Contents

1 Basic Test Results

```
********** TESTING FOLDER STRUCTURE START **********
Running test10.sh:
Checking your submission for presence of invalid (non-ASCII) characters...
No invalid characters found.

Your logins are: muaz.abdeen, is that ok?

********** TESTING FOLDER STRUCTURE END **********

********** PROJECT TEST START **********

Running: 'make'
chmod a+x JackAnalyzer
Running your program with command: 'JackAnalyzer test/ArrayTest'
ArrayTest XMLs created
The diff is OK on the file test/ArrayTest/Main.xml
Running your program with command: 'JackAnalyzer test/ArrayTest/Main.jack'
ArrayTest XMLs created
The diff is OK on the file test/ArrayTest/Main.xml

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

2 README

```
muaz.abdeen
================================================================================
Muaz Abdeen, ID 300575297, muaz.abdeen@mail.huji.ac.il
================================================================================

                           Project 10 - Compiler I: Syntax Analysis
                           ----------------------------------------


Submitted Files
---------------
(1)  README                 - This file.
(2)  JackTokenizer.py        - The JackTokenizer module implementation.
(3)  CompilationEngine.py   - The CompilationEngine module implementation.
(4)  JackAnalyzer.py        - The JackAnalyzer module implementation.
(5)  JackAnalyzer            - The run file.
(6)  Makefile                - The make file.


Remarks
-------
* ...
```

3 CompilationEngine.py

```
1   ########################################################
2   ##   FILE : CompilationEngine.py                      ##
3   ##   WRITER : muaz.abdeen, 300575297                  ##
4   ##   EXERCISE : nand2tetris projects10 2020A          ##
5   ########################################################
6
7   import JackTokenizer
8
9
10  class CompilationEngine:
11      """
12      Generates the compiler's output.
13      """
14
15      ############################
16      ##  MACROS & CONSTANTS  ##
17      ############################
18
19      AFTER = 0
20      BEFORE = 1
21
22      CLASS_VARIABLES = {"field", "static"}
23      SUBROUTINES = {"constructor", "function", "method"}
24      ROUTINE_VARIABLES = {"int", "char", "boolean"}
25      STATEMENTS = {"let", "do", "if", "while", "return"}
26      OPERATORS = {'+', '-', '*', '/', '&', '|', '<', '>', '='}
27      UNARY_OPERATORS = {'-', '~'}
28
29      SPECIAL_SYMBOLS = {'<': '&lt;', '>': '&gt;', '"': '&quot;', "&": '&amp;'}
30
31      ####################
32      ##  CONSTRUCTOR  ##
33      ####################
34
35      def __init__(self, input_file, output_file):
36          """
37          Creates a new compilation engine with the given input and output.
38          The next routine called must be compileClass()
39          :param input_file: The input file (.jack)
40          :type input_file: str
41          :param output_file: The output file (.xml)
42          :type output_file: str
43          """
44          self.tokenizer = JackTokenizer.JackTokenizer(input_file)
45          self.outputFile = open(output_file, mode='w')
46          self.nesting_counter = 0
47          # End of the Constructor
48
49      ################
50      ##  METHODS  ##
51      ################
52
53      def _writeRuleTags(self, token_type, place):
54          """
55          Writes the enclosing tags of a (non terminal) rule
56          :param token_type: type of the (token) rule
57          :type token_type: str
58          :param place: AFTER(= 0), or BEFORE(= 1)
59          :type place: int
```

```
60          :return: None
61          """
62          if place == self.BEFORE:
63              indentation = self.nesting_counter * '  '
64              self.outputFile.write(f'{indentation}<{token_type}>\n')
65              self.nesting_counter += 1
66          elif place == self.AFTER:
67              self.nesting_counter -= 1
68              indentation = self.nesting_counter * '  '
69              self.outputFile.write(f'{indentation}</{token_type}>\n')
70
71      def _parseTerminalRule(self, token_type):
72          token_value = self.tokenizer.tokenValue()
73          if token_value in self.SPECIAL_SYMBOLS.keys():
74              token_value = self.SPECIAL_SYMBOLS[token_value]
75          indentation = self.nesting_counter * '  '
76          self.outputFile.write(f'{indentation}<{token_type}> {token_value} </{token_type}>\n')
77          self.tokenizer.advance()
78
79      def CompileClass(self):
80          """
81          Compiles a complete class.
82          'class' className '{' classVarDec* subroutineDec* '}'
83          :return: None
84          """
85          token_type = self.tokenizer.tokenType()
86          token_value = self.tokenizer.tokenValue()
87          assert token_type == self.tokenizer.KEYWORD
88          assert token_value == 'class'
89          # <class>
90          self._writeRuleTags('class', self.BEFORE)
91          while self.tokenizer.current_token != '{':
92              self._parseTerminalRule(self.tokenizer.tokenType())
93          # classVarDec*
94          while self.tokenizer.tokenValue() in self.CLASS_VARIABLES:
95              self.CompileClassVarDec()
96          # subroutineDec*
97          while self.tokenizer.tokenValue() in self.SUBROUTINES:
98              self.CompileSubroutine()
99          # <symbol>}</symbol>
100         self._parseTerminalRule(self.tokenizer.tokenType())
101         # </class>
102         self._writeRuleTags('class', self.AFTER)
103         self.outputFile.close()
104         # End of CompileClass() method
105
106     def CompileClassVarDec(self):
107         """
108         Compiles a static declaration or a field declaration.
109         ('static' | 'field') type varName (',' varName)* ';'
110         :return: None
111         """
112         assert self.tokenizer.tokenValue() in self.CLASS_VARIABLES
113         # <classVarDec>
114         self._writeRuleTags('classVarDec', self.BEFORE)
115         # <keyword> 'static' | 'field' </keyword>
116         self._parseTerminalRule(self.tokenizer.tokenType())
117         # <type> XXX </type> : symbol or identifier
118         self._parseTerminalRule(self.tokenizer.tokenType())
119         # <identifier> varName </identifier>
120         while self.tokenizer.current_token != ';':
121             self._parseTerminalRule(self.tokenizer.tokenType())
122         # </classVarDec>
123         self._writeRuleTags('classVarDec', self.AFTER)
124         # End of CompileClassVarDec() method
125
126     def CompileSubroutine(self):
127         """
```

```
        Compiles a complete method, function, or constructor.
        ('constructor' | 'function' | 'method')
        ('void' | type) subroutineName '(' parameterList ')'
        subroutineBody
        :return: None
        """
        assert self.tokenizer.tokenValue() in self.SUBROUTINES
        # <subroutineDec>
        self._writeRuleTags('subroutineDec', self.BEFORE)
        # parse until (
        while self.tokenizer.tokenValue() != '(':
            self._parseTerminalRule(self.tokenizer.tokenType())
        # <symbol>(</symbol>
        self._parseTerminalRule(self.tokenizer.tokenType())
        # <parameterList>
        self._writeRuleTags('parameterList', self.BEFORE)
        if self.tokenizer.tokenValue() != ')':
            self.compileParameterList()
        # </parameterList>
        self._writeRuleTags('parameterList', self.AFTER)
        # <symbol>)</symbol>
        self._parseTerminalRule(self.tokenizer.tokenType())
        # <subroutineBody> '{' varDec* statements '}'
        self._writeRuleTags('subroutineBody', self.BEFORE)
        self._parseTerminalRule(self.tokenizer.tokenType())
        # varDec*
        while self.tokenizer.tokenValue() == 'var':
            self.compileVarDec()
        if self.tokenizer.tokenValue() in self.STATEMENTS:
            self.compileStatements()
        self._parseTerminalRule(self.tokenizer.tokenType())
        # </subroutineBody>
        self._writeRuleTags('subroutineBody', self.AFTER)
        # </subroutineDec>
        self._writeRuleTags('subroutineDec', self.AFTER)
        # End of CompileSubroutine() method

    def compileParameterList(self):
        """
        Compiles a (possibly empty) parameter list, not including the enclosing ().
        :return: None
        """
        assert (self.tokenizer.tokenValue() in self.ROUTINE_VARIABLES) or \
               self.tokenizer.tokenType() == self.tokenizer.IDENTIFIER
        # ((type varName) (',' type varName)*)?
        while self.tokenizer.tokenValue() != ')':
            self._parseTerminalRule(self.tokenizer.tokenType())
        # End of compileParameterList() method

    def compileVarDec(self):
        """
        Compiles a var declaration.
        'var' type varName (',' varName)* ';'
        :return: None
        """
        assert self.tokenizer.tokenValue() == 'var'
        # <varDec>
        self._writeRuleTags('varDec', self.BEFORE)
        while self.tokenizer.current_token != ';':
            self._parseTerminalRule(self.tokenizer.tokenType())
        # </varDec>
        self._writeRuleTags('varDec', self.AFTER)
        # End of compileVarDec() method

    def compileStatements(self):
        """
        Compiles a sequence of statements, not including the enclosing {}.
        :return: None
```

```
196             """
197             # assert self.tokenizer.tokenValue() in self.STATEMENTS
198             # <statements>
199             self._writeRuleTags('statements', self.BEFORE)
200             while self.tokenizer.tokenValue() in self.STATEMENTS:
201                 if self.tokenizer.tokenValue() == 'let':
202                     self.compileLet()
203                 elif self.tokenizer.tokenValue() == 'do':
204                     self.compileDo()
205                 elif self.tokenizer.tokenValue() == 'if':
206                     self.compileIf()
207                 elif self.tokenizer.tokenValue() == 'while':
208                     self.compileWhile()
209                 elif self.tokenizer.tokenValue() == 'return':
210                     self.compileReturn()
211             # <statements>
212             self._writeRuleTags('statements', self.AFTER)
213             # End of compileStatements() method
214 
215         def compileDo(self):
216             """
217             Compiles a do statement.
218             'do' subroutineCall ';'
219             subroutineCall : subroutineName '(' expressionList ')' |
220                              (className | varName) '.' subroutineName '(' expressionList ')'
221             :return: None
222             """
223             assert self.tokenizer.tokenValue() == 'do'
224             # <doStatement>
225             self._writeRuleTags('doStatement', self.BEFORE)
226             # <keyword> do </keyword>
227             self._parseTerminalRule(self.tokenizer.tokenType())
228             # subroutineCall
229             while self.tokenizer.current_token != '(':
230                 self._parseTerminalRule(self.tokenizer.tokenType())
231             # expressionList
232             self.CompileExpressionList()
233             while self.tokenizer.current_token != ';':
234                 self._parseTerminalRule(self.tokenizer.tokenType())
235             # </doStatement>
236             self._writeRuleTags('doStatement', self.AFTER)
237             # End of compileDo() method
238 
239         def compileLet(self):
240             """
241             Compiles a let statement.
242             'let' varName ('[' expression ']')? '=' expression ';'
243             :return: None
244             """
245             assert self.tokenizer.tokenValue() == 'let'
246             # <letStatement>
247             self._writeRuleTags('letStatement', self.BEFORE)
248 
249             self._parseTerminalRule(self.tokenizer.tokenType())     # let
250             self._parseTerminalRule(self.tokenizer.tokenType())     # varName
251             if self.tokenizer.tokenValue() == '[':
252                 self._parseTerminalRule(self.tokenizer.tokenType())  # [
253                 self.CompileExpression()
254                 self._parseTerminalRule(self.tokenizer.tokenType())  # ]
255             self._parseTerminalRule(self.tokenizer.tokenType())   # =
256             self.CompileExpression()
257             # assert self.tokenizer.tokenValue() == ';'
258             self._parseTerminalRule(self.tokenizer.tokenType())  # ;
259 
260             # </letStatement>
261             self._writeRuleTags('letStatement', self.AFTER)
262             # End of compileLet() method
263 
```

```
264     def compileWhile(self):
265         """
266         Compiles a while statement.
267         'while' '(' expression ')' '{' statements '}'
268         :return: None
269         """
270         assert self.tokenizer.tokenValue() == 'while'
271         # <whileStatement>
272         self._writeRuleTags('whileStatement', self.BEFORE)
273 
274         self._parseTerminalRule(self.tokenizer.tokenType())  # while
275         self._parseTerminalRule(self.tokenizer.tokenType())  # (
276         self.CompileExpression()
277         self._parseTerminalRule(self.tokenizer.tokenType())  # )
278         self._parseTerminalRule(self.tokenizer.tokenType())  # {
279         self.compileStatements()
280         self._parseTerminalRule(self.tokenizer.tokenType())  # }
281 
282         # </whileStatement>
283         self._writeRuleTags('whileStatement', self.AFTER)
284         # End of compileWhile() method
285 
286     def compileReturn(self):
287         """
288         Compiles a return statement.
289         'return' expression? ';'
290         :return: None
291         """
292         assert self.tokenizer.tokenValue() == 'return'
293         # <returnStatement>
294         self._writeRuleTags('returnStatement', self.BEFORE)
295 
296         self._parseTerminalRule(self.tokenizer.tokenType())  # return
297         if self.tokenizer.tokenValue() != ';':
298             self.CompileExpression()
299         self._parseTerminalRule(self.tokenizer.tokenType())  # ;
300 
301         # </returnStatement>
302         self._writeRuleTags('returnStatement', self.AFTER)
303         # End of compileReturn() method
304 
305     def compileIf(self):
306         """
307         Compiles a if statement possibly with a trailing else clause.
308         'if' '(' expression ')' '{' statements '}' ('else' '{' statements '}')?
309         :return: None
310         """
311         assert self.tokenizer.tokenValue() == 'if'
312         # <ifStatement>
313         self._writeRuleTags('ifStatement', self.BEFORE)
314 
315         self._parseTerminalRule(self.tokenizer.tokenType())  # if
316         self._parseTerminalRule(self.tokenizer.tokenType())  # (
317         self.CompileExpression()
318         self._parseTerminalRule(self.tokenizer.tokenType())  # )
319         self._parseTerminalRule(self.tokenizer.tokenType())  # {
320         self.compileStatements()
321         self._parseTerminalRule(self.tokenizer.tokenType())  # }
322         if self.tokenizer.tokenValue() == 'else':
323             self._parseTerminalRule(self.tokenizer.tokenType())  # else
324             self._parseTerminalRule(self.tokenizer.tokenType())  # {
325             self.compileStatements()
326             self._parseTerminalRule(self.tokenizer.tokenType())  # }
327 
328         # </ifStatement>
329         self._writeRuleTags('ifStatement', self.AFTER)
330         # End of compileIf() method
331 
```

```
    def CompileExpression(self):
        """
        Compiles an expression.
        term (op term)*
        :return: None
        """
        # <expression>
        self._writeRuleTags('expression', self.BEFORE)
        self.CompileTerm()
        while self.tokenizer.tokenValue() in self.OPERATORS:
            self._parseTerminalRule(self.tokenizer.tokenType())  # op
            self.CompileTerm()
        # </expression>
        self._writeRuleTags('expression', self.AFTER)
        # End of CompileExpression() method

    def CompileTerm(self):
        """
        Compiles a term.
        integerConstant | stringConstant | keywordConstant | varName |
        varName '[' expression ']' | subroutineCall | '(' expression ')' | unaryOp term
        :return: None
        """
        # assert (self.tokenizer.tokenType() in [self.tokenizer.KEYWORD, self.tokenizer.INT_CONST,
        #                                        self.tokenizer.STRING_CONST]) or \
        #        (self.tokenizer.tokenType() == self.tokenizer.IDENTIFIER) or \
        #        (self.tokenizer.tokenType() == self.tokenizer.SYMBOL)
        # <term>
        self._writeRuleTags('term', self.BEFORE)

        # (1) integerConstant | stringConstant | keywordConstant
        if self.tokenizer.tokenType() in [self.tokenizer.KEYWORD, self.tokenizer.INT_CONST,
                                          self.tokenizer.STRING_CONST]:
            self._parseTerminalRule(self.tokenizer.tokenType())
        # (2) varName | varName '[' expression ']' | subroutineCall
        elif self.tokenizer.tokenType() == self.tokenizer.IDENTIFIER:
            self._parseTerminalRule(self.tokenizer.tokenType())  # varName | subroutineName
            # assert (self.tokenizer.tokenValue() == '[') or\
            #        (self.tokenizer.tokenValue() == '.')
            if self.tokenizer.tokenValue() == '[':
                self._parseTerminalRule(self.tokenizer.tokenType())  # [
                self.CompileExpression()
                self._parseTerminalRule(self.tokenizer.tokenType())  # ]
            elif self.tokenizer.tokenValue() == '.':   # subroutineCall
                while self.tokenizer.tokenValue() != '(':
                    self._parseTerminalRule(self.tokenizer.tokenType())
            if self.tokenizer.tokenValue() == '(':
                self._parseTerminalRule(self.tokenizer.tokenType())  # (
                self.CompileExpressionList()    # expressionList
                self._parseTerminalRule(self.tokenizer.tokenType())  # )
        # (3) '(' expression ')' | unaryOp term
        elif self.tokenizer.tokenType() == self.tokenizer.SYMBOL:
            assert (self.tokenizer.tokenValue() == '(') or \
                   (self.tokenizer.tokenValue() in self.UNARY_OPERATORS)
            if self.tokenizer.tokenValue() == '(':
                self._parseTerminalRule(self.tokenizer.tokenType())  # (
                self.CompileExpression()
                self._parseTerminalRule(self.tokenizer.tokenType())  # )
            elif self.tokenizer.tokenValue() in self.UNARY_OPERATORS:
                self._parseTerminalRule(self.tokenizer.tokenType())  # unaryOp
                self.CompileTerm()

        # </term>
        self._writeRuleTags('term', self.AFTER)
        # End of CompileTerm() method

    def CompileExpressionList(self):
        """
```

```
400             Compiles a (possibly empty) comma-separated list of expressions.
401             (expression (',' expression)* )?
402             :return: None
403             """
404             # <expressionList>
405             self._writeRuleTags('expressionList', self.BEFORE)
406             if self.tokenizer.tokenValue() != ')':
407                 self.CompileExpression()
408                 while self.tokenizer.tokenValue() == ',':
409                     self._parseTerminalRule(self.tokenizer.tokenType())  # ,
410                     self.CompileExpression()
411             # </expressionList>
412             self._writeRuleTags('expressionList', self.AFTER)
413             # End of CompileExpressionList() method
414 
415         # Enf of JackAnalyzer class
```

4 JackAnalyzer

```
1  #!/bin/sh
2  python3 JackAnalyzer.py $*
```

5 JackAnalyzer.py

```
##########################################################
##   FILE : JackAnalyzer.py                             ##
##   WRITER : muaz.abdeen, 300575297                    ##
##   EXERCISE : nand2tetris projects10 2020A            ##
##########################################################

import sys
import os
import CompilationEngine


def main():
    if len(sys.argv) != 2:
        print("Usage: JackAnalyzer <file.jack or path>")
        sys.exit(-1)

    program_input = sys.argv[1]
    if os.path.isdir(program_input):
        for entry in os.scandir(program_input):
            if entry.is_file() and entry.name.endswith('.jack'):
                output_file = entry.name.replace('jack', 'xml')
                input_path = os.path.join(os.path.abspath(program_input), entry.name)
                output_path = os.path.join(os.path.abspath(program_input), output_file)
                jack_compiler = CompilationEngine.CompilationEngine(input_path, output_path)
                jack_compiler.tokenizer.advance()
                jack_compiler.CompileClass()

    elif program_input.endswith('.jack'):
        jack_compiler = CompilationEngine.CompilationEngine(program_input,
                                                            program_input.replace('jack', 'xml'))
        jack_compiler.tokenizer.advance()
        jack_compiler.CompileClass()


if __name__ == '__main__':
    main()
```

6 JackTokenizer.py

```
1  ##########################################################
2  ##   FILE : JackTokenizer.py                            ##
3  ##   WRITER : muaz.abdeen, 300575297                    ##
4  ##   EXERCISE : nand2tetris projects10 2020A            ##
5  ##########################################################
6
7  import re
8
9
10 class JackTokenizer:
11     """
12     Removes all comments and white space from the input stream and breaks
13     it into Jack-language tokens, as specified by the Jack grammar.
14     """
15
16     ############################
17     ##  MACROS & CONSTANTS  ##
18     ############################
19
20     NOT_FOUND = -1
21
22     KEYWORD = 'keyword'
23     SYMBOL = 'symbol'
24     INT_CONST = 'integerConstant'
25     STRING_CONST = 'stringConstant'
26     IDENTIFIER = 'identifier'
27
28     TOKENS_TYPES = {KEYWORD: r'\b(class|constructor|function|method|'
29                              r'field|static|var|'
30                              r'int|char|boolean|void|'
31                              r'true|false|null|'
32                              r'this|'
33                              r'let|do|if|else|while|return)\b',
34                     SYMBOL: r'([{}()[\]\.,;+\-\*/&|<>=~])',
35                     INT_CONST: r'([0-9]+)',
36                     STRING_CONST: r'"(.*?)"',
37                     IDENTIFIER: r'([a-zA-Z_][a-zA-Z_0-9]*)'}
38
39     ####################
40     ##  CONSTRUCTOR  ##
41     ####################
42
43     def __init__(self, file_name):
44         """
45         Opens the input file/stream and gets ready to tokenize it.
46         :param file_name: name of the file to parse (.jack) file
47         :type file_name: str
48         """
49         self.file = open(file_name, mode='r')
50         self.tokens = ''
51         self.inComment = False
52         self.token_type = ''
53         self.current_token = ''
54         # End of the Constructor
55
56     ################
57     ##  METHODS  ##
58     ################
59
```

```
60      def hasMoreTokens(self):
61          """
62          Do we have more tokens in the input?
63          :return: True if there are more tokens, False else
64          :rtype: bool
65          """
66          return (self.file is not None) or self.tokens
67
68      def advance(self):
69          """
70          Gets the next token from the input and makes it the current token.
71          This method should only be called if hasMoreTokens() is true.
72          Initially there is no current token.
73          :return: None
74          """
75          if self.tokens and not self.inComment:
76              self.tokens = self.tokens[len(self.current_token)+2:] \
77                            if self.token_type == self.STRING_CONST \
78                            else self.tokens[len(self.current_token):]
79          else:   # tokes empty or inComment
80              self.tokens = self.file.readline()
81              if not self.tokens:
82                  self.file.close()
83                  self.file = None
84                  return
85          # deals with comments and white spaces
86          self._removeComments()
87          if not self.tokens or self.inComment:
88              self.advance()
89          # End of advance() method
90
91      def _removeComments(self):
92          """
93          Removes all comments in current line
94          :return: None
95          """
96          self.tokens = self.tokens.lstrip()
97          # (1) inline comments:
98          comment1 = re.match(r'//.*?\n|/\*.*?\*/', self.tokens, re.DOTALL)
99          if comment1 and not self.inComment:
100             self.tokens = self.tokens[comment1.span()[1]:]
101             self._removeComments()
102         # (2) prefix multiline:
103         comment2 = re.match(r'/\*.*?\n', self.tokens, re.DOTALL)
104         if comment2 and not self.inComment:
105             self.inComment = True
106             # self.tokens = self.tokens[comment2.span()[1]:]
107             return
108         # (3) suffix multiline:
109         comment3 = re.match(r'.*?\*/', self.tokens, re.DOTALL)
110         if comment3 and self.inComment:
111             self.inComment = False
112             self.tokens = self.tokens[comment3.span()[1]:]
113             self._removeComments()
114         elif not comment3 and self.inComment:
115             return
116
117     def tokenType(self):
118         """
119         Returns the type of the current token.
120         (KEYWORD, SYMBOL, IDENTIFIER, INT_CONST, STRING_CONST)
121         :return: the current token type
122         :rtype: str
123         """
124         if not self.tokens:
125             return
126         for token_type, syntax in self.TOKENS_TYPES.items():
127             if re.match(syntax, self.tokens):
```

```
                self.token_type = token_type
                return token_type

    def tokenValue(self):
        """
        Returns the value of the current token.
        :return: returns the value of the current token.
        :rtype: str, int
        """
        self.current_token = re.match(self.TOKENS_TYPES[self.tokenType()], self.tokens).group(1)
        if self.token_type == self.INT_CONST:
            return int(self.current_token)
        return self.current_token

    # Enf of JackAnalyzer class
```

7 Makefile

```
1  ################################################################################
2  #
3  # Makefile for a script (e.g. Python), project 10
4  #
5  ################################################################################
6
7  all:
8      chmod a+x JackAnalyzer
```