CA4003 COMPILER CONSTRUCTION

Assignment 2:
Semantic Analysis & Intermediate Representation for Cal

November 2022

Katie Clancy
19452724

1.1 DECLARATION OF PLAIGERAISM

I declare that this material, which I now submit for assessment, is entirely my 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 work. I 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 have read and understood the Assignment Regulations. I 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 or any other person for assessment on this or any other course of study.

I have read and understood the referencing guidelines found at:
https://www.dcu.ie/ovpaa/policies-and-regulations

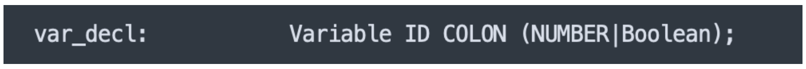
Name:	Katie Clancy
Date:	04/12/2022

<u>Re-attempt of Lexical & Syntax Analyser</u>

My attempt at assignment one didn’t go to plan, so I re-wrote some parts of it to give myself a better shot at this second assignment.

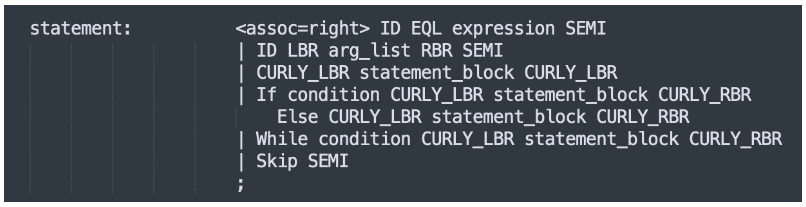
Firstly, I made some changes to my <u>grammar file</u>:

The language definition states that “variables or constants cannot be declared using the v o i d type”, and rather than using ‘type’ this time, I decided to use reserved words.



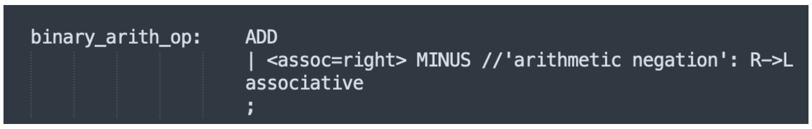
```
var_decl:           Variable ID COLON (NUMBER|Boolean);
```

I did some research in the *antlr4 reference*[1] book and discovered ‘<assoc=right>’, which is necessary if your grammar uses a right-associative ternary operator, so I implemented this as required in a couple of places throughout my grammar, like in my statement def:



```
statement:          <assoc=right> ID EQL expression SEMI
                    | ID LBR arg_list RBR SEMI
                    | CURLY_LBR statement_block CURLY_LBR
                    | If condition CURLY_LBR statement_block CURLY_RBR
                       Else CURLY_LBR statement_block CURLY_RBR
                    | While condition CURLY_LBR statement_block CURLY_RBR
                    | Skip SEMI
                    ;
```

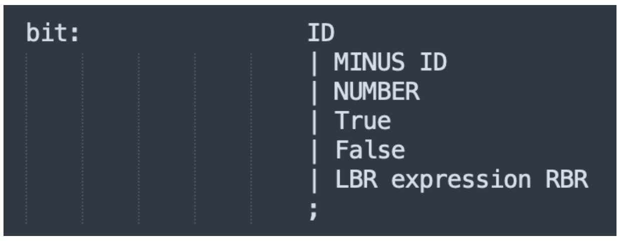
And:



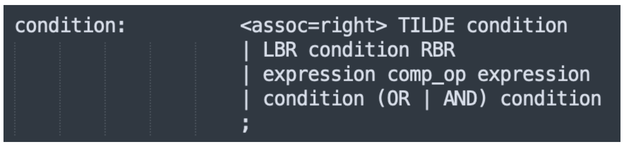
```
binary_arith_op:    ADD
                    | <assoc=right> MINUS //'arithmetic negation': R->L
                    associative
                    ;
```

Where in the language definition we are told that arithmetic negation is right-to-left associative.

‘LBR expression RBR’ is how I eliminated indirect recursion.
As we know, antlr can only handle direct left recursion, so having the recursive call after the left bracket and followed by the right bracket to achieve this was a necessary step in creating the grammar. I encountered this across a couple of rules, so I applied the same fix to them all.
(Some examples below:)

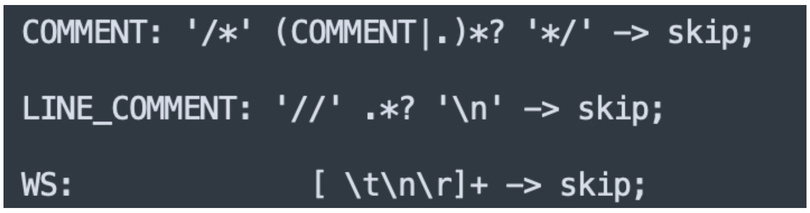


```
bit:                ID
                    | MINUS ID
                    | NUMBER
                    | True
                    | False
                    | LBR expression RBR
                    ;
```



```
condition:          <assoc=right> TILDE condition
                    | LBR condition RBR
                    | expression comp_op expression
                    | condition (OR | AND) condition
                    ;
```

I have allowed for comments with just '/* */', and also for 0 or more tokens between these comment denoters.
I used recursion to handle nested comments with an allowance for 0 or more of these.
Line comments can appear between ant two tokens, so to cater for this, I used the "wildcard" symbol of '.'. A "wildcard" can be replaced with anything so the theory is that in a line comment the '.' would be replaced with tokens.
The antlr4 reference book denotes an optional argument as '?'.



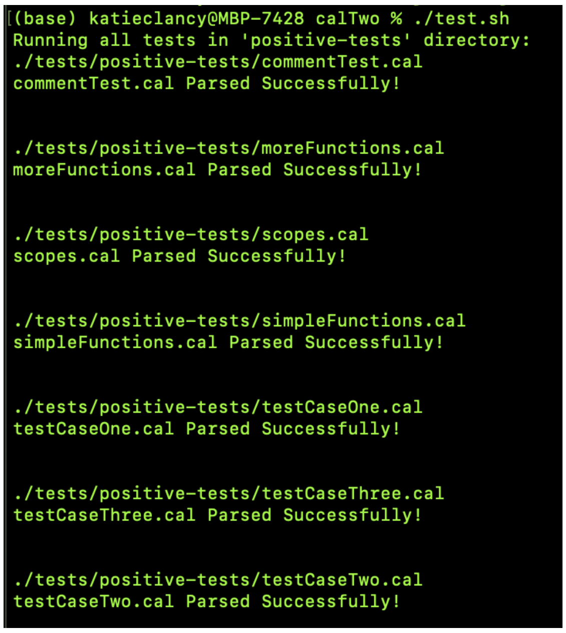
```
COMMENT: '/*' (COMMENT|.)*? '*/' -> skip;

LINE_COMMENT: '//' .*? '\n' -> skip;

WS:              [ \t\n\r]+ -> skip;
```

Next, I changed the working structure of my java parser.

I used much of the same logic from my previous attempt, but altered the way my try and catch exceptions executed.
The program takes a cal file as input and iterates through it, with an acting lexer & parser.
If the input file (test files) abides my the grammar rules, it will give a string output indicating successful parsing, otherwise an unsuccessful parsing message is given.

I used a 'test.sh' file to run all tests including tests for:
- Case insensitivity
- Simple functions
- Scope
- Core complex functions
- Comments

And got the following as output:



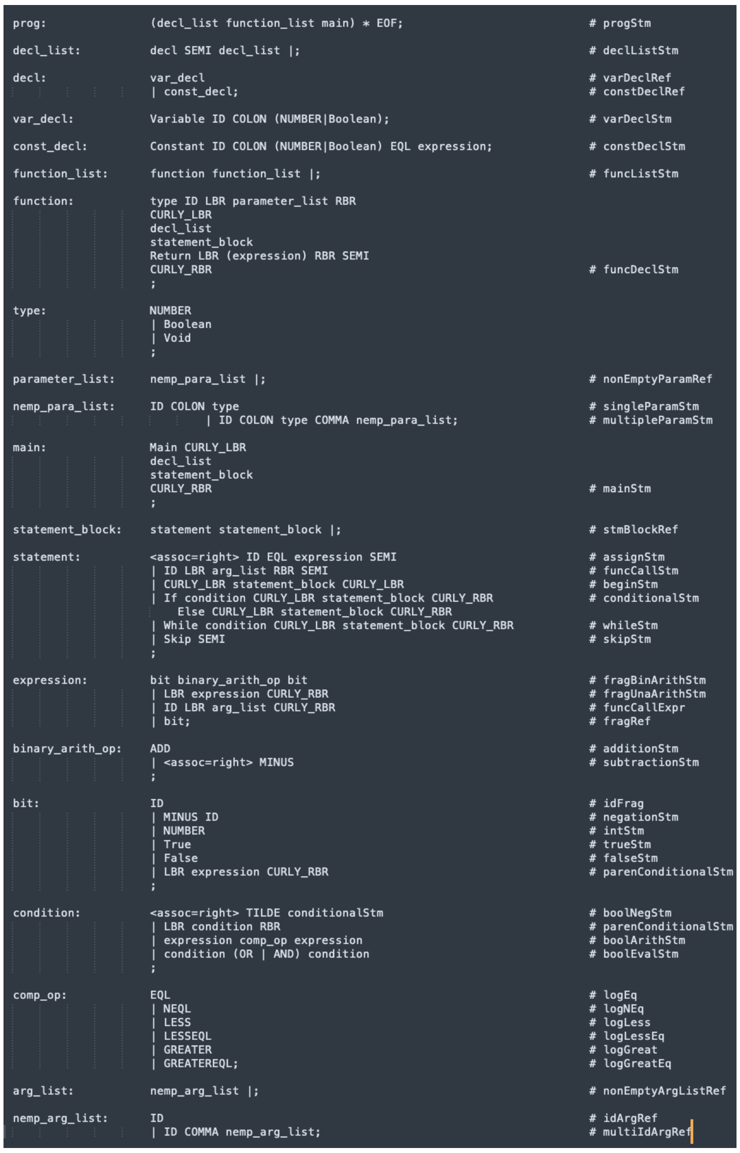
```
[(base) katieclancy@MBP-7428 calTwo % ./test.sh
Running all tests in 'positive-tests' directory:
./tests/positive-tests/commentTest.cal
commentTest.cal Parsed Successfully!


./tests/positive-tests/moreFunctions.cal
moreFunctions.cal Parsed Successfully!


./tests/positive-tests/scopes.cal
scopes.cal Parsed Successfully!


./tests/positive-tests/simpleFunctions.cal
simpleFunctions.cal Parsed Successfully!


./tests/positive-tests/testCaseOne.cal
testCaseOne.cal Parsed Successfully!


./tests/positive-tests/testCaseThree.cal
testCaseThree.cal Parsed Successfully!


./tests/positive-tests/testCaseTwo.cal
testCaseTwo.cal Parsed Successfully!
```

So I was happy that my parser was functioning as required.

Semantic Analysis

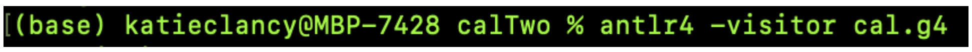
Then, I went on to begin the semantic analysis & intermediate representation phase.

Firstly, to enable precise listener events when working with parse trees (as read in the antlr4 definitive reference book), I added alternative labels to my grammar.
(I did lots of research on these labels to get a good understanding of their inner workings, which can be seen in my references list below.)



```
prog:               (decl_list function_list main) * EOF;                         # progStm

decl_list:          decl SEMI decl_list |;                                        # declListStm

decl:               var_decl                                                      # varDeclRef
                    | const_decl;                                                 # constDeclRef

var_decl:           Variable ID COLON (NUMBER|Boolean);                           # varDeclStm

const_decl:         Constant ID COLON (NUMBER|Boolean) EQL expression;            # constDeclStm

function_list:      function function_list |;                                     # funcListStm

function:           type ID LBR parameter_list RBR
                    CURLY_LBR
                    decl_list
                    statement_block
                    Return LBR (expression) RBR SEMI
                    CURLY_RBR                                                     # funcDeclStm
                    ;

type:               NUMBER
                    | Boolean
                    | Void
                    ;

parameter_list:     nemp_para_list |;                                             # nonEmptyParamRef

nemp_para_list:     ID COLON type                                                 # singleParamStm
                           | ID COLON type COMMA nemp_para_list;                  # multipleParamStm

main:               Main CURLY_LBR
                    decl_list
                    statement_block
                    CURLY_RBR                                                     # mainStm
                    ;

statement_block:    statement statement_block |;                                  # stmBlockRef

statement:          <assoc=right> ID EQL expression SEMI                          # assignStm
                    | ID LBR arg_list RBR SEMI                                    # funcCallStm
                    | CURLY_LBR statement_block CURLY_LBR                         # beginStm
                    | If condition CURLY_LBR statement_block CURLY_RBR            # conditionalStm
                       Else CURLY_LBR statement_block CURLY_RBR
                    | While condition CURLY_LBR statement_block CURLY_RBR         # whileStm
                    | Skip SEMI                                                   # skipStm
                    ;

expression:         bit binary_arith_op bit                                       # fragBinArithStm
                    | LBR expression CURLY_RBR                                    # fragUnaArithStm
                    | ID LBR arg_list CURLY_RBR                                   # funcCallExpr
                    | bit;                                                        # fragRef

binary_arith_op:    ADD                                                           # additionStm
                    | <assoc=right> MINUS                                         # subtractionStm
                    ;

bit:                ID                                                            # idFrag
                    | MINUS ID                                                    # negationStm
                    | NUMBER                                                      # intStm
                    | True                                                        # trueStm
                    | False                                                       # falseStm
                    | LBR expression CURLY_RBR                                    # parenConditionalStm
                    ;

condition:          <assoc=right> TILDE conditionalStm                            # boolNegStm
                    | LBR condition RBR                                           # parenConditionalStm
                    | expression comp_op expression                               # boolArithStm
                    | condition (OR | AND) condition                              # boolEvalStm
                    ;

comp_op:            EQL                                                           # logEq
                    | NEQL                                                        # logNEq
                    | LESS                                                        # logLess
                    | LESSEQL                                                     # logLessEq
                    | GREATER                                                     # logGreat
                    | GREATEREQL;                                                 # logGreatEq

arg_list:           nemp_arg_list |;                                              # nonEmptyArgListRef

nemp_arg_list:      ID                                                            # idArgRef
                    | ID COMMA nemp_arg_list;                                     # multiIdArgRef
```

These enable the creation of default behaviour methods for my rules in my visitor.
I created the automatically generated visitor by inputting the following command into the command line:



```
[(base) katieclancy@MBP-7428 calTwo % antlr4 -visitor cal.g4
```

This created a non-empty version of calBaseVisitor.java, with default behaviours of each method that I could then expand on.

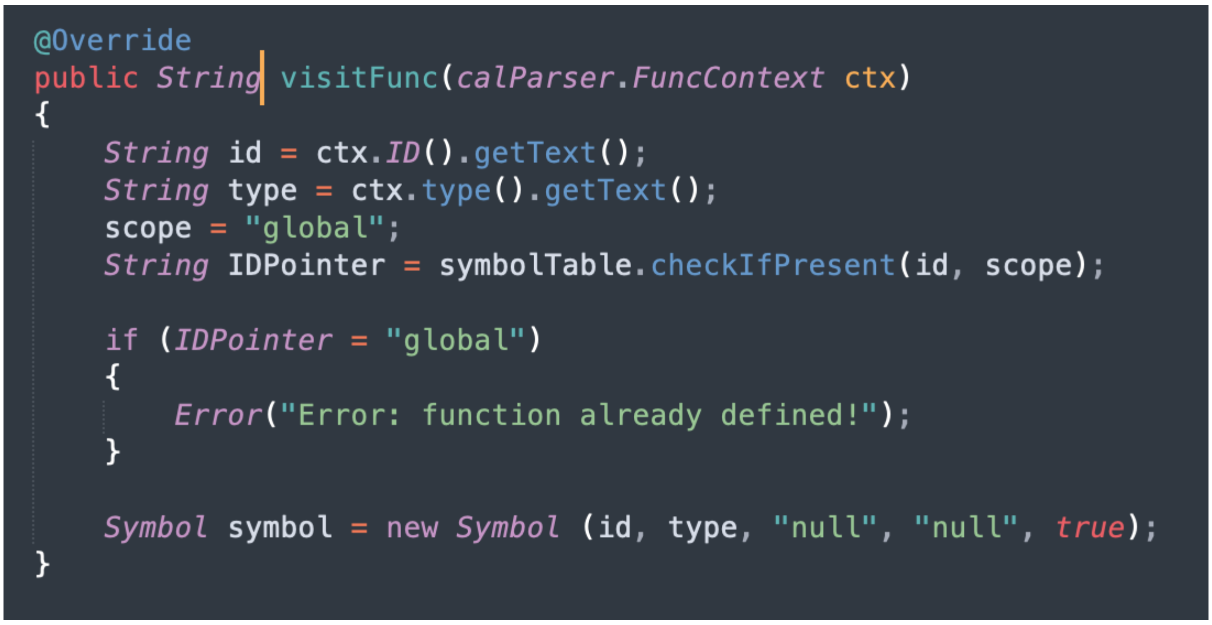
I then started putting together the code for my visitor in EvalVisitor.
This visitor program is inherent of the general visitor (which traverses the tree) - as mentioned in the antlr definitive reference book
I implemented visit method for each in EvalVisitor.java by expanding them to include instructions for rules in my grammar.

The functions for each rule give an exact path to follow, including error messages for variable and constant definitions.

These enabled me to perform semantic checks as defined in the assignment requirements, for example:

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

If this is true, an error will be given to say that the function has already been defined, as implemented in this method:



```
@Override
public String visitFunc(calParser.FuncContext ctx)
{
    String id = ctx.ID().getText();
    String type = ctx.type().getText();
    scope = "global";
    String IDPointer = symbolTable.checkIfPresent(id, scope);

    if (IDPointer = "global")
    {
        Error("Error: function already defined!");
    }

    Symbol symbol = new Symbol (id, type, "null", "null", true);
}
```

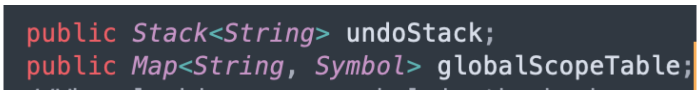
Where a function ID is in a global scope, meaning the global scope table has to be referred to to verify this. If its contained, the error is invoked. If not, its added to the table.

Some more examples of how semantic checks are implemented in my code can be seen in my code comments.

I put my symbol table in SymbolTable.java, rather than have it incorporated in the visitor class to make it tidier.

After some more research, I better understood the implementation and workings of the table so I was able to decipher that a symbol table will always make use of an *undo stack* & *hash map*, so the most efficient way to implement these was to create fields for them:

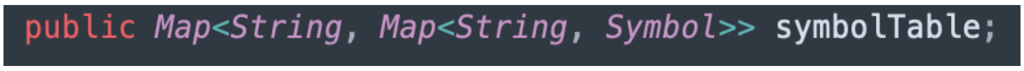
Following along with the definitive antlr reference book when creating this, I found that the function of an undo stack is to keep track of symbols in scope, and when exiting scope it defines what elements need to pop off the stack (see clearUndoStack), and the function of a hash map (essentially the symbol table) is that it stores elements in the global scope - these can be variables and functions (the global scope is the root), or the local scope - where local variables are held, live nested in functions.



```java
public Stack<String> undoStack;
public Map<String, Symbol> globalScopeTable;
```

So, I can determine whether a symbol is global or local by finding it in the hash map.

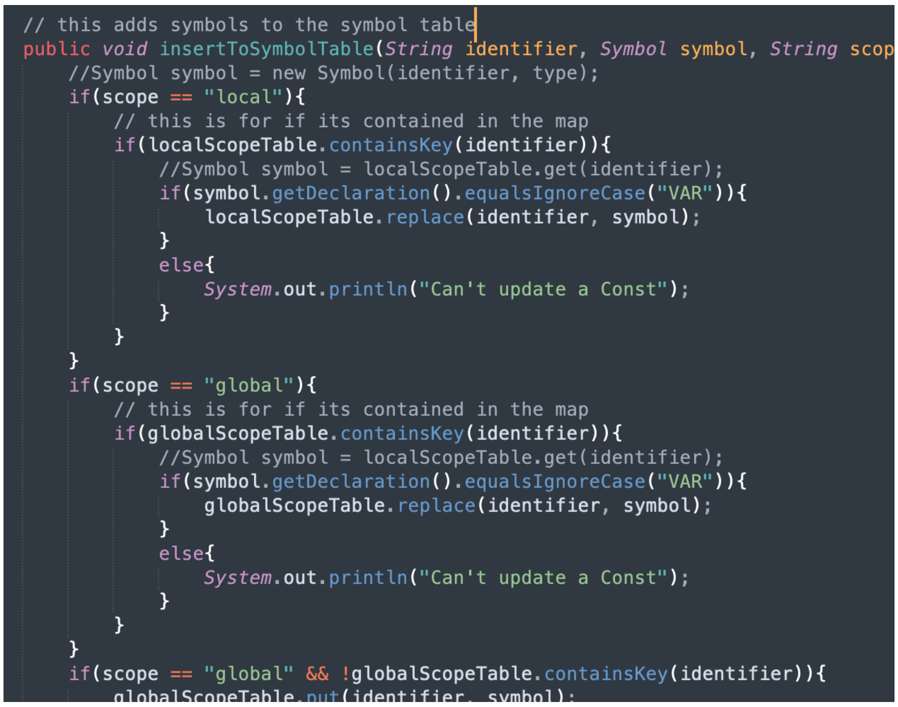
One of the main functions of the symbol table is that it will 'map' a symbol to a scope.



```java
public Map<String, Map<String, Symbol>> symbolTable;
```

Referring back to EvalVisitor, when iterating through the scope regardless of local or global, as a string, as this is the most efficient way to check the scope. Once we know what the scope is we can check for the id in the symbol table by checking if it is contained in the passed scope parameter. (Note we dont check the type - this is done in the visitor as a semantic check.)

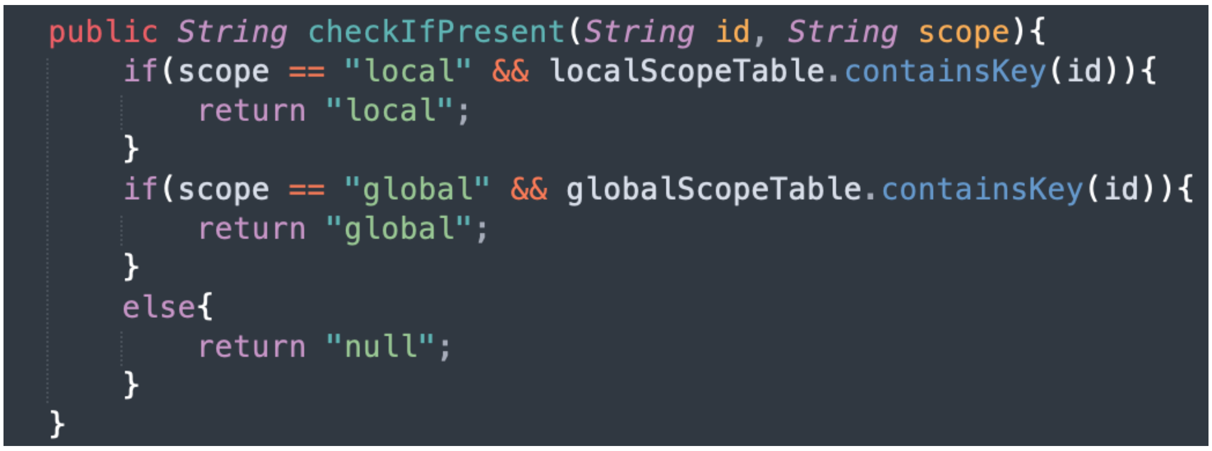
To add symbols to the table I defined insertToSymbolTable:



```java
// this adds symbols to the symbol table
public void insertToSymbolTable(String identifier, Symbol symbol, String scop
    //Symbol symbol = new Symbol(identifier, type);
    if(scope == "local"){
        // this is for if its contained in the map
        if(localScopeTable.containsKey(identifier)){
            //Symbol symbol = localScopeTable.get(identifier);
            if(symbol.getDeclaration().equalsIgnoreCase("VAR")){
                localScopeTable.replace(identifier, symbol);
            }
            else{
                System.out.println("Can't update a Const");
            }
        }
    }
    if(scope == "global"){
        // this is for if its contained in the map
        if(globalScopeTable.containsKey(identifier)){
            //Symbol symbol = localScopeTable.get(identifier);
            if(symbol.getDeclaration().equalsIgnoreCase("VAR")){
                globalScopeTable.replace(identifier, symbol);
            }
            else{
                System.out.println("Can't update a Const");
            }
        }
    }
    if(scope == "global" && !globalScopeTable.containsKey(identifier)){
        globalScopeTable.put(identifier, symbol);
```

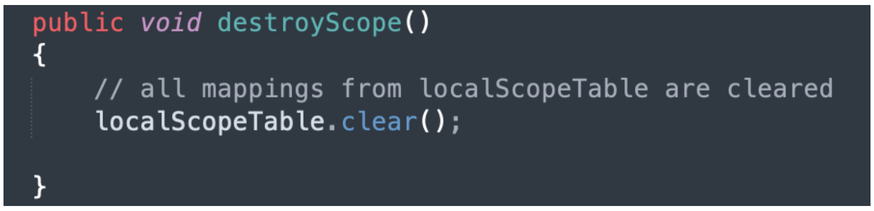
As discussed above, the first piece of logic implemented was to check the scope (global or local), which is then cross-checked with the scope contained in the ID. If it is contained, we check if it is defined as a constant or variable (variable means the table will be updated, constant means it wont be updated - this is what the implemented error methods in eval visitor deal with), and if not, it gets added to the table as it means the symbol was never added before.

If a scope contains a symbol, the checkIfPresent() method is invoked:



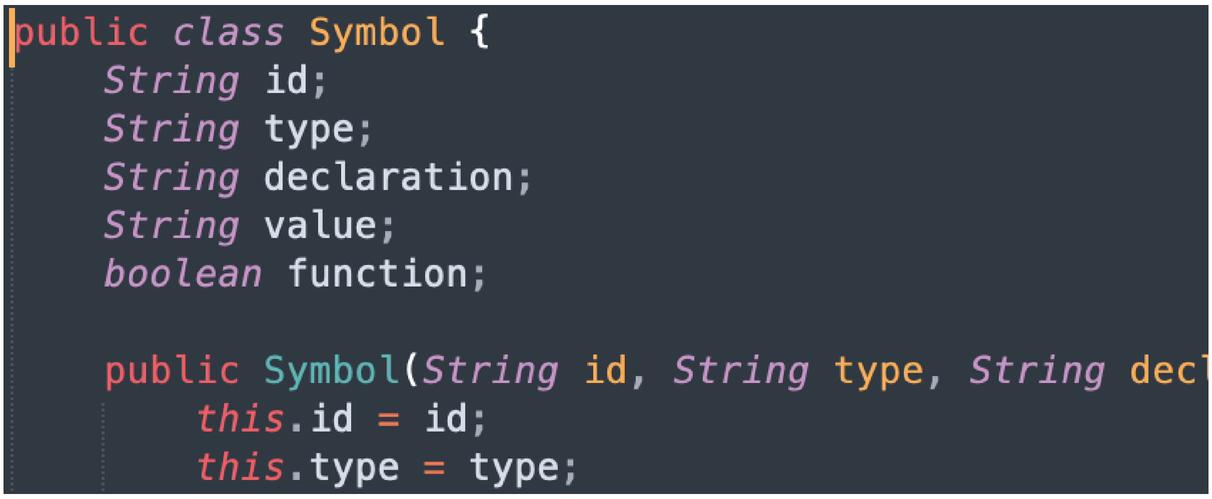
```java
public String checkIfPresent(String id, String scope){
    if(scope == "local" && localScopeTable.containsKey(id)){
        return "local";
    }
    if(scope == "global" && globalScopeTable.containsKey(id)){
        return "global";
    }
    else{
        return "null";
    }
}
```

If a special character is given as a parameter, the scope is destroyed.



```java
public void destroyScope()
{
    // all mappings from localScopeTable are cleared
    localScopeTable.clear();

}
```

Once I added all other relevant functions to SymbolTable, I began creating Symbol.java, which defines what a symbol object is made up of. This makes sure the table will map the symbol and scope (as mentioned previously) in a scope.



```java
public class Symbol {
    String id;
    String type;
    String declaration;
    String value;
    boolean function;

    public Symbol(String id, String type, String decl
        this.id = id;
        this.type = type;
```

When checking for a symbol in a table, we check the scope its in.

<u>Intermediate Representation</u>

I have included an TADVis.java file that represents an intermediate representation visitor file that is unfinished as due to the scope and difficulty of this assignment, I was unable to include. However, I did begin it and can show a few things I know about IR & TAC from some research:

- TAC is used to generate machine code.
- It is in the bottom end of mid level IR's.
- The structre looks like there is an instruction with an operator on its right side.

If I implememted the code in the file me ntioned above further, I probably would have created a process something like this:

I would have checked the structure of basic expressions and possibly converted them to a more machine code-like format which would more than likely require a seperate symbol table & visitor.

This assignment was extremely challenging and left me sleepless on many nights, especially considering my attempt at the first one was poor. But through time, dedication & a huge amount of research I believe I have given it everything I have. Although it is not perfect (and could be further improved if time allowed), I hope the evidence of vast research and genuine attempts at understanding the concepts involved (which I think I have done well), and the general great amount of work I have contributed will prevail me.

References

[1] ‘*The definitive antlr4 reference book*’ -
https://learning.oreilly.com/library/view/the-definitive-antlr/9781941222621/f_0004.xhtml

[2] *Symbol table breakdown* -
https://www.tutorialspoint.com/compiler_design/compiler_design_symbol_table.htm

[3] *More symbol table research* -
https://www.geeksforgeeks.org/symbol-table-compiler/#:~:text=Symbol%20Table%20is%20an%20important,%2C%20classes%2C%20objects%2C%20etc.

[4] *Lexer rules in Antlr* -
https://github.com/antlr/antlr4/blob/master/doc/lexer-rules.md

[5] *Alternative labels in Antlr* -
https://github.com/antlr/antlr4/blob/master/doc/lexer-rules.md

[6] *General Antlr workings* -
https://tomassetti.me/best-practices-for-antlr-parsers/#chapter6

[7] *Alternative labels & structure of visitor methods* -
https://www.youtube.com/watch?v=2o9ImGNI1uw

[8] *Working with visitors in antlr* -
https://www.youtube.com/watch?v=sgQuV_OhUGk

[9] *Visitors & listeners* -
https://tomassetti.me/listeners-and-visitors/

[10] *TAC & IR* -
https://web.stanford.edu/class/archive/cs/cs143/cs143.1128/handouts/240%20TAC%20Examples.pdf