DavidDuffrinProject11

November 18, 2016

1 Writing the Symbol Table Module

In [41]:
```
from collections import namedtuple

#A TableEntry is a bit like an Object. It has 4 fields, corresponding to t
# stored variable's name, type, kind, and the index of that variable withi
# the segment it is stored in. symbol tables map from names to table entri
TableEntry = namedtuple("TableEntry", ['name', 'type', 'kind', 'index'])

classTable = {}
subroutineTable = {}
staticCounter = 0
fieldCounter = 0
varCounter = 0
argCounter = 0

#Creates a new, empty, symbol table.
# This entails setting all the global variables in the
# module back to their original values.
def init():
    global classTable, subroutineTable, staticCounter, fieldCounter, varCo
    TableEntry = namedtuple("TableEntry", ['name', 'type', 'kind', 'index'
    classTable = {}
    subroutineTable = {}
    staticCounter = 0
    fieldCounter = 0
    varCounter = 0
    argCounter = 0

#Starts a new subroutine scope, by re-setting
# the contents of the subroutine portion of the
# symbol table. To do this, we set subroutineTable
# to an empty table, and varCounter and argCounter to 0.
def startSubroutine():
    global subroutineTable, varCounter, argCounter
    subroutineTable = {}
    varCounter = 0
```

```
    argCounter = 0

#Add a new identifier to the symbol table.
# "name" is the name the identifier should be stored under.
# "varType" is the type of the variable, for instance "int", "string", or
# "kind" is the storage location of the variable, either "ARG", "VAR", "FI
# Your textbook explains in Chapter 11 that variables should be stored in
# or the classTable depending on their kind.
def Define(name, varType, kind):
    global classTable, subroutineTable, staticCounter, fieldCounter, varCo
    if kind.upper() == "ARG":
        subroutineTable[name] = TableEntry(name, varType, kind, argCounter
        argCounter = argCounter + 1
    elif kind.upper() == "VAR":
        subroutineTable[name] = TableEntry(name, varType, kind, varCounter
        varCounter = varCounter + 1
    elif kind.upper() == "FIELD":
        classTable[name] = TableEntry(name, varType, kind, fieldCounter) #
        fieldCounter = fieldCounter + 1
    elif kind.upper() == "STATIC":
        classTable[name] = TableEntry(name, varType, kind, staticCounter)
        staticCounter = staticCounter + 1

#Returns the number of variables of the specified kind,
# where kind is one of static, field, arg or var.
def VarCount(kind):
    global staticCounter, fieldCounter, varCounter, argCounter
    if kind == "ARG":
        return argCounter
    elif kind == "VAR":
        return varCounter
    elif kind == "FIELD":
        return fieldCounter
    elif kind == "STATIC":
        return staticCounter

#Returns the kind of the passed identifier, if it has
# already been stored in the table. Otherwise, returns "NONE".
def KindOf(name):
    global classTable, subroutineTable
    if name in subroutineTable:
        return subroutineTable[name].kind
    elif name in classTable:
        return classTable[name].kind
    else:
        return "NONE"
```

```
#Returns the type of the passed identifier.
def TypeOf(name):
    global classTable, subroutineTable
    if name in subroutineTable:
        return subroutineTable[name].type
    elif name in classTable:
        return classTable[name].type
    else:
        return "NONE"

#Returns the index of the passed identifier within
# its assigned memory segment.
def IndexOf(name):
    global classTable, subroutineTable
    if name in subroutineTable:
        return subroutineTable[name].index
    elif name in classTable:
        return classTable[name].index
    else:
        return "NONE"
```

2 Testing the Symbol Table Module

In [42]:
```
#After completing the symbol table, run this cell to test your code. If it
# Table is likely to be correct.

init()

Define("myvar", "String", "VAR")
Define("myvar", "int", "STATIC")
Define("yourvar", "String", "VAR")
Define("yourOtherVar", "myclass", "VAR")
Define("yourOtherOtherVar", "myclass", "ARG")
Define("thatVar", "String", "STATIC")
Define("thisVar", "String", "FIELD")

assert(KindOf("myvar") == "VAR")
assert(TypeOf("myvar") == "String")
assert(KindOf("thatVar") == "STATIC")
assert(TypeOf("yourOtherVar") == "myclass")

assert(IndexOf("myvar") == 0)
assert(IndexOf("yourvar") == 1)
assert(IndexOf("yourOtherVar") == 2)
assert(IndexOf("yourOtherOtherVar") == 0)
assert(IndexOf("thatVar") == 1)
```

```
assert(IndexOf("thisVar") == 0)

assert(VarCount("STATIC") == 2)
assert(VarCount("FIELD") == 1)
assert(VarCount("VAR") == 3)
assert(VarCount("ARG") == 1)

startSubroutine()
assert(VarCount("STATIC") == 2)
assert(VarCount("FIELD") == 1)
assert(VarCount("VAR") == 0)
assert(VarCount("ARG") == 0)

assert(KindOf("myvar") == "STATIC")
assert(IndexOf("myvar") == 0)
```

3 Writing the CodeWriter Module

In [43]:
```
#Writes a VM push command, with the provided
# segment and index. Segment will be either
# one of the 4 segments used in the Symbol Table Module
# (VAR, ARG, STATIC, or FIELD), or else will be a valid
# VM language segment (possibly in uppercase).
# If it is one of the 4 from the Symbol
# table module, you should translate it to its corresponding VM segment.
# If it is in upper case, you should translate it to lowercase with segmer
# Index will be a numeric value, but should be printed as a string.
def writePush(segment, index):
    if segment.upper() == "VAR":
        segment = 'local'
    elif segment.upper() == "ARG":
        segment = 'argument'
    elif segment.upper() == "STATIC":
        segment = 'static'
    elif segment.upper() == "FIELD":
        segment = 'this'
    print("push "+ segment +" "+ str(index))

#Writes a VM pop command, with the provided
# segment and index.
# segment and index are as described in writePush above.
def writePop(segment, index):
    if segment.upper() == "VAR":
        segment = 'local'
    elif segment.upper() == "ARG":
        segment = 'argument'
    elif segment.upper() == "STATIC":
```

```
            segment = 'static'
        elif segment.upper() == "FIELD":
            segment = 'this'
        print("pop "+ segment +" "+ str(index))

    #Writes a VM arithmetic command, based on
    # the provided command type. Command is one of:
    # ADD, SUB, NEG, EQ, GT, LT, AND, OR, NOT
    # Notice that these commands may need to be translated
    # to lowercase values with command.lower().
    def WriteArithmetic(command):
        print(command.lower())

    #Writes a VM label command, using the provided
    # label, which will be a valid VM language label name.
    def WriteLabel(label):
        print("label "+label)

    #Writes a VM GOTO command, using the provided label
    # which will be a valid VM language label name.
    def WriteGoto(label):
        print("goto "+label)

    #Writes a VM if-goto command, using the provided label.
    # which will be a valid VM language label name.
    def WriteIf(label):
        print("if-goto "+label)

    #Erites a VM call command, using the provided name and
    # number of arguments. name will be a valid VM function
    # name, and nArgs will be a number (not a string).
    def writeCall(name, nArgs):
        print("call "+name+" "+str(nArgs))

    #Writes a VM Function command with the provided name and
    # number of local variables. Name will be a valid VM function
    # name, and nArgs will be a number (not a string).
    def writeFunction(name, nLocals):
        print("function "+name+" "+str(nLocals))

    #Writes a VM return command.
    def writeReturn():
        print("return")
```

In [44]:
```
#After finishing your Code Writer Module, you can test it here. If the fo
# your Code Writer module is correct.
```

```
writePush("ARG", 5) #"push arg 5"
writePop("that", 3) #"pop that 3"
WriteArithmetic("NEG") #"neg"
WriteLabel("mylabel") #"label mylabel"
WriteGoto("mylabel") #"goto mylabel"
WriteIf("mylabel") #"if-goto mylabel"
writeCall("myfunc", 5) #"call myfunc 5"
writeFunction("myfunc", 4) #"function myfunc 4"
writeReturn() #"return"
```

4 Revising the Parser Module

In [45]:
```
#Program Structure Parsing Module
#This Module contains the parsing functions for
# the second box in Figure 10.5 of your textbook.
# The third and fourth boxes are handelled in the Statement Parsing Module

#Each function should either consume a non-terminal of the
# corresponding type, and write out the appropreate VM code,
# or should cause a compilerError to be produced, if the next
# token is not of the right type.

from Project11Util import peekToken, verifyToken, peekTokenList, verifyTok

currentClass = ""
currentSubroutine = ""
sizeOfClass = 0
whileLabelCounter = 0
ifLabelCounter=0

#Note: Classes don't need to produce any Jack code, but our compiler will
# remember how big an instance of this class is supposed to be, for use in
# constructors.
def CompileClass():
    global currentClass, sizeOfClass, whileLabelCounter, ifLabelCounter
    whileLabelCounter = 0
    ifLabelCounter = 0
    init() #zero the symbol table.
    verifyToken("keyword", "class")
    currentClass = verifyToken("identifier","")
    verifyToken("symbol", "{")

    while peekTokenList("keyword", ["static","field"]):
        sizeOfClass += CompileClassVarDec()

    while peekTokenList("keyword", ["constructor", "function", "method"]):
        CompileSubroutine()
```

```
    verifyToken("symbol", "}")

# This function should verify the grammar of a Type non-terminal, but also
# return the non-terminal itself.
def compileType():
    if peekTokenList("keyword", ["int", "char", "boolean"]):
        return verifyTokenList("keyword", ["int", "char", "boolean"])
    else:
        return verifyToken("identifier", "")

#Note: VM language has no notion of variables being declared before they a
# used. CompileClassVarDec should therefore simply populate the symbol tab
# with the newly named variables.
#Additionally: this functions should return the number of field variables
# the declaration.
def CompileClassVarDec():
    numVars = 0
    location = verifyTokenList("keyword", ["static","field"])
    varType = compileType()
    name = verifyToken("identifier", "")

    Define(name, varType, location)
    numVars = numVars + 1

    while peekToken("symbol", ","):
        verifyToken("symbol", ",")
        Define(verifyToken("identifier", ""), varType, location)
        numVars = numVars + 1
    verifyToken("symbol", ";")
    if location == 'static':
        return 0
    else:
        return numVars


# Recall: A Jack Subroutine is like a VM language function.
# However, there are 3 types. A constructor needs to allocate
# some new space for the new object. Use a call to the OS function
# Memory.alloc to accomplish this, and then set the "this" field
# to contain the address of the new block of memory.
# If this is a method, we need to make sure to set "this" to contain
# the value stored in the first argument to the function.
def CompileSubroutine():
    global sizeOfClass, currentClass
    funcType = verifyTokenList("keyword", ["constructor","function","metho
```

```
        returnType = ""
        if peekToken("keyword", "void"):
            returnType = verifyToken("keyword", "void")
        else:
            returnType = compileType()
        funcName = verifyToken("identifier", "")

        startSubroutine() #In the Symbol Table Module.

        verifyToken("symbol", "(")
        if funcType == "method":
            Define("this", "NONE", "ARG") #Make sure that THIS is the 0'th arg
        compileParameterList()
        verifyToken("symbol", ")")
        verifyToken("symbol", "{")
        while peekToken("keyword", "var"):
            compileVarDec()

        writeFunction(currentClass+"."+ funcName, VarCount("VAR"))

        if funcType == "constructor":
            writePush("constant", sizeOfClass)
            writeCall("Memory.alloc", 1) #allocate a new block big enough to h
            writePop('pointer', 0) #Make this point to the new block.
        if funcType == "method":
            writePush("argument", 0)
            writePop("pointer", 0) #if this is a method, set "this" to the fir

        compileStatements()
        verifyToken("symbol","}")

    #Variable declarations again, so just update the symbol table.
    def compileVarDec():
        verifyToken("keyword", "var")
        varType = compileType()
        name = verifyToken("identifier", "")
        Define(name, varType, 'VAR')
        while peekToken("symbol", ","):
            verifyToken("symbol", ",")
            name = verifyToken("identifier","")
            Define(name, varType, 'VAR')
        verifyToken("symbol", ";")

    #Variable declarations again, so just update the symbol table.
    def compileParameterList():
        if not peekToken("symbol", ")"):
            varType = compileType()
            name = verifyToken("identifier","")
```

```
                Define(name, varType, 'ARG')
                while peekToken("symbol", ","):
                    verifyToken("symbol", ",")
                    varType = compileType()
                    name = verifyToken("identifier", "")
                    Define(name, varType, 'ARG')
```

In [46]:
```
#The Statement Parser Module is responsiable
#for parsing Statements and Expressions, per
# the last two boxes of Figure 10.5.

#Each function should either consume a non-terminal of the
# corresponding type, and write out the appropreate VM Language code,
# or should cause a compilerError to be produced, if the next
# token is not of the right type.

def compileStatements():
    while peekTokenList("keyword", ["if", "let", "while", "do", "return"]):
        if peekToken("keyword", "if"):
            compileIf()
        elif peekToken("keyword", "let"):
            compileLet()
        elif peekToken("keyword", "while"):
            compileWhile()
        elif peekToken("keyword", "do"):
            compileDo()
        elif peekToken("keyword", "return"):
            compileReturn()

# A do statement is always a call to a void function or method.
# If it is a method, it will be of the form:
# do b.eat(f);
# whereas if it is a function, it will look like:
# do Screen.drawRectangle();
# In both cases, make sure to pop the 0 that gets returned by
# a void subroutine off the stack after the function returns.
# For a method call, make sure to push the address of the calling
# object (b in the example) as the first argument of the call.
def compileDo():
    verifyToken("keyword", "do")
    name = verifyToken("identifier", "")
    argCounter = 0
    if peekToken("symbol", "."):
        verifyToken("symbol", ".")
        if KindOf(name) != "NONE":
            writePush(KindOf(name), IndexOf(name))
            name = TypeOf(name)+"."+verifyToken("identifier","")
            argCounter = 1
```

```
            else:
                name = name +"." + verifyToken("identifier", "")
        else:
            writePush('pointer', 0) #implicit this.
            name = currentClass+"."+name
            argCounter = 1
        verifyToken("symbol", "(")
        argCounter = argCounter + CompileExpressionList()
        verifyToken("symbol", ")")
        writeCall(name, argCounter)
        verifyToken("symbol", ';')
        writePop('temp', 0) #Remove the result of a call, since it is unused.

    # A let statement is always used to mutate the value of a variable.
    # We must also handle "indirect addressing", conducted using square bracke
    # let x[5] = 8 should set the memory location *(x+5) to 8.
    def compileLet():
        verifyToken("keyword", "let")
        name = verifyToken("identifier","")

        isArray = False
        if peekToken("symbol", "["):
            isArray = True
            writePush(KindOf(name), IndexOf(name)) #Pushes the address stored
            verifyToken("symbol", "[")
            CompileExpression() #Generates code that computes the index.
            verifyToken("symbol", "]")
            WriteArithmetic('add')
        verifyToken("symbol", "=")
        CompileExpression()
        verifyToken("symbol", ";")


        if isArray:
            writePop('temp', 0)
            writePop('pointer', 1)
            writePush('temp', 0)
            writePop('that', 0)
        else:
            writePop(KindOf(name), IndexOf(name))


    #The while statement involves the creation of a unique VM label
    # using the global whileLabelCounter. Chapter 11 describes exactly
    # how this translation should occur.
    def compileWhile():
        global whileLabelCounter
        verifyToken("keyword", "while")
```

```
    L1 = currentClass+"WHILELABEL"+str(whileLabelCounter)
    whileLabelCounter = whileLabelCounter + 1
    L2 = currentClass+"WHILELABEL"+str(whileLabelCounter)
    whileLabelCounter = whileLabelCounter + 1

    WriteLabel(L1)

    verifyToken("symbol", "(")
    CompileExpression()
    WriteArithmetic('not')
    WriteIf(L2)
    verifyToken("symbol", ")")

    verifyToken("symbol","{")
    compileStatements()
    verifyToken("symbol", "}")

    WriteGoto(L1)
    WriteLabel(L2)

#A return must check whether it is returning a void value,
# or a non-void value. The void case looks like:
# return;
#while the non-void case looks like:
# return x;
# Void functions should return a 0 value, which is popped from the stack
# by the code from compileDo.
def compileReturn():
    verifyToken("keyword", "return")
    if not peekToken("symbol", ";"):
        CompileExpression()
    else:
        writePush('constant', 0)
    verifyToken("symbol", ";")
    writeReturn()

#If's work much like in VM to Hack translation, but with
# the added complication of an "else" keyword.
def compileIf():
    global ifLabelCounter
    verifyToken("keyword", "if")
    L1 = currentClass+"IFLABEL"+str(ifLabelCounter)
    ifLabelCounter = ifLabelCounter+1
    L2 = currentClass+"IFLABEL"+str(ifLabelCounter)
    ifLabelCounter = ifLabelCounter+2

    verifyToken("symbol", "(")
```

```
        CompileExpression()
        WriteArithmetic('not')
        WriteIf(L1)
        verifyToken("symbol", ")")
        verifyToken("symbol", "{")
        compileStatements()
        verifyToken("symbol", "}")
        WriteGoto(L2)
        WriteLabel(L1)
        if peekToken("keyword", "else"):
            verifyToken("keyword", "else")
            verifyToken("symbol", "{")
            compileStatements()
            verifyToken("symbol", "}")
        WriteLabel(L2)

    #Expressions should track the operator that is present, and then
    # compile both Terms before pushing the operation.
    def CompileExpression():
        CompileTerm()
        if peekTokenList("symbol", ["+","-","*","/","&","|","<",">","="]):
            op = verifyTokenList("symbol", ["+","-","*","/","&","|","<",">","=
            CompileTerm()
            if op == "+":
                WriteArithmetic("ADD")
            elif op == "-":
                WriteArithmetic("SUB")
            elif op == "*":
                writeCall('Math.multiply', 2) #hint: OS's Math file.
            elif op == "/":
                writeCall('Math.divide', 2)
            elif op == "&":
                WriteArithmetic("AND")
            elif op == "|":
                WriteArithmetic("OR")
            elif op == "<":
                WriteArithmetic("LT")
            elif op == ">":
                WriteArithmetic("GT")
            elif op == "=":
                WriteArithmetic("EQ")
            else:
                compileError("INVALID OP!!!", "INVALID OP!!!") #shouldn't be p

    #CompileTerm is the most complex subroutine we'll write in this course.
    #Comments throughout will provide guidence.
    def CompileTerm():
        global currentClass
```

```
        #If we see an integer, we can just push it to the stack...
        if peekToken("int_const", ""):
            writePush('constant', verifyToken("int_const", ""))
        #If we see a string, make a new string using String.new, fill it in
        # character by character, and then push the address of the new string
        # on top of the stack.
        elif peekToken("string_const", ""):
            token = verifyToken("string_const", "")
            writePush('constant', len(token))
            writeCall('String.new', 1)
            for c in token:
                writePush("constant", ord(c)) #ord: the numeric value of this
                writeCall('String.appendChar', 2) #Hint: use String OS file.

        #If we see true, false, push -1 or 0. If we see null, push 0. If we se
        # this, push the address stored in the THIS register.
        elif peekTokenList("keyword", ["true", "false", "null", "this"]):
            val = verifyTokenList("keyword", ["true", "false", "null", "this"]
            if val == "true":
                writePush('constant', 1)
                WriteArithmetic('neg')
            elif val == "false" or val == "null":
                writePush('constant', 0)
            elif val == "this":
                writePush('pointer', 0)

        #If we see an identifier next, there are a lot of cases...
        elif peekToken("identifier", ""):
            name = verifyToken("identifier", "")

            #Indirect addressing: We'll compute the correct final
            # address, and store it in the THAT register.
            if peekToken("symbol", "["):
                writePush(KindOf(name), IndexOf(name))
                verifyToken("symbol", "[")
                CompileExpression()
                verifyToken("symbol", "]")
                WriteArithmetic('add')
                writePop('pointer', 1)
                writePush("that", 0)

            #A method call: push this, then compile all the arguments
            # (pushing them) and count how many there are (hint: look at
            # the return values for CompileExpressionList() below). Then write
            # a call with that many arguments + 1.
            elif peekToken("symbol", "("):
                writePush('this', 0)
```

```
                verifyToken("symbol", "(")
                argCount = CompileExpressionList()
                verifyToken("symbol", ")")
                name = currentClass + "." + name
                writeCall(name, argCount + 1)

            #A function call, or a call to a method of a different type of
            # object than the current class.
            # Determine which by consulting the symbol table.
            # Then, push any needed arguments and write the call.
            elif peekToken("symbol", "."):
                verifyToken("symbol", ".")
                funcName = verifyToken("identifier", "")
                argCount = 0
                if KindOf(name) != "NONE":
                    writePush(KindOf(name), IndexOf(name))
                    funcName = TypeOf(name)+ "." + funcName
                    argCount = 1
                else:
                    funcName = name+"."+funcName
                verifyToken("symbol", "(")
                argCount = argCount + CompileExpressionList()
                verifyToken("symbol", ")")
                writeCall(funcName, argCount)

            #Otherwise, this is just the name of a variable. Push
            # the value of that variable by consulting the symbol table.
            else:
                writePush(KindOf(name), IndexOf(name))

        #If the next token was not an identifier, and it's a '(', then
        # someone is trying to write out an expression with prescidence.
        # Just compile the expression, and then close the parens.
        elif peekToken("symbol", "("):
                verifyToken("symbol", "(")
                CompileExpression()
                verifyToken("symbol", ")")

        #The last case is for uniary operators "~" and "-".
        else:
            op = verifyTokenList("symbol", ["-", "~"])
            CompileTerm()
            if op == "-":
                WriteArithmetic("NEG")
            else:
                WriteArithmetic("NOT")

    # Both compiles a list of expressions, and returns the total number of exp
```

```
        # in the list.
        def CompileExpressionList():
            argCounter = 0
            if not peekToken("symbol", ")"):
                CompileExpression()
                argCounter = argCounter + 1
                while peekToken("symbol", ","):
                    verifyToken("symbol", ",")
                    CompileExpression()
                    argCounter = argCounter + 1
            return argCounter
```

5 Testing our Parser

```
In [48]: import os
         import Project11IO as IO
         from Project11Util import advance

         def Compile(testname, filename):
             IO.setFile(os.path.join('..',testname,filename+'.Jack'))
             IO.setSaveFile(os.path.join('..',testname,filename+'.vm'))
             advance()
             CompileClass()

         Compile("Seven", "Main")
         Compile("ConvertToBin", "Main")


         Compile("Square", "Square")
         Compile("Square", "SquareGame")
         Compile("Square", "Main")

         Compile("Average", "Main")

         Compile("Pong", "Main")
         Compile("Pong", "Ball")
         Compile("Pong", "Bat")
         Compile("Pong", "PongGame")

         Compile("ComplexArrays", "Main")

         Compile("Game", "Main")
         Compile("Game", "Player")
         Compile("Game", "Bullet")
         Compile("Game", "SpaceWars")
         Compile("Game", "Enemy")
         Compile("Game", "EnemyBullet")
```