

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

```
1 ***** TESTING FOLDER STRUCTURE START *****
2 Checking your submission for presence of invalid (non-ASCII) characters...
3 No invalid characters found.
4 Submission logins are: linorcohen
5 Is this OK?
6 ***** TESTING FOLDER STRUCTURE END *****
7
8 ***** PROJECT TEST START *****
9 Running 'make'.
10 'make' ran successfully.
11 Testing.
12
13 Running your program with command: 'JackCompiler tst/ComplexArrays'.
14 Main.vm was created in test ComplexArrays.
15 Checking validity of generated VM code.
16 Generated VM code passed the test successfully.
17
18 Running your program with command: 'JackCompiler tst/Seven'.
19 Main.vm was created in test Seven.
20 Checking validity of generated VM code.
21 Generated VM code passed the test successfully.
22 ***** PROJECT TEST END *****
23
24 Note: the tests you see above are all the presubmission tests
25 for this project. The tests might not check all the different
26 parts of the project or all corner cases, so write your own
27 tests and use them!
```

2 AUTHORS

1 linorcohen
2 Partner 1: Linor Cohen, linor.cohen@mail.huji.ac.il, 318861226
3 Remarks:

3 CompilationEngine.py

```
1  """
2  This file is part of nand2tetris, as taught in The Hebrew University, and
3  was written by Aviv Yaish. It is an extension to the specifications given
4  [here](https://www.nand2tetris.org) (Shimon Schocken and Noam Nisan, 2017),
5  as allowed by the Creative Common Attribution-NonCommercial-ShareAlike 3.0
6  Unported [License](https://creativecommons.org/licenses/by-nc-sa/3.0/).
7  """
8  import typing
9  import JackTokenizer
10 from SymbolTable import SymbolTable
11 import VMWriter
12
13
14 class CompilationEngine:
15     """Gets input from a JackTokenizer and emits its parsed structure into an
16     output stream.
17     """
18
19     FUNCTION = "function"
20     CONSTRUCTOR = "constructor"
21     METHOD = "method"
22     STATIC = "static"
23     FIELD = "field"
24     RETURN = "return"
25     WHILE = "while"
26     LET = "let"
27     DO = "do"
28     IF = "if"
29     ELSE = "else"
30
31     KEYWORD = "KEYWORD"
32     SYMBOL = "SYMBOL"
33     IDENTIFIER = "IDENTIFIER"
34     INT_CONST = "INT_CONST"
35     STRING_CONST = "STRING_CONST"
36
37     def __init__(self, input_stream: "JackTokenizer", class_symbol_table: "SymbolTable",
38                 vm_writer: "VMWriter", output_stream: typing.TextIO) -> None:
39         """
40         Creates a new compilation engine with the given input and output. The
41         next routine called must be compileClass()
42         :param input_stream: The input stream.
43         :param output_stream: The output stream.
44         """
45         self.class_symbol_table = class_symbol_table
46         self.vm_writer = vm_writer
47         self.tokenizer = input_stream
48         self.output_stream = output_stream
49
50         self.op_terms = {"+": "ADD", "-": "SUB", "*": "call Math.multiply 2", "/": "call Math.divide 2",
51                         "&": "AND", "|": "OR", "<": "LT", ">": "GT", "=": "EQ"}
52         self.unary_op_terms = {"^~": "SHIFTLLEFT", "#": "SHIFTRIGHT", "-": "NEG", "~": "NOT"}
53
54         self.class_name = ""
55         self.counter = 0
56         self.subroutine_symbol_table = SymbolTable()
57
58     def __advance_tokenizer(self) -> None:
59         """
```

```

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

```

```

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

```

```

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

```

```

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

```



```

332         self.vm_writer.write_push(var_kind, var_index)
333         # expression
334         self.__get_current_token_and_advance()
335         self.compile_expression()
336         # ]
337         self.__get_current_token_and_advance()
338         # add
339         self.vm_writer.write_arithmetic("ADD")
340         is_array = True
341     # symbol
342     self.__get_current_token()
343     # expression
344     self.__get_current_token_and_advance()
345     self.compile_expression()
346     # ;
347     self.__get_current_token()
348     if is_array:
349         # pop temp 0
350         self.vm_writer.write_pop("TEMP", 0)
351         # pop pointer 1
352         self.vm_writer.write_pop("POINTER", 1)
353         # push temp 0
354         self.vm_writer.write_push("TEMP", 0)
355         # pop that 0
356         self.vm_writer.write_pop("THAT", 0)
357     else:
358         self.vm_writer.write_pop(var_kind, var_index)
359
360 def compile_while(self, subroutine_type: str) -> None:
361     """Compiles a while statement."""
362     # keyword = while
363     self.__get_current_token()
364     # label L1
365     l1 = f"#{self.class_name}_L_{self.counter}"
366     self.counter += 1
367     self.vm_writer.write_label(l1)
368     # (
369     self.__get_current_token_and_advance()
370     # expression
371     self.__get_current_token_and_advance()
372     self.compile_expression()
373     # )
374     self.__get_current_token()
375     # not
376     self.vm_writer.write_arithmetic("NOT")
377     # if-goto L2
378     l2 = f"#{self.class_name}_L_{self.counter}"
379     self.counter += 1
380     self.vm_writer.write_if(l2)
381     # {
382     # statements
383     self.__get_current_token_and_advance()
384     self.compile_statements(subroutine_type)
385     # }
386     self.__get_current_token()
387     # goto L1
388     self.vm_writer.write_goto(l1)
389     # label L2
390     self.vm_writer.write_label(l2)
391
392 def compile_return(self, subroutine_type: str) -> None:
393     """Compiles a return statement."""
394     # keyword = return
395     self.__get_current_token()
396     # expression -> ?
397     symbol = self.__get_current_token_and_advance()
398     if symbol != ";":
399         # expression

```

```

400         self.compile_expression()
401     else:
402         if subroutine_type == self.CONSTRUCTOR:
403             self.vm_writer.write_push("POINTER", 0)
404         else:
405             self.vm_writer.write_push("CONST", 0)
406     self.vm_writer.write_return()
407
408 def compile_if(self, subroutine_type: str) -> None:
409     """Compiles a if statement, possibly with a trailing else clause."""
410     # keyword = if
411     self.__get_current_token()
412     # (
413     self.__get_current_token_and_advance()
414     # expression
415     self.__get_current_token_and_advance()
416     self.compile_expression()
417     # )
418     self.__get_current_token()
419     # not
420     self.vm_writer.write_arithmetic("NOT")
421     # if-goto L1
422     l1 = f"#{self.class_name}_L_{self.counter}"
423     self.counter += 1
424     self.vm_writer.write_if(l1)
425     # {
426     self.__get_current_token_and_advance()
427     # statements
428     self.__get_current_token_and_advance()
429     self.compile_statements(subroutine_type)
430     # }
431     self.__get_current_token()
432     # goto L2
433     l2 = f"#{self.class_name}_L_{self.counter}"
434     self.counter += 1
435     self.vm_writer.write_goto(l2)
436     # label L1
437     self.vm_writer.write_label(l1)
438     # else -> ?
439     token = self.__get_current_token_and_advance()
440     if token == self.ELSE:
441         # {
442         self.__get_current_token_and_advance()
443         # statements
444         self.__get_current_token_and_advance()
445         self.compile_statements(subroutine_type)
446         # }
447         self.__get_current_token()
448         self.__get_current_token_and_advance()
449     # label L2
450     self.vm_writer.write_label(l2)
451
452 def compile_expression(self) -> None:
453     """Compiles an expression."""
454     # term
455     self.compile_term()
456     # term -> *
457     token = self.__get_current_token()
458     while token != ")":
459         if token not in self.op_terms:
460             break
461         # op
462         op = token
463         # term
464         self.__get_current_token_and_advance()
465         # term
466         self.compile_term()
467         # output "op"

```

```

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

```

```

536         elif token in {".", "("}:
537             self.__subroutine_call_format(function_call_name, True)
538             self.__get_current_token_and_advance()
539
540     def compile_expression_list(self) -> int:
541         """Compiles a (possibly empty) comma-separated list of expressions."""
542         n = 0
543         # expression -> ?
544         token = self.__get_current_token()
545         while token != ")":
546             # expression
547             self.compile_expression()
548             n += 1
549             token = self.__get_current_token()
550             if token == ",":
551                 token = self.__get_current_token_and_advance()
552         return n

```

4 JackCompiler

```
1  #!/bin/sh
2  # This file only works on Unix-like operating systems, so it won't work on Windows.
3
4  ## Why do we need this file?
5  # The purpose of this file is to run your project.
6  # We want our users to have a simple API to run the project.
7  # So, we need a "wrapper" that will hide all details to do so,
8  # enabling users to simply type 'JackCompiler <path>' in order to use it.
9
10 ## What are '#!/bin/sh' and '$*'?
11 # '$*' is a variable that holds all the arguments this file has received. So, if you
12 # run "JackCompiler trout mask replica", $* will hold "trout mask replica".
13
14 ## What should I change in this file to make it work with my project?
15 # IMPORTANT: This file assumes that the main is contained in "JackCompiler.py".
16 #           If your main is contained elsewhere, you will need to change this.
17
18 python3 JackCompiler.py $*
19
20 # This file is part of nand2tetris, as taught in The Hebrew University, and
21 # was written by Aviv Yaish. It is an extension to the specifications given
22 # in https://www.nand2tetris.org (Shimon Schocken and Noam Nisan, 2017),
23 # as allowed by the Creative Commons Attribution-NonCommercial-ShareAlike 3.0
24 # Unported License: https://creativecommons.org/licenses/by-nc-sa/3.0/
```

5 JackCompiler.py

```
1  """
2  This file is part of nand2tetris, as taught in The Hebrew University, and
3  was written by Aviv Yaish. It is an extension to the specifications given
4  [here](https://www.nand2tetris.org) (Shimon Schocken and Noam Nisan, 2017),
5  as allowed by the Creative Common Attribution-NonCommercial-ShareAlike 3.0
6  Unported [License](https://creativecommons.org/licenses/by-nc-sa/3.0/).
7  """
8  import os
9  import sys
10 import typing
11 from CompilationEngine import CompilationEngine
12 from JackTokenizer import JackTokenizer
13 from SymbolTable import SymbolTable
14 from VMWriter import VMWriter
15
16
17 def compile_file(
18     input_file: typing.TextIO, output_file: typing.TextIO) -> None:
19     """Compiles a single file.
20
21     Args:
22         input_file (typing.TextIO): the file to compile.
23         output_file (typing.TextIO): writes all output to this file.
24     """
25     tokenizer = JackTokenizer(input_file)
26     vm_writer = VMWriter(output_file)
27     class_symbol_table = SymbolTable()
28     engine = CompilationEngine(tokenizer, class_symbol_table, vm_writer, output_file)
29     engine.compile_class()
30
31     output_file.close()
32
33
34 if "__main__" == __name__:
35     # Parses the input path and calls compile_file on each input file.
36     # This opens both the input and the output files!
37     # Both are closed automatically when the code finishes running.
38     # If the output file does not exist, it is created automatically in the
39     # correct path, using the correct filename.
40     if not len(sys.argv) == 2:
41         sys.exit("Invalid usage, please use: JackCompiler <input path>")
42     argument_path = os.path.abspath(sys.argv[1])
43     if os.path.isdir(argument_path):
44         files_to_assemble = [
45             os.path.join(argument_path, filename)
46             for filename in os.listdir(argument_path)]
47     else:
48         files_to_assemble = [argument_path]
49     for input_path in files_to_assemble:
50         filename, extension = os.path.splitext(input_path)
51         if extension.lower() != ".jack":
52             continue
53         output_path = filename + ".vm"
54         with open(input_path, 'r') as input_file, \
55             open(output_path, 'w') as output_file:
56             compile_file(input_file, output_file)
```

6 JackTokenizer.py

```
1  """
2  This file is part of nand2tetris, as taught in The Hebrew University, and
3  was written by Aviv Yaish. It is an extension to the specifications given
4  [here](https://www.nand2tetris.org) (Shimon Schocken and Noam Nisan, 2017),
5  as allowed by the Creative Common Attribution-NonCommercial-ShareAlike 3.0
6  Unported [License](https://creativecommons.org/licenses/by-nc-sa/3.0/).
7  """
8  import re
9  import typing
10 import shlex
11
12
13 class JackTokenizer:
14     """Removes all comments from the input stream and breaks it
15     into Jack language tokens, as specified by the Jack grammar.
16
17     # Jack Language Grammar
18
19     A Jack file is a stream of characters. If the file represents a
20     valid program, it can be tokenized into a stream of valid tokens. The
21     tokens may be separated by an arbitrary number of whitespace characters,
22     and comments, which are ignored. There are three possible comment formats:
23     /* comment until closing */ , /** API comment until closing */ , and
24     // comment until the line's end.
25
26     - 'xxx': quotes are used for tokens that appear verbatim ('terminals').
27     - xxx: regular typeface is used for names of language constructs
28         ('non-terminals').
29     - (): parentheses are used for grouping of language constructs.
30     - x | y: indicates that either x or y can appear.
31     - x?: indicates that x appears 0 or 1 times.
32     - x*: indicates that x appears 0 or more times.
33
34     ## Lexical Elements
35
36     The Jack language includes five types of terminal elements (tokens).
37
38     - keyword: 'class' | 'constructor' | 'function' | 'method' | 'field' |
39               'static' | 'var' | 'int' | 'char' | 'boolean' | 'void' | 'true' |
40               'false' | 'null' | 'this' | 'let' | 'do' | 'if' | 'else' |
41               'while' | 'return'
42     - symbol: '{' | '}' | '(' | ')' | '[' | ']' | '.' | ',' | ';' | '+' |
43               '-' | '*' | '/' | '%' | '|' | '<' | '>' | '=' | '~' | '^' | '#'
44     - integerConstant: A decimal number in the range 0-32767.
45     - StringConstant: '"' A sequence of Unicode characters not including
46                       double quote or newline '"'
47     - identifier: A sequence of letters, digits, and underscore ('_') not
48                  starting with a digit. You can assume keywords cannot be
49                  identifiers, so 'self' cannot be an identifier, etc'.
50
51     ## Program Structure
52
53     A Jack program is a collection of classes, each appearing in a separate
54     file. A compilation unit is a single class. A class is a sequence of tokens
55     structured according to the following context free syntax:
56
57     - class: 'class' className '{' classVarDec* subroutineDec* '}'
58     - classVarDec: ('static' | 'field') type varName (',' varName)* ';'
59     - type: 'int' | 'char' | 'boolean' | className
```

```

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

```



```

128     self.token_lst = self.EMPTY_LIST
129
130 def has_more_tokens(self) -> bool:
131     """Do we have more tokens in the input?"""
132
133     Returns:
134         bool: True if there are more tokens, False otherwise.
135     """
136     if self.token_idx + 1 == len(self.token_lst):
137         self.token_idx = self.INITIAL_VAL
138         comment = False
139         while len(self.input_lines) - 1 != self.n:
140             self.n += 1
141             cur_token_line = self.input_lines[self.n].strip()
142             if cur_token_line != self.EMPTY_STR:
143                 if cur_token_line[0:2] == "/*" and cur_token_line[-2:0] == "*/":
144                     continue
145                 if cur_token_line[0:2] == "/*" or cur_token_line[0:3] == "/*":
146                     comment = True
147                 if comment and cur_token_line[-2:] == "*/":
148                     comment = False
149                     continue
150                 if not comment and cur_token_line[0:2] != "//":
151                     return True
152             return False
153     return True
154
155 def __get_token_lst(self, line: str) -> typing.List[str]:
156     token_line = line.replace("'", ' " ')
157     temp_token_lst = list()
158     for phrase in shlex.split(token_line, posix=False):
159         if phrase[0] == "'":
160             phrase = phrase[0] + phrase[2:-2] + phrase[-1]
161             temp_token_lst.append(phrase)
162         else:
163             for word in phrase.split():
164                 if word in self.TOKEN_KEYWORDS:
165                     temp_token_lst.append(word)
166                 else:
167                     identifier = ""
168                     for char in word:
169                         if char not in self.TOKEN_SYMBOLS:
170                             identifier += char
171                     else:
172                         if identifier != "":
173                             temp_token_lst.append(identifier)
174                             identifier = ""
175                         temp_token_lst.append(char)
176                     if identifier != "":
177                         temp_token_lst.append(identifier)
178     return temp_token_lst
179
180 def advance(self) -> None:
181     """Gets the next token from the input and makes it the current token.
182     This method should be called if has_more_tokens() is true.
183     Initially there is no current token.
184     """
185     if self.token_idx == self.INITIAL_VAL:
186         cur_token_line = self.input_lines[self.n].strip()
187
188         inline_comments = [i for i in range(len(cur_token_line)) if
189                             cur_token_line.startswith(self.COMMENT_TYPE_1, i)]
190         for i in inline_comments:
191             if not self.__check_if_in_brackets(cur_token_line, i):
192                 cur_token_line = cur_token_line[0:i]
193                 break
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):
199             inline_comment_idx_end = cur_token_line.find(self.COMMENT_TYPE_2_END)
200             if inline_comment_idx_end != self.NOT_FOUND:
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)
205     if inline_comment_idx != self.NOT_FOUND:
206         cur_token_line = cur_token_line[0:inline_comment_idx]
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:
213     brackets = [m.start() for m in re.finditer('(', cur_token_line)]
214     i = 0
215     if len(brackets) == 0:
216         return False
217     while i + 2 <= len(brackets):
218         if brackets[i] < idx < brackets[i + 1]:
219             return True
220         i += 2
221     return False
222
223 def token_type(self) -> str:
224     """
225     Returns:
226         str: the type of the current token, can be
227             "KEYWORD", "SYMBOL", "IDENTIFIER", "INT_CONST", "STRING_CONST"
228     """
229     if self.token_lst[self.token_idx] in self.TOKEN_KEYWORDS:
230         return self.KEYWORD
231
232     if self.token_lst[self.token_idx] in self.TOKEN_SYMBOLS:
233         return self.SYMBOL
234
235     if self.token_lst[self.token_idx].isdecimal():
236         return self.INT_CONST
237
238     if self.token_lst[self.token_idx][-1] == '"' and self.token_lst[self.token_idx][0] == '"':
239         return self.STRING_CONST
240
241     return self.IDENTIFIER
242
243 def keyword(self) -> str:
244     """
245     Returns:
246         str: the keyword which is the current token.
247             Should be called only when token_type() is "KEYWORD".
248             Can return "CLASS", "METHOD", "FUNCTION", "CONSTRUCTOR", "INT",
249             "BOOLEAN", "CHAR", "VOID", "VAR", "STATIC", "FIELD", "LET", "DO",
250             "IF", "ELSE", "WHILE", "RETURN", "TRUE", "FALSE", "NULL", "THIS"
251     """
252     return self.token_lst[self.token_idx]
253
254 def symbol(self) -> str:
255     """
256     Returns:
257         str: the character which is the current token.
258             Should be called only when token_type() is "SYMBOL".
259             Recall that symbol was defined in the grammar like so:
260             symbol: '{' | '}' | '(' | ')' | '[' | ']' | '.' | ',' | ';' | '+' |
261             '-' | '*' | '/' | '%' | '|' | '<' | '>' | '=' | '~' | '^' | '#'
262     """
263     return self.token_lst[self.token_idx]

```

```

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

```

7 Makefile

```
1  # Makefile for a script (e.g. Python)
2
3  ## Why do we need this file?
4  # We want our users to have a simple API to run the project.
5  # So, we need a "wrapper" that will hide all details to do so,
6  # thus enabling our users to simply type 'JackCompiler <path>' in order to use it.
7
8  ## What are makefiles?
9  # This is a sample makefile.
10 # The purpose of makefiles is to make sure that after running "make" your
11 # project is ready for execution.
12
13 ## What should I change in this file to make it work with my project?
14 # Usually, scripting language (e.g. Python) based projects only need execution
15 # permissions for your run file executable to run.
16 # Your project may be more complicated and require a different makefile.
17
18 ## What is a makefile rule?
19 # A makefile rule is a list of prerequisites (other rules that need to be run
20 # before this rule) and commands that are run one after the other.
21 # The "all" rule is what runs when you call "make".
22 # In this example, all it does is grant execution permissions for your
23 # executable, so your project will be able to run on the graders' computers.
24 # In this case, the "all" rule has no prerequisites.
25
26 ## How are rules defined?
27 # The following line is a rule declaration:
28 # all:
29 #     chmod a+x JackCompiler
30
31 # A general rule looks like this:
32 # rule_name: prerequisite1 prerequisite2 prerequisite3 prerequisite4 ...
33 #     command1
34 #     command2
35 #     command3
36 #     ...
37 # Where each prerequisite is a rule name, and each command is a command-line
38 # command (for example chmod, javac, echo, etc').
39
40 # Beginning of the actual Makefile
41 all:
42     chmod a+x *
43
44 # This file is part of nand2tetris, as taught in The Hebrew University, and
45 # was written by Aviv Yaish. It is an extension to the specifications given
46 # in https://www.nand2tetris.org (Shimon Schocken and Noam Nisan, 2017),
47 # as allowed by the Creative Commons Attribution-NonCommercial-ShareAlike 3.0
48 # Unported License: https://creativecommons.org/licenses/by-nc-sa/3.0/
```

8 SymbolTable.py

```
1  """
2  This file is part of nand2tetris, as taught in The Hebrew University, and
3  was written by Aviv Yaish. It is an extension to the specifications given
4  [here](https://www.nand2tetris.org) (Shimon Schocken and Noam Nisan, 2017),
5  as allowed by the Creative Common Attribution-NonCommercial-ShareAlike 3.0
6  Unported [License](https://creativecommons.org/licenses/by-nc-sa/3.0/).
7  """
8  import typing
9
10
11  class SymbolTable:
12      """A symbol table that associates names with information needed for Jack
13      compilation: type, kind and running index. The symbol table has two nested
14      scopes (class/subroutine).
15      """
16
17      def __init__(self) -> None:
18          """Creates a new empty symbol table."""
19          self.index_table = {"STATIC": 0, "FIELD": 0, "ARG": 0, "VAR": 0}
20
21          self.cur_symbol_table = {}
22
23      def start_subroutine(self) -> None:
24          """Starts a new subroutine scope (i.e., resets the subroutine's
25          symbol table).
26          """
27          for key in self.index_table.keys():
28              self.index_table[key] = 0
29
30          self.cur_symbol_table.clear()
31
32      def define(self, name: str, var_type: str, kind: str) -> None:
33          """Defines a new identifier of a given name, type and kind and assigns
34          it a running index. "STATIC" and "FIELD" identifiers have a class scope,
35          while "ARG" and "VAR" identifiers have a subroutine scope.
36
37          Args:
38              name (str): the name of the new identifier.
39              var_type (str): the type of the new identifier.
40              kind (str): the kind of the new identifier, can be:
41                  "STATIC", "FIELD", "ARG", "VAR".
42          """
43          self.cur_symbol_table[name] = [var_type, kind.upper(), self.index_table[kind.upper()]]
44          self.index_table[kind.upper()] += 1
45
46      def var_count(self, kind: str) -> int:
47          """
48          Args:
49              kind (str): can be "STATIC", "FIELD", "ARG", "VAR".
50
51          Returns:
52              int: the number of variables of the given kind already defined in
53                  the current scope.
54          """
55          var_count = 0
56          for value in self.cur_symbol_table.values():
57              if value[1] == kind.upper():
58                  var_count += 1
59          return var_count
```

```

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

```

9 VMWriter.py

```
1  """
2  This file is part of nand2tetris, as taught in The Hebrew University, and
3  was written by Aviv Yaish. It is an extension to the specifications given
4  [here](https://www.nand2tetris.org) (Shimon Schocken and Noam Nisan, 2017),
5  as allowed by the Creative Common Attribution-NonCommercial-ShareAlike 3.0
6  Unported [License](https://creativecommons.org/licenses/by-nc-sa/3.0/).
7  """
8  import typing
9
10
11 class VMWriter:
12     """
13     Writes VM commands into a file. Encapsulates the VM command syntax.
14     """
15
16     def __init__(self, output_stream: typing.TextIO) -> None:
17         """Creates a new file and prepares it for writing VM commands."""
18         self.output_file = output_stream
19
20         self.segment_table = {"CONST": "constant", "ARG": "argument", "VAR": "local",
21                               "STATIC": "static", "FIELD": "this", "THAT": "that",
22                               "POINTER": "pointer", "TEMP": "temp"}
23
24     def write_push(self, segment: str, index: int) -> None:
25         """Writes a VM push command.
26
27         Args:
28             segment (str): the segment to push to, can be "CONST", "ARG",
29                           "LOCAL", "STATIC", "THIS", "THAT", "POINTER", "TEMP"
30             index (int): the index to push to.
31         """
32         self.output_file.write("push {segment} {index}\n".format(segment=self.segment_table[segment],
33                                                                    index=index))
34
35     def write_pop(self, segment: str, index: int) -> None:
36         """Writes a VM pop command.
37
38         Args:
39             segment (str): the segment to pop from, can be "CONST", "ARG",
40                           "LOCAL", "STATIC", "THIS", "THAT", "POINTER", "TEMP".
41             index (int): the index to pop from.
42         """
43         self.output_file.write("pop {segment} {index}\n".format(segment=self.segment_table[segment],
44                                                                    index=index))
45
46     def write_arithmetic(self, command: str) -> None:
47         """Writes a VM arithmetic command.
48
49         Args:
50             command (str): the command to write, can be "ADD", "SUB", "NEG",
51                           "EQ", "GT", "LT", "AND", "OR", "NOT", "SHIFTLEFT", "SHIFTRIGHT".
52         """
53         if command.split()[0] == "call":
54             self.output_file.write("call {command}\n".format(command=command))
55         else:
56             self.output_file.write("{command}\n".format(command=command.lower()))
57
58     def write_label(self, label: str) -> None:
59         """Writes a VM label command.
```

```

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

```