Declaration on Plagiarism

Assignment Submission Form

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

Name(s): Killian Byrne

Programme: *Compiler construction*
Module Code: *CA4003*
Assignment Title: *Assignment 2: Semantic Analysis and Intermediate Representation for the CCAL Language*
Submission Date: //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 facts, 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):Killian Byrne Date: 14/12/2019

Introduction:

This report will contain the following sections:

(1) Abstract syntax tree (AST)
(2) Symbol Table & Semantic checks
(3) Intermediate representation code
(4) How to run

In preparation for this assignment, I loaned the following books from the library: “Modern Compiler Implementation in Java” by Andrew W. Appel, and “Principles of Compiler Design” by Alfred V.Aho and Jeffrey D. Ullman.

Abstract syntax tree:

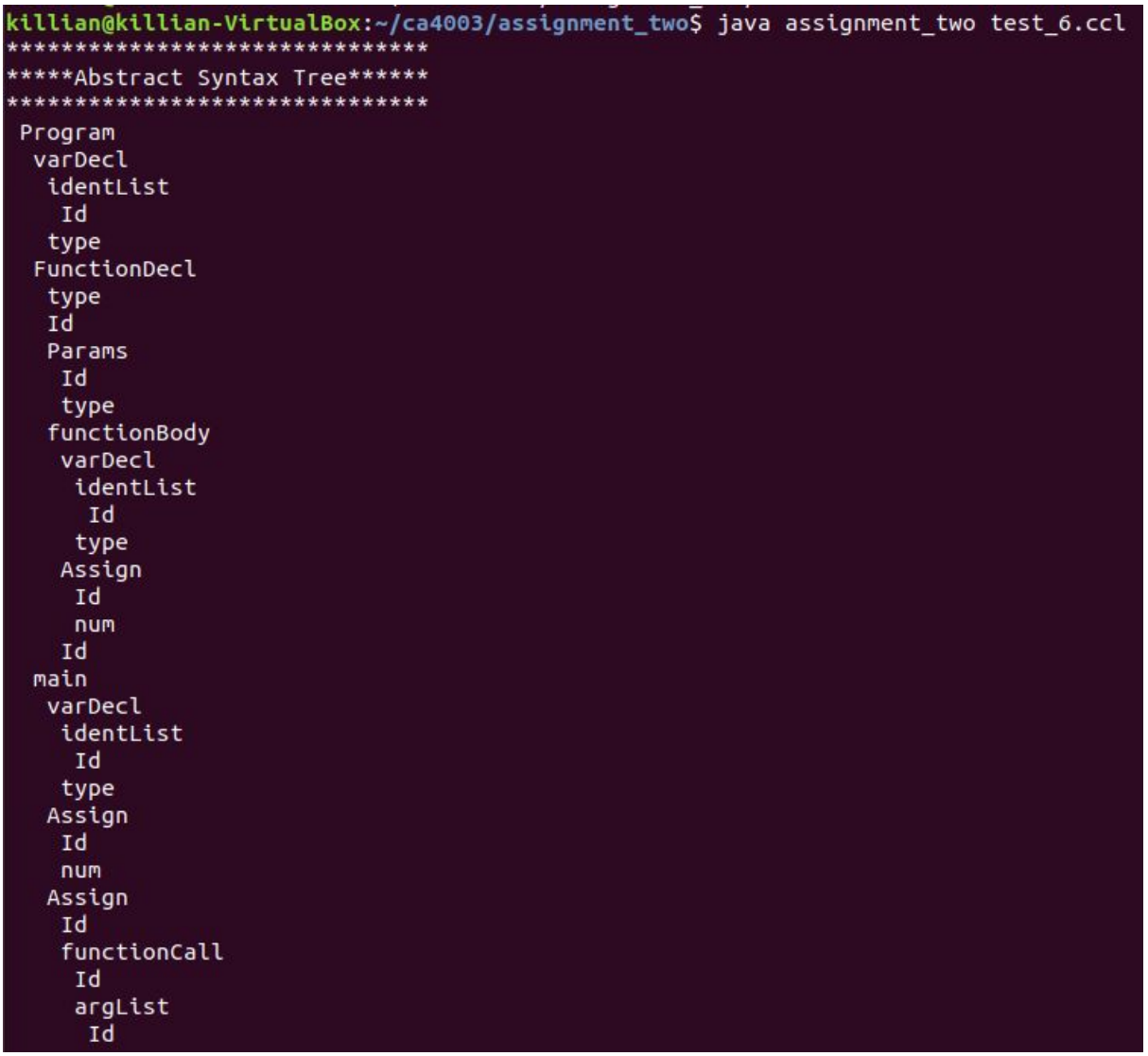
- I began this section by reading the Abstract syntax tree chapters in the two compiler books mentioned in the Introduction. I then read the CA4003 course notes (www.computing.dcu.ie/~davids/ca4003). To begin working on the AST, I found it useful to draw out sample AST’s by hand from the sample tests given on the ca4003 website.
- The first step’s involved creating a new directory within my ca4003 directory called “assignment_two” and copying my “assignment_one.jj” file from assignment one, into the new directory and renamed it as “assignment_two.jjt”. I then modified my program function from void to SimpleNode. The user code section was then modified to call SimpleNode as root, so that we can output the contents of the AST by calling ‘root.dump(“ “)’.
- Before I started decorating the jjt file, I refactored several production rules. Production rules that call other rules multiple times were refactored. For example, (decl())* was refactored to decl_list(): {} { (decl() decl_list() | {}). The

function production rule was also refactored to into function declaration (functionDecl()) and function body (function_body()). The reason for this was because function body would always contain three child nodes: declaration list, statement block and either a node resulting from the expression production rule path or in the case of nothing being returned, a return node is returned. Function declaration also returns parameter (#Params) nodes. This decorator is only called if there is more than 1 child nodes. Once initial refactoring had been done, I then started decorating production rules.

- My approach for the AST was to not try and decorate all the production rules at once. I took a top to bottom approach and decorated production rules in blocks like variable declaration, constant declaration, function declaration and function body. Once they were working as expected, I would continue on to the next block. A major error I faced with the AST was decorating the statement production rule. Initially I tried to decorate this by giving assigning three children to “#if(3)” and while three two children “#while(2)”. Once I identified that these decorators were the root of the error, the AST worked as expected. Both function production rules can be seen below:

```
void function_list () : {}
{
    (function() function_list() | {})
}

void function() #FunctionDecl : {}
{
        type() identifier() <LEFT_BRACKET> (parameter_list())? #Params(>1) <RIGHT_BRACKET>
        function_body()
}

void function_body() #functionBody(3) : {}
{
        <LEFT_BRACE>
        decl_list()
        (statement_block())
        <RETURN> <LEFT_BRACKET> (expression() | {} #Return) <RIGHT_BRACKET> <SEMICOLON>
        <RIGHT_BRACE>
}
```

- My AST was initially more complex as I was unaware as to what exactly was needed in my tree for semantic checks. Below is what the final output of my AST looks like, with test 6 as the input file.



Symbol table & Semantic checks:

- To begin this section, I opened the “assignment_twoVisitor.java” file, and examined the contents of the file. This file generates every time the jjtree is generated. It contains all nodes generated by the jjtree, so this showed me what nodes required checks from the visitor class. For generating a symbol table, I chose to use a visitor as opposed to performing this in the production rules as I could perform semantic checks simultaneously and use SimpleNode to navigate the tree in a more controlled manner than using production rules. I used a hashmap that contained an inner and outer hashmap. The first key of the inner hashmap is the symbol identifier which uses the STC class as the value for the identifier, and the first key of the scope is the scope of the symbol. The data structure is:
  HashMap<String, HashMap<String, STC>>

- The visitor creates a new symbol table every time program node is visited. From here, every child node is visited. The STC class is instantiated every time a declaration node is visited, and then added to the table for current scope. Before this action is completed, a semantic check is done to check if it already exists in the current table. An example of the symbol table generated for test 7 can be seen below:

```
*****************************
**********Symbol Table*********
*****************************

Scope: Program
        ID: multiply| DataType: Function| Type: integer| Values: {}
Scope: multiply
        ID: result| DataType: Var| Type: integer| Values: {}
        ID: x| DataType: ParamVar| Type: integer| Values: {}
        ID: y| DataType: ParamVar| Type: integer| Values: {}
Scope: Main
        ID: result| DataType: Var| Type: integer| Values: {}
        ID: arg2| DataType: Var| Type: integer| Values: {}
        ID: arg1| DataType: Var| Type: integer| Values: {}
        ID: five| DataType: Const| Type: integer| Values: {fragment=5}
```

- The semantic checks carried out are: is every identifier declared within scope before it is used, Is no identifier declared more than once in the same scope, is the left hand side of an assignment a variable of the correct type, is the program trying to overwrite a constant, are condition parameters of boolean variables, is there a function for every invoked identifier.
- Snippets of code containing the checks can be seen below. Checks are tagged with their names and the functions they are found in in STVisitor.java.

Not within scope (checkConditionIds & ASTAssign):

```
if(t.kind == 31 && (ST.get(scope).get(t.image) == null && ST.get("Program").get(t.image) == null) )
{
  System.out.println("Identifier: "+t.image+"\n\tNot declared in scope: "+scope+" Or Program");
  System.out.println("Error Line: "+t.beginLine+" Column: "+t.beginColumn+"\n");
  numErrors++;
```

Declared more than once (ASTconstDecl):

```
//if it is already in the symbol table throw a error
else
{
  System.out.println("Identifier: "+idList.get(j).image+"\n\tAlready declared in scope: "+scope);
  System.out.println("Error Line: "+idList.get(j).beginLine+" Column: "+idList.get(j).beginColumn+"\n");
  numErrors++;
}
```

Invalid type assign (ASTAssign):

```
if(t.kind == 33 && lhType != "integer")
{
  System.out.println("Invalid type assign: \n\t"+lhToken.image+":"+lhType+" <- "+t.image+":integer");
  System.out.println("Error Line: "+t.beginLine+" Column: "+t.beginColumn+"\n");
  numErrors++;
```

Cannot assign to constant (ASTAssign):

```
else
{
  System.out.println("Type:"+lhSTC.dType);
  System.out.println("Cannot assign to constant: "+lhSTC.name.image);
  System.out.println("Error Line: "+lhSTC.name.beginLine+" Column: "+lhSTC.name.beginColumn+"\n");
  numErrors++;
```

Not a boolean condition (ASTcondition):

```
if(stc.type.image != "boolean")
{
          System.out.println("Unreachable Condition: "+t.image+"\n\t Not a Boolean Identifier");
          System.out.println("Error Line: "+t.beginLine+" Column: "+t.beginColumn+"\n");
          numErrors++;
```

Invalid number of parameters (checkFunctionCall):

```
int paramsPassed = ((List<Token>)node.jjtGetChild(1).jjtAccept(this,null)).size();
if((paramsPassed) != numParams)
{
  System.out.println("Call to Function: "+t.image+"\n\tInvalid number of paramaters");
  System.out.println("Error Line: "+t.beginLine+" Column: "+t.beginColumn+"\n");
  numErrors++;
```

Here is an example of sample output using a file containing errors:

```
******************************
*******Semantic Analysis*******
******************************

Identifier: i
        Already declared in scope: Program
Error Line: 2 Column: 5

Identifier: no_dec_var
        Not declared in scope: Main Or Program
Error Line: 17 Column: 5

Type:Const
Cannot assign to constant: X
Error Line: 3 Column: 7

Call to Function: test_fn
        Invalid number of paramaters
Error Line: 22 Column: 11

Call to Function: test_fn
        Invalid number of paramaters
Error Line: 22 Column: 11
```

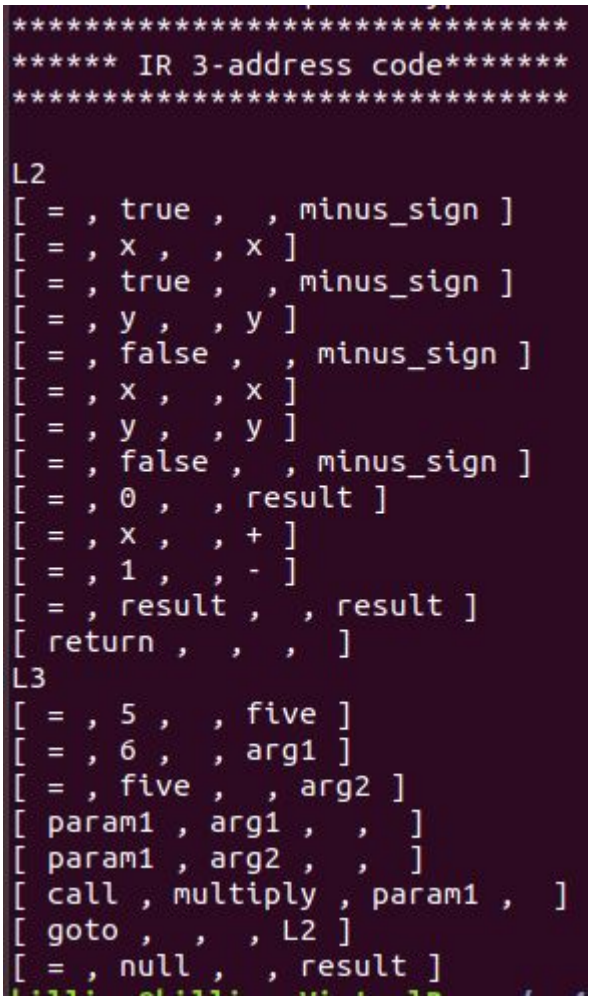
IR generation using 3-address code:

- Intermediate generation code cannot be generated unless all semantic checks are passed. The first step was to create an addressCode class to hold data for each instruction. The address code class creates Quadruples data structure to hold the data (In hindsight “Quadruples” would have been a more appropriate name for the class, doh!). A HashMap seemed to be the most appropriate form of holding the address code data. The structure of the HashMap is made up of a key that is the block label and the value is a vector of AddressCode.

```
HashMap<String,Vector<AddressCode>>
```

```
public class AddressCode
{
        String op;
        String arg1;
        String arg2;
        String result;

        public AddressCode(String iop, String iResult, String iarg1, String iarg2)
        {
                op = iop;
                arg1 = iarg1;
                arg2 = iarg2;
                result = iResult;
        }

        public AddressCode(String iop, String iResult, String iarg1 )
        {
                op = iop;
                arg1 = iarg1;
                arg2 = "";
                result = iResult;
        }

        public AddressCode(String iop, String iarg1 )
        {
                op = iop;
                arg1 = iarg1;
                arg2 = "";
                result = "";
        }

        public void printAddrCode()
        {
                System.out.println("[ "+op+" , "+arg1+" , "+arg2+" , "+result+" ]");
        }
}
```

- The visitor navigates through by starting at program and visiting each child node from program. When visitor encounters any assignments or expressions, it generates instruction sets for them and creates temporary variables to hold the data. Every time a function declaration is encountered, a new label is created.
- An example of sample IR code output can be seen below:



(4) How to run:

- To build the jjtree and compile all java files, run the bash script:

  sh compile.sh

- Which is made up of:

```
jjtree assignment_two.jjt
sleep 3
javacc assignment_two.jj
sleep 1
javac *.java
```

- To run the parser against test files, run the bash script:

  sh run_tests.sh

- Which contains the following files in the following order:
  - Test_1.ccl
  - Test_2.ccl
  - Test_3.ccl
  - Test_4.ccl
  - Test_5.ccl

- Test_6.ccl
- Test_7.ccl
- test_6_broken.ccl