# Contents

| 1 | Basic Test Results   | 2  |
|---|----------------------|----|
| 2 | AUTHORS              | 3  |
| 3 | CompilationEngine.py | 4  |
| 4 | JackCompiler         | 12 |
| 5 | JackCompiler.py      | 13 |
| 6 | JackTokenizer.py     | 14 |
| 7 | Makefile             | 19 |
| 8 | SymbolTable.py       | 20 |
| 9 | VMWriter.py          | 23 |

#### 1 Basic Test Results

```
****** FOLDER STRUCTURE TEST START *******
2
    Extracting submission...
        Extracted zip successfully
3
4
    Finding usernames...
         Submission logins are: nogafri
         Is this OK?
8
    Checking for non-ASCII characters with the command 'grep -IHPnsr [^\x00-\x7F] <dir>' ...
9
10
         No invalid characters found.
11
    ****** FOLDER STRUCTURE TEST END *******
12
13
14
    ****** PROJECT TEST START ******
15
    Running 'make'...
16
         'make' ran successfully.
17
18
    Finding JackCompiler...
19
         Found in the correct path.
20
21
    Testing translation of Main of Seven (running on a single file)...
22
23
         Testing your JackCompiler with command: './JackCompiler tst/Seven/Main.jack'...
         Testing your output with command: './VMEmulator.sh tst/Seven/Seven.tst'...
24
25
             Test passed.
26
27
    Testing ComplexArrays, where ComplexArrays is a directory...
         Testing your JackCompiler with command: './JackCompiler ComplexArrays/'...
28
29
         Testing your output with command: './VMEmulator.sh ComplexArrays/ComplexArrays.tst'...
             Test passed.
30
31
    Testing Seven, where Seven is a directory...
         Testing your JackCompiler with command: './JackCompiler Seven/'...
33
         Testing your output with command: './VMEmulator.sh Seven/Seven.tst'...
34
             Test passed.
35
36
    ****** PROJECT TEST END ******
37
38
39
40
    ****** PRESUBMISSION TESTS PASSED *******
41
42
    *******************
43
   Note: the tests you see above are all the presubmission tests
44
   for this project. The tests might not check all the different
   parts of the project or all corner cases, so write your own
46
    tests and use them!
```

# 2 AUTHORS

- nogafri
  Partner 1: Noga Friedman, noga.fri@mail.huji.ac.il, 209010479
  Remarks:

## 3 CompilationEngine.py

```
1
    This file is part of nand2tetris, as taught in The Hebrew University, and
    was written by Aviv Yaish. It is an extension to the specifications given
    [here] (https://www.nand2tetris.org) (Shimon Schocken and Noam Nisan, 2017),
    as allowed by the Creative Common Attribution-NonCommercial-ShareAlike 3.0
    Unported [License] (https://creativecommons.org/licenses/by-nc-sa/3.0/).
8
    import typing
    import JackTokenizer
9
    import SymbolTable
    import VMWriter
11
12
    class CompilationEngine:
        """Gets input from a JackTokenizer and emits its parsed structure into an
14
15
        output stream.
16
        def __init__(self, input_stream: "JackTokenizer", output_stream: VMWriter) -> None:
17
18
            Creates a new compilation engine with the given input and output. The
19
20
            next routine called must be compileClass()
21
            :param input_stream: The input stream.
            :param output_stream: The output stream.
22
23
            self.tokenizer = input_stream
            self.vm_writer = output_stream
25
            self.symbol_table = SymbolTable.SymbolTable()
26
27
            self.class_name = None
            self.subroutine_name = None
28
29
30
        def compile class(self) -> None:
             """Compiles a complete class.
31
             syntax: 'class' className '{' classVarDec* subroutineDec* '}'."""
            self.tokenizer.advance() # class
33
34
            self.tokenizer.advance() # className
            self.class_name = self.tokenizer.identifier()
35
            self.tokenizer.advance() # {
36
37
            while self.tokenizer.next_token() in ["static","field"]:
38
                self.compile class var dec()
            while self.tokenizer.next_token() in ["constructor", "function", "method"]:
39
40
                self.compile_subroutine()
            self.tokenizer.advance() # }
41
42
43
        def compile_class_var_dec(self) -> None:
             """Compiles a static declaration or a field declaration.
44
            syntax: ('static' | 'field') type varName (',' varName)* ';'."""
45
            while self.tokenizer.next_token() in ["static", "field"]:
46
47
                self.tokenizer.advance() # static / field
                kind = self.tokenizer.keyword()
                self.tokenizer.advance() # type
49
                type = self.get_type() # (int | char | boolean | className)
50
                self.tokenizer.advance() # varName
51
                name = self.tokenizer.identifier()
52
53
                self.symbol_table.define(name, type, kind)
54
                while self.tokenizer.next_token() == ",": # (, varName)*
55
                    self.tokenizer.advance() # ,
                    self.tokenizer.advance() # varName
57
                    name = self.tokenizer.identifier()
                    self.symbol_table.define(name, type, kind)
```

```
60
                 self.tokenizer.advance() # ;
61
62
         def compile_subroutine(self) -> None:
63
64
65
             Compiles a complete method, function, or constructor.
              You can assume that classes with constructors have at least one field,
66
             you will understand why this is necessary in project 11.
67
68
             syntax: ('constructor' | 'function' | 'method') ('void' | type) subroutineName '(' parameterList ')' subroutineBody
69
             self.tokenizer.advance() # constructor | function | method
70
             function_type = self.tokenizer.keyword()
71
             self.tokenizer.advance() # void / type
72
73
             self.tokenizer.advance() # subroutineName
74
             self.subroutine_name = self.class_name + "." + self.tokenizer.identifier()
             {\tt self.symbol\_table.start\_subroutine(self.subroutine\_name)} \quad \textit{\# reset the subroutine's symbol table}
75
76
             self.symbol_table.set_scope(self.subroutine_name) # set the current scope to the current subroutine's scope
             self.tokenizer.advance() # (
77
             {\tt self.compile\_parameter\_list(function\_type)}
78
              self.tokenizer.advance() # )
79
             self.compile_subroutine_body(function_type)
80
81
82
         def compile_subroutine_body(self, function_type: str) -> None:
              """Compiles the body of a subroutine.""
83
84
             self.tokenizer.advance() # {
85
             while self.tokenizer.next_token() == "var":
                 self.compile_var_dec()
86
87
             num_vars = self.symbol_table.var_count("var")
             self.vm_writer.write_function(self.subroutine_name, num_vars) # function name num_vars
88
89
90
             if function_type == "method":
                  self.vm_writer.write_push("argument", 0)
91
92
                  self.vm_writer.write_pop("pointer", 0)
93
             elif function_type == "constructor":
                 num_fields = self.symbol_table.globals_count("field")
94
                  self.vm_writer.write_push("constant", num_fields)
95
96
                  self.vm_writer.write_call("Memory.alloc", 1)
97
                  self.vm_writer.write_pop("pointer", 0)
98
             self.compile statements()
99
             self.tokenizer.advance() # }
100
             self.symbol_table.set_scope("class") # reset the current scope to class scope
101
102
103
         def compile_parameter_list(self, function_type) -> None:
              """Compiles a (possibly empty) parameter list, not including the enclosing "()".
104
             syntax: (type varName (',' type varName)*)? """
105
              if function_type == "method":
106
                 self.symbol_table.define("this", "self", "arg")
107
108
             while self.tokenizer.next_token_type() != "SYMBOL": # while more parameters are left
109
                 self.tokenizer.advance() # type
110
                  type = self.get_type() # int | char | boolean | className
111
112
                  self.tokenizer.advance() # varName
113
                 name = self.tokenizer.identifier()
                 self.symbol_table.define(name, type, "arg")
114
115
                  if self.tokenizer.next_token() == ",": # while more parameters are left
116
117
                      self.tokenizer.advance() # ,
118
119
         def compile_var_dec(self) -> None:
120
              """Compiles a var declaration.
              syntax: 'var' type varName (',' varName)* ';'."""
121
             self.tokenizer.advance() # var
122
123
             kind = self.tokenizer.kevword()
124
             self.tokenizer.advance() # type
             type = self.get_type() # int / char / boolean / className
125
             self.tokenizer.advance() # varName
126
127
             name = self.tokenizer.identifier()
```

```
128
             self.symbol_table.define(name, type, kind)
129
             while self.tokenizer.next_token() == ",": # while more variables are left
130
                  self.tokenizer.advance() # ,
131
132
                 self.tokenizer.advance() # varName
133
                 name = self.tokenizer.identifier()
134
                 self.symbol_table.define(name, type, kind)
135
136
             self.tokenizer.advance() #;
137
         def compile statements(self) -> None:
138
139
              """Compiles a sequence of statements, not including the enclosing {}".
             syntax: statement*"""
140
             while self.tokenizer.next_token() in ["let", "if", "while", "do", "return"]:
141
142
                 if self.tokenizer.next_token() == "let":
                     self.compile_let()
143
                  elif self.tokenizer.next_token() == "if":
144
                     self.compile_if()
145
                  elif self.tokenizer.next_token() == "while":
146
                      self.compile_while()
147
                  elif self.tokenizer.next_token() == "do":
148
149
                     self.compile_do()
                  elif self.tokenizer.next_token() == "return":
150
                      self.compile_return()
151
152
153
         def compile_do(self) -> None:
              """Compiles a do statement.
154
              syntax: 'do' subroutineCall ';'."""
155
             self.tokenizer.advance() # do
156
157
             self.compile_subroutine_call()
158
              self.vm_writer.write_pop("temp", 0)
             self.tokenizer.advance() # :
159
160
161
         def compile_let(self) -> None:
              """Compiles a let statement.
162
163
              syntax: 'let' varName ('[' expression ']')? '=' expression ';'."""
164
             array = False
165
             self.tokenizer.advance() # let
             self.tokenizer.advance() # varName
166
             name = self.tokenizer.identifier()
167
168
             if self.tokenizer.next_token() == "[": # varName '[' expression ']'
169
170
                 arrav = True
171
                  self.tokenizer.advance() # [
                 self.compile_expression()
172
173
                 self.tokenizer.advance()
174
                  self.compile_array(name)
             self.tokenizer.advance() # =
175
176
             self.compile_expression()
177
             if array:
                  self.vm_writer.write_pop("temp", 0)
178
179
                  self.vm_writer.write_pop("pointer", 1)
180
                  self.vm_writer.write_push("temp", 0)
181
                 self.vm_writer.write_pop("that", 0)
182
                 self.pop_variable(name)
183
184
             self.tokenizer.advance() #;
185
         def compile_while(self) -> None:
186
              """Compiles a while statement.
187
              syntax: 'while' '(' 'expression' ')' '{' statements '}'."""
188
             count = self.symbol_table.index["while"]
189
              self.symbol_table.index["while"] += 1
190
             self.vm_writer.write_label("WHILE_EXP" + str(count))
191
192
             self.tokenizer.advance() # while
193
             self.tokenizer.advance()
             self.compile_expression()
194
195
             self.vm_writer.write_arithmetic("not")
```

```
196
              self.vm_writer.write_if("WHILE_END" + str(count))
197
              self.tokenizer.advance() # )
              self.tokenizer.advance() # {
198
              self.compile_statements()
199
              self.vm_writer.write_goto("WHILE_EXP" + str(count))
200
              self.vm_writer.write_label("WHILE_END" + str(count))
201
202
              self.tokenizer.advance() # }
203
204
          def compile_return(self) -> None:
              """Compiles a return statement.
205
              syntax: 'return' expression? ';'"""
206
207
              self.tokenizer.advance() # return
208
              empty = True
             if self.is_term():
209
210
                 empty = False
                 self.compile_expression()
211
212
              if empty:
213
                  self.vm_writer.write_push("constant", 0)
              self.vm_writer.write_return()
214
215
              self.tokenizer.advance() #;
216
         def compile_if(self) -> None:
217
              """Compiles a if statement, possibly with a trailing else clause.
218
              syntax: 'if' '(' expression ')' '{' statements '}' ('else' '{' statements '}')?"""
219
              self.tokenizer.advance() # if
220
221
              self.tokenizer.advance() # (
              self.compile_expression()
222
223
              self.tokenizer.advance() # )
224
              count = self.symbol_table.index["if"]
225
              self.symbol_table.index["if"] += 1
226
              self.vm_writer.write_if("IF_TRUE" + str(count))
              self.vm_writer.write_goto("IF_FALSE" + str(count))
227
              self.vm_writer.write_label("IF_TRUE" + str(count))
228
229
              self.tokenizer.advance() # {
              self.compile_statements()
230
231
              self.tokenizer.advance() # }
              if self.tokenizer.next_token() == "else":
232
                  self.vm_writer.write_goto("IF_END" + str(count))
233
                  self.vm_writer.write_label("IF_FALSE" + str(count))
234
                  self.tokenizer.advance() # else
235
                  self.tokenizer.advance() # {
236
237
                  self.compile_statements()
                  self.tokenizer.advance() # }
238
239
                  self.vm_writer.write_label("IF_END" + str(count))
240
                  self.vm_writer.write_label("IF_FALSE" + str(count))
241
242
         def compile_expression(self) -> None:
243
              \verb|'''''Compiles| an expression.
244
245
              syntax: term (op term)*"""
              self.compile_term() # term
246
              while self.tokenizer.next_token() in ["+", "-", "*", "/", "&", "|", "<", ">", "="]:
247
248
                  self.tokenizer.advance() # op
249
                  op = self.tokenizer.symbol()
                  self.compile_term()
250
                  if op == "+":
251
                      self.vm_writer.write_arithmetic("add")
252
                  elif op == "-":
253
                      self.vm_writer.write_arithmetic("sub")
254
255
                  elif op == "*":
256
                      self.vm_writer.write_call("Math.multiply", 2)
                  elif op == "/":
257
258
                      self.vm_writer.write_call("Math.divide", 2)
                  elif op == "&":
259
                      self.vm_writer.write_arithmetic("and")
260
                  elif op == "|":
261
                     self.vm_writer.write_arithmetic("or")
262
263
                  elif op == "<":
```

```
264
                      self.vm_writer.write_arithmetic("lt")
265
                  elif op == ">":
266
                      self.vm_writer.write_arithmetic("gt")
                  elif op == "=":
267
                      self.vm_writer.write_arithmetic("eq")
268
269
270
          def compile_term(self) -> None:
               ""Compiles a term.
271
272
              This routine is faced with a slight difficulty when
              trying to decide between some of the alternative parsing rules.
273
              Specifically, if the current token is an identifier, the routing must
274
275
              distinguish between a variable, an array entry, and a subroutine call.
              A single look-ahead token, which may be one of "[", "(", or "." suffices
276
              to distinguish between the three possibilities. Any other token is not
277
278
              part of this term and should not be advanced over.
              syntax: integerConstant | stringConstant | keywordConstant | varName | varName '[' expression ']' |
279
280
               subroutineCall | '(' expression ')' | unaryOp term
281
              array = False
282
              if self.tokenizer.next_token_type() == "INT_CONST":
283
284
                  self.tokenizer.advance()
                  value = str(self.tokenizer.int_val()) # integerConstant
285
                  self.vm_writer.write_push("constant", value)
286
287
              elif self.tokenizer.next_token_type() == "STRING_CONST":
288
289
                  self.tokenizer.advance()
                  value = self.tokenizer.string_val() # stringConstant
290
291
                  self.vm_writer.write_push("constant", len(value))
                  self.vm_writer.write_call("String.new", 1)
292
293
                  for char in value:
294
                      self.vm_writer.write_push("constant", ord(char))
                      self.vm_writer.write_call("String.appendChar", 2)
295
296
297
              elif self.tokenizer.next_token() in ["true", "false", "null", "this"]:
                  self.tokenizer.advance()
298
299
                  value = self.tokenizer.keyword() # keywordConstant
                  if value == "this":
300
                      self.vm_writer.write_push("pointer", 0)
301
302
                      self.vm_writer.write_push("constant", 0)
303
                      if value == "true":
304
305
                          self.vm_writer.write_arithmetic("not")
306
              elif self.tokenizer.next_token() == "(": # '(' expression ')'
307
                  self.tokenizer.advance() # (
308
309
                  self.compile_expression()
310
                  self.tokenizer.advance()
311
              elif self.tokenizer.next_token() in ["-", "~", "^", "#"]: # unaryOp term
312
313
                  self.tokenizer.advance()
                  op = self.tokenizer.symbol() # unaryOp
314
315
                  self.compile_term()
                  if op == "-":
316
                      self.vm_writer.write_arithmetic("neg")
317
                  elif op == "~":
318
                  self.vm_writer.write_arithmetic("not")
elif op == "^":
319
320
                      self.vm_writer.write_arithmetic("shiftleft")
321
                  elif op == "#":
322
323
                      self.vm_writer.write_arithmetic("shiftright")
324
              \textbf{else:} \quad \textit{\# identifier: varName | varName '[' expression ']' | subroutineCall}
325
                  self.tokenizer.advance() # varName | subroutineName
326
                  name = self.tokenizer.identifier()
327
328
                  num_args = 0
329
                  if self.tokenizer.next_token() == "[": # varName '[' expression ']'
330
331
                      array = True
```

```
332
                      self.tokenizer.advance() # [
                      self.compile_expression()
333
                      self.tokenizer.advance() # ]
334
                      self.compile_array(name)
335
336
                  if self.tokenizer.next_token() == "(":
337
338
                      num_args += 1
                      self.vm_writer.write_push("pointer", 0)
339
340
                      self.tokenizer.advance() # (
                      num_args += self.compile_expression_list()
341
                      self.tokenizer.advance() # )
342
343
                      self.vm_writer.write_call(self.class_name + "." + name, num_args)
344
345
                  elif self.tokenizer.next_token() == ".":
346
                      self.tokenizer.advance()
                      self.tokenizer.advance()
347
348
                      second_name = self.tokenizer.identifier()
                      if name in self.symbol_table.cur_scope or name in self.symbol_table.class_scope:
349
350
                          self.push_variable(name)
                          full_name = self.symbol_table.type_of(name) + "." + second_name
351
                          num_args += 1
352
353
                      else:
                          full_name = name + "." + second_name
354
355
                      self.tokenizer.advance() # (
356
                      num_args += self.compile_expression_list()
                      self.tokenizer.advance() # )
357
                      self.vm_writer.write_call(full_name, num_args)
358
359
                  else: # varName
360
361
                      if array:
362
                          self.vm_writer.write_pop("pointer", 1)
                          self.vm_writer.write_push("that", 0)
363
364
                      else:
365
                          self.push_variable(name)
366
367
          def compile_expression_list(self) -> int:
368
              """Compiles a (possibly empty) comma-separated list of expressions.
369
370
              int: The number of expressions in the list.
371
372
373
              count = 0
             if self.is_term():
374
375
                  self.compile_expression()
376
                 count += 1
              while self.tokenizer.next_token() == ",":
377
378
                  self.tokenizer.advance() # ,
                  self.compile_expression()
379
380
                  count += 1
              return count
381
382
383
          def is_term(self) -> bool: # added
384
              """Checks if the current token is a term.
385
386
                 bool: True if the next token is a term, False otherwise.
387
388
              return self.tokenizer.next_token_type() == "INT_CONST" or self.tokenizer.next_token_type() == "STRING_CONST" or\
389
                  self.tokenizer.next_token_type() == "KEYWORD" or self.tokenizer.next_token_type() == "IDENTIFIER" or \
390
                  self.tokenizer.next_token() in ["(", "-", "~"]
391
392
393
          def get_type(self) -> str: # added
              """Gets the type of the current token.
394
395
396
              str: The type of the current token.
397
398
399
              if self.tokenizer.token_type() == "KEYWORD":
```

```
400
                 return self.tokenizer.keyword() # int / char / boolean
             else:
401
402
                 return self.tokenizer.identifier() # className
403
         def compile_subroutine_call(self) -> None: # added
404
              """Compiles a subroutine call. The function is called after the first identifier of the subroutine call.
405
             syntax: subroutineName '(' expressionList ')' | (className | varName) '.' subroutineName '(' expressionList ')'
406
407
408
             num_args = 0
             self.tokenizer.advance() # subroutineName | className | varName
409
             first_name = self.tokenizer.identifier()
410
411
             if self.tokenizer.next_token() == ".": # className | varName '.' subroutineName (chain of calls)
412
413
                 self.tokenizer.advance() # .
414
                  self.tokenizer.advance() # subroutineName
                  second_name = self.tokenizer.identifier()
415
416
                  if first_name in self.symbol_table.cur_scope or first_name in self.symbol_table.class_scope:
                      self.push_variable(first_name)
417
                      full_name = self.symbol_table.type_of(first_name) + "." + second_name
418
                     num_args += 1
419
                  else:
420
                      full_name = first_name + "." + second_name
421
422
423
             else: # subroutineName
424
                  self.vm_writer.write_push("pointer", 0)
425
                  full_name = self.class_name + "." + first_name
                 num_args += 1
426
427
             self.tokenizer.advance() # (
428
429
             num_args += self.compile_expression_list()
430
              self.vm_writer.write_call(full_name, num_args)
             self.tokenizer.advance() # )
431
432
433
         def compile_array(self, name: str) -> None: # added
              """Compiles an array by pushing the base address of the array and the index to the stack.
434
435
436
             if name in self.symbol_table.cur_scope:
                  if self.symbol_table.kind_of(name) == "var":
437
                      self.vm_writer.write_push("local", self.symbol_table.index_of(name))
438
                  elif self.symbol_table.kind_of(name) == "arg":
439
                      self.vm_writer.write_push("argument", self.symbol_table.index_of(name))
440
441
                  if self.symbol_table.kind_of(name) == "field":
442
                      self.vm_writer.write_push("this", self.symbol_table.index_of(name))
443
                  elif self.symbol_table.kind_of(name) == "static":
444
                     self.vm_writer.write_push("static", self.symbol_table.index_of(name))
445
446
              self.vm_writer.write_arithmetic("add")
447
448
         def push_variable(self, name: str) -> None: # added
449
               ""Pushes the variable to the stack.
450
451
             name (str): The name of the variable.
452
453
454
             if name in self.symbol_table.cur_scope:
                 if self.symbol_table.kind_of(name) == "var":
455
                      self.vm_writer.write_push("local", self.symbol_table.index_of(name))
456
                  elif self.symbol_table.kind_of(name) == "arg":
457
                     self.vm_writer.write_push("argument", self.symbol_table.index_of(name))
458
459
             else:
                 if self.symbol_table.kind_of(name) == "field":
460
                      self.vm_writer.write_push("this", self.symbol_table.index_of(name))
461
                  elif self.symbol_table.kind_of(name) == "static":
462
                      self.vm_writer.write_push("static", self.symbol_table.index_of(name))
463
464
465
         def pop_variable(self, name: str) -> None: # added
                "Pops the variable from the stack.
466
```

467

```
468
              name (str): The name of the variable.
469
470
              if name in self.symbol_table.cur_scope:
471
                  if self.symbol_table.kind_of(name) == "var":
472
                  self.vm_writer.write_pop("local", self.symbol_table.index_of(name))
elif self.symbol_table.kind_of(name) == "arg":
473
474
                       self.vm_writer.write_pop("argument", self.symbol_table.index_of(name))
475
476
              else:
                  if self.symbol_table.kind_of(name) == "field":
477
                       self.vm_writer.write_pop("this", self.symbol_table.index_of(name))
478
                  elif self.symbol_table.kind_of(name) == "static":
479
                       self.vm_writer.write_pop("static", self.symbol_table.index_of(name))
480
```

## 4 JackCompiler

```
#!/bin/sh
   # This file only works on Unix-like operating systems, so it won't work on Windows.
   ## Why do we need this file?
   # The purpose of this file is to run your project.
   # We want our users to have a simple API to run the project.
   # So, we need a "wrapper" that will hide all details to do so,
   # enabling users to simply type 'JackCompiler <path>' in order to use it.
   ## What are '#!/bin/sh' and '$*'?
10
   11
    # run "JackCompiler trout mask replica", $* will hold "trout mask replica".
12
13
   ## What should I change in this file to make it work with my project?
14
15
    # IMPORTANT: This file assumes that the main is contained in "JackCompiler.py".
                If your main is contained elsewhere, you will need to change this.
16
17
   python3 JackCompiler.py $*
18
19
   # This file is part of nand2tetris, as taught in The Hebrew University, and
   # was written by Aviv Yaish. It is an extension to the specifications given
21
   # in https://www.nand2tetris.org (Shimon Schocken and Noam Nisan, 2017),
23 # as allowed by the Creative Common Attribution-NonCommercial-ShareAlike 3.0
   # Unported License: https://creativecommons.org/licenses/by-nc-sa/3.0/
```

#### 5 JackCompiler.py

```
1
    This file is part of nand2tetris, as taught in The Hebrew University, and
    was written by Aviv Yaish. It is an extension to the specifications given
    [here] (https://www.nand2tetris.org) (Shimon Schocken and Noam Nisan, 2017),
    as allowed by the Creative Common Attribution-NonCommercial-ShareAlike 3.0
    Unported [License] (https://creativecommons.org/licenses/by-nc-sa/3.0/).
8
    import os
    import sys
9
    import typing
    from CompilationEngine import CompilationEngine
11
   from JackTokenizer import JackTokenizer
12
   from SymbolTable import SymbolTable
    from VMWriter import VMWriter
14
15
16
    def compile_file(
17
18
            input_file: typing.TextIO, output_file: typing.TextIO) -> None:
        """Compiles a single file.
19
20
21
            input_file (typing.TextIO): the file to compile.
22
23
            output_file (typing.TextIO): writes all output to this file.
24
        tokenizer = JackTokenizer(input_file)
25
26
        vm_writer = VMWriter(output_file)
27
        compilation_engine = CompilationEngine(tokenizer, vm_writer)
        compilation_engine.compile_class()
28
29
30
    if "__main__" == __name__:
31
        # Parses the input path and calls compile_file on each input file.
        # This opens both the input and the output files!
33
34
        # Both are closed automatically when the code finishes running.
        # If the output file does not exist, it is created automatically in the
35
        # correct path, using the correct filename.
36
37
        if not len(sys.argv) == 2:
            sys.exit("Invalid usage, please use: JackCompiler <input path>")
38
39
        argument_path = os.path.abspath(sys.argv[1])
40
        if os.path.isdir(argument_path):
            files_to_assemble = [
41
42
                 os.path.join(argument_path, filename)
43
                 for filename in os.listdir(argument_path)]
44
45
            files_to_assemble = [argument_path]
        for input_path in files_to_assemble:
46
47
            filename, extension = os.path.splitext(input_path)
            if extension.lower() != ".jack":
49
                continue
            output_path = filename + ".vm"
50
            with open(input_path, 'r') as input_file, \
51
                    open(output_path, 'w') as output_file:
52
53
                 compile_file(input_file, output_file)
```

### 6 JackTokenizer.py

```
1
    This file is part of nand2tetris, as taught in The Hebrew University, and
    was written by Aviv Yaish. It is an extension to the specifications given
    [here] (https://www.nand2tetris.org) (Shimon Schocken and Noam Nisan, 2017),
4
     as allowed by the Creative Common Attribution-NonCommercial-ShareAlike 3.0
    Unported [License] (https://creativecommons.org/licenses/by-nc-sa/3.0/).
6
8
    import typing
9
    import re
10
11
    class JackTokenizer:
         """Removes all comments from the input stream and breaks it
12
         into Jack language tokens, as specified by the Jack grammar.
13
14
15
         # Jack Language Grammar
16
        A Jack file is a stream of characters. If the file represents a
17
         valid program, it can be tokenized into a stream of valid tokens. The
18
         tokens may be separated by an arbitrary number of whitespace characters,
19
         and comments, which are ignored. There are three possible comment formats:
20
21
         /* comment until closing */ , /** API comment until closing */ , and
        // comment until the line's end.
22
23
         - 'xxx': quotes are used for tokens that appear verbatim ('terminals').
24
         - xxx: regular typeface is used for names of language constructs
25
                ('non-terminals').
26
         - (): parentheses are used for grouping of language constructs.
27
        -x/y: indicates that either x or y can appear.
28
         - x?: indicates that x appears 0 or 1 times.
29
         - x*: indicates that x appears 0 or more times.
30
31
        ## Lexical Elements
33
34
         The Jack language includes five types of terminal elements (tokens).
35
         - keyword: 'class' | 'constructor' | 'function' | 'method' | 'field' |
'static' | 'var' | 'int' | 'char' | 'boolean' | 'void' | 'true' |
36
37
                    'false' | 'null' | 'this' | 'let' | 'do' | 'if' | 'else' |
38
                    'while' | 'return'
39
         - symbol: '{' | '}' | '(' | ')' | '[' | ']' | '.' | ',' | ';' | '+' |
40
                   '-' | '*' | '/' | '&' | '|' | '<' | '>' | '=' | '~' | '^' | '#'
41
42
         - integerConstant: A decimal number in the range 0-32767.
         - StringConstant: '"' A sequence of Unicode characters not including
43
                           double quote or newline '"'
44
         - identifier: A sequence of letters, digits, and underscore ('_') not
45
                       starting with a digit. You can assume keywords cannot be
46
                       identifiers, so 'self' cannot be an identifier, etc'.
47
48
        ## Program Structure
49
50
        A Jack program is a collection of classes, each appearing in a separate
51
         file. A compilation unit is a single class. A class is a sequence of tokens
52
        structured according to the following context free syntax:
53
54
        - class: 'class' className '{' classVarDec* subroutineDec* '}'
55
         - classVarDec: ('static' | 'field') type varName (',' varName)* ';'
         - type: 'int' | 'char' | 'boolean' | className
57
        - subroutineDec: ('constructor' | 'function' | 'method') ('void' | type)
         - subroutineName '(' parameterList ')' subroutineBody
```

```
- parameterList: ((type varName) (',' type varName)*)?
 60
          - subroutineBody: '{' varDec* statements '}'
 61
          - varDec: 'var' type varName (',' varName)* ';'
 62
          - className: identifier
 63
          - subroutineName: identifier
 64
 65
          - varName: identifier
 66
         ## Statements
 67
 68
          - statements: statement*
 69
          - statement: letStatement \mid ifStatement \mid whileStatement \mid doStatement \mid
 70
                       returnStatement
 71
          - letStatement: 'let' varName ('[' expression ']')? '=' expression ';'
 72
         - ifStatement: 'if' '(' expression ')' '{' statements '}' ('else' '{'
 73
 74
                        statements '}')?
         - whileStatement: 'while' '(' 'expression' ')' '{' statements '}'
 75
          - doStatement: 'do' subroutineCall ';'
 76
          - returnStatement: 'return' expression? ';'
 77
 78
          ## Expressions
 79
 80
 81
          - expression: term (op term)*
          - term: integerConstant | stringConstant | keywordConstant | varName |
 82
                  varName '['expression']' | subroutineCall | '(' expression ')' |
 83
 84
                  unaryOp term
          - subroutineCall: subroutineName '(' expressionList ')' | (className |
 85
                            varName) '.' subroutineName '(' expressionList ')'
 86
          - expressionList: (expression (',' expression)*)?
 87
          - op: '+' | '-' | '*' | '/' | '&' | '|' | '<' | '>' | '='
 88
          - unaryOp: '-' | '~' | '^' | '#'
 89
 90
          - keywordConstant: 'true' | 'false' | 'null' | 'this'
 91
 92
          Note that ^, # correspond to shiftleft and shiftright, respectively.
 93
 94
 95
          def __init__(self, input_stream: typing.TextIO) -> None:
 96
               """Opens the input stream and gets ready to tokenize it.
 97
 98
              input_stream (typing.TextIO): input stream.
"""
 99
100
              self.input_lines = list(input_stream.read().splitlines()) # read lines into list
101
102
              self.clean_list() # clean list to prepare for parsing
103
              self.line_tokens = [] # the tokens found in the current line
             self.cur_token = None # the current token we are translating to the output stream
104
105
106
          def clean_list(self) -> None: # added
              """Cleans and prepares the input_lines list for parsing. Removes comments, trailing whitespaces and empty lines.
107
108
              self.remove_comments() # remove all comments from the input
109
              self.input_lines = [line.strip() for line in self.input_lines] # remove leading and trailing whitespace
110
              self.input_lines = [line for line in self.input_lines if line not in [" ", ""]] # remove empty lines
111
112
          def remove_comments(self) -> None: # added
113
              """Removes all comments from the input lines list unless the comment is inside a double quotes string.
114
115
116
              clean lines = []
117
              inside_double_quotes = False
             inside_multi_line_comment = False
118
119
120
              for line in self.input_lines:
121
                 cleaned_line = ""
                  i = 0
122
                  while i < len(line):
123
                      char = line[i]
124
125
                      if char == '"':
126
127
                          if inside_multi_line_comment: # if inside a multi-line comment, ignore double quotes
```

```
128
                             i += 1
129
                             continue
                         inside\_double\_quotes = not inside\_double\_quotes
130
131
                         cleaned_line += char
                         i += 1
132
133
                         continue
134
                     if not inside_double_quotes:
135
136
                         if char == '/' and i + 1 < len(line) and line[i + 1] == '/': # single-line comment, skip the rest
137
                             break
                         elif char == '/' and i + 1 < len(line) and line[i + 1] == '*': # start of multi-line comment
138
                             inside_multi_line_comment = True
139
                             i += 2
140
141
                             continue
142
                         elif char == '*' and i + 1 < len(line) and line[i + 1] == '/': # end of multi-line comment
                             inside_multi_line_comment = False
143
144
                             i += 2
145
                             continue
                         if not inside_multi_line_comment: # if not inside a comment, add the character to the cleaned line
146
                             cleaned_line += char
147
148
149
                     else:
150
                         cleaned_line += char
                     i += 1
151
152
153
                 clean_lines.append(cleaned_line)
154
155
             self.input_lines = clean_lines
156
157
         def next_token(self) -> str: # added
             """Returns the next token in the input stream without advancing the current token.
158
159
160
             if self.line_tokens != []:
                 return self.line_tokens[0]
161
162
163
164
                 token = self.parse_line(True)
165
                 return token
166
         def next_token_type(self) -> str: # added
167
              """Returns the type of the next token in the input stream without advancing the current token.
168
169
170
             token = self.next token()
171
             return self.token_type(token)
172
173
         def parse_line(self, flag=False): # added
174
              ""Parses the first line in the input list into tokens and saves them in a list.
              If the flag parameter is False, it saves the parsed line into self.line_tokens.
175
176
              Else, it returns the first token in the parsed line and doesn't save the parsing (used for checking the next
177
              token during compilation without advancing the current token).
178
             re_keyword = "\b(?:class|constructor|function|method|field|static|var|int|char|boolean" \
179
180
                          "|void|true|false|null|this|let|do|if|else|while|return)\b"
             181
             re_int = "[0-9]+"
182
             re_str = "\"[^\"\n]*\""
183
             re_identifier = r"[a-zA-Z_]\w*"
184
185
             if flag: # used when checking the next token
186
187
                 input_lines = self.input_lines.copy()
                 future_tokens = re.compile(re_keyword + "|" + re_symbol + "|" + re_int + "|" + re_str + "|" + re_identifier)
188
189
                 future_tokens = future_tokens.findall(input_lines[0])
                 return future_tokens[0]
190
191
             self.line_tokens = re.compile(re_keyword + "|" + re_symbol + "|" + re_int + "|" + re_str + "|" + re_identifier)
192
193
             self.line_tokens = self.line_tokens.findall(self.input_lines[0])
194
195
         def has_more_tokens(self) -> bool:
```

```
196
              """Do we have more tokens in the input?
197
198
              Returns:
              bool: True if there are more tokens, False otherwise.
199
200
              # check if there are more tokens in the current line or if there are more lines to parse
201
              return self.line_tokens != [] or self.input_lines != []
202
203
204
          def advance(self) -> None:
              """Gets the next token from the input and makes it the current token.
205
              This method should be called if has more_tokens() is true.
206
207
              Initially there is no current token.
208
              if self.has_more_tokens():
209
210
                   while self.line_tokens == []: # parse lines until a line with tokens is found
211
212
                       self.parse_line()
213
                       self.input_lines.pop(0)
214
215
                   self.cur_token = self.line_tokens.pop(0) # get the next token from the list and remove it
216
          def token_type(self, token=False) -> str:
217
218
219
              Aras:
220
                  token (str): the token to check its type. If not given, checks the self.cur_token.
221
              Returns:
222
223
                  str: the type of the token, can be
                   "KEYWORD", "SYMBOL", "IDENTIFIER", "INT_CONST", "STRING_CONST"
224
225
226
              if not token:
                  token = self.cur token
227
228
              229
230
231
                                      "do", "if", "else", "while", "return"]:
232
                  return "KEYWORD"
233
              if token in ['{', '}', '(', ')', '[', ']', '.', ',', ';', '+', '-', '*', '/', '&', '|', '<', '>', '=', '~', '^', '#']:
234
235
                  return "SYMBOL"
236
237
              if token.isdigit():
                  return "INT_CONST"
238
              if token.startswith('"') and token.endswith('"'):
239
                  return "STRING_CONST"
240
              return "IDENTIFIER" # if none of the above
241
242
          def keyword(self) -> str:
243
244
245
              Returns:
                  str: the keyword which is the current token.
246
247
                   Should be called only when token\_type() is "KEYWORD".
248
                   Can return "CLASS", "METHOD", "FUNCTION", "CONSTRUCTOR", "INT",
                  "BOOLEAN", "CHAR", "VOID", "VAR", "STATIC", "FIELD", "LET", "DO",
"IF", "ELSE", "WHILE", "RETURN", "TRUE", "FALSE", "NULL", "THIS"
249
250
251
252
              return self.cur_token
253
          def symbol(self) -> str:
254
255
256
257
                  str: the character which is the current token.
                   Should be called only when token_type() is "SYMBOL".
258
                  Recall that symbol was defined in the grammar like so:
259
                  symbol: '{' | '}' | '(' | ')' | '[' | ']' | '.' | ',' | ';' | '+' |
'-' | '*' | '/' | '8' | '|' | '<' | '>' | '=' | '~' | '^' | '#'
260
261
262
263
              if self.cur_token == '<':</pre>
```

```
264
                  return '<'
265
              if self.cur_token == '>':
                  return '>'
266
267
              if self.cur_token == '&':
                  return '&'
268
              return self.cur_token
269
270
         def identifier(self) -> str:
271
272
273
              Returns:
                  str:\ the\ identifier\ which\ is\ the\ current\ token.
274
275
                  Should be called only when token_type() is "IDENTIFIER".
                  Recall that identifiers were defined in the grammar like so:
276
                  identifier: A sequence of letters, digits, and underscore ('_') not
277
278
                        starting with a digit. You can assume keywords cannot be
                        identifiers, so 'self' cannot be an identifier, etc'.
279
280
281
              return self.cur_token
282
283
          def int_val(self) -> int:
284
              Returns:
285
                  str: the integer value of the current token.
286
                  Should be called only when token_type() is "INT_CONST".
287
288
                  {\it Recall\ that\ integer Constant\ was\ defined\ in\ the\ grammar\ like\ so:}
                  integerConstant: A decimal number in the range 0-32767.
289
290
291
              return int(self.cur_token)
292
293
          def string_val(self) -> str:
294
              11 11 11
              Returns:
295
                  str: the string value of the current token, without the double
296
297
                  quotes. Should be called only when token_type() is "STRING_CONST".
                  Recall that StringConstant was defined in the grammar like so:
298
299
                  StringConstant: \ '"' \ \textit{A sequence of Unicode characters not including}
                             double quote or newline '"'
300
301
              return self.cur_token[1:-1]
```

#### 7 Makefile

```
# Makefile for a script (e.g. Python)
1
2
    ## Why do we need this file?
3
    # We want our users to have a simple API to run the project.
4
    # So, we need a "wrapper" that will hide all details to do so,
    # thus enabling our users to simply type 'JackCompiler <path>' in order to use it.
    ## What are makefiles?
    # This is a sample makefile.
9
10
    # The purpose of makefiles is to make sure that after running "make" your
    # project is ready for execution.
11
12
    ## What should I change in this file to make it work with my project?
13
    # Usually, scripting language (e.g. Python) based projects only need execution
14
    # permissions for your run file executable to run.
15
    # Your project may be more complicated and require a different makefile.
17
18
    ## What is a makefile rule?
    # A makefile rule is a list of prerequisites (other rules that need to be run
19
    # before this rule) and commands that are run one after the other.
20
21
    # The "all" rule is what runs when you call "make".
    # In this example, all it does is grant execution permissions for your
22
    # executable, so your project will be able to run on the graders' computers.
23
    # In this case, the "all" rule has no pregrequisites.
24
25
26
   ## How are rules defined?
27
    # The following line is a rule declaration:
    # 0.1.1.:
28
29
    #
          chmod a+x JackCompiler
30
    # A general rule looks like this:
31
    # rule_name: prerequisite1 prerequisite2 prerequisite3 prerequisite4 ...
        command1
33
34
    #
         command2
35
        command3
    #
36
37
    # Where each pregrequisite is a rule name, and each command is a command-line
    # command (for example chmod, javac, echo, etc').
38
39
40
    # Beginning of the actual Makefile
    all:
41
42
        chmod a+x *
43
    # This file is part of nand2tetris, as taught in The Hebrew University, and
44
    # was written by Aviv Yaish. It is an extension to the specifications given
    # in https://www.nand2tetris.org (Shimon Schocken and Noam Nisan, 2017),
46
47
    # as allowed by the Creative Common Attribution-NonCommercial-ShareAlike 3.0
    # Unported License: https://creativecommons.org/licenses/by-nc-sa/3.0/
```

### 8 SymbolTable.py

```
1
    This file is part of nand2tetris, as taught in The Hebrew University, and
    was written by Aviv Yaish. It is an extension to the specifications given
    [here] (https://www.nand2tetris.org) (Shimon Schocken and Noam Nisan, 2017),
    as allowed by the Creative Common Attribution-NonCommercial-ShareAlike 3.0
    Unported [License] (https://creativecommons.org/licenses/by-nc-sa/3.0/).
6
8
    import typing
9
10
    class SymbolTable:
        """A symbol table that associates names with information needed for Jack
11
        compilation: type, kind and running index. The symbol table has two nested
12
        scopes (class/subroutine).
13
14
15
16
        def __init__(self) -> None:
             """Creates a new empty symbol table."""
17
            self.class_scope = {}
18
            self.subroutine_scope = {}
19
20
            self.cur_scope = None
            self.index = {"static": 0, "field": 0, "arg": 0, "var": 0, "while": 0, "if": 0}
21
22
23
        def start_subroutine(self, name) -> None:
             """Starts a new subroutine scope (i.e., resets the subroutine's
24
            symbol table).
25
26
27
            self.subroutine_scope[name] = {}
            self.index["arg"] = 0
28
29
            self.index["var"] = 0
            self.index["while"] = 0
30
            self.index["if"] = 0
31
        def define(self, name: str, type: str, kind: str) -> None:
33
34
             """Defines a new identifier of a given name, type and kind and assigns
            it a running index. "STATIC" and "FIELD" identifiers have a class scope,
35
            while "ARG" and "VAR" identifiers have a subroutine scope.
36
37
38
            Args:
                 name (str): the name of the new identifier.
39
40
                 type (str): the type of the new identifier.
                 kind (str): the kind of the new identifier, can be:
41
42
                 "STATIC", "FIELD", "ARG", "VAR".
43
            if kind in ["static", "field"]:
44
45
                 self.class_scope[name] = (type, kind, self.index[kind])
                self.index[kind] += 1
46
            else: # kind in ["argument", "local"]
47
                self.cur_scope[name] = (type, kind, self.index[kind])
                self.index[kind] += 1
49
50
51
        def globals_count(self, kind: str) -> int:
52
53
            Args:
                kind (str): can be "STATIC", "FIELD", "ARG", "VAR".
54
55
                int: the number of variables of the given kind already defined in
57
58
                the class scope.
```

```
60
              return len([v for (k, v) in self.class_scope.items() if v[1] == kind])
 61
         def var_count(self, kind: str) -> int:
 62
 63
 64
              Args:
                 kind (str): can be "STATIC", "FIELD", "ARG", "VAR".
 65
 66
             Returns:
 67
 68
                 int: the number of variables of the given kind already defined in
                 the current scope.
 69
 70
 71
              return len([v for (k, v) in self.cur_scope.items() if v[1] == kind])
 72
         def kind_of(self, name: str):
 73
 74
             Args:
 75
 76
                 name (str): name of an identifier.
 77
 78
             Returns:
 79
                 str: the kind of the named identifier in the current scope, or None
                 if the identifier is unknown in the current scope.
 80
 81
             if name in self.cur_scope:
 82
                 return self.cur_scope[name][1]
 83
 84
              elif name in self.class_scope:
 85
                return self.class_scope[name][1]
              else:
 86
 87
                 return None
 88
 89
          def type_of(self, name: str):
90
              Args:
 91
                 name (str): name of an identifier.
 92
 93
             Returns:
 94
              str: the type of the named identifier in the current scope.
 95
 96
             if name in self.cur_scope:
97
                 return self.cur_scope[name][0]
              elif name in self.class_scope:
99
100
                 return self.class_scope[name][0]
101
                 return None
102
103
          def index_of(self, name: str):
104
105
106
             Args:
                 name (str): name of an identifier.
107
108
109
              int: the index assigned to the named identifier.
110
111
112
             if name in self.cur_scope:
113
                 return self.cur_scope[name][2]
              elif name in self.class_scope:
114
                 return self.class_scope[name][2]
115
116
              else:
                 return None
117
118
119
          def set_scope(self, subroutine_name: str):
              """Sets the current scope to the given scope.
120
121
122
              subroutine_name (str): the scope to set.
123
124
             if subroutine_name == "class":
                 self.cur_scope = self.class_scope
126
127
             else:
```

self.cur\_scope = self.subroutine\_scope[subroutine\_name]

### 9 VMWriter.py

```
1
    This file is part of nand2tetris, as taught in The Hebrew University, and
    was written by Aviv Yaish. It is an extension to the specifications given
    [here](https://www.nand2tetris.org) (Shimon Schocken and Noam Nisan, 2017),
4
    as allowed by the Creative Common Attribution-NonCommercial-ShareAlike 3.0
    Unported [License] (https://creativecommons.org/licenses/by-nc-sa/3.0/).
6
8
    import typing
    import SymbolTable
9
10
    class VMWriter:
11
12
         Writes VM commands into a file. Encapsulates the VM command syntax.
13
14
15
         def __init__(self, output_stream: typing.TextIO) -> None:
16
             """Creates a new file and prepares it for writing VM commands."""
17
18
             self.output_stream = output_stream
19
             self.cur_symbol_table = None
20
21
         def write_push(self, segment: str, index: int) -> None:
             """Writes a VM push command.
22
23
24
             Args:
                 segment (str): the segment to push to, can be "CONST", "ARG",
25
                 "LOCAL", "STATIC", "THIS", "THAT", "POINTER", "TEMP"
26
27
                 index (int): the index to push to.
28
29
             self.output_stream.write("push " + segment + " " + str(index) + "\n")
30
         def write_pop(self, segment: str, index: int) -> None:
31
             """Writes a VM pop command.
33
34
             Args:
                 segment (str): the segment to pop from, can be "CONST", "ARG", "LOCAL", "STATIC", "THIS", "THAT", "POINTER", "TEMP".
35
36
37
                 index (int): the index to pop from.
38
             self.output\_stream.write("pop " + segment + " " + str(index) + " \backslash n")
39
40
         def write_arithmetic(self, command: str) -> None:
41
42
             """Writes a VM arithmetic command.
43
44
             Aras:
                 command (str): the command to write, can be "ADD", "SUB", "NEG",
45
                 "EQ", "GT", "LT", "AND", "OR", "NOT", "SHIFTLEFT", "SHIFTRIGHT".
46
47
             self.output_stream.write(command.lower() + "\n")
49
50
         def write_label(self, label: str) -> None:
             """Writes a VM label command.
51
52
53
             label (str): the label to write.
54
55
             self.output_stream.write("label " + label + "\n")
56
57
58
         def write_goto(self, label: str) -> None:
```

```
60
            """Writes a VM goto command.
61
62
            Args:
            label (str): the label to go to.
63
64
            self.output_stream.write("goto " + label + "\n")
65
66
        def write_if(self, label: str) -> None:
67
68
             """Writes\ a\ VM\ if-goto\ command.
69
70
            label (str): the label to go to.
71
72
            self.output_stream.write("if-goto " + label + "\n")
73
74
        def write_call(self, name: str, n_args: int) -> None:
75
76
            """Writes a VM call command.
77
78
            Args:
79
                name (str): the name of the function to call.
            n_args (int): the number of arguments the function receives.
80
81
            self.output_stream.write("call " + name + " " + str(n_args) + "\n")
82
83
        def write_function(self, name: str, n_locals: int) -> None:
84
            """Writes a VM function command.
85
86
87
                name (str): the name of the function.
88
                n\_locals (int): the number of local variables the function uses.
89
90
            self.output_stream.write("function " + name + " " + str(n_locals) + "\n")
91
92
93
        def write_return(self) -> None:
             """Writes a VM return command."""
94
95
            self.output_stream.write("return\n")
```