A black background with grey leaves

AI-generated content may be incorrect.

Lexical Analyzer

Build Scanner

A grey logo on a black background

AI-generated content may be incorrect.

**Prepared By**

Student Name : Arwa Hamada Student ID : 200038818

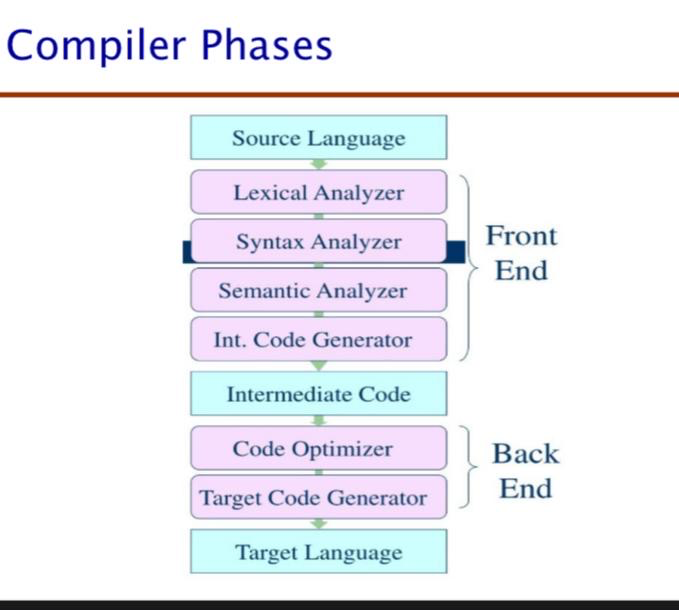
**Under Supervision**

Name of Doctor : Nehal

Name of T.A. : Eng.Fares

1. **Introduction**

A compiler is a specialized software program that translates code written in a high-level programming language into a lower-level language, often machine code, which can be executed by a computer. This process involves multiple stages, commonly referred to as the phases of a compiler. These phases work in sequence to analyze, transform, and generate optimized code.

* 1. **Phases of Compiler**

The main phases include:

1. Lexical Analysis: This phase reads the source code and converts it into tokens. It removes whitespaces and comments and groups characters into meaningful lexemes.
2. Syntax Analysis (Parsing): It checks the syntax of the program using grammar rules and builds a syntax tree that represents the structure of the source code.
3. Semantic Analysis: This phase ensures that the program follows the rules of the language (such as type checking) and generates an annotated syntax tree.
4. Intermediate Code Generation: The compiler translates the syntax tree into an intermediate representation that is easier to analyze and optimize.
5. Code Optimization: It improves the intermediate code to make it more efficient without changing its output.
6. Code Generation: Converts the optimized intermediate code into machine code specific to the target hardware.
7. Code Linking and Assembly: The final machine code is linked with libraries and assembled into an executable file.
8. **Lexical Analyzer**

A lexical analyzer is the first phase of a compiler. It reads the source code character by character and groups the characters into meaningful sequences called "tokens." These tokens represent the basic building blocks of the programming language, such as keywords, identifiers, operators, and constants.

1. **Software Tools**

Software tools are applications or systems used to create, maintain, and manage software. These tools can vary widely in purpose, from aiding in the development of programs to providing debugging and performance analysis. Software tools include:

Integrated Development Environments (IDEs): These are comprehensive tools used for writing, testing, and debugging code (e.g., Visual Studio, Eclipse).

Compilers and Interpreters: These tools translate code written in high-level programming languages into machine code or another intermediate language (e.g., GCC for C++, Python interpreter).

Version Control Systems: These are tools for managing and tracking changes to code over time (e.g., Git, SVN).

Debuggers: Tools for identifying and fixing errors in code (e.g., GDB, Xcode debugger).

Build Systems: These tools automate the process of compiling and linking code (e.g., Make, CMake).

* 1. **Computer Program**

A computer program is a set of instructions or commands that a computer follows to perform a specific task or solve a problem. Programs are written using programming languages and are executed by a computer’s processor. Examples include applications, operating systems, utilities, and games. Programs are made up of code that tells the computer what actions to perform, such as calculations, input/output operations, or interacting with hardware.

* 1. **Programming Language**

A programming language is a formal language used to write instructions that a computer can understand and execute. Programming languages provide syntax (rules for structure) and semantics (meaning of instructions) that developers follow to communicate with computers. Examples of programming languages include:

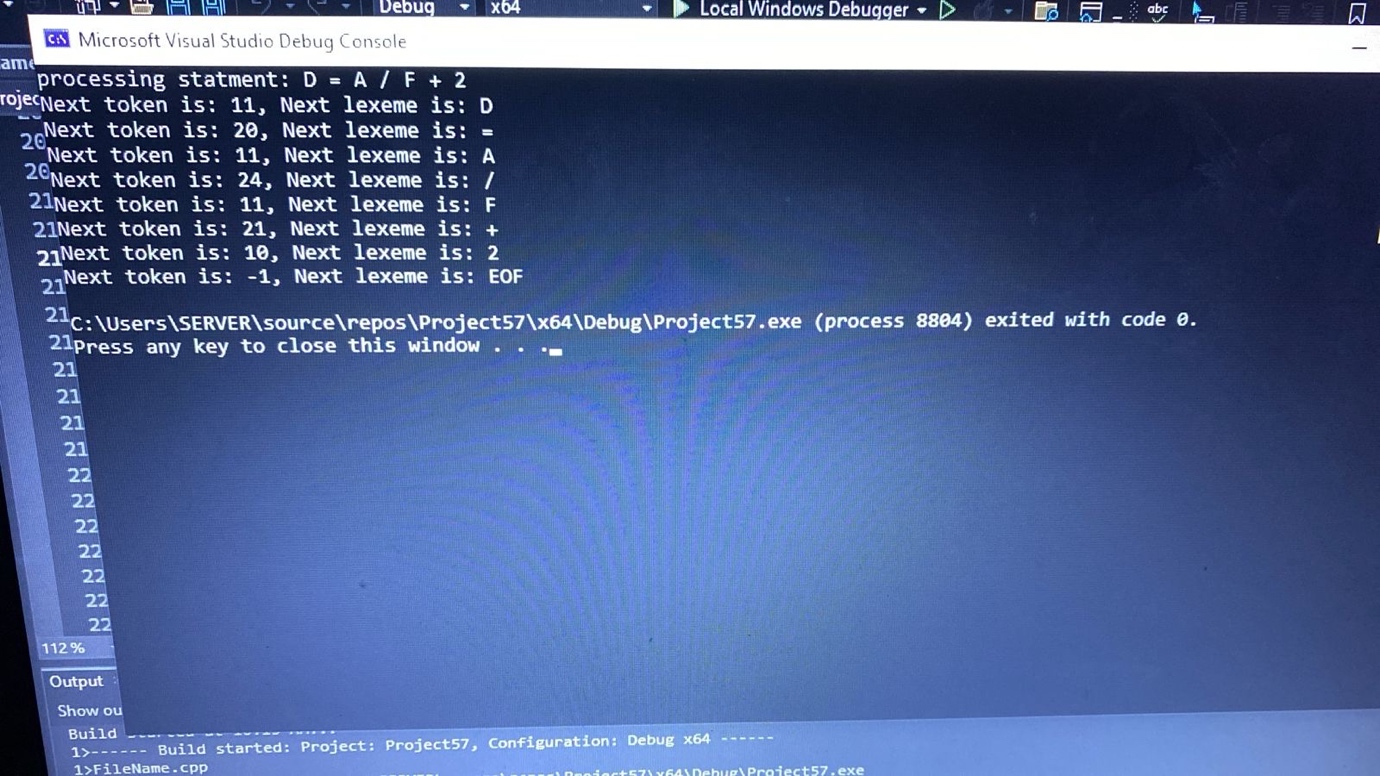
High-level languages: These are closer to human languages and abstract away machine-level details (e.g., Python, Java, C++).

Low-level languages: These are closer to machine code and give more control over hardware (e.g., Assembly language, C).

Domain-specific languages: These are tailored to specific tasks or domains (e.g., SQL for databases, HTML for web pages).

.Programming languages are essential for software development, as they provide the tools to create and manipulate data, interact with users, and perform various computational tasks.

1. **Implementation of a Lexical Analyzer**

****

#include <iostream>

// this library for cout and cin statments.

#include <string>

//this library for string.

#include <cctype>

//this library for handling with files.

#include <cstring>

//this library provides functions for handling c-sytyle string.

using namespace std;

//this is statement provides do not write std:: every line.

#define LETTER 0

//represents alphabeic characters (A-Z, a-z)

#define DIGIT 1

//represents numric digits (0-9).

#define UNKNOWN 99

//represents unrecogonized symbols.

// Token codes

#define EOF\_TOKEN -1

//this is line for represent the end of file marker.

#define INT\_LIT 10

// likely represent literal token.

#define IDENT 11

//this is line for represents an identifier token.

#define ASSIGN\_OP 20

//this is line for represents the assigmnet opreator =.

#define ADD\_OP 21

//this is line for represents the addition opreator +.

#define SUB\_OP 22

//this is line for represents the subtraction opreator -.

#define MULT\_OP 23

//this is line for represent the multplication opreator \*.

#define DIV\_OP 24

//this is line for represent the divison opreator /.

#define LEFT\_PAREN 25

//this is line for represents the left paranthesis (.

#define RIGHT\_PAREN 26

//this is line for represent the right paranthesis ).

class Lexer {

//this is the class that contain all processes that related to lexeical analyzer.

private:

//that allow for all attributes used in class only.

int charClass;

//this variable for identify integer variable to store classification of characters.

string lexeme;

// for identify of a text string variable to store lexemes.

char nextChar;

//to store next character which is will be processed.

int token;

// to store the current token.

int nextToken;

// to store the next token.

string input;

//used to store the statement that user are input to be analysing.

int index;

//this variable used to follow the position of the immediatelycharacter that the program is analysing.

public:

//it is flixable that make all attributs used in and out the class.

Lexer(const string& userInput) {

// this is a constructor that take a const reference to a string (filename) like input.

input = userInput;

//it assigns the value of userinput to the member variable input of the lexer class.

index = 0;

//intilizes an index to keep track of the current position while processing the iput text.

getChar();

}

//this is closing parantheses for constructor lexer.

void addChar() {

// If we have not hit the max length yet, add the character to our current word

if (lexeme.length() < 99) {

lexeme += nextChar;

// just tack it on the end

}

//this is closing parantheses for if condition.

else {

// this word is getting too big time to complain

cout << "Error - lexeme is too long" << endl;

}

//this is closing parantheses for else statement.

}

//this is closing parantheses for function addchar.

void getChar() {

//As long as there are still characters remaining to read.

if (index < input.length()) {

// checks if the current index is less than the lengh of the input string.

nextChar = input[index++];

// assigns the character at the current index in input to nextchar.

if (isalpha(nextChar))

// checks if nextchar is an alphapitic letter using isalpha().

charClass = LETTER;

// if a nextchar is a letter.

else if (isdigit(nextChar))

// if not a letter cause may be that it was be a digit.

charClass = DIGIT;

// if nextchar is a digit.

else

charClass = UNKNOWN;

// if nextchar nethier a letter nor a digit.

1. }
2. else {

charClass = EOF\_TOKEN;

// if the if condition fails.

}

//this is closing parantheses for else statement.

}

//this is closing parantheses for function getchar.

void getNonBlank() {

// skip past any boring spaces or tabs

while (isspace(nextChar))

getChar();

// just keep grabbing characters until we hit something interesting

}

//this is closing parantheses for function getNonBlank.

int lookup(char ch) {

// check what symbol we are looking at and assign the right token type.

switch (ch) {

case '(': addChar(); nextToken = LEFT\_PAREN; break;

// this is opening parenthesis.

case ')': addChar(); nextToken = RIGHT\_PAREN; break;

// this is represents closing parenthesis.

case '+': addChar(); nextToken = ADD\_OP; break;

// this is represents addition

case '-': addChar(); nextToken = SUB\_OP; break;

// this is represents subtraction

case '\*': addChar(); nextToken = MULT\_OP; break;

// this is represents multiplication

case '/': addChar(); nextToken = DIV\_OP; break;

// this is represents division

case '=': addChar(); nextToken = ASSIGN\_OP; break;

// this is represents equals sign

default: addChar(); nextToken = EOF\_TOKEN; break;

// anything else means we are done

}

//this is closing parantheses for switch.

return nextToken;

// tell the caller what we found

}

//this is closing parantheses for function lookup.

int lex() {

//this functin we calling anthoer functions in it.

lexeme = "";

// start fresh with an empty word

// first, skip any whitespace so we can get to the good stuff

getNonBlank();

// now figure out what we are dealing with

switch (charClass) {

case LETTER:

// starts with a letter - probably a variable name

addChar();

// take this first letter

getChar();

// move to next character

// keep going as long as we are getting letters or numbers

while (charClass == LETTER || charClass == DIGIT) {

addChar();

// add to our growing word

getChar();

// move along }

//this is closing parantheses for while.

nextToken = IDENT;

// we have got ourselves an identifier

break;

case DIGIT:

// starts with a number - let is read the whole number

addChar();

// take this first digit

getChar();

// on to the next

// keep eating up digits until we hit something else

while (charClass == DIGIT) {

addChar();

getChar();

}

//this is closing parantheses for while statement.

nextToken = INT\_LIT;

// we have collected a complete number

break;

case UNKNOWN:

// some symbol or operator

lookup(nextChar);

// figure out what it is exactly

getChar();

// move forward

break;

case EOF\_TOKEN: // we have reached the end

nextToken = EOF\_TOKEN;

lexeme = "EOF";

// just for clarity in output

break;

}

//this is closing parantheses for switch.

// show what we found (useful for debugging)

cout << "Next token is: " << nextToken << ", Next lexeme is: " << lexeme << endl;

return nextToken; // pass back what type of token we found

}

//this is closing parantheses for function lex.

void processTokens() {

// keep processing tokens one by one until we run out

do {

lex();

} while (nextToken != EOF\_TOKEN); // stop when we reach the end

}

//this is closing parantheses for function processToken.

};

//this is closing parantheses for class lexer.

int main() {

string userInput = "D = A / F + 2";

// this is the statement lexeme and the program will process it to get the tokens.

cout << "processing statment: " << userInput << endl;

Lexer lexer(userInput);

// start breaking it down into tokens

lexer.processTokens();

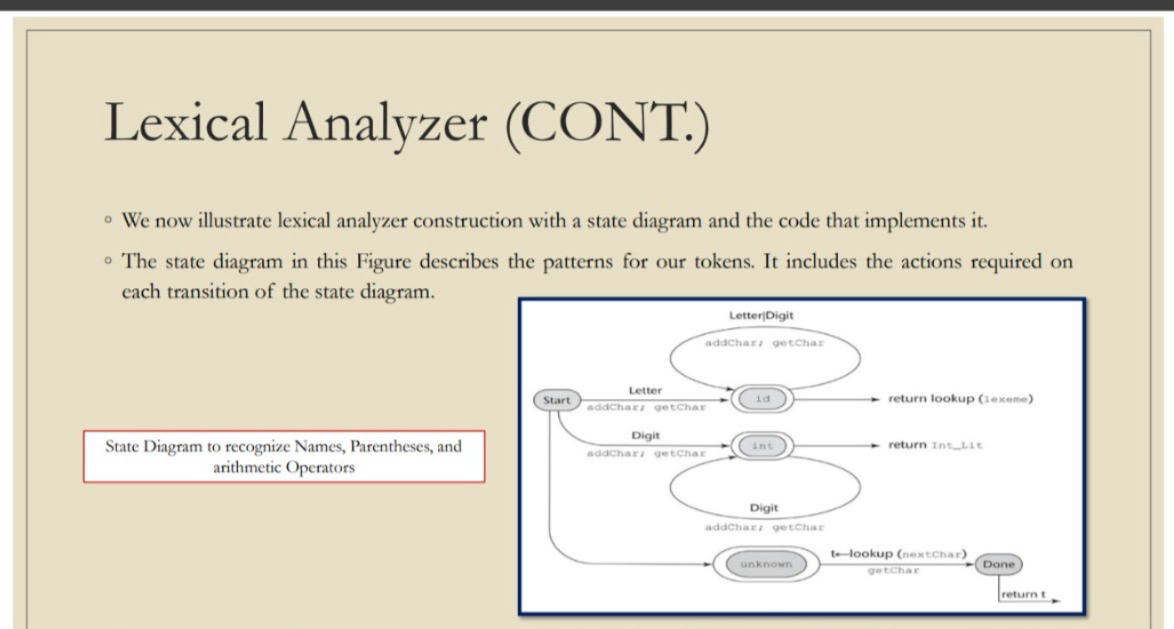
//we here calling the function that processing the lexeme statement from class lexeme.

return 0;

// all done}

// this is closing parantheses for function main.

**The digram :**



**1. Start State**

* The analysis begins at this Start state.
* Based on the first character read, it decides which path to take:
  + Letter: could be a variable name or keyword.
  + Digit: could be a number.
  + Anything else: might be an operator or unknown symbol.

**2. Letter Path**

* If a letter is read:
  + It goes to a state labeled id.
  + Action: addChar; getChar — adds the character to the token and reads the next character.
  + It continues reading characters (letters or digits) to complete the identifier.
  + Eventually: it calls lookup(lexeme) to check if it’s a keyword (like if, while) or just an identifier (like x, count).
  + Then it returns the token (e.g., ID or KEYWORD).

**3. Digit Path**

* If a digit is read:
  + It goes to a state labeled int.
  + Action: addChar; getChar — continues reading digits to form an integer.
  + It eventually returns Int\_Lit, meaning it recognized an integer literal.

**4. Unknown Path**

* If the character is not a letter or digit:
  + It goes to an unknown state.
  + These are likely symbols or operators (e.g., +, -, \*, (, )).
  + Action: t = lookup(nextChar) — identifies what kind of symbol it is.
  + Then, it moves to the Done state and returns the token t.

**Actions in the Diagram**

* addChar: Adds current character to the token being built.
* getChar: Reads the next character from the source code.
* lookup(): Determines what type of token we have (identifier, operator, etc.).
* return: Ends the tokenization process for that token

1. **References**

[1] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, Compilers: Principles, Techniques, and Tools. 2nd ed. Boston, MA, USA: Addison-Wesley, 2007.

[2] cplusplus.com, “C++ tutorial,” [Online]. Available: http://www.cplusplus.com/. [Accessed: Oct. 26, 2023].

[3] ISO/IEC 14882:2017, Programming Languages — C++. International Organization for Standardization, 2017.

[4] GNU Project, “GNU Compiler Collection (GCC),” [Online]. Available: https://gcc.gnu.org/. [Accessed: Nov. 1, 2023].

**Important Note: -**

Technical reports include a mixture of text, tables, and figures. Consider how you can present the information best for your reader. Would a table or figure help to convey your ideasq more effectively than a paragraph describing the same data?

Figures and tables should: -

* Be numbered
* Be referred to in-text, e.g. *In Table 1*…, and
* Include a simple descriptive label - above a table and below a figure.