Parsing and Code Generation

Prof. Naga Kandasamy
ECE Department, Drexel University

This problem is due December 17, 2023, by 11:59 pm. Please submit original work.

Using the code provided in *code_generator.py* as a starting point, develop a program that accepts a simple C-style program as input and generates corresponding VM commands to execute it. Edit the provided file as desired. The program takes as input two command-line arguments: *some-file.txt*, name of the text file containing the program; and *some-file.vm*, name of the text file to output, containing the VM commands.

```
> Python code_generator_v2.py file-name.txt file-name.vm
file-name.txt: file containing the source program
file-name.vm: output file containing the relevant VM commands
```

Following is an example of the high-level code, from *prog2.txt*, that you must parse and generate VM commands for.

```
{
    var1 = 5;
    var2 = 10;
    var3 = var1 + var1 * var2;
    var4 = var1 + var3;
}
```

Your output file must encapsulate the VM commands for the given program within a function. For example, if *prog2.txt* contains the original program, the VM file must be formatted as follows:

```
function prog2 4
// Code to execute program
return
```

Note that the number of local variables within the *prog2* must be correctly listed, which in our example is four. For testing purposes, the function will be called from *main.vm* as follows:

```
function main 0
call prog2 0
return
```

Edit the *visit_Assign()* method within the *CodeGenerator* class to achieve the desired functionality. Variables `var1`, `var2`, and so on, must be stored in the local segment within the function stack in the order in which they are encountered. Use a *symbol table* to track various variables within the source code. You may also assume the availability of binary-operator functions *add*, *sub*, and *mult*. They take two input arguments from the working stack, calculate and place the result of the operation back on the stack.

Note: You need not submit the *sys.vm* and *main.vm* files.

Testing Your Code Generator

For five points each, generate VM code using your program for the following test cases: *prog1.txt*, *prog2.txt*, *prog3.txt*, and *prog4.txt*. In all cases, you may assume that the variables have global scope even though some of the test cases involve nested compound statements.

The test examples are simple enough that you should be able to check the correctness of your implementation by visually inspecting the generated code.

Submission Instructions

Submit via BBLearn, the source code for the generator.

Implementation Tips

You may add new members within the *CodeGenerator* class for the symbol table, offset location, etc. Do not use global variables for these. You may also add helper methods within *CodeGenerator* as desired.

To return the code generated by traversing the AST back to the *generate()* function within the *CodeGenerator* class, remember that the top-level node within the AST visited first is the *CompoundStatement* node. So, modify the *visit_compoundStatement()* method along these lines:

```
def visit_CompoundStatement(self, node):
    vm_code = []

    for child in node.children:
        vm_code.extend(self.visit(child))

    return vm_code
```

The above code visits the children of the compound statement one at a time and appends the code generated at each child node to *vm_code*. The final list is returned to the caller.