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 JackAnalyzer...
     Found in the correct path.

Testing translation of Main of Square (running on a single file)...
     Testing your JackAnalyzer with command: './JackAnalyzer tst/Square/Main.jack'...
     Diff succeeded on the test.

Testing Square, where Square is a directory...
     Testing your JackAnalyzer with command: './JackAnalyzer Square/'...
     Diff succeeded on the test.

Testing ArrayTest, where ArrayTest is a directory...
     Testing your JackAnalyzer with command: './JackAnalyzer ArrayTest/'...
     Diff succeeded on the test.

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


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

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

2 AUTHORS

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

3 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

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

    def __init__(self, input_stream: "JackTokenizer", output_stream) -> 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.input_stream = input_stream
        self.output_stream = output_stream
        self.input_stream.advance()  # start reading the first token

    def compile_class(self) -> None:
        """Compiles a complete class.
         syntax: 'class' className '{' classVarDec* subroutineDec* '}'."""
        self.output_stream.write("<class>\n")
        self.output_stream.write("<keyword> " + self.input_stream.keyword() + " </keyword>\n")  # class
        self.input_stream.advance()
        self.output_stream.write("<identifier> " + self.input_stream.identifier() + " </identifier>\n")  # className
        self.input_stream.advance()
        self.output_stream.write("<symbol> " + self.input_stream.symbol() + " </symbol>\n")  # {
        self.input_stream.advance()
        while self.input_stream.token_type() == "KEYWORD" and self.input_stream.keyword() in ["static", "field"]:
            self.compile_class_var_dec()
        while self.input_stream.token_type() == "KEYWORD" and self.input_stream.keyword() in ["constructor", "function", "me
            self.compile_subroutine()
        self.output_stream.write("<symbol> " + self.input_stream.symbol() + " </symbol>\n")  # }
        self.output_stream.write("</class>\n")
        self.input_stream.advance()

    def compile_class_var_dec(self) -> None:
        """Compiles a static declaration or a field declaration.
        syntax: ('static' | 'field') type varName (',' varName)* ';'."""
        self.output_stream.write("<classVarDec>\n")
        self.output_stream.write("<keyword> " + self.input_stream.keyword() + " </keyword>\n")  # static | field
        self.input_stream.advance()
        self.write_type()  # type (int | char | boolean | className)
        self.input_stream.advance()
        self.output_stream.write("<identifier> " + self.input_stream.identifier() + " </identifier>\n")  # varName
        self.input_stream.advance()
        while self.input_stream.symbol() == ",":  # (, varName)*
            self.output_stream.write("<symbol> " + self.input_stream.symbol() + " </symbol>\n")  # ,
            self.input_stream.advance()
            self.output_stream.write("<identifier> " + self.input_stream.identifier() + " </identifier>\n")  # varName
            self.input_stream.advance()
        self.output_stream.write("<symbol> " + self.input_stream.symbol() + " </symbol>\n")  # ;
```

```
60          self.output_stream.write("</classVarDec>\n")
61          self.input_stream.advance()
62
63      def compile_subroutine(self) -> None:
64          """
65          Compiles a complete method, function, or constructor.
66          You can assume that classes with constructors have at least one field,
67          you will understand why this is necessary in project 11.
68          syntax: ('constructor' | 'function' | 'method') ('void' | type) subroutineName '(' parameterList ')' subroutineBody
69          """
70          self.output_stream.write("<subroutineDec>\n")
71          self.output_stream.write("<keyword> " + self.input_stream.keyword() + " </keyword>\n")  # constructor | function | m
72          self.input_stream.advance()
73          self.write_type()  # void | type (int | char | boolean | className)
74          self.input_stream.advance()
75          self.output_stream.write("<identifier> " + self.input_stream.identifier() + " </identifier>\n")  # subroutineName
76          self.input_stream.advance()
77          self.output_stream.write("<symbol> " + self.input_stream.symbol() + " </symbol>\n")  # (
78          self.input_stream.advance()
79          self.compile_parameter_list()
80          self.output_stream.write("<symbol> " + self.input_stream.symbol() + " </symbol>\n")  # )
81          self.input_stream.advance()
82          self.output_stream.write("<subroutineBody>\n")
83          self.output_stream.write("<symbol> " + self.input_stream.symbol() + " </symbol>\n")  # {
84          self.input_stream.advance()
85          while self.input_stream.token_type() == "KEYWORD" and self.input_stream.keyword() == "var":
86              self.compile_var_dec()
87          self.compile_statements()
88          self.output_stream.write("<symbol> " + self.input_stream.symbol() + " </symbol>\n")  # }
89          self.output_stream.write("</subroutineBody>\n")
90          self.output_stream.write("</subroutineDec>\n")
91          self.input_stream.advance()
92
93      def compile_parameter_list(self) -> None:
94          """Compiles a (possibly empty) parameter list, not including the enclosing "()".
95          syntax: (type varName (',' type varName)*)? """
96          self.output_stream.write("<parameterList>\n")
97          if self.input_stream.token_type() == "SYMBOL" and self.input_stream.symbol() == ")":
98              self.output_stream.write("</parameterList>\n")
99              return  # empty parameter list
100         self.write_type()  # type (int | char | boolean | className)
101         self.input_stream.advance()
102         self.output_stream.write("<identifier> " + self.input_stream.identifier() + " </identifier>\n")  # varName
103         self.input_stream.advance()
104         while self.input_stream.symbol() == ",":  # (, type varName)*
105             self.output_stream.write("<symbol> " + self.input_stream.symbol() + " </symbol>\n")  # ,
106             self.input_stream.advance()
107             self.write_type()  # type (int | char | boolean | className)
108             self.input_stream.advance()
109             self.output_stream.write("<identifier> " + self.input_stream.identifier() + " </identifier>\n")  # varName
110             self.input_stream.advance()
111         self.output_stream.write("</parameterList>\n")
112
113     def compile_var_dec(self) -> None:
114         """Compiles a var declaration.
115          syntax: 'var' type varName (',' varName)* ';'."""
116         self.output_stream.write("<varDec>\n")
117         self.output_stream.write("<keyword> " + self.input_stream.keyword() + " </keyword>\n")  # var
118         self.input_stream.advance()
119         self.write_type()  # type (int | char | boolean | className)
120         self.input_stream.advance()
121         self.output_stream.write("<identifier> " + self.input_stream.identifier() + " </identifier>\n")  # varName
122         self.input_stream.advance()
123         while self.input_stream.symbol() == ",":
124             self.output_stream.write("<symbol> " + self.input_stream.symbol() + " </symbol>\n")  # ,
125             self.input_stream.advance()
126             self.output_stream.write("<identifier> " + self.input_stream.identifier() + " </identifier>\n")  # varName
127             self.input_stream.advance()
```

```
128             self.output_stream.write("<symbol> " + self.input_stream.symbol() + " </symbol>\n")  # ;
129             self.output_stream.write("</varDec>\n")
130             self.input_stream.advance()
131 
132         def compile_statements(self) -> None:
133             """Compiles a sequence of statements, not including the enclosing {}".
134             syntax: statement*"""
135             self.output_stream.write("<statements>\n")
136             while self.input_stream.token_type() == "KEYWORD" and self.input_stream.keyword() in ["let", "if", "while", "do", "r
137                 if self.input_stream.keyword() == "let":
138                     self.compile_let()
139                 elif self.input_stream.keyword() == "if":
140                     self.compile_if()
141                 elif self.input_stream.keyword() == "while":
142                     self.compile_while()
143                 elif self.input_stream.keyword() == "do":
144                     self.compile_do()
145                 elif self.input_stream.keyword() == "return":
146                     self.compile_return()
147             self.output_stream.write("</statements>\n")
148 
149         def compile_do(self) -> None:
150             """Compiles a do statement.
151              syntax: 'do' subroutineCall ';'."""
152             self.output_stream.write("<doStatement>\n")
153             self.output_stream.write("<keyword> " + self.input_stream.keyword() + " </keyword>\n")  # do
154             self.input_stream.advance()
155             self.output_stream.write("<identifier> " + self.input_stream.identifier() + " </identifier>\n")  # subroutineName |
156             self.input_stream.advance()
157             self.subroutine_call()  # subroutineCall
158             self.output_stream.write("<symbol> " + self.input_stream.symbol() + " </symbol>\n")  # ;
159             self.output_stream.write("</doStatement>\n")
160             self.input_stream.advance()
161 
162         def compile_let(self) -> None:
163             """Compiles a let statement.
164              syntax: 'let' varName ('[' expression ']')? '=' expression ';'."""
165             self.output_stream.write("<letStatement>\n")
166             self.output_stream.write("<keyword> " + self.input_stream.keyword() + " </keyword>\n")  # let
167             self.input_stream.advance()
168             self.output_stream.write("<identifier> " + self.input_stream.identifier() + " </identifier>\n")  # varName
169             self.input_stream.advance()
170             if self.input_stream.symbol() == "[":
171                 self.output_stream.write("<symbol> " + self.input_stream.symbol() + " </symbol>\n")  # [
172                 self.input_stream.advance()
173                 self.compile_expression()  # expression
174                 self.output_stream.write("<symbol> " + self.input_stream.symbol() + " </symbol>\n")  # ]
175                 self.input_stream.advance()
176             self.output_stream.write("<symbol> " + self.input_stream.symbol() + " </symbol>\n")  # =
177             self.input_stream.advance()
178             self.compile_expression()  # expression
179             self.output_stream.write("<symbol> " + self.input_stream.symbol() + " </symbol>\n")  # ;
180             self.output_stream.write("</letStatement>\n")
181             self.input_stream.advance()
182 
183         def compile_while(self) -> None:
184             """Compiles a while statement.
185              syntax: 'while' '(' 'expression' ')' '{' statements '}'."""
186             self.output_stream.write("<whileStatement>\n")
187             self.output_stream.write("<keyword> " + self.input_stream.keyword() + " </keyword>\n")  # while
188             self.input_stream.advance()
189             self.output_stream.write("<symbol> " + self.input_stream.symbol() + " </symbol>\n")  # (
190             self.input_stream.advance()
191             self.compile_expression()  # expression
192             self.output_stream.write("<symbol> " + self.input_stream.symbol() + " </symbol>\n")  # )
193             self.input_stream.advance()
194             self.output_stream.write("<symbol> " + self.input_stream.symbol() + " </symbol>\n")  # {
195             self.input_stream.advance()
```

```
196             self.compile_statements()  # statements
197             self.output_stream.write("<symbol> " + self.input_stream.symbol() + " </symbol>\n")  # }
198             self.output_stream.write("</whileStatement>\n")
199             self.input_stream.advance()
200 
201         def compile_return(self) -> None:
202             """Compiles a return statement.
203             syntax: 'return' expression? ';'"""
204             self.output_stream.write("<returnStatement>\n")
205             self.output_stream.write("<keyword> " + self.input_stream.keyword() + " </keyword>\n")  # return
206             self.input_stream.advance()
207             if self.input_stream.symbol() != ";":
208                 self.compile_expression()  # expression
209             self.output_stream.write("<symbol> " + self.input_stream.symbol() + " </symbol>\n")  # ;
210             self.output_stream.write("</returnStatement>\n")
211             self.input_stream.advance()
212 
213         def compile_if(self) -> None:
214             """Compiles a if statement, possibly with a trailing else clause.
215             syntax: 'if' '(' expression ')' '{' statements '}' ('else' '{' statements '}')?"""
216             self.output_stream.write("<ifStatement>\n")
217             self.output_stream.write("<keyword> " + self.input_stream.keyword() + " </keyword>\n")  # if
218             self.input_stream.advance()
219             self.output_stream.write("<symbol> " + self.input_stream.symbol() + " </symbol>\n")  # (
220             self.input_stream.advance()
221             self.compile_expression()  # expression
222             self.output_stream.write("<symbol> " + self.input_stream.symbol() + " </symbol>\n")  # )
223             self.input_stream.advance()
224             self.output_stream.write("<symbol> " + self.input_stream.symbol() + " </symbol>\n")  # {
225             self.input_stream.advance()
226             self.compile_statements()  # statements
227             self.output_stream.write("<symbol> " + self.input_stream.symbol() + " </symbol>\n")  # }
228             self.input_stream.advance()
229             if self.input_stream.token_type() == "KEYWORD" and self.input_stream.keyword() == "else":
230                 self.output_stream.write("<keyword> " + self.input_stream.keyword() + " </keyword>\n")  # else
231                 self.input_stream.advance()
232                 self.output_stream.write("<symbol> " + self.input_stream.symbol() + " </symbol>\n")  # {
233                 self.input_stream.advance()
234                 self.compile_statements()  # statements
235                 self.output_stream.write("<symbol> " + self.input_stream.symbol() + " </symbol>\n")  # }
236                 self.input_stream.advance()
237             self.output_stream.write("</ifStatement>\n")
238 
239         def compile_expression(self) -> None:
240             """Compiles an expression.
241             syntax: term (op term)*"""
242             self.output_stream.write("<expression>\n")
243             self.compile_term()  # term
244             while self.input_stream.token_type() == "SYMBOL" and self.input_stream.symbol() in ["+", "-", "*", "/", "&amp;", "|"
245                 self.output_stream.write("<symbol> " + self.input_stream.symbol() + " </symbol>\n")  # op
246                 self.input_stream.advance()
247                 self.compile_term()  # term
248             self.output_stream.write("</expression>\n")
249 
250         def compile_term(self) -> None:
251             """Compiles a term.
252             This routine is faced with a slight difficulty when
253             trying to decide between some of the alternative parsing rules.
254             Specifically, if the current token is an identifier, the routing must
255             distinguish between a variable, an array entry, and a subroutine call.
256             A single look-ahead token, which may be one of "[", "(", or "." suffices
257             to distinguish between the three possibilities. Any other token is not
258             part of this term and should not be advanced over.
259             syntax: integerConstant | stringConstant | keywordConstant | varName | varName '[' expression ']' |
260              subroutineCall | '(' expression ')' | unaryOp term
261             """
262             self.output_stream.write("<term>\n")
263             if self.input_stream.token_type() == "INT_CONST":
```

```
            self.output_stream.write("<integerConstant> " + str(self.input_stream.int_val()) + " </integerConstant>\n")  # i
            self.input_stream.advance()
        elif self.input_stream.token_type() == "STRING_CONST":
            self.output_stream.write("<stringConstant> " + self.input_stream.string_val() + " </stringConstant>\n")  # strin
            self.input_stream.advance()
        elif self.input_stream.token_type() == "KEYWORD" and self.input_stream.keyword() in ["true", "false", "null", "this"
            self.output_stream.write("<keyword> " + self.input_stream.keyword() + " </keyword>\n")  # keywordConstant
            self.input_stream.advance()
        elif self.input_stream.symbol() == "(":  # '(' expression ')'
            self.output_stream.write("<symbol> " + self.input_stream.symbol() + " </symbol>\n")  # (
            self.input_stream.advance()
            self.compile_expression()  # expression
            self.output_stream.write("<symbol> " + self.input_stream.symbol() + " </symbol>\n")  # )
            self.input_stream.advance()
        elif self.input_stream.token_type() == "SYMBOL" and self.input_stream.symbol() in ["-", "~"]:  # unaryOp term
            self.output_stream.write("<symbol> " + self.input_stream.symbol() + " </symbol>\n")  # unaryOp
            self.input_stream.advance()
            self.compile_term()  # term

        else:  # identifier, one of three options: varName | varName '[' expression ']' | subroutineCall
            self.output_stream.write("<identifier> " + self.input_stream.identifier() + " </identifier>\n")  # varName | sub
            self.input_stream.advance()
            if self.input_stream.symbol() == "[":  # varName '[' expression ']'
                self.output_stream.write("<symbol> " + self.input_stream.symbol() + " </symbol>\n")  # [
                self.input_stream.advance()
                self.compile_expression()  # expression
                self.output_stream.write("<symbol> " + self.input_stream.symbol() + " </symbol>\n")  # ]
                self.input_stream.advance()
            elif self.input_stream.symbol() in ["(", "."]:  # subroutineCall
                self.subroutine_call()
            # else: varName (already written)
        self.output_stream.write("</term>\n")

    def compile_expression_list(self) -> None:
        """Compiles a (possibly empty) comma-separated list of expressions."""
        self.output_stream.write("<expressionList>\n")
        if self.input_stream.token_type() != "SYMBOL" or self.input_stream.symbol() != ")":
            self.compile_expression()
            while self.input_stream.symbol() == ",":
                self.output_stream.write("<symbol> " + self.input_stream.symbol() + " </symbol>\n")  # ,
                self.input_stream.advance()
                self.compile_expression()
        self.output_stream.write("</expressionList>\n")

    def write_type(self) -> None:  # added
        """Writes the type of the current token to the output stream."""
        if self.input_stream.token_type() == "KEYWORD":
            self.output_stream.write("<keyword> " + self.input_stream.keyword() + " </keyword>\n")  # int | char | boolean
        else:
            self.output_stream.write("<identifier> " + self.input_stream.identifier() + " </identifier>\n")  # className

    def 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 ')'"""
        while self.input_stream.symbol() == ".":  # used while instead of if to allow chaining of subroutine calls
            self.output_stream.write("<symbol> " + self.input_stream.symbol() + " </symbol>\n")  # .
            self.input_stream.advance()
            self.output_stream.write("<identifier> " + self.input_stream.identifier() + " </identifier>\n")  # subroutineNam
            self.input_stream.advance()
        self.output_stream.write("<symbol> " + self.input_stream.symbol() + " </symbol>\n")  # (
        self.input_stream.advance()
        self.compile_expression_list()  # expressionList
        self.output_stream.write("<symbol> " + self.input_stream.symbol() + " </symbol>\n")  # )
        self.input_stream.advance()
```

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

```
"""
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()

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:
            print(f"Analyzing {input_path}...")
            analyze_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 parse_line(self) -> None:  # added
        """Parses the current line into tokens and saves them in a list.
        """
        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*"

        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:
        """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) -> str:
        """
        Returns:
            str: the type of the current token, can be
```

```
            "KEYWORD", "SYMBOL", "IDENTIFIER", "INT_CONST", "STRING_CONST"
        """
        if self.cur_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 self.cur_token in ['{', '}', '(', ')', '[', ']', '.', ',', ';', '+',
                              '-', '*', '/', '&', '|', '<', '>', '=', '~', '^', '#']:
            return "SYMBOL"
        if self.cur_token.isdigit():
            return "INT_CONST"
        if  self.cur_token.startswith('"') and self.cur_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:
        """
```

```
264             Returns:
265                 str: the string value of the current token, without the double
266                 quotes. Should be called only when token_type() is "STRING_CONST".
267                 Recall that StringConstant was defined in the grammar like so:
268                 StringConstant: '"' A sequence of Unicode characters not including
269                           double quote or newline '"'
270             """
271             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 '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/
```