LEXICAL ANALYZER

Build Scanner

Prepared By
NAME:
OMAR ADEL FAWZY
Student ID:
200038809
Under Supervision
Dr:Nehal Abdelrahman

1. Introduction:

Programming means talking to the computer in a language it understands. We speak Arabic or English, but computers understand other languages like C++ or Python.

When you program, you write steps for the computer to follow. These steps tell it what to do, like solving math, showing pictures, or moving a robot.

Programming is like giving instructions one by one. Everything in phones, computers, and even washing machines works because someone wrote a program.

It’s not hard if you go slowly. Just start with the basics, and you will learn more step by step.

1.1. Phases of Compiler

The compiler doesn't just turn code into machine language in one go — it goes through different stages to finish the job.

1. First, it checks the words
It starts by reading the code line by line and splitting it into small parts called *tokens*. These could be things like variable names, symbols, or keywords. This step is called *lexical analysis*.

2. Then, it looks at the structure
After breaking the code into tokens, the compiler sees if the code is written in the correct way. It checks if the brackets match, if the statements are in the right order, and so on. This is *syntax analysis*.

3. Next, it checks if it makes sense
Even if the code looks correct, the compiler makes sure everything makes logical sense — like if a variable is declared before it's used, or if a function is called properly. This is *semantic analysis*.

4. After that, it turns the code into a middle form
Now the compiler converts the code into something between high-level code and machine code. This is called *intermediate code*. It's not for computers yet, but it's easier to work with.

5. Then, it makes the code better

The compiler improves the intermediate code by removing extra steps or making it faster. This part is called *optimization*.

6. Later, it makes real machine code
After optimizing, the compiler changes the code into machine code — the kind the computer really understands. This step is called *code generation*.

7. Finally, it puts everything together
The compiler connects the final code with any extra files or libraries, and builds one file that can run on your computer. This is *linking and assembling*.

So, the compiler works step by step — each part helps the next one — until the final program is ready to run.

2. Lexical Analyzer

The lexical analyzer is the first stage the compiler uses to deal with the code. Its job is to scan the code from the beginning and chop it into useful parts.

It reads the code letter by letter and turns it into small chunks called tokens. These tokens help the rest of the compiler understand the code better.

Some examples of tokens are:

- Main words used in the language, like while, else, void
- Names you create, like number, get Data
- Signs, such as +, -, ==
- Special marks like { , } , ;

During this phase, things that aren't important to how the code runs — like extra space or comments — are removed.

Also, if you type something wrong, like a symbol that doesn't belong in the language, this is the part that will catch the mistake and show you an error.

So basically, the lexical analyzer organizes and filters the code, getting it ready for the next step, which checks the grammar and structure.

3. Software Tools:

What Are Software Tools?

Software tools are programs that help developers write, edit, test, run, and manage other programs.
In simple words:

Software tools are tools that help you create other software.
They make the job of a programmer easier, faster, and more organized.
Types of Software Tools (With Simple Explanations)

1. Editors
• These are the programs where you actually *write* your code.
• Examples: Notepad++, Visual Studio Code, Sublime Text
• They usually help with syntax highlighting (coloring the code), auto-completion, and formatting.

Example:
Writing C++ code? You'll probably use VS Code or Code:Blocks to type your code.

2. Compilers
• A compiler *translates* your code from a programming language (like C or Java) into machine code (0s and 1s) that the computer can run.
• Without a compiler, your code can't be executed by the computer.

Example:

You write code in C? Use a compiler like GCC to convert it into an .exe file.

3. Interpreters

• Similar to compilers, but instead of translating the whole code at once, interpreters run your code line by line while the program is running.

• Used for languages like Python and JavaScript.

Difference between Compiler and Interpreter:

• Compiler: Translates all code before running.

• Interpreter: Translates and runs code one line at a time.

4. Debuggers

• Help you *find and fix errors* in your code.

• You can pause your program, inspect values, and see where things went wrong.

Example:

If your program gives wrong results, use a debugger to see what's happening step by step.

5. Linkers

• Linkers combine different pieces of code (often in separate files) into one final executable program.

• It connects functions and variables across multiple files.

Example:

If you write some functions in one file and your main code in another, the linker joins them together.

6. Loaders

• Once your program is ready, the *loader* loads it into the computer's memory (RAM) so it can run.

• Without this, the compiled code would just sit on the disk and do nothing.

7. IDEs (Integrated Development Environments)

• These are all-in-one tools that include an editor, compiler, debugger, and more — all in one place.

• They are perfect for writing, testing, and managing code easily.

Popular IDEs:

• Visual Studio
• Eclipse
• Code::Blocks
• IntelliJ IDEA

Why Are Software Tools Important?

• They save time and reduce errors.
• They help you write cleaner, more organized code.
• They're essential for learning, building, and managing real-world projects

3.1. Computer Program

What is a Computer Program?

A computer program is a set of instructions written in a programming language that tells a computer what tasks to perform and how to perform them.

The instructions are executed step by step, allowing the computer to complete specific actions such as calculations, displaying text, interacting with the user, or even playing a game.

How It Works (In Simple Terms):

Think of a program like a recipe.
Just like a recipe tells you:

1. Get the ingredients
2. Mix them
3. Put in the oven
4. Wait 20 minutes

A program tells the computer:

1. Take some input
2. Do something with it
3. Show the result
4. Repeat if needed

Examples of Computer Programs:

- A calculator app
- A web browser (like Chrome)
- A mobile game
- WhatsApp or Messenger
- Even your operating system (like Windows or Android)

Who Writes Programs?

Programs are written by programmers or software developers using special languages like:

- Python
- C++
- Java
- JavaScript

Each language has its own rules, but the goal is the same: tell the computer what to do.

3.2. Programming Language

A programming language is just a special way to talk to the computer.

You can’t speak to a computer in English.
So we use something like C++, which is a programming language, to give the computer instructions.

4. Implementation of a Lexical Analyzer:

```
#include <ctype.h>
#include <stdio.h>

int charClass;
char lexeme[100];
char nextChar;
int lexLen;
int token;
int nextToken;
FILE *in_fp;

#define LETTER 0
#define DIGIT 1
#define UNKNOWN 99
#define INT_LIT 10
#define IDENT 11
#define ASSIGN_OP 20
#define ADD_OP 21
#define SUB_OP 22
#define MULT_OP 23
#define DIV_OP 24
#define LEFT_PAREN 25
#define RIGHT_PAREN 26
```

```
#define EOF -1

void addChar();
void getChar();
void getNonBlank();
int lex();
int lookup(char ch);

int main() {
   if ((in_fp = fopen("front.in", "r")) == NULL)
      printf("ERROR - cannot open front.in \n");
   else {
      getChar();
      do {
         lex();
      } while (nextToken != EOF);
   }
   return 0;
}

int lookup(char ch) {
   switch (ch) {
      case '(':
         addChar();
         nextToken = LEFT_PAREN;
         break;
      case ')':
         addChar();
         nextToken = RIGHT_PAREN;
         break;
      case '+':
            addChar();
            nextToken = ADD_OP;
            break;
        case '-':
            addChar();
            nextToken = SUB_OP;
            break;
        case '*':
            addChar();
            nextToken = MULT_OP;
            break;
        case '/':
            addChar();
            nextToken = DIV_OP;
            break;
        default:
            addChar();
            nextToken = EOF;
            break;
    }
    return nextToken;
}

void addChar() {
    if (lexLen <= 98) {
        lexeme[lexLen++] = nextChar;
        lexeme[lexLen] = '\0';
    } else
        printf("Error - lexeme is too long \n");
}
```

```
void getChar() {
   if ((nextChar = getc(in_fp)) != EOF) {
      if (isalpha(nextChar))
         charClass = LETTER;
      else if (isdigit(nextChar))
         charClass = DIGIT;
      else
         charClass = UNKNOWN;
   } else
      charClass = EOF;
}

void getNonBlank() {
   while (isspace(nextChar))
      getChar();
}

int lex() {
   lexLen = 0;
   getNonBlank();
   switch (charClass) {
      case LETTER:
         addChar();
         getChar();
         while (charClass == LETTER || charClass == DIGIT) {
            addChar();
            getChar();
         }
         nextToken = IDENT;
         break;
      case DIGIT:
```

MISR UNIVERSITY
FOR SCIENCE & TECHNOLOGY
College of Information Technology

جـامعــة مـصــر
للعلـــوم والتكنـــولـــوجيـــا
كليـة تكنولوجيـا المعلومــات

```
            addChar();
            getChar();
            while (charClass == DIGIT) {
               addChar();
               getChar();
            }
            nextToken = INT_LIT;
            break;
         case UNKNOWN:
            lookup(nextChar);
            getChar();
            break;
         case EOF:
            nextToken = EOF;
            lexeme[0] = 'E';
            lexeme[1] = 'O';
            lexeme[2] = 'F';
            lexeme[3] = '\0';
            break;
      }
      printf("Next token is: %d, Next lexeme is %s\n", nextToken, lexeme);
      return nextToken;
   }
```

Al-Motamayez District 6th of October, P.O Box 77, Giza, Egypt.
+(202) 38247455 / 6 / 7 +(202) 38247417 / 38247428 16878
info@must.edu.eg www.must.edu.eg

(Detailed Line-by-Line Explanation):

[1] #include <ctype.h>

This line includes the C library that helps with checking characters.

For example, isalpha() checks if a character is a letter, and isdigit() checks if it’s a digit.

[2] #include <stdio.h>

This includes the standard input/output library in C.

It allows us to use functions like printf() to print text and fopen() to open files.

[4] int charClass;

This variable will hold the class of the current character (e.g., letter, digit, or unknown).

[5] char lexeme[100];

This is a character array that will store the current token we are reading.

A token is like a word, number, or symbol in the input.

[6] char nextChar;

This variable will hold the next character read from the file.

[7] int lexLen;

This variable keeps count of how many characters we’ve stored in lexeme.

[8] int token;

This holds the current token type.

[9] int nextToken;

This holds the next token returned by the lexical analyzer function.

[10] FILE *in_fp;

This is a file pointer used to read from the input file front.in.

[12 - 27] #define lines

These lines define constants.

Instead of using numbers directly, we give them names to make the code easier to understand.

For example:

- LETTER means the character is a letter
- DIGIT means it is a number
- ADD_OP means the + operator
- INT_LIT means integer literal
- EOF is used for end of file

—

[29 - 34] Function declarations

These tell the compiler about the functions we'll be using later.

We define the functions after main(), but we declare them first so the compiler knows about them.

[36] int main()

This is where the program starts.

[37] Checks if the file front.in can be opened for reading.

If it can't be opened, print an error message.

[39] If the file is opened successfully, read the first character using getChar().

[40 - 41] A loop that keeps calling lex() until the end of the file (EOF) is reached.

[43] End of the main function.

[46 - 65] int lookup(char ch)
This function handles symbols like +, -, *, /, (, and).

It checks if the character is one of these operators. If so, it calls addChar() to add it to the lexeme, and assigns the correct token type.

If the character is not one of the known symbols, it sets the token to EOF.

[67 - 73] void addChar()
This function adds the current character to the lexeme array.
We only add the character if the total length is not more than 98, to avoid overflow.
lexeme[lexLen] = '\0'; adds a null character to end the string.

[75 - 82] void getChar()
Reads the next character from the input file and puts it in nextChar.
Then it checks the character class:

- If it’s a letter, charClass is set to LETTER
- If it’s a digit, set to DIGIT
- Otherwise, set to UNKNOWN
- If there’s no more input (end of file), set to EOF

[84 - 85] void getNonBlank()

This function skips spaces or tabs.

It keeps calling getChar() until it finds a non-space character.

[87 - 112] int lex()

This is the most important function — the lexical analyzer.

[88] Reset the length of the current lexeme to 0.

[89] Skip any blank spaces.

[90 - 112] Use a switch statement to check what type the character is:

- [91 - 96] If it’s a letter:

Keep reading letters or digits to form an identifier (like a variable name).

Then set the token type to IDENT.

- [97 - 101] If it’s a digit:

Keep reading digits to form an integer literal.

Then set the token type to INT_LIT.

- [102 - 104] If it’s an unknown character:

Pass it to lookup() to check if it’s an operator. Then read the next character.

- [105 - 110] If the character class is EOF:

Set the token to EOF and store "EOF" in the lexeme.

[111] Print the token and the lexeme for debugging or display.

[112] Return the token.

With Arabic Explanation:

1. int lookup(char ch);

الوظيفة:

تفحص هذه الدالة الرموز (مثل: +, -, *, (، وغيرها) وتُحدّد نوع كل رمز (token type) وتخزن قيمته في nextToken.

متى تُستخدم؟

عند قراءة رمز غير حرف أو رقم (unknown character).

2. void addChar();

الوظيفة:

تضيف الحرف الحالي (nextChar) إلى متغير lexeme (الذي يُمثل الكلمة أو الرمز الكامل الذي نقرأه)، وتحدث طوله في lexLen.

متى تُستخدم؟

عند تجميع الحروف أو الأرقام لتكوين كلمات أو أرقام كاملة.

3. void getChar();

الوظيفة:

تقرأ حرفاً جديداً من الملف وتُخزنه في nextChar.

ثم تُحدد نوع الحرف (حرف – رقم – غير معروف – نهاية الملف) وتُخزنه في charClass.

متى تُستخدم؟

قبل تحليل أو تصنيف أي حرف.

4. void getNonBlank();

الوظيفة:

تتجاهل الفراغات والمسافات البيضاء (مثل الفراغ أو التاب أو السطر الجديد)، وتنتقل إلى أول حرف حقيقي له معنى.

متى تُستخدم؟

قبل البدء في تحليل الرموز، لتجنب الفراغات التي ليس لها معنى.

5. int lex();

الوظيفة:

هذه هي الدالة الرئيسية للتحليل اللغوي (lexical analysis).

- تُكوّن الرموز (tokens) من الحروف أو الأرقام.
- تُحدد نوع كل رمز (مثل: متغير، رقم، عملية جمع، إلخ).
- تُطبع النتيجة على الشاشة.

متى تُستخدم؟

في كل دورة من دورات قراءة الملف، لاستخراج الرمز التالي (next token).

6. int main();

الوظيفة:

هي نقطة البداية لتشغيل البرنامج

- تفتح ملف front.in للقراءة.
- تستدعي getChar() أولاً لتحميل أول حرف.
- ثم تُكرر استدعاء lex() حتى تصل إلى نهاية الملف.

5. References:

CONCEPTS OF PROGRAMMING LANGUAGES TWELFTH EDITION ROBERT W. SEBESTA University of Colorado at Colorado Springs