Contents

1 Basic Test Results

```
********** FOLDER STRUCTURE TEST START **********
Extracting submission...
     Extracted zip successfully

Finding usernames...
     Submission logins are: e342791191,neriya.zarka
     Is this OK?

Checking for non-ASCII characters with the command 'grep -IHPnsr [^\x00-\x7F] <dir>' ...
     No invalid characters found.

**********  FOLDER STRUCTURE TEST END  **********


********** PROJECT TEST START **********
Running 'make'...
     'make' ran successfully.

Finding JackCompiler...
     Found in the correct path.

Testing translation of Main of Seven (running on a single file)...
     Testing your JackCompiler with command: './JackCompiler tst/Seven/Main.jack'...
     Testing your output with command: './VMEmulator.sh tst/Seven/Seven.tst'...
          Test passed.

Testing ComplexArrays, where ComplexArrays is a directory...
     Testing your JackCompiler with command: './JackCompiler ComplexArrays/'...
     Testing your output with command: './VMEmulator.sh ComplexArrays/ComplexArrays.tst'...
          Test passed.

Testing Seven, where Seven is a directory...
     Testing your JackCompiler with command: './JackCompiler Seven/'...
     Testing your output with command: './VMEmulator.sh Seven/Seven.tst'...
          Test passed.

**********  PROJECT TEST END  **********


************************************************
**********   PRESUBMISSION TESTS PASSED   **********
************************************************

Note: the tests you see above are all the presubmission tests
for this project. The tests might not check all the different
parts of the project or all corner cases, so write your own
tests and use them!
```

2 AUTHORS

```
e342791191,neriya.zarka
Partner 1: Neriya Zarka, neriya.zarka@mail.huji.ac.il, 323014761
Partner 2: Elsa Sebagh, elsa.sebagh@mail.huji.ac.il, 342791191
Remarks:
```

3 CompilationEngine.py

```
"""
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/).
"""
import typing
from SymbolTable import SymbolTable

from VMWriter import VMWriter


class CompilationEngine:
    """Gets input from a JackTokenizer and emits its parsed structure into an
    output stream.
    """

    def __init__(self, input_stream: "JackTokenizer", output_stream) -> None:
        """
        Creates a new compilation engine with the given input and output. The
        next routine called must be compileClass()
        :param input_stream: The input stream.
        :param output_stream: The output stream.
        """
        # Your code goes here!
        # Note that you can write to output_stream like so:
        self.lines = input_stream.get_output()
        index_list = [i for i in range(len(self.lines))]
        self.current_line = 0
        self.class_name = ''
        self.VM_writer = VMWriter(output_stream)
        self.symbol_table = SymbolTable()
        self.label_index = 0
        if self.lines[self.current_line] == "<tokens>":
            self.current_line += 1
            self.compile_class()


    def get_label(self, command: str):
        return_index = self.label_index
        self.label_index += 1
        return self.class_name + '.' + command + '.' + str(return_index)

    def current_token(self) -> str:
        if self.current_line >= len(self.lines):
            return None
        if self.current_token_type() == "stringConstant":
            return " ".join(self.lines[self.current_line].split()[1:-1])
        if len(self.lines[self.current_line].split()) == 1:
            return None

        return self.lines[self.current_line].split()[1]

    def current_token_type(self) -> str:
        if self.current_line >= len(self.lines): return None
        return self.lines[self.current_line].split()[0][1:-1]

    # Elsa - done
```

```
60      def compile_class(self) -> None:
61          """Compiles a complete class."""
62          self.current_line += 1
63          self.class_name = self.current_token()
64          self.current_line += 1
65          self.current_line += 1
66
67          # Compile all var
68          self.compile_class_var_dec()
69          # Compile all subroutines
70          self.compile_subroutine()
71          self.current_line += 1  # get final { and end
72
73      # Elsa - done
74      def compile_class_var_dec(self) -> None:
75          """Compiles a static declaration or a field declaration."""
76          # Goes over all the variables
77          while self.current_token() == "static" or self.current_token() == "field":
78              kind = self.current_token()
79              self.current_line += 1
80              type = self.current_token()
81              self.current_line += 1
82              name = self.current_token()
83              self.symbol_table.define(name, type, kind.upper())
84              self.current_line += 1
85              while self.current_token() == ",":
86                  self.current_line += 1
87                  name = self.current_token()
88                  self.symbol_table.define(name, type, kind.upper())
89                  self.current_line += 1
90              self.current_line += 1
91
92      # Elsa - done
93      def compile_subroutine(self) -> None:
94          """
95          Compiles a complete method, function, or constructor.
96          You can assume that classes with constructors have at least one field,
97          you will understand why this is necessary in project 11.
98          """
99          # Goes over all subroutine
100         while self.current_token() in ["constructor", "function", "method"]:
101             self.symbol_table.start_subroutine()
102             type = self.current_token()
103             self.current_line += 1  # method/function/constructor
104             self.current_line += 1  # return type TODO check if there is something to do with it
105             name = self.current_token()
106             self.current_line += 1  # now current token is opening parenthesis
107             self.current_line += 1  # first argument or closing parenthesis
108             self.compile_parameter_list(type)
109             self.current_line += 1  # {
110             self.compile_var_dec()
111             self.VM_writer.write_function(self.class_name + "." + name, self.symbol_table.var_count("VAR"))
112             if type == "constructor":
113                 self.VM_writer.write_push("constant", self.symbol_table.var_count("FIELD"))
114                 self.VM_writer.write_call("Memory.alloc", 1)
115                 self.VM_writer.write_pop("pointer", 0)
116             elif type == "method":
117                 self.VM_writer.write_push("arg", 0)
118                 self.VM_writer.write_pop("pointer", 0)
119             self.compile_statements()
120             self.current_line += 1  # }
121
122     # Elsa - done
123     def compile_parameter_list(self, func_type) -> None:
124         """Compiles a (possibly empty) parameter list, not including the
125         enclosing "()".
126         """
127         if func_type == "method": self.symbol_table.define("this", self.class_name, "ARG")
```

```
128             while self.current_token() != ")":
129                 # num_parameters += 1  #Neriya: what is this line for?
130                 if self.current_token() == ",":
131                     self.current_line += 1
132                 else:
133                     type = self.current_token()
134                     self.current_line += 1
135                     name = self.current_token()
136                     self.current_line += 1
137                     self.symbol_table.define(name, type, "ARG")
138             self.current_line += 1  # closing parenthesis
139 
140         # Elsa - done
141         def compile_var_dec(self) -> None:
142             """Compiles a var declaration."""
143             while self.current_token() == "var":
144                 self.current_line += 1
145                 type = self.current_token()
146                 self.current_line += 1
147                 name = self.current_token()
148                 self.symbol_table.define(name, type, "VAR")
149                 self.current_line += 1
150                 while self.current_token() == ",":
151                     self.current_line += 1
152                     name = self.current_token()
153                     self.symbol_table.define(name, type, "VAR")
154                     self.current_line += 1
155                 self.current_line += 1
156 
157         # Neriya - done (no change needed)
158         def compile_statements(self) -> None:
159             """Compiles a sequence of statements, not including the enclosing
160             "{}".
161             """
162             statement_tokens = {"let": self.compile_let,
163                                 "if": self.compile_if,
164                                 "while": self.compile_while,
165                                 "do": self.compile_do,
166                                 "return": self.compile_return}
167 
168             while self.current_token() in statement_tokens.keys():
169                 statement_tokens[self.current_token()]()
170                 # self.current_line +=1
171 
172         # Neriya- done
173         def compile_do(self) -> None:
174             """Compiles a do statement."""
175             self.current_line += 1
176             self.compile_subroutine_call()
177             self.VM_writer.write_pop("temp", 0)
178             self.current_line += 1
179 
180         # Elsa - done
181         def compile_let(self) -> None:
182             """Compiles a let statement."""
183             is_array = False
184             self.current_line += 1  # let
185             var_name = self.current_token()  # variable name
186             self.current_line += 1  # = or [ if it's and array
187 
188             if self.current_token() == "[":
189                 self.current_line += 1
190                 self.compile_expression()
191                 self.VM_writer.write_push(self.symbol_table.kind_of(var_name), (self.symbol_table.index_of(var_name)))
192                 self.VM_writer.write_arithmetic("ADD")
193                 self.current_line += 1
194                 is_array = True
195             self.current_line += 1  # begining of the expression
```

```
196             self.compile_expression() # Compile what's after the =
197             var_type = self.symbol_table.kind_of(var_name)
198             var_index = self.symbol_table.index_of(var_name)
199             if is_array:
200                 self.VM_writer.write_pop("TEMP", 0)
201                 self.VM_writer.write_pop("POINTER", 1)
202                 self.VM_writer.write_push("TEMP", 0)
203                 self.VM_writer.write_pop("THAT", 0)
204             else:
205                 self.VM_writer.write_pop(var_type, var_index)
206             self.current_line += 1  # I added this line
207 
208         # Neriya - done
209         def compile_while(self) -> None:
210             """Compiles a while statement."""
211             self.current_line += 1
212             start_while_label = self.get_label("WHILE_START")
213             self.VM_writer.write_label(start_while_label)
214             self.current_line += 1
215             self.compile_expression()
216             self.current_line += 1
217 
218             self.VM_writer.write_arithmetic("NOT")
219 
220             end_while_label = self.get_label("WHILE_END")
221             self.VM_writer.write_if(end_while_label)
222             self.current_line += 1
223             self.compile_statements()
224             self.current_line += 1
225 
226             self.VM_writer.write_goto(start_while_label)
227 
228             self.VM_writer.write_label(end_while_label)
229 
230         # Neriya - done
231         def compile_return(self) -> None:
232             """Compiles a return statement."""
233             self.current_line += 1
234             if self.current_token() != ";":
235                 self.compile_expression()
236             else:
237                 self.VM_writer.write_push("constant", 0)
238             self.current_line += 1
239             self.VM_writer.write_return()
240 
241         # Neriya - done
242         def compile_if(self) -> None:
243             """Compiles a if statement, possibly with a trailing else clause."""
244             self.current_line += 1
245             self.current_line += 1
246 
247             self.compile_expression()
248             self.current_line += 1
249 
250             self.VM_writer.write_arithmetic("NOT")
251             else_label = self.get_label("ELSE")
252             self.VM_writer.write_if(else_label)
253             self.current_line += 1
254 
255             self.compile_statements()
256             self.current_line += 1
257 
258             END_IF_label = self.get_label("END_IF")
259             self.VM_writer.write_goto(END_IF_label)
260 
261             self.VM_writer.write_label(else_label)
262 
263             if self.current_token() == "else":
```

```
264             self.current_line += 1
265
266             self.current_line += 1
267
268             self.compile_statements()
269             self.current_line += 1
270
271         self.VM_writer.write_label(END_IF_label)
272
273         # self.write_end("ifStatement")
274
275     # Neriya - done
276     def compile_expression(self) -> None:
277         """Compiles an expression.
278         do at the end
279         """
280         self.compile_term()
281         vm_operations = {
282             "*": "Math.multiply",
283             "/": "Math.divide",
284         }
285         operators = {'+': "ADD", "-": "SUB", "&": "AND", "|": "OR", "<": "LT", ">": "GT", "=": "EQ", "&lt;": "LT",
286                      "&gt;": "GT", "&amp;": "AND"}
287
288         while self.current_token() in vm_operations or self.current_token() in operators:
289             operator = self.current_token()
290             self.current_line += 1
291             self.compile_term()
292             if operator in vm_operations:
293                 self.VM_writer.write_call(vm_operations[operator], 2)
294             else:
295                 self.VM_writer.write_arithmetic(operators[operator])
296
297     # Neriya - almost done
298     def compile_term(self) -> None:
299         """Compiles a term.
300         This routine is faced with a slight difficulty when
301         trying to decide between some of the alternative parsing rules.
302         Specifically, if the current token is an identifier, the routing must
303         distinguish between a variable, an array entry, and a subroutine call.
304         A single look-ahead token, which may be one of "[", "(", or "." suffices
305         to distinguish between the three possibilities. Any other token is not
306         part of this term and should not be advanced over.
307         """
308         if self.current_token_type() is None:
309             return
310         elif self.current_token_type() == "integerConstant":
311             self.VM_writer.write_push("constant", self.current_token())
312             self.current_line += 1
313
314         elif self.current_token_type() == "stringConstant":
315             self.VM_writer.write_push("constant", len(self.current_token()) + 1)
316             self.VM_writer.write_call("String.new", 1)
317             for char in self.current_token():
318                 self.VM_writer.write_push("constant", ord(char))
319                 self.VM_writer.write_call("String.appendChar", 2)
320             self.VM_writer.write_push("constant", 32)
321             self.VM_writer.write_call("String.appendChar", 2)  # Add white space
322             self.current_line += 1
323
324         elif self.current_token() in ["true", "false", "null", "this"]:
325             if self.current_token() == "true":
326                 self.VM_writer.write_push("constant", 1)
327                 self.VM_writer.write_arithmetic("NEG")
328             elif self.current_token() == "false" or self.current_token() == "null":
329                 self.VM_writer.write_push("constant", 0)
330             elif self.current_token() == "this":
331                 self.VM_writer.write_push("pointer", 0)
```

```
            self.current_line += 1

        elif self.current_token_type() == "identifier":
            name = self.current_token()
            self.current_line += 1
            if self.current_token() == "[":
                is_array = True
                self.current_line += 1
                self.compile_expression()
                self.VM_writer.write_push(self.symbol_table.kind_of(name), self.symbol_table.index_of(name))
                self.VM_writer.write_arithmetic("ADD")
                self.VM_writer.write_pop("pointer", 1)
                self.VM_writer.write_push("that", 0)
                self.current_line += 1


            elif self.current_token() == "(" or self.current_token() == ".":  # subroutine call
                self.current_line -= 1 # Go back to function name
                self.compile_subroutine_call()  # todo: support compile_subroutine_call(name)
                    #  i thik it will fix the problem in compile_subroutine_call

            else:
                # regular variable
                self.VM_writer.write_push(self.symbol_table.kind_of(name), self.symbol_table.index_of(name))

        elif self.current_token() == "(":
            self.current_line += 1
            self.compile_expression()
            self.current_line += 1

        elif self.current_token() in ["-", "~"]:
            unary_op = self.current_token()
            self.current_line += 1
            self.compile_term()
            if unary_op == "-":
                self.VM_writer.write_arithmetic("NEG")
            elif unary_op == "~":
                self.VM_writer.write_arithmetic("NOT")



    # Elsa - done
    def compile_subroutine_call(self) -> None:
        """Compiles a subroutine call."""
        first_name = self.current_token()
        num_arguments = 0
        self.current_line += 1
        if self.current_token() == ".":  # Let's say call looks like this: foo.mult(1, 2, 3)
            self.current_line += 1
            last_name = self.current_token()
            if self.symbol_table.index_of(first_name) != None:
                self.VM_writer.write_push(self.symbol_table.kind_of(first_name),
                                          self.symbol_table.index_of(first_name))  # push foo
                fullName = self.symbol_table.type_of(first_name) + '.' + last_name
                num_arguments += 1
            else:
                fullName = first_name + '.' + last_name
        elif self.current_token() == "(":
            self.VM_writer.write_push('pointer', 0)
            num_arguments += 1
            fullName = self.class_name + '.' + first_name
        self.current_line += 2  # now parenthesis

        num_arguments += self.compile_expression_list()
        self.VM_writer.write_call(fullName, num_arguments)

    # Elsa - done
    def compile_expression_list(self) -> int:
```

```
400             """Compiles a (possibly empty) comma-separated list of expressions."""
401             num_arguments = 0
402             if self.current_token() == ",": self.current_line -=1
403             # i addes self.current_token() != ";"
404             while self.current_token() != ")" and self.current_token() != "}" and self.current_token() \
405                     and self.current_token() != ";":
406                 self.compile_expression()
407                 self.current_line += 1
408                 num_arguments += 1
409             self.current_line += 1
410             if self.current_token() != ";": self.current_line -= 1
411             return num_arguments
412
413
414     if __name__ == "__main__":
415         with open("Square/MainT.xml", 'r') as input_file, \
416                 open("Square/MY_Main.xml", 'w') as output_file:
417             CE1 = CompilationEngine(input_file, output_file)
418         with open("Square/SquareT.xml", 'r') as input_file, \
419                 open("Square/MY_Square.xml", 'w') as output_file:
420             CE2 = CompilationEngine(input_file, output_file)
421         with open("Square/SquareGameT.xml", 'r') as input_file, \
422                 open("Square/MY_SquareGame.xml", 'w') as output_file:
423             CE3 = CompilationEngine(input_file, output_file)
```

4 JackCompiler

```
#!/bin/sh
# This file only works on Unix-like operating systems, so it won't work on Windows.

## Why do we need this file?
# The purpose of this file is to run your project.
# We want our users to have a simple API to run the project.
# So, we need a "wrapper" that will hide all  details to do so,
# enabling users to simply type 'JackCompiler <path>' in order to use it.

## What are '#!/bin/sh' and '$*'?
# '$*' is a variable that holds all the arguments this file has received. So, if you
# run "JackCompiler trout mask replica", $* will hold "trout mask replica".

## What should I change in this file to make it work with my project?
# IMPORTANT: This file assumes that the main is contained in "JackCompiler.py".
#            If your main is contained elsewhere, you will need to change this.

python3 JackCompiler.py $*

# 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
# in 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/
```

5 JackCompiler.py

```
"""
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/).
"""
import os
import sys
import typing
from CompilationEngine import CompilationEngine
from JackTokenizer import JackTokenizer


def compile_file(
        input_file: typing.TextIO, output_file: typing.TextIO) -> None:
    """Compiles a single file.

    Args:
        input_file (typing.TextIO): the file to compile.
        output_file (typing.TextIO): writes all output to this file.
    """
    # Your code goes here!
    # This function should be relatively similar to "analyze_file" in
    # JackAnalyzer.py from the previous project.
    tokenizer = JackTokenizer(input_file)
    engine = CompilationEngine(tokenizer, output_file)



if "__main__" == __name__:
    # Parses the input path and calls compile_file on each input file.
    # This opens both the input and the output files!
    # Both are closed automatically when the code finishes running.
    # If the output file does not exist, it is created automatically in the
    # correct path, using the correct filename.
    if not len(sys.argv) == 2:
        sys.exit("Invalid usage, please use: JackCompiler <input path>")
    argument_path = os.path.abspath(sys.argv[1])
    if os.path.isdir(argument_path):
        files_to_assemble = [
            os.path.join(argument_path, filename)
            for filename in os.listdir(argument_path)]
    else:
        files_to_assemble = [argument_path]
    for input_path in files_to_assemble:
        filename, extension = os.path.splitext(input_path)
        if extension.lower() != ".jack":
            continue
        output_path = filename + ".vm"
        with open(input_path, 'r') as input_file, \
                open(output_path, 'w') as output_file:
            compile_file(input_file, output_file)
```

6 JackTokenizer.py

```
"""
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/).
"""
import typing

statements = ["let", "while", "if", "do", "return"]
symbols = ['{', '}', '(', ')', '[', ']', '.', ',', ';', '+',
           '-', '*', '/', '&', '|', '<', '>', '=', '~', '^', '#']
keywords = ['class', 'constructor', 'function', 'method', 'field',
            'static', 'var', 'int', 'char', 'boolean', 'void', 'true',
            'false', 'null', 'this', 'let', 'do', 'if', 'else',
            'while', 'return']
tags = {"KEYWORD": "<keyword> <token> </keyword>\n",
        "IDENTIFIER": "<identifier> <token> </identifier>\n",
        "SYMBOL": "<symbol> <token> </symbol>\n",
        "STRING_CONST": "<stringConstant> <token> </stringConstant>\n",
        "INT_CONST": "<integerConstant> <token> </integerConstant>\n"}

def clean_line(command: str) -> str:
    return command.split("//")[0].rstrip() # Can't remove whitespaces at the begining of the string cause it messes with the

class JackTokenizer:
    """Removes all comments from the input stream and breaks it
    into Jack language tokens, as specified by the Jack grammar.

    # Jack Language Grammar

    A Jack file is a stream of characters. If the file represents a
    valid program, it can be tokenized into a stream of valid tokens. The
    tokens may be separated by an arbitrary number of whitespace characters,
    and comments, which are ignored. There are three possible comment formats:
    /* comment until closing */ , /** API comment until closing */ , and

    - xxx: regular typeface is used for names of language constructs
    - (): parentheses are used for grouping of language constructs.
    - x | y: indicates that either x or y can appear.
    - x?: indicates that x appears 0 or 1 times.
    - x*: indicates that x appears 0 or more times.

    ## Lexical Elements

    The Jack language includes five types of terminal elements (tokens).

    - keyword: 'class' | 'constructor' | 'function' | 'method' | 'field' |
               'static' | 'var' | 'int' | 'char' | 'boolean' | 'void' | 'true' |
               'false' | 'null' | 'this' | 'let' | 'do' | 'if' | 'else' |
               'while' | 'return'
    - symbol: '{' | '}' | '(' | ')' | '[' | ']' | '.' | ',' | ';' | '+' |
              '-' | '*' | '/' | '&' | '|' | '<' | '>' | '=' | '~' | '^' | '#'
    - integerConstant: A decimal number in the range 0-32767.
    - StringConstant: '"' A sequence of Unicode characters not including
                      double quote or newline '"'
    - identifier: A sequence of letters, digits, and underscore ('_') not
                  starting with a digit. You can assume keywords cannot be
                  identifiers, so 'self' cannot be an identifier, etc'.
```

```
60
61      ## Program Structure
62
63      A Jack program is a collection of classes, each appearing in a separate
64      file. A compilation unit is a single class. A class is a sequence of tokens
65      structured according to the following context free syntax:
66
67      - class: 'class' className '{' classVarDec* subroutineDec* '}'
68      - classVarDec: ('static' | 'field') type varName (',' varName)* ';'
69      - type: 'int' | 'char' | 'boolean' | className
70      - subroutineDec: ('constructor' | 'function' | 'method') ('void' | type)
71      - subroutineName '(' parameterList ')' subroutineBody
72      - parameterList: ((type varName) (',' type varName)*)?
73      - subroutineBody: '{' varDec* statements '}'
74      - varDec: 'var' type varName (',' varName)* ';'
75      - className: identifier
76      - subroutineName: identifier
77      - varName: identifier
78
79      ## Statements
80
81      - statements: statement*
82      - statement: letStatement | ifStatement | whileStatement | doStatement |
83                   returnStatement
84      - letStatement: 'let' varName ('[' expression ']')? '=' expression ';'
85      - ifStatement: 'if' '(' expression ')' '{' statements '}' ('else' '{'
86                     statements '}')?
87      - whileStatement: 'while' '(' 'expression' ')' '{' statements '}'
88      - doStatement: 'do' subroutineCall ';'
89      - returnStatement: 'return' expression? ';'
90
91      ## Expressions
92
93      - expression: term (op term)*
94      - term: integerConstant | stringConstant | keywordConstant | varName |
95              varName '['expression']' | subroutineCall | '(' expression ')' |
96              unaryOp term
97      - subroutineCall: subroutineName '(' expressionList ')' | (className |
98                        varName) '.' subroutineName '(' expressionList ')'
99      - expressionList: (expression (',' expression)* )?
100     - op: '+' | '-' | '*' | '/' | '&' | '|' | '<' | '>' | '='
101     - unaryOp: '-' | '~' | '^' | '#'
102     - keywordConstant: 'true' | 'false' | 'null' | 'this'
103
104     Note that ^, # correspond to shiftleft and shiftright, respectively.
105     """
106
107
108     def __init__(self, input_stream: typing.TextIO) -> None:
109         """Opens the input stream and gets ready to tokenize it.
110
111         Args:
112             input_stream (typing.TextIO): input stream.
113         """
114         # Your code goes here!
115         # A good place to start is to read all the lines of the input:
116         self.input_lines = input_stream.read().splitlines()
117         self.current_token = ""
118         self.current_line = 0
119         self.current_index = -1 # End position inside the line
120         self.output = []
121         self.tokenize()
122
123     def get_output(self):
124         return self.output
125
126     def tokenize(self) -> None:
127         self.output.append("<tokens>")
```

```
        while(self.has_more_tokens()):
            self.advance()
            if self.current_token:
                token_type = self.token_type()
                if(token_type == "KEYWORD"):
                    token = self.current_token
                elif(token_type == "IDENTIFIER"):
                    token = self.identifier()
                elif(token_type == "STRING_CONST"):
                    token = self.string_val()
                elif(token_type == "INT_CONST"):
                    token = self.int_val()
                else:
                    token = self.symbol()
                self.output.append(tags[token_type].replace("<token>", str(token)))
        self.output.append("</tokens>\n")



    def has_more_tokens(self) -> bool:
        """Do we have more tokens in the input?

        Returns:
            bool: True if there are more tokens, False otherwise.
        """
        # Your code goes here!
        return self.current_line <= len(self.input_lines) - 1 and self.current_index <= len(self.input_lines[self.current_li


    def advance(self) -> None:
        """Gets the next token from the input and makes it the current token.
        This method should be called if has_more_tokens() is true.
        Initially there is no current token.
        """
        if(self.current_line >= len(self.input_lines)):
            self.current_token = ""
            return
        # Takes care of multi line comments
        if self.input_lines[self.current_line][self.current_index + 1:].startswith("/*"):
            self.current_index += 2
            # if self.input_lines[self.current_line][self.current_index] == "*": self.current_index +=1
            line_index = self.current_line
            letter_index = self.current_index
            while line_index < len(self.input_lines):
                if "*/" in self.input_lines[line_index]: # If we found the end of the comment
                    letter_index = self.input_lines[line_index].find("*/") + 1
                    if letter_index >= len(self.input_lines[line_index]) - 1: # If we got to the end of the line
                        letter_index = -1
                        line_index += 1
                    break
                else: line_index += 1
            if line_index >= len(self.input_lines): return
            else:
                self.current_line = line_index
                self.current_index = letter_index
            self.advance()
            return

        line = clean_line(self.input_lines[self.current_line][self.current_index + 1:])
        if not line: # Finished the current line, go to the next one
            self.current_line += 1
            self.current_index = -1
            self.advance()
            return
        if line[0] == " " or line[0] == "\t": # Remove whitespaces and tabs
            self.current_index += 1
            self.advance()
            return
```

```
        if line[0] in symbols: # Next token is a symbol
            self.current_index += 1
            self.current_token = self.input_lines[self.current_line][self.current_index]
        else: # Next token is a keyword
            index = 0
            # If it's a string constant
            if line[index] == "\"":
                index += 1
                while line[index] != "\"":
                    index += 1
                index +=1
            # If it's a string constant
            elif line[index] == "'":
                index += 1
                while line[index] != "'":
                    index += 1
                index += 1
            else:
                while line[index] not in symbols and line[index] != " ":
                    index += 1
            self.current_token = line[0:index].replace("\n", "")
            if line[index] == " " or line[index] == "\t":
                index += 1
            self.current_index += index



    def token_type(self) -> str:
        """
        Returns:
            str: the type of the current token, can be
            "KEYWORD", "SYMBOL", "IDENTIFIER", "INT_CONST", "STRING_CONST"
        """
        # Your code goes here!
        if self.current_token in symbols:
            return "SYMBOL"
        elif self.current_token in keywords:
            return "KEYWORD"
        elif self.current_token.isdigit():
            return "INT_CONST"
        elif self.current_token[0] == "\"" or self.current_token[0] == "'":
            return "STRING_CONST"
        else:
            return "IDENTIFIER"

    def keyword(self) -> str:
        """
        Returns:
            str: the keyword which is the current token.
            Should be called only when token_type() is "KEYWORD".
            Can return "CLASS", "METHOD", "FUNCTION", "CONSTRUCTOR", "INT",
            "BOOLEAN", "CHAR", "VOID", "VAR", "STATIC", "FIELD", "LET", "DO",
            "IF", "ELSE", "WHILE", "RETURN", "TRUE", "FALSE", "NULL", "THIS"
        """
        # Your code goes here!
        return '"' + self.current_token.upper() + '"'

    def symbol(self) -> str:
        """
        Returns:
            str: the character which is the current token.
            Should be called only when token_type() is "SYMBOL".
            Recall that symbol was defined in the grammar like so:
            symbol: '{' | '}' | '(' | ')' | '[' | ']' | '.' | ',' | ';' | '+' |
              '-' | '*' | '/' | '&' | '|' | '<' | '>' | '=' | '~' | '^' | '#'
        """
        # Your code goes here!
        if self.current_token == '>':
```

```
            return "&gt;"
        elif self.current_token == "<":
            return "&lt;"
        elif self.current_token == "&":
            return "&amp;"
        else:
            return self.current_token

    def identifier(self) -> str:
        """
        Returns:
            str: the identifier which is the current token.
            Should be called only when token_type() is "IDENTIFIER".
            Recall that identifiers were defined in the grammar like so:
            identifier: A sequence of letters, digits, and underscore ('_') not
                  starting with a digit. You can assume keywords cannot be
                  identifiers, so 'self' cannot be an identifier, etc'.
        """
        # Your code goes here!
        return self.current_token

    def int_val(self) -> int:
        """
        Returns:
            str: the integer value of the current token.
            Should be called only when token_type() is "INT_CONST".
            Recall that integerConstant was defined in the grammar like so:
            integerConstant: A decimal number in the range 0-32767.
        """
        # Your code goes here!
        return int(self.current_token)

    def string_val(self) -> str:
        """
        Returns:
            str: the string value of the current token, without the double
            quotes. Should be called only when token_type() is "STRING_CONST".
            Recall that StringConstant was defined in the grammar like so:
            StringConstant: '"' A sequence of Unicode characters not including
                      double quote or newline '"'
        """
        # Your code goes here!
        return self.current_token.replace("\"", "").replace("'", "")
```

7 Makefile

```
# Makefile for a script (e.g. Python)

## Why do we need this file?
# 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,
# thus enabling our users to simply type 'JackCompiler <path>' in order to use it.

## What are makefiles?
# This is a sample makefile.
# The purpose of makefiles is to make sure that after running "make" your
# project is ready for execution.

## What should I change in this file to make it work with my project?
# Usually, scripting language (e.g. Python) based projects only need execution
# permissions for your run file executable to run.
# Your project may be more complicated and require a different makefile.

## What is a makefile rule?
# A makefile rule is a list of prerequisites (other rules that need to be run
# before this rule) and commands that are run one after the other.
# The "all" rule is what runs when you call "make".
# In this example, all it does is grant execution permissions for your
# executable, so your project will be able to run on the graders' computers.
# In this case, the "all" rule has no preqrequisites.

## How are rules defined?
# The following line is a rule declaration:
# all:
#     chmod a+x JackCompiler

# A general rule looks like this:
# rule_name: prerequisite1 prerequisite2 prerequisite3 prerequisite4 ...
#     command1
#     command2
#     command3
#     ...
# Where each preqrequisite is a rule name, and each command is a command-line
# command (for example chmod, javac, echo, etc').

# Beginning of the actual Makefile
all:
    chmod a+x *

# 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
# in 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 SymbolTable.py

```
"""
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/).
"""
import typing



class SymbolTable:
    """A symbol table that associates names with information needed for Jack
    compilation: type, kind and running index. The symbol table has two nested
    scopes (class/subroutine).
    """

    def __init__(self) -> None:
        """Creates a new empty symbol table."""
        # Your code goes here!
        self.class_table = {}
        self.subroutine_table = {}
        self.indexes = {
            "STATIC": 0, "FIELD": 0, "ARG": 0, "VAR": 0
        }

    def start_subroutine(self) -> None:
        """Starts a new subroutine scope (i.e., resets the subroutine's
        symbol table).
        """
        # Your code goes here!
        self.subroutine_table = {}
        self.indexes["ARG"]= 0
        self.indexes["VAR"]= 0


    def define(self, name: str, type: str, kind: str) -> None:
        """Defines a new identifier of a given name, type and kind and assigns
        it a running index. "STATIC" and "FIELD" identifiers have a class scope,
        while "ARG" and "VAR" identifiers have a subroutine scope.

        Args:
            name (str): the name of the new identifier.
            type (str): the type of the new identifier.
            kind (str): the kind of the new identifier, can be:
            "STATIC", "FIELD", "ARG", "VAR".
        """
        # Your code goes here!
        if kind == "STATIC" or kind == "FIELD":
            self.class_table[name] = {"index": self.indexes[kind], "type": type, "kind": kind}
        else:
            self.subroutine_table[name] = {"index": self.indexes[kind], "type": type, "kind": kind}
        self.indexes[kind] += 1

    def var_count(self, kind: str) -> int:
        """
        Args:
            kind (str): can be "STATIC", "FIELD", "ARG", "VAR".

```

```
60          Returns:
61              int: the number of variables of the given kind already defined in
62              the current scope.
63          """
64          # Your code goes here!
65          return self.indexes[kind]
66
67      def kind_of(self, name: str) -> str:
68          """
69          Args:
70              name (str): name of an identifier.
71
72          Returns:
73              str: the kind of the named identifier in the current scope, or None
74              if the identifier is unknown in the current scope.
75          """
76          # Your code goes here!
77          if name in self.subroutine_table.keys():
78              return self.subroutine_table[name]["kind"]
79          elif name in self.class_table.keys():
80              return self.class_table[name]["kind"]
81          else:
82              return None
83
84      def type_of(self, name: str) -> str:
85          """
86          Args:
87              name (str):  name of an identifier.
88
89          Returns:
90              str: the type of the named identifier in the current scope.
91          """
92          # Your code goes here!
93          if name in self.subroutine_table.keys():
94              return self.subroutine_table[name]["type"]
95          elif name in self.class_table.keys():
96              return self.class_table[name]["type"]
97          else:
98              return None
99
100     def index_of(self, name: str) -> int:
101         """
102         Args:
103             name (str):  name of an identifier.
104
105         Returns:
106             int: the index assigned to the named identifier.
107         """
108         # Your code goes here!
109         if name in self.subroutine_table.keys():
110             return self.subroutine_table[name]["index"]
111         elif name in self.class_table.keys():
112             return self.class_table[name]["index"]
113         else:
114             return None
```

9 VMWriter.py

```
"""
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/).
"""
import typing


class VMWriter:
    """
    Writes VM commands into a file. Encapsulates the VM command syntax.
    """

    def __init__(self, output_stream: typing.TextIO) -> None:
        """Creates a new file and prepares it for writing VM commands."""
        # Your code goes here!
        # Note that you can write to output_stream like so:
        # output_stream.write("Hello world! \n")
        self.output_stream = output_stream

    def write_push(self, segment: str, index: int) -> None:
        """Writes a VM push command.

        Args:
            segment (str): the segment to push to, can be "CONST", "ARG",
            "LOCAL", "STATIC", "THIS", "THAT", "POINTER", "TEMP"
            index (int): the index to push to.
        """
        # Your code goes here!
        if segment.lower() == "var": segment = "local"
        elif segment.lower() == "arg": segment = "argument"
        elif segment.lower() == "field": segment = "this"
        self.output_stream.write("push " + segment.lower()+ " " + str(index) + "\n")

    def write_pop(self, segment: str, index: int) -> None:
        """Writes a VM pop command.

        Args:
            segment (str): the segment to pop from, can be "CONST", "ARG",
            "LOCAL", "STATIC", "THIS", "THAT", "POINTER", "TEMP".
            index (int): the index to pop from.
        """
        # Your code goes here!
        if segment.lower() == "var": segment = "local"
        elif segment.lower() == "arg": segment = "argument"
        elif segment.lower() == "field": segment = "this"
        self.output_stream.write("pop " + segment.lower() + " " + str(index) + "\n")


    def write_arithmetic(self, command: str) -> None:
        """Writes a VM arithmetic command.

        Args:
            command (str): the command to write, can be "ADD", "SUB", "NEG",
            "EQ", "GT", "LT", "AND", "OR", "NOT", "SHIFTLEFT", "SHIFTRIGHT".
        """
        # Your code goes here!
```

```
60          self.output_stream.write(command.lower() + "\n")
61
62
63      def write_label(self, label: str) -> None:
64          """Writes a VM label command.
65
66          Args:
67              label (str): the label to write.
68          """
69          # Your code goes here!
70          self.output_stream.write("label " + label + "\n")
71
72
73      def write_goto(self, label: str) -> None:
74          """Writes a VM goto command.
75
76          Args:
77              label (str): the label to go to.
78          """
79          # Your code goes here!
80          self.output_stream.write("goto " + label + "\n")
81
82
83      def write_if(self, label: str) -> None:
84          """Writes a VM if-goto command.
85
86          Args:
87              label (str): the label to go to.
88          """
89          # Your code goes here!
90          self.output_stream.write("if-goto " + label + "\n")
91
92      def write_call(self, name: str, n_args: int) -> None:
93          """Writes a VM call command.
94
95          Args:
96              name (str): the name of the function to call.
97              n_args (int): the number of arguments the function receives.
98          """
99          # Your code goes here!
100         self.output_stream.write("call " + name +" " + str(n_args) + "\n")
101
102     def write_function(self, name: str, n_locals: int) -> None:
103         """Writes a VM function command.
104
105         Args:
106             name (str): the name of the function.
107             n_locals (int): the number of local variables the function uses.
108         """
109         # Your code goes here!
110         self.output_stream.write("function " + name + " " + str(n_locals) + "\n")
111
112
113     def write_return(self) -> None:
114         """Writes a VM return command."""
115         # Your code goes here!
116         self.output_stream.write("return" + "\n")
```