Semantic Analysis and Intermediate Representation for the CCAL Language

Table of Contents

Introduction

The purpose of this assignment was to extend my previous work by adding semantic analysis checks and intermediate representation generation to the lexical and syntax analyser I previously implemented for the CCAL language from the first assignment. I couldn't fully complete this assignment but I tried to implement this as best I could with reference to the notes and guides I found online. I first began by copying my previous assignment code to this assignment. I extended the cal.java file by initiating and calling the EvalVisitor [2, 3] to visit the tree.

```
EvalVisitor eval = new EvalVisitor();
eval.visit(tree);
```

Symbol Table

For the symbol table, I used an undo stack and a HashMap as mentioned in the notes in order to achieve the required tasks efficiently [1]. The idea for the stack is to determine whether or not the current state is in the global or local scope. According to the CAL language definition, any declaration made outside of a function is considered to be global in scope.

```
public class SymbolTable {

    // undo stack and hashtable
    private Stack<String> undoStack;
    public Map<String, Symbol> symbolTable;

    // initiating empty stack
    public SymbolTable() {
        undoStack = new Stack<String>();
        symbolTable = new HashMap<String, Symbol>();
    }
    ...
}
```

The purpose of the helper methods is to help generate a symbol table for handling scope. Some of the helper methods include:

- addId(): adding entries to the symbol table
- getId(): using the id, it should obtain a symbol from the symbol table HashMap
- intoScope(): for the special marker, I chose the dollar symbol ($). The idea is to have the special marker being pushed into the stack to indicate when we enter a local scope and for a function to have its local declarations removed after being evaluated.
- exitScope(): here, we want values to be popped off the stack so long as the special marker "$" is not reached in order to remove local declarations.

Abstract Syntax Tree

ANTLR automatically generates a parse tree when using the -visitor argument to the ANTLR CLI. The idea is to implement a visitor to walk the tree in order to achieve what this assignment is asking for. I created the file EvalVisitor.java which extends the automatically generated calBaseVisitor.

```
import java.util.*;

public class EvalVisitor extends calBaseVisitor<Integer> {
...
}
```

The idea here is to allow the EvalVisitor to access the methods from the generated base visitor from the grammar file in order to parse the tree.

Semantic Analysis

I attempted to implement some semantic checks by overriding the methods generated by ANTLR in the calParser.java file [2-4]. For instance, when checking for constant declarations, I checked if the constant id is defined by specifying the special marker which indicates if we are in local scope or not.

```
// if it's in a function it's local
if(inFunction) {
    constID = '$' + constID;
    if(memory.containsKey(constID)) {
        throw new RuntimeException("Error...constant " + constID + " is
already defined");
    }
    symbolTable.addId(constID, constType, "local", "constant");
}
// else it's global
else {
    if(memory.containsKey(constID)) {
        throw new RuntimeException("Error...constant " + constID + " is
already defined");
    }
    symbolTable.addId(constID, constType, "global", "constant");
}
```

A similar approach is taken for visiting variable declarations.

```
if(varType.equalsIgnoreCase("void")) {
    throw new RuntimeException("Error...variable " + varID + " cannot be
void.");
}

// if it's in a function it's local, else it's global
if(inFunction) {
    // since it's local, we include the special marker
    varID = '$' + varID;
    // if it already contains this, var is already defined
    if(memory.containsKey(varID)) {
        throw new RuntimeException("Error...variable " + varID + "
already defined.");
    }
    symbolTable.addId(varID, varType, "local", "variable");
}

else {
    // since it's global, we check again if it contains the special
marker
    if (memory.containsKey(varID)) {
        throw new RuntimeException("Error...variable " + varID + "
already defined.");
    }
    symbolTable.addId(varID, varType, "global", "variable");
}
```

References

[1] Sinclair, D., 2021. [online] *Semantic Analysis*. Available at: <https://www.computing.dcu.ie/~davids/courses/CA4003/CA4003_Semantic_Analysis_2p.pdf> [Accessed 22 November 2021].

[2] Newbedev.com. 2021. [online] *ANTLR4 visitor pattern on simple arithmetic example*. Available at: <https://newbedev.com/antlr4-visitor-pattern-on-simple-arithmetic-example> [Accessed 22 November 2021].

[3] Kiers, B., 2021. *How to write a antlr4 visitor*. [online] Stack Overflow. Available at: <https://stackoverflow.com/questions/65038949/how-to-write-a-antlr4-visitor> [Accessed 22 November 2021].

[4] Smirnov, P., 2021. *In ANTLR how to parse the nested function using java*. [online] Stack Overflow. Available at: <https://stackoverflow.com/questions/58339878/in-antlr-how-to-parse-the-nested-function-using-java> [Accessed 22 November 2021].