Assignment 2: Semantic Analysis and Intermediate Representation for the CCAL Language

The aim of this assignment was to add semantic analysis checks and intermediate representation generation to the lexical and syntax analyser that I had implemented in assignment 1 for the CCAL language. Unfortunately due to the 10 week semester, assignments from other modules, my final year project and the pressure of exams coming up before christmas I was unable to complete this assignment fully. However I got an understanding of what was expected and implemented as much as I could given the constraints.

Symbol Table

The first thing I tried implementing was the symbol table that could handle scope. I researched the course notes and online material such as:
https://www.geeksforgeeks.org/symbol-table-compiler/.
https://www.tutorialspoint.com/compiler_design/pdf/compiler_design_symbol_table.pdf.
These helped me understand the architecture of a symbol table, the purpose of it and identified the different methods I would need. I decided to create a symbol table object called SymbolTable. This contained four fields:

1. undoStack - A stack to keep track of symbols when in scope and determine what elements we need to pop off when we exit a scope.
2. globalScopeTable - A hashmap that will contain the elements in the global scope.
3. localScopeTable - A hashmap that will contain the elements in the local scope.
4. symbolTable - a map that will store all of the above

My implementation of a symbol table relied on hashmaps being used. I created a map called symbolTable which would store and map a string(the scope, either global or local) to another map which would store a string(the id) and a Symbol object. I created two hashmaps for handling scope in this assignment. I realised that we could either be in the global scope or in a local scope (in a function or main). By only having to keep track of two scopes made it easier to implement later on. These maps were put into the symbolTable map. I created an undoStack as defined in the notes but never really needed to use this as I found it easier to handle scope using the maps defined above.

```
public class SymbolTable{
    //a symbol table will always have a hashmap and an undo stack
    //therefore we can create fields of these


    public Stack<String> undoStack; //an undo stack will keep track of
symbols when in scope and determine what elements we need to pop off
when we exit a scope
    public Map<String, Symbol> globalScopeTable; //This will contain
the elements in the global scope. The scopes will either be global or
local
```

```
    //When looking up a symbol in the hashmap, we can determine if the
symbol is in the local scope or the global scope
    public Map<String, Symbol> localScopeTable;
    public Map<String, Map<String, Symbol>> symbolTable; //a symbol
table will map a scope to a symbol.

    public SymbolTable(){
        globalScopeTable = new HashMap<String, Symbol>();
        localScopeTable = new HashMap<String, Symbol>();
        symbolTable = new HashMap<String, Map<String, Symbol>>();
        symbolTable.put("global", globalScopeTable);
        symbolTable.put("local", localScopeTable);
        undoStack = new Stack<String>();

    }
```

The SymbolTable object contained various methods. I first created an insert method called insertToSymbolTable(). This took an identifier, a Symbol object and a scope as parameters.

```
public void insertToSymbolTable(String identifier, Symbol symbol,
String scope){ //scope can be defined in the method in evalVisitor as
we enter a scope we can set scope to "local" or "global"
```

The aim of this method was to add symbols to the symbol table. The first thing to check when adding was what scope we are in. This would determine if the symbol was added to the global scope or the local scope. From here, we would check if the respective scope already contained the identifier. If it did, we could check if the declaration of the Symbol being added was a VAR or CONST. If it was present and a VAR then we could update the symbol table. If it was present and a CONST then we can't update the Symbol table and return an error. If the symbol wasn't ever added before, then we can add it straight to the symbol table.

I next defined a method called destroyScope(). This was the way I decided to remove elements from the local scope rather than using the undo stack. A hashmap comes with a nice method called clear() which would remove them when needed.

```
 public void destroyScope(){
        localScopeTable.clear(); //Removes all of the mappings from the
localScopeTable map

    }
```

I catered for the undo stack also when I was creating the symbol table and so have a method called clearUndoStack which would pop off elements from the undoStack until it reaches the specialCharacter given as a parameter in the method.

I also needed a getSymbol() method which would return the symbol from the correct scope given the id and scope.

One of the most important methods was the checkIfPresent() method (or lookup as it's often known as). This method returned a pointer to the scope that the symbol was contained in if it was present. It would return "global", "local" or "null" depending on if it was in the symbol table. This method was very useful throughout the implementation later on.

```
public String checkIfPresent(String id, String scope){
        //This method checks what scope the symbol is in and gives us a
pointer to that scope
        if(scope == "local" && localScopeTable.containsKey(id)){
            return "local";
        }
        if(scope == "global" && globalScopeTable.containsKey(id)){
            return "global";
        }
        else{
            return "null";
        }
    }
```

Symbol Object

I decided to create a symbol object which defined what a symbol was. I looked at the grammar in my CCAL.g4 file and checked what attributes make up a Symbol. A symbol could have an id, a type, a declaration (VAR or CONST), a value and a boolean isFunction properties. Having this object made it much easier to add, update and lookup the symbol table.

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

EvalVisitor

The next step was to create the visitor EvalVisitor.java. I extended the CCALBaseVisitor file as this was generated by ANTLR4 and I can therefore use it in my implementation. I researched online about visitors and came across this site that helped me understand why using alternative labels are the best first thing to do: https://tomassetti.me/best-practices-for-antlr-parsers/.

I placed alternative labels in my CCAL.g4 file and when compiled this generated methods associated with each label in the CCALBaseVisitor file. According to this site it makes it easier to recognize the different alternatives and to act differently depending on the alternative the parser found.

```
decl:    var_decl          #declVarDeclr
        | const_decl     #declrConstDeclr
```

The labels can be seen here “#declVarDeclr” and “#declrConstDeclr”.

These generated methods in the baseVisitor such as:

```
@Override public T visitDeclVarDeclr(CCALParser.DeclVarDeclrContext
ctx){
```

Semantic Checks

Using the evalVisitor.java file as defined above I was able to implement some semantic checks.

Is every identifier declared within scope before it is used:

For this semantic check, I will use an example where I implemented it. In my visitFragID method, I had to check if an identifier was declared within scope before it was used. In this method, I grab the ID when it is seen. We then use the symbolTable’s method checkIfPresent(id, scope) to check if the id is contained in the symbol table. This method returns a pointer to the id in the symbol table and returns either “global” or “local” if the id is contained in the symbol table and returns “null” if it doesn’t exist. If the pointer returns “null”, then we can give an error “Has not been declared”. If the pointer returns “global” then we know that the id has been declared in the global scope. If the pointer returns “local”, then we know that the id has been declared in the local scope and can proceed.

```
 @Override

    public String visitFragID(CCALParser.FragIDContext ctx){

        //When an identifier is used, it is looked up in the symbol
table

        String id = ctx.ID().getText();

        String idPointer = symbolTable.checkIfPresent(id, scope);
//this will either return "global", "local", or "null"

        if(idPointer == "global"){

            String value = symbolTable.getSymbol(id,
idPointer).getValue(); //we get the value
```

```
            String type = symbolTable.getSymbol(id,
idPointer).getType();

            String valueAndType = value + "&" + type;

            return valueAndType;

        }

        if(idPointer == "local"){

            String value = symbolTable.getSymbol(id,
idPointer).getValue(); //get the value associated with the id

            String type = symbolTable.getSymbol(id,
idPointer).getType();

            String valueAndType = value + "&" + type;

            return valueAndType;

        }



        if(idPointer == "null"){

            //throw an error as not in the symbol table

            Error("Has not been declared");

        }

        return "";



    }
```

Is the left-hand side of an assignment a variable of the correct type:

For this semantic check, I will use an example where I implemented it. In my visitStatementStm() method I checked if the left hand side of an assignment was a variable of the correct type. I realised that statements were only declared in the local scope as only functions and main had statements. This narrowed down the checking. I grabbed the ID when it is seen and use the checkIfPresent method from my symbol table. If it returns “null” then we know it hasn’t been declared (another check of the previous semantic check). If it is

contained in local then we can get the symbol object that ID associates with. We then visit the expression and get the return type stored in a variable. If the symbol.getType() equals to the expression type then we can assign the value to the symbol. Otherwise we give a “type mismatch” error.

```
@Override
    public String visitStatementStm(CCALParser.StatementStmContext
ctx){

        //<assoc=right> ID EQUALS expression SEMI (this part of the
grammar)

        //statement is only used in a function or main therefore it is
going to be in a local scope

        scope = "local";

        //Is the left-hand side of an assignment a variable of the
correct type?

        //we need to cater for this semantic check here as an id will
have been declared with a type

        //so we check if that id is present in the symbol table and
check the type if it is

        // we the check the type of the expression and if they are the
same then all good

        //otherwise error occurs as not the same type.

        String id = ctx.ID().getText();

        //first thing we do is check has the id been declared?

        String IDPointer = symbolTable.checkIfPresent(id, scope);

        if(IDPointer == "null"){ //we know that if this returns null,
the id has never been declared and so we give an error

            Error("This has not been declared");

        }

        if(IDPointer == "local"){

            //we know it's contained in the scope so we can proceed
```

```
            //next we do our type checking

            Symbol symbol = symbolTable.getSymbol(id, scope);

            String idType = symbol.getType();

            String exprType = visit(ctx.expression()); //the return
value of expression should be the type

            //System.out.println(exprType);

            //now we need to perform our semantic check

            //need to see if both types are the same

            //check if the expression has a "+" sign in the returned
value

            //if it does, we want to split on this plus sign to get the
type and the value

            String[] valueAndType = exprType.split("&"); //all of the
fragment alternatives will need to return a type and value, therefore
can split on "&" symbol for all

            String type = valueAndType[0];

            String value = valueAndType[1];

            if(idType.equalsIgnoreCase(type)){

                //we know that if the types match, then we can assign
the expression value to the id's value

                //symbol.setValue() //need to get the value of the
expression

                //got the split, now need to add to symbol if same type
and also handle if not same type

                symbol.setValue(value); //set the value of the
expression to the symbol

                return idType;

            }

            if(idType != type){
```

```
                Error("Type mismatch");

            }

        }

        return "";



    }
```

Is no identifier declared more than once in the same scope:

For this semantic check, I will use an example where I implemented it. In my visitFunction() method, I was able to check if an identifier was declared more than once in the same scope. When the id in the function is seen, it is checked with the symbol table to see if it is already in the symbol table in the global scope. If it is present, then we give an error that the function has already been defined. Otherwise, it can be added to the symbol table.

```
@Override
    public String visitFunction(CCALParser.FunctionContext ctx){
        String id = ctx.ID().getText();
        String type = ctx.type().getText();
        scope = "global"; //function id will be defined in the global
scope. We will have to check the globalScopeTable to see if it is
contained there. If it is, then this is an error as two functions with
the same id can't be defined (an error will occur)
        //otherwise if it is not in the symbol table, we can add it to
the symbol table.
        String IDPointer = symbolTable.checkIfPresent(id, scope);
//returns either global, local or null

        if(IDPointer == "global"){
            Error("Function has already been defined.");
        }

        Symbol symbol = new Symbol(id, type, "null", "null", true);
//otherwise create a symbol
```

```
        symbolTable.insertToSymbolTable(id, symbol, scope); //add it to
the symbolTable
```

Are the arguments of an arithmetic operator the integer variables or integer constants:

I was able to handle this check directly in my Symbol Table. This can be seen below:

```
if(symbol.getDeclaration().equalsIgnoreCase("VAR")){
                    localScopeTable.replace(identifier, symbol);
                }
                else{
                    System.out.println("Can't update a Const");
```

In this, we check if the integer is a VAR or a CONST. If it is a VAR, we know the variable can be updated. If it is a CONST, it can't be changed.

Intermediate Representation using 3-address code:

Unfortunately, I didn't get to implement this part of the assignment due to the constraints previously mentioned. My understanding from the notes is that in 3-address code there is at most one operator on the right hand side of an instruction. I would have to have checked expressions that looked like x = y + z - w and translated them to t1 = y + z. t2 = t1 - w. This would have involved some other visitor and a new symbol table to do the translations.