Contents

1 Basic Test Results

```
********** TESTING FOLDER STRUCTURE START **********
Checking your submission for presence of invalid (non-ASCII) characters...
No invalid characters found.
Submission logins are: linorcohen
Is this OK?
**********  TESTING FOLDER STRUCTURE END  **********

********** PROJECT TEST START **********
Running 'make'.
'make' ran successfully.
Testing.
Running command: './VMtranslator tst/FibonacciElement'
FibonacciElement.asm: passed the test
Running command: './VMtranslator tst/StaticsTest'
StaticsTest.asm: passed the test
**********  PROJECT TEST END  **********

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 linorcohen
2 Partner 1: Linor Cohen, linor.cohen@mail.huji.ac.il, 318861226
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
import textwrap
import os


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

    STATIC_ADDR = 16
    TEMP_ADDR = 5
    EMPTY = ""

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

        Args:
            output_stream (typing.TextIO): output stream.
        """
        self.output_file = output_stream
        self.filename = self.EMPTY
        self.function = self.EMPTY
        self.return_idx = 0
        self.counter = 0
        self.segment_table = {"local": "LCL", "argument": "ARG",
                              "this": "THIS", "that": "THAT",
                              "static": self.STATIC_ADDR,
                              "temp": self.TEMP_ADDR}

        self.jmp_table = {"eq": "JEQ", "gt": "JGT", "lt": "JLT"}

        self.operator_table = {"add": "+", "sub": "-", "neg": "-", "not": "!",
                               "shiftright": ">>", "shiftleft": "<<",
                               "and": "&", "or": "|"}

    def write_command_comment(self, command: str) -> None:
        """
        write a command comment before its assembly code
        :param command: a command
        """
        self.output_file.write("// " + command)

    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, input_extension = \
            os.path.splitext(os.path.basename(filename))

    def write_arithmetic(self, command: str) -> None:
```

```
60          """Writes assembly code that is the translation of the given
61          arithmetic command. For the commands eq, lt, gt, you should correctly
62          compare between all numbers our computer supports, and we define the
63          value "true" to be -1, and "false" to be 0.
64
65          Args:
66              command (str): an arithmetic command.
67          """
68          self.write_command_comment(command)
69          text = ""
70          if command in {"sub", "add"}:
71              text = self.__sub_add_commands(command)
72          elif command in {"neg", "not"}:
73              text = self.__neg_not_commands(command)
74          elif command in {"eq", "lt", "gt"}:
75              text = self.__boolean_commands(command)
76          elif command in {"shiftleft", "shiftright"}:
77              text = self.__shift_commands(command)
78          elif command in {"and", "or"}:
79              text = self.__and_or_commands(command)
80          self.output_file.write(textwrap.dedent(text))
81
82      def __sub_add_commands(self, command: str) -> str:
83          """
84          returns the assembly code for sub or add VM command.
85          :param command: (str) an sub or add command.
86          :return: assembly code translation of the command
87          """
88          return """
89          @SP
90          M=M-1
91          A=M
92          D=M
93          A=A-1
94          D=M{operator}D
95          M=D
96          """.format(operator=self.operator_table[command])
97
98      def __neg_not_commands(self, command: str) -> str:
99          """
100         returns the assembly code for neg or not VM command.
101         :param command: (str) an neg or not command.
102         :return: assembly code translation of the command
103         """
104         return """
105         @SP
106         A=M-1
107         M={operator}M
108         """.format(operator=self.operator_table[command])
109
110     def __boolean_commands(self, command: str) -> str:
111         """
112         returns the assembly code for lt,gt or eq VM command.
113         :param command: (str) an lt,gt or eq command.
114         :return: assembly code translation of the command
115         """
116         self.counter += 1
117         return """
118         @SP
119         M=M-1
120         A=M
121         D=M
122         @{file_name}.NEG_{command}_{i}
123         D;JLT
124         @SP
125         A=M-1
126         D=M
127         @{file_name}.POS_NEG_{command}_{i}
```

```
128             D;JLT
129             @{file_name}.SAME_SIGN_{command}_{i}
130             0;JMP
131             ({file_name}.NEG_{command}_{i})
132             @SP
133             A=M-1
134             D=M
135             @{file_name}.SAME_SIGN_{command}_{i}
136             D;JLT
137             D=1
138             @{file_name}.CHECK_COMMAND_{command}_{i}
139             0;JMP
140             ({file_name}.POS_NEG_{command}_{i})
141             D=-1
142             @{file_name}.CHECK_COMMAND_{command}_{i}
143             0;JMP
144             ({file_name}.SAME_SIGN_{command}_{i})
145             @SP
146             A=M
147             D=M
148             @SP
149             A=M-1
150             D=M-D
151             ({file_name}.CHECK_COMMAND_{command}_{i})
152             @{file_name}.TRUE_{command}_{i}
153             D;{command_jmp}
154             @SP
155             A=M-1
156             M=0
157             @{file_name}.{command}_{i}
158             0;JMP
159             ({file_name}.TRUE_{command}_{i})
160             @SP
161             A=M-1
162             M=-1
163             ({file_name}.{command}_{i})
164             """.format(file_name=self.filename, sub=self.__sub_add_commands("sub"), command=command.upper(),
165                        command_jmp=self.jmp_table[command], i=self.counter)
166 
167     def __shift_commands(self, command: str) -> str:
168         """
169         returns the assembly code for shiftleft or shiftright VM command.
170         :param command: (str) an shiftleft or shiftright command.
171         :return: assembly code translation of the command
172         """
173         return """
174         @SP
175         A=M-1
176         M=M{operator}
177         """.format(operator=self.operator_table[command])
178 
179     def __and_or_commands(self, command: str) -> str:
180         """
181         returns the assembly code for and or or VM command.
182         :param command: (str) an and or or command.
183         :return: assembly code translation of the command
184         """
185         return """
186         @SP
187         M=M-1
188         A=M
189         D=M
190         A=A-1
191         M=D{operator}M
192         """.format(operator=self.operator_table[command])
193 
194     def write_push_pop(self, command: str, segment: str, index: int) -> None:
195         """Writes assembly code that is the translation of the given
```

```
196             command, where command is either C_PUSH or C_POP.
197
198             Args:
199                 command (str): "C_PUSH" or "C_POP".
200                 segment (str): the memory segment to operate on.
201                 index (int): the index in the memory segment.
202             """
203             self.write_command_comment(
204                 f"{command[2:].lower()} {segment} {index}")
205             text = ""
206             if command == "C_PUSH":
207                 text = self.__get_push_command(segment, index)
208             elif command == "C_POP":
209                 text = self.__get_pop_command(segment, index)
210             self.output_file.write(textwrap.dedent(text))
211
212         def __get_push_command(self, segment: str, index: int) -> str:
213             """
214             returns the assembly code for the given push command
215             :param segment: the memory segment to operate on.
216             :param index: the index in the memory segment.
217             :return: assembly code translation of the command
218             """
219             if segment in {"local", "argument", "this", "that", "temp"}:
220                 return self.__lcl_arg_this_that_temp_push(segment, index)
221             elif segment == "static":
222                 return self.__static_push(index)
223             elif segment == "constant":
224                 return self.__constent_push(index)
225             elif segment == "pointer":
226                 return self.__pointer_push(index)
227
228         def __get_pop_command(self, segment: str, index: int) -> str:
229             """
230             returns the assembly code for the given pop command
231             :param segment: the memory segment to operate on.
232             :param index: the index in the memory segment.
233             :return: assembly code translation of the command
234             """
235             if segment in {"local", "argument", "this", "that", "temp"}:
236                 return self.__lcl_arg_this_that_temp_pop(segment, index)
237             elif segment == "static":
238                 return self.__static_pop(index)
239             elif segment == "pointer":
240                 return self.__pointer_pop(index)
241
242         def __lcl_arg_this_that_temp_push(self, segment: str, index: int) -> str:
243             """
244             returns the assembly code for push local, argument, this, that,
245             temp VM command.
246             :param segment: the memory segment to operate on.
247             :param index:  the index in the memory segment.
248             :return: assembly code translation of the command
249             """
250             return """
251             @{segmentPointer}
252             D={is_temp}
253             @{i}
254             A=D+A
255             D=M
256             @SP
257             A=M
258             M=D
259             @SP
260             M=M+1
261             """.format(segmentPointer=self.segment_table[segment], i=index,
262                        is_temp=(lambda x: "A" if x == "temp" else "M")(segment))
263
```

```
264     def __lcl_arg_this_that_temp_pop(self, segment: str, index: int) -> str:
265         """
266         returns the assembly code for pop local, argument, this, that,
267         temp VM command.
268         :param segment: the memory segment to operate on.
269         :param index:  the index in the memory segment.
270         :return: assembly code translation of the command
271         """
272         return """
273         @{segmentPointer}
274         D={is_temp}
275         @{i}
276         D=D+A
277         @R13
278         M=D
279         @SP
280         M=M-1
281         A=M
282         D=M
283         @R13
284         A=M
285         M=D
286         """.format(segmentPointer=self.segment_table[segment], i=index,
287                    is_temp=(lambda x: "A" if x == "temp" else "M")(segment))
288 
289     def __static_push(self, index: int) -> str:
290         """
291         returns the assembly code for push static VM command.
292         :param index:  the index in the memory segment.
293         :return: assembly code translation of the command
294         """
295         return """
296         @{file_name}.{i}
297         D=M
298         @SP
299         A=M
300         M=D
301         @SP
302         M=M+1
303         """.format(file_name=self.filename, i=index)
304 
305     def __static_pop(self, index: int) -> str:
306         """
307         returns the assembly code for pop static VM command.
308         :param index: the index in the memory segment.
309         :return: assembly code translation of the command
310         """
311         return """
312         @SP
313         M=M-1
314         A=M
315         D=M
316         @{file_name}.{i}
317         M=D
318         """.format(file_name=self.filename, i=index)
319 
320     def __constent_push(self, index: int) -> str:
321         """
322         returns the assembly code for push constant VM command.
323         :param index:  the index in the memory segment.
324         :return: assembly code translation of the command
325         """
326         return """
327         @{i}
328         D=A
329         @SP
330         A=M
331         M=D
```

```
            @SP
            M=M+1
            """.format(i=index)

    def __pointer_push(self, index: int) -> str:
        """
        returns the assembly code for push pointer (0/1 == THIS/THAT)
        VM command.
        :param index: the index in the memory segment.
        :return: assembly code translation of the command
        """
        return """
        @THIS
        D=A
        @{i}
        A=D+A
        D=M
        @SP
        A=M
        M=D
        @SP
        M=M+1
        """.format(i=index)

    def __pointer_pop(self, index: int) -> str:
        """
        returns the assembly code for pop pointer (0/1 == THIS/THAT)
        VM command.
        :param index: the index in the memory segment.
        :return: assembly code translation of the command
        """
        return """
        @THIS
        D=A
        @{i}
        D=D+A
        @R13
        M=D
        @SP
        M=M-1
        A=M
        D=M
        @R13
        A=M
        M=D
        """.format(i=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.
        """
        self.write_command_comment(f'label {label}')
        self.output_file.write(textwrap.dedent("""
        ({label})
        """.format(label=f'{self.filename}.{self.function}${label}')))

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

        Args:
            label (str): the label to go to.
```

```
        """
        self.write_command_comment(f'goto {label}')
        self.output_file.write(textwrap.dedent("""
        @{label}
        0;JMP
        """).format(label=f'{self.filename}.{self.function}${label}'))

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

        Args:
            label (str): the label to go to.
        """
        self.write_command_comment(f'if-goto {label}')
        self.output_file.write(textwrap.dedent("""
        @SP
        M=M-1
        A=M
        D=M
        @{label}
        D;JNE
        """).format(label=f'{self.filename}.{self.function}${label}'))

    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.

        Args:
            function_name (str): the name of the function.
            n_vars (int): the number of local variables of the function.
        """
        self.function = function_name
        self.write_command_comment(f'function {function_name} {n_vars}')
        self.output_file.write(textwrap.dedent("""
        ({label})
        """.format(label=function_name)))
        for i in range(n_vars):
            self.output_file.write(textwrap.dedent("""
        @SP
        A=M
        M=0
        @SP
        M=M+1
        """))

    def write_call(self, function_name: str, n_args: int) -> None:
        """Writes assembly code that affects the call command.
        Let "Xxx.foo" be a function within the file Xxx.vm.
        The handling of each "call" command within Xxx.foo's code generates and
        injects a symbol "Xxx.foo$ret.i" into the assembly code stream, where
        "i" is a running integer (one such symbol is generated for each "call"
        command within "Xxx.foo").
        This symbol is used to mark the return address within the caller's
        code. In the subsequent assembly process, the assembler translates this
        symbol into the physical memory address of the command immediately
        following the "call" command.

        Args:
            function_name (str): the name of the function to call.
            n_args (int): the number of arguments of the function.
        """
        return_label = f'{self.filename}.{function_name}$ret.{self.return_idx}'
        self.return_idx += 1
        self.write_command_comment(f'call {function_name} {n_args}')
```

```
468             self.output_file.write(
469                 textwrap.dedent(self.__set_call_saved_params(return_label) + """
470             @{nArgs}
471             D=A
472             @5
473             D=D+A
474             @SP
475             D=M-D
476             @ARG
477             M=D
478             @SP
479             D=M
480             @LCL
481             M=D
482             @{label}
483             0;JMP
484             ({return_label})
485             """.format(nArgs=n_args, label=function_name,
486                        return_label=return_label)))
487 
488         def __set_call_saved_params(self, return_label: str) -> str:
489             """
490             this function set call command parameters at the saved places
491             :param return_label: return label of the call
492             :return: asm code for the saved params
493             """
494             return \
495                 """{r_address}{save_lcl}{save_arg}{save_this}{save_that}""".format(
496                     r_address=self.__get_call_push_code(return_label, True),
497                     save_lcl=self.__get_call_push_code("local", False),
498                     save_arg=self.__get_call_push_code("argument", False),
499                     save_this=self.__get_call_push_code("this", False),
500                     save_that=self.__get_call_push_code("that", False))
501 
502         def __get_call_push_code(self, segment: str, is_label: bool) -> str:
503             """
504             this function gets call command push asm code
505             :param segment: call segment
506             :param is_label: true if label, false otherwise
507             :return: asm code
508             """
509             return """
510             @{segmentPointer}
511             D={is_label}
512             @SP
513             A=M
514             M=D
515             @SP
516             M=M+1""".format(segmentPointer=(
517                 lambda x: segment if x else self.segment_table[segment])(is_label),
518                             is_label=(lambda x: "A" if x else "M")(is_label))
519 
520         def write_return(self) -> None:
521             """Writes assembly code that affects the return command."""
522             self.write_command_comment('return')
523             self.output_file.write(textwrap.dedent("""
524             @LCL
525             D=M
526             @R14
527             M=D
528             @5
529             A=D-A
530             D=M
531             @R15
532             M=D
533             @SP
534             A=M-1
535             D=M
```

```
            @ARG
            A=M
            M=D
            @ARG
            D=M
            @SP
            M=D+1""" + self.__set_return_params() + """
            @R15
            A=M
            0;JMP
            """))

        # def write_return(self) -> None:
        #     """Writes assembly code that affects the return command."""
        #     self.write_command_comment('return')
        #     self.output_file.write(textwrap.dedent("""
        #     @LCL
        #     D=M
        #     @R14
        #     M=D
        #     @5
        #     A=D-A
        #     D=M
        #     @R15
        #     M=D
        #
        #     //pop argument 0
        #     @2
        #     D=M
        #     @0
        #     D=D+A
        #     @R13
        #     M=D
        #     @SP
        #     M=M-1
        #     A=M
        #     D=M
        #     @R13
        #     A=M
        #     M=D
        #     @2
        #     D=M
        #     @SP
        #     M=D+1
        #     @R14
        #     A=M-1
        #     D=M
        #     @4
        #     M=D
        #     @2
        #     D=A
        #     @R14
        #     A=M-D
        #     D=M
        #     @3
        #     M=D
        #     @3
        #     D=A
        #     @R14
        #     A=M-D
        #     D=M
        #     @2
        #     M=D
        #     @4
        #     D=A
        #     @R14
        #     A=M-D
        #     D=M
```

```
604         #     @1
605         #     M=D
606         #     @R15
607         #     A=M
608         #     0;JMP"""))
609
610         def __set_return_params(self) -> str:
611             """
612             this function sets the return parameters
613             :return: asm code of the return params
614             """
615             return """{save_that}{save_this}{save_arg}{save_local}""".format(
616                 save_that=self.__get_return_params("that"),
617                 save_this=self.__get_return_params("this"),
618                 save_arg=self.__get_return_params("argument"),
619                 save_local=self.__get_return_params("local"))
620
621         def __get_return_params(self, segment: str) -> str:
622             """
623             this function gets the return parameters
624             @segment: the return segment
625             :return: asm code of the return params
626             """
627             return """
628             @R14
629             M=M-1
630             A=M
631             D=M
632             @{segmentPointer}
633             M=D""".format(segmentPointer=self.segment_table[segment])
634
635         def bootstrap_init(self) -> None:
636             """
637             bootstrap initializer
638             :return: asm code of the bootstrap initializer
639             """
640             self.write_command_comment(f'bootstrap initialize')
641             self.output_file.write(textwrap.dedent("""
642             @256
643             D=A
644             @SP
645             M=D
646             """))
647             self.write_call("Sys.init", 0)
648
649         def close(self) -> None:
650             """
651             close the open file
652             """
653             self.output_file.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)
    code_writer = CodeWriter(output_file)

    # the current file is the first file (remove before passing a single file)
    if bootstrap:
        code_writer.bootstrap_init()

    code_writer.set_file_name(input_file.name)

    while parser.has_more_commands():
        parser.advance()
        c_type = parser.command_type()
        # return
        if c_type == parser.C_RETURN:
            code_writer.write_return()
            continue

        arg1 = parser.arg1()
        # arithmetic
        if c_type == parser.C_ARITHMETIC:
            code_writer.write_arithmetic(arg1)
        # push pop
        elif c_type in {parser.C_POP, parser.C_PUSH}:
            code_writer.write_push_pop(c_type, arg1, parser.arg2())
        # label
        elif c_type == parser.C_LABEL:
            code_writer.write_label(arg1)
        # goto
        elif c_type == parser.C_GOTO:
            code_writer.write_goto(arg1)
        # if-goto
        elif c_type == parser.C_IF:
            code_writer.write_if(arg1)
        # function
```

```
60          elif c_type == parser.C_FUNCTION:
61              code_writer.write_function(arg1, parser.arg2())
62          # call
63          elif c_type == parser.C_CALL:
64              code_writer.write_call(arg1, parser.arg2())
65
66
67  if "__main__" == __name__:
68      # Parses the input path and calls translate_file on each input file.
69      # This opens both the input and the output files!
70      # Both are closed automatically when the code finishes running.
71      # If the output file does not exist, it is created automatically in the
72      # correct path, using the correct filename.
73      if not len(sys.argv) == 2:
74          sys.exit("Invalid usage, please use: VMtranslator <input path>")
75      argument_path = os.path.abspath(sys.argv[1])
76      if os.path.isdir(argument_path):
77          files_to_translate = [
78              os.path.join(argument_path, filename)
79              for filename in os.listdir(argument_path)]
80          output_path = os.path.join(argument_path, os.path.basename(
81              argument_path))
82      else:
83          files_to_translate = [argument_path]
84          output_path, extension = os.path.splitext(argument_path)
85      output_path += ".asm"
86      bootstrap = True
87      with open(output_path, 'w') as output_file:
88          for input_path in files_to_translate:
89              filename, extension = os.path.splitext(input_path)
90              if extension.lower() != ".vm":
91                  continue
92              with open(input_path, 'r') as input_file:
93                  translate_file(input_file, output_file, bootstrap)
94              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 <segment> <number>
      - pop <segment that is not constant> <number>
      - <segment> 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
    """

    C_ARITHMETIC = "C_ARITHMETIC"
    C_PUSH = "C_PUSH"
    C_POP = "C_POP"
    C_LABEL = "C_LABEL"
    C_GOTO = "C_GOTO"
    C_IF = "C_IF"
    C_FUNCTION = "C_FUNCTION"
    C_RETURN = "C_RETURN"
    C_CALL = "C_CALL"
    command_table = {"push": C_PUSH, "pop": C_POP, "label": C_LABEL,
                     "goto": C_GOTO, "if-goto": C_IF, "function": C_FUNCTION,
```

```
                    "return": C_RETURN, "call": C_CALL}
    INITIAL_VAL = -1
    COMMENT = "//"
    NULL = "null"
    EMPTY_LST = []
    EMPTY = ""
    NOT_FOUND = -1

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

        Args:
            input_file (typing.TextIO): input file.
        """
        self.input_lines = input_file.read().splitlines()
        self.n = self.INITIAL_VAL
        self.cur_command_lst = self.EMPTY_LST

    def has_more_commands(self) -> bool:
        """Are there more commands in the input?

        Returns:
            bool: True if there are more commands, False otherwise.
        """
        while len(self.input_lines) - 1 != self.n:
            self.n += 1
            cur_command = self.input_lines[self.n].strip()
            if cur_command != self.EMPTY and cur_command[
                                             0:2] != self.COMMENT:
                return True
        return False

    def advance(self) -> None:
        """Reads the next command from the input and makes it the current
        command. Should be called only if has_more_commands() is true.
        Initially there is no current command.
        """
        # remove inline comments:
        cur_command = self.input_lines[self.n].strip()
        inline_comment_idx = cur_command.find(self.COMMENT)
        if inline_comment_idx != self.NOT_FOUND:
            cur_command = cur_command[0:inline_comment_idx]

        self.cur_command_lst = cur_command.split()

    def command_type(self) -> str:
        """
        Returns:
            str: the type of the current VM command.
            "C_ARITHMETIC" is returned for all arithmetic commands.
            For other commands, can return:
            "C_PUSH", "C_POP", "C_LABEL", "C_GOTO", "C_IF", "C_FUNCTION",
            "C_RETURN", "C_CALL".
        """
        if self.cur_command_lst[0] not in self.command_table:
            return self.C_ARITHMETIC
        return self.command_table[self.cur_command_lst[0]]

    def arg1(self) -> str:
        """
        Returns:
            str: the first argument of the current command. In case of
            "C_ARITHMETIC", the command itself (add, sub, etc.) is returned.
            Should not be called if the current command is "C_RETURN".
        """
        if self.command_type() == self.C_ARITHMETIC:
            return self.cur_command_lst[0]
        return self.cur_command_lst[1]
```

```
128
129     def arg2(self) -> int:
130         """
131         Returns:
132             int: the second argument of the current command. Should be
133             called only if the current command is "C_PUSH", "C_POP",
134             "C_FUNCTION" or "C_CALL".
135         """
136         return int(self.cur_command_lst[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/
```