David Duffrin Project 10

November 3, 2016

1 Compiling Jack

In [18]:

```
# The Tokenizer Module
import Project10IO as IO


#Like the parser modules from earlier Projects, the Tokenizer
# will advance one token at a time, and populate these global
# variables.
nextType=""
nextInt=""
nextString=""
nextSymbol=""
nextKW=""
nextID=""

#A function to check whether there are more tokens.
def hasMoreTokens():
    global line
    if line == "EOF":
        return False
    else:
        return True

#Consumes characters until
# the next character in the stream is c.
#Returns a string containing all the characters
# that were consumed.
def eatUntil(c):
    s = ""
    while IO.peek() != c:
        s = s+IO.peek()
        IO.consume()
    return s

#This function should be called when
# the next token is a string. It consumes
# everything between the next two quotation marks, including the quotes.
# After the function is called, nextType should be STRING_CONST
# and nextSTring should be whatever was between the quotes.
def consumeString():
    global nextType, nextString
    s = ""
    IO.consume() #eat the first quote.
    while IO.peek() != '"':
        s = s + IO.peek()
        IO.consume()
    IO.consume() #eat the trailing quote.
    nextType = "STRING_CONST"
    nextString = s

#This function should be called when
# the next token is a integer. It consumes
# everything up to, but not including, the next non-numeric character.
# After the function is called, nextType should be INT_CONST
# and nextInt should be all the numbers that were consumed.
def consumeInt():
    global nextType, nextInt
    s = ""
    while IO.peek().isdigit():  #Hint: while the next character is a numbe
        s = s + IO.peek()
        IO.consume()
    nextType = "INT_CONST"
    nextInt = int(s)

#This function peeks at the next character to check whether the
# next token is a Symbol. A symbol is any character that isn't
# a number, letter, or underscore. The function returns True
# if the next token is a symbol, and False otherwise.
def isSymbol():
    s = IO.peek()
    if not IO.peek().isalnum() and s != "_":
        return True
    else:
        return False

#This function should be called when
# the next token is a symbol. It consumes only the next character.
# After the function is called, nextType should be SYMBOL
# and nextSymbol should be the consumed character.
def consumeSymbol():
    global nextSymbol, nextType
    nextType = "SYMBOL"
    nextSymbol = IO.peek()
    IO.consume()
```

```
#This function should be called when
# the next token is either a Keyword or an Identifier. It consumes
# everything up to the next character that is not a letter, number or _.
# After the function is called, if the consumed characters form a keyword,
# nextType should be KEYWORD, and nextKW should be equal to the consumed
# string. Otherwise, nextType should be IDENTIFIER, and nextID should be
# equal to the consumed string.
def consumeKWorID():
    global nextID, nextType, nextKW
    s = ""
    while IO.peek().isalnum() or IO.peek() == "_":
        s = s + IO.peek()
        IO.consume()

    KWlist = ['class', 'constructor', 'function', 'method', 'field', 'stat
              'var', 'int', 'char', 'boolean', 'void', 'true', 'false', 'r
              'this', 'let', 'do', 'if', 'else', 'while', 'return'] #List
    if s in KWlist:
        nextKW = s
        nextType = "KEYWORD"
    else:
        nextID = s
        nextType = "IDENTIFIER"


#When this function is called any whitespace before the next token is cons
#and then the next token is consumed, and the global variables are populat
# accordingly.
def advance():
    #Eat any leading whitespace or comments
    IO.eatUntilNextToken()

    #Use the first charactor to decide what to consume next.
    firstChar = IO.peek()
    if firstChar == '"':
        consumeString()
    elif firstChar.isdigit():
        consumeInt()
    elif not firstChar.isalpha():
        consumeSymbol()
    else:
        consumeKWorID()

#The remaining functions simply provide a way to access
# the populated global variables, and print error messages
# if they are accessed improperly.
def tokenType():
    global nextType
    return nextType

def keyword():
    global nextType, nextKW
    if nextType == "KEYWORD":
        return nextKW
    else:
        raise("Value error! Called keyWord() on line: " +IO.currentLine())

def symbol():
    global nextSymbol, nextType
    if nextType == "SYMBOL":
        return nextSymbol
    else:
        raise("Value error! Called symbol() on line: " +IO.currentLine())


def identifier():
    global nextID, nextType
    if nextType == "IDENTIFIER":
        return nextID
    else:
        raise("Value error! Called identifier() on line: " +IO.currentLine


def intVal():
    global nextInt, nextType
    if nextType == "INT_CONST":
        return str(nextInt)
    else:
        raise("Value error! Called intVal() on line: " +IO.currentLine())


def stringVal():
    global nextString, nextType
    if nextType == "STRING_CONST":
        return nextString
    else:
        raise("Value error! Called stringVal() on line: " +IO.currentLine
```

In [19]:

```
#Utilities Module
#This module contains a number of functions that will
# be exceedingly useful when we write the Parser.
#These functions do not need to be modified, and only the top four will
# be directly used in the next section. Make sure you understand what
# the top four do though, and how to use them.

#verifyToken accepts two arguments, ttype and token.
# If the next token is of type ttype, and has exactly
# the same value as the token argument, then the token is
# consumed, and printed using the printTerminal command.
# Otherwise, the program halts by calling compileError.
# The type is not case sensitive.
# Example: verifyToken("SYMBOL", ";")
# Example: verifyToken("keyword", "while")
def verifyToken(ttype, token="", peek=False):
    result = ""
    if tokenType() != ttype.upper() or (getNextToken() != token and token
        if not peek:
            compileError(ttype,token)
        return False
    else:
        if not peek:
            printTerminal(ttype.lower(), getNextToken())
            advance()
        return True

#peekToken returns true if the next token matches the
# type and value of the passed arguments. The token type
# is not case sensitive.
# Example: if peekToken("symbol", ";"):
#              //do something if the next token is ';'.
#          else
#              //do something if the next token is not ';'.
def peekToken(ttype, tvalue):
    return verifyToken(ttype, tvalue, peek=True)

#Like verifyToken, but tokenList should be a list of possible
# token values, and the program will halt with an error if
# the next token is _none_ of the listed values.
# Example: verifyTokenList("keyword", ["let", "do", "if", "while", "retur
def verifyTokenList(ttype, tokenList, peek=False):
    for t in tokenList:
        if peekToken(ttype, t):
            if not peek:
                printTerminal(ttype.lower(), getNextToken())
                advance()
            return True
    if not peek:
        compileError(ttype, tokenList)
    return False

#List peekToken, but returns true if _any_ of the values in
```

```
# tvalueList match the next token, and the next token is of the
# listed type.
def peekTokenList(ttype, tvalueList):
    return verifyTokenList(ttype, tvalueList, peek=True)


#Returns the next token, whatever it is.
def getNextToken():
    if tokenType() == "STRING_CONST":
        return stringVal()
    if tokenType() == "INT_CONST":
         return intVal()
    if tokenType() == "IDENTIFIER":
        return identifier()
    if tokenType() == "SYMBOL":
        return symbol()
    if tokenType() == "KEYWORD":
        return keyword()
    return "ERROR!!!"

#Halts the program and prints an error message.
def compileError( ttype, token):
    s = "Compilation error. Expected token \"" + token + "\" of type \"" +
    s = s + "Found token type \""+tokenType() + "\" with value \""
    s = s + getNextToken() + "\""
    raise ValueError(s)

#Prints a terminal value in the correct format
# <type> value </type>
def printTerminal(ttype, token):
    if token == "<":
        token = "&lt;"
    if token == ">":
        token = "&gt;"
    if token == "&":
        token = "&amp;"
    if ttype == "string_const":
        ttype = "stringConstant"
    if ttype == "int_const":
        ttype = "integerConstant"
    print("<"+ttype+"> "+token+" </"+ttype+">")
```

2 Writing the Parser Module

```
In [38]: #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 XML,
# or should cause a compilerError to be produced, if the next
# token is not of the right type.

def CompileClass(): # 'class' className '{' classVarDec* subroutineDec* '
    peekToken("keyword", "class")
    print("<class>") #We've made sure we're in a class, now we print the
    verifyToken("keyword", "class")
    verifyToken("identifier","")
    verifyToken("symbol", "{")


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

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

    verifyToken("symbol", "}")
    print("</class>")

def compileType():
    if peekTokenList("keyword", ['int','char','boolean']):
        verifyTokenList("keyword", ['int','char','boolean'])
    else:
        verifyToken("identifier", "")

def CompileClassVarDec(): # ('static' | 'field' ) type varName (',' varNa
    print("<classVarDec>")
    verifyTokenList("keyword", ['static','field'])

    compileType()
    verifyToken("identifier")
    while peekToken("symbol", ","):
        verifyToken("symbol", ",")
        verifyToken("identifier")

    verifyToken("symbol", ";")
    print("</classVarDec>")

def CompileSubroutine():
    print("<subroutineDec>")
    verifyTokenList("keyword", ['method','function','constructor']) # ('c
    if peekTokenList("keyword", ['void','int','char','boolean']): # ('voi
        verifyTokenList('keyword', ['void','int','char','boolean'])
    else:
        verifyToken("identifier")

    verifyToken("identifier") # subroutineName '(' parameterList ')' subro
    verifyToken("symbol", '(')
    compileParameterList()
    verifyToken("symbol", ')')
    compileSubroutineBody()

    print("</subroutineDec>")

def compileSubroutineBody(): # '{' varDec* statements '}'
    print("<subroutineBody>")
    verifyToken("symbol", "{")
    while peekToken('keyword','var'): #while var
        compileVarDec()
    compileStatements()
    verifyToken('symbol', "}")
    print("</subroutineBody>")

def compileVarDec(): # 'var' type varName (',' varName)* ';'
    print("<varDec>")
    verifyToken('keyword', 'var')
    compileType() # identifier could be defined or a name
    verifyToken("identifier")
    while peekToken("symbol", ","):
        verifyToken("symbol", ",")
        verifyToken("identifier")
    verifyToken("symbol",";")
    print("</varDec>")

def compileParameterList(): # ( (type varName) (',' type varName)*)?
    print("<parameterList>")
    if not peekToken('symbol', ')'):
        compileType()
        verifyToken("identifier")
        while peekToken('symbol', ','):
            verifyToken('symbol', ',')
            compileType()
            verifyToken('identifier')

    print("</parameterList>")
```

```
In [35]: #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 XML,
# or should cause a compilerError to be produced, if the next
# token is not of the right type.

def compileStatements():
    print("<statements>")
    while peekTokenList("keyword", ["let","if","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()
    print("</statements>")

def compileDo(): # 'do' subroutineCall ';'
    print("<doStatement>")
    verifyToken("keyword", "do")
    verifyToken("identifier") ### subroutineName '(' expressionList ')' |
    if peekToken("symbol", "."):
        verifyToken("symbol", ".")
        verifyToken("identifier")
    verifyToken("symbol", "(")
    CompileExpressionList()
    verifyToken("symbol", ")")

    verifyToken("symbol", ';')
    print("</doStatement>")

def compileLet(): # 'let' varName ('[' expression ']')? '=' expression ';
    print("<letStatement>")
    verifyToken("keyword", "let")
    verifyToken("identifier")

    if peekToken("symbol", "["):   #Case to handle arrays
        verifyToken("symbol", "[")
        CompileExpression()
        verifyToken("symbol", "]")
    verifyToken("symbol", "=")
    CompileExpression()
    verifyToken('symbol', ';')
    print("</letStatement>")

def compileWhile(): # 'while' '(' expression ')' '{' statements '}'
    print("<whileStatement>")
    verifyToken("keyword", "while")
    verifyToken("symbol", "(")
    CompileExpression()
    verifyToken("symbol", ")")
    verifyToken("symbol","{")
    compileStatements()
    verifyToken("symbol", "}")
    print("</whileStatement>")

def compileReturn(): # 'return' expression? ';'
    print("<returnStatement>")
    verifyToken("keyword", "return")
    if not peekToken("symbol", ";"):
        CompileExpression()

    verifyToken('symbol', ';')
    print("</returnStatement>")

def compileIf(): # 'if' '(' expression ')' '{' statements '}' ( 'else' '{
    print("<ifStatement>")
    verifyToken("keyword", "if")
    verifyToken("symbol", "(")
    CompileExpression()
    verifyToken("symbol", ")")
    verifyToken("symbol", "{")
    compileStatements()
    verifyToken("symbol", "}")

    if peekToken("keyword", "else"):
        verifyToken("keyword", "else")
        verifyToken("symbol", "{")
        compileStatements()
        verifyToken("symbol", "}")
    print("</ifStatement>")

def CompileExpression(): # term (op term)*
    print("<expression>")
    CompileTerm()
    if peekTokenList("symbol", ["+","-","*","/","&","|","<",">","="]):
        verifyTokenList("symbol", ["+","-","*","/","&","|","<",">","="])
        CompileTerm()
    print("</expression>")

def CompileTerm(): #integerConstant | stringConstant | keywordConstant |
    print("<term>")
    if peekToken("INT_CONST", ""):
        verifyToken("int_const", "")
    elif peekToken("STRING_CONST", ""):
        verifyToken("STRING_CONST", "")
    elif peekTokenList("keyword", ['true', 'false', 'null', 'this']):
        verifyTokenList("keyword", ['true', 'false', 'null', 'this'])
    elif peekToken("identifier", ""):  #If we see a variable name...
        verifyToken("identifier", "")
        if peekToken("symbol", "["):  #Case for arrays
            verifyToken("symbol", "[")
            CompileExpression()
            verifyToken("symbol", "]")
        elif peekToken("symbol", "("): #Case for a function call...
            verifyToken("symbol", "(") # '(' expressionList ')'
            CompileExpressionList()
            verifyToken("symbol", ")")
        elif peekToken("symbol", "."):  #Case for, e.g. Butterfly.eat()
            verifyToken("symbol", ".") # '.' subroutineName '(' expressio
            verifyToken("identifier")
            verifyToken("symbol", "(")
            CompileExpressionList()
            verifyToken("symbol", ")")
    elif peekToken("symbol", "("): #If we didn't see a variable name, but
            verifyToken("symbol", "(")
            CompileExpression()
            verifyToken("symbol", ")")
    else:
        verifyTokenList("symbol", ["-", "~"])
        CompileTerm()
    print("</term>")

def CompileExpressionList():
    print("<expressionList>")
    if not peekToken("symbol", ")"):
        CompileExpression()
        while peekToken("symbol", ","):
            verifyToken("symbol", ",")
            CompileExpression()
    print("</expressionList>")
```

3 Testing our Parser

```
In [42]: import os

         def Compile(testname, filename):
             IO.setFile(os.path.join('..',testname,filename+'.Jack'))
```

```
        IO.setSaveFile(os.path.join('..',testname,filename+'.out.xml'))
        advance()
        CompileClass()

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

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

    Compile("ArrayTest", "Main")


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

In [ ]: