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 StackTest...
     Testing your VMtranslator with command: './VMtranslator StackTest/StackTest.vm'...
     Testing your output with command: './CPUEmulator.sh StackTest/StackTest.tst'...
          Test passed.

Testing BasicTest...
     Testing your VMtranslator with command: './VMtranslator BasicTest/BasicTest.vm'...
     Testing your output with command: './CPUEmulator.sh BasicTest/BasicTest.tst'...
          Test passed.

Testing SimpleAdd...
     Testing your VMtranslator with command: './VMtranslator SimpleAdd/SimpleAdd.vm'...
     Testing your output with command: './CPUEmulator.sh SimpleAdd/SimpleAdd.tst'...
          Test passed.

Testing PointerTest...
     Testing your VMtranslator with command: './VMtranslator PointerTest/PointerTest.vm'...
     Testing your output with command: './CPUEmulator.sh PointerTest/PointerTest.tst'...
          Test passed.

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

```
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.label_counter = 0
        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}

    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.
        """
        # Your code goes here!
        # This function is useful when translating code that handles the
        # static segment. For example, in order to prevent collisions between two
        # .vm files which push/pop to the static segment, one can use the current
        # file's name in the assembly variable's name and thus differentiate between
        # static variables belonging to different files.
        # To avoid problems with Linux/Windows/MacOS differences with regards
        # to filenames and paths, you are advised to parse the filename in
        # the function "translate_file" in Main.py using python's os library,
        # For example, using code similar to:
        # input_filename, input_extension = os.path.splitext(os.path.basename(input_file.name))
        self.filename = filename

    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")
        self.output_stream.write("A=M\n")
        self.output_stream.write("M=M" + operation + "D\n")  # perform operation (+, -, &, |)
```

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

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

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

```
            self.output_stream.write("D=A+D\n")  # add index to base address
            self.output_stream.write("@R13\n")  # store address in R13
            self.output_stream.write("M=D\n")
            self.output_stream.write("@SP\n")  # pop value into D
            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("@R13\n")  # load address from R13
            self.output_stream.write("A=M\n")
            self.output_stream.write("M=D\n")

    def write_push_pop(self, command: str, segment: str, index: int) -> None:
        """Writes assembly code that is the translation of the given
        command, where command is either C_PUSH or C_POP.

        Args:
            command (str): "C_PUSH" or "C_POP".
            segment (str): the memory segment to operate on.
            index (int): the index in the memory segment.
        """
        if command == "C_PUSH":
            self.write_push(segment, index)

        elif command == "C_POP":
            self.write_pop(segment, index)

    def write_label(self, label: str) -> None:
        """Writes assembly code that affects the label command.
        Let "Xxx.foo" be a function within the file Xxx.vm. The handling of
        each "label bar" command within "Xxx.foo" generates and injects the symbol
        "Xxx.foo$bar" into the assembly code stream.
        When translating "goto bar" and "if-goto bar" commands within "foo",
        the label "Xxx.foo$bar" must be used instead of "bar".

        Args:
            label (str): the label to write.
        """
        # This is irrelevant for project 7,
        # you will implement this in project 8!
        pass

    def write_goto(self, label: str) -> None:
        """Writes assembly code that affects the goto command.

        Args:
            label (str): the label to go to.
        """
        # This is irrelevant for project 7,
        # you will implement this in project 8!
        pass

    def write_if(self, label: str) -> None:
        """Writes assembly code that affects the if-goto command.

        Args:
            label (str): the label to go to.
        """
        # This is irrelevant for project 7,
        # you will implement this in project 8!
        pass

    def write_function(self, function_name: str, n_vars: int) -> None:
        """Writes assembly code that affects the function command.
        The handling of each "function Xxx.foo" command within the file Xxx.vm
        generates and injects a symbol "Xxx.foo" into the assembly code stream,
        that labels the entry-point to the function's code.
        In the subsequent assembly process, the assembler translates this
        symbol into the physical address where the function code starts.
```

```
332
333             Args:
334                 function_name (str): the name of the function.
335                 n_vars (int): the number of local variables of the function.
336             """
337             # This is irrelevant for project 7,
338             # you will implement this in project 8!
339             # The pseudo-code of "function function_name n_vars" is:
340             # (function_name)       // injects a function entry label into the code
341             # repeat n_vars times:  // n_vars = number of local variables
342             #   push constant 0     // initializes the local variables to 0
343             pass
344
345         def write_call(self, function_name: str, n_args: int) -> None:
346             """Writes assembly code that affects the call command.
347             Let "Xxx.foo" be a function within the file Xxx.vm.
348             The handling of each "call" command within Xxx.foo's code generates and
349             injects a symbol "Xxx.foo$ret.i" into the assembly code stream, where
350             "i" is a running integer (one such symbol is generated for each "call"
351             command within "Xxx.foo").
352             This symbol is used to mark the return address within the caller's
353             code. In the subsequent assembly process, the assembler translates this
354             symbol into the physical memory address of the command immediately
355             following the "call" command.
356
357             Args:
358                 function_name (str): the name of the function to call.
359                 n_args (int): the number of arguments of the function.
360             """
361             # This is irrelevant for project 7,
362             # you will implement this in project 8!
363             # The pseudo-code of "call function_name n_args" is:
364             # push return_address   // generates a label and pushes it to the stack
365             # push LCL              // saves LCL of the caller
366             # push ARG              // saves ARG of the caller
367             # push THIS             // saves THIS of the caller
368             # push THAT             // saves THAT of the caller
369             # ARG = SP-5-n_args     // repositions ARG
370             # LCL = SP              // repositions LCL
371             # goto function_name    // transfers control to the callee
372             # (return_address)      // injects the return address label into the code
373             pass
374
375         def write_return(self) -> None:
376             """Writes assembly code that affects the return command."""
377             # This is irrelevant for project 7,
378             # you will implement this in project 8!
379             # The pseudo-code of "return" is:
380             # frame = LCL                   // frame is a temporary variable
381             # return_address = *(frame-5)   // puts the return address in a temp var
382             # *ARG = pop()                  // repositions the return value for the caller
383             # SP = ARG + 1                  // repositions SP for the caller
384             # THAT = *(frame-1)             // restores THAT for the caller
385             # THIS = *(frame-2)             // restores THIS for the caller
386             # ARG = *(frame-3)              // restores ARG for the caller
387             # LCL = *(frame-4)              // restores LCL for the caller
388             # goto return_address           // go to the return address
389             pass
```

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) -> None:
    """Translates a single file.

    Args:
        input_file (typing.TextIO): the file to translate.
        output_file (typing.TextIO): writes all output to this file.
    """
    parser = Parser(input_file)
    code_writer = CodeWriter(output_file)
    output_filename, output_extension = os.path.splitext(os.path.basename(output_file.name))
    code_writer.set_file_name(output_filename)

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

```
60              for filename in os.listdir(argument_path)]
61          output_path = os.path.join(argument_path, os.path.basename(
62              argument_path))
63      else:
64          files_to_translate = [argument_path]
65          output_path, extension = os.path.splitext(argument_path)
66      output_path += ".asm"
67      with open(output_path, 'w') as output_file:
68          for input_path in files_to_translate:
69              filename, extension = os.path.splitext(input_path)
70              if extension.lower() != ".vm":
71                  continue
72              with open(input_path, 'r') as input_file:
73                  translate_file(input_file, output_file)
```

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/
```