew Nolan Date: __16/12/19_____

1. Abstract Syntax Tree

For the first part of this assignment I had to implement an Abstract Syntax Tree.We had to implement the Abstract syntax tree by adding our code from the first assignment to a jjt file and then adding the relevant syntax to build the tree.I began by defining my root Node which was Program. I added the relevant decorators to the grammar that I needed in order to build the AST. Below is an example of a tree that my program generated. I had to add non-terminals such as id() and number() in order to get values for different nodes on the tree. This allowed me to access id and types for later use in the semantic checks. I used the sample code in the CCAL specification in order to test the AST.



```
var i:integer;

integer test_fn (x:integer)
{
    var i:integer;

    i = 2;
    return(x);
}

main
{
    var i:integer;

    i = 1;
    i = test_fn (i);

}
```

```
.ccl
Abstract Syntax Tree:
Program
 Var
  ID
  Type
 Function
  Type
  ID
  Param_list
   Parameters
    ID
    Type
  Var
   ID
   Type
  Statement
   Assign
    ID
    Number
  Return
   ID
 Main
  Var
   ID
   Type
  Statement
   Assign
    ID
    Number
  Statement
   Assign
    ID
    FunctionCall
     ID
```

.

2. Symbol Table

The second part of the assignment was to implement a Symbol Table. I took advice from the notes and used the method of chaining 3 hash tables together. I decided to use this method as hash tables as very efficient with regards to what I would be using them for. The have an O(1) for insertion and lookup. I started off by writing an add and print function for the symbol tables. The symbol table would be separated based off scope and its corresponding values and types were put into the other hash tables. These two functions allowed me to test the symbol table.

```
public void addToTable(String id, String value, String type, String scope ){

    LinkedList<String> CurrentScope = symbolTable.get(scope);

    if(CurrentScope == null){
        CurrentScope = new LinkedList<>();
        CurrentScope.add(id);
        symbolTable.put(scope,CurrentScope);
    }
    else{
        CurrentScope.addFirst(id);
    }
    values.put(id+scope, value);
    types.put(id+scope, type);
}

public void PrintTable(){
    String scope;
    Enumeration table = symbolTable.keys();

    while (table.hasMoreElements()){
        scope = (String) table.nextElement();
        System.out.println("\nScope: "+scope);
        LinkedList<String> ID_List = symbolTable.get(scope);
        for (String id : ID_List) {
            String val = values.get(id + scope);
            String type = types.get(id + scope);
            System.out.print(val + " " + id + " " + type + "\n");
        }
    }
}
```

I later added helper functions to the symbol table in order to make the semantic checks a lot easier. I added functions such as getFunctions, getDupsInScopes and get types. Below is an example of the Symbol table produced.

```
SymbolTable:

Scope: main
Var i integer

Scope: global
Function test_fn integer
Var i integer

Scope: test_fn
Var i integer
Parameters x integer
Functions
test_fn
```

3. Semantic Checks

The next part was to implement all of the semantic checks that were described in the CCAL specification. I was unable to implement all of the mentioned semantic checks. Below are the ones I managed to implement.

The semantic checks are done in the file SemCheck.java The checks which were given in the assignment brief are as follows:

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

  Using the helper class I wrote in my symbol table class. This check was carried out after the visitor was done traversing the tree. Once it is finished, the hash table is checked to see if there are two or more declarations for the same variable in the same scope.

```
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;
}
```

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

This check is done within the plus and minus operators. A check is done on their Child nodes to see if either one of them is not equal to num. If one is not num they fail.

- **Are the arguments of a Boolean operator Boolean variables or Boolean constants?**

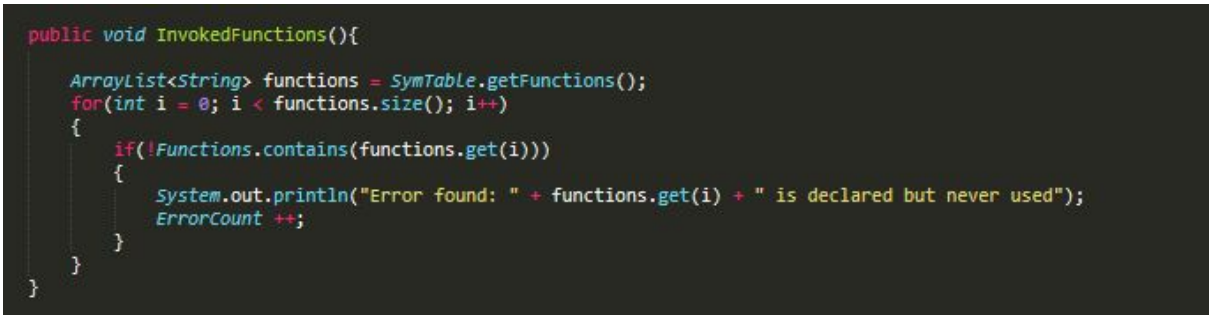
Like the previous check, it checks to see the values of the nodes, if they are not equal then they fail.

- **Is there a function for every invoked identifier?**

This check uses another helper function I wrote in the symbol table class. The function returns a list of the functions defined within the scope.

```
public ArrayList<String> getFunctions() {
    LinkedList<String> list = symbolTable.get("global");
    ArrayList<String> functionNames = new ArrayList<String>();
    for (String func : list){
        if(values.get(func + "global") != null){
            String functionName = values.get(func + "global");
            if(functionName.equals("Function")){
                functionNames.add(func);
            }
        }
    }
    return functionNames;
}
```

Once the visitor has traversed the tree this function is called.



```
public void InvokedFunctions(){

    ArrayList<String> functions = SymTable.getFunctions();
    for(int i = 0; i < functions.size(); i++)
    {
        if(!Functions.contains(functions.get(i)))
        {
            System.out.println("Error found: " + functions.get(i) + " is declared but never used");
            ErrorCount ++;
        }
    }
}
```

This function compares the list of functions defined in the scope to the list of functions called. The list of functions called gets updated in “ASTFunctionCall” node.

To run the code

1) Jjtree Assignment2.jjt
2) Javacc Assignment2.jj
3) Javac *.java
4) Java CCALTokeniser test.ccl