Contents

1 Basic Test Results

```
********** TEST START **********

  preparing sub.tar
dos2unix: converting file /tmp/bodek.kgzgqf/nand2tet/Project10/roigreenberg/presubmission/testdir/stud/sub.tar/README to Uni
  checking sub.tar
chmod a+x JackAnalyzer
exec ./JackAnalyzer test/ArrayTest
exec ./JackAnalyzer test/Square
java: no process found
python: no process found

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

2 README

```
roigreenberg,inbaravni

================================================================================

Roi Greenberg, ID 30557123, roi.greenberg@mail.huji.ac.il
Inbar Avni, ID 201131760, inbar.avni@mail.huji.ac.il

================================================================================



             Project 10 - Compiler I - Syntax Analysis

             ----------------------------------------




Submitted Files

---------------

README - This file.

JackAnalyzer.py - the main.
JackTokenizer.py - parse the file
CompileEngine.py - write the XML code
JackAnalyzer - An executable.
Makefile - a makefile.

Remarks

-------

* No remarks for that time.
```

3 CompilationEngine.py

```
from JackTokenizer import *

__author__ = 'roigreenberg'


class Structure:
    CLASS = 'class'
    CLASSVARDEC = 'classVarDec'
    TYPE = 'type'
    SUBROUTINEDEC = 'subroutineDec'
    PARAM_LIST = 'parameterList'
    SUBROUTINEBODY = 'subroutineBody'
    VAR_DEC = 'varDec'
    STATEMENTS = 'statements'
    STATEMENT_LET = 'letStatement'
    STATEMENT_RETURN = 'returnStatement'
    STATEMENT_DO = 'doStatement'
    STATEMENT_IF = 'ifStatement'
    STATEMENT_WHILE = 'whileStatement'
    EXPRESSION_LIST = 'expressionList'
    EXPRESSION = 'expression'
    INTEGER_CONSTANT = 'integerConstant'
    STRING_CONSTANT = 'stringConstant'
    KEYWORD_CONSTANT = 'keywordConstant'
    TERM = 'term'


symbol = ['{', '}', '(', ')', '[', ']', '.', ', ', ';', '+', '-', '*', '/', '&', '|', '<', '>', '=', '~']
subDec = ['constructor', 'function', 'method']
classDec = ['static', 'field']
types = ['int', 'char', 'boolean', 'void']
stat = ['let', 'if', 'while', 'do', 'return']
op = ['+', '-', '*', '/', '&amp;', '|', '&lt;', '&gt;', '=']
unaryOp = ['-', '~']
keywordConst = ['true', 'false', 'null', 'this']


class CompilationEngine:

    varName = []
    subName = []
    className = []

    def __init__(self, finput, foutput):

        # self.fin = open(input, "r")
        self.fout = foutput  # open(output, "w")
        self.t = JackTokenizer(finput)
        self.CompileClass()

    def writeO(self, t):
        self.fout.write(" <" + t + "> \n")

    def writeC(self, t):

        self.fout.write(" </" + t + "> \n")

    def writeOC(self, t, c):

```

```
60          self.fout.write(" <" + t + "> " + c + " </" + t + "> \n")
61
62      def CompileClass(self):
63          self.writeO(Structure.CLASS)
64          # self.t = ; #get type
65          self.writeOC(self.t.tokenType(), self.t.keyWord()) # class
66          self.t.advance()
67          self.writeOC(self.t.tokenType(), self.t.identifier()) # class name
68          if self.t.identifier() not in self.className:
69              self.className.append(self.t.identifier())
70          self.t.advance()
71          self.writeOC(self.t.tokenType(), self.t.symbol()) # '{'
72          self.t.advance()
73
74          while self.t.keyWord() in classDec:
75              self.CompileClassVarDec()
76
77          while self.t.keyWord() in subDec:
78              self.CompileSubroutine()
79
80          self.writeOC(self.t.tokenType(), self.t.symbol()) # '}'
81
82          self.writeC(Structure.CLASS)
83
84
85
86
87
88
89
90
91      def CompileClassVarDec(self):
92          self.writeO(Structure.CLASSVARDEC)
93          self.writeOC(self.t.tokenType(), self.t.keyWord())  # static|field
94          self.t.advance()
95
96          if self.t.tokenType() is Type.KEYWORD:
97              self.writeOC(self.t.tokenType(), self.t.keyWord())  # type
98          else:
99              self.writeOC(self.t.tokenType(), self.t.identifier())  # type
100             if self.t.identifier() not in self.className:
101                 self.className.append(self.t.identifier())
102         self.t.advance()
103         self.writeOC(self.t.tokenType(), self.t.identifier())  # varName
104         self.varName.append(self.t.identifier())
105         self.t.advance()
106         while (self.t.tokenType() is Type.SYMBOL) & (self.t.symbol() == ','):
107             self.writeOC(self.t.tokenType(), self.t.symbol())  # ','
108             self.t.advance()
109             self.writeOC(self.t.tokenType(), self.t.identifier())  # varName3
110             self.varName.append(self.t.identifier())
111             self.t.advance()
112
113         self.writeOC(self.t.tokenType(), self.t.symbol())  # ';'
114         self.t.advance()
115         self.writeC(Structure.CLASSVARDEC)
116
117     def CompileSubroutine(self):
118         self.writeO(Structure.SUBROUTINEDEC)
119
120         self.writeOC(self.t.tokenType(), self.t.keyWord())  # subDec
121         self.t.advance()
122         if self.t.tokenType() is Type.KEYWORD:
123             self.writeOC(self.t.tokenType(), self.t.keyWord())  # type
124         else:
125             self.writeOC(self.t.tokenType(), self.t.identifier())  # type
126             if self.t.identifier() not in self.className:
127                 self.className.append(self.t.identifier())
```

```
128             self.t.advance()
129             self.writeOC(self.t.tokenType(), self.t.identifier())  # subName
130             self.subName.append(self.t.identifier())
131             self.t.advance()
132             self.writeOC(self.t.tokenType(), self.t.symbol())  # '('
133             self.t.advance()
134             self.compileParameterList()
135             self.writeOC(self.t.tokenType(), self.t.symbol())  # ')'
136             self.t.advance()
137             self.writeO(Structure.SUBROUTINEBODY)
138             self.writeOC(self.t.tokenType(), self.t.symbol())  # '{'
139             self.t.advance()
140             while (self.t.tokenType() is Type.KEYWORD) & (self.t.keyWord() == 'var'):
141                 self.compileVarDec()
142             self.compileStatements()
143             self.writeOC(self.t.tokenType(), self.t.symbol())  # '}'
144             self.writeC(Structure.SUBROUTINEBODY)
145 
146             self.t.advance()
147             self.writeC(Structure.SUBROUTINEDEC)
148 
149         def compileParameterList(self):
150             self.writeO(Structure.PARAM_LIST)
151 
152             if self.t.tokenType() != Type.SYMBOL:
153                 if self.t.tokenType() is Type.KEYWORD:
154                     self.writeOC(self.t.tokenType(), self.t.keyWord())  # type
155                 else:
156                     self.writeOC(self.t.tokenType(), self.t.identifier())  # type
157                     if self.t.identifier() not in self.className:
158                         self.className.append(self.t.identifier())
159                 self.t.advance()
160                 self.writeOC(self.t.tokenType(), self.t.identifier())  # varName
161                 self.varName.append(self.t.identifier())
162                 self.t.advance()
163                 while (self.t.tokenType() is Type.SYMBOL) & (self.t.symbol() == ','):
164                     self.writeOC(self.t.tokenType(), self.t.symbol())  # ','
165                     self.t.advance()
166                     if self.t.tokenType() is Type.KEYWORD:
167                         self.writeOC(self.t.tokenType(), self.t.keyWord())  # type
168                     else:
169                         self.writeOC(self.t.tokenType(), self.t.identifier())  # type
170                         if self.t.identifier() not in self.className:
171                             self.className.append(self.t.identifier())
172                     self.t.advance()
173                     self.writeOC(self.t.tokenType(), self.t.identifier())  # varName
174                     self.varName.append(self.t.identifier())
175                     self.t.advance()
176             self.writeC(Structure.PARAM_LIST)
177 
178         def compileVarDec(self):
179             self.writeO(Structure.VAR_DEC)
180 
181             self.writeOC(self.t.tokenType(), self.t.keyWord())  # var
182             self.t.advance()
183             if self.t.tokenType() is Type.KEYWORD:
184                 self.writeOC(self.t.tokenType(), self.t.keyWord())  # type
185             else:
186                 self.writeOC(self.t.tokenType(), self.t.identifier())  # type
187                 if self.t.identifier() not in self.className:
188                     self.className.append(self.t.identifier())
189             self.t.advance()
190             self.writeOC(self.t.tokenType(), self.t.identifier())  # varName
191             self.varName.append(self.t.identifier())
192             self.t.advance()
193             while (self.t.tokenType() is Type.SYMBOL) & (self.t.symbol() == ','):
194                 self.writeOC(self.t.tokenType(), self.t.symbol())  # ','
195                 self.t.advance()
```

```
196             self.writeOC(self.t.tokenType(), self.t.identifier())  # varName
197             self.varName.append(self.t.identifier())
198             self.t.advance()
199 
200         self.writeOC(self.t.tokenType(), self.t.symbol())  # ';'
201         self.t.advance()
202         self.writeC(Structure.VAR_DEC)
203 
204     def compileStatements(self):
205         self.writeO(Structure.STATEMENTS)
206         while (self.t.tokenType() is Type.KEYWORD) & (self.t.keyWord() in stat):
207             if self.t.keyWord() == 'let':
208                 self.compileLet()
209             elif self.t.keyWord() == 'if':
210                 self.compileIf()
211             elif self.t.keyWord() == 'while':
212                 self.compileWhile()
213             elif self.t.keyWord() == 'do':
214                 self.compileDo()
215             elif self.t.keyWord() == 'return':
216                 self.compileReturn()
217 
218         self.writeC(Structure.STATEMENTS)
219 
220     def compileDo(self):
221         self.writeO(Structure.STATEMENT_DO)
222 
223         self.writeOC(self.t.tokenType(), self.t.keyWord())  # stat
224         self.t.advance()
225 
226         self.compileSubCall()
227 
228         self.writeOC(self.t.tokenType(), self.t.symbol())  # ';'
229         self.t.advance()
230 
231         self.writeC(Structure.STATEMENT_DO)
232 
233     def compileLet(self):
234         self.writeO(Structure.STATEMENT_LET)
235 
236         self.writeOC(self.t.tokenType(), self.t.keyWord())  # stat
237         self.t.advance()
238         self.writeOC(self.t.tokenType(), self.t.keyWord())  # varName
239         self.t.advance()
240 
241         if self.t.symbol() == '[':
242             self.writeOC(self.t.tokenType(), self.t.symbol())  # '['
243             self.t.advance()
244             self.compileExpression()
245             self.writeOC(self.t.tokenType(), self.t.symbol())  # ']'
246             self.t.advance()
247 
248         self.writeOC(self.t.tokenType(), self.t.symbol())  # '='
249         self.t.advance()
250         self.compileExpression()
251         self.writeOC(self.t.tokenType(), self.t.symbol())  # ';'
252         self.t.advance()
253 
254         self.writeC(Structure.STATEMENT_LET)
255 
256     def compileWhile(self):
257         self.writeO(Structure.STATEMENT_WHILE)
258 
259         self.writeOC(self.t.tokenType(), self.t.keyWord())  # stat
260         self.t.advance()
261 
262         self.writeOC(self.t.tokenType(), self.t.symbol())  # '('
263         self.t.advance()
```

```
264             self.compileExpression()
265             self.writeOC(self.t.tokenType(), self.t.symbol())  # ')'
266             self.t.advance()
267 
268             self.writeOC(self.t.tokenType(), self.t.symbol())  # '{'
269             self.t.advance()
270             self.compileStatements()
271             self.writeOC(self.t.tokenType(), self.t.symbol())  # '}'
272             self.t.advance()
273 
274             self.writeC(Structure.STATEMENT_WHILE)
275 
276         def compileReturn(self):
277             self.writeO(Structure.STATEMENT_RETURN)
278 
279             self.writeOC(self.t.tokenType(), self.t.keyWord())  # stat
280             self.t.advance()
281 
282             if not ((self.t.tokenType() is Type.SYMBOL) and (self.t.symbol() == ';')):
283                 self.compileExpression()
284 
285             self.writeOC(self.t.tokenType(), self.t.symbol())  # ';'
286             self.t.advance()
287 
288             self.writeC(Structure.STATEMENT_RETURN)
289 
290         def compileIf(self):
291             self.writeO(Structure.STATEMENT_IF)
292 
293             self.writeOC(self.t.tokenType(), self.t.keyWord())  # stat
294             self.t.advance()
295 
296             self.writeOC(self.t.tokenType(), self.t.symbol())  # '('
297             self.t.advance()
298             self.compileExpression()
299             self.writeOC(self.t.tokenType(), self.t.symbol())  # ')'
300             self.t.advance()
301 
302             self.writeOC(self.t.tokenType(), self.t.symbol())  # '{'
303             self.t.advance()
304             self.compileStatements()
305             self.writeOC(self.t.tokenType(), self.t.symbol())  # '}'
306             self.t.advance()
307 
308             if (self.t.tokenType() is Type.KEYWORD) & (self.t.keyWord() == 'else'):
309                 self.writeOC(self.t.tokenType(), self.t.keyWord())  # else
310                 self.t.advance()
311                 self.writeOC(self.t.tokenType(), self.t.symbol())  # '{'
312                 self.t.advance()
313                 self.compileStatements()
314                 self.writeOC(self.t.tokenType(), self.t.symbol())  # '}'
315                 self.t.advance()
316 
317             self.writeC(Structure.STATEMENT_IF)
318 
319         def compileExpression(self):
320             self.writeO(Structure.EXPRESSION)
321             self.compileTerm()
322 
323             while (self.t.tokenType() is Type.SYMBOL) & (self.t.symbol() in op):
324 
325                 self.writeOC(self.t.tokenType(), self.t.symbol())  # 'op'
326                 self.t.advance()
327                 self.compileTerm()
328 
329             self.writeC(Structure.EXPRESSION)
330 
331         def compileTerm(self):
```

```
332             self.writeO(Structure.TERM)
333             if self.t.tokenType() == Type.STRING_CONST:
334 
335                 self.writeOC(self.t.tokenType(), self.t.stringVal())  # string
336                 self.t.advance()
337             elif self.t.tokenType() == Type.INT_CONST:
338 
339                 self.writeOC(self.t.tokenType(), self.t.intVal())  # int
340                 self.t.advance()
341             elif (self.t.tokenType() is Type.KEYWORD) & (self.t.keyWord() in keywordConst):
342 
343                 self.writeOC(self.t.tokenType(), self.t.keyWord())  # keyboard const
344                 self.t.advance()
345             elif (self.t.tokenType() is Type.IDENTIFIER) & (self.t.identifier() in self.varName):
346 
347                 self.writeOC(self.t.tokenType(), self.t.identifier())  # var name
348                 self.t.advance()
349                 if (self.t.tokenType() is Type.SYMBOL) & (self.t.symbol() == '['):
350 
351                     self.writeOC(self.t.tokenType(), self.t.symbol())  # '['
352                     self.t.advance()
353                     self.compileExpression()
354                     self.writeOC(self.t.tokenType(), self.t.symbol())  # ']'
355                     self.t.advance()
356                 elif self.t.symbol() == '.':
357                     self.writeOC(self.t.tokenType(), self.t.symbol())  # '.'
358                     self.t.advance()
359                     self.writeOC(self.t.tokenType(), self.t.identifier())  # subName
360                     self.t.advance()
361 
362                     self.writeOC(self.t.tokenType(), self.t.symbol())  # '('
363                     self.t.advance()
364                     self.compileExpressionList()
365                     self.writeOC(self.t.tokenType(), self.t.symbol())  # ')'
366                     self.t.advance()
367 
368             elif (self.t.tokenType() is Type.SYMBOL) & (self.t.symbol() == '('):
369 
370                 self.writeOC(self.t.tokenType(), self.t.symbol())  # '('
371                 self.t.advance()
372                 self.compileExpression()
373                 self.writeOC(self.t.tokenType(), self.t.symbol())  # ')'
374                 self.t.advance()
375             elif (self.t.tokenType() is Type.SYMBOL) & (self.t.symbol() in unaryOp):
376 
377                 self.writeOC(self.t.tokenType(), self.t.symbol())  # unary op
378                 self.t.advance()
379                 self.compileTerm()
380             else:# (self.t.tokenType() is Type.IDENTIFIER) & (self.t.identifier() in self.subName):
381 
382                 self.compileSubCall()
383 
384             self.writeC(Structure.TERM)
385 
386         def compileExpressionList(self):
387             self.writeO(Structure.EXPRESSION_LIST)
388 
389             if not ((self.t.tokenType() is Type.SYMBOL) & (self.t.symbol() == ')')):
390                 self.compileExpression()
391 
392                 while ((self.t.tokenType() is Type.SYMBOL) & (self.t.symbol() == ',')):
393                     self.writeOC(self.t.tokenType(), self.t.symbol())  # ','
394                     self.t.advance()
395                     self.compileExpression()
396 
397             self.writeC(Structure.EXPRESSION_LIST)
398 
399         def compileSubCall(self):
```

```
400             self.writeOC(self.t.tokenType(), self.t.identifier())  # sub/class/varName
401             self.t.advance()
402 
403             if self.t.symbol() == '.':
404                 self.writeOC(self.t.tokenType(), self.t.symbol())  # '.'
405                 self.t.advance()
406                 self.writeOC(self.t.tokenType(), self.t.identifier())  # subName
407                 self.t.advance()
408 
409             self.writeOC(self.t.tokenType(), self.t.symbol())  # '('
410             self.t.advance()
411             self.compileExpressionList()
412             self.writeOC(self.t.tokenType(), self.t.symbol())  # ')'
413             self.t.advance()
```

4 JackAnalyzer

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

5 JackAnalyzer.py

```
__author__ = 'inbaravni'

from JackTokenizer import *
from CompilationEngine import *
import sys
import os



def main(argv):
    # file given
    if (os.path.isfile(argv[0])):
        try:
            file = open(argv[0].split('.')[-2] + '.xml','w')    # Trying to create a new file
            CompilationEngine(argv[0], file)
            file.close()
        except:
            print('Can\'t create xml file')




    # directory given
    else:
        #path = os.path.abspath(argv[0])+'/'
        path = argv[0]
        if path[-1] != '/':
            path = path+'/'
        name = path + path.split('/')[-2]
        for each_file in os.listdir(argv[0]):
            if each_file.endswith(".jack"):
                try:
                    file = open((path+each_file).split('.')[-2] + '.xml','w')    # Trying to create a new file
                    CompilationEngine((path+each_file), file)
                    file.close();
                except:
                    print('Can\'t create an xml file')









if __name__ == "__main__":
    main(sys.argv[1:])
```

6 JackTokenizer.py

```
1  import re
2
3  __author__ = 'inbaravni'
4
5
6  #
7  #
8  # with open('/cs/stud/inbaravni/safe/NAND2tetris/ex10/List.jack', 'r') as self.f:
9  #         data = self.f.read().replace('\s', '')
10 #
11 # array = data.split('')
12 # print(array)
13
14 symbol_array = ['{', '}', '(', ')', '[', ']', '.', ',', ';',
15                 '+', '-', '*', '/', '&', '|', '<', '>', '=', '~']
16
17
18 class Type:
19     KEYWORD = 'keyword'
20     SYMBOL = 'symbol'
21     IDENTIFIER = 'identifier'
22     INT_CONST = 'integerConstant'
23     STRING_CONST = 'stringConstant'
24
25 keyword_array = ['class', 'constructor', 'function', 'method', 'field', 'static',
26                  'var', 'int', 'char', 'boolean', 'void', 'true', 'false', 'null',
27                  'this', 'let', 'do', 'if', 'else', 'while', 'return']
28
29 # var_array = ['var', 'field', 'static']
30 # sub_array = ['constructor', 'function', 'method']
31 # type_array = ['int', 'char', 'boolean', 'void']
32 class Keyword:
33
34     CLASS = 'class'
35     CONSTRUCTOR = 'constructor'
36     FUNCTION = 'function'
37     METHOD = 'method'
38     FIELD = 'field'
39     STATIC = 'static'
40     VAR = 'var'
41     INT = 'int'
42     CHAR = 'char'
43     BOOLEAN = 'boolean'
44     VOID = 'void'
45     TRUE = 'true'
46     FALSE = 'false'
47     NULL = 'null'
48     THIS = 'this'
49     LET = 'let'
50     DO = 'do'
51     IF = 'if'
52     ELSE = 'else'
53     WHILE = 'while'
54     RETURN = 'return'
55
56
57 class JackTokenizer:
58
59     currentStringVal = ''
```

```
60
61      # curDec = 'class'
62      # varNames = []
63      # subName = []
64      # className = []
65
66      def __init__(self, file_name):
67
68          #
69          with open(file_name, 'r') as self.f:
70              text = self.f.read()
71              # ftext = ''
72              # in_c = 0
73              # in_s = 0
74              # for t in text:
75              #     if in_c == 2:
76              #     if t == '/':
77              #         in_c = 1
78              #     if t == '/' or t == '\*':
79              #         in_c = 2;
80              arr = re.split("(//)|(/\*)|(\*/)|(\")|(\n)", text)
81
82              array = []
83
84              for token in arr:
85                  if (token != '')  and (token is not None):
86                      array.append(token)
87
88              data = ''
89              in_c = 0
90              in_line_c = 0
91              in_s = 0
92              for token in array:
93                  if in_line_c == 1:
94                      if token == '\n':
95                          in_line_c = 0
96                      continue
97                  if in_c == 1:
98                      if token == '*/':
99                          in_c = 0
100                     continue
101                 if in_s == 1:
102                     if token == '"':
103                         in_s = 0
104                     t = re.sub('&','&amp;',token)
105                     t = re.sub('<','&lt;',t)
106                     t = re.sub('>','&gt;',t)
107                     data+=t
108                     continue
109                 if token == '//':
110                     in_line_c = 1;
111                     data+=" "
112                     continue
113                 if token == '/*':
114                     in_c = 1
115                     data+=" "
116                     continue
117                 if token == '"':
118                     in_s = 1
119                     data+=token
120                     continue
121                 data+=token
122             # print(data)
123
124
125
126             # regex_space_comments = '((/\*.*?(\n.*?)*?\*/)|(//.*?\n)|[\n\t ]+)'
127             # data = re.sub(regex_space_comments, ' ', self.f.read())
```

```
128             # data = re.sub('([\n\t ]+)', ' ', data)
129         symbols = r'(".*?")|([\[\];"(){}\.\,\-\+\*/&|<>~=])|[ \n\t]'
130 
131         self.arr = re.split(symbols, data)
132 
133         self.index = 0
134 
135         self.array = []
136 
137         for token in self.arr:
138             if (token != '') and (token != ' ') and (token is not None):
139                 self.array.append(token)
140         self.arraySize = len(self.array)
141 
142     def hasMoreTokens(self):
143 
144         if self.index + 1 <= self.arraySize - 1:
145             return True
146         return False
147 
148     def advance(self):
149         self.index += 1
150         # if self.array[self.index] == '"':
151         #     self.createString()
152 
153 
154         # if self.tokenType() == Type.SYMBOL:
155         #     if (self.array[self.index] == ';'):
156         #         self.curDec = ''
157         #     elif (self.array[self.index] == ')'):
158         #         self.curDec = ''
159         # elif self.tokenType() == Type.KEYWORD:
160         #     if (self.array[self.index] == 'class'):
161         #         self.curDec = 'class'
162         #     elif self.array[self.index] in var_array:
163         #         self.curDec = 'var'
164         #     elif self.array[self.index] in sub_array:
165         #         self.curDec = 'sub'
166         #     elif (self.curDec == 'var') & (self.array[self.index] in type_array):
167         #         self.curDec = 'var_type'
168         #     elif (self.curDec == 'sub') & (self.array[self.index] in type_array):
169         #         self.curDec = 'sub_type'
170         # elif self.tokenType() == Type.IDENTIFIER:
171         #     if self.curDec == 'class':
172         #         self.className.append(self.array[self.index])
173         #         self.curDec = ''
174         #     elif self.curDec == 'var':
175         #         if self.array[self.index] not in self.className:
176 
177         #             self.className.append(self.array[self.index])
178         #         self.curDec = 'var_type'
179         #     elif self.curDec == 'sub':
180         #         if self.array[self.index] not in self.className:
181         #             print("      found class name")
182         #             self.className.append(self.array[self.index])
183         #         self.curDec = 'sub_type'
184         #     elif self.curDec == 'var_type':
185         #         print("      found var name")
186         #         self.varNames.append(self.array[self.index])
187         #         self.curDec = 'var'
188         #     elif self.curDec == 'sub_type':
189         #         print("      found sub name")
190         #         self.subName.append(self.array[self.index])
191         #         self.curDec = 'var'
192         #
193         # print("                                                    " + self.array[self.index])
194         # print("                                          " + self.curDec)
195 
```

```
196     def tokenType(self):
197
198         if self.array[self.index] in symbol_array:
199             return Type.SYMBOL
200         elif self.array[self.index] in keyword_array:
201             return Type.KEYWORD
202         elif self.representsInt(self.array[self.index]):
203             return Type.INT_CONST
204         elif self.array[self.index][0] == '"':
205             return Type.STRING_CONST
206         else:
207             return Type.IDENTIFIER
208
209     def keyWord(self):
210         return self.array[self.index]
211
212     def symbol(self):
213         if self.array[self.index] == '<':
214             return "&lt;"
215         elif self.array[self.index] == '>':
216             return "&gt;"
217         elif self.array[self.index] == '"':
218             return "&quot;"
219         elif self.array[self.index] == "&":
220             return "&amp;"
221         else:
222             return self.array[self.index]
223
224     def identifier(self):
225         return self.array[self.index]
226
227     def intVal(self):
228         return self.array[self.index]
229
230     def stringVal(self):
231         return self.array[self.index][1:-1]
232         return self.currentStringVal
233
234     def representsInt(self, s):
235         try:
236             int(s)
237             return True
238         except ValueError:
239             return False
240
241     def createString(self):
242         self.currentStringVal = ''
243         self.index += 1
244         if self.array[self.index] != '"':
245             self.currentStringVal += (self.array[self.index])
246             self.index += 1
247         while self.array[self.index] != '"':
248             self.currentStringVal += (" " + self.array[self.index])
249             self.index += 1
```

7 Makefile

```
1  ###############################################################################
2  #
3  # Makefile for Python project
4  #
5  # Students:
6  # Roi Greenberg, ID 30557123, roi.greenberg@mail.huji.ac.il
7  # Inbar Avni, ID 201131760, inbar.avni@mail.huji.ac.il
8  #
9  ###############################################################################
10
11
12
13 SRCS=*.py
14 EXEC=JackAnalyzer
15
16 TAR=tar
17 TARFLAGS=cvf
18 TARNAME=project10.tar
19 TARSRCS=$(SRCS) $(EXEC) README Makefile
20
21 all:
22     chmod a+x $(EXEC)
23 tar:
24     $(TAR) $(TARFLAGS) $(TARNAME) $(TARSRCS)
25
26 clean:
27     rm -f *~
```