PROJECT 9

Assignment description

The Jack to VM compiler construction spans two assignments. This is the second assignment corresponding to the compiler, and in this assignment we extend the Jack syntax analyzer completed in the previous assignment into a full scale Jack compiler.

Objective Extend the syntax analyzer of the previous assignment into a full-scale Jack compiler. This requires the generation of executable VM code in the software modules that generate XML code.

Contract Implement code generation in the functions in charge of compiling the grammar rules defining the non-terminals: *subroutineCall* and *ifStatement*.

Resources The main tool in this assignment is the partially implemented Jack compiler written in Python. You will also need the supplied *TextComparer* utility, which allows to compare the output files generated by your analyzer to the files supplied by us.

Background

The Jack compiler supplied with this assignment was obtained extending the Jack analyzer from the previous assignment. The compiler is partially documented in sections 11.2, 11.3 and 11.4 of the text book. It is strongly suggested to have a full understanding of these sections before peeking into the code supplied with the assignment.

Details

Section 11.3 of the text book describes an implementation of a Jack compiler. You will start from this implementation and complete the compiler by implementing the code generation in the following methods in the *CompliationEngine* module:

1. *_CompileIf()*: it implements the grammar rule `ifStatement:...` .Observe that the parsing and code generation of
   `'if' '(' expression ')' '{' statements '}'`
   is already implemented. You just need to implement the compilation of the optional `else` part.
2. *_CompileCall()*: it implements the grammar rule `subroutineCall: ...`

You should complete the implementations of these routines following the order indicated above. In order to test your implementation of each one of the routines, use the *.jack* file in the corresponding folder:

1. *ElseTest*
2. *SubroutineCallTest*

Each one of these folders contains at least two numbered subfolders containing test programs that test the correctness of your implementation. We suggest to follow the order specified by the numbering on the subfolders.

Each subfolder contains a *CorrectOutput* folder with the *.vm* output file that is expected from your

compiler. You can use the *TextComparer* utility to check if your implementation behaves as expected. Your *.vm* output file need not contain the same comments as the output file provided in the CorrectOutput folder.

Hints:

1. Get acquainted with the implementation of the *vmWriter*.
    - How do you write a push and how do you specify the memory segment?
    - How do you write a call? What do you need to specify?
2. Get acquainted with the symbol table.
    - How do you retrieve 'kind' and 'index' of a variable?
    - What does the *CompilationEngine._KindToSegment(..)* function do?
3. What does the *CompilationEngine._CompileExpressionList()* return?
4. How do you retrieve the name of the class you are compiling? Hint: *self.className*

Submission:

Your project must be completed by the deadline. *All projects should be done individually*. One submission per student, in a zipped file named *projectMM-familyname-firstname.zip* including:

- CompilationEngine.py file, which should include the complete implementation of the above named functions. (OBS: If you make changes to any other of the supplied files, or if your implementation of CompilationEngine.py needs any additional files created by you, please include them in the submission.)

- Filled in Excel sheet. (Please do not change the name or extension of the file. Please answer closed (Yes/No) questions with *Yes* or *No*.)

Copying and other forms of plagiarism and cheating will be reported for disciplinary action.