DCU School of Computing

Assignment Submission

Student Name(s):	**Shane Creedon**
Student Number(s):	**15337356**
Programme:	**BSc in Computer Applications**
Project Title:	**Compiler Construction - Assignment Two**
Module code:	**CA4003**
Lecturer:	**David Sinclair**
Project Due Date:	**17th December 2018**

1 Abstract Syntax Tree Implementation

The abstract syntax tree is a format which is used for representing the syntax of a programming language. For this, assignment, I was tasked to build my own using JJTree and provide semantic checking alongside Intermediate Code Generation. Before I continue, these are some references to links that guided and helped me during my research.

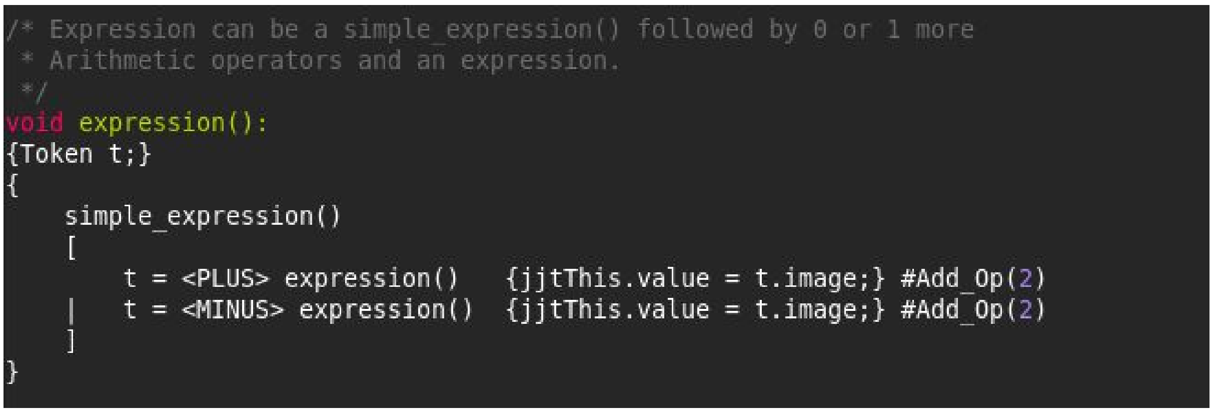
https://javacc.org/jjtree
https://www.tutorialspoint.com/compiler_design/compiler_design_semantic_analysis.htm
http://user.it.uu.se/~kostis/Teaching/KT1-11/Slides/handout08.pdf

How did I implement the Abstract Syntax Tree (AST)?
I had to do a lot of research on the concept of ASTs to grasp them fully. When it came to JJTree, in order to specify a token in our AST, I would denote a point in the grammar with the '**#**' symbol, followed by the name we would desire for our node in our tree. For example: Given there is some arithmetic operation which adds two numbers, we could specify an **Add** node in our tree as such:

Expression() [Add_Operation() Expression] **#Add_Op(2)**

The **(2)** described at the end of the statement tells JJTree to pop 2 nodes off the stack, push the new **Add_Op** node to the stack and then add the 2 popped nodes previously as it's children. For many binary operations this is very helpful.

Fig 1.0 - Expression production rule with AST node declaration



```
/* Expression can be a simple_expression() followed by 0 or 1 more
 * Arithmetic operators and an expression.
 */
void expression():
{Token t;}
{
    simple_expression()
    [
        t = <PLUS> expression()   {jjtThis.value = t.image;} #Add_Op(2)
    |   t = <MINUS> expression()  {jjtThis.value = t.image;} #Add_Op(2)
    ]
}
```

The above image is of an expression production rule I wrote in my grammar during the Syntax analyse phase of our compiler build. I was able to apply the above mentioned technique to many of my production rule expressions denoted throughout my grammar to enforce the creation of AST tree nodes.

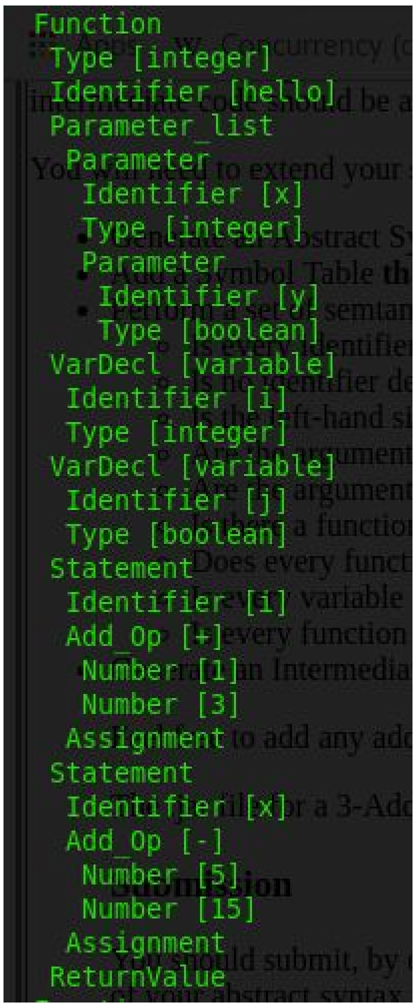
The local variable denoted as **Token t** in the above production rule acts as a mechanism for us to later extract the token we assign it to. As seen above, t is assigned to either the

<PLUS> token or the **<MINUS>** token. This allows us to obtain the '+' or '-' token during the traversal of the tree using the **visitor design pattern.** We can do this by returning **node.value**, but this will be explained later in the report.

After declaring enough of my nodes within the AST, I built the tree, with **jjtree 'Assignment.jjt'.** I then performed syntax analyse using JavaCC: **javacc 'Assignment.jj'**. Finally, we can compile all our java files and run the script. **javac *.java**
To run: **Java Assignment input.cal**.

Once I had my AST node structure set up, my AST looked as below.

Figure 1.1 - Snippet of AST Structure



```
Function
 Type [integer]
 Identifier [hello]
 Parameter_list
  Parameter
   Identifier [x]
   Type [integer]
   Parameter
    Identifier [y]
    Type [boolean]
 VarDecl [variable]
  Identifier [i]
  Type [integer]
 VarDecl [variable]
  Identifier [j]
  Type [boolean]
 Statement
  Identifier [i]
  Add_Op [+]
   Number [1]
   Number [3]
  Assignment
 Statement
  Identifier [x]
  Add_Op [-]
   Number [5]
   Number [15]
  Assignment
 ReturnValue
```

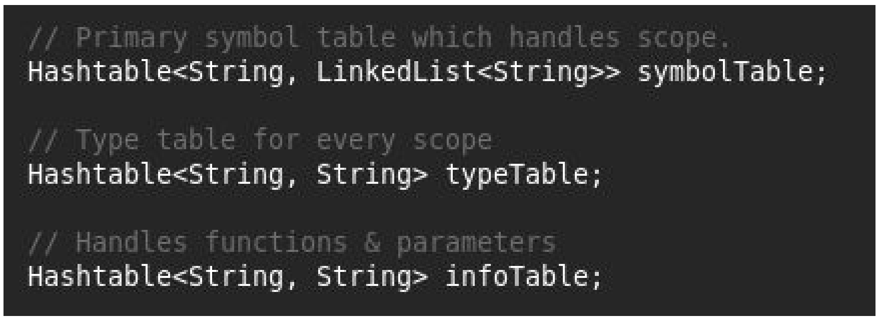
This is just a small snippet from a much larger AST print out. I decided to take a function definition to demonstrate how I decided to build my AST. Functions have 3 guaranteed children, being: Type, Identifier, and Parameter_list. Optional AST nodes include Declarations/Statements, inclusive of their children.

We can iterate through all the parameters of our function using the **[2]** index AST node which describes them. Variables have an identifier and a type, and Constants have an identifiers, a type and a value. Using the nodes from our AST, we can directly tune how our programming language works semantically. For example: If the function is of type Integer, we must require that the return value is also of that type. We can do this using the AST.

2 Symbol Table Implementation

The symbol table was a crucial part of the design and implementation of the semantic checking. I modified the Symbol Table java file **'STC.java'** as found on @David Sinclair's website. I decided to use three **Hash Tables** in order to keep track of several different factors in our program.

Fig 2.0 - 3 Symbol Table Structures

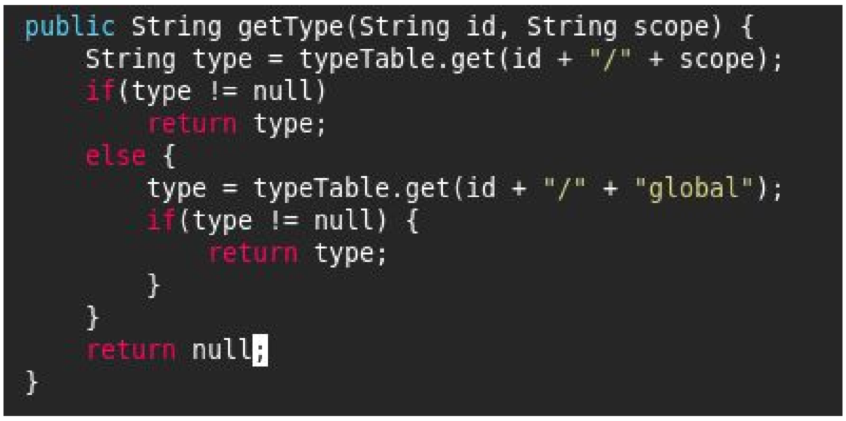


```
// Primary symbol table which handles scope.
Hashtable<String, LinkedList<String>> symbolTable;

// Type table for every scope
Hashtable<String, String> typeTable;

// Handles functions & parameters
Hashtable<String, String> infoTable;
```

As specified by the comments, the primary symbol table kept track of our scope. It functioned be using the scope as a key which would access a **LinkedList.** Each scope would have its own LinkedList of declarations and parameters.

Upon initialisation of the Symbol Table, I instantly appended the **'global'** scope to it as a key, with the value as a newly generated linkedlist. Global variables are present in every CAL program.

Fig 2.1 - Symbol Table Class: getType Method



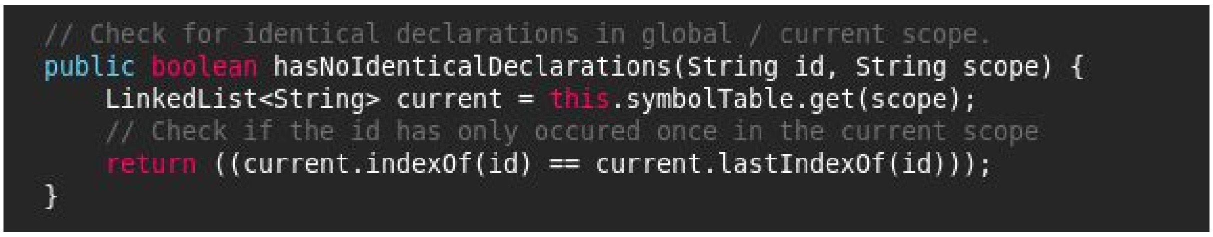
```
public String getType(String id, String scope) {
    String type = typeTable.get(id + "/" + scope);
    if(type != null)
        return type;
    else {
        type = typeTable.get(id + "/" + "global");
        if(type != null) {
            return type;
        }
    }
    return null;
}
```

Within my Symbol Table class, I developed a getType method which would take 2 parameters representing an ID & Scope. This method is the most crucial method for semantic checking. I used this method countless types within my Type Checker & Intermediate Representation visitor class. I was able to index the correct type of a node using the above method.

In the scenario where the type does not exist in the current scope, I set up a mechanism to scan the global scope if the ID does not exist locally.

By design, the local scope will take precedence over the global scope if a declaration shares the same name in both.

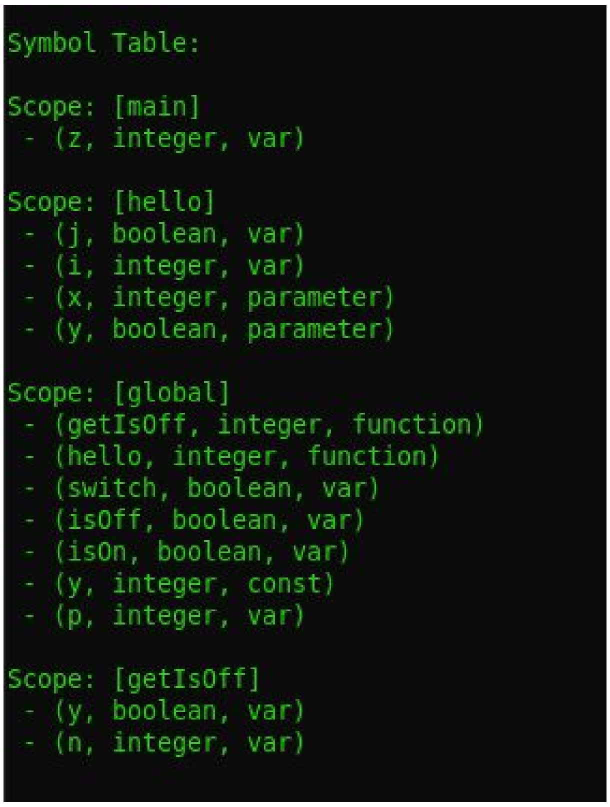
Fig 2.2 - Method which checks current scope for any duplicate declarations



```
// Check for identical declarations in global / current scope.
public boolean hasNoIdenticalDeclarations(String id, String scope) {
    LinkedList<String> current = this.symbolTable.get(scope);
    // Check if the id has only occured once in the current scope
    return ((current.indexOf(id) == current.lastIndexOf(id)));
}
```

The above method scans the scope passed in via parameter for any identical declarations by form of ID. If there are it will return false. I utilise this method in the type checking visitor to warn the user about duplicate variables.

Fig 2.3 - Symbol Table Printed



```
Symbol Table:

Scope: [main]
 - (z, integer, var)

Scope: [hello]
 - (j, boolean, var)
 - (i, integer, var)
 - (x, integer, parameter)
 - (y, boolean, parameter)

Scope: [global]
 - (getIsOff, integer, function)
 - (hello, integer, function)
 - (switch, boolean, var)
 - (isOff, boolean, var)
 - (isOn, boolean, var)
 - (y, integer, const)
 - (p, integer, var)

Scope: [getIsOff]
 - (y, boolean, var)
 - (n, integer, var)
```

From the above image, we can see how our symbol table works. Each function has its own scope. The function ID acts as the key into the scope declarations/parameters.

The **Global** scope contains the names of declarations/parameters but also the names of the functions defined throughout the program. This is a useful mechanism to have when checking if any useless functions are declared.

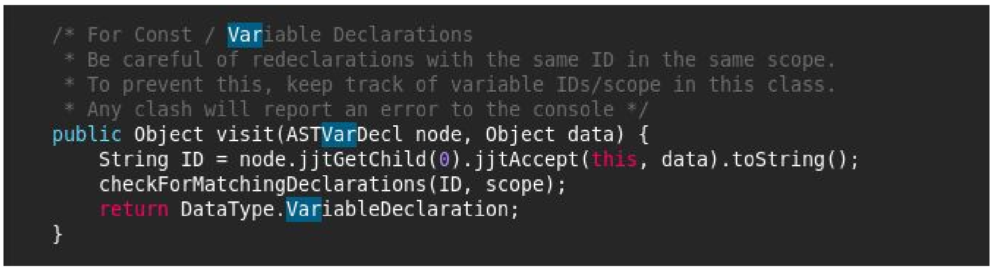
Additionally, variable with ID: **y**, is declared in both **getIsOff()** & **hello()**, yet do not conflict. They are within their own scope, within their own function so there is no reason to throw an error.

3 Semantic Checking

In this section I will describe in detail how I performed semantic checking using the visitor pattern in conjunction with an Abstract Syntax Tree. Initially, I built a **Print Visitor** to understand how AST traversal worked. After becoming confident with it i decided to tackle the **Type Check Visitor**, which describes the semantic layout of the program.

I initially started with building rules to type check my declarations. In my opinion, they are the easiest case to type check and provide a good understanding of the process.

Fig 3.0 - Variable Declarations

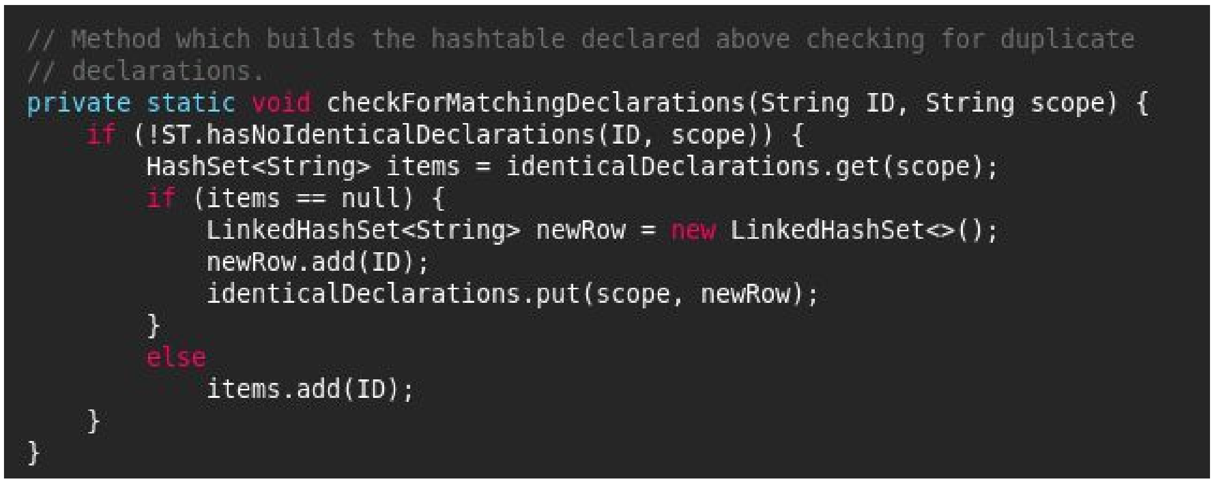


```
/* For Const / Variable Declarations
 * Be careful of redeclarations with the same ID in the same scope.
 * To prevent this, keep track of variable IDs/scope in this class.
 * Any clash will report an error to the console */
public Object visit(ASTVarDecl node, Object data) {
    String ID = node.jjtGetChild(0).jjtAccept(this, data).toString();
    checkForMatchingDeclarations(ID, scope);
    return DataType.VariableDeclaration;
}
```

The above image describes how our type checker visitor views variable declarations. As shown above by our AST terminal print, nodes in the tree with the title **VarDecl** have 2 children: An ID & A Type.
I was able to access the raw AST node titles using **node.jjtGetChild(0) or node.jjtGetChild(1),** representing the ID and Type respectively.

In order to access the actual value of the the node, they must match with a non-terminal in our language. In many cases, this just ends up being integers and identifiers.
When handling addition/subtraction or conjunction/disjunction, it is essential both nodes on either side belong to the same type.

Fig 3.1 - Method builds a list of identical declarations we've seen

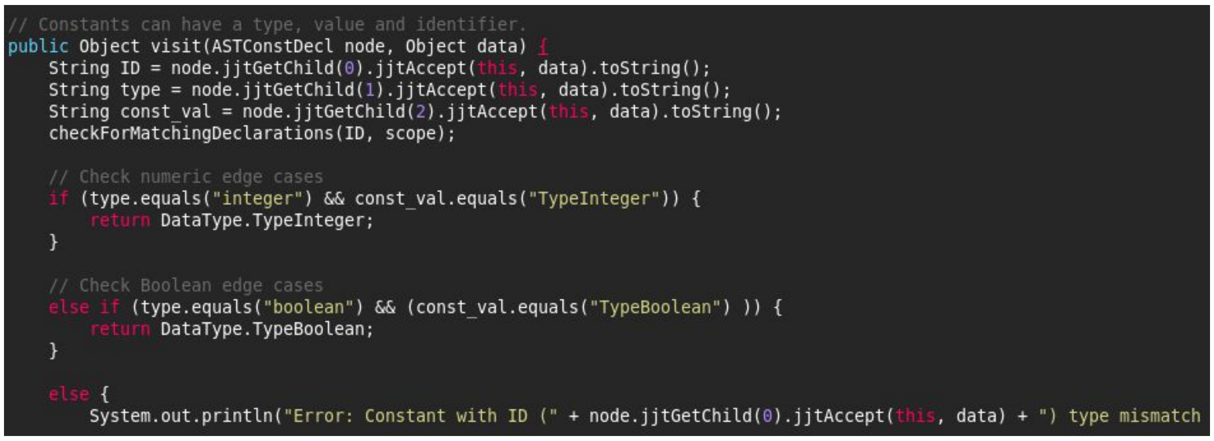


```java
// Method which builds the hashtable declared above checking for duplicate
// declarations.
private static void checkForMatchingDeclarations(String ID, String scope) {
    if (!ST.hasNoIdenticalDeclarations(ID, scope)) {
        HashSet<String> items = identicalDeclarations.get(scope);
        if (items == null) {
            LinkedHashSet<String> newRow = new LinkedHashSet<>();
            newRow.add(ID);
            identicalDeclarations.put(scope, newRow);
        }
        else
            items.add(ID);
    }
}
```

To further improve the semantic typing of the compiler as well as the semantic 'cleanness' of my code, I built a **checkForMatchingDeclarations()** method.
Everytime a new variable/constant declaration is made, I call this method to check if the declaration ID has been announced before. If it has, I append it to a hashtable declared globally at the beginning of the visitor class.

Once the program has been fully type checked, I wanted the compiler to report back any duplicate declarations for the purpose of code optimisation.

Fig 3.2 - Constant Declarations



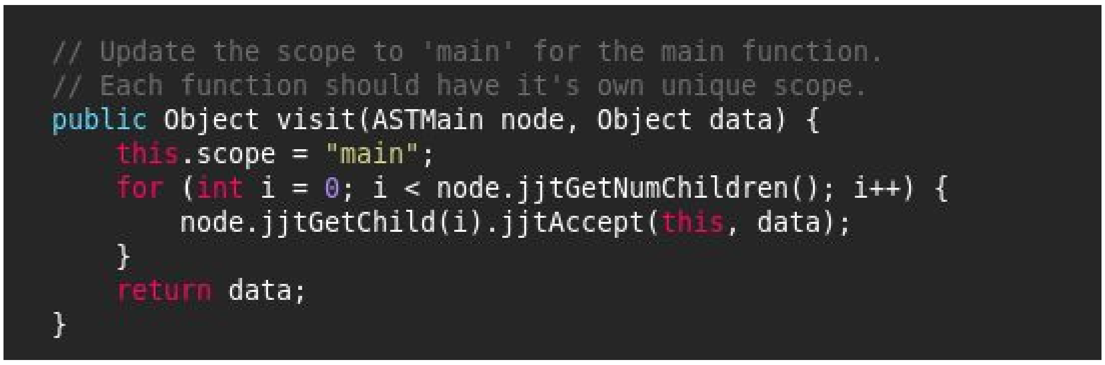
```java
// Constants can have a type, value and identifier.
public Object visit(ASTConstDecl node, Object data) {
    String ID = node.jjtGetChild(0).jjtAccept(this, data).toString();
    String type = node.jjtGetChild(1).jjtAccept(this, data).toString();
    String const_val = node.jjtGetChild(2).jjtAccept(this, data).toString();
    checkForMatchingDeclarations(ID, scope);

    // Check numeric edge cases
    if (type.equals("integer") && const_val.equals("TypeInteger")) {
        return DataType.TypeInteger;
    }

    // Check Boolean edge cases
    else if (type.equals("boolean") && (const_val.equals("TypeBoolean") )) {
        return DataType.TypeBoolean;
    }

    else {
        System.out.println("Error: Constant with ID (" + node.jjtGetChild(0).jjtAccept(this, data) + ") type mismatch
```

With Constant Declarations, I take a somewhat intuitive approach by simply looking at the type and value, ensuring they matching correctly.

Constant declarations have 3 children: An ID, a Type & a value. We can index these 3 children using **node.jjtGetChild(i).jjtAccept(this,data)** on each and storing it in a variable.

For the purposes of constant declarations, the ID is only used for checking if other declarations with the name ID already exist within the same scope. Otherwise, only the Type/Value are used for type checking. If the type and value match, we return the appropriate type, otherwise we report an error message to the console.

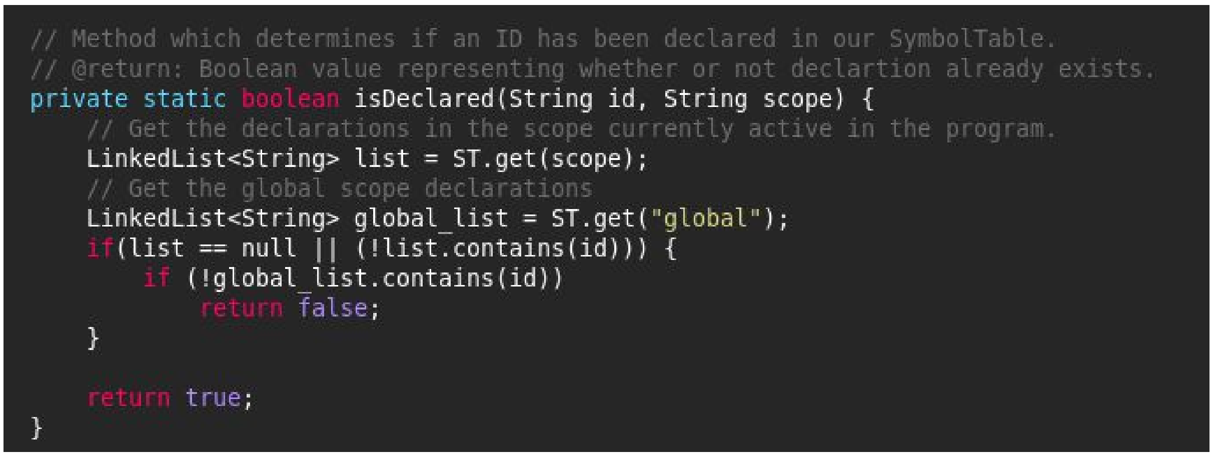
Fig 3.3 - Constant Declarations



```java
// Update the scope to 'main' for the main function.
// Each function should have it's own unique scope.
public Object visit(ASTMain node, Object data) {
    this.scope = "main";
    for (int i = 0; i < node.jjtGetNumChildren(); i++) {
        node.jjtGetChild(i).jjtAccept(this, data);
    }
    return data;
}
```

Considering, the universal function ‘main’ is within every CAL program, we can automatically set it as the scope. Within the TypeCheckVisitor Class, we have a scope variable which acts as the current scope we’re in. As soon as we reach the ASTMain node, we enter main scope. This is how functions work as well. As soon as we reach the function node, we update the scope to the functions ID.

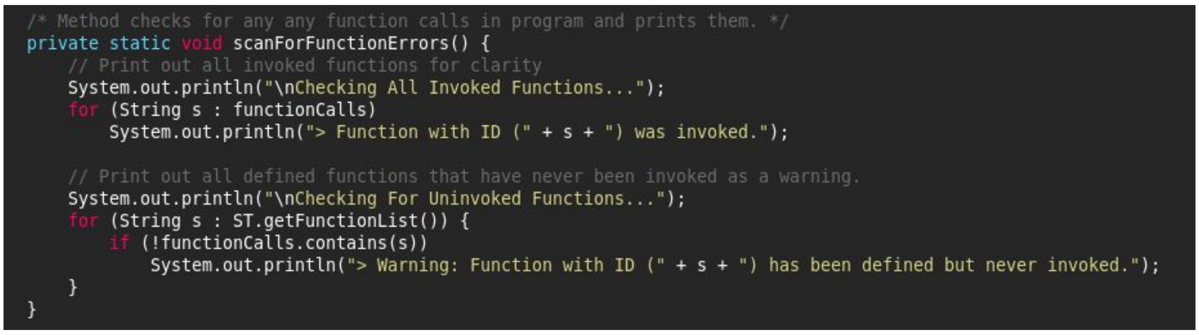
Fig 3.4 - Method checking if declaration is declared in current scope/global.



```java
// Method which determines if an ID has been declared in our SymbolTable.
// @return: Boolean value representing whether or not declartion already exists.
private static boolean isDeclared(String id, String scope) {
    // Get the declarations in the scope currently active in the program.
    LinkedList<String> list = ST.get(scope);
    // Get the global scope declarations
    LinkedList<String> global_list = ST.get("global");
    if(list == null || (!list.contains(id))) {
        if (!global_list.contains(id))
            return false;
    }

    return true;
}
```

Another helper function I wrote to help my type checker determine if a declaration has been declared already. This is an extremely useful method for extracting errorful identifiers that have no declaration previously. The method looks at your local scope for the declaration and then subsequently at the global scope. If neither contain it, it has not been declared before.

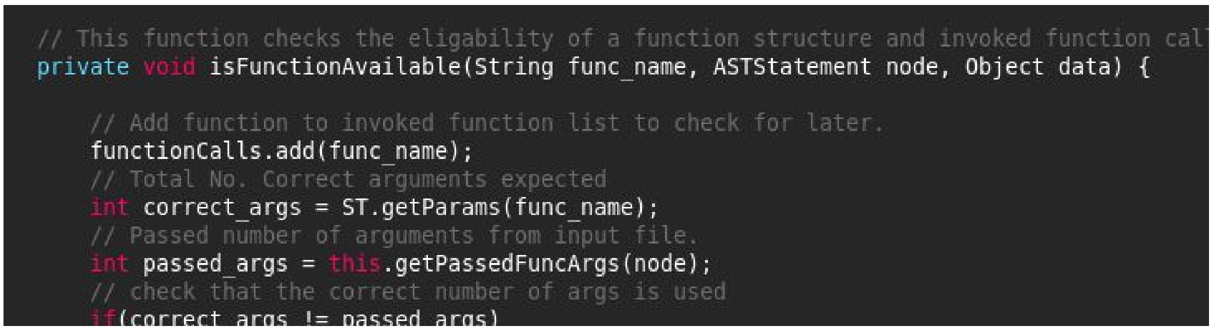
Fig 3.5 - Scans for all Invoked & Uninvoked functions



```java
/* Method checks for any any function calls in program and prints them. */
private static void scanForFunctionErrors() {
    // Print out all invoked functions for clarity
    System.out.println("\nChecking All Invoked Functions...");
    for (String s : functionCalls)
        System.out.println("> Function with ID (" + s + ") was invoked.");

    // Print out all defined functions that have never been invoked as a warning.
    System.out.println("\nChecking For Uninvoked Functions...");
    for (String s : ST.getFunctionList()) {
        if (!functionCalls.contains(s))
            System.out.println("> Warning: Function with ID (" + s + ") has been defined but never invoked.");
    }
}
```

I created the above method to help identify any functions that had been defined but had never been invoked. These functions are seemingly useless to the program and should optimally be removed or find a usage for them. It is for this reason that the compiler **warns** the user about the unused function, but does not report it as an error.

Fig 3.6 - Check argument properties of all functions



```
// This function checks the eligability of a function structure and invoked function cal
private void isFunctionAvailable(String func_name, ASTStatement node, Object data) {

    // Add function to invoked function list to check for later.
    functionCalls.add(func_name);
    // Total No. Correct arguments expected
    int correct_args = ST.getParams(func_name);
    // Passed number of arguments from input file.
    int passed_args = this.getPassedFuncArgs(node);
    // check that the correct number of args is used
    if(correct_args != passed_args)
```

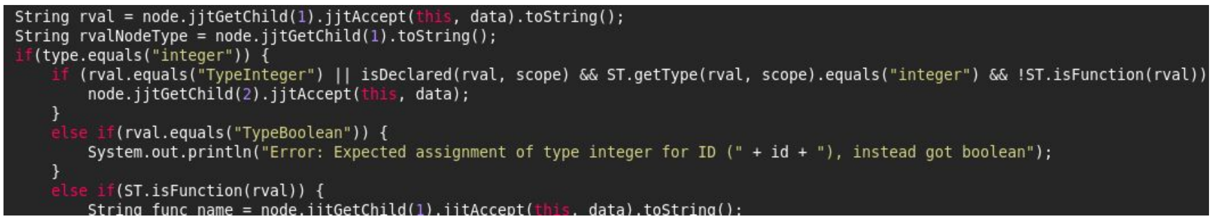
The method described above is quite important for type checking function parameters / invoked function arguments. The method is rather long so the above is just a small snippet of its functionality.

The method performs several different actions.

- It is able to correctly determine how many arguments a function call needs to match with the actual function parameters.
- It adds all function calls to a global list, similar to how declarations are added to a global list. This is for checking for useless functions later.
- This function is also is capable of type checking the arguments passed into a function call with the actual parameters of the function.

There are several other functions like this. One important one is how return types handle arguments. I have another function similar to the above which performs such requirements.

Fig 3.7 - Statements and how they work.



```
String rval = node.jjtGetChild(1).jjtAccept(this, data).toString();
String rvalNodeType = node.jjtGetChild(1).toString();
if(type.equals("integer")) {
    if (rval.equals("TypeInteger") || isDeclared(rval, scope) && ST.getType(rval, scope).equals("integer") && !ST.isFunction(rval))
        node.jjtGetChild(2).jjtAccept(this, data);
    }
    else if(rval.equals("TypeBoolean")) {
        System.out.println("Error: Expected assignment of type integer for ID (" + id + "), instead got boolean");
    }
    else if(ST.isFunction(rval)) {
        String func_name = node.jjtGetChild(1).jjtAccept(this, data).toString();
```

Statements gave me the most trouble when developing the compiler. They were somewhat tricky to approach and I spent countless hours working on their structure. To summarise how I approached them:

I split statements into two sides. Generally Identifiers would be left-leaning and values/expressions/function calls would be right-leaning. We can then evaluate if the type of the left-side match with the right-side.

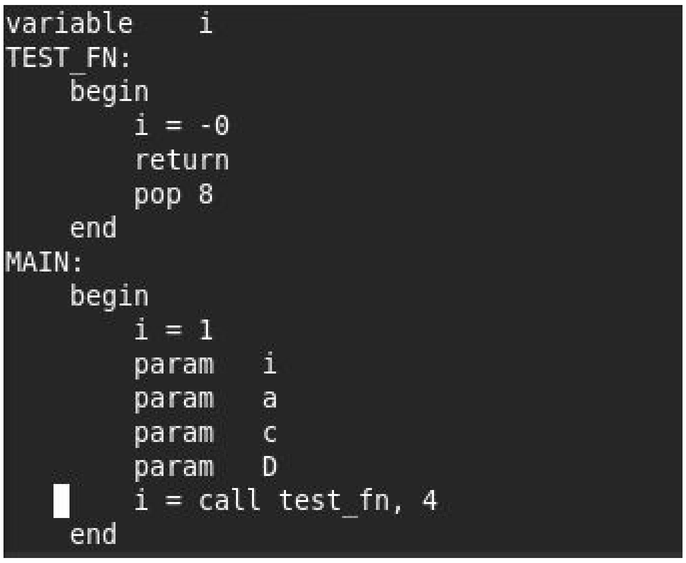
Firstly, we check if the left-side variable is already declared using our isDeclared() function mentioned previously. We can then evaluate the type checking.

Initially, we look at integers. If the left-most variable is of type integer then we would expect the right-most side to evaluate to integer as well. If this is not the case, we report an error to the terminal.

We then look at the case of boolean and the same principles apply. In the case of function calls, we apply the method described above to check argument types and how many of them there are.

4 Intermediate Code Generation

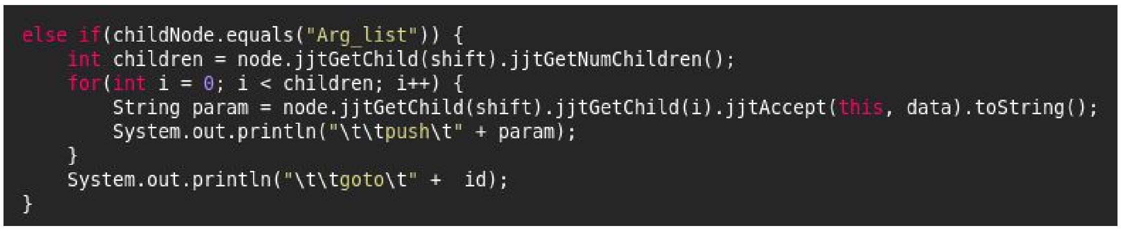
Fig 4.0 - Output of our IR Code visitor



```
variable    i
TEST_FN:
    begin
        i = -0
        return
        pop 8
    end
MAIN:
    begin
        i = 1
        param   i
        param   a
        param   c
        param   D
        i = call test_fn, 4
    end
```

Using the visitor pattern, I was able to correctly build an intermediate representation of the CAL code passed in. As can be seen, there was a function call in main with 4 parameters. test_fn(x: integer), only had 1 parameter and 1 variable declaration within, so we pop 8 bytes once the function is complete. Considering this fact, we can clearly see that test_fn does not take 4 parameters as called from main. This will print an error to the user from our Type Check Visitor

Fig 4.1 - A look at how we generate IR code



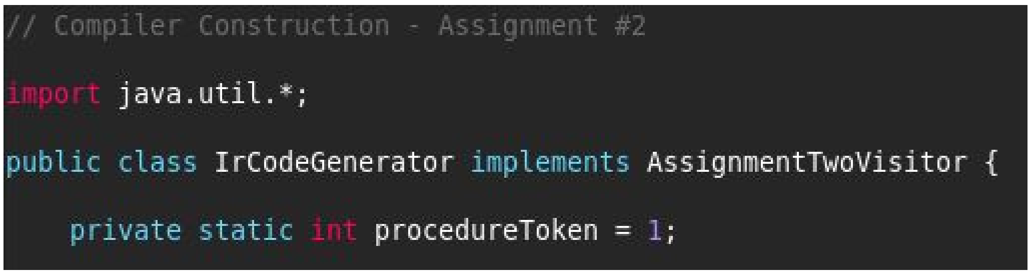
```
else if(childNode.equals("Arg_list")) {
    int children = node.jjtGetChild(shift).jjtGetNumChildren();
    for(int i = 0; i < children; i++) {
        String param = node.jjtGetChild(shift).jjtGetChild(i).jjtAccept(this, data).toString();
        System.out.println("\t\tpush\t" + param);
    }
    System.out.println("\t\tgoto\t" +  id);
}
```
.

The above image is a small snippet of how IR generation works. I decided to change the PrintStream for standard output to a file with the same name as the cal input file you pass to the system except it has the '.ir. Extension and not the '.cal' extension.

In relation to the above image, we are able to print out various high-level keywords and annotations which give meaning to our program. In the case of the above, we have an argument list which can be translated to a function call. When this happens, we push all of the parameters onto the stack and 'goto {function_id}'.

Often there would be times where I would need to adapt my AST and change the order of the tree-node structure to better suit how the IR code is generated.

Fig 4.2 - Procedure Tokens & There usefulness



```
// Compiler Construction - Assignment #2

import java.util.*;

public class IrCodeGenerator implements AssignmentTwoVisitor {

    private static int procedureToken = 1;
```

The procedure token acts as the 'program counter' of our compiler. Each time we encounter a new conditional scope is entered, (If, While, For, etc...) we increment our precedureToken counter. We can use this same concept for the '**goto**' token as well, Such as: **'goto L1', 'goto L2'** etc…

5 Conclusion

In conclusion, this module was my favourite among all of our selections this year. While the concept of how compilers work is quite intuitive and simple to understand, the level of detail and thought that go into them is astounding and I really enjoyed learning about them and discovering how I would approach building one myself.

I learned about **Lexical Analysis** through which our compiler converts our program into a stream of tokens (lexemes).
I learned about **Syntax Analysis** (parsing) through which the structure of our program is defined through the generated syntax tree.
I learned about **Semantic Analysis** which type checks our program. It checks if the parse tree previously generated follows the rules of the language. It also keeps track of identifiers and declarations and such via the symbol table.
I learned about **Intermediate Code Generation**, which of course, generates an intermediate code form that should be easier to translate into target machine code.

I learned about **Code Optimisation** which acts on the IR code. Removal of unnecessary code lines and reorder statements to improve program speed.
Finally, I learned about **Code Generation** which takes our optimised IR code and translates it to a target machine.

As a result of this module, I got to see a lot of computer science come together in an amazing way. I understand a lot more about why programming languages work the way that they do and also will undoubtedly become a better coder for having that understanding.