**CS.211 Assignment**

**Nouran khalid 200048219**

*1-Introduction*

* 1. *phases of compiler*

phases: A compiler operates in multiple phases, each responsible for a different aspect of translation.

Phase Description

Lexical Analysis Converts source code into tokens.

Syntax Analysis Checks grammatical correctness and builds a parse tree.

Semantic Analysis Ensures valid meaning and detects type errors.

Intermediate Code Generation Produces an intermediate representation.

Code Optimization Improves performance by reducing redundancies.

Code Generation Converts optimized code to machine code.

Error Handling Identifies and reports errors.

***2- Lexical Analyzer***

A Lexical Analyzer: reads the source code character by character and groups them into meaningful tokens.

***3-Software Tools***

* 1. ***Computer Program***

Visual studio

* 1. ***Programming Language***

python

1. ***Implementation of a Lexical Analyzer (explanation is in red)***

LETTER = 0

DIGIT = 1

UNKNOWN = 99

Defining

LETTER is for the alphabet letters

DIGIT is for the digit numbers

UNKNOWN is for symbols

INT\_LIT = 10 integer number

IDENT = 11 Identifier

ASSIGN\_OP = 20 Assignment operator

ADD\_OP = 21 +

SUB\_OP = 22 -

Token types for the lexer so it will recognize and return

MULT\_OP = 23 \*

DIV\_OP = 24 /

LEFT\_PAREN = 25 (

RIGHT\_PAREN = 26 )

charClass = None type of character being read

lexeme = '' String built from characters

nextChar = '' The current character getting read

lexLen = 0 Length of the lexeme so far

nextToken = None The current token type

in\_fp = None File pointer for input

def addChar():

global lexeme, lexLen

Adds the current nextChar to the lexeme string. And limiting the lexeme length to 99 characters

if lexLen <= 98:

lexeme += nextChar

lexLen += 1

else:

print("Error - lexeme is too long")

def getChar():

global nextChar, charClass

nextChar = in\_fp.read(1) Read a single character

if nextChar: to classifies as a LETTER, DIGIT, UNKNOWN, EOF

if nextChar.isalpha():

charClass = LETTER

elif nextChar.isdigit():

charClass = DIGIT

else:

charClass = UNKNOWN

else:

charClass = EOF

def getNonBlank(): calls getChar() until it finds a non blank character

global nextChar

while nextChar.isspace(): Skip spaces

getChar()

def lookup(ch):

global nextToken

lookup\_table = {

'(': LEFT\_PAREN,

')': RIGHT\_PAREN,

'+': ADD\_OP,

'-': SUB\_OP,

checking if the character is an operator or parenthesis.

If it is assigns the token code

If not found return EOF

'\*': MULT\_OP,

'/': DIV\_OP

}

addChar()

nextToken = lookup\_table.get(ch, EOF)

return nextToken

def lex():

Clear the past lexeme and Skips over blank spaces to find the next real character

global lexeme, lexLen, nextToken

lexeme = ''

lexLen = 0

getNonBlank()

If the character is a letter it starts building an identifier and Keeps adding letters or digits to the lexeme

It Ends when a non-letter or digit is found

if charClass == LETTER:

addChar()

getChar()

while charClass in (LETTER, DIGIT):

addChar()

getChar()

nextToken = IDENT

elif charClass == DIGIT:

addChar()

getChar()

If it's a digit it builds an integer literal

while charClass == DIGIT:

addChar()

getChar()

nextToken = INT\_LIT

elif charClass == UNKNOWN:

If it's a symbol it looks it up and assigns the correct token

lookup(nextChar)

getChar()

If we reach the end of the file return EOF

elif charClass == EOF:

nextToken = EOF

lexeme = 'EOF'

print(f"Next token is: {nextToken}, Next lexeme is {lexeme}")

Outputs the result and returns the token type

return nextToken

def main():

global in\_fp

try:

with open("front.in", "r") as in\_fp\_obj:

Opens the file front.in and Calls getChar() once to initialize and Keeps calling lex() until it returns EOF

global in\_fp

in\_fp = in\_fp\_obj

getChar()

while True:

if lex() == EOF:

break

except FileNotFoundError:

print("ERROR - cannot open front.in") Error message if file doesn’t exist