

lexical analyzer



**Prepared By**

1. Salma Sameh 200020339
2. Mahmoud Nouh 200010285
3. Abdelrahman Bakr 200031361
4. Rabea Ibrahim 200031640
5. Amr Osama 200031891

**Under Supervision**

Dr.Nehal A.Mohamed

T. A.:- Eng:- Mariam faried

**Summary**

In C-, the lexical analysis phase is the first phase of the compilation process. In this step, the lexical analyzer (also known as the Scanner) breaks the code into tokens, which are the smallest individual units in terms of programming. In the lexical analysis phase, we parse the input string, removing the whitespaces. It also helps in simplifying the subsequent stages. In this step, we do not check for the syntax of the code. Our focus is to break down the code into small tokens.

##### ***Table of Contents***

Table of Contents

[1. Introduction 5](#_Toc166243896)

[1.2.2 Syntax Analyzer 11](#_Toc166243897)

[1.3 Programming Language 14](#_Toc166243898)

[1.3.3 Basic Grammar Rules 18](#_Toc166243899)

[1.4 Software Tools 19](#_Toc166243900)

[**1.** **Flex** 19](#_Toc166243901)

[1.5 Screenshots 20](#_Toc166243902)

[1.5.2 Output of Lexical Analyzer 23](#_Toc166243903)

##### ***List of Tables***

|  |  |  |
| --- | --- | --- |
| **Table No.** | **Title** | **Page** |
| **1** | **Tokens** | **10** |
| **2** | **Advantages and and disadvantages** | **11** |
| **3**  **4** | **Advantages and disadvantages**  **Keywords** | **13**  **18** |

##### ***List of Figures***

|  |  |  |
| --- | --- | --- |
| **Figure No.** | **Caption** | **Page** |
| **1** | **Compiler** | **5** |
| **2** | **Front end** | **7** |
| **3** | **Lexical Analyzer** | **8** |
| **4** | **Lexical Analyzer as Finite Automata** | **10** |
| **5** | **Syntax Analyzer** | **13** |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |

# **Introduction**

**What is Compiler?**

The**compiler**is software that converts a program written in a high-level language (Source Language) to a low-level language (Object/Target/Machine Language/0, 1’s).

A translator or language processor is a program that translates an input program written in a programming language into an equivalent program in another language. The compiler is a type of translator, which takes a program written in a high-level programming language as input and translates it into an equivalent program in low-level languages such as machine language or assembly language.

The program written in a high-level language is known as a source program, and the program converted into a low-level language is known as an object (or target) program. Without compilation, no program written in a high-level language can be executed. For every programming language, we have a different compiler; however, the basic tasks performed by every compiler are the same. The process of translating the source code into machine code involves several stages, including lexical analysis, syntax analysis, semantic analysis, code generation, and optimization.

Compiler is an intelligent program as compare to an assembler. Compiler verifies all types of limits, ranges, errors , etc. Compiler program takes more time to run and it occupies huge amount of memory space. The speed of compiler is slower than other system software. It takes time because it enters through the program and then does translation of the full program. When compiler runs on same machine and produces machine code for the same machine on which it is running. Then it is called as self compiler or resident compiler. Compiler may run on one machine and produces the machine codes for other computer then in that case it is called as cross compiler.

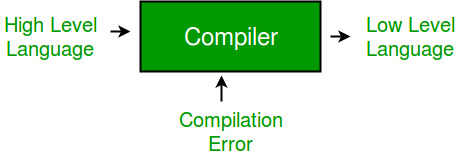


Figure 1: Compiler

**1.2 Description of Frontend of the Compiler**

The compiler has two modules namely the front end and the back end. Front-end constitutes the Lexical analyzer, semantic analyzer, syntax analyzer, and intermediate code generator. And the rest are assembled to form the back end.

Lexical Analyzer –

It is also called a scanner. It takes the output of the preprocessor (which performs file inclusion and macro expansion) as the input which is in a pure high-level language. It reads the characters from the source program and groups them into lexemes (sequence of characters that “go together”). Each lexeme corresponds to a token. Tokens are defined by regular expressions which are understood by the lexical analyzer. It also removes lexical errors (e.g., erroneous characters), comments, and white space.

Syntax Analyzer –

It is sometimes called a parser. It constructs the parse tree. It takes all the tokens one by one and uses Context-Free Grammar to construct the parse tree.

Why Grammar?

The rules of programming can be entirely represented in a few productions. Using these productions we can represent what the program actually is. The input has to be checked whether it is in the desired format or not.

The parse tree is also called the derivation tree. Parse trees are generally constructed to check for ambiguity in the given grammar. There are certain rules associated with the derivation tree.

* Any identifier is an expression
* Any number can be called an expression
* Performing any operations in the given expression will always result in an expression. For example, the sum of two expressions is also an expression.
* The parse tree can be compressed to form a syntax tree.
* Syntax error can be detected at this level if the input is not in accordance with the grammar.

Semantic Analyzer – It verifies the parse tree, whether it’s meaningful or not. It furthermore produces a verified parse tree. It also does type checking, Label checking, and Flow control checking.

Intermediate Code Generator – It generates intermediate code, which is a form that can be readily executed by a machine We have many popular intermediate codes. Example – Three address codes etc. Intermediate code is converted to machine language using the last two phases which are platform dependent.

Till intermediate code, it is the same for every compiler out there, but after that, it depends on the platform. To build a new compiler we don’t need to build it from scratch. We can take the intermediate code from the already existing compiler and build the last two parts.

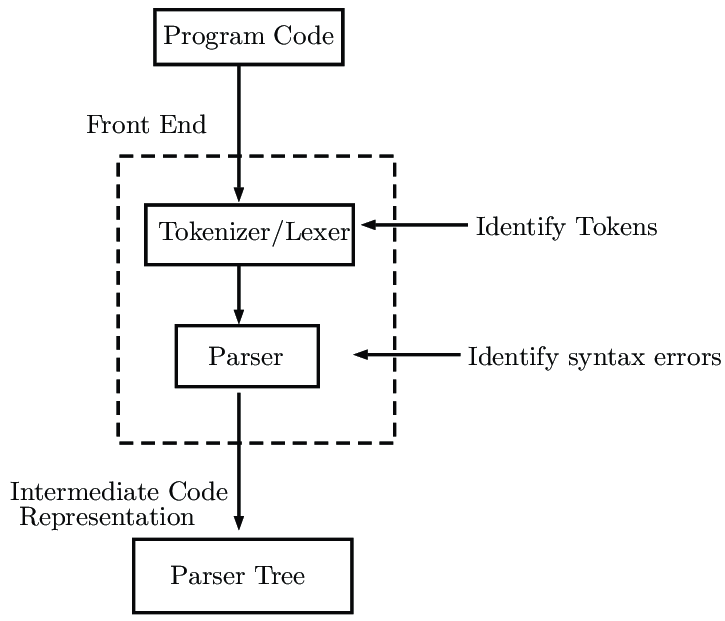


Figure 2: Front End

**1.2.1 Lexical Analyzer**

Lexical Analysis is the first phase of the compiler also known as a scanner. It converts the High level input program into a sequence of Tokens.

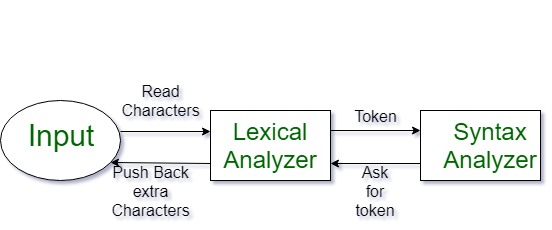
* Lexical Analysis can be implemented with the Deterministic finite Automata.
* ****The output is a sequence of tokens that is sent to the parser for syntax analysis.

Figure 3: Lexical Analyzer

**What is a Token?**

A lexical token is a sequence of characters that can be treated as a unit in the grammar of the programming languages. Example of tokens:

* Type token (id, number, real, . . . )
* Punctuation tokens (IF, void, return, . . . )
* Alphabetic tokens (keywords)

Example of Non-Tokens:

* Comments, preprocessor directive, macros, blanks, tabs, newline, etc.

Lexeme: The sequence of characters matched by a pattern to form the corresponding token or a sequence of input characters that comprises a single token is called a lexeme. eg- “float”, “abs\_zero\_Kelvin”, “=”, “-”, “273”, “;” .

**How Lexical Analyzer Works?**

1. Input preprocessing: This stage involves cleaning up the input text and preparing it for lexical analysis. This may include removing comments, whitespace, and other non-essential characters from the input text.
2. Tokenization: This is the process of breaking the input text into a sequence of tokens. This is usually done by matching the characters in the input text against a set of patterns or regular expressions that define the different types of tokens.
3. Token classification: In this stage, the lexer determines the type of each token. For example, in a programming language, the lexer might classify keywords, identifiers, operators, and punctuation symbols as separate token types.
4. Token validation: In this stage, the lexer checks that each token is valid according to the rules of the programming language. For example, it might check that a variable name is a valid identifier, or that an operator has the correct syntax.
5. Output generation: In this final stage, the lexer generates the output of the lexical analysis process, which is typically a list of tokens. This list of tokens can then be passed to the next stage of compilation or interpretation.

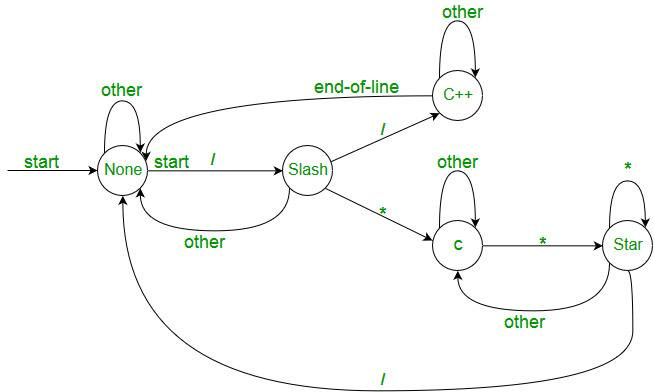


Figure 4: Lexical Analyzer as Finite Automata

The lexical analyzer identifies the error with the help of the automation machine and the grammar of the given language on which it is based like C, C++, and gives row number and column number of the error.

|  |  |  |  |
| --- | --- | --- | --- |
| Lexemes | Tokens | Lexemes | Tokens |
| while | WHILE | a | IDENTIEFIER |
| ( | LAPREN | = | ASSIGNMENT |
| a | IDENTIFIER | a | IDENTIFIER |
| >= | COMPARISON | – | ARITHMETIC |
| b | IDENTIFIER | 2 | INTEGER |
| ) | RPAREN | ; | SEMICOLON |

Table 1: Tokens

|  |
| --- |
| Advantages |
| * Simplifies Parsing:Breaking down the source code into tokens makes it easier for computers to understand and work with the code. This helps programs like compilers or interpreters to figure out what the code is supposed to do. It’s like breaking down a big puzzle into smaller pieces, which makes it easier to put together and solve. |
| * Error Detection: Lexical analysis will detect lexical errors such as misspelled keywords or undefined symbols early in the compilation process. This helps in improving the overall efficiency of the compiler or interpreter by identifying errors sooner rather than later. |
| * Efficiency: Once the source code is converted into tokens, subsequent phases of compilation or interpretation can operate more efficiently. Parsing and semantic analysis become faster and more streamlined when working with tokenized input. |
| Disadvantages |
| * Limited Context: Lexical analysis operates based on individual tokens and does not consider the overall context of the code. This can sometimes lead to ambiguity or misinterpretation of the code’s intended meaning especially in languages with complex syntax or semantics. |
| * Overhead: Although lexical analysis is necessary for the compilation or interpretation process, it adds an extra layer of overhead. Tokenizing the source code requires additional computational resources which can impact the overall performance of the compiler or interpreter. |
| * Debugging Challenges: Lexical errors detected during the analysis phase may not always provide clear indications of their origins in the original source code. Debugging such errors can be challenging especially if they result from subtle mistakes in the lexical analysis process. |

Table 2: advantages and disadvantage

# **1.2.2 Syntax Analyzer**

When an input string (source code or a program in some language) is given to a compiler, the compiler processes it in several phases, starting from lexical analysis (scans the input and divides it into tokens) to target code generation.

Syntax Analysis or Parsing is the second phase, i.e. after lexical analysis. It checks the syntactical structure of the given input, i.e. whether the given input is in the correct syntax (of the language in which the input has been written) or not. It does so by building a data structure, called a Parse tree or Syntax tree. The parse tree is constructed by using the pre-defined Grammar of the language and the input string. If the given input string can be produced with the help of the syntax tree (in the derivation process), the input string is found to be in the correct syntax. if not, the error is reported by the syntax analyzer.

Syntax analysis, also known as parsing, is a process in compiler design where the compiler checks if the source code follows the grammatical rules of the programming language. This is typically the second stage of the compilation process, following lexical analysis.

The main goal of syntax analysis is to create a parse tree or abstract syntax tree (AST) of the source code, which is a hierarchical representation of the source code that reflects the grammatical structure of the program.

There are several types of parsing algorithms used in syntax analysis, including:

* LL parsing: This is a top-down parsing algorithm that starts with the root of the parse tree and constructs the tree by successively expanding non-terminals. LL parsing is known for its simplicity and ease of implementation.
* LR parsing: This is a bottom-up parsing algorithm that starts with the leaves of the parse tree and constructs the tree by successively reducing terminals. LR parsing is more powerful than LL parsing and can handle a larger class of grammars.
* LR(1) parsing: This is a variant of LR parsing that uses lookahead to disambiguate the grammar.
* LALR parsing: This is a variant of LR parsing that uses a reduced set of lookahead symbols to reduce the number of states in the LR parser.
* Once the parse tree is constructed, the compiler can perform semantic analysis to check if the source code makes sense and follows the semantics of the programming language.
* The parse tree or AST can also be used in the code generation phase of the compiler design to generate intermediate code or machine code.

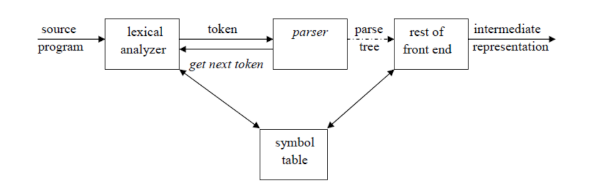


Figure 5: Syntax Analyzer

|  |
| --- |
| Advantages : |
| * Advantages of using syntax analysis in compiler design include: |
| * Structural validation: Syntax analysis allows the compiler to check if the source code follows the grammatical rules of the programming language, which helps to detect and report errors in the source code. |
| * Improved code generation: Syntax analysis can generate a parse tree or abstract syntax tree (AST) of the source code, which can be used in the code generation phase of the compiler design to generate more efficient and optimized code. |
| * Easier semantic analysis: Once the parse tree or AST is constructed, the compiler can perform semantic analysis more easily, as it can rely on the structural information provided by the parse tree or AST. |
| Disadvantages: |
| Disadvantages of using syntax analysis in compiler design include: |
| * Complexity: Parsing is a complex process, and the quality of the parser can greatly impact the performance of the resulting code. Implementing a parser for a complex programming language can be a challenging task, especially for languages with ambiguous grammars. |
| * Reduced performance: Syntax analysis can add overhead to the compilation process, which can reduce the performance of the compiler. |
| * Limited error recovery: Syntax analysis algorithms may not be able to recover from errors in the source code, which can lead to incomplete or incorrect parse trees and make it difficult for the compiler to continue the compilation process. |
| * Inability to handle all languages: Not all languages have formal grammars, and some languages may not be easily parseable. |

Table 3: advantages and disadvantages

# **1.3 Programming Language**

"C Minus" (C-) is a simplified subset of the C programming language. It was originally developed as a teaching tool for introductory programming courses, focusing on fundamental concepts while avoiding some of the complexities of full-fledged C programming. C- retains many features of C but omits certain advanced features and syntax elements, making it easier for beginners to grasp.

Key characteristics of C- include:

* Simplicity: C- simplifies the syntax and semantics of C, making it easier for beginners to understand. It removes some of the more intricate features found in C, such as complex pointer arithmetic and advanced control structures.
* Fundamental Concepts: C- emphasizes fundamental programming concepts such as variables, loops, conditionals, functions, and basic data types. This allows beginners to focus on mastering essential programming skills without getting bogged down by complex language features.
* Readability: The simplified syntax of C- enhances code readability, making it easier for students and novice programmers to follow and understand program logic.
* Educational Tool: C- is often used as a teaching tool in introductory programming courses, providing students with a gentle introduction to programming concepts before progressing to more advanced languages like C or C++.
* Transitional Language: Because C- is a subset of C, students who learn C- can easily transition to using full-fledged C once they have mastered the basics. This makes it a valuable stepping stone for learners interested in systems programming or software development.
* Despite its simplicity and educational value, C- has limited practical use outside of teaching environments. It lacks many features required for real-world programming tasks and is not suitable for building complex or performance-critical software. However, for learning the basics of programming and understanding foundational concepts, C- serves as an excellent starting point.

Advantages:

* Simplicity: C- offers a simplified version of the C programming language, making it easier for beginners to grasp fundamental programming concepts without being overwhelmed by complex syntax or features.
* Ease of Learning: With its simplified syntax and focus on fundamental concepts, C- provides an accessible entry point for novice programmers to learn programming logic, control structures, and basic data types.
* Readability: The streamlined syntax of C- enhances code readability, making it easier for both students and instructors to understand and review code, which is crucial for educational purposes.
* Smooth Transition to C: Because C- is a subset of C, students who learn C- can easily transition to using full-fledged C once they have mastered the basics. This facilitates a seamless progression to more advanced programming languages and concepts.
* Educational Value: C- is specifically designed as a teaching tool for introductory programming courses, providing students with a solid foundation in programming principles that they can build upon as they advance in their studies or careers.

Disadvantages:

* Limited Practical Use: C- lacks many features and functionalities of full-fledged C programming, limiting its practical applications outside of educational settings. It is not suitable for building complex or performance-critical software.
* Restricted Feature Set: The simplified nature of C- means that it lacks certain advanced features found in C, such as dynamic memory allocation, advanced data structures, and complex control flow constructs. This may limit the scope of projects that can be implemented using C-.
* Transition to Other Languages: While C- serves as a stepping stone to learning C, transitioning to other programming languages may require additional effort, as students may need to learn new syntax and features not present in C or C-.
* Limited Community and Resources: Compared to mainstream programming languages, C- has a smaller user base and less community support, resulting in fewer resources, libraries, and learning materials available for students and developers.
* Risk of Misunderstanding: Since C- is a simplified subset of C, there is a risk that students may develop misconceptions or misunderstandings about programming concepts that are not fully represented in C-. It is essential for instructors to supplement C- instruction with explanations of concepts that may be missing or simplified in C-.

**1.3.1 List of Reserved Words**

* auto
* break
* case
* char
* const
* continue
* default
* do
* double
* else
* enum
* extern
* float
* for
* goto
* if
* int
* long
* register
* return
* short
* signed
* sizeof
* static
* struct
* switch
* typedef
* union
* unsigned
* void
* volatile
* while

These reserved words are keywords that have predefined meanings in the C- language and cannot be used as identifiers (such as variable names or function names) in C- programs. They are an integral part of the language's syntax and semantics, playing crucial roles in defining program structure, control flow, and data types.

**1.3.2 List of Keywords**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| auto | break | case | char | Const | continue | default | do |
| double | else | enum | extern | Float | for | goto | if |
| int | long | register | return | Short | signed | sizeof | static |
| struct | switch | typedef | union | Unsigned | void | volatile | while |

Table 4: keywords

# **1.3.3 Basic Grammar Rules**

1. program → declList

2. declList → declList decl | decl

3. decl → varDecl | funDecl

4. varDecl → typeSpec varDeclList ;

5. scopedVarDecl → static typeSpec varDeclList ; | typeSpec varDeclList ;

6. varDeclList → varDeclList , varDeclInit | varDeclInit

7. varDeclInit → varDeclId | varDeclId : simpleExp

8. varDeclId → ID | ID [ NUMCONST ]

9. typeSpec → int | bool | char

10. funDecl → typeSpec ID ( parms ) stmt | ID ( parms ) stmt

11. parms → parmList |

12. parmList → parmList ; parmTypeList | parmTypeList

13. parmTypeList → typeSpec parmIdList

14. parmIdList → parmIdList , parmId | parmId

15. parmId → ID | ID [ ]

16. stmt → expStmt | compoundStmt | selectStmt | iterStmt | returnStmt | breakStmt

17. expStmt → exp ; | ;

18. compoundStmt → { localDecls stmtList }

19. localDecls → localDecls scopedVarDecl |

20. stmtList → stmtList stmt |

21. selectStmt → if simpleExp then stmt | if simpleExp then stmt else stmt

22. iterStmt → while simpleExp do stmt | for ID = iterRange do stmt

23. iterRange → simpleExp to simpleExp | simpleExp to simpleExp by simpleExp

24. returnStmt → return ; | return exp ;

25. breakStmt → break ;

# **1.4 Software Tools**

## **Flex**

Flex, short for "Fast Lexical Analyzer Generator," is a powerful software tool used for generating lexical analyzers or scanners. It is often used in conjunction with the Bison parser generator to create complete compiler frontends for programming languages.

Flex is a versatile and efficient tool for generating lexical analyzers or scanners for use in compiler construction and other text processing tasks. Its support for regular expressions, ease of use, and integration with other tools such as Bison make it a popular choice for developers working on language processing projects.

1. **Command prompt**

The command prompt, also known as the command line interface (CLI) or command shell, is a text-based interface for interacting with a computer's operating system (OS). It provides a way for users to input commands directly to the system, enabling them to perform various tasks, execute programs, and manage files and directories without relying on a graphical user interface (GUI).

the command prompt provides a powerful and flexible interface for interacting with the operating system and performing a wide range of tasks efficiently, especially for users comfortable with text-based interactions and those requiring automation and scripting capabilities.

1. **Developer Command Prompt for VS 2022**

The "Developer Command Prompt for Visual Studio" is a specialized command prompt environment provided by Microsoft Visual Studio. It is tailored for developers who use Visual Studio as their primary integrated development environment (IDE) for building software projects.

1. **Visual Studio**

Visual Studio is an integrated development environment (IDE) developed by Microsoft. It provides developers with a comprehensive set of tools for building a wide range of software applications, including web applications, desktop applications, mobile apps, cloud services, and games. Visual Studio supports multiple programming languages, frameworks, and platforms, making it a versatile and powerful tool for software development.

# **1.5 Screenshots**

A computer screen shot of a program

Description automatically generated**1.5.1 Input of Lexical Analyzer**

A screen shot of a computer program

Description automatically generated

A computer screen shot of white text

Description automatically generated

A screen shot of a computer program

Description automatically generated

# **1.5.2 Output of Lexical Analyzer**