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: 'JackAnalyzer tst/ArrayTest'.
Main.xml was created in test ArrayTest.
The diff is OK on the file Main.xml in test ArrayTest.
Running your program with command: 'JackAnalyzer tst/Square'.
Main.xml was created in test Square.
The diff is OK on the file Main.xml in test Square.
SquareGame.xml was created in test Square.
The diff is OK on the file SquareGame.xml in test Square.
Square.xml was created in test Square.
The diff is OK on the file Square.xml in test Square.
**********  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


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"

    CLASS_TAG = "class"
    CLASS_VAR_DEC_TAG = "classVarDec"
    SUBROUTINE_DEC_TAG = "subroutineDec"
    SUBROUTINE_BODY_TAG = "subroutineBody"
    VAR_DEC_TAG = "varDec"
    PARAMETER_LIST_TAG = "parameterList"
    STATEMENTS_TAG = "statements"
    EXPRESSION_LIST_TAG = "expressionList"
    TERM_TAG = "term"
    EXPRESSION_TAG = "expression"
    IF_STATEMENT_TAG = "ifStatement"
    RETURN_STATEMENT_TAG = "returnStatement"
    WHILE_STATEMENT_TAG = "whileStatement"
    LET_STATEMENT_TAG = "letStatement"
    DO_STATEMENT_TAG = "doStatement"

    KEYWORD_TAG = "keyword"
    SYMBOL_TAG = "symbol"
    IDENTIFIER_TAG = "identifier"
    INT_CONST_TAG = "integerConstant"
    STRING_CONST_TAG = "stringConstant"

    def __init__(self, input_stream: JackTokenizer, 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.tokenizer = input_stream
        self.output_stream = output_stream

        self.op_terms = {"+", "-", "*", "/", "&", "|", "<", ">", "="}
        self.unary_op_terms = {"^", "#", "-", "~"}

        self.indentation = ""

    def __write_open_tag(self, tag: str) -> None:
        """
        this method writes the open tag to the output file
        :param tag: given token tag
        """
        self.output_stream.write(self.indentation + "<" + tag + ">")

    def __write_close_tag(self, tag: str) -> None:
        """
        this method writes the closing tag to the output file
        :param tag: given token tag
        """
        self.output_stream.write(self.indentation + "</" + tag + ">")
        self.output_stream.write("\n")

    def __write_open_and_close_tag(self, tag: str, token: str) -> None:
        """
        this method writes open and close tag to the output file, used for inline tags
        :param tag: given token tag
        :param token: given token
        """
        self.__write_open_tag(tag)
        if token == "<":
            token = "&lt;"
        elif token == ">":
            token = "&gt;"
        elif token == "&":
            token = "&amp;"
        self.output_stream.write(" " + token)
        self.output_stream.write(" </" + tag + ">")
        self.output_stream.write("\n")

    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) -> typing.Tuple[str, 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) -> typing.Tuple[str, 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(), self.KEYWORD_TAG
        elif t_type == self.SYMBOL:
            return self.tokenizer.symbol(), self.SYMBOL_TAG
        elif t_type == self.IDENTIFIER:
            return self.tokenizer.identifier(), self.IDENTIFIER_TAG
        elif t_type == self.INT_CONST:
            return str(self.tokenizer.int_val()), self.INT_CONST_TAG
        elif t_type == self.STRING_CONST:
            return self.tokenizer.string_val(), self.STRING_CONST_TAG

    def compile_class(self) -> None:
        """Compiles a complete class."""
        self.__write_open_tag(self.CLASS_TAG)
        self.output_stream.write("\n")
        self.indentation += "  "
        # class
        self.__write_next_advanced_token()
        # className
        self.__write_next_advanced_token()
        # {
        self.__write_next_advanced_token()
        # classVarDec -> *
        token, token_type = self.__get_current_token_and_advance()
        while token in {self.FIELD, self.STATIC}:
            self.compile_class_var_dec()
            token, token_type = self.__get_current_token_and_advance()
        # subroutineDec -> *
        while token in {self.METHOD, self.CONSTRUCTOR, self.FUNCTION}:
            self.compile_subroutine()
            token, token_type = self.__get_current_token_and_advance()
        # }
        self.__write_open_and_close_tag(token_type, token)
        self.indentation = self.indentation[:-2]
        self.__write_close_tag(self.CLASS_TAG)

    def __write_next_advanced_token(self) -> None:
        """
        this method advance the token and writs the open close tag of the current token
        """
        token, token_type = self.__get_current_token_and_advance()
        self.__write_open_and_close_tag(token_type, token)

    def __writes_current_token(self) -> None:
        """
        this method writes the current token without advancing the tokenizer
        """
        token, token_type = self.__get_current_token()
        self.__write_open_and_close_tag(token_type, token)

    def compile_class_var_dec(self) -> None:
        """Compiles a static declaration or a field declaration."""
        self.__write_open_tag(self.CLASS_VAR_DEC_TAG)
        self.indentation += "  "
        self.output_stream.write("\n")
        # field or static
        self.__writes_current_token()
        # type
        token, token_type = self.__get_current_token_and_advance()
        self.__write_open_and_close_tag(token_type, token)
        # varName -> *
        while token != ";":
            # identifier
            self.__write_next_advanced_token()
            # symbol
            token, token_type = self.__get_current_token_and_advance()
            self.__write_open_and_close_tag(token_type, token)
        self.indentation = self.indentation[:-2]
        self.__write_close_tag(self.CLASS_VAR_DEC_TAG)

    def compile_subroutine(self) -> None:
```

```python
        """
        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.
        """
        self.__write_open_tag(self.SUBROUTINE_DEC_TAG)
        self.indentation += "  "
        self.output_stream.write("\n")
        # keyword
        self.__writes_current_token()
        # identifier
        self.__write_next_advanced_token()
        # identifier
        self.__write_next_advanced_token()
        # (
        self.__write_next_advanced_token()
        # parameter list
        self.compile_parameter_list()
        # )
        self.__writes_current_token()
        # subroutine body
        self.__compile_subroutine_body()
        self.indentation = self.indentation[:-2]
        self.__write_close_tag(self.SUBROUTINE_DEC_TAG)

    def __compile_subroutine_body(self) -> None:
        """
        this method compile a subroutine body
        """
        self.__write_open_tag(self.SUBROUTINE_BODY_TAG)
        self.indentation += "  "
        self.output_stream.write("\n")
        # {
        self.__write_next_advanced_token()
        # var -> *
        token, token_type = self.__get_current_token_and_advance()
        while token == "var":
            self.compile_var_dec()
            token, token_type = self.__get_current_token_and_advance()
        # statements
        self.compile_statements()
        # }
        self.__writes_current_token()
        self.indentation = self.indentation[:-2]
        self.__write_close_tag(self.SUBROUTINE_BODY_TAG)

    def compile_parameter_list(self) -> None:
        """Compiles a (possibly empty) parameter list, not including the
        enclosing "()".
        """
        self.__write_open_tag(self.PARAMETER_LIST_TAG)
        self.indentation += "  "
        self.output_stream.write("\n")
        # varName -> *
        token, token_type = self.__get_current_token_and_advance()
        while token != ")":
            self.__write_open_and_close_tag(token_type, token)
            token, token_type = self.__get_current_token_and_advance()
        self.indentation = self.indentation[:-2]
        self.__write_close_tag(self.PARAMETER_LIST_TAG)

    def compile_var_dec(self) -> None:
        """Compiles a var declaration."""
        self.__write_open_tag(self.VAR_DEC_TAG)
        self.indentation += "  "
        self.output_stream.write("\n")
        # keyword
        self.__writes_current_token()
        # identifier
        token, token_type = self.__get_current_token_and_advance()
        self.__write_open_and_close_tag(token_type, token)
        # varName -> *
        while token != ";":
            # identifier
            self.__write_next_advanced_token()
            # symbol
            token, token_type = self.__get_current_token_and_advance()
            self.__write_open_and_close_tag(token_type, token)
        self.indentation = self.indentation[:-2]
        self.__write_close_tag(self.VAR_DEC_TAG)

    def compile_statements(self) -> None:
        """Compiles a sequence of statements, not including the enclosing
        "{}".
        """
        self.__write_open_tag(self.STATEMENTS_TAG)
        self.indentation += "  "
        self.output_stream.write("\n")
        token, token_type = self.__get_current_token()
        while token != "}":
            if token == self.IF:
                self.compile_if()
                token, token_type = self.__get_current_token()
            else:
                if token == self.DO:
                    self.compile_do()
                elif token == self.LET:
                    self.compile_let()
                elif token == self.WHILE:
                    self.compile_while()
                elif token == self.RETURN:
                    self.compile_return()
                token, token_type = self.__get_current_token_and_advance()
        self.indentation = self.indentation[:-2]
        self.__write_close_tag(self.STATEMENTS_TAG)

    def __subroutine_call_format(self) -> None:
        """
        this method compile the subroutine call format
        """
        # . -> ?
        token, token_type = self.__get_current_token()
        if token == ".":
            # symbol
            self.__write_open_and_close_tag(token_type, token)
            # identifier
            self.__write_next_advanced_token()
            # ( -> ?
            token, token_type = self.__get_current_token_and_advance()
        # (
        self.__write_open_and_close_tag(token_type, token)
        # expression list
        self.__get_current_token_and_advance()
        self.compile_expression_list()
        # )
        self.__writes_current_token()

    def compile_do(self) -> None:
        """Compiles a do statement."""
        self.__write_open_tag(self.DO_STATEMENT_TAG)
        self.indentation += "  "
        self.output_stream.write("\n")
        # keyword
        self.__writes_current_token()
        # identifier
        self.__write_next_advanced_token()
        # . -> ?
        self.__get_current_token_and_advance()
        # subroutine call
        self.__subroutine_call_format()
        # ;
        self.__write_next_advanced_token()
        self.indentation = self.indentation[:-2]
        self.__write_close_tag(self.DO_STATEMENT_TAG)

    def compile_let(self) -> None:
        """Compiles a let statement."""
        self.__write_open_tag(self.LET_STATEMENT_TAG)
        self.indentation += "  "
        self.output_stream.write("\n")
        # keyword
        self.__writes_current_token()
        # identifier
        token, token_type = self.__get_current_token_and_advance()
        self.__write_open_and_close_tag(self.IDENTIFIER_TAG, token)
        # [ -> ?
        token, token_type = self.__get_current_token_and_advance()
        if token == "[":
            self.__write_open_and_close_tag(token_type, token)
            # expression
            self.__get_current_token_and_advance()
            self.compile_expression()
            # ]
            self.__writes_current_token()
            self.__get_current_token_and_advance()
        # symbol
        self.__writes_current_token()
        # expression
        self.__get_current_token_and_advance()
        self.compile_expression()
        # ;
        self.__writes_current_token()
        self.indentation = self.indentation[:-2]
        self.__write_close_tag(self.LET_STATEMENT_TAG)

    def compile_while(self) -> None:
        """Compiles a while statement."""
        self.__write_open_tag(self.WHILE_STATEMENT_TAG)
        self.indentation += "  "
        self.output_stream.write("\n")
        # keyword
        self.__writes_current_token()
        # (
        self.__write_next_advanced_token()
        # expression
        self.__get_current_token_and_advance()
        self.compile_expression()
        # )
        self.__writes_current_token()
        # {
        # statements
        self.__write_next_advanced_token()
        self.compile_statements()
        # }
        self.__writes_current_token()
        self.indentation = self.indentation[:-2]
        self.__write_close_tag(self.WHILE_STATEMENT_TAG)

    def compile_return(self) -> None:
        """Compiles a return statement."""
        self.__write_open_tag(self.RETURN_STATEMENT_TAG)
        self.indentation += "  "
        self.output_stream.write("\n")
        # keyword
```

```python
        self.__writes_current_token()
        # expression -> ?
        token, token_type = self.__get_current_token_and_advance()
        if token != ";":
            # expression
            self.compile_expression()
            self.__writes_current_token()
        else:
            self.__write_open_and_close_tag(token_type, token)
        self.indentation = self.indentation[:-2]
        self.__write_close_tag(self.RETURN_STATEMENT_TAG)

    def compile_if(self) -> None:
        """Compiles a if statement, possibly with a trailing else clause."""
        self.__write_open_tag(self.IF_STATEMENT_TAG)
        self.indentation += "  "
        self.output_stream.write("\n")
        # keyword
        self.__writes_current_token()
        # (
        self.__write_next_advanced_token()
        # expression
        self.__get_current_token_and_advance()
        self.compile_expression()
        # )
        self.__writes_current_token()
        # {
        self.__write_next_advanced_token()
        # statements
        self.__get_current_token_and_advance()
        self.compile_statements()
        # }
        self.__writes_current_token()
        # else -> ?
        token, token_type = self.__get_current_token_and_advance()
        if token == self.ELSE:
            self.__write_open_and_close_tag(token_type, token)
            # {
            self.__write_next_advanced_token()
            # statements
            self.__get_current_token_and_advance()
            self.compile_statements()
            # }
            self.__writes_current_token()
            self.__get_current_token_and_advance()
        self.indentation = self.indentation[:-2]
        self.__write_close_tag(self.IF_STATEMENT_TAG)

    def compile_expression(self) -> None:
        """Compiles an expression."""
        self.__write_open_tag(self.EXPRESSION_TAG)
        self.indentation += "  "
        self.output_stream.write("\n")
        # term
        self.compile_term()
        # term -> *
        token, token_type = self.__get_current_token()
        while token != ")":
            if token not in self.op_terms:
                break
            # op
            self.__write_open_and_close_tag(token_type, token)
            token, token_type = self.__get_current_token_and_advance()
            # term
            self.compile_term()
            token, token_type = self.__get_current_token()
        self.indentation = self.indentation[:-2]
        self.__write_close_tag(self.EXPRESSION_TAG)

    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.
        """
        self.__write_open_tag(self.TERM_TAG)
        self.indentation += "  "
        self.output_stream.write("\n")
        # identifier / symbol
        token, token_type = self.__get_current_token()
        self.__write_open_and_close_tag(token_type, token)
        # unary term -> ?
        if token in self.unary_op_terms:
            self.__get_current_token_and_advance()
            # term
            self.compile_term()
        # expression - > ?
        elif token == "(":
            # expression
            self.__get_current_token_and_advance()
            self.compile_expression()
            # )
            self.__writes_current_token()
            self.__get_current_token_and_advance()
        else:
            token, token_type = self.__get_current_token_and_advance()
            # [ -> ?
            if token == "[":
                self.__write_open_and_close_tag(token_type, token)
                # expression
                self.__get_current_token_and_advance()
                self.compile_expression()
                # ]
                self.__writes_current_token()
                self.__get_current_token_and_advance()
            # subroutine call -> ?
            elif token in {".", "("}:
                self.__subroutine_call_format()
                self.__get_current_token_and_advance()
        self.indentation = self.indentation[:-2]
        self.__write_close_tag(self.TERM_TAG)

    def compile_expression_list(self) -> None:
        """Compiles a (possibly empty) comma-separated list of expressions."""
        self.__write_open_tag(self.EXPRESSION_LIST_TAG)
        self.indentation += "  "
        self.output_stream.write("\n")
        # expression -> ?
        token, token_type = self.__get_current_token()
        while token != ")":
            # expression
            self.compile_expression()
            token, token_type = self.__get_current_token()
            if token == ",":
                self.__write_open_and_close_tag(token_type, token)
                token, token_type = self.__get_current_token_and_advance()
        self.indentation = self.indentation[:-2]
        self.__write_close_tag(self.EXPRESSION_LIST_TAG)
```

4 JackAnalyzer

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

python3 JackAnalyzer.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 JackAnalyzer.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


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

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

    output_file.close()


if "__main__" == __name__:
    # Parses the input path and calls analyze_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: JackAnalyzer <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 + ".xml"
        with open(input_path, 'r') as input_file, \
                open(output_path, 'w') as output_file:
            analyze_file(input_file, output_file)
```

6 JackTokenizer.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 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
```

```
            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
```

```python
                               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]

    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 'JackAnalyzer <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 JackAnalyzer

# 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/
```