Contents

1 Basic Test Results

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

Finding usernames...
     Submission logins are: nogafri
     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 VMtranslator...
     Found in the correct path.

Testing FibonacciElement, where FibonacciElement is a directory...
     Testing your VMtranslator with command: './VMtranslator tst/FibonacciElement/'...
     Testing your output with command: './CPUEmulator.sh tst/FibonacciElement/FibonacciElement.tst'...
          Test passed.

Testing StaticsTest, where StaticsTest is a directory...
     Testing your VMtranslator with command: './VMtranslator tst/StaticsTest/'...
     Testing your output with command: './CPUEmulator.sh tst/StaticsTest/StaticsTest.tst'...
          Test passed.

Testing OrderOfFiles...
     Testing your code with one order of files in the directory...
     Testing your code with a different order...
          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

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

3 CodeWriter.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 CodeWriter:
    """Translates VM commands into Hack assembly code."""

    def __init__(self, output_stream: typing.TextIO) -> None:
        """Initializes the CodeWriter.

        Args:
            output_stream (typing.TextIO): output stream.
        """
        self.output_stream = output_stream
        self.filename = ""
        self.ram0_to_ram4 = {"local": "LCL", "argument": "ARG", "this": "THIS", "that": "THAT"}
        self.memory_segments = {"local": "LCL", "argument": "ARG", "this": "THIS", "that": "THAT",
                                "pointer": 3, "temp": 5}
        self.label_counter = 0  # for labels in comparison operations
        self.address_counter = 0  # for return address in write_call
        self.current_function = ""  # updates in write_function

    def set_file_name(self, filename: str) -> None:
        """Informs the code writer that the translation of a new VM file is
        started.

        Args:
            filename (str): The name of the VM file.
        """
        self.filename = filename
        print("translating file:", filename + ".vm")

    def _write_init(self) -> None:  # function added by me (according to youtube lecture API)
        """Writes assembly code that intializes the VM code (bootstrap code).
         Must be placed at the beginning of the generated *.asm file.
        """
        self.output_stream.write("@256\n")
        self.output_stream.write("D=A\n")
        self.output_stream.write("@SP\n")
        self.output_stream.write("M=D\n")
        self.write_call("Sys.init", 0)

    def _binary_operation(self, operation: str) -> None:  # function added by me
        """Writes assembly code for binary operations.

        Args:
            operation (str): binary operation.
        """
        self.output_stream.write("@SP\n") # pop first value
        self.output_stream.write("M=M-1\n")
        self.output_stream.write("A=M\n")
        self.output_stream.write("D=M\n")
        self.output_stream.write("@SP\n") # pop second value
        self.output_stream.write("M=M-1\n")
```

```
60          self.output_stream.write("A=M\n")
61          self.output_stream.write("M=M" + operation + "D\n")  # perform operation (+, -, &, |)
62          self.output_stream.write("@SP\n") # push result
63          self.output_stream.write("M=M+1\n")
64
65      def _comparison_operation(self, operation: str) -> None:  # function added by me
66          """Writes assembly code for comparison operations.
67
68          Args:
69              operation (str): comparison operation.
70          """
71          self.label_counter += 1
72          # could cause overflow if the values are large and one of the values is negative and the other is positive,
73          # so in the case of different signs, the function checks who is the negative and who is the positive
74          # and returns the appropriate result (instead of subtracting one from the other)
75          self.output_stream.write("@SP\n")  # pop first value
76          self.output_stream.write("M=M-1\n")
77          self.output_stream.write("A=M\n")
78          self.output_stream.write("D=M\n")
79          self.output_stream.write("@R13\n")  # store first value in R13
80          self.output_stream.write("M=D\n")
81
82          self.output_stream.write("@FIRST_POS" + str(self.label_counter) + "\n")  # check if first value is positive
83          self.output_stream.write("D;JGT\n")
84
85          self.output_stream.write("@SP\n")  # pop second value
86          self.output_stream.write("M=M-1\n")
87          self.output_stream.write("A=M\n")
88          self.output_stream.write("D=M\n")
89
90          self.output_stream.write("@SECOND_POS" + str(self.label_counter) + "\n")  # check if second value is positive
91          self.output_stream.write("D;JGT\n")
92
93          self.output_stream.write("@R13\n")  # load first value from R13
94          self.output_stream.write("D=D-M\n")  # perform operation to check if first value is greater than second value
95          self.output_stream.write("@COMPARE" + str(self.label_counter) + "\n")  # (reached only if both negative/0)
96          self.output_stream.write("0;JMP\n")
97
98          self.output_stream.write("(FIRST_POS" + str(self.label_counter) + ")\n")  # if first value is positive
99          self.output_stream.write("@SP\n")  # pop second value
100         self.output_stream.write("M=M-1\n")
101         self.output_stream.write("A=M\n")
102         self.output_stream.write("D=M\n")
103
104         self.output_stream.write("@SECOND_NEG" + str(self.label_counter) + "\n")  # check if second value is negative
105         self.output_stream.write("D;JLT\n")
106
107         self.output_stream.write("@R13\n")
108         self.output_stream.write("D=D-M\n")  # perform operation to check if first value is greater than second value
109         self.output_stream.write("@COMPARE" + str(self.label_counter) + "\n")  # (reached only if both positive/0)
110         self.output_stream.write("0;JMP\n")
111
112         self.output_stream.write("(SECOND_POS" + str(self.label_counter) + ")\n")
113         self.output_stream.write("D=1\n")      # reached if first value is negative and second value is positive
114         self.output_stream.write("@COMPARE" + str(self.label_counter) + "\n")
115         self.output_stream.write("0;JMP\n")
116
117         self.output_stream.write("(SECOND_NEG" + str(self.label_counter) + ")\n")
118         self.output_stream.write("D=-1\n")     # reached if first value is positive and second value is negative
119         self.output_stream.write("@COMPARE" + str(self.label_counter) + "\n")
120         self.output_stream.write("0;JMP\n")
121
122         self.output_stream.write("(COMPARE" + str(self.label_counter) + ")\n")
123         self.output_stream.write("@TRUE" + str(self.label_counter) + "\n")  # jump if operation result is true
124         self.output_stream.write("D;" + operation + "\n")
125
126         self.output_stream.write("D=0\n")
127         self.output_stream.write("@END" + str(self.label_counter) + "\n")  # if operation result is false
```

```
128             self.output_stream.write("0;JMP\n")
129
130             self.output_stream.write("(TRUE" + str(self.label_counter) + ")\n")
131             self.output_stream.write("D=-1\n")
132             self.output_stream.write("@END" + str(self.label_counter) + "\n")
133             self.output_stream.write("0;JMP\n")
134
135             self.output_stream.write("(END" + str(self.label_counter) + ")\n")
136             self.output_stream.write("@SP\n") # push result, D=0 if false and D=-1 if true
137             self.output_stream.write("A=M\n")
138             self.output_stream.write("M=D\n")
139             self.output_stream.write("@SP\n")
140             self.output_stream.write("M=M+1\n")
141
142         def _unary_operation(self, operation: str) -> None:  # function added by me
143             """Writes assembly code for unary operations.
144
145             Args:
146                 operation (str): unary operation.
147             """
148             self.output_stream.write("@SP\n") # pop value
149             self.output_stream.write("M=M-1\n")
150             self.output_stream.write("A=M\n")
151             if operation == ">>" or operation == "<<":
152                 self.output_stream.write("M=M" + operation + "\n")  # perform operation (<<, >>)
153             else:
154                 self.output_stream.write("M=" + operation + "M\n") # perform operation (-, !)
155             self.output_stream.write("@SP\n") # push result
156             self.output_stream.write("M=M+1\n")
157
158         def write_arithmetic(self, command: str) -> None:
159             """Writes assembly code that is the translation of the given
160             arithmetic command. For the commands eq, lt, gt, you should correctly
161             compare between all numbers our computer supports, and we define the
162             value "true" to be -1, and "false" to be 0.
163
164             Args:
165                 command (str): an arithmetic command.
166             """
167             # arithmetic operations:
168             if command == "add":
169                 self._binary_operation("+")
170             elif command == "sub":
171                 self._binary_operation("-")
172             elif command == "neg":
173                 self._unary_operation("-")
174
175             # comparison operations:
176             elif command == "eq":
177                 self._comparison_operation("JEQ")
178             elif command == "gt":
179                 self._comparison_operation("JGT")
180             elif command == "lt":
181                 self._comparison_operation("JLT")
182
183             # logical operations:
184             elif command == "and":
185                 self._binary_operation("&")
186             elif command == "or":
187                 self._binary_operation("|")
188             elif command == "not":
189                 self._unary_operation("!")
190
191             elif command == "shiftleft":
192                 self._unary_operation("<<")
193             elif command == "shiftright":
194                 self._unary_operation(">>")
195
```

```
196     def _push_D(self) -> None:  # function added by me
197         """
198         Writes assembly code that pushes the value in D onto the stack.
199         """
200         self.output_stream.write("@SP\n")
201         self.output_stream.write("A=M\n")
202         self.output_stream.write("M=D\n")
203         self.output_stream.write("@SP\n")
204         self.output_stream.write("M=M+1\n")
205 
206     def write_push(self, segment: str, index: int) -> None:
207         """Writes assembly code for the push command.
208 
209         Args:
210             segment (str): the memory segment to operate on.
211             index (int): the index in the memory segment.
212         """
213         if segment == "constant":
214             self.output_stream.write("@" + str(index) + "\n")  # load constant into D
215             self.output_stream.write("D=A\n")
216             self._push_D()  # push value onto stack
217 
218         elif segment in self.ram0_to_ram4:  # local, argument, this, that
219             self.output_stream.write("@" + str(index) + "\n")  # load index into D
220             self.output_stream.write("D=A\n")
221             self.output_stream.write("@" + self.ram0_to_ram4[segment] + "\n")  # load base address into A
222             self.output_stream.write("A=M+D\n")  # add index to base address
223             self.output_stream.write("D=M\n")  # load value at address into D
224             self._push_D()  # push value onto stack
225 
226         elif segment == "temp" or segment == "pointer":  # temp 0-7 = RAM 5-12, pointer 0 = RAM 3, pointer 1 = RAM 4
227             self.output_stream.write("@" + str(index) + "\n")  # load index into D
228             self.output_stream.write("D=A\n")
229             self.output_stream.write("@" + str(self.memory_segments[segment]) + "\n")  # load base address into A
230             self.output_stream.write("A=A+D\n")  # add index to base address
231             self.output_stream.write("D=M\n")  # load value at address into D
232             self._push_D()  # push value onto stack
233 
234         elif segment == "static":
235             self.output_stream.write("@" + self.filename + "." + str(index) + "\n")
236             self.output_stream.write("D=M\n")
237             self._push_D()  # push value onto stack
238 
239     def write_pop(self, segment: str, index: int) -> None:
240         """Writes assembly code for the pop command.
241 
242         Args:
243             segment (str): the memory segment to operate on.
244             index (int): the index in the memory segment.
245         """
246         if segment == "static":
247             self.output_stream.write("@SP\n")
248             self.output_stream.write("M=M-1\n")
249             self.output_stream.write("A=M\n")
250             self.output_stream.write("D=M\n")
251             self.output_stream.write("@" + self.filename + "." + str(index) + "\n")
252             self.output_stream.write("M=D\n")
253 
254         else:  # local, argument, this, that, pointer, temp
255             self.output_stream.write("@" + str(index) + "\n")  # load index into D
256             self.output_stream.write("D=A\n")
257             self.output_stream.write("@" + str(self.memory_segments[segment]) + "\n")
258             if segment in self.ram0_to_ram4:
259                 self.output_stream.write("A=M\n")
260             self.output_stream.write("D=A+D\n")  # add index to base address
261             self.output_stream.write("@R13\n")  # store address in R13
262             self.output_stream.write("M=D\n")
263             self.output_stream.write("@SP\n")  # pop value into D
```

```
264             self.output_stream.write("M=M-1\n")
265             self.output_stream.write("A=M\n")
266             self.output_stream.write("D=M\n")
267             self.output_stream.write("@R13\n")  # load address from R13
268             self.output_stream.write("A=M\n")
269             self.output_stream.write("M=D\n")
270 
271     def write_push_pop(self, command: str, segment: str, index: int) -> None:
272         """Writes assembly code that is the translation of the given
273         command, where command is either C_PUSH or C_POP.
274 
275         Args:
276             command (str): "C_PUSH" or "C_POP".
277             segment (str): the memory segment to operate on.
278             index (int): the index in the memory segment.
279         """
280         if command == "C_PUSH":
281             self.write_push(segment, index)
282 
283         elif command == "C_POP":
284             self.write_pop(segment, index)
285 
286     def write_label(self, label: str) -> None:
287         """Writes assembly code that affects the label command.
288         Let "Xxx.foo" be a function within the file Xxx.vm. The handling of
289         each "label bar" command within "Xxx.foo" generates and injects the symbol
290         "Xxx.foo$bar" into the assembly code stream.
291         When translating "goto bar" and "if-goto bar" commands within "foo",
292         the label "Xxx.foo$bar" must be used instead of "bar".
293 
294         Args:
295             label (str): the label to write.
296         """
297         self.output_stream.write("(" + self.current_function + "$" + label + ")\n")
298 
299     def write_goto(self, label: str) -> None:
300         """Writes assembly code that affects the goto command.
301 
302         Args:
303             label (str): the label to go to.
304         """
305         self.output_stream.write("@" + self.current_function + "$" + label + "\n")
306         self.output_stream.write("0;JMP\n")
307 
308     def write_if(self, label: str) -> None:
309         """Writes assembly code that affects the if-goto command.
310 
311         Args:
312             label (str): the label to go to.
313         """
314         self.output_stream.write("@SP\n")  # pop value into D
315         self.output_stream.write("M=M-1\n")
316         self.output_stream.write("A=M\n")
317         self.output_stream.write("D=M\n")
318         self.output_stream.write("@" + self.current_function + "$" + label + "\n")
319         self.output_stream.write("D;JNE\n")  # jump if D != 0
320 
321     def write_function(self, function_name: str, n_vars: int) -> None:
322         """Writes assembly code that affects the function command.
323         The handling of each "function Xxx.foo" command within the file Xxx.vm
324         generates and injects a symbol "Xxx.foo" into the assembly code stream,
325         that labels the entry-point to the function's code.
326         In the subsequent assembly process, the assembler translates this
327         symbol into the physical address where the function code starts.
328 
329         Args:
330             function_name (str): the name of the function.
331             n_vars (int): the number of local variables of the function.
```

```
332             """
333             self.current_function = function_name
334             self.output_stream.write("(" + function_name + ")\n")
335             for i in range(n_vars):
336                 self.write_push("constant", 0)  # initializes the local variables to 0
337 
338         def write_call(self, function_name: str, n_args: int) -> None:
339             """Writes assembly code that affects the call command.
340             Let "Xxx.foo" be a function within the file Xxx.vm.
341             The handling of each "call" command within Xxx.foo's code generates and
342             injects a symbol "Xxx.foo$ret.i" into the assembly code stream, where
343             "i" is a running integer (one such symbol is generated for each "call"
344             command within "Xxx.foo").
345             This symbol is used to mark the return address within the caller's
346             code. In the subsequent assembly process, the assembler translates this
347             symbol into the physical memory address of the command immediately
348             following the "call" command.
349 
350             Args:
351                 function_name (str): the name of the function to call.
352                 n_args (int): the number of arguments of the function.
353             """
354             self.address_counter += 1
355             return_address = function_name + "$ret" + str(self.address_counter)
356 
357             self.output_stream.write("@" + return_address + "\n")
358             self.output_stream.write("D=A\n")
359             self._push_D()  # push return address onto stack
360 
361             for seg in ["LCL", "ARG", "THIS", "THAT"]:  # saves seg of the caller
362                 self.output_stream.write("@" + seg + "\n")
363                 self.output_stream.write("D=M\n")
364                 self._push_D()  # push seg onto stack
365 
366             self.output_stream.write("@5\n")
367             self.output_stream.write("D=A\n")
368             self.output_stream.write("@" + str(n_args) + "\n")  # reposition ARG
369             self.output_stream.write("D=D+A\n")
370             self.output_stream.write("@SP\n")
371             self.output_stream.write("D=M-D\n")
372             self.output_stream.write("@ARG\n")
373             self.output_stream.write("M=D\n")
374 
375             self.output_stream.write("@SP\n")
376             self.output_stream.write("D=M\n")
377             self.output_stream.write("@LCL\n")  # reposition LCL
378             self.output_stream.write("M=D\n")
379 
380             self.output_stream.write("@" + function_name + "\n")  # transfer control to the callee
381             self.output_stream.write("0;JMP\n")
382 
383             self.output_stream.write("(" + return_address + ")\n")  # inject return address label into the code
384 
385         def write_return(self) -> None:
386             """Writes assembly code that affects the return command.
387             """
388             self.output_stream.write("@LCL\n")  # put return address in a temp var
389             self.output_stream.write("D=M\n")
390             self.output_stream.write("@R13\n")
391             self.output_stream.write("M=D\n")
392 
393             self.output_stream.write("@5\n")
394             self.output_stream.write("A=D-A\n")
395             self.output_stream.write("D=M\n")
396             self.output_stream.write("@R14\n")
397             self.output_stream.write("M=D\n")
398 
399             self.output_stream.write("@SP\n")  # reposition return value for the caller
```

```
400             self.output_stream.write("M=M-1\n")
401             self.output_stream.write("A=M\n")
402             self.output_stream.write("D=M\n")
403             self.output_stream.write("@ARG\n")
404             self.output_stream.write("A=M\n")
405             self.output_stream.write("M=D\n")
406 
407             self.output_stream.write("@ARG\n")  # reposition SP for the caller
408             self.output_stream.write("D=M+1\n")
409             self.output_stream.write("@SP\n")
410             self.output_stream.write("M=D\n")
411 
412             for seg in ["THAT", "THIS", "ARG", "LCL"]:
413                 self.output_stream.write("@R13\n")  # restore seg for caller
414                 self.output_stream.write("AM=M-1\n")
415                 self.output_stream.write("D=M\n")
416                 self.output_stream.write("@" + seg + "\n")
417                 self.output_stream.write("M=D\n")
418 
419             self.output_stream.write("@R14\n")  # go to return address
420             self.output_stream.write("A=M\n")
421             self.output_stream.write("0;JMP\n")
422 
423         def close(self) -> None:  # function added by me
424             """Closes the output file.
425             """
426             self.output_stream.close()
```

4 Main.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 Parser import Parser
from CodeWriter import CodeWriter

def translate_file(
        input_file: typing.TextIO, output_file: typing.TextIO,
        bootstrap: bool) -> None:
    """Translates a single file.

    Args:
        input_file (typing.TextIO): the file to translate.
        output_file (typing.TextIO): writes all output to this file.
        bootstrap (bool): if this is True, the current file is the
            first file we are translating.
    """
    parser = Parser(input_file)
    input_filename, input_extension = os.path.splitext(os.path.basename(input_file.name))  # gets the filename without the e
    code_writer.set_file_name(input_filename)
    if bootstrap:
        code_writer._write_init()  # runs only for the first file

    while parser.has_more_commands():
        parser.advance()
        command = parser.command_type()
        if command == "C_ARITHMETIC":
            code_writer.write_arithmetic(parser.arg1())
        elif command == "C_PUSH" or parser.command_type() == "C_POP":
            code_writer.write_push_pop(parser.command_type(), parser.arg1(), parser.arg2())
        elif command == "C_LABEL":
            code_writer.write_label(parser.arg1())
        elif command == "C_GOTO":
            code_writer.write_goto(parser.arg1())
        elif command == "C_IF":
            code_writer.write_if(parser.arg1())
        elif command == "C_FUNCTION":
            code_writer.write_function(parser.arg1(), parser.arg2())
        elif command == "C_RETURN":
            code_writer.write_return()
        elif command == "C_CALL":
            code_writer.write_call(parser.arg1(), parser.arg2())


if "__main__" == __name__:
    # Parses the input path and calls translate_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: VMtranslator <input path>")
```

```
    argument_path = os.path.abspath(sys.argv[1])
    if os.path.isdir(argument_path):
        files_to_translate = [
            os.path.join(argument_path, filename)
            for filename in os.listdir(argument_path)]
        output_path = os.path.join(argument_path, os.path.basename(
            argument_path))
    else:
        files_to_translate = [argument_path]
        output_path, extension = os.path.splitext(argument_path)
    output_path += ".asm"
    bootstrap = True
    with open(output_path, 'w') as output_file:
        code_writer = CodeWriter(output_file)
        for input_path in files_to_translate:
            filename, extension = os.path.splitext(input_path)
            if extension.lower() != ".vm":
                continue
            with open(input_path, 'r') as input_file:
                translate_file(input_file, output_file, bootstrap)
            bootstrap = False
```

5 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 'VMtranslator <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 VMtranslator

# 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/
```

6 Parser.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 Parser:
    """
    # Parser

    Handles the parsing of a single .vm file, and encapsulates access to the
    input code. It reads VM commands, parses them, and provides convenient
    access to their components.
    In addition, it removes all white space and comments.

    ## VM Language Specification

    A .vm file is a stream of characters. If the file represents a
    valid program, it can be translated into a stream of valid assembly
    commands. VM commands may be separated by an arbitrary number of whitespace
    characters and comments, which are ignored. Comments begin with "//" and
    last until the line's end.
    The different parts of each VM command may also be separated by an arbitrary
    number of non-newline whitespace characters.

    - Arithmetic commands:
      - add, sub, and, or, eq, gt, lt
      - neg, not, shiftleft, shiftright
    - Memory segment manipulation:
      - push &lt;segment&gt; <number>
      - pop &lt;segment that is not constant&gt; <number>
      - &lt;segment&gt; can be any of: argument, local, static, constant, this, that,
                                 pointer, temp
    - Branching (only relevant for project 8):
      - label <label-name>
      - if-goto <label-name>
      - goto <label-name>
      - <label-name> can be any combination of non-whitespace characters.
    - Functions (only relevant for project 8):
      - call <function-name> <n-args>
      - function <function-name> <n-vars>
      - return
    """

    def __init__(self, input_file: typing.TextIO) -> None:
        """Gets ready to parse the input file.

        Args:
            input_file (typing.TextIO): input file.
        """
        self.lines = input_file.read().splitlines()  # saves every line as an element in a list
        self.num_lines = len(self.lines)
        self.cur_line_num = -1
        self.cur_line = ""

```

```
60      def has_more_commands(self) -> bool:
61          """Are there more commands in the input?
62
63          Returns:
64              bool: True if there are more commands, False otherwise.
65          """
66          return self.cur_line_num < self.num_lines - 1  # True if the next potential line is within the list limits
67
68      def advance(self) -> None:
69          """Reads the next command from the input and makes it the current
70          command. Should be called only if has_more_commands() is true. Initially
71          there is no current command.
72          """
73          while self.has_more_commands():
74              self.cur_line_num += 1
75              self.cur_line = self.lines[self.cur_line_num]
76
77              self.cur_line = self.cur_line.replace('\t', '').replace('\n', '')  # removes all tabs and newlines
78              self.cur_line = self.cur_line.split("//", 1)[0]  # removes everything from "//" onwards (comments)
79              self.cur_line = self.cur_line.strip()  # removes all leading and trailing whitespaces
80
81              if self.cur_line == "":
82                  continue
83              break  # exits function if found a valid line
84
85      def command_type(self) -> str:
86          """
87          Returns:
88              str: the type of the current VM command.
89              "C_ARITHMETIC" is returned for all arithmetic commands.
90              For other commands, can return:
91              "C_PUSH", "C_POP", "C_LABEL", "C_GOTO", "C_IF", "C_FUNCTION",
92              "C_RETURN", "C_CALL".
93          """
94          if self.cur_line in ["add", "sub", "neg", "eq", "gt", "lt", "and", "or", "not", "shiftleft", "shiftright"]:
95              return "C_ARITHMETIC"
96          elif self.cur_line.startswith("push"):
97              return "C_PUSH"
98          elif self.cur_line.startswith("pop"):
99              return "C_POP"
100         elif self.cur_line.startswith("label"):
101             return "C_LABEL"
102         elif self.cur_line.startswith("goto"):
103             return "C_GOTO"
104         elif self.cur_line.startswith("if-goto"):
105             return "C_IF"
106         elif self.cur_line.startswith("function"):
107             return "C_FUNCTION"
108         elif self.cur_line.startswith("return"):
109             return "C_RETURN"
110         elif self.cur_line.startswith("call"):
111             return "C_CALL"
112
113     def arg1(self) -> str:
114         """
115         Returns:
116             str: the first argument of the current command. In case of
117             "C_ARITHMETIC", the command itself (add, sub, etc.) is returned.
118             Should not be called if the current command is "C_RETURN".
119         """
120         if self.cur_line in ["add", "sub", "neg", "eq", "gt", "lt", "and", "or", "not", "shiftleft", "shiftright"]:
121             return self.cur_line
122
123         # two arguments commands:
124         # first split returns a list of the form: ["command", " arg1 arg2"]
125         # second split returns a list of the form: [" ", "arg1", "arg2"]
126         elif self.cur_line.startswith("push"):
127             return self.cur_line.split("push", 1)[1].split(" ", 2)[1]
```

```
128             elif self.cur_line.startswith("pop"):
129                 return self.cur_line.split("pop", 1)[1].split(" ", 2)[1]
130             elif self.cur_line.startswith("function"):
131                 return self.cur_line.split("function", 1)[1].split(" ", 2)[1]
132             elif self.cur_line.startswith("call"):
133                 return self.cur_line.split("call", 1)[1].split(" ", 2)[1]
134 
135             # one argument commands:
136             # first split returns a list of the form: ["command", " arg1"]
137             # second split returns a list of the form: [" ", "arg1"]
138             elif self.cur_line.startswith("label"):
139                 return self.cur_line.split("label", 1)[1].split(" ", 1)[1]
140             elif self.cur_line.startswith("goto"):
141                 return self.cur_line.split("goto", 1)[1].split(" ", 1)[1]
142             elif self.cur_line.startswith("if-goto"):
143                 return self.cur_line.split("if-goto", 1)[1].split(" ", 1)[1]
144 
145         def arg2(self) -> int:
146             """
147             Returns:
148                 int: the second argument of the current command. Should be
149                 called only if the current command is "C_PUSH", "C_POP",
150                 "C_FUNCTION" or "C_CALL".
151             """
152             # first split returns a list of the form: ["command", " arg1 arg2"]
153             # second split returns a list of the form: [" ", "arg1", "arg2"]
154 
155             if self.cur_line.startswith("push"):
156                 return int(self.cur_line.split("push", 1)[1].split(" ", 2)[2])
157             elif self.cur_line.startswith("pop"):
158                 return int(self.cur_line.split("pop", 1)[1].split(" ", 2)[2])
159             elif self.cur_line.startswith("function"):
160                 return int(self.cur_line.split("function", 1)[1].split(" ", 2)[2])
161             elif self.cur_line.startswith("call"):
162                 return int(self.cur_line.split("call", 1)[1].split(" ", 2)[2])
```

7 VMtranslator

```
#!/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 'VMtranslator <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 "VMtranslator 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 "Main.py".
#            If your main is contained elsewhere, you will need to change this.

python3 Main.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/
```