

Contents

1	Basic Test Results	2
2	AUTHORS	3
3	CompilationEngine.py	4
4	JackAnalyzer	12
5	JackAnalyzer.py	13
6	JackTokenizer.py	14
7	Makefile	19

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 Running your program with command: 'JackAnalyzer tst/ArrayTest'.
13 Main.xml was created in test ArrayTest.
14 The diff is OK on the file Main.xml in test ArrayTest.
15 Running your program with command: 'JackAnalyzer tst/Square'.
16 Main.xml was created in test Square.
17 The diff is OK on the file Main.xml in test Square.
18 SquareGame.xml was created in test Square.
19 The diff is OK on the file SquareGame.xml in test Square.
20 Square.xml was created in test Square.
21 The diff is OK on the file Square.xml in test Square.
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
11
12  class CompilationEngine:
13      """Gets input from a JackTokenizer and emits its parsed structure into an
14      output stream.
15      """
16
17      FUNCTION = "function"
18      CONSTRUCTOR = "constructor"
19      METHOD = "method"
20      STATIC = "static"
21      FIELD = "field"
22      RETURN = "return"
23      WHILE = "while"
24      LET = "let"
25      DO = "do"
26      IF = "if"
27      ELSE = "else"
28
29      KEYWORD = "KEYWORD"
30      SYMBOL = "SYMBOL"
31      IDENTIFIER = "IDENTIFIER"
32      INT_CONST = "INT_CONST"
33      STRING_CONST = "STRING_CONST"
34
35      CLASS_TAG = "class"
36      CLASS_VAR_DEC_TAG = "classVarDec"
37      SUBROUTINE_DEC_TAG = "subroutineDec"
38      SUBROUTINE_BODY_TAG = "subroutineBody"
39      VAR_DEC_TAG = "varDec"
40      PARAMETER_LIST_TAG = "parameterList"
41      STATEMENTS_TAG = "statements"
42      EXPRESSION_LIST_TAG = "expressionList"
43      TERM_TAG = "term"
44      EXPRESSION_TAG = "expression"
45      IF_STATEMENT_TAG = "ifStatement"
46      RETURN_STATEMENT_TAG = "returnStatement"
47      WHILE_STATEMENT_TAG = "whileStatement"
48      LET_STATEMENT_TAG = "letStatement"
49      DO_STATEMENT_TAG = "doStatement"
50
51      KEYWORD_TAG = "keyword"
52      SYMBOL_TAG = "symbol"
53      IDENTIFIER_TAG = "identifier"
54      INT_CONST_TAG = "integerConstant"
55      STRING_CONST_TAG = "stringConstant"
56
57      def __init__(self, input_stream: JackTokenizer, output_stream: typing.TextIO) -> None:
58          """
59          Creates a new compilation engine with the given input and output. The
```

```

60         next routine called must be compileClass()
61         :param input_stream: The input stream.
62         :param output_stream: The output stream.
63         """
64         self.tokenizer = input_stream
65         self.output_stream = output_stream
66
67         self.op_terms = {"+", "-", "*", "/", "&", "|", "<", ">", "="}
68         self.unary_op_terms = {"^", "#", "_", "~"}
69
70         self.indentation = ""
71
72     def __write_open_tag(self, tag: str) -> None:
73         """
74         this method writes the open tag to the output file
75         :param tag: given token tag
76         """
77         self.output_stream.write(self.indentation + "<" + tag + ">")
78
79     def __write_close_tag(self, tag: str) -> None:
80         """
81         this method writes the closing tag to the output file
82         :param tag: given token tag
83         """
84         self.output_stream.write(self.indentation + "</" + tag + ">")
85         self.output_stream.write("\n")
86
87     def __write_open_and_close_tag(self, tag: str, token: str) -> None:
88         """
89         this method writes open and close tag to the output file, used for inline tags
90         :param tag: given token tag
91         :param token: given token
92         """
93         self.__write_open_tag(tag)
94         if token == "<":
95             token = "&lt;";
96         elif token == ">":
97             token = "&gt;";
98         elif token == "&":
99             token = "&amp;";
100         self.output_stream.write(" " + token)
101         self.output_stream.write(" </" + tag + ">")
102         self.output_stream.write("\n")
103
104     def __advance_tokenizer(self) -> None:
105         """
106         this method advance the tokenizer if has more tokens
107         """
108         if self.tokenizer.has_more_tokens():
109             self.tokenizer.advance()
110
111     def __get_current_token_and_advance(self) -> typing.Tuple[str, str]:
112         """
113         this method advance the token and get the current token
114         :return: Tuple(token, token tag type)
115         """
116         self.__advance_tokenizer()
117         return self.__get_current_token()
118
119     def __get_current_token(self) -> typing.Tuple[str, str]:
120         """
121         this method return the tuple of the current token and the current token type tag.
122         :return: Tuple(token, token tag type)
123         """
124         t_type = self.tokenizer.token_type()
125         if t_type == self.KEYWORD:
126             return self.tokenizer.keyword(), self.KEYWORD_TAG
127         elif t_type == self.SYMBOL:

```

```

128         return self.tokenizer.symbol(), self.SYMBOL_TAG
129     elif t_type == self.IDENTIFIER:
130         return self.tokenizer.identifier(), self.IDENTIFIER_TAG
131     elif t_type == self.INT_CONST:
132         return str(self.tokenizer.int_val()), self.INT_CONST_TAG
133     elif t_type == self.STRING_CONST:
134         return self.tokenizer.string_val(), self.STRING_CONST_TAG
135
136 def compile_class(self) -> None:
137     """Compiles a complete class."""
138     self.__write_open_tag(self.CLASS_TAG)
139     self.output_stream.write("\n")
140     self.indentation += " "
141     # class
142     self.__write_next_advanced_token()
143     # className
144     self.__write_next_advanced_token()
145     # {
146     self.__write_next_advanced_token()
147     # classVarDec -> *
148     token, token_type = self.__get_current_token_and_advance()
149     while token in {self.FIELD, self.STATIC}:
150         self.compile_class_var_dec()
151         token, token_type = self.__get_current_token_and_advance()
152     # subroutineDec -> *
153     while token in {self.METHOD, self.CONSTRUCTOR, self.FUNCTION}:
154         self.compile_subroutine()
155         token, token_type = self.__get_current_token_and_advance()
156     # }
157     self.__write_open_and_close_tag(token_type, token)
158     self.indentation = self.indentation[:-2]
159     self.__write_close_tag(self.CLASS_TAG)
160
161 def __write_next_advanced_token(self) -> None:
162     """
163     this method advance the token and writs the open close tag of the current token
164     """
165     token, token_type = self.__get_current_token_and_advance()
166     self.__write_open_and_close_tag(token_type, token)
167
168 def __writes_current_token(self) -> None:
169     """
170     this method writes the current token without advancing the tokenizer
171     """
172     token, token_type = self.__get_current_token()
173     self.__write_open_and_close_tag(token_type, token)
174
175 def compile_class_var_dec(self) -> None:
176     """Compiles a static declaration or a field declaration."""
177     self.__write_open_tag(self.CLASS_VAR_DEC_TAG)
178     self.indentation += " "
179     self.output_stream.write("\n")
180     # field or static
181     self.__writes_current_token()
182     # type
183     token, token_type = self.__get_current_token_and_advance()
184     self.__write_open_and_close_tag(token_type, token)
185     # varName -> *
186     while token != ";":
187         # identifier
188         self.__write_next_advanced_token()
189         # symbol
190         token, token_type = self.__get_current_token_and_advance()
191         self.__write_open_and_close_tag(token_type, token)
192     self.indentation = self.indentation[:-2]
193     self.__write_close_tag(self.CLASS_VAR_DEC_TAG)
194
195 def compile_subroutine(self) -> None:

```

```

196         """
197         Compiles a complete method, function, or constructor.
198         You can assume that classes with constructors have at least one field,
199         you will understand why this is necessary in project 11.
200         """
201         self.__write_open_tag(self.SUBROUTINE_DEC_TAG)
202         self.indentation += " "
203         self.output_stream.write("\n")
204         # keyword
205         self.__writes_current_token()
206         # identifier
207         self.__write_next_advanced_token()
208         # identifier
209         self.__write_next_advanced_token()
210         # (
211         self.__write_next_advanced_token()
212         # parameter list
213         self.compile_parameter_list()
214         # )
215         self.__writes_current_token()
216         # subroutine body
217         self.__compile_subroutine_body()
218         self.indentation = self.indentation[:-2]
219         self.__write_close_tag(self.SUBROUTINE_DEC_TAG)
220
221     def __compile_subroutine_body(self) -> None:
222         """
223         this method compile a subroutine body
224         """
225         self.__write_open_tag(self.SUBROUTINE_BODY_TAG)
226         self.indentation += " "
227         self.output_stream.write("\n")
228         # {
229         self.__write_next_advanced_token()
230         # var -> *
231         token, token_type = self.__get_current_token_and_advance()
232         while token == "var":
233             self.compile_var_dec()
234             token, token_type = self.__get_current_token_and_advance()
235         # statements
236         self.compile_statements()
237         # }
238         self.__writes_current_token()
239         self.indentation = self.indentation[:-2]
240         self.__write_close_tag(self.SUBROUTINE_BODY_TAG)
241
242     def compile_parameter_list(self) -> None:
243         """Compiles a (possibly empty) parameter list, not including the
244         enclosing "()""""
245         """
246         self.__write_open_tag(self.PARAMETER_LIST_TAG)
247         self.indentation += " "
248         self.output_stream.write("\n")
249         # varName -> *
250         token, token_type = self.__get_current_token_and_advance()
251         while token != ")":
252             self.__write_open_and_close_tag(token_type, token)
253             token, token_type = self.__get_current_token_and_advance()
254         self.indentation = self.indentation[:-2]
255         self.__write_close_tag(self.PARAMETER_LIST_TAG)
256
257     def compile_var_dec(self) -> None:
258         """Compiles a var declaration."""
259         self.__write_open_tag(self.VAR_DEC_TAG)
260         self.indentation += " "
261         self.output_stream.write("\n")
262         # keyword
263         self.__writes_current_token()

```

```

264     # identifier
265     token, token_type = self.__get_current_token_and_advance()
266     self.__write_open_and_close_tag(token_type, token)
267     # varName -> *
268     while token != ";":
269         # identifier
270         self.__write_next_advanced_token()
271         # symbol
272         token, token_type = self.__get_current_token_and_advance()
273         self.__write_open_and_close_tag(token_type, token)
274     self.indentation = self.indentation[:-2]
275     self.__write_close_tag(self.VAR_DEC_TAG)
276
277 def compile_statements(self) -> None:
278     """Compiles a sequence of statements, not including the enclosing
279     "{}".
280     """
281     self.__write_open_tag(self.STATEMENTS_TAG)
282     self.indentation += " "
283     self.output_stream.write("\n")
284     token, token_type = self.__get_current_token()
285     while token != "}":
286         if token == self.IF:
287             self.compile_if()
288             token, token_type = self.__get_current_token()
289         else:
290             if token == self.DO:
291                 self.compile_do()
292             elif token == self.LET:
293                 self.compile_let()
294             elif token == self.WHILE:
295                 self.compile_while()
296             elif token == self.RETURN:
297                 self.compile_return()
298             token, token_type = self.__get_current_token_and_advance()
299     self.indentation = self.indentation[:-2]
300     self.__write_close_tag(self.STATEMENTS_TAG)
301
302 def __subroutine_call_format(self) -> None:
303     """
304     this method compile the subroutine call format
305     """
306     # . -> ?
307     token, token_type = self.__get_current_token()
308     if token == ".":
309         # symbol
310         self.__write_open_and_close_tag(token_type, token)
311         # identifier
312         self.__write_next_advanced_token()
313         # ( -> ?
314         token, token_type = self.__get_current_token_and_advance()
315     # (
316     self.__write_open_and_close_tag(token_type, token)
317     # expression list
318     self.__get_current_token_and_advance()
319     self.compile_expression_list()
320     # )
321     self.__writes_current_token()
322
323 def compile_do(self) -> None:
324     """Compiles a do statement."""
325     self.__write_open_tag(self.DO_STATEMENT_TAG)
326     self.indentation += " "
327     self.output_stream.write("\n")
328     # keyword
329     self.__writes_current_token()
330     # identifier
331     self.__write_next_advanced_token()

```



```

332     # . -> ?
333     self.__get_current_token_and_advance()
334     # subroutine call
335     self.__subroutine_call_format()
336     # ;
337     self.__write_next_advanced_token()
338     self.indentation = self.indentation[:-2]
339     self.__write_close_tag(self.DO_STATEMENT_TAG)
340
341     def compile_let(self) -> None:
342         """Compiles a let statement."""
343         self.__write_open_tag(self.LET_STATEMENT_TAG)
344         self.indentation += "  "
345         self.output_stream.write("\n")
346         # keyword
347         self.__writes_current_token()
348         # identifier
349         token, token_type = self.__get_current_token_and_advance()
350         self.__write_open_and_close_tag(self.IDENTIFIER_TAG, token)
351         # [ -> ?
352         token, token_type = self.__get_current_token_and_advance()
353         if token == "[":
354             self.__write_open_and_close_tag(token_type, token)
355             # expression
356             self.__get_current_token_and_advance()
357             self.compile_expression()
358             # ]
359             self.__writes_current_token()
360             self.__get_current_token_and_advance()
361             # symbol
362             self.__writes_current_token()
363             # expression
364             self.__get_current_token_and_advance()
365             self.compile_expression()
366             # ;
367             self.__writes_current_token()
368             self.indentation = self.indentation[:-2]
369             self.__write_close_tag(self.LET_STATEMENT_TAG)
370
371     def compile_while(self) -> None:
372         """Compiles a while statement."""
373         self.__write_open_tag(self.WHILE_STATEMENT_TAG)
374         self.indentation += "  "
375         self.output_stream.write("\n")
376         # keyword
377         self.__writes_current_token()
378         # (
379         self.__write_next_advanced_token()
380         # expression
381         self.__get_current_token_and_advance()
382         self.compile_expression()
383         # )
384         self.__writes_current_token()
385         # {
386         # statements
387         self.__write_next_advanced_token()
388         self.compile_statements()
389         # }
390         self.__writes_current_token()
391         self.indentation = self.indentation[:-2]
392         self.__write_close_tag(self.WHILE_STATEMENT_TAG)
393
394     def compile_return(self) -> None:
395         """Compiles a return statement."""
396         self.__write_open_tag(self.RETURN_STATEMENT_TAG)
397         self.indentation += "  "
398         self.output_stream.write("\n")
399         # keyword

```

```

400     self.__writes_current_token()
401     # expression -> ?
402     token, token_type = self.__get_current_token_and_advance()
403     if token != ";":
404         # expression
405         self.compile_expression()
406         self.__writes_current_token()
407     else:
408         self.__write_open_and_close_tag(token_type, token)
409     self.indentation = self.indentation[:-2]
410     self.__write_close_tag(self.RETURN_STATEMENT_TAG)
411
412 def compile_if(self) -> None:
413     """Compiles a if statement, possibly with a trailing else clause."""
414     self.__write_open_tag(self.IF_STATEMENT_TAG)
415     self.indentation += " "
416     self.output_stream.write("\n")
417     # keyword
418     self.__writes_current_token()
419     # (
420     self.__write_next_advanced_token()
421     # expression
422     self.__get_current_token_and_advance()
423     self.compile_expression()
424     # )
425     self.__writes_current_token()
426     # {
427     self.__write_next_advanced_token()
428     # statements
429     self.__get_current_token_and_advance()
430     self.compile_statements()
431     # }
432     self.__writes_current_token()
433     # else -> ?
434     token, token_type = self.__get_current_token_and_advance()
435     if token == self.ELSE:
436         self.__write_open_and_close_tag(token_type, token)
437         # {
438         self.__write_next_advanced_token()
439         # statements
440         self.__get_current_token_and_advance()
441         self.compile_statements()
442         # }
443         self.__writes_current_token()
444         self.__get_current_token_and_advance()
445     self.indentation = self.indentation[:-2]
446     self.__write_close_tag(self.IF_STATEMENT_TAG)
447
448 def compile_expression(self) -> None:
449     """Compiles an expression."""
450     self.__write_open_tag(self.EXPRESSION_TAG)
451     self.indentation += " "
452     self.output_stream.write("\n")
453     # term
454     self.compile_term()
455     # term -> *
456     token, token_type = self.__get_current_token()
457     while token != ")":
458         if token not in self.op_terms:
459             break
460         # op
461         self.__write_open_and_close_tag(token_type, token)
462         token, token_type = self.__get_current_token_and_advance()
463         # term
464         self.compile_term()
465         token, token_type = self.__get_current_token()
466     self.indentation = self.indentation[:-2]
467     self.__write_close_tag(self.EXPRESSION_TAG)

```

```

468
469 def compile_term(self) -> None:
470     """Compiles a term.
471     This routine is faced with a slight difficulty when
472     trying to decide between some of the alternative parsing rules.
473     Specifically, if the current token is an identifier, the routing must
474     distinguish between a variable, an array entry, and a subroutine call.
475     A single look-ahead token, which may be one of "[", "(", or "." suffices
476     to distinguish between the three possibilities. Any other token is not
477     part of this term and should not be advanced over.
478     """
479     self.__write_open_tag(self.TERM_TAG)
480     self.indentation += " "
481     self.output_stream.write("\n")
482     # identifier / symbol
483     token, token_type = self.__get_current_token()
484     self.__write_open_and_close_tag(token_type, token)
485     # unary term -> ?
486     if token in self.unary_op_terms:
487         self.__get_current_token_and_advance()
488         # term
489         self.compile_term()
490     # expression - > ?
491     elif token == "(":
492         # expression
493         self.__get_current_token_and_advance()
494         self.compile_expression()
495         # )
496         self.__writes_current_token()
497         self.__get_current_token_and_advance()
498     else:
499         token, token_type = self.__get_current_token_and_advance()
500         # [ -> ?
501         if token == "[":
502             self.__write_open_and_close_tag(token_type, token)
503             # expression
504             self.__get_current_token_and_advance()
505             self.compile_expression()
506             # ]
507             self.__writes_current_token()
508             self.__get_current_token_and_advance()
509         # subroutine call -> ?
510         elif token in {".", "("}:
511             self.__subroutine_call_format()
512             self.__get_current_token_and_advance()
513     self.indentation = self.indentation[:-2]
514     self.__write_close_tag(self.TERM_TAG)
515
516 def compile_expression_list(self) -> None:
517     """Compiles a (possibly empty) comma-separated list of expressions."""
518     self.__write_open_tag(self.EXPRESSION_LIST_TAG)
519     self.indentation += " "
520     self.output_stream.write("\n")
521     # expression -> ?
522     token, token_type = self.__get_current_token()
523     while token != ",":
524         # expression
525         self.compile_expression()
526         token, token_type = self.__get_current_token()
527         if token == ",":
528             self.__write_open_and_close_tag(token_type, token)
529             token, token_type = self.__get_current_token_and_advance()
530     self.indentation = self.indentation[:-2]
531     self.__write_close_tag(self.EXPRESSION_LIST_TAG)

```

4 JackAnalyzer

```
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 'JackAnalyzer <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 "JackAnalyzer 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 "JackAnalyzer.py".
16 #           If your main is contained elsewhere, you will need to change this.
17
18 python3 JackAnalyzer.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 Common Attribution-NonCommercial-ShareAlike 3.0
24 # Unported License: https://creativecommons.org/licenses/by-nc-sa/3.0/
```

5 JackAnalyzer.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
14
15 def analyze_file(
16     input_file: typing.TextIO, output_file: typing.TextIO) -> None:
17     """Analyzes a single file.
18
19     Args:
20         input_file (typing.TextIO): the file to analyze.
21         output_file (typing.TextIO): writes all output to this file.
22     """
23     tokenizer = JackTokenizer(input_file)
24     engine = CompilationEngine(tokenizer, output_file)
25     engine.compile_class()
26
27     output_file.close()
28
29
30 if "__main__" == __name__:
31     # Parses the input path and calls analyze_file on each input file.
32     # This opens both the input and the output files!
33     # Both are closed automatically when the code finishes running.
34     # If the output file does not exist, it is created automatically in the
35     # correct path, using the correct filename.
36     if not len(sys.argv) == 2:
37         sys.exit("Invalid usage, please use: JackAnalyzer <input path>")
38     argument_path = os.path.abspath(sys.argv[1])
39     if os.path.isdir(argument_path):
40         files_to_assemble = [
41             os.path.join(argument_path, filename)
42             for filename in os.listdir(argument_path)]
43     else:
44         files_to_assemble = [argument_path]
45     for input_path in files_to_assemble:
46         filename, extension = os.path.splitext(input_path)
47         if extension.lower() != ".jack":
48             continue
49         output_path = filename + ".xml"
50         with open(input_path, 'r') as input_file, \
51             open(output_path, 'w') as output_file:
52             analyze_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 'JackAnalyzer <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 JackAnalyzer
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/
```