Contents

1 Basic Test Results

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

  preparing sub.tar
dos2unix: converting file /tmp/bodek.kgzgqf/nand2tet/Project11/roigreenberg/presubmission/testdir/stud/sub.tar/README to Uni
  checking sub.tar
chmod +X JackCompiler
starting test after make
START RUNNING Average
START RUNNING ComplexArrays
START RUNNING ConvertToBin
START RUNNING Seven

********** 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 11 - Compiler II - Code Generation

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




Submitted Files

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

README - This file.

JackCompiler.py - top-level driver that sets up and invokes the other modules
JackTokenizer.py - parse the file
SymbolTable.py - symbol table
VMWriter.py - output module for generation VM code
CompilationEngine.py - recursive top-down compilation engine
Makefile - a makefile.

Remarks

-------

* No remarks for that time.
```

3 CompilationEngine.py

```
from JackTokenizer import *
from VMWriter import *
from SymbolTable import *
import inspect

__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']
ops = ['+', '-', '*', '/', '&', '|', 'lt', 'gt', '=']
unaryOp = ['-', '~']
keywordConst = ['true', 'false', 'null', 'this']


class CompilationEngine:

    varNames = []
    subNames = []
    classNames = []
    whileI = 0
    ifI = 0

    def __init__(self, finput, foutput):

        self.w = VMWriter(foutput)  # open(output, "w")
        self.t = JackTokenizer(finput)
        self.s = SymbolTable()
        self.whileI = 0
        self.ifI = 0

        self.className = ''
        self.CompileClass()

    def CompileClass(self):
```

```
60          # print (inspect.stack()[0][3])
61          self.t.advance()  # class
62          self.t.advance()  # class name
63          self.className = self.t.identifier()
64          if self.t.identifier() not in self.classNames:
65              self.classNames.append(self.className)
66          self.t.advance()  # '{'
67          self.t.advance()
68
69          while self.t.keyWord() in classDec:
70              self.CompileClassVarDec()
71
72          while self.t.keyWord() in subDec:
73              self.CompileSubroutine()
74
75          self.w.close()
76
77      def CompileClassVarDec(self):
78          # print(inspect.stack()[0][3])
79          kind = self.t.keyWord()  # static|field
80
81          self.t.advance()
82
83          if self.t.tokenType() is Type.KEYWORD:
84              typeV = self.t.keyWord()  # type
85          else:
86              typeV = self.t.identifier()  # type
87              if type not in self.classNames:
88                  self.classNames.append(type)
89          self.t.advance()
90          name = self.t.identifier()  # varName
91          self.varNames.append(name)
92          self.s.define(name, typeV, kind)
93          self.t.advance()
94          while (self.t.tokenType() is Type.SYMBOL) & (self.t.symbol() == ','):
95              self.t.advance()
96              name = self.t.identifier()  # varName3
97              # # print(name)
98              self.varNames.append(name)
99              # # print(name + " " + kind + " "+ typeV)
100             self.s.define(name, typeV, kind)
101             self.t.advance()
102
103         self.t.advance()
104         # # print("var added")
105     def CompileSubroutine(self):
106         # print(inspect.stack()[0][3])
107         kind = self.t.keyWord()  # subDec
108         self.t.advance()
109         if self.t.tokenType() is not Type.KEYWORD:
110             if self.t.identifier() not in self.classNames:
111                 self.classNames.append(self.t.identifier())
112         self.t.advance()
113         name = self.t.identifier()  # subName
114         self.subNames.append(name)
115         self.s.startSubroutine()
116         if kind == 'method':
117             self.s.define('instance', self.className, 'ARG')
118
119         self.t.advance()
120         self.t.advance()
121         self.compileParameterList()
122         self.t.advance()
123         self.t.advance()
124         while (self.t.tokenType() is Type.KEYWORD) & (self.t.keyWord() == 'var'):
125             self.compileVarDec()
126
127         fName = self.className + '.' + name
```

```
128
129             numLocals = self.s.varCount('VAR')
130             self.w.writeFunction(fName, numLocals)
131             # # print("D")
132             if kind == 'constructor':
133                 numFields = self.s.varCount('FIELD')
134                 self.w.writePush('CONST', numFields)
135                 self.w.writeCall('Memory.alloc', 1)
136                 self.w.writePop('POINTER', 0)
137             elif kind == 'method':
138                 self.w.writePush('ARG', 0)
139                 self.w.writePop('POINTER', 0)
140
141             self.compileStatements()
142             self.t.advance()
143         def compileParameterList(self):
144             # print(inspect.stack()[0][3])
145             if self.t.tokenType() != Type.SYMBOL:
146
147                 if self.t.tokenType() is Type.KEYWORD:
148                     varVype = self.t.keyWord()  # type
149                 else:
150                     varVype = self.t.identifier()  # type
151                     if self.t.identifier() not in self.classNames:
152                         self.classNames.append(self.t.identifier())
153                 self.t.advance()
154                 name = self.t.identifier()  # varName
155                 self.varNames.append(self.t.identifier())
156                 self.s.define(name, varVype, 'ARG')
157                 self.t.advance()
158                 # # print("PP")
159                 while (self.t.tokenType() is Type.SYMBOL) & (self.t.symbol() == ','):
160                     self.t.advance()
161                     if self.t.tokenType() is Type.KEYWORD:
162                         typeV = self.t.keyWord()  # type
163                     else:
164                         typeV = self.t.identifier()  # type
165                         if self.t.identifier() not in self.classNames:
166                             self.classNames.append(self.t.identifier())
167                     self.t.advance()
168                     name = self.t.identifier()  # varName
169                     self.varNames.append(self.t.identifier())
170                     self.s.define(name, typeV, 'ARG')
171                     self.t.advance()
172         def compileVarDec(self):
173             # print(inspect.stack()[0][3])
174             self.t.advance()
175             if self.t.tokenType() is Type.KEYWORD:
176                 type = self.t.keyWord()  # type
177             else:
178                 type = self.t.identifier()  # type
179                 if self.t.identifier() not in self.classNames:
180                     self.classNames.append(self.t.identifier())
181             self.t.advance()
182             name = self.t.identifier()  # varName
183             self.s.define(name, type, 'VAR')
184             self.varNames.append(self.t.identifier())
185             self.t.advance()
186             while (self.t.tokenType() is Type.SYMBOL) & (self.t.symbol() == ','):
187                 self.t.advance()
188                 name = self.t.identifier()  # varName
189                 self.s.define(name, type, 'VAR')
190                 self.varNames.append(self.t.identifier())
191                 self.t.advance()
192
193             self.t.advance()
194         def compileStatements(self):
195             # print(inspect.stack()[0][3])
```

```
196             while (self.t.tokenType() is Type.KEYWORD) & (self.t.keyWord() in stat):
197                 if self.t.keyWord() == 'let':
198                     self.compileLet()
199                 elif self.t.keyWord() == 'if':
200                     self.compileIf()
201                 elif self.t.keyWord() == 'while':
202                     self.compileWhile()
203                 elif self.t.keyWord() == 'do':
204                     self.compileDo()
205                 elif self.t.keyWord() == 'return':
206                     self.compileReturn()
207 
208         def compileDo(self):
209             # print(inspect.stack()[0][3])
210             self.t.advance()
211 
212             self.compileSubCall()
213             self.w.writePop('TEMP', 0)
214             self.t.advance()
215 
216         def compileLet(self):
217             # print(inspect.stack()[0][3])
218             self.t.advance()
219             name = self.t.keyWord()  # varName
220             kind = self.s.kindOf(name)
221             index = self.s.indexOf(name)
222             self.t.advance()
223 
224             if self.t.symbol() == '[':
225                 self.t.advance()
226                 self.compileExpression()
227 
228                 self.w.writePush(kind, index)
229                 self.w.writeArithmetic('add')
230 
231                 self.t.advance()
232                 self.t.advance()
233 
234                 self.compileExpression()
235                 self.w.writePop('TEMP', 0)
236 
237                 self.t.advance()
238 
239                 self.w.writePop('POINTER', 1)
240                 self.w.writePush('TEMP', 0)
241 
242                 self.w.writePop('THAT', 0)
243 
244                 return
245 
246             self.t.advance()
247             self.compileExpression()
248             self.w.writePop(kind, index)
249 
250             self.t.advance()
251 
252         def compileWhile(self):
253             # print(inspect.stack()[0][3])
254             whileIndex = self.whileI
255             self.whileI += 1
256 
257             self.w.writeLabel('WHILE_START.' + str(whileIndex))
258 
259             self.t.advance()
260 
261             self.t.advance()
262             self.compileExpression()
263 
```

```
264             self.w.writeArithmetic('not')
265             self.w.writeIf('WHILE_END.' + str(whileIndex))
266             self.t.advance()
267 
268             self.t.advance()
269             self.compileStatements()
270 
271             self.w.writeGoto('WHILE_START.' + str(whileIndex))
272             self.w.writeLabel('WHILE_END.' + str(whileIndex))
273 
274             self.t.advance()
275 
276 
277 
278         def compileReturn(self):
279             # print(inspect.stack()[0][3])
280             self.t.advance()
281 
282             if not ((self.t.tokenType() is Type.SYMBOL) and (self.t.symbol() == ';')):
283                 self.compileExpression()
284             else:
285                 self.w.writePush('CONST', 0)
286 
287             self.t.advance()
288 
289             self.w.writeReturn()
290 
291         def compileIf(self):
292             # print(inspect.stack()[0][3])
293 
294             ifIndex = self.ifI
295             self.ifI += 1
296             self.t.advance()
297 
298             self.t.advance()
299             self.compileExpression()
300             # print("in num: "+ str(ifIndex))
301             self.w.writeIf('IF_TRUE.' + str(ifIndex))
302             self.w.writeGoto('IF_FALSE.' + str(ifIndex))
303             self.w.writeLabel('IF_TRUE.' + str(ifIndex))
304 
305             self.t.advance()
306 
307             self.t.advance()
308             self.compileStatements()
309 
310             self.w.writeGoto('IF_END.' + str(ifIndex))
311             self.w.writeLabel('IF_FALSE.' + str(ifIndex))
312 
313             self.t.advance()
314 
315             if (self.t.tokenType() is Type.KEYWORD) & (self.t.keyWord() == 'else'):
316                 self.t.advance()
317                 self.t.advance()
318                 self.compileStatements()
319 
320                 self.t.advance()
321 
322             self.w.writeLabel('IF_END.' + str(ifIndex))
323 
324             self.ifI += 1
325 
326         def compileExpression(self):
327             # print(inspect.stack()[0][3])
328             self.compileTerm()
329 
330             while (self.t.tokenType() is Type.SYMBOL) & (self.t.symbol() in ops):
331 
```

```
            opr = self.t.symbol()  # 'op'

            self.t.advance()
            self.compileTerm()

            if opr == '*':
                self.w.writeCall('Math.multiply', 2)
            elif opr == '/':
                self.w.writeCall('Math.divide', 2)
            else:
                self.w.writeArithmetic(opr)

    def compileTerm(self):
        # print(inspect.stack()[0][3])
        # # print ("Term: " + self.t.tokenType() + " PP " + self.t.symbol())
        if self.t.tokenType() == Type.STRING_CONST:

            string = self.t.stringVal()  # string
            self.w.writePush('CONST', len(string))
            self.w.writeCall('String.new', 1)

            for st in string:
                self.w.writePush('CONST', ord(st))
                self.w.writeCall('String.appendChar', 2)

            self.t.advance()

        elif self.t.tokenType() == Type.INT_CONST:

            intV = self.t.intVal()  # int
            self.w.writePush('CONST', intV)
            self.t.advance()

        elif (self.t.tokenType() is Type.KEYWORD) & (self.t.keyWord() in keywordConst):

            keyword = self.t.keyWord()  # keyboard const
            if keyword == 'this':
                self.w.writePush('POINTER', 0)
            else:
                self.w.writePush('CONST', 0)
                if keyword == 'true':
                    self.w.writeArithmetic('not')

            self.t.advance()

        elif (self.t.tokenType() is Type.IDENTIFIER) & (self.t.identifier() in self.varNames):

            var = self.t.identifier()  # var name
            kind = self.s.kindOf(var)
            index = self.s.indexOf(var)
            typeVar = self.s.typeOf(var)
            argNum = 0

            self.t.advance()
            if (self.t.tokenType() is Type.SYMBOL) & (self.t.symbol() == '['):

                self.t.advance()
                self.compileExpression()

                self.w.writePush(kind, index)
                self.w.writeArithmetic('add')
                self.w.writePop('POINTER', 1)
                self.w.writePush('THAT', 0)

                self.t.advance()
            elif self.t.symbol() == '.':
                self.t.advance()
```

```
                subName = self.t.identifier()  # subName

                self.w.writePush(kind,index)  #TODO

                funName = typeVar + '.' + subName
                argNum += 1

                self.t.advance()

                self.t.advance()
                argNum += self.compileExpressionList()
                self.w.writeCall(funName, argNum)
                self.t.advance()
            else:
                self.w.writePush(kind,index)

        elif (self.t.tokenType() is Type.SYMBOL) & (self.t.symbol() == '('):

            self.t.advance()
            self.compileExpression()
            self.t.advance()
        elif (self.t.tokenType() is Type.SYMBOL) & (self.t.symbol() in unaryOp):

            uOp = self.t.symbol()  # unary op
            self.t.advance()
            self.compileTerm()
            self.w.writeArithmetic("u"+uOp)
        else:  # (self.t.tokenType() is Type.IDENTIFIER) & (self.t.identifier() in self.subName):

            self.compileSubCall()

    def compileExpressionList(self):
        # print(inspect.stack()[0][3])
        countArg = 0

        if not ((self.t.tokenType() is Type.SYMBOL) & (self.t.symbol() == ')')):
            self.compileExpression()
            countArg += 1

            while ((self.t.tokenType() is Type.SYMBOL) & (self.t.symbol() == ',')):
                self.t.advance()
                self.compileExpression()
                countArg += 1

        return countArg

    def compileSubCall(self):
        # print(inspect.stack()[0][3])
        idnt = self.t.identifier()  # sub/class/
        kind = self.s.kindOf(idnt)
        index = self.s.indexOf(idnt)
        typeVar = self.s.typeOf(idnt)
        argNum = 0
        self.t.advance()
        # # print(typeVar)
        # # print("S " + self.t.symbol())
        if self.t.symbol() == '.':
            self.t.advance()

            subName = self.t.identifier()  # subName

            # print("F  " + idnt + " "+ subName + " " + str(typeVar))
            if typeVar == None:
                funName = idnt + '.' + subName
            else:
                self.w.writePush(kind, index)
                funName = typeVar + '.' + subName
                argNum += 1
```

```
468
469             self.t.advance()
470             #
471             # self.t.advance()
472             # argNum += self.compileExpressionList()
473             # self.w.writeCall(funName, argNum)
474             # self.t.advance()
475         else:
476             # # print("L")
477             funName = self.className + '.' + idnt
478             argNum += 1
479             self.w.writePush('POINTER', 0)
480
481         # self.t.advance()
482         self.t.advance()
483
484         # print("O " + funName + "   " + self.t.identifier())
485         argNum += self.compileExpressionList()
486         # # print("P " + funName + " " + str(argNum))
487         self.w.writeCall(funName, argNum)
488
489         self.t.advance()
```

4 JackCompiler

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

5 JackCompiler.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] + '.vm','w')    # Trying to create a new file
            CompilationEngine(argv[0], file)
            file.close()
        except:
            print('Can\'t create vm 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] + '.vm','w')    # Trying to create a new file
                    CompilationEngine((path+each_file), file)
                    file.close();
                except:
                    print('Can\'t create an vm 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              print(array)
88
89              data = ''
90              in_c = 0
91              in_line_c = 0
92              in_s = 0
93              for token in array:
94                  if in_line_c == 1:
95                      if token == '\n':
96                          in_line_c = 0
97                      continue
98                  if in_c == 1:
99                      if token == '*/':
100                         in_c = 0
101                     continue
102                 if in_s == 1:
103                     if token == '"':
104                         in_s = 0
105                     data+=token
106                     continue
107                 if token == '//':
108                     in_line_c = 1;
109                     data+=" "
110                     continue
111                 if token == '/*':
112                     in_c = 1
113                     data+=" "
114                     continue
115                 if token == '"':
116                     in_s = 1
117                     data+=token
118                     continue
119                 data+=token
120             # print(data)
121
122
123
124             # regex_space_comments = '((/\*.*?(\n.*?)*?\*/)|(//.*?\n)|[\n\t ]+)'
125             # data = re.sub(regex_space_comments, ' ', self.f.read())
126             # data = re.sub('([\n\t ]+)', ' ', data)
127         symbols = r'(".*?")|([\[\];"(){}\.\,\-\+\*/&|<>~=])|[ \n\t]'
```

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

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

7 Makefile

```
#############################################################################
#
# Makefile for Python project
#
# Students:
# Roi Greenberg, ID 30557123, roi.greenberg@mail.huji.ac.il
# Inbar Avni, ID 201131760, inbar.avni@mail.huji.ac.il
#
#############################################################################



SRCS=*.py
EXEC=JackCompiler

TAR=tar
TARFLAGS=cvf
TARNAME=project11.tar
TARSRCS=$(SRCS) $(EXEC) README Makefile

all:
    chmod +X $(EXEC)
tar:
    $(TAR) $(TARFLAGS) $(TARNAME) $(TARSRCS)

clean:
    rm -f *~
```

8 SymbolTable.py

```
__author__ = 'inbaravni'



class SymbolTable:

    classTable = []
    routinTable = []
    count_static = 0
    count_field = 0
    count_arg = 0
    count_var = 0

    def __init__(self):
        self.classTable = {};
        self.count_static = 0;
        self.count_field = 0;


    def startSubroutine(self):
        self.routinTable = {};
        self.count_arg = 0;
        self.count_var = 0;


    def define(self, name, type, kind):
        #class scope
        # print("add to table: name: "+name +" type:" + type + " kind: " + kind)
        if (kind == "static"):
            self.classTable[name] = [type, kind, self.count_static]
            # print("add to class table: type:" + type + " kind: " + kind + " count: " + str(self.count_static))
            self.count_static+=1

        elif (kind == "field"):
            self.classTable[name] = [type, kind, self.count_field]
            # print("add to class table: type:" + type + " kind: " + kind + " count: " + str(self.count_field))
            self.count_field+=1
        #subroutine scope
        elif (kind == "ARG"):
            self.routinTable[name] = [type, kind, self.count_arg]
            # print("add to sub table: type:" + type + " kind: " + kind + " count: " + str(self.count_arg))
            self.count_arg+=1
        elif (kind == "VAR"):
            self.routinTable[name] = [type, kind, self.count_var]
            # print("add to class table: type:" + type + " kind: " + kind + " count: " + str(self.count_var))
            self.count_var+=1


    def varCount(self, kind):
        # print("V")
        if (kind == "STATIC"):
            return self.count_static
        elif (kind == "FIELD"):
            return self.count_field
        elif (kind == "ARG"):
            return self.count_arg
        elif (kind == "VAR"):
            return self.count_var

```

```
60
61      def kindOf(self, name):
62          #subroutine scope
63          if (self.routinTable.get(name) != None):
64              return self.routinTable[name][1];
65          #class scope
66          elif (self.classTable.get(name) != None):
67              return self.classTable[name][1];
68          #not found
69          else:
70              return None
71
72
73      def typeOf(self, name):
74          #subroutine scope
75          if (self.routinTable.get(name) != None):
76              return self.routinTable[name][0];
77          #class scope
78          elif (self.classTable.get(name) != None):
79              return self.classTable[name][0];
80
81
82      def indexOf(self, name):
83          #subroutine scope
84          if (self.routinTable.get(name) != None):
85              return self.routinTable[name][2];
86          #class scope
87          elif (self.classTable.get(name) != None):
88              return self.classTable[name][2];
```

9 VMWriter.py

```
__author__ = 'inbaravni'

from SymbolTable import *
op = ['+','-','&','>','<','|','=']
uop = ['u~','u-']
ARITHMETIC = {
    '+': 'add',
    '-': 'sub',
    '&': 'and',
    '>': 'gt',
    '<': 'lt',
    '|': 'or',
    '=': 'eq',
  }

UOP = {
    'u~': 'not',
    'u-': 'neg'
}

seg = ['THAT','THIS','TEMP','STATIC','FIELD','field','VAR','var','ARG','CONST','POINTER']
KIND_TO_SEG = {
    'THAT': 'that',
    'THIS': 'this',
    'TEMP': 'temp',
    'STATIC': 'static',
    'FIELD': 'this',
    'field': 'this',
    'VAR': 'local',
    'var': 'local',
    'ARG': 'argument',
    'POINTER': 'pointer',
    'CONST': 'constant'
  }

class VMWriter:

    outputFile = ''

    def __init__(self, outputFileName):
        # print(outputFileName)
        # try:
        #     self.outputFile = open(outputFileName+'.vm', 'w+')
        # except:
        #     print("Can\'t create a .vm file")
        self.outputFile = outputFileName

    def writePush(self, segment, index):
        # print("E " + str(segment) + " "+ str(index))
        if segment in seg:
            self.outputFile.write("push " + KIND_TO_SEG[segment] + " " + str(index) + "\n")
        else:
            self.outputFile.write("push " + segment + " " + str(index) + "\n")
        # print("D")
    def writePop(self, segment, index):
        if segment in seg:
            self.outputFile.write("pop " + KIND_TO_SEG[segment] + " " + str(index) + "\n")
        else:
            self.outputFile.write("pop " + segment + " " + str(index) + "\n")
```

```
60
61      def writeArithmetic(self, command):
62          if command in op:
63              self.outputFile.write(ARITHMETIC[command] + "\n")
64          elif command in uop:
65              self.outputFile.write(UOP[command] + "\n")
66          else:
67              self.outputFile.write(command + "\n")
68
69      def writeLabel(self, label):
70          self.outputFile.write("label " + label + "\n")
71
72      def writeGoto(self, label):
73          self.outputFile.write("goto " + label + "\n")
74
75      def writeIf(self, label):
76          self.outputFile.write("if-goto " + label + "\n")
77
78      def writeCall(self, name, nArgs):
79          self.outputFile.write("call " + name + " " + str(nArgs) + "\n")
80          # print("call " + name + " " + str(nArgs))
81
82      def writeFunction(self, name, nArgs):
83          self.outputFile.write("function " + name + " " + str(nArgs) + "\n")
84
85      def writeReturn(self):
86          self.outputFile.write("return" + "\n")
87
88      def close(self):
89          self.outputFile.close()
```