# CA4003 - Compiler Construction Assignment 2

Michael O'Hara -16414554

## Declaration on Plagiarism

## Assignment Submission Form

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

| |
|---|
| Name(s): **Michael O'Hara** |
| 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): ____Michael O'Hara_____ Date: __16/12/19_____

# 1. Abstract Syntax Tree

For the first part of this assignment I had to implement an Abstract Syntax Tree. To start this I had to take the code from the first assignment from this module and turn it into a .jjt file. Then make syntax changes in number of places in the file in order to create an Abstract Syntax Tree. I began by defining a root Node called “ProgramStart” which would be used to represent every possible node in my AST. Then i went through adding any of the needed decorators in order to fill out the tree. A number of non terminals such as ID() were added for multiple reasons. One being it allowed me to use these for the later semantic checks but it also allowed me to add the values for those nodes onto the tree.

In order to print out the Abstract Syntax Tree the user code calls “*root.dump(“”);”* on the parser. This is what my AST looked like for one of the code examples from the CCAL language. As shown below:

Code:

```
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);
}
```

AST Generated by code:

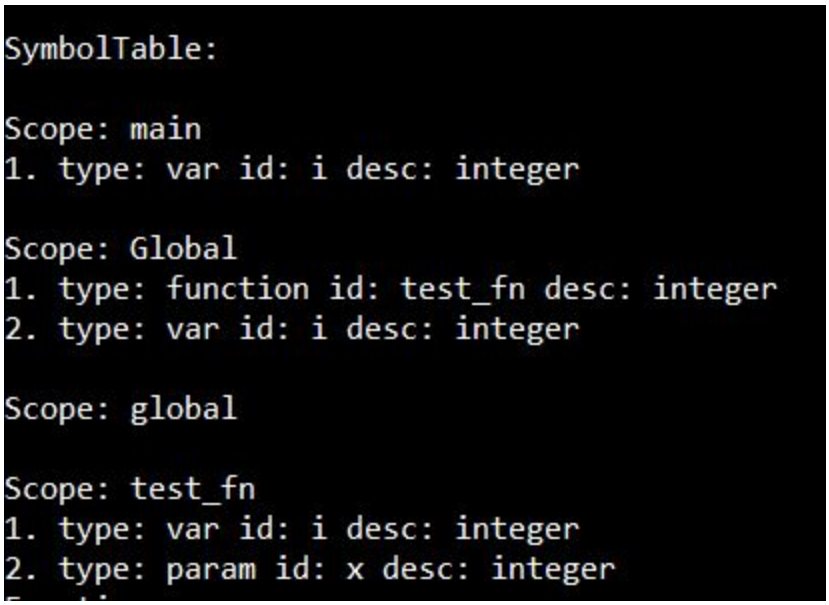
```
Abstract Syntax Tree:
ProgramStart
 Var
  ID
  Type
 Function
  Type
  ID
  Param_list
   Parameters
    ID
    Type
  Var
   ID
   Type
  Statement
   Assign
    ID
    Number
  Return
   ID
 Main
  Var
   ID
   Type
  Statement
   Assign
    ID
    Number
  Statement
   Assign
    ID
    FunctionCall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

## 2. Symbol Table

The second part of the assignment was to implement a Symbol Table. I first had a look around online to see if I could find the most efficient method of doing this. The tutorial I found at this website here (https://www.tutorialspoint.com/compiler_design/compiler_design_symbol_table.html)

From reading the article I was able to determine that hash table would be the most efficient way to do it as they are O(1) for insertion and lookup if done in the right way. This part was a bit more challenging due to the fact we have mainly covered python our time on this course but my time on INTRA gave me a better grasp of java so I was able to make a good attempt.

I made some helper functions in order to make doing the semantic checks easier, I also made functions to add to the table and to print out the table in an easily visible format:



# 3. Semantic Checks

The next part of the assignment was the Semantic checks. These were more difficult and time consuming than the other 2 previous parts. Due to poor time management skills on my part I was unable to complete all of them. The semantic checks are done in the file SemCheck.java. The checks which were given in the assignment brief that i was able to finish are as follows:

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

I was able to  complete this check using the class getDuplicateInvoke class that I had written in my Symbol Table class. It waits until the traversing of the tree is finished and then checks the hash table to see if a declaration for the same variable inside the same scope occurs more than once

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

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

The above 2 were done similarly to each other. If the operator is a constant it must be declared a certain value which remains unchanged and a variable must be init a value before its usage. This can be checked in symbol table by assigning a description to the ids of each entry. Then the table can be checked to determine if it is constant or var.

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

This check uses a helper function I wrote for the symbol table to return a list of each of the functions defined within the specific scope. Shown as follows:

```
public ArrayList<String> GetFunctions(){
    ArrayList<String> functionList = new ArrayList<String>();
    LinkedList<String> scope = symbolTable.get("global");
    for(int i =0; i < scope.size(); i++)
    {
            String val = values.get(scope.get(i)+"global");
            if(val.equals("Function"))
            {
                System.out.println(scope.get(i));
                functionList.add(scope.get(i));
            }
    }
    System.out.print(functionList);
    return functionList;
}
```

This then calls a function called checkInvokedFucntions which compares the list of functions defined in the scope to the list of functions called.

These checks are done by visiting every node and the returning data and doing the check. If any errors occur they are counted and then the total number of errors are displayed at the end of the output in the terminal window.

## 4. IR Code Generation

I was unable to complete this section due to bad time management on my part. But would have looked at it similarly to the semantic checks, in the sense that I would visit each of the nodes and perform the necessary actions to complete the code generation

## To run the files the commands needed are:

jjtree Assignment2.jjt

javacc Assignment2.jj

Javac *.java

java CCALTokeniser test.ccl