Contents

| 1 | Basic Test Results | 2 |
|---|----------------------|----|
| 2 | AUTHORS | 3 |
| 3 | CompilationEngine.py | 4 |
| 4 | JackCompiler | 13 |
| 5 | JackCompiler.py | 14 |
| 6 | JackTokenizer.py | 15 |
| 7 | Makefile | 20 |
| 8 | SymbolTable.py | 21 |
| 9 | VMWriter.py | 23 |

1 Basic Test Results

```
****** TESTING FOLDER STRUCTURE START *******
    Checking your submission for presence of invalid (non-ASCII) characters...
    No invalid characters found.
    Submission logins are: linorcohen
4
    Is this OK?
    ****** TESTING FOLDER STRUCTURE END *******
    ****** PROJECT TEST START *******
    Running 'make'.
9
10
    'make' ran successfully.
    Testing.
11
12
    Running your program with command: 'JackCompiler tst/ComplexArrays'.
    Main.vm was created in test ComplexArrays.
14
    Checking validity of generated {\tt VM} code.
15
    Generated VM code passed the test successfully.
16
17
    Running your program with command: 'JackCompiler tst/Seven'.
18
    Main.vm was created in test Seven.
19
    Checking validity of generated VM code.
20
21
    Generated VM code passed the test successfully.
    ******* PROJECT TEST END *******
22
23
    Note: the tests you see above are all the presubmission tests
24
   for this project. The tests might not check all the different
25
26\, \, parts of the project or all corner cases, so write your own
   tests and use them!
```

2 AUTHORS

- linorcohen
 Partner 1: Linor Cohen, linor.cohen@mail.huji.ac.il, 318861226
 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
   from SymbolTable import SymbolTable
    import VMWriter
11
12
    class CompilationEngine:
14
        """Gets input from a JackTokenizer and emits its parsed structure into an
15
16
         output stream.
17
18
        FUNCTION = "function"
19
        CONSTRUCTOR = "constructor"
20
21
        METHOD = "method"
        STATIC = "static"
22
        FIELD = "field"
23
        RETURN = "return"
24
        WHILE = "while"
25
        LET = "let"
26
27
        D0 = "do"
        IF = "if"
28
29
        ELSE = "else"
30
        KEYWORD = "KEYWORD"
31
         SYMBOL = "SYMBOL"
         IDENTIFIER = "IDENTIFIER"
33
         INT_CONST = "INT_CONST"
34
        STRING_CONST = "STRING_CONST"
35
36
37
         def __init__(self, input_stream: "JackTokenizer", class_symbol_table: "SymbolTable",
                      vm_writer: "VMWriter", output_stream: typing.TextIO) -> None:
38
39
40
             Creates a new compilation engine with the given input and output. The
             next routine called must be compileClass()
41
42
             :param input_stream: The input stream.
43
             :param output_stream: The output stream.
44
45
             self.class_symbol_table = class_symbol_table
             self.vm_writer = vm_writer
46
             self.tokenizer = input_stream
47
             self.output_stream = output_stream
49
             self.op_terms = {"+": "ADD", "-": "SUB", "*": "call Math.multiply 2", "/": "call Math.divide 2",
50
             "&": "AND", "|": "OR", "<": "LT", ">": "GT", "=": "EQ"}
self.unary_op_terms = {"^": "SHIFTLEFT", "#": "SHIFTRIGHT", "-": "NEG", "~": "NOT"}
51
52
53
             self.class_name = ""
54
55
             self.counter = 0
             self.subroutine_symbol_table = SymbolTable()
57
58
         def __advance_tokenizer(self) -> None:
```

```
60
             this method advance the tokenizer if has more tokens
61
62
             if self.tokenizer.has_more_tokens():
                 self.tokenizer.advance()
63
64
65
         def __get_current_token_and_advance(self) -> str:
66
             this method advance the token and get the current token
67
68
              :return: Tuple(token, token tag type)
              11 11 11
69
70
             self.__advance_tokenizer()
             return self.__get_current_token()
71
72
73
         def __get_current_token(self) -> str:
74
             this method return the tuple of the current token and the current token type tag.
75
76
             :return: Tuple(token, token tag type)
77
             t_type = self.tokenizer.token_type()
78
             if t_type == self.KEYWORD:
                 return self.tokenizer.keyword()
80
81
             elif t_type == self.SYMBOL:
82
                 return self.tokenizer.symbol()
             elif t_type == self.IDENTIFIER:
83
84
                 return self.tokenizer.identifier()
             elif t_type == self.INT_CONST:
85
                 return str(self.tokenizer.int_val())
86
87
             elif t_type == self.STRING_CONST:
                 return self.tokenizer.string_val()
88
89
         def __get_var_info_from_table(self, var_name: str) -> typing.Tuple[str, str, str]:
90
91
92
             this method return the var info from its symbol table.
93
             :param var_name: variable name
              :return: variable type, variable kind, variable index
94
95
96
             # symbol in class table
97
             if self.subroutine_symbol_table.kind_of(var_name) is not None:
98
                  # symbol in subroutine table
                 return self.subroutine_symbol_table.type_of(var_name), \
99
100
                         self.subroutine_symbol_table.kind_of(var_name), \
                         self.subroutine_symbol_table.index_of(var_name)
101
102
             return self.class_symbol_table.type_of(var_name), self.class_symbol_table.kind_of(
103
                  var_name), self.class_symbol_table.index_of(var_name)
104
105
         def compile_class(self) -> None:
106
              """Compiles a complete class."""
             # class
107
108
             self.__get_current_token_and_advance()
109
             # className
             self.class_name = self.__get_current_token_and_advance()
110
             # {
111
112
             self.__get_current_token_and_advance()
113
              # classVarDec -> *
             token = self.__get_current_token_and_advance()
114
             while token in {self.FIELD, self.STATIC}:
115
116
                  self.compile_class_var_dec()
                 token = self.__get_current_token_and_advance()
117
              # subroutineDec -> >
118
119
             while token in {self.METHOD, self.CONSTRUCTOR, self.FUNCTION}:
120
                  self.compile_subroutine()
121
                 token = self.__get_current_token_and_advance()
              # 7
122
123
         def compile_class_var_dec(self) -> None:
124
              """Compiles a static declaration or a field declaration."""
125
              # field or static
126
127
             kind = self.__get_current_token()
```

```
128
             # type
             token = self.__get_current_token_and_advance()
129
             var_type = token
130
              # varName ->
131
             while token != ";":
132
133
                  # varName
134
                 name = self.__get_current_token_and_advance()
                  {\it \# add to class\_table}
135
136
                  self.class_symbol_table.define(name, var_type, kind)
137
                  # symbol
                  token = self.__get_current_token_and_advance()
138
139
         def compile_subroutine(self) -> None:
140
141
142
             Compiles a complete method, function, or constructor.
             You can assume that classes with constructors have at least one field.
143
144
             # keyword - method, function, or constructor.
145
             subroutine_type = self.__get_current_token()
146
              # identifier - return type
147
             self.__get_current_token_and_advance()
148
149
              # identifier - name
150
             subroutine_name = self.__get_current_token_and_advance()
             # reset the subroutine symbol table
151
152
             self.subroutine_symbol_table.start_subroutine()
153
             # add the object this to subroutine table
             if subroutine_type == self.METHOD:
154
155
                  self.subroutine_symbol_table.define("this", self.class_name, "ARG")
156
157
             self.__get_current_token_and_advance()
158
             # parameter list
             self.compile_parameter_list()
159
160
             # )
161
             self.__get_current_token()
             # subroutine body
162
             self.__compile_subroutine_body(subroutine_name, subroutine_type)
163
164
165
         def __compile_subroutine_body(self, subroutine_name: str, subroutine_type: str) -> None:
166
             this method compile a subroutine body
167
168
             # {
169
170
             self.__get_current_token_and_advance()
171
             n = 0
             # var -> *
172
             var_type = self.__get_current_token_and_advance()
173
174
             while var_type == "var":
                 n += self.compile_var_dec()
175
176
                 var_type = self.__get_current_token_and_advance()
177
             # function className.subroutineName n
             self.vm_writer.write_function(f"""{self.class_name}.{subroutine_name}""", n)
178
             if subroutine_type == self.CONSTRUCTOR:
179
180
                  # push const nField
181
                 self.vm_writer.write_push("CONST", self.class_symbol_table.var_count(self.FIELD))
                  # call Memory.alloc 1
182
                 self.vm_writer.write_call("Memory.alloc", 1)
183
184
                  # pop pointer 0
185
                 self.vm_writer.write_pop("POINTER", 0)
             if subroutine_type == self.METHOD:
186
187
                  # push argument 0
188
                  self.vm_writer.write_push("ARG", 0)
189
                  # pop pointer 0
                  self.vm_writer.write_pop("POINTER", 0)
190
             # statements
191
192
             self.compile_statements(subroutine_type)
193
             self.__get_current_token()
194
195
```

```
196
         def compile_parameter_list(self) -> None:
              """Compiles a (possibly empty) parameter list, not including the
197
              enclosing "()".
198
199
             is_first = True
200
201
             # varName -> *
              var_type = self.__get_current_token_and_advance()
202
             while var_type != ")":
203
204
                  if is_first:
                      name = self.__get_current_token_and_advance()
205
206
                      # add to subroutine table
207
                      self.subroutine_symbol_table.define(name, var_type, "ARG")
                      var_type = self.__get_current_token_and_advance()
208
                      is_first = False
209
210
                  else:
                      var_type = self.__get_current_token_and_advance()
211
212
                      name = self.__get_current_token_and_advance()
                      # add to subroutine table
213
                      self.subroutine_symbol_table.define(name, var_type, "ARG")
214
215
                      var_type = self.__get_current_token_and_advance()
216
         def compile_var_dec(self) -> int:
217
              """Compiles a var declaration."""
218
              # keyword
219
220
             kind = self.__get_current_token()
221
              # identifier
             token = self.__get_current_token_and_advance()
222
             var_type = token
223
             n = 0
224
225
             # varName -> *
             while token != ";":
                 # identifier
227
228
                  name = self.__get_current_token_and_advance()
229
                  # add to subroutine table
                  self.subroutine_symbol_table.define(name, var_type, kind.upper())
230
231
                  n += 1
232
                  # symbol
233
                  token = self.__get_current_token_and_advance()
234
235
         def __compile_string(self, string_value: str) -> None:
236
237
              this method compile a string constance
238
239
              :param string_value: the string variable
240
             # push const length
241
242
             self.vm_writer.write_push("CONST", len(string_value))
              # call string.new
243
244
             self.vm_writer.write_call("String.new", 1)
^{245}
             for char in string_value:
                  # push const char
246
                  self.vm_writer.write_push("CONST", str(ord(char)))
247
248
                  # call string.appendChar
                  self.vm_writer.write_call("String.appendChar", 2)
249
250
         def compile_statements(self, subroutine_type: str) -> None:
251
              """Compiles a sequence of statements, not including the enclosing
252
              "{}".
253
254
255
              token = self.__get_current_token()
              while token != "}":
256
                  if token == self.IF:
257
                      self.compile_if(subroutine_type)
258
                      token = self.__get_current_token()
259
260
                  else:
261
                     if token == self.DO:
                          self.compile_do()
262
263
                      elif token == self.LET:
```

```
264
                          self.compile_let()
                      elif token == self.WHILE:
265
                          self.compile_while(subroutine_type)
266
                      elif token == self.RETURN:
267
                          self.compile_return(subroutine_type)
268
269
                      token = self.__get_current_token_and_advance()
270
         def __subroutine_call_format(self, obj_name: str, is_term: bool) -> None:
271
272
              this method compile the subroutine call format
273
274
275
             function_call_name = obj_name
276
             is_method = 0
277
             # . -> ?
278
              symbol = self.__get_current_token()
              if symbol == ".":
279
280
                  var_type, var_kind, var_index = self.__get_var_info_from_table(obj_name)
                  if var_type is not None and var_kind is not None and var_kind is not None: # varName
281
                      if not is term:
282
                          # push obj
283
                          self.vm_writer.write_push(var_kind, var_index)
284
285
                      is_method = 1
                      function_call_name = var_type
286
287
                  # functionName
288
                  function_name = self.__get_current_token_and_advance()
289
                  function_call_name += symbol + function_name
290
291
                  self.__get_current_token_and_advance()
              else:
292
293
                  self.vm_writer.write_push("POINTER", 0)
294
                  function_call_name = self.class_name + "." + obj_name
                  is\_method = 1
295
              # (
296
297
              # expression list
             self.__get_current_token_and_advance()
298
299
              n = self.compile_expression_list()
300
301
             self.__get_current_token()
              \# output "call f n" or "call f n+1"
302
              self.vm_writer.write_call(function_call_name, n + is_method)
303
304
          def compile_do(self) -> None:
305
              """Compiles a do statement."""
306
307
              # keyword = do
             self.__get_current_token()
308
309
              # varName or className
310
             name = self.__get_current_token_and_advance()
311
312
              self.__get_current_token_and_advance()
313
              # subroutine call
             self.__subroutine_call_format(name, False)
314
315
              # ;
316
              self.__get_current_token_and_advance()
              self.vm_writer.write_pop("TEMP", 0)
317
318
         def compile_let(self) -> None:
319
              """Compiles a let statement."""
320
              # keyword = let
321
              self.__get_current_token()
322
323
              # identifier
324
              var_name = self.__get_current_token_and_advance()
325
              var_type, var_kind, var_index = self.__get_var_info_from_table(var_name)
326
              # [ -> ?
             symbol = self.__get_current_token_and_advance()
327
328
              # handle array
              is_array = False
329
             if symbol == "[":
330
331
                  # push x
```

```
332
                  self.vm_writer.write_push(var_kind, var_index)
333
                  # expression
334
                  self.__get_current_token_and_advance()
                  self.compile_expression()
335
                  # ]
336
337
                  self.__get_current_token_and_advance()
                  # add
338
                  self.vm_writer.write_arithmetic("ADD")
339
340
                  is_array = True
              # symbol
341
              self.__get_current_token()
342
343
              # expression
344
              self.__get_current_token_and_advance()
              self.compile_expression()
345
346
              self.__get_current_token()
347
348
              if is_array:
                  # pop temp O
349
                  self.vm_writer.write_pop("TEMP", 0)
350
351
                  # pop pointer 1
                  self.vm_writer.write_pop("POINTER", 1)
352
353
                  # push temp 0
                  self.vm_writer.write_push("TEMP", 0)
354
355
                  # pop that O
                  self.vm_writer.write_pop("THAT", 0)
356
357
                  self.vm_writer.write_pop(var_kind, var_index)
358
359
          def compile_while(self, subroutine_type: str) -> None:
360
              """Compiles a while statement."""
361
362
              # keyword = while
              self.__get_current_token()
363
364
              # label L1
365
              11 = f"""{self.class_name}_L_{self.counter}"""
              self.counter += 1
366
367
              self.vm_writer.write_label(l1)
368
369
              self.__get_current_token_and_advance()
370
              # expression
              self.__get_current_token_and_advance()
371
372
              self.compile_expression()
373
              self.__get_current_token()
374
375
              # not
              self.vm_writer.write_arithmetic("NOT")
376
377
              # if-goto L2
378
              12 = f"""{self.class_name}_L_{self.counter}"""
              self.counter += 1
379
380
              self.vm_writer.write_if(12)
              # {
381
              # statements
382
383
              self.__get_current_token_and_advance()
384
              self.compile_statements(subroutine_type)
385
              # }
              self.__get_current_token()
386
              # goto L1
387
388
              self.vm_writer.write_goto(11)
389
              # label L2
              self.vm_writer.write_label(12)
390
391
392
          def compile_return(self, subroutine_type: str) -> None:
              """Compiles a return statement.""
393
394
              # keyword = return
              self.__get_current_token()
395
396
              # expression -> ?
              symbol = self.__get_current_token_and_advance()
397
              if symbol != ";":
398
399
                  # expression
```

```
400
                 self.compile_expression()
401
             else:
                 if subroutine_type == self.CONSTRUCTOR:
402
403
                      self.vm_writer.write_push("POINTER", 0)
404
                  else:
                      self.vm_writer.write_push("CONST", 0)
405
              self.vm_writer.write_return()
406
407
408
          def compile_if(self, subroutine_type: str) -> None:
              """Compiles a if statement, possibly with a trailing else clause."""
409
              # keyword = if
410
411
              self.__get_current_token()
412
              # (
413
             self.__get_current_token_and_advance()
414
              # expression
             self.__get_current_token_and_advance()
415
416
             self.compile_expression()
417
             self.__get_current_token()
418
              # not
419
             self.vm_writer.write_arithmetic("NOT")
420
421
              # if-goto L1
             11 = f"""{self.class_name}_L_{self.counter}"""
422
             self.counter += 1
423
424
             self.vm_writer.write_if(l1)
425
             # {
             self.__get_current_token_and_advance()
426
427
              # statements
             self.__get_current_token_and_advance()
428
429
             self.compile_statements(subroutine_type)
430
              # }
             self.__get_current_token()
431
432
             # goto L2
433
             12 = f"""{self.class_name}_L_{self.counter}"""
             self.counter += 1
434
435
             self.vm_writer.write_goto(12)
436
              # label L1
437
             self.vm_writer.write_label(11)
              # else -> ?
438
             token = self.__get_current_token_and_advance()
439
440
             if token == self.ELSE:
                 # {
441
                 self.__get_current_token_and_advance()
442
443
                  # statements
                 self.__get_current_token_and_advance()
444
445
                 self.compile_statements(subroutine_type)
446
                 self.__get_current_token()
447
448
                 self.__get_current_token_and_advance()
              # label L2
449
             self.vm_writer.write_label(12)
450
451
452
         def compile_expression(self) -> None:
              """Compiles an expression."""
453
              # term
454
             self.compile_term()
455
456
             # term -> *
             token = self.__get_current_token()
457
             while token != ")":
458
459
                 if token not in self.op_terms:
460
                      break
                  # op
461
                 op = token
462
                  # term
463
464
                 self.__get_current_token_and_advance()
465
                 self.compile_term()
466
467
                  # output "op"
```

```
468
                  self.vm_writer.write_arithmetic(self.op_terms[op])
                  token = self.__get_current_token()
469
470
          def compile_term(self) -> None:
471
              """Compiles a term.
472
              This routine is faced with a slight difficulty when
473
              trying to decide between some of the alternative parsing rules.
474
              Specifically, if the current token is an identifier, the routing must
475
476
              distinguish between a variable, an array entry, and a subroutine call.
              A single look-ahead token, which may be one of "[", "(", or "." suffices
477
              to distinguish between the three possibilities. Any other token is not
478
479
              part of this term and should not be advanced over.
480
              # identifier / symbol
481
482
              token = self.__get_current_token()
              # push const
483
484
              if token.isnumeric():
                  self.vm_writer.write_push("CONST", int(token))
485
              elif token == "true":
486
                  self.vm_writer.write_push("CONST", 1)
487
                  self.vm_writer.write_arithmetic("NEG")
488
              elif token == "false":
489
                  self.vm_writer.write_push("CONST", 0)
490
              elif token == "null":
491
492
                  self.vm_writer.write_push("CONST", 0)
              elif token == "this":
493
                  self.vm_writer.write_push("POINTER", 0)
494
495
              elif self.tokenizer.token_type() == "STRING_CONST":
                 self.__compile_string(token)
496
497
              # push var
498
              elif self.class_symbol_table.kind_of(token) is not None or \
                      self.subroutine_symbol_table.kind_of(token) is not None:
499
500
                  var_type, var_kind, var_index = self.__get_var_info_from_table(token)
501
                  self.vm_writer.write_push(var_kind, var_index)
              # unaru term -> ?
502
503
              if token in self.unary_op_terms:
504
                  self.__get_current_token_and_advance()
505
                  # term
                  self.compile_term()
506
                  # output unaryOp
507
508
                  self.vm_writer.write_arithmetic(self.unary_op_terms[token])
              # expression - > ?
509
              elif token == "(":
510
511
                  # expression
                  self.__get_current_token_and_advance()
512
513
                  self.compile_expression()
514
                  # self.__get_current_token()
515
516
                  self.__get_current_token_and_advance()
517
                  function_call_name = token
518
                  token = self.__get_current_token_and_advance()
519
520
                  # handle array
                  # [ -> ?
521
                  if token == "[":
522
                      # expression
523
524
                      self.__get_current_token_and_advance()
525
                      self.compile_expression()
                      # 7
526
527
                      # self.__get_current_token()
528
                      self.__get_current_token_and_advance()
529
                      # a.d.d.
                      self.vm_writer.write_arithmetic("ADD")
530
                      # pop pointer 1
531
                      self.vm_writer.write_pop("POINTER", 1)
532
533
                      # push that 0
                      self.vm_writer.write_push("THAT", 0)
534
535
                  \# subroutine call -> ?
```

```
elif token in {".", "("}:
536
                       {\tt self.\_subroutine\_call\_format(function\_call\_name,\ True)}
537
                       {\tt self.\_get\_current\_token\_and\_advance()}
538
539
          def compile_expression_list(self) -> int:
540
              """Compiles a (possibly empty) comma-separated list of expressions."""
541
542
             n = 0
              # expression -> ?
543
544
              token = self.__get_current_token()
              while token != ")":
545
546
                  # expression
547
                  self.compile_expression()
                  n += 1
548
                  token = self.__get_current_token()
if token == ",":
549
550
                      token = self.__get_current_token_and_advance()
551
              return n
552
```

4 JackCompiler

```
#!/bin/sh
   # This file only works on Unix-like operating systems, so it won't work on Windows.
2
   ## 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
        class_symbol_table = SymbolTable()
        engine = CompilationEngine(tokenizer, class_symbol_table,vm_writer, output_file)
28
29
        engine.compile_class()
30
        output_file.close()
31
33
    if "__main__" == __name__:
34
         # Parses the input path and calls compile_file on each input file.
35
        # This opens both the input and the output files!
36
37
        # Both are closed automatically when the code finishes running.
        # If the output file does not exist, it is created automatically in the
38
39
        # correct path, using the correct filename.
40
        if not len(sys.argv) == 2:
            sys.exit("Invalid usage, please use: JackCompiler <input path>")
41
42
        argument_path = os.path.abspath(sys.argv[1])
43
        if os.path.isdir(argument_path):
            files_to_assemble = [
44
45
                 os.path.join(argument_path, filename)
                 for filename in os.listdir(argument_path)]
46
47
        else:
            files_to_assemble = [argument_path]
        for input_path in files_to_assemble:
49
50
            filename, extension = os.path.splitext(input_path)
            if extension.lower() != ".jack":
51
                continue
52
53
            output_path = filename + ".vm"
            with open(input_path, 'r') as input_file, \
54
                     open(output_path, 'w') as output_file:
55
                 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),
    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 re
    import typing
9
    import shlex
11
12
    class JackTokenizer:
13
         """Removes all comments from the input stream and breaks it
14
15
         into Jack language tokens, as specified by the Jack grammar.
16
         # Jack Language Grammar
17
18
         A Jack file is a stream of characters. If the file represents a
19
         valid\ program,\ it\ can\ be\ tokenized\ into\ a\ stream\ of\ valid\ tokens.\ The
20
21
         tokens may be separated by an arbitrary number of whitespace characters,
         and comments, which are ignored. There are three possible comment formats:
22
23
         /* comment until closing */ , /** API comment until closing */ , and
         // comment until the line's end.
24
25
         - 'xxx': quotes are used for tokens that appear verbatim ('terminals').
26
27
         - xxx: regular typeface is used for names of language constructs
                ('non-terminals').
28
         - (): parentheses are used for grouping of language constructs.
29
         -x / y: indicates that either x or y can appear.
30
31
         - x?: indicates that x appears 0 or 1 times.
         - x*: indicates that x appears 0 or more times.
33
34
         ## Lexical Elements
35
         The Jack language includes five types of terminal elements (tokens).
36
37
         - keyword: 'class' | 'constructor' | 'function' | 'method' | 'field' |
38
                    'static' | 'var' | 'int' | 'char' | 'boolean' | 'void' | 'true' |
39
                     'false' | 'null' | 'this' | 'let' | 'do' | 'if' | 'else' |
40
                     'while' | 'return'
41
         - symbol: '{' | '}' | '(' | ')' | '[' | ']' | '.' | ',' | ';' | '+' | '-' | '*' | '/' | '8' | '|' | '<' | '>' | '=' | '~' | '^' | '#'
42
43
         - integerConstant: A decimal number in the range 0-32767.
44
         - StringConstant: '"' A sequence of Unicode characters not including
45
                           double quote or newline '"'
46
47
         - identifier: A sequence of letters, digits, and underscore (^{\prime}_^{\prime}) not
                       starting with a digit. You can assume keywords cannot be
                       identifiers, so 'self' cannot be an identifier, etc'.
49
50
         ## Program Structure
51
52
         A Jack program is a collection of classes, each appearing in a separate
53
        file. A compilation unit is a single class. A class is a sequence of tokens
54
55
         structured according to the following context free syntax:
         - class: 'class' className '{' classVarDec* subroutineDec* '}'
57
         - classVarDec: ('static' | 'field') type varName (',' varName)* ';'
58
         - type: 'int' | 'char' | 'boolean' | className
```

```
- subroutineDec: ('constructor' | 'function' | 'method') ('void' | type)
 60
         - subroutineName '(' parameterList ')' subroutineBody
 61
         - parameterList: ((type varName) (',' type varName)*)?
 62
         - subroutineBody: '{' varDec* statements '}'
 63
         - varDec: 'var' type varName (',' varName)* ';'
 64
         - className: identifier
 65
         - subroutineName: identifier
 66
         - varName: identifier
 67
 68
         ## Statements
 69
 70
         - statements: statement*
 71
          - statement: letStatement | ifStatement | whileStatement | doStatement |
 72
 73
                      returnStatement
 74
          - letStatement: 'let' varName ('[' expression ']')? '=' expression ';'
         - ifStatement: 'if' '(' expression ')' '\{' statements '\}' ('else' '\{' else' '\}' else' '\}'
 75
                        statements '}')?
 76
         - whileStatement: 'while' '(' 'expression' ')' '{' statements '}'
 77
         - doStatement: 'do' subroutineCall ';'
 78
         - returnStatement: 'return' expression? ';'
 79
 80
 81
         ## Expressions
 82
 83
         - expression: term (op term)*
          - term: integerConstant | stringConstant | keywordConstant | varName |
 84
                 varName '['expression']' | subroutineCall | '(' expression ')' |
 85
                 unaryOp term
 86
          - subroutineCall: subroutineName '(' expressionList ')' | (className |
 87
                           varName) '.' subroutineName '(' expressionList ')
 88
         - expressionList: (expression (',' expression)*)?
 89
         - op: '+' | '-' | '** | '/' | '&' | '|' | '<' | '>' | '=' - unaryOp: '-' | '~' | '^' | '#'
 90
 91
         - keywordConstant: 'true' | 'false' | 'null' | 'this'
 92
 93
         Note that ^, # correspond to shiftleft and shiftright, respectively.
 94
 95
 96
         INITIAL_VAL = -1
 97
         EMPTY_STR = ""
 98
         EMPTY_LIST = []
 99
         NOT_FOUND = -1
100
         COMMENT_TYPE_1 = "//"
101
         COMMENT_TYPE_2 = "/*"
102
         COMMENT_TYPE_2_END = "*/"
103
         COMMENT_TYPE_3 = "/**"
104
105
         TOKEN_SYMBOLS = {"{", "}", "(", ")", "[", "]", ".", ",", ";", "+", "-", "*", "/", "&", "|", "<", ">", "=",
106
                           "~", "^", "#"}
107
108
         109
110
                            "while", "return"}
111
112
         KEYWORD = "KEYWORD"
113
         SYMBOL = "SYMBOL"
114
         IDENTIFIER = "IDENTIFIER"
115
         INT_CONST = "INT_CONST"
116
         STRING_CONST = "STRING_CONST"
117
118
119
         def __init__(self, input_stream: typing.TextIO) -> None:
120
              """Opens the input stream and gets ready to tokenize it.
121
122
                 input_stream (typing.TextIO): input stream.
123
124
125
             self.input_lines = input_stream.read().splitlines()
             self.n = self.INITIAL_VAL
126
127
             self.token_idx = self.INITIAL_VAL
```

```
128
              self.token_lst = self.EMPTY_LIST
129
         def has_more_tokens(self) -> bool:
130
              """Do we have more tokens in the input?
131
132
133
             Returns:
              bool: True if there are more tokens, False otherwise.
134
135
136
              if self.token_idx + 1 == len(self.token_lst):
                 self.token_idx = self.INITIAL_VAL
137
                  comment = False
138
139
                  while len(self.input_lines) - 1 != self.n:
                      self.n += 1
140
                      cur_token_line = self.input_lines[self.n].strip()
141
142
                      if cur_token_line != self.EMPTY_STR:
                          if cur_token_line[0:2] == "/*" and cur_token_line[-2:0] == "*/":
143
144
                              continue
                          if cur_token_line[0:2] == "/*" or cur_token_line[0:3] == "/**":
145
                              comment = True
146
                          if comment and cur_token_line[-2:] == "*/":
147
                              comment = False
148
149
                              continue
                          if not comment and cur_token_line[0:2] != "//":
150
151
                              return True
152
                  return False
153
              return True
154
155
         def __get_token_lst(self, line: str) -> typing.List[str]:
              token_line = line.replace('"', ' " ')
156
157
              temp_token_lst = list()
158
              for phrase in shlex.split(token_line, posix=False):
                  if phrase[0] == '"'
159
                      phrase = phrase[0] + phrase[2:-2] + phrase[-1]
160
161
                      temp_token_lst.append(phrase)
                  else:
162
163
                      for word in phrase.split():
                          if word in self.TOKEN_KEYWORDS:
164
165
                              temp_token_lst.append(word)
166
                              identifier = ""
167
168
                              for char in word:
                                  if char not in self.TOKEN_SYMBOLS:
169
                                      identifier += char
170
171
                                      if identifier != "":
172
                                           temp_token_lst.append(identifier)
173
174
                                           identifier = ""
                                      temp_token_lst.append(char)
175
                              if identifier != "":
176
177
                                  temp_token_lst.append(identifier)
             return temp_token_lst
178
179
180
         def advance(self) -> None:
              """Gets the next token from the input and makes it the current token.
181
              This method should be called if has_more_tokens() is true.
182
              Initially there is no current token.
183
184
              if self.token_idx == self.INITIAL_VAL:
185
                  cur_token_line = self.input_lines[self.n].strip()
186
187
188
                  inline_comments = [i for i in range(len(cur_token_line)) if
                                     cur_token_line.startswith(self.COMMENT_TYPE_1, i)]
189
190
                  for i in inline_comments:
                      if not self.__check_if_in_brackets(cur_token_line, i):
191
192
                          cur_token_line = cur_token_line[0:i]
193
194
195
                  inline_comments = [i for i in range(len(cur_token_line)) if
```

```
196
                                       cur_token_line.startswith(self.COMMENT_TYPE_2, i)]
197
                  for i in inline comments:
198
                       if not self.__check_if_in_brackets(cur_token_line, i):
                           inline_comment_idx_end = cur_token_line.find(self.COMMENT_TYPE_2_END)
199
                           if inline_comment_idx_end != self.NOT_FOUND:
200
201
                               cur_token_line = cur_token_line[0:i] + cur_token_line[inline_comment_idx_end + 2:]
202
                               break
203
204
                  inline_comment_idx = cur_token_line.find(self.COMMENT_TYPE_3)
                  if inline_comment_idx != self.NOT_FOUND:
205
                       cur_token_line = cur_token_line[0:inline_comment_idx]
206
207
208
                  self.token_lst = self.__get_token_lst(cur_token_line)
209
210
              self.token_idx += 1
211
212
          def __check_if_in_brackets(self, cur_token_line: str, idx: int) -> bool:
              brackets = [m.start() for m in re.finditer('"', cur_token_line)]
213
214
              i = 0
              if len(brackets) == 0:
215
                  return False
216
              while i + 2 <= len(brackets):</pre>
217
                  if brackets[i] < idx < brackets[i + 1]:</pre>
218
219
                      return True
                  i += 2
220
221
              return False
222
223
          def token_type(self) -> str:
224
225
              Returns:
226
                  str: the type of the current token, can be
                   "KEYWORD", "SYMBOL", "IDENTIFIER", "INT_CONST", "STRING_CONST"
227
228
229
              if self.token_lst[self.token_idx] in self.TOKEN_KEYWORDS:
                  return self.KEYWORD
230
231
232
              if self.token_lst[self.token_idx] in self.TOKEN_SYMBOLS:
                  return self.SYMBOL
233
234
              if self.token_lst[self.token_idx].isdecimal():
235
                  return self.INT_CONST
236
237
              if self.token_lst[self.token_idx][-1] == '"' and self.token_lst[self.token_idx][0] == '"':
238
239
                  return self.STRING_CONST
240
              return self.IDENTIFIER
241
          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".
                  Can return "CLASS", "METHOD", "FUNCTION", "CONSTRUCTOR", "INT",
248
                  "BOOLEAN", "CHAR", "VOID", "VAR", "STATIC", "FIELD", "LET", "DO",
"IF", "ELSE", "WHILE", "RETURN", "TRUE", "FALSE", "NULL", "THIS"
249
250
251
252
              return self.token_lst[self.token_idx]
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: '{' | '}' | '(' | ')' | '[' | ']' | '.' | ',' | ';' | '+' |
260
                     '-' | '*' | '/' | '&' | '|' | '<' | '>' | '=' | '~' | '^' | '#'
261
262
263
              return self.token_lst[self.token_idx]
```

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

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
        chmod a+x *
42
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
    {\it Unported~[License](https://creative commons.org/licenses/by-nc-sa/3.0/)}.
6
8
    import typing
9
10
    class SymbolTable:
11
         \hbox{\it """A symbol table that associates names with information needed for Jack}
12
         compilation: type, kind and running index. The symbol table has two nested
13
        scopes (class/subroutine).
14
15
16
        def __init__(self) -> None:
17
             """Creates a new empty symbol table."""
18
             self.index_table = {"STATIC": 0, "FIELD": 0, "ARG": 0, "VAR": 0}
19
20
21
             self.cur_symbol_table = {}
22
23
        def start_subroutine(self) -> None:
             """Starts a new subroutine scope (i.e., resets the subroutine's
24
25
            sumbol table).
26
27
            for key in self.index_table.keys():
                 self.index_table[key] = 0
28
29
30
            self.cur_symbol_table.clear()
31
         def define(self, name: str, var_type: str, kind: str) -> None:
              """Defines a new identifier of a given name, type and kind and assigns
33
             it a running index. "STATIC" and "FIELD" identifiers have a class scope,
34
            while "ARG" and "VAR" identifiers have a subroutine scope.
35
36
37
                name (str): the name of the new identifier.
38
39
                 var\_type (str): the type of the new identifier.
40
                 kind (str): the kind of the new identifier, can be:
                 "STATIC", "FIELD", "ARG", "VAR".
41
42
43
             self.cur_symbol_table[name] = [var_type, kind.upper(), self.index_table[kind.upper()]]
             self.index table[kind.upper()] += 1
44
45
        def var_count(self, kind: str) -> int:
46
47
             Args:
                kind (str): can be "STATIC", "FIELD", "ARG", "VAR".
49
50
51
                int: the number of variables of the given kind already defined in
52
53
                the current scope.
54
55
            var_count = 0
            for value in self.cur_symbol_table.values():
                if value[1] == kind.upper():
57
                    var_count += 1
            return var_count
```

```
60
        def kind_of(self, name: str) -> typing.Optional[typing.Any]:
61
62
63
            Args:
                name (str): name of an identifier.
64
65
66
                str: the kind of the named identifier in the current scope, or None
67
68
                if the identifier is unknown in the current scope.
69
            if name not in self.cur_symbol_table:
70
71
                return None
            return self.cur_symbol_table[name][1]
72
73
74
        def type_of(self, name: str) -> typing.Optional[typing.Any]:
75
76
                name (str): name of an identifier.
77
78
             str: the type of the named identifier in the current scope. """
80
81
            if name not in self.cur_symbol_table:
82
                return None
83
            return self.cur_symbol_table[name][0]
84
85
        def index_of(self, name: str) -> typing.Optional[typing.Any]:
86
87
            Args:
88
89
                name (str): name of an identifier.
90
91
            int: the index assigned to the named identifier. """
92
93
            if name not in self.cur_symbol_table:
94
95
                return None
            return self.cur_symbol_table[name][2]
```

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
    {\it Unported~[License](https://creative commons.org/licenses/by-nc-sa/3.0/)}.
6
8
    import typing
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_file = output_stream
19
            20
21
                                   "POINTER": "pointer", "TEMP": "temp"}
22
23
        def write_push(self, segment: str, index: int) -> None:
24
             """Writes a VM push command.
25
26
27
            Args:
                segment (str): the segment to push to, can be "CONST", "ARG",
28
29
                 "LOCAL", "STATIC", "THIS", "THAT", "POINTER", "TEMP"
                index (int): the index to push to.
30
31
            self.output_file.write("""push {segment} {index}\n""".format(segment=self.segment_table[segment],
                                                                           index=index))
33
34
        def write_pop(self, segment: str, index: int) -> None:
35
             """Writes a VM pop command.
36
37
38
            Args:
                segment (str): the segment to pop from, can be "CONST", "ARG", "LOCAL", "STATIC", "THIS", "THAT", "POINTER", "TEMP".
39
40
                index (int): the index to pop from.
41
42
            self.output_file.write("""pop {segment} {index}\n""".format(segment=self.segment_table[segment],
43
                                                                          index=index))
44
45
        def write_arithmetic(self, command: str) -> None:
46
             """Writes a VM arithmetic command.
47
48
49
            Aras:
                 command (str): the command to write, can be "ADD", "SUB", "NEG",
50
                 "EQ", "GT", "LT", "AND", "OR", "NOT", "SHIFTLEFT", "SHIFTRIGHT".
51
52
53
            if command.split()[0] == "call":
                self.output_file.write("""{command}\n""".format(command=command))
54
55
            else:
                self.output_file.write("""{command}\n""".format(command=command.lower()))
56
57
58
        def write_label(self, label: str) -> None:
             """Writes a VM label command.
59
```

```
60
61
             Args:
             label (str): the label to write.
62
63
             self.output_file.write("""label {label}\n""".format(label=label.upper()))
64
65
         def write_goto(self, label: str) -> None:
66
              """Writes a VM goto command.
67
68
69
             Args:
             label (str): the label to go to.
70
71
             self.output_file.write("""goto {label}\n""".format(label=label.upper()))
72
73
74
         def write_if(self, label: str) -> None:
              """Writes a VM if-goto command.
75
76
77
             Args:
             label (str): the label to go to.
78
79
             self.output_file.write("""if-goto {label}\n""".format(label=label.upper()))
80
81
         def write_call(self, name: str, n_args: int) -> None:
82
              """Writes a VM call command.
83
84
85
             Args:
                 name (str): the name of the function to call.
86
87
                 n_args (int): the number of arguments the function receives.
88
             self.output_file.write("""call {name} {n_args}\n""".format(name=name, n_args=n_args))
89
90
         def write_function(self, name: str, n_locals: int) -> None:
91
              """Writes\ a\ VM\ function\ command.
92
93
94
             Args:
95
                 name (str): the name of the function.
                 n_locals (int): the number of local variables the function uses.
96
97
             self.output_file.write("""function {name} {n_locals}\n""".format(name=name, n_locals=n_locals))
98
99
         def write_return(self) -> None:
100
              """Writes a VM return command."""
101
             self.output_file.write("return\n")
102
103
         def close(self) -> None:
104
             self.output_file.close()
105
```