**COVER PAGE**

**CS323 Programming Assignments**

1. Names [ 1. Vanessa Delfin ], (4pm class [ ] or 5:30pm class [ x ] )

[ 2. Navie Vurdien ], (4pm class [ ] or 5:30pm class [ x ] )

2. Assignment Number [ 3 ]

3. Due Dates **Softcopy**  [ 05/07/2018 ], **Hardcopy** [ 05/08/2018 ]

4. Turn-In Dates **Softcopy** [ 05/07/2018 ], **Hardcopy** [ 05/08/2018 ]

5. Executable FileName [assignment3python3.6.py]

6. LabRoom [CS-200]

7. Operating System [Windows]

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**To be filled out by the Instructor:**

GRADE:

COMMENTS:

**CS323 Documentation Assignment 3**

**1. Problem Statement**

The main purpose of third part of the project is that is simplifies RAT18S language. <Function Definitions> and type “real” was removed.

Generate assembly instructions

Symbol table

Tool

execution

**2. How to use your program**

In the CS200 lab, use the command prompt and run the following command in the correct directory:

python assignment3python3.6.py

**3. Design of your program**

Symbol table

Name type mem loc

Num int 2000

**4. Any Limitation**

None

**5. Any shortcomings**

None

**Source Code:**

# works with python3.6  
  
from enum import Enum  
import string  
  
# %% start of lexer  
keyword = ['while', 'if', 'for', 'else', 'get', 'int', 'endif', 'return', 'put', 'function', 'real', 'boolean']  
  
separator = ['(', ')', '[', ']', '{', '}', '!', ';', ',', ':']  
  
regex = ['==', '^=', '>', '<', '=>', '<=']  
  
  
class State(Enum):  
 REJECT = 0, 'REJECT'  
 INTEGER = 1, 'INTEGER'  
 REAL = 2, 'REAL'  
 OPERATOR = 3, 'OPERATOR'  
 SEPARATOR = 4, 'SEPARATOR'  
 IDENTIFIER = 5, 'IDENTIFIER'  
 KEYWORD = 6, 'KEYWORD'  
 UNKNOWN = 7, 'UNKNOWN'  
 SPACE = 8, 'SPACE'  
 BOOLEAN = 9, 'BOOLEAN'  
  
 def \_\_new\_\_(cls, value, name):  
 member = object.\_\_new\_\_(cls)  
 member.\_value\_ = value  
 member.fullname = name  
 return member  
  
 def \_\_int\_\_(self):  
 return self.value  
  
  
# still working on the states   
stateTable = [  
 [0, State.INTEGER, State.REAL, State.OPERATOR, State.SEPARATOR, State.IDENTIFIER, State.KEYWORD, State.UNKNOWN,  
 State.SPACE],  
 [State.INTEGER, State.INTEGER, State.REAL, State.REJECT, State.REJECT, State.IDENTIFIER, State.REJECT, State.REJECT,  
 State.REJECT],  
 [State.REAL, State.REAL, State.UNKNOWN, State.REJECT, State.REJECT, State.REJECT, State.REJECT, State.REJECT,  
 State.REJECT],  
 [State.OPERATOR, State.REJECT, State.REJECT, State.OPERATOR, State.REJECT, State.REJECT, State.REJECT, State.REJECT,  
 State.REJECT],  
 [State.SEPARATOR, State.REJECT, State.REJECT, State.REJECT, State.REJECT, State.REJECT, State.REJECT, State.REJECT,  
 State.REJECT],  
 [State.IDENTIFIER, State.IDENTIFIER, State.REJECT, State.REJECT, State.REJECT, State.IDENTIFIER, State.REJECT,  
 State.REJECT, State.REJECT],  
 [State.KEYWORD, State.REJECT, State.REJECT, State.REJECT, State.REJECT, State.REJECT, State.REJECT, State.REJECT,  
 State.REJECT],  
 [State.UNKNOWN, State.UNKNOWN, State.UNKNOWN, State.UNKNOWN, State.UNKNOWN, State.UNKNOWN, State.UNKNOWN,  
 State.UNKNOWN, State.REJECT],  
 [State.SPACE, State.REJECT, State.REJECT, State.REJECT, State.REJECT, State.REJECT, State.REJECT, State.REJECT,  
 State.REJECT],  
]  
  
  
def checkToken(token):  
 # print("Check Token", token)  
 if token.isdigit():  
 # print("is digit")  
 return State.INTEGER  
 elif token.isspace():  
 # print("is space")  
 return State.SPACE  
 elif token == '$':  
 return State.IDENTIFIER  
 elif token == '.':  
 # print("is real")  
 return State.REAL  
 for i in string.punctuation:  
 if token == i:  
 if token in separator:  
 # print("is separator")  
 return State.SEPARATOR  
 # print("is punc")  
 return State.OPERATOR  
 if token.isalpha():  
 # print("is alpha")  
 return State.IDENTIFIER  
 else:  
 # print("is unknown")  
 return State.UNKNOWN  
  
  
comment = False  
past\_line = 0  
  
  
def Lexer(expression, line\_num):  
 global col, comment  
 tokens = []  
 col = State.REJECT  
 currentState = State.REJECT  
 prevState = State.REJECT  
 currentToken = ""  
 for token in expression:  
 col = checkToken(token)  
 currentState = stateTable[int(currentState)][int(col)]  
 # print("currentToken is ", currentToken, "currentState is ", currentState.fullname)  
 if currentState == State.REJECT:  
 currentToken = currentToken.replace(" ", "")  
 if prevState != State.SPACE and currentToken:  
 if currentToken == '!':  
 if comment:  
 comment = False  
 else:  
 comment = True  
 elif not comment:  
 if currentToken in keyword:  
 tokens.append({'token': currentToken, 'lexeme': State.KEYWORD, 'line\_num': line\_num})  
 elif prevState == State.IDENTIFIER and (currentToken[len(currentToken) - 1].isdigit()):  
 tokens.append({'token': currentToken, 'lexeme': State.UNKNOWN, 'line\_num': line\_num})  
 elif currentToken == '%%':  
 tokens.append({'token': currentToken, 'lexeme': State.SEPARATOR, 'line\_num': line\_num})  
 elif prevState == State.REAL and (  
 currentToken.index(".") == 0 or currentToken.index(".") == len(currentToken) - 1):  
 tokens.append({'token': currentToken, 'lexeme': State.UNKNOWN, 'line\_num': line\_num})  
 elif currentToken == 'true' or currentToken == 'false':  
 tokens.append({'token': currentToken, 'lexeme': State.BOOLEAN, 'line\_num': line\_num})  
 else:  
 tokens.append({'token': currentToken, 'lexeme': prevState, 'line\_num': line\_num})  
 currentToken = token  
 currentState = checkToken(token)  
 else:  
 currentToken = currentToken.replace(" ", "")  
 currentToken += token  
 prevState = currentState  
 if currentState != State.SPACE and currentToken and currentState != State.REJECT:  
 tokens.append({'token': currentToken, 'lexeme': prevState, 'line\_num': line\_num})  
 return tokens  
  
  
# %% end of lexer  
  
# %% start of syntaxanalyzer  
def printToken(fn, index, output):  
 print("Token: ", fn[index]['lexeme'], "\tLexeme:", fn[index]['token'], file=output)  
  
  
def I(fn, index, output):  
 if fn[index]['token'] == 'if':  
 addr = len(assemblyStack)  
 index += 1  
 if fn[index]['token'] == '(':  
 index += 1  
 index = C(fn, index, output)  
 if fn[index]['token'] == ')':  
 index += 1  
 index = S(fn, index, output)  
 back\_patch(len(assemblyStack))  
 if fn[index]['token'] == 'else':  
 index += 1  
 index = S(fn, index, output)  
 if fn[index]['token'] == 'endif':  
 return index + 1  
 else:  
 print("endif expected at line ", fn[index]['line\_num'], file=output)  
 else:  
 print(") expected at line ", fn[index]['line\_num'], file=output)  
 else:  
 print(") expected at line ", fn[index]['line\_num'], file=output)  
 else:  
 print("if expected at line ", fn[index]['line\_num'], file=output)  
  
  
def S(fn, index, output):  
 if fn[index]['token'] == '{':  
 index += 1  
 while index < len(fn) and fn[index]['token'] != '}':  
 index = S(fn, index, output)  
 if index > len(fn):  
 print("expected } at line ", fn[len(fn) - 1]['line\_num'], file=output)  
 else:  
 return index + 1  
 elif fn[index]['token'] == 'int':  
 index += 1  
 while fn[index]['lexeme'] == State.IDENTIFIER:  
 if fn[index]['token'] not in symbolTable:  
 symbolTable[fn[index]['token']] = [fn[index]['token'], 2000 + len(symbolTable), State.IDENTIFIER, None]  
 else:  
 print("identifier reinitialization at line ", fn[index]['line\_num'], file=output)  
 return None  
 index += 1  
 if fn[index]['token'] == ',':  
 index += 1  
 elif fn[index]['token'] == ';':  
 return index + 1  
 elif fn[index]['token'] == 'if':  
 return I(fn, index, output)  
 elif fn[index]['token'] == 'while':  
 return whileStatements(fn, index, output)  
 elif fn[index]['token'] == 'put':  
 index += 1  
 if fn[index]['token'] == '(':  
 index += 1  
 if fn[index]['lexeme'] in [State.IDENTIFIER, State.INTEGER]:  
 if fn[index]['lexeme'] == State.IDENTIFIER:  
 index = E(fn, index, fn[index]['token'], output)  
 else:  
 index = E(fn, index, "", output)  
 gen\_instr("STDOUT", "")  
 if fn[index]['token'] == ')':  
 index += 1  
 if fn[index]['token'] == ';':  
 return index + 1  
 else:  
 print("; expected at line ", fn[index]['line\_num'], file=output)  
 else:  
 print(") expected at line ", fn[index]['line\_num'], file=output)  
 else:  
 print("identifier or integer expected at line ", fn[index]['line\_num'], file=output)  
 else:  
 print("( expected at line ", fn[index]['line\_num'], file=output)  
 elif fn[index]['token'] == 'get':  
 index += 1  
 if fn[index]['token'] == '(':  
 index += 1  
 while index < len(fn) and fn[index]['lexeme'] == State.IDENTIFIER:  
 gen\_instr("STDIN", "")  
 gen\_instr("POPM", symbolTable[fn[index]['token']][1])  
 index += 1  
 if fn[index]['token'] == ',':  
 index += 1  
 if fn[index]['token'] == ')':  
 index += 1  
 if fn[index]['token'] == ';':  
 return index + 1  
 else:  
 print("; expected at line ", fn[index]['line\_num'], file=output)  
 else:  
 print(") expected at line ", fn[index]['line\_num'], file=output)  
 else:  
 print("( expected at line ", fn[index]['line\_num'], file=output)  
 elif fn[index]['lexeme'] == State.IDENTIFIER or fn[index]['lexeme'] == State.INTEGER:  
 if index+1 < len(fn) and fn[index+1]['token'] == '=':  
 return A(fn, index, output)  
 index = E(fn, index, "", output)  
 if fn[index]['token'] == ';':  
 return index + 1  
 elif fn[index]['token'] == '%%':  
 return S(fn, index+1, output)  
 else:  
 print("identifier, if, while, put or get expected at line ", fn[index]['line\_num'], file=output)  
  
  
def C(fn, index, output):  
 if fn[index]['lexeme'] == State.IDENTIFIER:  
 index = E(fn, index, fn[index]['token'], output)  
 else:  
 index = E(fn, index, "", output)  
 if fn[index]['token'] in regex:  
 op = fn[index]['token']  
 index += 1  
 if fn[index]['lexeme'] == State.IDENTIFIER:  
 index = E(fn, index, fn[index]['token'], output)  
 else:  
 index = E(fn, index, "", output)  
 if op == '<':  
 gen\_instr("LES", "")  
 jumpStack.append(len(assemblyStack))  
 gen\_instr("JUMPZ", "")  
 elif op == '>':  
 gen\_instr("GRT", "")  
 jumpStack.append(len(assemblyStack))  
 gen\_instr("JUMPZ", "")  
 elif op == '^=':  
 gen\_instr("NEQ", "")  
 jumpStack.append(len(assemblyStack))  
 gen\_instr("JUMPZ", "")  
 elif op == '==':  
 gen\_instr("EQU", "")  
 jumpStack.append(len(assemblyStack))  
 gen\_instr("JUMPZ", "")  
 elif op == '=>':  
 gen\_instr("GEQ", "")  
 jumpStack.append(len(assemblyStack))  
 gen\_instr("JUMPZ", "")  
 elif op == '<=':  
 gen\_instr("LEQ", "")  
 jumpStack.append(len(assemblyStack))  
 gen\_instr("JUMPZ", "")  
 return index  
 else:  
 print("boolean operator expected at line ", fn[index]['line\_num'], file=output)  
  
  
def whileStatements(fn, index, output):  
 if fn[index]['token'] == 'while':  
 addr = len(assemblyStack)  
 gen\_instr("LABEL", "")  
 index += 1  
 if fn[index]['token'] == '(':  
 index += 1  
 index = C(fn, index, output)  
 if fn[index]['token'] == ')':  
 index += 1  
 index = S(fn, index, output)  
 gen\_instr("JUMP", addr + 1)  
 back\_patch(len(assemblyStack))  
 return index  
 else:  
 print(") expected at line ", fn[index]['line\_num'], file=output)  
 else:  
 print("( expected at line ", fn[index]['line\_num'], file=output)  
 else:  
 print("while expected at line ", fn[index]['line\_num'], file=output)  
  
  
def F(fn, index, save, output):  
 global symbolTable  
 if fn[index]['lexeme'] == State.IDENTIFIER:  
 gen\_instr("PUSHM", symbolTable[fn[index]['token']][1])  
 return index + 1  
 elif fn[index]['lexeme'] == State.INTEGER:  
 gen\_instr("PUSHI", fn[index]['token'])  
 return index + 1  
 else:  
 print("identifier expected at line ", fn[index]['line\_num'], file=output)  
  
  
def T\_prime(fn, index, save, output):  
 if fn[index]['token'] == '/':  
 index += 1  
 index = F(fn, index, save, output)  
 gen\_instr("DIV", "")  
 return T\_prime(fn, index, save, output)  
 elif fn[index]['token'] == "\*":  
 index += 1  
 index = F(fn, index, save, output)  
 gen\_instr("MUL", "")  
 return T\_prime(fn, index, save, output)  
 return index  
  
  
def T(fn, index, save, output):  
 index = F(fn, index, save, output)  
 return T\_prime(fn, index, save, output)  
  
  
def E\_prime(fn, index, save, output):  
 if fn[index]['token'] == '-':  
 index += 1  
 index = T(fn, index, save, output)  
 gen\_instr("SUB", "")  
 return E\_prime(fn, index, save, output)  
 elif fn[index]['token'] == '+':  
 index += 1  
 index = T(fn, index, save, output)  
 gen\_instr("ADD", "")  
 return E\_prime(fn, index, save, output)  
 return index  
  
  
def E(fn, index, save, output):  
 index = T(fn, index, save, output)  
 return E\_prime(fn, index, save, output)  
  
  
def A(fn, index, output):  
 if fn[index]['lexeme'] == State.IDENTIFIER:  
 save = fn[index]['token']  
 index += 1  
 if fn[index]['token'] == '=':  
 index += 1  
 if fn[index]['lexeme'] == State.IDENTIFIER:  
 index = E(fn, index, fn[index]['token'], output)  
 else:  
 index = E(fn, index, State.INTEGER, output)  
 gen\_instr("POPM", symbolTable[save][1])  
 if fn[index]['token'] == ';':  
 return index + 1  
 else:  
 print(" = expected at line ", fn[index]["line\_num"], file=output)  
 else:  
 print("identifier expected at line ", fn[index]['line\_num'], file=output)  
 return index  
  
  
def syntaxAnalyzer(fn, output):  
 global past\_line  
 index = 0  
 while index < len(fn):  
 index = S(fn, index, output)  
 if index is None:  
 break  
 return True  
  
  
# end of syntaxAnalyzer  
  
  
memLoc = 2000  
#  
# Symbol Table Structure Example  
# {  
# "A": [  
# tokenItself ("A"),  
# memoryLocation starts at 2000,  
# Value type (State.IDENTIFIER),  
# Value (int, real, boolean)  
# ]  
# }  
#  
# symbolTable["A"][0] = (identifier string)  
# symbolTable["A"][1] = (memorylocation int)  
# symbolTable["A"][2] = (currentState State.\_\_\_)  
# symbolTable["A"][3] = (value of object)  
#  
symbolTable = {}  
#  
# Assembly Stack Structure Example  
# [  
# {  
# address: (1),  
# operation: "",  
# operand: ""  
# }  
# ]  
#  
# assemblyStack[0]["address"] = (address int)  
# assemblyStack[0]["operation"] = (operation string)  
# assemblyStack[0]["operand"] = (address string (can be nil))  
#  
assemblyStack = []  
#  
# Jump stack  
#  
jumpStack = []  
  
  
def back\_patch(jump\_addr):  
 addr = jumpStack[len(jumpStack) - 1]  
 assemblyStack[addr]["operand"] = jump\_addr + 1  
  
  
def gen\_instr(op, oprnd):  
 global assemblyStack  
 instruction = {"address": len(assemblyStack), "operation": op, "operand": oprnd}  
 assemblyStack.append(instruction)  
  
  
# outputs assemblystack  
def assemblyCode(output):  
 for code in assemblyStack:  
 print(code["address"] + 1, "\t", code["operation"], "\t", code["operand"], file=output)  
  
  
filename = input('Enter a input filename: ')  
  
results = []  
line\_n = 1  
  
with open(filename) as inputfile:  
 for line in inputfile:  
 results += Lexer(line, line\_n)  
 line\_n += 1  
  
filename = input('Enter a output filename: ')  
try:  
 with open(filename, "w+") as outputfile:  
 syntaxAnalyzer(results, outputfile)  
 assemblyCode(outputfile)  
except KeyError or TypeError:  
 print(symbolTable)  
 print(assemblyStack)

**Test Cases**

**Test Case1**

! Find largest value between two numbers!  
int num, nu2m, large$;  
num = 0;  
nu2m = 15;  
if(num > nu2m)  
{  
 large$ = num;  
}  
else  
{  
 large$ = nu2m;  
}  
endif

**Test Case2**

! this is comment for this sample code which  
 mid$size  
 converts Fahrenheit into Celcius !  
  
%%  
 int low, high, step$; ! declarations !  
  
 get (low, high, step$);  
 while (low < high )   
 { put (low);  
 low = low + step$;  
 }

**Test Case3**

!Find area of a circle!  
  
%%  
  
int choose$; ! Choose to use radius or diameter !  
int radius, diameter, area$;  
  
get(choose$);  
get(radius, diameter, area$);  
  
if(choose$ == 1)  
{  
 put(radius);  
}  
else  
{  
 if(choose$ == 2)  
 {  
 put(radius + diameter);  
 }  
 endif  
}  
endif

**Test Case4**

%%  
int i, max, sum;  
  
sum=0;  
i=1;  
get(max);  
while(i<max){  
sum = sum + i;  
i = i + 1;  
}  
put(sum + max);

**Output Test Case**

**#1**

1 PUSHI 0  
2 POPM 2000  
3 PUSHI 15  
4 POPM 2001  
5 PUSHM 2000  
6 PUSHM 2001  
7 GRT   
8 JUMPZ 11  
9 PUSHM 2000  
10 POPM 2002  
11 PUSHM 2001  
12 POPM 2002

**#2**

1 STDIN   
2 POPM 2000  
3 STDIN   
4 POPM 2001  
5 STDIN   
6 POPM 2002  
7 LABEL   
8 PUSHM 2000  
9 PUSHM 2001  
10 LES   
11 JUMPZ 19  
12 PUSHM 2000  
13 STDOUT   
14 PUSHM 2000  
15 PUSHM 2002  
16 ADD   
17 POPM 2000  
18 JUMP 7

**#3**

1 STDIN   
2 POPM 2000  
3 STDIN   
4 POPM 2001  
5 STDIN   
6 POPM 2002  
7 STDIN   
8 POPM 2003  
9 PUSHM 2000  
10 PUSHI 1  
11 EQU   
12 JUMPZ 15  
13 PUSHM 2001  
14 STDOUT   
15 PUSHM 2000  
16 PUSHI 2  
17 EQU   
18 JUMPZ 23  
19 PUSHM 2001  
20 PUSHM 2002  
21 ADD   
22 STDOUT

**#4**

1 PUSHI 0  
2 POPM 2002  
3 PUSHI 1  
4 POPM 2000  
5 STDIN   
6 POPM 2001  
7 LABEL   
8 PUSHM 2000  
9 PUSHM 2001  
10 LES   
11 JUMPZ 21  
12 PUSHM 2002  
13 PUSHM 2000  
14 ADD   
15 POPM 2002  
16 PUSHM 2000  
17 PUSHI 1  
18 ADD   
19 POPM 2000  
20 JUMP 7  
21 PUSHM 2002  
22 PUSHM 2001  
23 ADD   
24 STDOUT