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 your program with command: 'JackCompiler tst/ComplexArrays'.
Main.vm was created in test ComplexArrays.
Checking validity of generated VM code.
Generated VM code passed the test successfully.

Running your program with command: 'JackCompiler tst/Seven'.
Main.vm was created in test Seven.
Checking validity of generated VM code.
Generated VM code passed the test successfully.
**********  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

```
linorcohen
Partner 1: Linor Cohen, linor.cohen@mail.huji.ac.il, 318861226
Remarks:
```

3 CompilationEngine.py

```python
"""
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 JackTokenizer
from SymbolTable import SymbolTable
import VMWriter


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

    FUNCTION = "function"
    CONSTRUCTOR = "constructor"
    METHOD = "method"
    STATIC = "static"
    FIELD = "field"
    RETURN = "return"
    WHILE = "while"
    LET = "let"
    DO = "do"
    IF = "if"
    ELSE = "else"

    KEYWORD = "KEYWORD"
    SYMBOL = "SYMBOL"
    IDENTIFIER = "IDENTIFIER"
    INT_CONST = "INT_CONST"
    STRING_CONST = "STRING_CONST"

    def __init__(self, input_stream: "JackTokenizer", class_symbol_table: "SymbolTable",
                 vm_writer: "VMWriter", output_stream: typing.TextIO) -> None:
        """
        Creates a new compilation engine with the given input and output. The
        next routine called must be compileClass()
        :param input_stream: The input stream.
        :param output_stream: The output stream.
        """
        self.class_symbol_table = class_symbol_table
        self.vm_writer = vm_writer
        self.tokenizer = input_stream
        self.output_stream = output_stream

        self.op_terms = {"+": "ADD", "-": "SUB", "*": "call Math.multiply 2", "/": "call Math.divide 2",
                         "&": "AND", "|": "OR", "<": "LT", ">": "GT", "=": "EQ"}
        self.unary_op_terms = {"^": "SHIFTLEFT", "#": "SHIFTRIGHT", "-": "NEG", "~": "NOT"}

        self.class_name = ""
        self.counter = 0
        self.subroutine_symbol_table = SymbolTable()

    def __advance_tokenizer(self) -> None:
        """
        this method advance the tokenizer if has more tokens
        """
        if self.tokenizer.has_more_tokens():
            self.tokenizer.advance()

    def __get_current_token_and_advance(self) -> str:
        """
        this method advance the token and get the current token
        :return: Tuple(token, token tag type)
        """
        self.__advance_tokenizer()
        return self.__get_current_token()

    def __get_current_token(self) -> str:
        """
        this method return the tuple of the current token and the current token type tag.
        :return: Tuple(token, token tag type)
        """
        t_type = self.tokenizer.token_type()
        if t_type == self.KEYWORD:
            return self.tokenizer.keyword()
        elif t_type == self.SYMBOL:
            return self.tokenizer.symbol()
        elif t_type == self.IDENTIFIER:
            return self.tokenizer.identifier()
        elif t_type == self.INT_CONST:
            return str(self.tokenizer.int_val())
        elif t_type == self.STRING_CONST:
            return self.tokenizer.string_val()

    def __get_var_info_from_table(self, var_name: str) -> typing.Tuple[str, str, str]:
        """
        this method return the var info from its symbol table.
        :param var_name: variable name
        :return: variable type, variable kind, variable index
        """
        # symbol in class table
        if self.subroutine_symbol_table.kind_of(var_name) is not None:
            # symbol in subroutine table
            return self.subroutine_symbol_table.type_of(var_name), \
                   self.subroutine_symbol_table.kind_of(var_name), \
                   self.subroutine_symbol_table.index_of(var_name)
        return self.class_symbol_table.type_of(var_name), self.class_symbol_table.kind_of(
            var_name), self.class_symbol_table.index_of(var_name)

    def compile_class(self) -> None:
        """Compiles a complete class."""
        # class
        self.__get_current_token_and_advance()
        # className
        self.class_name = self.__get_current_token_and_advance()
        # {
        self.__get_current_token_and_advance()
        # classVarDec -> *
        token = self.__get_current_token_and_advance()
        while token in {self.FIELD, self.STATIC}:
            self.compile_class_var_dec()
            token = self.__get_current_token_and_advance()
        # subroutineDec -> *
        while token in {self.METHOD, self.CONSTRUCTOR, self.FUNCTION}:
            self.compile_subroutine()
            token = self.__get_current_token_and_advance()
        # }

    def compile_class_var_dec(self) -> None:
        """Compiles a static declaration or a field declaration."""
        # field or static
        kind = self.__get_current_token()
        # type
        token = self.__get_current_token_and_advance()
        var_type = token
        # varName -> *
        while token != ";":
            # varName
            name = self.__get_current_token_and_advance()
            # add to class_table
            self.class_symbol_table.define(name, var_type, kind)
            # symbol
            token = self.__get_current_token_and_advance()

    def compile_subroutine(self) -> None:
        """
        Compiles a complete method, function, or constructor.
        You can assume that classes with constructors have at least one field.
        """
        # keyword - method, function, or constructor.
        subroutine_type = self.__get_current_token()
        # identifier - return type
        self.__get_current_token_and_advance()
        # identifier - name
        subroutine_name = self.__get_current_token_and_advance()
        # reset the subroutine symbol table
        self.subroutine_symbol_table.start_subroutine()
        # add the object this to subroutine table
        if subroutine_type == self.METHOD:
            self.subroutine_symbol_table.define("this", self.class_name, "ARG")
        # (
        self.__get_current_token_and_advance()
        # parameter list
        self.compile_parameter_list()
        # )
        self.__get_current_token()
        # subroutine body
        self.__compile_subroutine_body(subroutine_name, subroutine_type)

    def __compile_subroutine_body(self, subroutine_name: str, subroutine_type: str) -> None:
        """
        this method compile a subroutine body
        """
        # {
        self.__get_current_token_and_advance()
        n = 0
        # var -> *
        var_type = self.__get_current_token_and_advance()
        while var_type == "var":
            n += self.compile_var_dec()
            var_type = self.__get_current_token_and_advance()
        # function className.subroutineName n
        self.vm_writer.write_function(f"""{self.class_name}.{subroutine_name}""", n)
        if subroutine_type == self.CONSTRUCTOR:
            # push const nField
            self.vm_writer.write_push("CONST", self.class_symbol_table.var_count(self.FIELD))
            # call Memory.alloc 1
            self.vm_writer.write_call("Memory.alloc", 1)
            # pop pointer 0
            self.vm_writer.write_pop("POINTER", 0)
        if subroutine_type == self.METHOD:
            # push argument 0
            self.vm_writer.write_push("ARG", 0)
            # pop pointer 0
            self.vm_writer.write_pop("POINTER", 0)
        # statements
        self.compile_statements(subroutine_type)
        # }
        self.__get_current_token()

    def compile_parameter_list(self) -> None:
        """Compiles a (possibly empty) parameter list, not including the
        enclosing "()".
        """
        is_first = True
        # varName -> *
        var_type = self.__get_current_token_and_advance()
        while var_type != ")":
            if is_first:
                name = self.__get_current_token_and_advance()
                # add to subroutine table
                self.subroutine_symbol_table.define(name, var_type, "ARG")
                var_type = self.__get_current_token_and_advance()
                is_first = False
            else:
                var_type = self.__get_current_token_and_advance()
                name = self.__get_current_token_and_advance()
                # add to subroutine table
                self.subroutine_symbol_table.define(name, var_type, "ARG")
                var_type = self.__get_current_token_and_advance()

    def compile_var_dec(self) -> int:
        """Compiles a var declaration."""
        # keyword
        kind = self.__get_current_token()
        # identifier
        token = self.__get_current_token_and_advance()
        var_type = token
        n = 0
        # varName -> *
        while token != ";":
            # identifier
            name = self.__get_current_token_and_advance()
            # add to subroutine table
            self.subroutine_symbol_table.define(name, var_type, kind.upper())
            n += 1
            # symbol
            token = self.__get_current_token_and_advance()
        return n

    def __compile_string(self, string_value: str) -> None:
        """
        this method compile a string constance
        :param string_value: the string variable
        """
        # push const length
        self.vm_writer.write_push("CONST", len(string_value))
        # call string.new
        self.vm_writer.write_call("String.new", 1)
        for char in string_value:
            # push const char
            self.vm_writer.write_push("CONST", str(ord(char)))
            # call string.appendChar
            self.vm_writer.write_call("String.appendChar", 2)

    def compile_statements(self, subroutine_type: str) -> None:
        """Compiles a sequence of statements, not including the enclosing
        "{}".
        """
        token = self.__get_current_token()
        while token != "}":
            if token == self.IF:
                self.compile_if(subroutine_type)
                token = self.__get_current_token()
            else:
                if token == self.DO:
                    self.compile_do()
                elif token == self.LET:
```

```python
                    self.compile_let()
                elif token == self.WHILE:
                    self.compile_while(subroutine_type)
                elif token == self.RETURN:
                    self.compile_return(subroutine_type)
                token = self.__get_current_token_and_advance()

    def __subroutine_call_format(self, obj_name: str, is_term: bool) -> None:
        """
        this method compile the subroutine call format
        """
        function_call_name = obj_name
        is_method = 0
        # . -> ?
        symbol = self.__get_current_token()
        if symbol == ".":
            var_type, var_kind, var_index = self.__get_var_info_from_table(obj_name)
            if var_type is not None and var_kind is not None and var_kind is not None:  # varName
                if not is_term:
                    # push obj
                    self.vm_writer.write_push(var_kind, var_index)
                is_method = 1
                function_call_name = var_type
            # functionName
            function_name = self.__get_current_token_and_advance()
            function_call_name += symbol + function_name
            # ( -> ?
            self.__get_current_token_and_advance()
        else:
            self.vm_writer.write_push("POINTER", 0)
            function_call_name = self.class_name + "." + obj_name
            is_method = 1
        # (
        # expression list
        self.__get_current_token_and_advance()
        n = self.compile_expression_list()
        # )
        self.__get_current_token()
        # output "call f n" or "call f n+1"
        self.vm_writer.write_call(function_call_name, n + is_method)

    def compile_do(self) -> None:
        """Compiles a do statement."""
        # keyword = do
        self.__get_current_token()
        # varName or className
        name = self.__get_current_token_and_advance()
        # . -> ?
        self.__get_current_token_and_advance()
        # subroutine call
        self.__subroutine_call_format(name, False)
        # ;
        self.__get_current_token_and_advance()
        self.vm_writer.write_pop("TEMP", 0)

    def compile_let(self) -> None:
        """Compiles a let statement."""
        # keyword = let
        self.__get_current_token()
        # identifier
        var_name = self.__get_current_token_and_advance()
        var_type, var_kind, var_index = self.__get_var_info_from_table(var_name)
        # [ -> ?
        symbol = self.__get_current_token_and_advance()
        # handle array
        is_array = False
        if symbol == "[":
            # push x
            self.vm_writer.write_push(var_kind, var_index)
            # expression
            self.__get_current_token_and_advance()
            self.compile_expression()
            # ]
            self.__get_current_token_and_advance()
            # add
            self.vm_writer.write_arithmetic("ADD")
            is_array = True
        # symbol
        self.__get_current_token()
        # expression
        self.__get_current_token_and_advance()
        self.compile_expression()
        # ;
        self.__get_current_token()
        if is_array:
            # pop temp 0
            self.vm_writer.write_pop("TEMP", 0)
            # pop pointer 1
            self.vm_writer.write_pop("POINTER", 1)
            # push temp 0
            self.vm_writer.write_push("TEMP", 0)
            # pop that 0
            self.vm_writer.write_pop("THAT", 0)
        else:
            self.vm_writer.write_pop(var_kind, var_index)

    def compile_while(self, subroutine_type: str) -> None:
        """Compiles a while statement."""
        # keyword = while
        self.__get_current_token()
        # label L1
        l1 = f"""{self.class_name}_L_{self.counter}"""
        self.counter += 1
        self.vm_writer.write_label(l1)
        # (
        self.__get_current_token_and_advance()
        # expression
        self.__get_current_token_and_advance()
        self.compile_expression()
        # )
        self.__get_current_token()
        # not
        self.vm_writer.write_arithmetic("NOT")
        # if-goto L2
        l2 = f"""{self.class_name}_L_{self.counter}"""
        self.counter += 1
        self.vm_writer.write_if(l2)
        # {
        # statements
        self.__get_current_token_and_advance()
        self.compile_statements(subroutine_type)
        # }
        self.__get_current_token()
        # goto L1
        self.vm_writer.write_goto(l1)
        # label L2
        self.vm_writer.write_label(l2)

    def compile_return(self, subroutine_type: str) -> None:
        """Compiles a return statement."""
        # keyword = return
        self.__get_current_token()
        # expression -> ?
        symbol = self.__get_current_token_and_advance()
        if symbol != ";":
            # expression
            self.compile_expression()
        else:
            if subroutine_type == self.CONSTRUCTOR:
                self.vm_writer.write_push("POINTER", 0)
            else:
                self.vm_writer.write_push("CONST", 0)
        self.vm_writer.write_return()

    def compile_if(self, subroutine_type: str) -> None:
        """Compiles a if statement, possibly with a trailing else clause."""
        # keyword = if
        self.__get_current_token()
        # (
        self.__get_current_token_and_advance()
        # expression
        self.__get_current_token_and_advance()
        self.compile_expression()
        # )
        self.__get_current_token()
        # not
        self.vm_writer.write_arithmetic("NOT")
        # if-goto L1
        l1 = f"""{self.class_name}_L_{self.counter}"""
        self.counter += 1
        self.vm_writer.write_if(l1)
        # {
        self.__get_current_token_and_advance()
        # statements
        self.__get_current_token_and_advance()
        self.compile_statements(subroutine_type)
        # }
        self.__get_current_token()
        # goto L2
        l2 = f"""{self.class_name}_L_{self.counter}"""
        self.counter += 1
        self.vm_writer.write_goto(l2)
        # label L1
        self.vm_writer.write_label(l1)
        # else -> ?
        token = self.__get_current_token_and_advance()
        if token == self.ELSE:
            # {
            self.__get_current_token_and_advance()
            # statements
            self.__get_current_token_and_advance()
            self.compile_statements(subroutine_type)
            # }
            self.__get_current_token()
            self.__get_current_token_and_advance()
        # label L2
        self.vm_writer.write_label(l2)

    def compile_expression(self) -> None:
        """Compiles an expression."""
        # term
        self.compile_term()
        # term -> *
        token = self.__get_current_token()
        while token != ")":
            if token not in self.op_terms:
                break
            # op
            op = token
            # term
            self.__get_current_token_and_advance()
            # term
            self.compile_term()
            # output "op"
            self.vm_writer.write_arithmetic(self.op_terms[op])
            token = self.__get_current_token()

    def compile_term(self) -> None:
        """Compiles a term.
        This routine is faced with a slight difficulty when
        trying to decide between some of the alternative parsing rules.
        Specifically, if the current token is an identifier, the routing must
        distinguish between a variable, an array entry, and a subroutine call.
        A single look-ahead token, which may be one of "[", "(", or "." suffices
        to distinguish between the three possibilities. Any other token is not
        part of this term and should not be advanced over.
        """
        # identifier / symbol
        token = self.__get_current_token()
        # push const
        if token.isnumeric():
            self.vm_writer.write_push("CONST", int(token))
        elif token == "true":
            self.vm_writer.write_push("CONST", 1)
            self.vm_writer.write_arithmetic("NEG")
        elif token == "false":
            self.vm_writer.write_push("CONST", 0)
        elif token == "null":
            self.vm_writer.write_push("CONST", 0)
        elif token == "this":
            self.vm_writer.write_push("POINTER", 0)
        elif self.tokenizer.token_type() == "STRING_CONST":
            self.__compile_string(token)
        # push var
        elif self.class_symbol_table.kind_of(token) is not None or \
                self.subroutine_symbol_table.kind_of(token) is not None:
            var_type, var_kind, var_index = self.__get_var_info_from_table(token)
            self.vm_writer.write_push(var_kind, var_index)
        # unary term -> ?
        if token in self.unary_op_terms:
            self.__get_current_token_and_advance()
            # term
            self.compile_term()
            # output unaryOp
            self.vm_writer.write_arithmetic(self.unary_op_terms[token])
        # expression - > ?
        elif token == "(":
            # expression
            self.__get_current_token_and_advance()
            self.compile_expression()
            # )
            # self.__get_current_token()
            self.__get_current_token_and_advance()
        else:
            function_call_name = token
            token = self.__get_current_token_and_advance()
            # handle array
            # [ -> ?
            if token == "[":
                # expression
                self.__get_current_token_and_advance()
                self.compile_expression()
                # ]
                # self.__get_current_token()
                self.__get_current_token_and_advance()
                # add
                self.vm_writer.write_arithmetic("ADD")
                # pop pointer 1
                self.vm_writer.write_pop("POINTER", 1)
                # push that 0
                self.vm_writer.write_push("THAT", 0)
            # subroutine call -> ?
```

```python
            elif token in {".", "("}:
                self.__subroutine_call_format(function_call_name, True)
                self.__get_current_token_and_advance()

    def compile_expression_list(self) -> int:
        """Compiles a (possibly empty) comma-separated list of expressions."""
        n = 0
        # expression -> ?
        token = self.__get_current_token()
        while token != ")":
            # expression
            self.compile_expression()
            n += 1
            token = self.__get_current_token()
            if token == ",":
                token = self.__get_current_token_and_advance()
        return n
```

4 JackCompiler

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

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

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

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

python3 JackCompiler.py $*

# This file is part of nand2tetris, as taught in The Hebrew University, and
# was written by Aviv Yaish. It is an extension to the specifications given
# in https://www.nand2tetris.org (Shimon Schocken and Noam Nisan, 2017),
# as allowed by the Creative Common Attribution-NonCommercial-ShareAlike 3.0
# Unported License: https://creativecommons.org/licenses/by-nc-sa/3.0/
```

5 JackCompiler.py

```python
"""
This file is part of nand2tetris, as taught in The Hebrew University, and
was written by Aviv Yaish. It is an extension to the specifications given
[here](https://www.nand2tetris.org) (Shimon Schocken and Noam Nisan, 2017),
as allowed by the Creative Common Attribution-NonCommercial-ShareAlike 3.0
Unported [License](https://creativecommons.org/licenses/by-nc-sa/3.0/).
"""
import os
import sys
import typing
from CompilationEngine import CompilationEngine
from JackTokenizer import JackTokenizer
from SymbolTable import SymbolTable
from VMWriter import VMWriter


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

    Args:
        input_file (typing.TextIO): the file to compile.
        output_file (typing.TextIO): writes all output to this file.
    """
    tokenizer = JackTokenizer(input_file)
    vm_writer = VMWriter(output_file)
    class_symbol_table = SymbolTable()
    engine = CompilationEngine(tokenizer, class_symbol_table,vm_writer, output_file)
    engine.compile_class()

    output_file.close()


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

6 JackTokenizer.py

```
"""
This file is part of nand2tetris, as taught in The Hebrew University, and
was written by Aviv Yaish. It is an extension to the specifications given
[here](https://www.nand2tetris.org) (Shimon Schocken and Noam Nisan, 2017),
as allowed by the Creative Common Attribution-NonCommercial-ShareAlike 3.0
Unported [License](https://creativecommons.org/licenses/by-nc-sa/3.0/).
"""
import re
import typing
import shlex


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

    # Jack Language Grammar

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

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

    ## Lexical Elements

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

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

    ## Program Structure

    A Jack program is a collection of classes, each appearing in a separate
    file. A compilation unit is a single class. A class is a sequence of tokens
    structured according to the following context free syntax:

    - class: 'class' className '{' classVarDec* subroutineDec* '}'
    - classVarDec: ('static' | 'field') type varName (',' varName)* ';'
    - type: 'int' | 'char' | 'boolean' | className
    - subroutineDec: ('constructor' | 'function' | 'method') ('void' | type)
    - subroutineName '(' parameterList ')' subroutineBody
    - parameterList: ((type varName) (',' type varName)*)?
    - subroutineBody: '{' varDec* statements '}'
    - varDec: 'var' type varName (',' varName)* ';'
    - className: identifier
    - subroutineName: identifier
    - varName: identifier

    ## Statements

    - statements: statement*
    - statement: letStatement | ifStatement | whileStatement | doStatement |
                 returnStatement
    - letStatement: 'let' varName ('[' expression ']')? '=' expression ';'
    - ifStatement: 'if' '(' expression ')' '{' statements '}' ('else' '{'
                   statements '}')?
    - whileStatement: 'while' '(' 'expression' ')' '{' statements '}'
    - doStatement: 'do' subroutineCall ';'
    - returnStatement: 'return' expression? ';'

    ## Expressions

    - expression: term (op term)*
    - term: integerConstant | stringConstant | keywordConstant | varName |
            varName '['expression']' | subroutineCall | '(' expression ')' |
            unaryOp term
    - subroutineCall: subroutineName '(' expressionList ')' | (className |
                      varName) '.' subroutineName '(' expressionList ')'
    - expressionList: (expression (',' expression)* )?
    - op: '+' | '-' | '*' | '/' | '&' | '|' | '<' | '>' | '='
    - unaryOp: '-' | '~' | '^' | '#'
    - keywordConstant: 'true' | 'false' | 'null' | 'this'

    Note that ^, # correspond to shiftleft and shiftright, respectively.
    """

    INITIAL_VAL = -1
    EMPTY_STR = ""
    EMPTY_LIST = []
    NOT_FOUND = -1
    COMMENT_TYPE_1 = "//"
    COMMENT_TYPE_2 = "/*"
    COMMENT_TYPE_2_END = "*/"
    COMMENT_TYPE_3 = "/**"

    TOKEN_SYMBOLS = {"{", "}", "(", ")", "[", "]", ".", ",", ";", "+", "-", "*", "/", "&", "|", "<", ">", "=",
                     "~", "^", "#"}

    TOKEN_KEYWORDS = {"class", "constructor", "function", "method", "field", "static", "var", "int",
                      "char", "boolean", "void", "true", "false", "null", "this", "let", "do", "if", "else",
                      "while", "return"}

    KEYWORD = "KEYWORD"
    SYMBOL = "SYMBOL"
    IDENTIFIER = "IDENTIFIER"
    INT_CONST = "INT_CONST"
    STRING_CONST = "STRING_CONST"

    def __init__(self, input_stream: typing.TextIO) -> None:
        """Opens the input stream and gets ready to tokenize it.

        Args:
            input_stream (typing.TextIO): input stream.
        """
        self.input_lines = input_stream.read().splitlines()
        self.n = self.INITIAL_VAL
        self.token_idx = self.INITIAL_VAL
```

```python
        self.token_lst = self.EMPTY_LIST

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

        Returns:
            bool: True if there are more tokens, False otherwise.
        """
        if self.token_idx + 1 == len(self.token_lst):
            self.token_idx = self.INITIAL_VAL
            comment = False
            while len(self.input_lines) - 1 != self.n:
                self.n += 1
                cur_token_line = self.input_lines[self.n].strip()
                if cur_token_line != self.EMPTY_STR:
                    if cur_token_line[0:2] == "/*" and cur_token_line[-2:0] == "*/":
                        continue
                    if cur_token_line[0:2] == "/*" or cur_token_line[0:3] == "/**":
                        comment = True
                    if comment and cur_token_line[-2:] == "*/":
                        comment = False
                        continue
                    if not comment and cur_token_line[0:2] != "//":
                        return True
            return False
        return True

    def __get_token_lst(self, line: str) -> typing.List[str]:
        token_line = line.replace('"', ' " ')
        temp_token_lst = list()
        for phrase in shlex.split(token_line, posix=False):
            if phrase[0] == '"':
                phrase = phrase[0] + phrase[2:-2] + phrase[-1]
                temp_token_lst.append(phrase)
            else:
                for word in phrase.split():
                    if word in self.TOKEN_KEYWORDS:
                        temp_token_lst.append(word)
                    else:
                        identifier = ""
                        for char in word:
                            if char not in self.TOKEN_SYMBOLS:
                                identifier += char
                            else:
                                if identifier != "":
                                    temp_token_lst.append(identifier)
                                    identifier = ""
                                temp_token_lst.append(char)
                        if identifier != "":
                            temp_token_lst.append(identifier)
        return temp_token_lst

    def advance(self) -> None:
        """Gets the next token from the input and makes it the current token.
        This method should be called if has_more_tokens() is true.
        Initially there is no current token.
        """
        if self.token_idx == self.INITIAL_VAL:
            cur_token_line = self.input_lines[self.n].strip()

            inline_comments = [i for i in range(len(cur_token_line)) if
                               cur_token_line.startswith(self.COMMENT_TYPE_1, i)]
            for i in inline_comments:
                if not self.__check_if_in_brackets(cur_token_line, i):
                    cur_token_line = cur_token_line[0:i]
                    break

            inline_comments = [i for i in range(len(cur_token_line)) if
```

```
                               cur_token_line.startswith(self.COMMENT_TYPE_2, i)]
            for i in inline_comments:
                if not self.__check_if_in_brackets(cur_token_line, i):
                    inline_comment_idx_end = cur_token_line.find(self.COMMENT_TYPE_2_END)
                    if inline_comment_idx_end != self.NOT_FOUND:
                        cur_token_line = cur_token_line[0:i] + cur_token_line[inline_comment_idx_end + 2:]
                        break

            inline_comment_idx = cur_token_line.find(self.COMMENT_TYPE_3)
            if inline_comment_idx != self.NOT_FOUND:
                cur_token_line = cur_token_line[0:inline_comment_idx]

            self.token_lst = self.__get_token_lst(cur_token_line)

        self.token_idx += 1

    def __check_if_in_brackets(self, cur_token_line: str, idx: int) -> bool:
        brackets = [m.start() for m in re.finditer('"', cur_token_line)]
        i = 0
        if len(brackets) == 0:
            return False
        while i + 2 <= len(brackets):
            if brackets[i] < idx < brackets[i + 1]:
                return True
            i += 2
        return False

    def token_type(self) -> str:
        """
        Returns:
            str: the type of the current token, can be
            "KEYWORD", "SYMBOL", "IDENTIFIER", "INT_CONST", "STRING_CONST"
        """
        if self.token_lst[self.token_idx] in self.TOKEN_KEYWORDS:
            return self.KEYWORD

        if self.token_lst[self.token_idx] in self.TOKEN_SYMBOLS:
            return self.SYMBOL

        if self.token_lst[self.token_idx].isdecimal():
            return self.INT_CONST

        if self.token_lst[self.token_idx][-1] == '"' and self.token_lst[self.token_idx][0] == '"':
            return self.STRING_CONST

        return self.IDENTIFIER

    def keyword(self) -> str:
        """
        Returns:
            str: the keyword which is the current token.
            Should be called only when token_type() is "KEYWORD".
            Can return "CLASS", "METHOD", "FUNCTION", "CONSTRUCTOR", "INT",
            "BOOLEAN", "CHAR", "VOID", "VAR", "STATIC", "FIELD", "LET", "DO",
            "IF", "ELSE", "WHILE", "RETURN", "TRUE", "FALSE", "NULL", "THIS"
        """
        return self.token_lst[self.token_idx]

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

```python

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

    def int_val(self) -> int:
        """
        Returns:
            str: the integer value of the current token.
            Should be called only when token_type() is "INT_CONST".
            Recall that integerConstant was defined in the grammar like so:
            integerConstant: A decimal number in the range 0-32767.
        """
        return int(self.token_lst[self.token_idx])

    def string_val(self) -> str:
        """
        Returns:
            str: the string value of the current token, without the double
            quotes. Should be called only when token_type() is "STRING_CONST".
            Recall that StringConstant was defined in the grammar like so:
            StringConstant: '"' A sequence of Unicode characters not including
                      double quote or newline '"'
        """
        return self.token_lst[self.token_idx][1:-1]
```

7 Makefile

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

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

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

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

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

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

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

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

# This file is part of nand2tetris, as taught in The Hebrew University, and
# was written by Aviv Yaish. It is an extension to the specifications given
# in https://www.nand2tetris.org (Shimon Schocken and Noam Nisan, 2017),
# as allowed by the Creative Common Attribution-NonCommercial-ShareAlike 3.0
# Unported License: https://creativecommons.org/licenses/by-nc-sa/3.0/
```

8 SymbolTable.py

```python
"""
This file is part of nand2tetris, as taught in The Hebrew University, and
was written by Aviv Yaish. It is an extension to the specifications given
[here](https://www.nand2tetris.org) (Shimon Schocken and Noam Nisan, 2017),
as allowed by the Creative Common Attribution-NonCommercial-ShareAlike 3.0
Unported [License](https://creativecommons.org/licenses/by-nc-sa/3.0/).
"""
import typing


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

    def __init__(self) -> None:
        """Creates a new empty symbol table."""
        self.index_table = {"STATIC": 0, "FIELD": 0, "ARG": 0, "VAR": 0}

        self.cur_symbol_table = {}

    def start_subroutine(self) -> None:
        """Starts a new subroutine scope (i.e., resets the subroutine's
        symbol table).
        """
        for key in self.index_table.keys():
            self.index_table[key] = 0

        self.cur_symbol_table.clear()

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

        Args:
            name (str): the name of the new identifier.
            var_type (str): the type of the new identifier.
            kind (str): the kind of the new identifier, can be:
            "STATIC", "FIELD", "ARG", "VAR".
        """
        self.cur_symbol_table[name] = [var_type, kind.upper(), self.index_table[kind.upper()]]
        self.index_table[kind.upper()] += 1

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

        Returns:
            int: the number of variables of the given kind already defined in
            the current scope.
        """
        var_count = 0
        for value in self.cur_symbol_table.values():
            if value[1] == kind.upper():
                var_count += 1
        return var_count

    def kind_of(self, name: str) -> typing.Optional[typing.Any]:
        """
        Args:
            name (str): name of an identifier.

        Returns:
            str: the kind of the named identifier in the current scope, or None
            if the identifier is unknown in the current scope.
        """
        if name not in self.cur_symbol_table:
            return None
        return self.cur_symbol_table[name][1]

    def type_of(self, name: str) -> typing.Optional[typing.Any]:
        """
        Args:
            name (str):  name of an identifier.

        Returns:
            str: the type of the named identifier in the current scope.
        """
        if name not in self.cur_symbol_table:
            return None
        return self.cur_symbol_table[name][0]

    def index_of(self, name: str) -> typing.Optional[typing.Any]:
        """
        Args:
            name (str):  name of an identifier.

        Returns:
            int: the index assigned to the named identifier.
        """
        if name not in self.cur_symbol_table:
            return None
        return self.cur_symbol_table[name][2]
```

9 VMWriter.py

```python
"""
This file is part of nand2tetris, as taught in The Hebrew University, and
was written by Aviv Yaish. It is an extension to the specifications given
[here](https://www.nand2tetris.org) (Shimon Schocken and Noam Nisan, 2017),
as allowed by the Creative Common Attribution-NonCommercial-ShareAlike 3.0
Unported [License](https://creativecommons.org/licenses/by-nc-sa/3.0/).
"""
import typing


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

    def __init__(self, output_stream: typing.TextIO) -> None:
        """Creates a new file and prepares it for writing VM commands."""
        self.output_file = output_stream

        self.segment_table = {"CONST": "constant", "ARG": "argument", "VAR": "local",
                              "STATIC": "static", "FIELD": "this", "THAT": "that",
                              "POINTER": "pointer", "TEMP": "temp"}

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

        Args:
            segment (str): the segment to push to, can be "CONST", "ARG",
            "LOCAL", "STATIC", "THIS", "THAT", "POINTER", "TEMP"
            index (int): the index to push to.
        """
        self.output_file.write("""push {segment} {index}\n""".format(segment=self.segment_table[segment],
                                                                      index=index))

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

        Args:
            segment (str): the segment to pop from, can be "CONST", "ARG",
            "LOCAL", "STATIC", "THIS", "THAT", "POINTER", "TEMP".
            index (int): the index to pop from.
        """
        self.output_file.write("""pop {segment} {index}\n""".format(segment=self.segment_table[segment],
                                                                     index=index))

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

        Args:
            command (str): the command to write, can be "ADD", "SUB", "NEG",
            "EQ", "GT", "LT", "AND", "OR", "NOT", "SHIFTLEFT", "SHIFTRIGHT".
        """
        if command.split()[0] == "call":
            self.output_file.write("""{command}\n""".format(command=command))
        else:
            self.output_file.write("""{command}\n""".format(command=command.lower()))

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

        Args:
            label (str): the label to write.
        """
        self.output_file.write("""label {label}\n""".format(label=label.upper()))

    def write_goto(self, label: str) -> None:
        """Writes a VM goto command.

        Args:
            label (str): the label to go to.
        """
        self.output_file.write("""goto {label}\n""".format(label=label.upper()))

    def write_if(self, label: str) -> None:
        """Writes a VM if-goto command.

        Args:
            label (str): the label to go to.
        """
        self.output_file.write("""if-goto {label}\n""".format(label=label.upper()))

    def write_call(self, name: str, n_args: int) -> None:
        """Writes a VM call command.

        Args:
            name (str): the name of the function to call.
            n_args (int): the number of arguments the function receives.
        """
        self.output_file.write("""call {name} {n_args}\n""".format(name=name, n_args=n_args))

    def write_function(self, name: str, n_locals: int) -> None:
        """Writes a VM function command.

        Args:
            name (str): the name of the function.
            n_locals (int): the number of local variables the function uses.
        """
        self.output_file.write("""function {name} {n_locals}\n""".format(name=name, n_locals=n_locals))

    def write_return(self) -> None:
        """Writes a VM return command."""
        self.output_file.write("return\n")

    def close(self) -> None:
        self.output_file.close()
```