CA4003 Compiler Construction – Assignment 2

Assignment 2: Semantic Analysis and Intermediate Representation

Declaration on Plagiarism

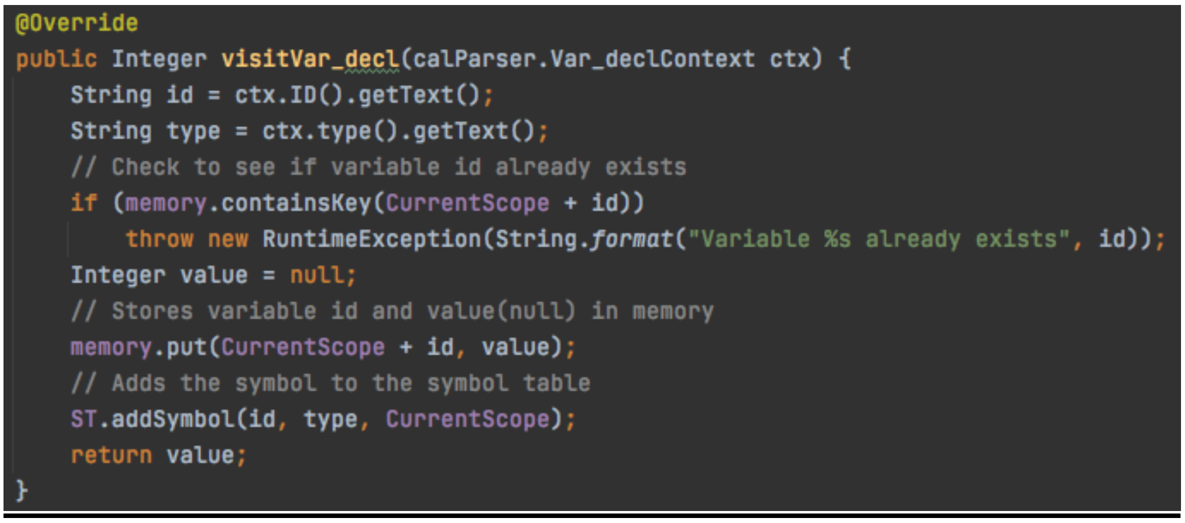
Assignment Submission Form

Name(s):	Paul McNally
Programme:	CASE 4
Module Code:	CA4003
Assignment Title:	Assignment 2: Semantic Analysis and Intermediate Representation
Submission Date:	14/12/2020
Module Coordinator:	David Sinclair

Name(s): Paul McNally Date: 14/12/2020

Semantic Analysis, visitor (EvalVisitor.java) and Symbol Table:

For implementation of the visitor interface (EvalVisitor) I started first by writing the visit variable declaration. When a variable is first declared the function will get the id of the variable as well as the type e.g., integer or Boolean. It then checks to see if the variable already exists in our memory hashmap and if so, it'll throw an error. If the variable id does not already exist, it is stored in memory, the scope of the id is also stored in memory in case there is a situation where a variable with the same id is initialized within a local scope the variable id is also added to the symbol table along with its type and current scope (the currentscope by default is set to "global")

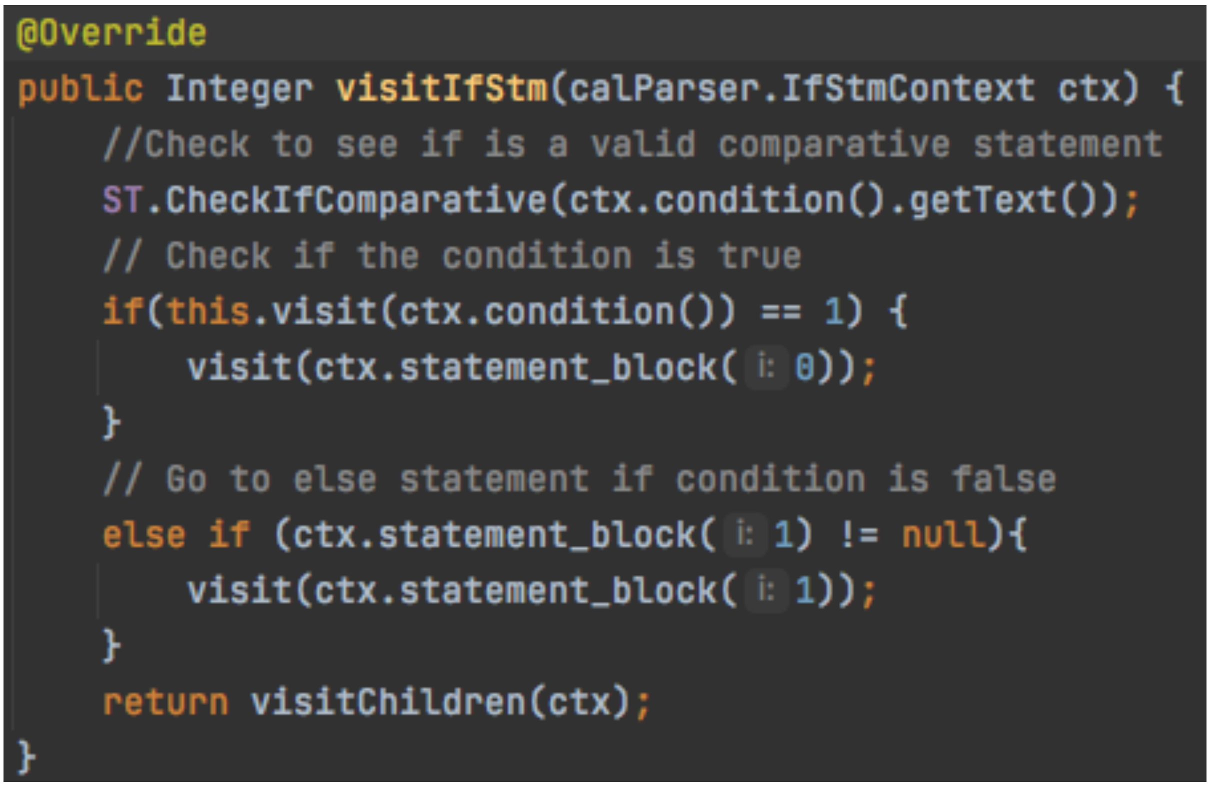


```
@Override
public Integer visitVar_decl(calParser.Var_declContext ctx) {
    String id = ctx.ID().getText();
    String type = ctx.type().getText();
    // Check to see if variable id already exists
    if (memory.containsKey(CurrentScope + id))
        throw new RuntimeException(String.format("Variable %s already exists", id));
    Integer value = null;
    // Stores variable id and value(null) in memory
    memory.put(CurrentScope + id, value);
    // Adds the symbol to the symbol table
    ST.addSymbol(id, type, CurrentScope);
    return value;
}
```

The visit constant decl function is implemented in a very similar way to the above except it also to accommodate storing functions as the value for constants and so when checking if the function type matches the constant type it calls the CompareTypeAndValue function in the symbol table.

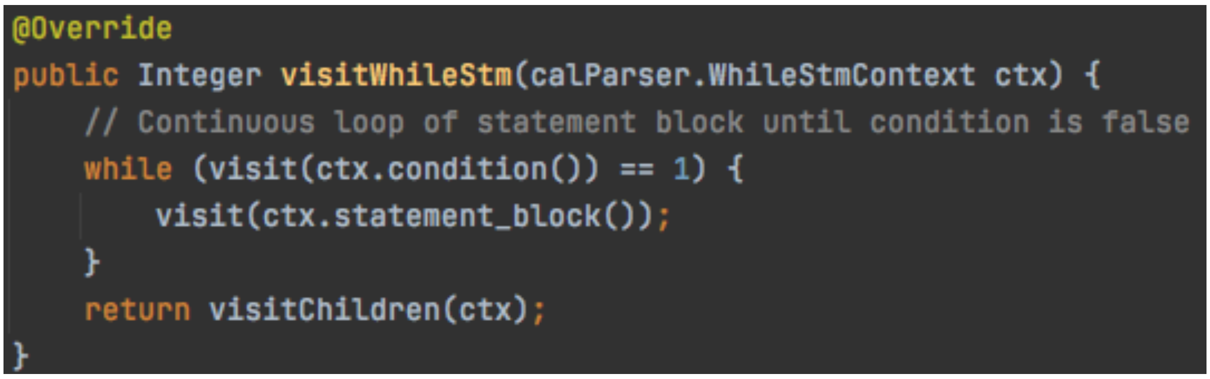
For assignments, the function checks to see if the id is stored in the memory, because if it's not, then that means it's also not in the symbol table so we can then add it to the symbol table. And finally update the id in memory.

For If/Else statements, a semantic check is calld in the SymbolTable, it checks to see whether the conditon statement is indeed a boolean and if so then it proceeds. If the condition is evaluated to be true, it proceeds with the statement block. If the condition is evaluated to be false and there is an else statement, then it proceeds with the else statement block.



```java
@Override
public Integer visitIfStm(calParser.IfStmContext ctx) {
    //Check to see if is a valid comparative statement
    ST.CheckIfComparative(ctx.condition().getText());
    // Check if the condition is true
    if(this.visit(ctx.condition()) == 1) {
        visit(ctx.statement_block( i: 0));
    }
    // Go to else statement if condition is false
    else if (ctx.statement_block( i: 1) != null){
        visit(ctx.statement_block( i: 1));
    }
    return visitChildren(ctx);
}
```

For while statements if the condition is evaluated to be true, it will carry out the statement block and continue to visit condition until the condition becomes false.

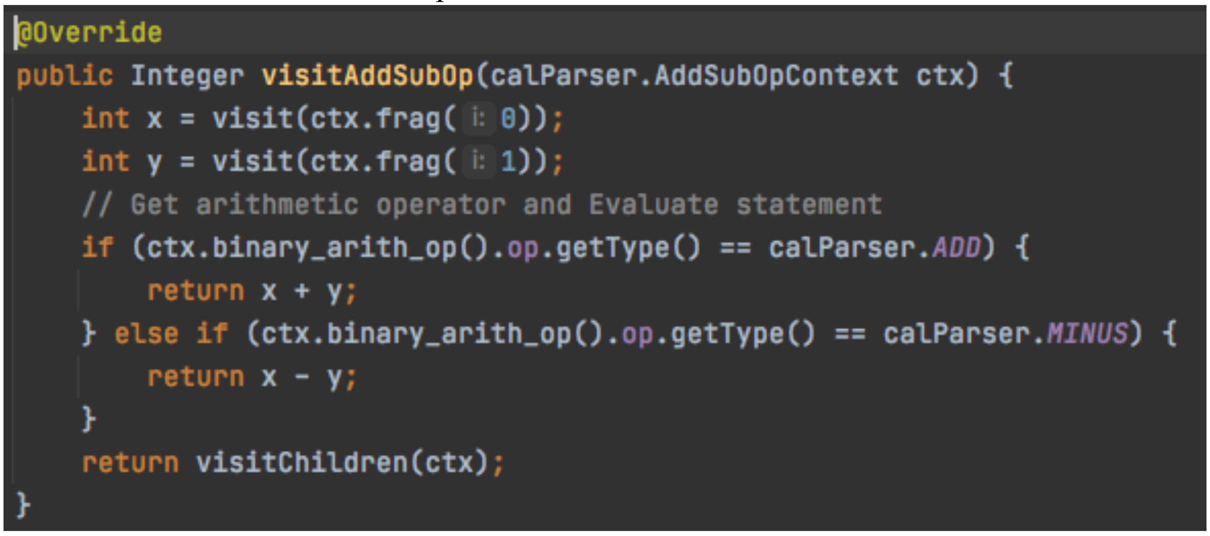


```java
@Override
public Integer visitWhileStm(calParser.WhileStmContext ctx) {
    // Continuous loop of statement block until condition is false
    while (visit(ctx.condition()) == 1) {
        visit(ctx.statement_block());
    }
    return visitChildren(ctx);
}
```

The visitCondition function is quite long as it has to accommodate for a lot of possibilities e.g. (condition, (condition), condition and/or condition, (condition and/or condition). So there are tests to see if first brackets are present, then whether there is two or one conditions. After all this it retrieves the values for of the fragments on the left and right side of the comparative operator by visiting the FragExp. Finally it uses a switch statement to find the case with the correct comparative operator and evaluates returning either true(1) or false(0).

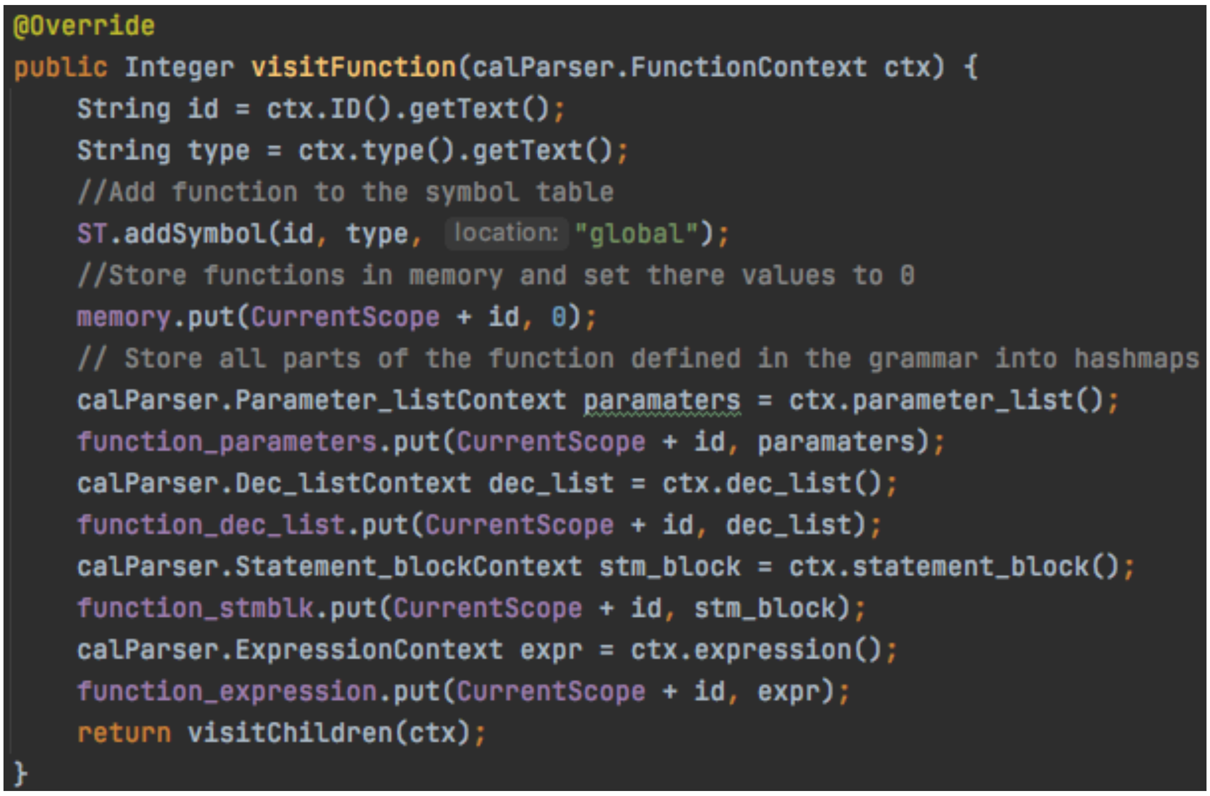
VisitFragExp and VisitFrag execute very similarly.
VisitFragExp first tests to see if the fragment is an integer by testing if the first character is a digit because therefore it will be an integer. Next it tests whether it’s true or false and lastly if it’s an id it will get the value from memory and return it.
VisitFrag does this in reverse order as we can tell straight away whether the fragment is an id.

The visitAddSubOp will get the integer values of the left and right side and then perform either an addition or subtraction operation.



```java
@Override
public Integer visitAddSubOp(calParser.AddSubOpContext ctx) {
    int x = visit(ctx.frag( i: 0));
    int y = visit(ctx.frag( i: 1));
    // Get arithmetic operator and Evaluate statement
    if (ctx.binary_arith_op().op.getType() == calParser.ADD) {
        return x + y;
    } else if (ctx.binary_arith_op().op.getType() == calParser.MINUS) {
        return x - y;
    }
    return visitChildren(ctx);
}
```

When there is a function, I created four separate hashmaps. First the function name is stored in the memory. And the four hashmaps are used so I can easily store the function parameters, dec_list, statement black and expression and retrieve them if and when the function is called in the program.

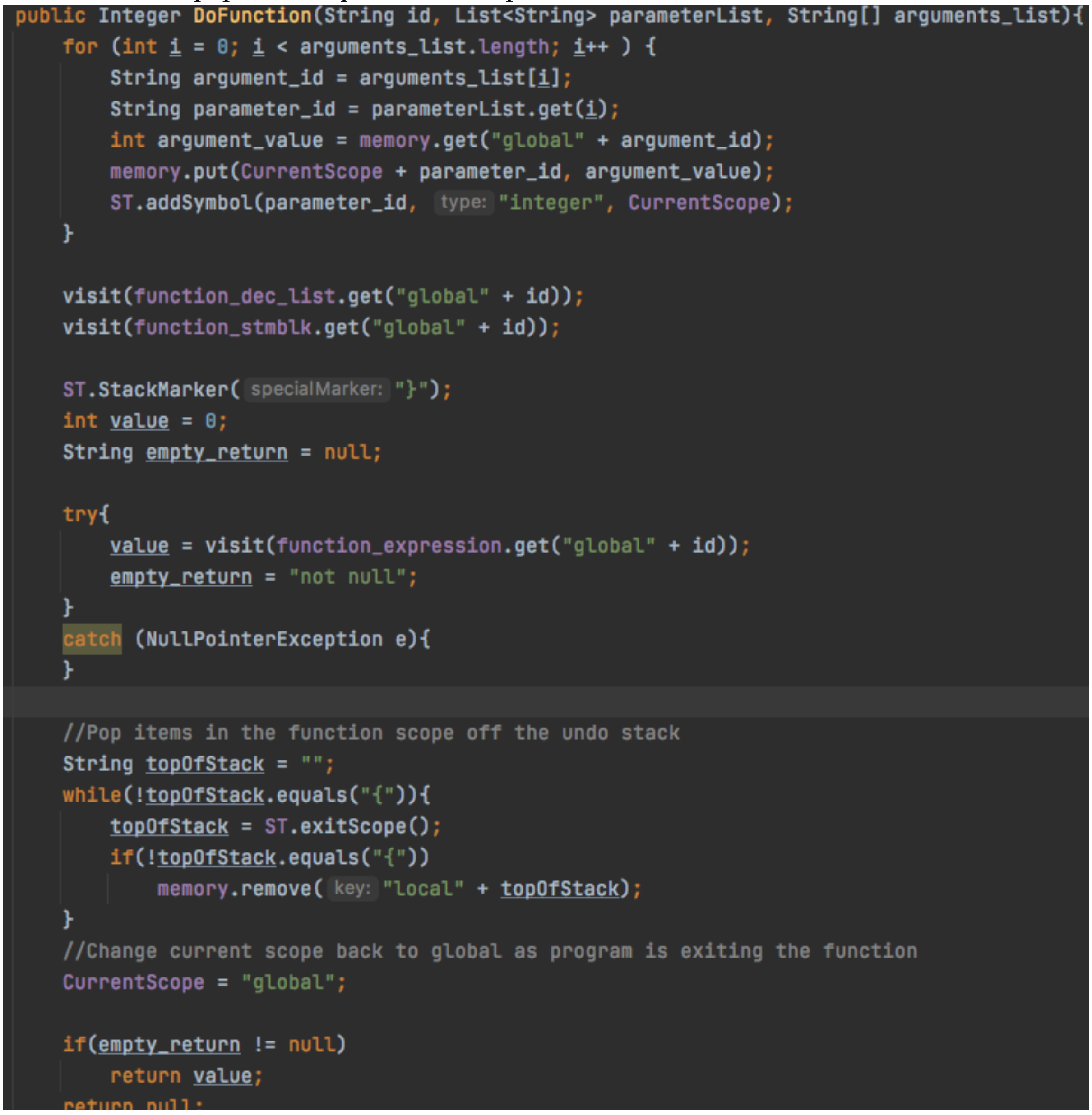


```java
@Override
public Integer visitFunction(calParser.FunctionContext ctx) {
    String id = ctx.ID().getText();
    String type = ctx.type().getText();
    //Add function to the symbol table
    ST.addSymbol(id, type, location: "global");
    //Store functions in memory and set there values to 0
    memory.put(CurrentScope + id, 0);
    // Store all parts of the function defined in the grammar into hashmaps
    calParser.Parameter_listContext paramaters = ctx.parameter_list();
    function_parameters.put(CurrentScope + id, paramaters);
    calParser.Dec_listContext dec_list = ctx.dec_list();
    function_dec_list.put(CurrentScope + id, dec_list);
    calParser.Statement_blockContext stm_block = ctx.statement_block();
    function_stmblk.put(CurrentScope + id, stm_block);
    calParser.ExpressionContext expr = ctx.expression();
    function_expression.put(CurrentScope + id, expr);
    return visitChildren(ctx);
}
```

When a function is called, depending and whether it is assigned to a variable or constant or not assigned to anything, the tree will either go to VisitArgList or VisitArgStm. Both are implemented almost identically. Both will change the current scope to local push a special marker "{" onto the undo stack so we know we are now in a local scope. Both functions will generate a parameter list and a argument list and pass these along with the function id to a function called “DoFunction” which carries out the function.

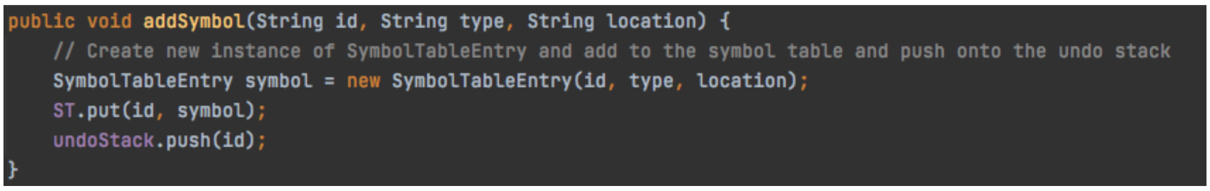
“DoFunction” adds the local function parameters to the symbol table and they are also pushed onto the stack and carries out the function statement block.

When it reaches the end of the function it pushes a marker onto the undo stack to mark the end the local scope and changes the current scope back to global. The undo stack then proceeds to pop all the local scope items off the stack and remove the items from the symbol table too until it pops of the special marker pushed onto the stack at the start.



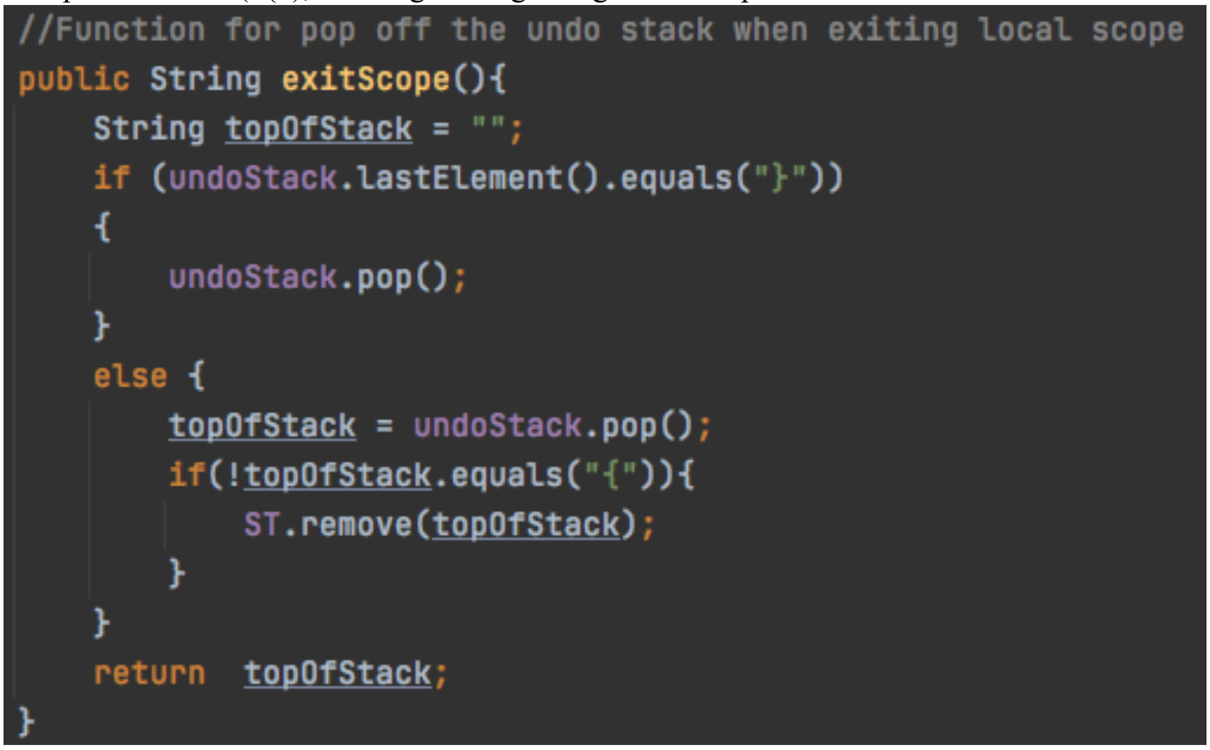
```java
public Integer DoFunction(String id, List<String> parameterList, String[] arguments_list){
    for (int i = 0; i < arguments_list.length; i++ ) {
        String argument_id = arguments_list[i];
        String parameter_id = parameterList.get(i);
        int argument_value = memory.get("global" + argument_id);
        memory.put(CurrentScope + parameter_id, argument_value);
        ST.addSymbol(parameter_id, type: "integer", CurrentScope);
    }

    visit(function_dec_list.get("global" + id));
    visit(function_stmblk.get("global" + id));

    ST.StackMarker( specialMarker: "}");
    int value = 0;
    String empty_return = null;

    try{
        value = visit(function_expression.get("global" + id));
        empty_return = "not null";
    }
    catch (NullPointerException e){
    }

    //Pop items in the function scope off the undo stack
    String topOfStack = "";
    while(!topOfStack.equals("{")){
        topOfStack = ST.exitScope();
        if(!topOfStack.equals("{"))
            memory.remove( key: "local" + topOfStack);
    }
    //Change current scope back to global as program is exiting the function
    CurrentScope = "global";

    if(empty_return != null)
        return value;
    return null;
```

When a symbol is added to the symbol table, an instance of the SymbolTableEntry class is created, this contains the symbol id, type and location(scope). The symbol id is used as the key the symbol table hashmap. The symbol id is also pushed onto the undo stack. For the SymbolTableEntry class I used the example shown on the CA4003 website.



```java
public void addSymbol(String id, String type, String location) {
    // Create new instance of SymbolTableEntry and add to the symbol table and push onto the undo stack
    SymbolTableEntry symbol = new SymbolTableEntry(id, type, location);
    ST.put(id, symbol);
    undoStack.push(id);
}
```

As already mentioned when exiting a local scope, items are popped off the undo stack until the special marker(“{“), marking the beginning of the scope is removed from the stack.

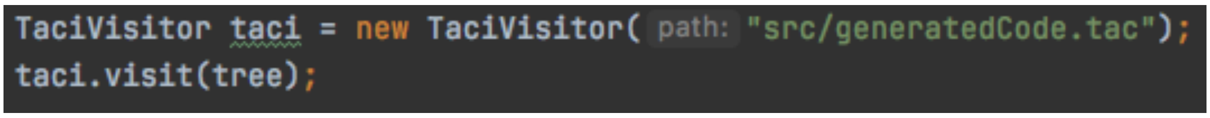


```java
//Function for pop off the undo stack when exiting local scope
public String exitScope(){
    String topOfStack = "";
    if (undoStack.lastElement().equals("}"))
    {
        undoStack.pop();
    }
    else {
        topOfStack = undoStack.pop();
        if(!topOfStack.equals("{")){
            ST.remove(topOfStack);
        }
    }
    return  topOfStack;
}
```

<u>Intermediate Representation using 3-address code:</u>

Due to time constraints, I struggled to comprehend this part of the assignment, and understand exactly how to implement and unfortunately for the visitor there is no implementation for function.

My TaciVistor is called the same way as EvalVisitor is in the cal.java file.

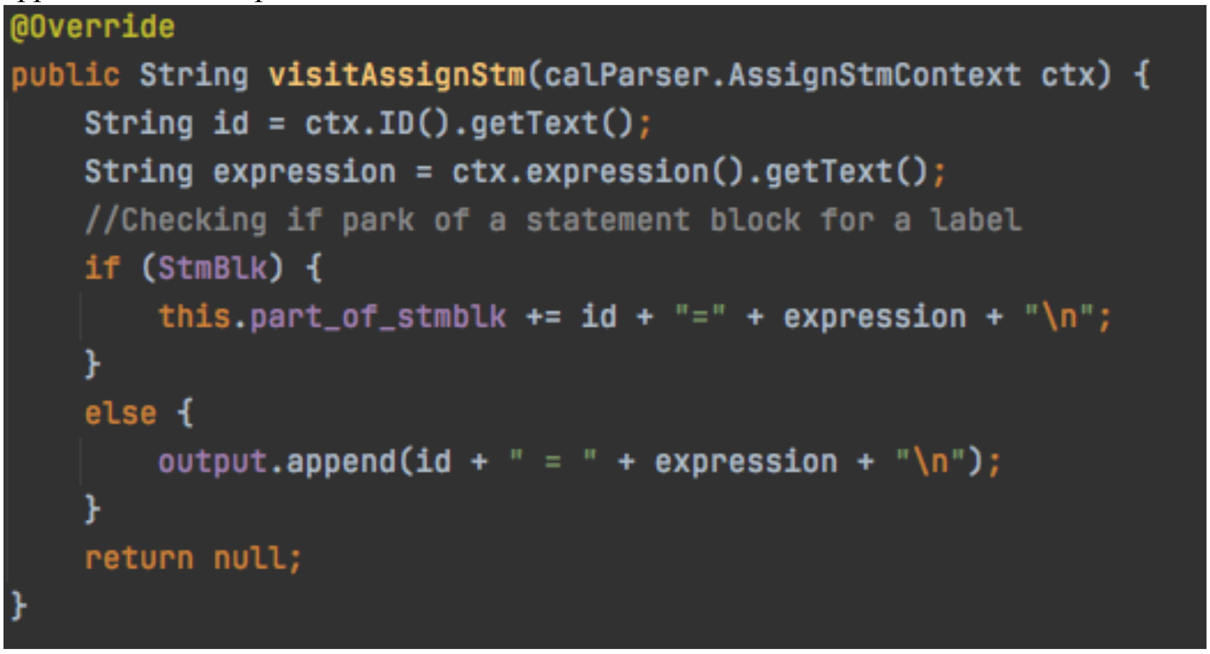


```java
TaciVisitor taci = new TaciVisitor( path: "src/generatedCode.tac");
taci.visit(tree);
```

The 3-address will be stored in a file called “generatedCode.tac” I use a java class called IrCode.java to write to this file as this makes it easier to append to the top of the file for adding Labels.

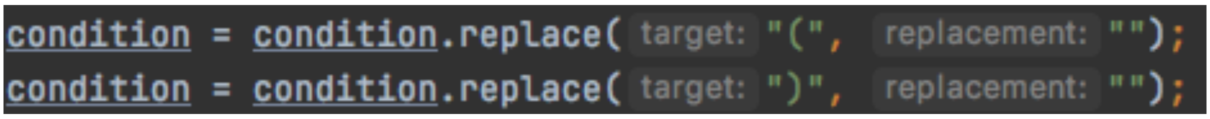
As I struggled with this part of the assignment, the only functions in the TaciVisitor are visit main assign statement, constant dec, if statement and while statements.

VisitMain simply appends the label “main:” to the start of the output file. Visit assign statement gets the id and expression and returns the statement as a string so it can be then appended to the output file.



```java
@Override
public String visitAssignStm(calParser.AssignStmContext ctx) {
    String id = ctx.ID().getText();
    String expression = ctx.expression().getText();
    //Checking if park of a statement block for a label
    if (StmBlk) {
        this.part_of_stmblk += id + "=" + expression + "\n";
    }
    else {
        output.append(id + " = " + expression + "\n");
    }
    return null;
}
```

For if statements, for each statement block a label is created and added to the top and the file along with the statement block and if the condition is true it will goto that label. If there is an else statement it will simply rewrite the if statement with “ifz” instead of “if” along with the goto else label. My visitIfstm is able to deal with brackets around the condition and also if there is 2 conditions with an & but not or.



```java
condition = condition.replace( target: "(", replacement: "");
condition = condition.replace( target: ")", replacement: "");
```

(If there is brackets around the condition, they are replaced with an empty string)

While statements have a similar format to if statements and will obviously will keep returning to the statement block label until the condition is false.