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 JackCompiler...
     Found in the correct path.

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

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

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

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


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

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

2 AUTHORS

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

3 CompilationEngine.py

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

class CompilationEngine:
    """Gets input from a JackTokenizer and emits its parsed structure into an
    output stream.
    """
    def __init__(self, input_stream: "JackTokenizer", output_stream: VMWriter) -> 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.tokenizer = input_stream
        self.vm_writer = output_stream
        self.symbol_table = SymbolTable.SymbolTable()
        self.class_name = None
        self.subroutine_name = None

    def compile_class(self) -> None:
        """Compiles a complete class.
         syntax: 'class' className '{' classVarDec* subroutineDec* '}'."""
        self.tokenizer.advance()  # class
        self.tokenizer.advance()  # className
        self.class_name = self.tokenizer.identifier()
        self.tokenizer.advance()  # {
        while self.tokenizer.next_token() in ["static","field"]:
            self.compile_class_var_dec()
        while self.tokenizer.next_token() in ["constructor", "function", "method"]:
            self.compile_subroutine()
        self.tokenizer.advance()  # }

    def compile_class_var_dec(self) -> None:
        """Compiles a static declaration or a field declaration.
        syntax: ('static' | 'field') type varName (',' varName)* ';'."""
        while self.tokenizer.next_token() in ["static", "field"]:
            self.tokenizer.advance()  # static | field
            kind = self.tokenizer.keyword()
            self.tokenizer.advance()   # type
            type = self.get_type()  # (int | char | boolean | className)
            self.tokenizer.advance()  # varName
            name = self.tokenizer.identifier()
            self.symbol_table.define(name, type, kind)

            while self.tokenizer.next_token() == ",":  # (, varName)*
                self.tokenizer.advance()  # ,
                self.tokenizer.advance()  # varName
                name = self.tokenizer.identifier()
                self.symbol_table.define(name, type, kind)
```

```python

            self.tokenizer.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,
        you will understand why this is necessary in project 11.
        syntax: ('constructor' | 'function' | 'method') ('void' | type) subroutineName '(' parameterList ')' subroutineBody
        """
        self.tokenizer.advance()  # constructor | function | method
        function_type = self.tokenizer.keyword()
        self.tokenizer.advance()  # void | type
        self.tokenizer.advance()  # subroutineName
        self.subroutine_name = self.class_name + "." + self.tokenizer.identifier()
        self.symbol_table.start_subroutine(self.subroutine_name)  # reset the subroutine's symbol table
        self.symbol_table.set_scope(self.subroutine_name)  # set the current scope to the current subroutine's scope
        self.tokenizer.advance()  # (
        self.compile_parameter_list(function_type)
        self.tokenizer.advance()  # )
        self.compile_subroutine_body(function_type)

    def compile_subroutine_body(self, function_type: str) -> None:
        """Compiles the body of a subroutine."""
        self.tokenizer.advance()  # {
        while self.tokenizer.next_token() == "var":
            self.compile_var_dec()
        num_vars = self.symbol_table.var_count("var")
        self.vm_writer.write_function(self.subroutine_name, num_vars)  # function name num_vars

        if function_type == "method":
            self.vm_writer.write_push("argument", 0)
            self.vm_writer.write_pop("pointer", 0)
        elif function_type == "constructor":
            num_fields = self.symbol_table.globals_count("field")
            self.vm_writer.write_push("constant", num_fields)
            self.vm_writer.write_call("Memory.alloc", 1)
            self.vm_writer.write_pop("pointer", 0)

        self.compile_statements()
        self.tokenizer.advance()  # }
        self.symbol_table.set_scope("class")  # reset the current scope to class scope

    def compile_parameter_list(self, function_type) -> None:
        """Compiles a (possibly empty) parameter list, not including the enclosing "()".
        syntax: (type varName (',' type varName)*)? """
        if function_type == "method":
            self.symbol_table.define("this", "self", "arg")

        while self.tokenizer.next_token_type() != "SYMBOL":  # while more parameters are left
            self.tokenizer.advance()  # type
            type = self.get_type()  # int | char | boolean | className
            self.tokenizer.advance()  # varName
            name = self.tokenizer.identifier()
            self.symbol_table.define(name, type, "arg")

            if self.tokenizer.next_token() == ",":  # while more parameters are left
                self.tokenizer.advance()  # ,

    def compile_var_dec(self) -> None:
        """Compiles a var declaration.
         syntax: 'var' type varName (',' varName)* ';'."""
        self.tokenizer.advance()  # var
        kind = self.tokenizer.keyword()
        self.tokenizer.advance()  # type
        type = self.get_type()  # int | char | boolean | className
        self.tokenizer.advance()  # varName
        name = self.tokenizer.identifier()
        self.symbol_table.define(name, type, kind)

        while self.tokenizer.next_token() == ",":  # while more variables are left
            self.tokenizer.advance()  # ,
            self.tokenizer.advance()  # varName
            name = self.tokenizer.identifier()
            self.symbol_table.define(name, type, kind)

        self.tokenizer.advance()  # ;

    def compile_statements(self) -> None:
        """Compiles a sequence of statements, not including the enclosing {}".
        syntax: statement*"""
        while self.tokenizer.next_token() in ["let", "if", "while", "do", "return"]:
            if self.tokenizer.next_token() == "let":
                self.compile_let()
            elif self.tokenizer.next_token() == "if":
                self.compile_if()
            elif self.tokenizer.next_token() == "while":
                self.compile_while()
            elif self.tokenizer.next_token() == "do":
                self.compile_do()
            elif self.tokenizer.next_token() == "return":
                self.compile_return()

    def compile_do(self) -> None:
        """Compiles a do statement.
         syntax: 'do' subroutineCall ';'."""
        self.tokenizer.advance()  # do
        self.compile_subroutine_call()
        self.vm_writer.write_pop("temp", 0)
        self.tokenizer.advance() # ;

    def compile_let(self) -> None:
        """Compiles a let statement.
         syntax: 'let' varName ('[' expression ']')? '=' expression ';'."""
        array = False
        self.tokenizer.advance()  # let
        self.tokenizer.advance()  # varName
        name = self.tokenizer.identifier()

        if self.tokenizer.next_token() == "[":  # varName '[' expression ']'
            array = True
            self.tokenizer.advance()  # [
            self.compile_expression()
            self.tokenizer.advance()  # ]
            self.compile_array(name)
        self.tokenizer.advance()  # =
        self.compile_expression()
        if array:
            self.vm_writer.write_pop("temp", 0)
            self.vm_writer.write_pop("pointer", 1)
            self.vm_writer.write_push("temp", 0)
            self.vm_writer.write_pop("that", 0)
        else:
            self.pop_variable(name)
        self.tokenizer.advance()  # ;

    def compile_while(self) -> None:
        """Compiles a while statement.
         syntax: 'while' '(' 'expression' ')' '{' statements '}'."""
        count = self.symbol_table.index["while"]
        self.symbol_table.index["while"] += 1
        self.vm_writer.write_label("WHILE_EXP" + str(count))
        self.tokenizer.advance()  # while
        self.tokenizer.advance()  # (
        self.compile_expression()
        self.vm_writer.write_arithmetic("not")
        self.vm_writer.write_if("WHILE_END" + str(count))
        self.tokenizer.advance()  # )
        self.tokenizer.advance()  # {
        self.compile_statements()
        self.vm_writer.write_goto("WHILE_EXP" + str(count))
        self.vm_writer.write_label("WHILE_END" + str(count))
        self.tokenizer.advance()  # }

    def compile_return(self) -> None:
        """Compiles a return statement.
        syntax: 'return' expression? ';'"""
        self.tokenizer.advance()  # return
        empty = True
        if self.is_term():
            empty = False
            self.compile_expression()
        if empty:
            self.vm_writer.write_push("constant", 0)
        self.vm_writer.write_return()
        self.tokenizer.advance()  # ;

    def compile_if(self) -> None:
        """Compiles a if statement, possibly with a trailing else clause.
        syntax: 'if' '(' expression ')' '{' statements '}' ('else' '{' statements '}')?"""
        self.tokenizer.advance()  # if
        self.tokenizer.advance()  # (
        self.compile_expression()
        self.tokenizer.advance()  # )
        count = self.symbol_table.index["if"]
        self.symbol_table.index["if"] += 1
        self.vm_writer.write_if("IF_TRUE" + str(count))
        self.vm_writer.write_goto("IF_FALSE" + str(count))
        self.vm_writer.write_label("IF_TRUE" + str(count))
        self.tokenizer.advance()  # {
        self.compile_statements()
        self.tokenizer.advance()  # }
        if self.tokenizer.next_token() == "else":
            self.vm_writer.write_goto("IF_END" + str(count))
            self.vm_writer.write_label("IF_FALSE" + str(count))
            self.tokenizer.advance()  # else
            self.tokenizer.advance()  # {
            self.compile_statements()
            self.tokenizer.advance()  # }
            self.vm_writer.write_label("IF_END" + str(count))
        else:
            self.vm_writer.write_label("IF_FALSE" + str(count))

    def compile_expression(self) -> None:
        """Compiles an expression.
        syntax: term (op term)*"""
        self.compile_term()  # term
        while self.tokenizer.next_token() in ["+", "-", "*", "/", "&", "|", "<", ">", "="]:
            self.tokenizer.advance()  # op
            op = self.tokenizer.symbol()
            self.compile_term()
            if op == "+":
                self.vm_writer.write_arithmetic("add")
            elif op == "-":
                self.vm_writer.write_arithmetic("sub")
            elif op == "*":
                self.vm_writer.write_call("Math.multiply", 2)
            elif op == "/":
                self.vm_writer.write_call("Math.divide", 2)
            elif op == "&amp;":
                self.vm_writer.write_arithmetic("and")
            elif op == "|":
                self.vm_writer.write_arithmetic("or")
            elif op == "&lt;":
                self.vm_writer.write_arithmetic("lt")
            elif op == "&gt;":
                self.vm_writer.write_arithmetic("gt")
            elif op == "=":
                self.vm_writer.write_arithmetic("eq")

    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.
        syntax: integerConstant | stringConstant | keywordConstant | varName | varName '[' expression ']' |
         subroutineCall | '(' expression ')' | unaryOp term
        """
        array = False
        if self.tokenizer.next_token_type() == "INT_CONST":
            self.tokenizer.advance()
            value = str(self.tokenizer.int_val())  # integerConstant
            self.vm_writer.write_push("constant", value)

        elif self.tokenizer.next_token_type() == "STRING_CONST":
            self.tokenizer.advance()
            value = self.tokenizer.string_val()  # stringConstant
            self.vm_writer.write_push("constant", len(value))
            self.vm_writer.write_call("String.new", 1)
            for char in value:
                self.vm_writer.write_push("constant", ord(char))
                self.vm_writer.write_call("String.appendChar", 2)

        elif self.tokenizer.next_token() in ["true", "false", "null", "this"]:
            self.tokenizer.advance()
            value = self.tokenizer.keyword()  # keywordConstant
            if value == "this":
                self.vm_writer.write_push("pointer", 0)
            else:
                self.vm_writer.write_push("constant", 0)
                if value == "true":
                    self.vm_writer.write_arithmetic("not")

        elif self.tokenizer.next_token() == "(":  # '(' expression ')'
            self.tokenizer.advance()  # (
            self.compile_expression()
            self.tokenizer.advance()  # )

        elif self.tokenizer.next_token() in ["-", "~", "^", "#"]:  # unaryOp term
            self.tokenizer.advance()
            op = self.tokenizer.symbol()  # unaryOp
            self.compile_term()
            if op == "-":
                self.vm_writer.write_arithmetic("neg")
            elif op == "~":
                self.vm_writer.write_arithmetic("not")
            elif op == "^":
                self.vm_writer.write_arithmetic("shiftleft")
            elif op == "#":
                self.vm_writer.write_arithmetic("shiftright")

        else:  # identifier: varName | varName '[' expression ']' | subroutineCall
            self.tokenizer.advance()  # varName | subroutineName
            name = self.tokenizer.identifier()
            num_args = 0

            if self.tokenizer.next_token() == "[":  # varName '[' expression ']'
                array = True
                self.tokenizer.advance()  # [
                self.compile_expression()
                self.tokenizer.advance()  # ]
                self.compile_array(name)

            if self.tokenizer.next_token() == "(":
                num_args += 1
                self.vm_writer.write_push("pointer", 0)
                self.tokenizer.advance()  # (
                num_args += self.compile_expression_list()
                self.tokenizer.advance()  # )
                self.vm_writer.write_call(self.class_name + "." + name, num_args)

            elif self.tokenizer.next_token() == ".":
                self.tokenizer.advance()
                self.tokenizer.advance()
                second_name = self.tokenizer.identifier()
                if name in self.symbol_table.cur_scope or name in self.symbol_table.class_scope:
                    self.push_variable(name)
                    full_name = self.symbol_table.type_of(name) + "." + second_name
                    num_args += 1
                else:
                    full_name = name + "." + second_name
                self.tokenizer.advance()  # (
                num_args += self.compile_expression_list()
                self.tokenizer.advance()  # )
                self.vm_writer.write_call(full_name, num_args)

            else:  # varName
                if array:
                    self.vm_writer.write_pop("pointer", 1)
                    self.vm_writer.write_push("that", 0)
                else:
                    self.push_variable(name)

    def compile_expression_list(self) -> int:
        """Compiles a (possibly empty) comma-separated list of expressions.

        Returns:
            int: The number of expressions in the list.
        """
        count = 0
        if self.is_term():
            self.compile_expression()
            count += 1
        while self.tokenizer.next_token() == ",":
            self.tokenizer.advance()  # ,
            self.compile_expression()
            count += 1
        return count

    def is_term(self) -> bool:  # added
        """Checks if the current token is a term.

        Returns:
            bool: True if the next token is a term, False otherwise.
        """
        return self.tokenizer.next_token_type() == "INT_CONST" or self.tokenizer.next_token_type() == "STRING_CONST" or\
            self.tokenizer.next_token_type() == "KEYWORD" or self.tokenizer.next_token_type() == "IDENTIFIER" or \
            self.tokenizer.next_token() in ["(", "-", "~"]

    def get_type(self) -> str:  # added
        """Gets the type of the current token.

        Returns:
            str: The type of the current token.
        """
        if self.tokenizer.token_type() == "KEYWORD":
            return self.tokenizer.keyword()  # int | char | boolean
        else:
            return self.tokenizer.identifier()  # className

    def compile_subroutine_call(self) -> None:  # added
        """Compiles a subroutine call. The function is called after the first identifier of the subroutine call.
        syntax: subroutineName '(' expressionList ')' | (className | varName) '.' subroutineName '(' expressionList ')'
        """
        num_args = 0
        self.tokenizer.advance()  # subroutineName | className | varName
        first_name = self.tokenizer.identifier()

        if self.tokenizer.next_token() == ".":  # className | varName '.' subroutineName (chain of calls)
            self.tokenizer.advance()  # .
            self.tokenizer.advance()  # subroutineName
            second_name = self.tokenizer.identifier()
            if first_name in self.symbol_table.cur_scope or first_name in self.symbol_table.class_scope:
                self.push_variable(first_name)
                full_name = self.symbol_table.type_of(first_name) + "." + second_name
                num_args += 1
            else:
                full_name = first_name + "." + second_name

        else:  # subroutineName
            self.vm_writer.write_push("pointer", 0)
            full_name = self.class_name + "." + first_name
            num_args += 1

        self.tokenizer.advance()  # (
        num_args += self.compile_expression_list()
        self.vm_writer.write_call(full_name, num_args)
        self.tokenizer.advance()  # )

    def compile_array(self, name: str) -> None:  # added
        """Compiles an array by pushing the base address of the array and the index to the stack.
        """
        if name in self.symbol_table.cur_scope:
            if self.symbol_table.kind_of(name) == "var":
                self.vm_writer.write_push("local", self.symbol_table.index_of(name))
            elif self.symbol_table.kind_of(name) == "arg":
                self.vm_writer.write_push("argument", self.symbol_table.index_of(name))
        else:
            if self.symbol_table.kind_of(name) == "field":
                self.vm_writer.write_push("this", self.symbol_table.index_of(name))
            elif self.symbol_table.kind_of(name) == "static":
                self.vm_writer.write_push("static", self.symbol_table.index_of(name))
        self.vm_writer.write_arithmetic("add")

    def push_variable(self, name: str) -> None:  # added
        """Pushes the variable to the stack.

        Args:
            name (str): The name of the variable.
        """
        if name in self.symbol_table.cur_scope:
            if self.symbol_table.kind_of(name) == "var":
                self.vm_writer.write_push("local", self.symbol_table.index_of(name))
            elif self.symbol_table.kind_of(name) == "arg":
                self.vm_writer.write_push("argument", self.symbol_table.index_of(name))
        else:
            if self.symbol_table.kind_of(name) == "field":
                self.vm_writer.write_push("this", self.symbol_table.index_of(name))
            elif self.symbol_table.kind_of(name) == "static":
                self.vm_writer.write_push("static", self.symbol_table.index_of(name))

    def pop_variable(self, name: str) -> None:  # added
        """Pops the variable from the stack.

```

```python
        Args:
            name (str): The name of the variable.
        """
        if name in self.symbol_table.cur_scope:
            if self.symbol_table.kind_of(name) == "var":
                self.vm_writer.write_pop("local", self.symbol_table.index_of(name))
            elif self.symbol_table.kind_of(name) == "arg":
                self.vm_writer.write_pop("argument", self.symbol_table.index_of(name))
        else:
            if self.symbol_table.kind_of(name) == "field":
                self.vm_writer.write_pop("this", self.symbol_table.index_of(name))
            elif self.symbol_table.kind_of(name) == "static":
                self.vm_writer.write_pop("static", self.symbol_table.index_of(name))
```

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

```
"""
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)
    compilation_engine = CompilationEngine(tokenizer, vm_writer)
    compilation_engine.compile_class()


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

6 JackTokenizer.py

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

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.
    """

    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 = list(input_stream.read().splitlines())  # read lines into list
        self.clean_list()  # clean list to prepare for parsing
        self.line_tokens = []  # the tokens found in the current line
        self.cur_token = None  # the current token we are translating to the output stream

    def clean_list(self) -> None:  # added
        """Cleans and prepares the input_lines list for parsing. Removes comments, trailing whitespaces and empty lines.
        """
        self.remove_comments()  # remove all comments from the input
        self.input_lines = [line.strip() for line in self.input_lines]  # remove leading and trailing whitespace
        self.input_lines = [line for line in self.input_lines if line not in [" ", ""]]  # remove empty lines

    def remove_comments(self) -> None:  # added
        """Removes all comments from the input lines list unless the comment is inside a double quotes string.
        """
        clean_lines = []
        inside_double_quotes = False
        inside_multi_line_comment = False

        for line in self.input_lines:
            cleaned_line = ""
            i = 0
            while i < len(line):
                char = line[i]

                if char == '"':
                    if inside_multi_line_comment:  # if inside a multi-line comment, ignore double quotes
```

```
                        i += 1
                        continue
                    inside_double_quotes = not inside_double_quotes
                    cleaned_line += char
                    i += 1
                    continue

                if not inside_double_quotes:
                    if char == '/' and i + 1 < len(line) and line[i + 1] == '/':  # single-line comment, skip the rest
                        break
                    elif char == '/' and i + 1 < len(line) and line[i + 1] == '*':  # start of multi-line comment
                        inside_multi_line_comment = True
                        i += 2
                        continue
                    elif char == '*' and i + 1 < len(line) and line[i + 1] == '/':  # end of multi-line comment
                        inside_multi_line_comment = False
                        i += 2
                        continue
                    if not inside_multi_line_comment:  # if not inside a comment, add the character to the cleaned line
                        cleaned_line += char

                else:
                    cleaned_line += char
                i += 1

            clean_lines.append(cleaned_line)

        self.input_lines = clean_lines

    def next_token(self) -> str:  # added
        """Returns the next token in the input stream without advancing the current token.
        """
        if self.line_tokens != []:
            return self.line_tokens[0]

        else:
            token = self.parse_line(True)
            return token

    def next_token_type(self) -> str:  # added
        """Returns the type of the next token in the input stream without advancing the current token.
        """
        token = self.next_token()
        return self.token_type(token)

    def parse_line(self, flag=False):  # added
        """Parses the first line in the input list into tokens and saves them in a list.
         If the flag parameter is False, it saves the parsed line into self.line_tokens.
         Else, it returns the first token in the parsed line and doesn't save the parsing (used for checking the next
         token during compilation without advancing the current token).
        """
        re_keyword = "\b(?:class|constructor|function|method|field|static|var|int|char|boolean" \
                     "|void|true|false|null|this|let|do|if|else|while|return)\b"
        re_symbol = "[\\&\\*\\+\\(\\)\\.\\/\\,\\-\\]\\;\\~\\}\\|\\{\\>\\=\\[\\<\\^#]"
        re_int = "[0-9]+"
        re_str = "\"[^\"\n]*\""
        re_identifier = r"[a-zA-Z_]\w*"

        if flag:  # used when checking the next token
            input_lines = self.input_lines.copy()
            future_tokens = re.compile(re_keyword + "|" + re_symbol + "|" + re_int + "|" + re_str + "|" + re_identifier)
            future_tokens = future_tokens.findall(input_lines[0])
            return future_tokens[0]

        self.line_tokens = re.compile(re_keyword + "|" + re_symbol + "|" + re_int + "|" + re_str + "|" + re_identifier)
        self.line_tokens = self.line_tokens.findall(self.input_lines[0])

    def has_more_tokens(self) -> bool:
```

```python
        """Do we have more tokens in the input?

        Returns:
            bool: True if there are more tokens, False otherwise.
        """
        # check if there are more tokens in the current line or if there are more lines to parse
        return self.line_tokens != [] or self.input_lines != []

    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.has_more_tokens():

            while self.line_tokens == []:  # parse lines until a line with tokens is found
                self.parse_line()
                self.input_lines.pop(0)

            self.cur_token = self.line_tokens.pop(0)  # get the next token from the list and remove it

    def token_type(self, token=False) -> str:
        """
        Args:
            token (str): the token to check its type. If not given, checks the self.cur_token.

        Returns:
            str: the type of the token, can be
            "KEYWORD", "SYMBOL", "IDENTIFIER", "INT_CONST", "STRING_CONST"
        """
        if not token:
            token = self.cur_token

        if token in ["class", "constructor", "function", "method",
                      "field", "static", "var", "int", "char", "boolean",
                      "void", "true", "false", "null", "this", "let",
                      "do", "if", "else", "while", "return"]:
            return "KEYWORD"
        if token in ['{', '}', '(', ')', '[', ']', '.', ',', ';', '+',
                     '-', '*', '/', '&', '|', '<', '>', '=', '~', '^', '#']:
            return "SYMBOL"
        if token.isdigit():
            return "INT_CONST"
        if  token.startswith('"') and token.endswith('"'):
            return "STRING_CONST"
        return "IDENTIFIER"  # if none of the above

    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.cur_token

    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: '{' | '}' | '(' | ')' | '[' | ']' | '.' | ',' | ';' | '+' |
              '-' | '*' | '/' | '&' | '|' | '<' | '>' | '=' | '~' | '^' | '#'
        """
        if self.cur_token == '<':
            return '&lt;'
        if self.cur_token == '>':
            return '&gt;'
        if self.cur_token == '&':
            return '&amp;'
        return self.cur_token

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

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

    def string_val(self) -> str:
        """
        Returns:
            str: the string value of the current token, without the double
            quotes. Should be called only when token_type() is "STRING_CONST".
            Recall that StringConstant was defined in the grammar like so:
            StringConstant: '"' A sequence of Unicode characters not including
                      double quote or newline '"'
        """
        return self.cur_token[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.class_scope = {}
        self.subroutine_scope = {}
        self.cur_scope = None
        self.index = {"static": 0, "field": 0, "arg": 0, "var": 0, "while": 0, "if": 0}

    def start_subroutine(self, name) -> None:
        """Starts a new subroutine scope (i.e., resets the subroutine's
        symbol table).
        """
        self.subroutine_scope[name] = {}
        self.index["arg"] = 0
        self.index["var"] = 0
        self.index["while"] = 0
        self.index["if"] = 0

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

        Args:
            name (str): the name of the new identifier.
            type (str): the type of the new identifier.
            kind (str): the kind of the new identifier, can be:
            "STATIC", "FIELD", "ARG", "VAR".
        """
        if kind in ["static", "field"]:
            self.class_scope[name] = (type, kind, self.index[kind])
            self.index[kind] += 1
        else:  # kind in ["argument", "local"]
            self.cur_scope[name] = (type, kind, self.index[kind])
            self.index[kind] += 1

    def globals_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 class scope.
        """
        return len([v for (k, v) in self.class_scope.items() if v[1] == kind])

    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.
        """
        return len([v for (k, v) in self.cur_scope.items() if v[1] == kind])

    def kind_of(self, name: str):
        """
        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 in self.cur_scope:
            return self.cur_scope[name][1]
        elif name in self.class_scope:
            return self.class_scope[name][1]
        else:
            return None

    def type_of(self, name: str):
        """
        Args:
            name (str):  name of an identifier.

        Returns:
            str: the type of the named identifier in the current scope.
        """
        if name in self.cur_scope:
            return self.cur_scope[name][0]
        elif name in self.class_scope:
            return self.class_scope[name][0]
        else:
            return None

    def index_of(self, name: str):
        """
        Args:
            name (str):  name of an identifier.

        Returns:
            int: the index assigned to the named identifier.
        """
        if name in self.cur_scope:
            return self.cur_scope[name][2]
        elif name in self.class_scope:
            return self.class_scope[name][2]
        else:
            return None

    def set_scope(self, subroutine_name: str):
        """Sets the current scope to the given scope.

        Args:
            subroutine_name (str): the scope to set.
        """
        if subroutine_name == "class":
            self.cur_scope = self.class_scope
        else:
            self.cur_scope = self.subroutine_scope[subroutine_name]
```

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
import SymbolTable

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_stream = output_stream
        self.cur_symbol_table = None

    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_stream.write("push " + segment + " " + str(index) + "\n")

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

        Args:
            segment (str): the segment to pop from, can be "CONST", "ARG",
            "LOCAL", "STATIC", "THIS", "THAT", "POINTER", "TEMP".
            index (int): the index to pop from.
        """
        self.output_stream.write("pop " + segment + " " + str(index) + "\n")

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

        Args:
            command (str): the command to write, can be "ADD", "SUB", "NEG",
            "EQ", "GT", "LT", "AND", "OR", "NOT", "SHIFTLEFT", "SHIFTRIGHT".
        """
        self.output_stream.write(command.lower() + "\n")

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

        Args:
            label (str): the label to write.
        """
        self.output_stream.write("label " + label + "\n")


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

        Args:
            label (str): the label to go to.
        """
        self.output_stream.write("goto " + label + "\n")

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

        Args:
            label (str): the label to go to.
        """
        self.output_stream.write("if-goto " + label + "\n")

    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_stream.write("call " + name + " " + str(n_args) + "\n")

    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_stream.write("function " + name + " " + str(n_locals) + "\n")

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