Semantic Analysis and Intermediate Representation

Plagiarism Declaration

Name(s): Janesh Sharma
Student ID(s): 17473124
Programme: CASE4
Module Code: CA4003
Assignment Title: Semantic Analysis and Intermediate Representation
Submission Date: 13/12/20
Module Coordinator: David Sinclair

I/We declare that this material, which I/We now submit for assessment, is entirely my/our 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/our work. I/We 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/We have read and understood the Assignment Regulations. I/We 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 sources cited are identified in the assignment references. This assignment, or any part of it, has not been previously submitted by me/us or any other person for assessment on this or any other course of study.

I/We have read and understood the referencing guidelines found at
http://www.dcu.ie/info/regulations/plagiarism.shtml,
https://www4.dcu.ie/students/az/plagiarism
and/or recommended in the assignment guidelines.

Name(s): _____Janesh Sharma_____ Date: _____13/12/20_____

Table Of Contents

Abstract Syntax Tree

ANTLR generates a parse tree automatically for us if we add the -visitor to antlr cli. Although strictly not an AST the parse tree that ANTLR generates automatically is close enough for our purpose. Once we have the tree we can walk the tree by implementing the visitor pattern. To Implement my evalVisitor.java class I extended upon the antlr generated calBaseVisitor and set a type of Integer for all methods to return as seen below:

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

This gives us access to the interface methods that ANTLR generates from our grammar file allowing us to parse the tree simply by implementing the methods and recursing through the nodes in the generated parse tree.

Symbol Table

To implement my symbol table table I used a HashMap and an undo stack as defined below:

```
public class SymbolTable {
    public Map<String, Symbol> ST = new HashMap<>();
    public Stack<String> stack = new Stack<>();
}
```

I use a stack of strings as that is all I need to determine whether we are in local or global scope since per the definition of the CAL language every declaration made outside of a function is global.This implementation saves memory when compared to the alternate approach of keeping a stack of HashMaps in memory throughout the runtime of the program.

We then have a couple of helper methods that perform all the necessary functions needed to generate a symbol table than can handle scope such as:

1. getSymbol() - this method keys into the hashmap and returns a Symbol
2. addSymbol() - this method adds an entry in the symbol table in a key value pair of id -> symbol. **Note:** A symbol has three properties, id, type, scope, and kind; where kind can be “constant”, “variable” or “function”.
3. enterScope() - here we push a special marker to the stack, here i chose a “#” symbol. This tells us where we entered into a local scope and thus allows us to remove any local declarations made inside a function once the function has been evaluated and exits scope.
4. exitScope() - here we repeatedly pop a value of the stack until we reach our special symbol(“#”) thereby removing all local declarations. These symbols are also removed from the symbol table and the memory HashMap in the EvalVistor.

Every declaration is added to memory, the symbol table and the stack. Local symbols are removed from all, once a functions’ local scope has ended.

Avoiding Overwriting in memory

Given a situation where you declare a variable with the same id in both local and global scope. If the id in the memory of the EvalVisitor is the same for both and a function with a local declaration of that id changes the value of that id, then the id for global scope would also be written. To avoid memory identifiers being overwritten I add a “#” to any declaration made inside a function, in this way ids of the “same” id can co-exist in memory without affecting one another. This id with a “#” in front of it is also saved to the symbol table and stack.

Semantic Analysis

To perform semantic analysis I utilised the symbol table to perform semantic analysis in the following cases, if there is an error found that error is thrown with an appropriate message logged to the console.

Constant check

A constant cannot be updated once it is declared. In the method below we get the symbol from the symbol table and check if it is a constant. This method is called in the assign statement to ensure no constant is updated. Variables can be assigned a new value once declared thus the same check is not necessary for a variable.

```
    // constant can't be updated
```

```
    public boolean checkConstError(String id) {
        Symbol symbol = getSymbol(id);
        boolean isConst = symbol.kind == "constant";
        if (!isConst)
            return true;
        throw new RuntimeException("Error: constant " + id + " can't be
updated once declared, use variable instead");
    }
```

Constant Declaration Check

The type defined in the constant is checked against the id/integer/boolean being assigned.This check is not necessary for variables as variables are only assigned values after a declaration.

Identifier Undefined Check

Here we check if an identifier is defined, if not an error is thrown as shown below.

```
      // id does not exist, thus is undefined
      throw new RuntimeException(String.format("Error: %s is undefined",
val));
```

Identifier Already Declared Check

Here we check if a variable or constant is already declared. This is checked in both local and global scope by checking if the id exists in memory, if it does then it has already been declared and an error is thrown.

```
            if (memory.containsKey(id))
                throw new RuntimeException(String.format("Error:
constant %s is already defined", id));
```

Function Declared Check

When a function is called, it is first checked if the function exists in memory, if not an error is thrown.

```
        if (!function.containsKey(id))
            throw new RuntimeException(String.format("Error: function %s
is not defined", id));
```

Correct Function Call Arguments

When a function is called a check is performed to check if the number of arguments passed to the function equals the number of parameters defined as in the function declaration as seen in the code below.

```
        if (argList.length != func.params.size())
            throw new RuntimeException(String.format("Error: function %s
call requires %d arguments but received %d", id, func.params.size(),
argList.length));
```

Intermediate Representation

To generate an intermediate representation of the CAL language to three address code I first had to work out how statements can be broken down into three address code properly for example with if and while statements.

To generate intermediate code I implemented another visitor file similarly to my EvalVisitor except this time returning a String from every visitor method.

When the IrVisitor.java file executes the three address code is saved to the /cal/src/ directory in a file called output.tac. If this output.tac file doesn not exist it is created, if it does, it is overwritten upon each subsequent execution of the program.