Chapter 1: Lexical Analysis

What are we studying in this chapter?

- Compilers
  - Analysis of source program
  - The phases of a compiler
  - Cousins of a compiler
  - The grouping of phases
  - Compiler-construction tools
- Lexical analysis
  - The role of Lexical Analyzer
  - Input buffering
  - Specifications of tokens
  - Recognition of tokens **- 6 hours**

1.1 Introduction

We know that the programming languages are used to describe computations to people and to machines. The programming languages are very important since all the software running on all the computers was written in some programming language. But, before we run a program on any computer, it must be translated into a form which can be understood and executed by the computer. For this purpose we use language processors. In the first section, we discuss something about language processors.

1.1.1 Language processors

Now, let us see "What is a language processor?"

Definition: A language processor is a program that accepts source language as the input and translates it into another language called target language. The language processors are used for software development without worrying much about the architecture of the machine i.e., the programmers can ignore the machine-dependent details during software development. It performs translating and interpreting a specified programming language.

Now, let us see "What are the various types of language processors?" The various types of language processors are shown below:

- Preprocessor
- Compiler
- Interpreter
- Assembler
- Hybrid compiler

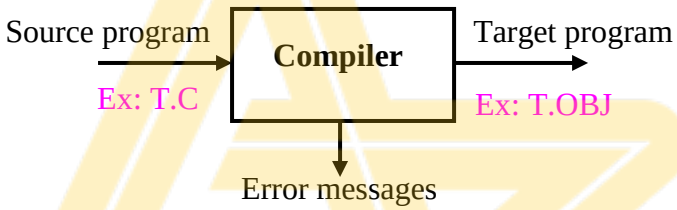
1.2 💻 Lexical Analyzer

Now, let us see "What is a compiler? What are the activities performed by the compiler?"

Definition: A compiler is a translator that accepts the source program written in high level language such as C/C++ and converts it into object program or low level language such as assembly language. The various activities done by the compiler are shown below:

- The syntax errors are identified and appropriate error messages are displayed along with line numbers
- An optional list file can be generated by the compiler
- The compiler replaces each executable statement in high level language into one or more machine language instructions
- Converts HLL into assembly language or object program if the program is syntactically correct.

This can be pictorially represented as shown below:

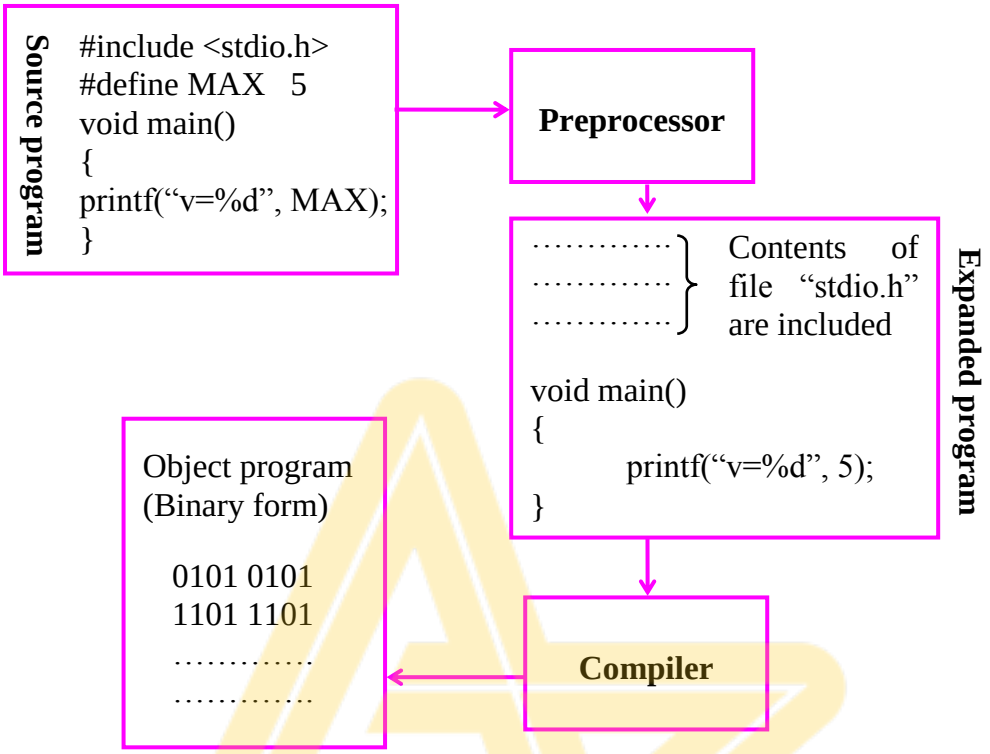


If the target program is an executable machine code, it can be called by the user to process the inputs and produce output which is pictorially represented as shown below:

Now, let us see "What is a preprocessor?"

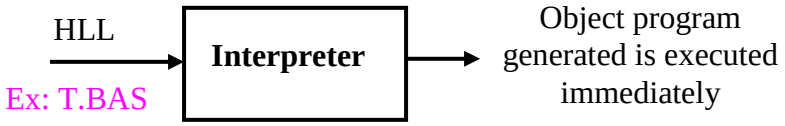
Definition: The preprocessor is a program that accepts source program as the input and prepares the input program for compilation. A source program is normally divided into modules and will be stored in separate files. All these files are collected by the preprocessor with the help of #include directive. The preprocessor may also expand the shorthands called macros into source language statements with the help of #define directive.

Thus, the expanded version of the source program is input to the compiler. This is pictorially represented as shown below:



Now, let us see "What is an interpreter?"

Definition: Interpreter is a translator that accepts the source program written in high level language and converts it into machine code and executes it. The interpreter converts one high level language statement into machine code and then executes immediately. The statements within the loop are translated into machine code each time the control enters into the loop. Whenever an error is encountered, the execution of the program is halted and an error message is displayed. This can be pictorially represented as shown below:



Some of the interpreters are Java virtual machine, BASIC etc.

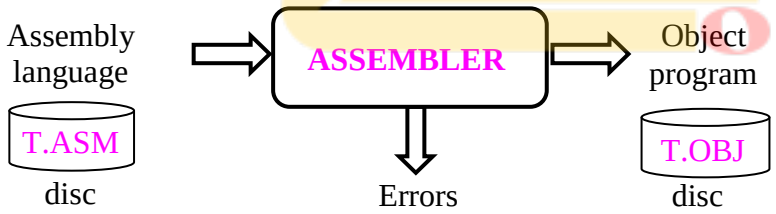
Now, let us see "What are the differences between a compiler and interpreter?" The difference between compiler and interpreter are shown below:

1.4 💻 Lexical Analyzer

COMPILER	INTERPRETER
♦ Converts the entire source program into machine code. The linker uses the machine code and creates an executable file	♦ The interpreter takes one statement at a time, translates it into machine code and execute that instruction
♦ The compiler will not execute the machine code	♦ The interpreter executes the machine code generated
♦ Compiler requires more memory while translating	♦ Interpreter requires less memory while translating
♦ Compiler is not required to execute the executable code	♦ Interpreter is required whenever a program has to be executed
♦ Compiler is costlier	♦ It is cheaper than the compiler
♦ Compiled program runs faster	♦ Interpreted code runs slower
♦ Object program is created only when there are no syntax and semantic errors in the program	♦ Program can be executed even if the syntax errors are present in later stages but execution stops whenever syntax error is encountered.

Now, let us see "What is an assembler? What are the activities performed by the compiler?"

Definition: An assembler is a translator which converts assembly language program into equivalent machine code. The machine code thus produced is written into a file called object program. The block diagram of the assembler is shown below:



Now, let us see "What are the various functions of the assembler? The various functions of the assembler are shown below:

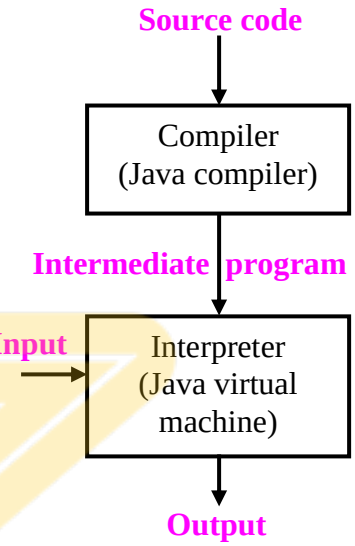
- ♦ Convert assembly language instructions to their machine language equivalent
- ♦ Convert the data constants into equivalent machine representation. For example, if decimal data 4095 is used as operand, it is converted into hexadecimal value (machine representation) FFF.
- ♦ Build the machine instruction using the appropriate instruction format.

- Process the instructions given to the assembler directives. These directives help the assembler while converting from assembly language to machine language
- Create the list file. The list file consists of source code along with address for each instruction and corresponding object code.
- Display appropriate errors on the screen so that the programmer can correct the program and re-assemble it.
- Create the object program. The object program consists of machine instructions and instructions to the loader.

Now, let us see "What is hybrid compiler?"

Definition: A translator that has features of a compiler as well as features of an interpreter is called *hybrid compiler*. The hybrid compilers translate the source program into intermediate byte code for later interpretation.

For example, the java language processor combines compilation and interpretation as shown in the figure. Observe the following points:



- The java source program is first translated into intermediate form called *bytecodes* by the java compiler.
- The *bytecodes* are then interpreted by java virtual machine.
- During execution, the inputs are accepted and appropriate results are displayed as the output.

Advantages: The *bytecodes* compiled on one machine can be interpreted on another machine which has java virtual machine and hence portability issues will not arise.

Note: Now, let us see "What is portability with respect to software?" The software written for one machine (with Intel processor) is copied into another machine (with Motorola processor) and that software is perfectly executed in both machines, then we say that the software is portable. Otherwise, the software is not portable.

Ex1: The object program obtained from a C compiler in one machine (with Intel processor) can be copied into another machine (with Motorola processor). But, the copied object program cannot be loaded and executed by the Motorola processor. So, we say object program is not portable.

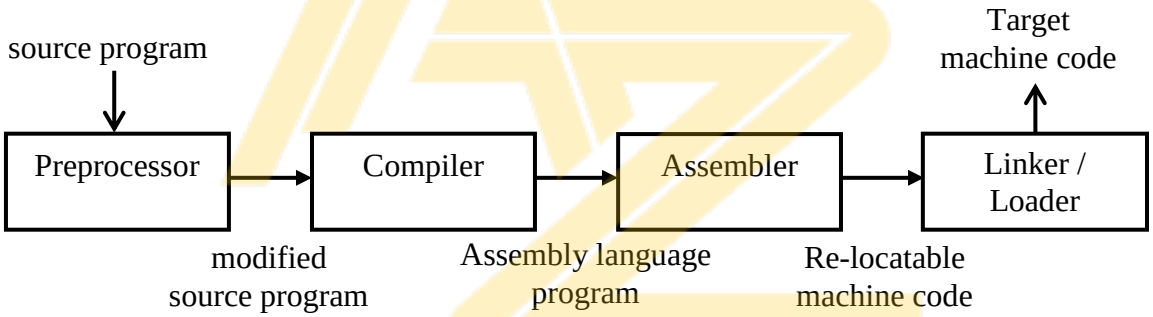
Ex2: Any C program may be copied from one machine (with Intel processor) to another machine (with Motorola processor). The C program can be compiled using C compilers in both machines and can be executed in both machines. This is possible if both machines have C compilers. Now, we say that C programs are portable.

Ex3: For all java programs, the byte codes can be generated using java compilers. Since, all the machines with java virtual machine can execute the byte codes, we say that all java programs are portable.

1.1.2 Language processing system (Cousins of the compiler)

Now, let us see "What are the cousins of the compiler?" or let us "Explain language processing system" During software development, in addition to a compiler several other programs may be required to create an executable target program. All the programs that are helpful in generating the required executable target program are called *cousins of the compiler*. All these cousins of the compiler are together represent language processing system.

The language processing system or the various cousins of the compiler and their interaction are shown in block diagram below:



Thus, the various programs that constitute language processing system (or cousins of the compiler) are:

Preprocessor: The preprocessor is executed before the actual compilation of code begins. A source program may be divided into modules and can be stored in separate files. The preprocessor will collect all source files and may expand macros into source language statements. Thus the output of the preprocessor is also a source program but expanded and modified. This expanded source program is input to the compiler. The main activities performed by the preprocessor are:

1) Macro processing which is identified using #define directive
2) File inclusion which is identified using #include directive
3) It also helps in conditional compilation with the help of the directives such as #if, #else etc.

Compiler: The compiler is a translator that accepts the expanded and modified source program as the input from preprocessor and converts it into assembly language program. If any errors are present, they are displayed along with appropriate error messages and line numbers so that the user can correct the program. The various activities performed by the compiler are:

1) The syntax errors are identified and appropriate error messages are displayed along with line numbers
2) An optional list file can be generated by the compiler
3) The compiler replaces each executable statement in high level language into one or more machine language instructions
4) Converts HLL into assembly language or object program if the program is syntactically correct.

Assembler: An assembler is a translator which converts assembly language program into equivalent machine code. The machine code thus produced is written into a file called object program. The various functions of the assembler are shown below:

1) Convert assembly language instructions to their machine language equivalent
2) Convert the data constants into equivalent machine representation. For example, if decimal data 4095 is used as operand, it is converted into hexadecimal value (machine representation) FFF.
3) Build the machine instruction using the appropriate instruction format.
4) Process the instructions given to the assembler directives. These directives help the assembler while converting from assembly language to machine language
5) Create the list file. The list file consists of source code along with address for each instruction and corresponding object code.
6) Display appropriate errors on the screen so that the programmer can correct the program and re-assemble it.
7) Create the object program. The object program consists of machine instructions and instructions to the loader.

Linker: Large programs are often compiled in pieces and generate object programs and stored in files. The *linker* or *linkage editor* accepts one or more object programs generated by a compiler and links the library files and creates an executable file. The linker resolves external reference symbols where code in one file may refer to a location in another file.

Loader: A **loader** is the part of an operating system that is responsible for loading programs into main memory for execution. This is one of the important stages in the process of executing a program. Loading a program involves reading the contents of executable file, loading the file containing the program text into memory, and then carrying out other tasks to prepare the executable for running. Once loading is complete, the operating system starts the program by passing control to the loaded program code and execution begins.

1.8 💻 Lexical Analyzer

1.2 Analysis of source program

Now, let us "Discuss the analysis of source program with respect to compilation process" The process of converting from high level language to target language is called compilation. The entire compilation process is mapped into two parts namely: Analysis and synthesis

Analysis part: It acts as the front end of the compiler and performs various activities as shown below:

- Accepts the source program and breaks it into pieces and imposes a grammatical structure on them
- It then uses the above structure to create an intermediate representation of the source program
- It also displays appropriate error messages whenever a syntax error or sematic error is encountered during compilation. This helps the programmer to correct the program.
- It also gathers the information about the source program and stores the variables and data in a data structure called symbol table along with type of data, line numbers etc.
- The symbol table thus obtained along with intermediate representation of the source program is sent to syntheses part.

Synthesis part: This is the back end of the compilation process and performs the following activities:

- It accepts the intermediate representation of source program along with symbol table from the analysis part.
- It generates the target program or object program as the output.

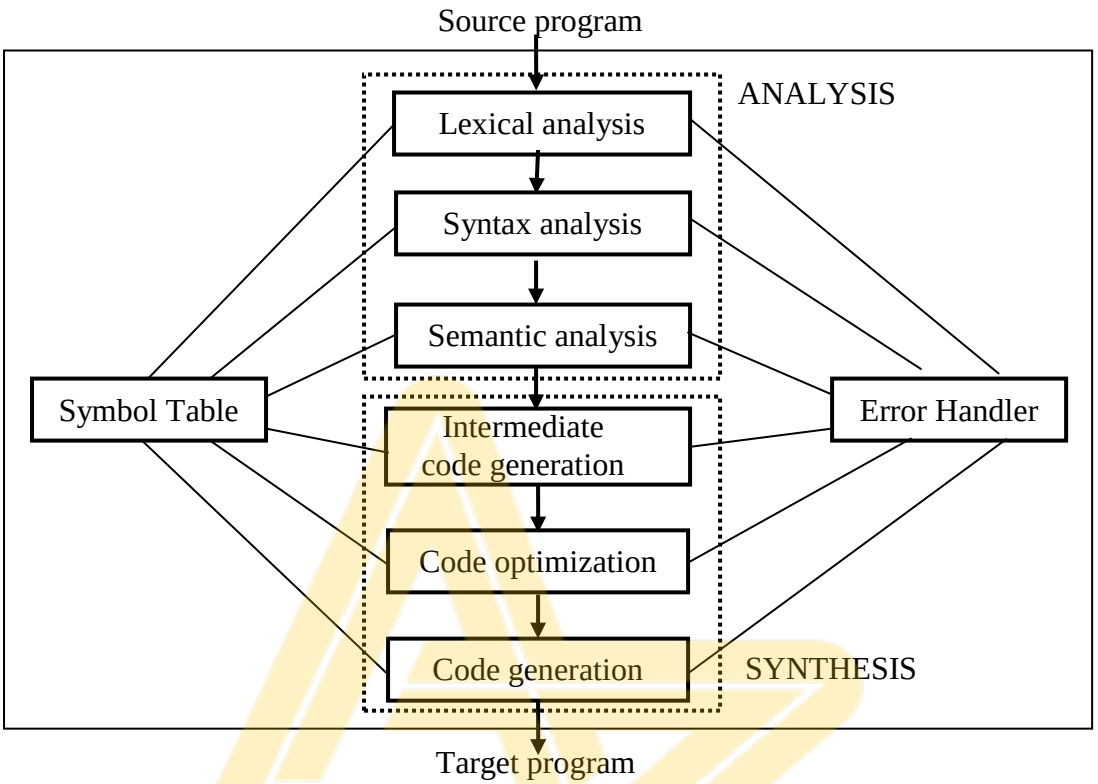
To understand the compilation process in more detail, this large system should be divided into various modules with proper interfaces as discussed in the next section.

1.2.1 The phases of a compiler

In this section, let us see "What are the different phases of the compiler?" or "Explain the block diagram of the compiler"

The various phases of the compiler and the block diagram is shown below:

- Lexical analysis
- Syntax analysis
- Semantic analysis
- Intermediate code generation
- Code optimization
- Code generation



Lexical analysis: It is the first phase of the compiler. It accepts the source program as the input and produces tokens as the output. During this phase, the lexical analyzer performs following activities:

- Removes white spaces (tabs, spaces and newline characters).
- Removes comments from the program.
- Interacts with symbol table to insert variables and constants into symbol table and returns a token consisting of an integer code and pointer to variable and constant in the symbol table.
- Detects invalid identifiers and interacts with error handler to display the error messages.

Example 1.1: Show the various tokens generated by the lexical analyzer for the following statement:

a = b + c

The various tokens generated by the lexical analyzer for the statement "a = b + c" are shown below:

1.10 Lexical Analyzer

lexemes	tokens
a	ID, 1
=	ASSIGN
b	ID, 2
+	PLUS
c	ID, 3

Note: For all symbols and keywords used in the program the unique numbers are assigned and available to both lexical analyzer and syntax analyzer as shown in header file:

```
#define     ASSIGN    1
#define     PLUS      2
#define     MUL       3
#define     LP        4
….
….
#define     ID        100
#define     NUM       101
```

Syntax analysis: It is the second phase of the compiler. It accepts the tokens from the lexical analyzer and imposes a grammatical structure on them and checks whether the program is syntactically correct or not. The various activities done by the parser are:

- Detects syntax errors (such as undefined symbol, semicolon expected etc.)
- Interacts with symbol table and the error handler to display appropriate error messages
- Generates parse tree (or syntax tree which is the compressed form of the parse tree)

Semantic analysis: It is the third phase of the compiler. Type checking and type conversion is done by this phase. Other operations that are performed by this phase are:

- Collects type information (such as int, float, char etc.) of all variables and constants for subsequent code generation phase
- Checks the type of each operand in an arithmetic expression and report error if any (For example, subscript variable *i* in an array *a* should not be **float** or **double)**
- Type conversion at the appropriate place is done in this phase. (For example, conversion from 10 to 10.0)
- Detects error whenever there is a mismatch in the type of arguments and parameters in the function call and function header

Intermediate code generation: It is the fourth phase of the compiler. The syntax tree which is the output of the semantic analyzer is input to this phase. Using this syntax tree,

it generates intermediate code which is suitable for generating the code. Some of the activities that are performed by this phase are:

- High level language statements such as while, if, switch etc., are translated into low level conditional statements
- Produces stream of simple instructions and fed to the next phase of the compiler

For example, consider the statement:

```
b = a*b + c
```

The intermediate code generated by this phase may be:

```
t1 = a*b
t2 = t1 + c
b = t2
```

Note: The above intermediate code is input to code optimizer which is the next phase of the compiler

Code optimization: This is the fifth phase of the compiler. The intermediate code generated in the previous phase is the input. The output is another intermediate code that does the same job as the original but saves time and space. That is, output of this phase is the optimized code.

For example, consider the statement:

```
a = b + c*d
e = f + c*d
```

The above two statements might be evaluated as:

```
t1 = c*d
a = b+t1
e=f+t1
```

Observe that computation of c*d is done only once. So, the code is optimized.

Code generation: This is the last phase of the compiler. The optimized code obtained from the previous phase is the input. It converts the intermediate code into equivalent assembly language instructions. This phase has to utilize the registers very efficiently to

generate the efficient code. For example, if the intermediate code generated by the previous phase is:

a = b + c

The code generated by this phase can be:

```
LOAD    B
ADD     C
STORE   A
```

Symbol table: This is the module that can be accessed by all phases of the compiler. The information of all the variables, constants along with type of variables and constants are stored here.

Error handler: This routine is invoked whenever an error is encountered in any of the phases. This routine attempts a correction so that subsequent statements need not be discarded and many errors can be identified in a single compilation.

1.2.2 The grouping of phases

In this section, let us see "What is the difference between a phase and pass?"

First, let us see "What is a phase?"

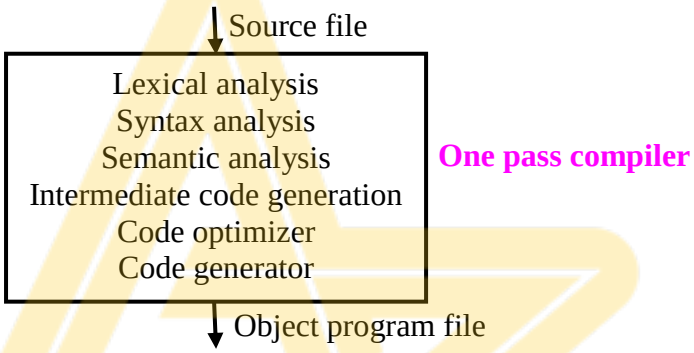
Definition: Since compiler is a very complex program, to understand the design of the compiler, it is logically divided into various sub-programs called modules. These sub-programs or *modules* are also called *phases*. Each phase or module accepts one representation of the program as the input and generates another representation of the program as the output. Thus, a *phase* is nothing but a module of the compiler that accepts one representation of the program as the input and produces another representation of the program as the output. The various phases of the compiler along with input and output are pictorially shown in page 1.9.

Now, let us see "What is a pass?"

Definition: A group of one or more phases of a compiler that perform *analysis* or *synthesis* of the source program is called a *pass* of the compiler. Even though the compiler is divided into various phases, in the actual implementation of the compiler, one or more phase may be grouped together into a pass. Each pass reads an input file and writes an output file. The compilers based on the number of passes are classified as shown below:

- One pass compiler
- Two pass compiler
- Three pass compiler
- Multipass compiler

- **One pass compiler:** If analysis and syntheses of source program is done in one flow from beginning to end of the program, then the compiler is called *one pass compiler.* In this case, all the phases of a compiler starting from *lexical analysis phase* to *code generation phase* are combined together as one unit to convert the source program into object program in one single flow. This is pictorially represented as shown below:



The single pass compiler or one pass compiler is beneficial because of the following reasons:

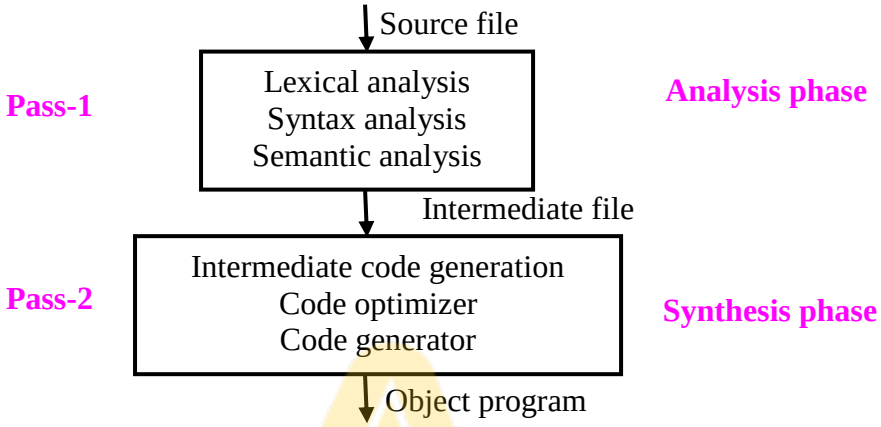
- Simplifies the job of writing a compiler
- One pass compilers generally perform compilations faster than multi-pass compilers

- **Two pass compiler:** If analysis is of source program is done in one pass and syntheses of source program is done in another pass, then the compiler is called *two pass compiler.*

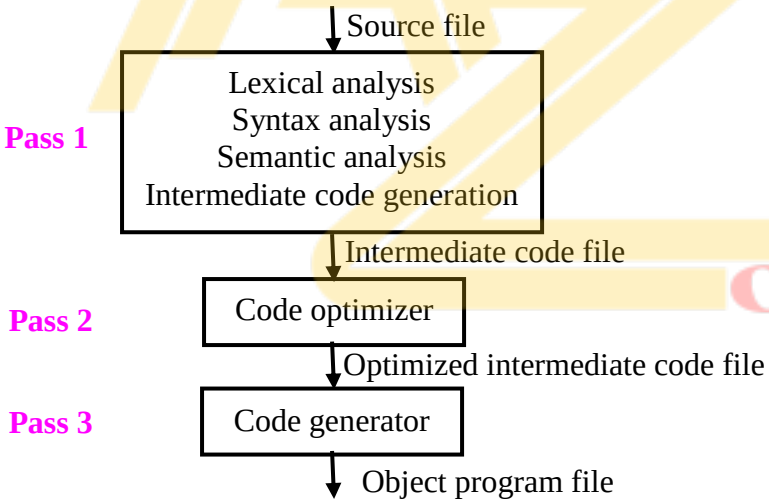
In this case, the first three phases i.e., *lexical analysis* phase, *syntax analysis* phase, *semantic analysis phase* are implemented as one unit and constitute *first pass of compiler*. This is often called *front-end of compiler.* The other three phases i.e., *intermediate code generation* phase, *code optimizing* phase and *code generation* phase are implemented as second unit and constitute *second pass of compiler*. This is often called *back-end of compiler.*

- The output of the first pass is normally stored a file.
- The second pass accepts this file as the input and produces the object program.

This is pictorially represented as shown below:



- **Three pass compiler:** If analysis and synthesis of source program is done in three different passes, then the compiler is called *three pass compiler.* In this case, the first four phases i.e., *lexical analysis* phase to *intermediate code generation phase* are implemented as one unit (pass), the *code optimizer* as second unit (pass) and *code generation* as third unit(pass). All the three passes are pictorially represented as shown below:



  - **Pass 1:** The front end of the phases such as *lexical analysis*, *syntax analysis*, *semantic analysis* and *intermediate code generation* phases may be grouped into one unit called pass1 of compiler. Pass 1 accepts the source program and produces the intermediate code file. This file is internal and stored in main memory. In some compilers, this file is also stored in the disk and hence it is visible when we list the directory after pass 1.

- **Pass 2:** The intermediate file created in pass 1 is input to pass 2. It produces another intermediate code file which is optimized. Each line in this file contains simple statements or expressions with maximum of 2 or 3 operands along with operators.

- **Pass 3:** The optimized intermediate file created in pass 2 is input to pass 3. It converts the intermediate code into assembly language program or the target program. It may have built in assembler that converts assembly language into machine code.

Note: Even though single pass compiler has some advantages, it has some disadvantages too. Some of the disadvantages are:

- It is not possible to perform many of the sophisticated optimizations needed to generate high quality code.
- It is difficult to count the number of passes made by an optimizing compiler. For example, one expression may be analyzed many times whereas another expression may be analyzed only once.

All the above disadvantages are overcome using multi-pass compiler. Now, let us see "What is a multi-pass compiler? Explain the need for multiple passes in compiler?"

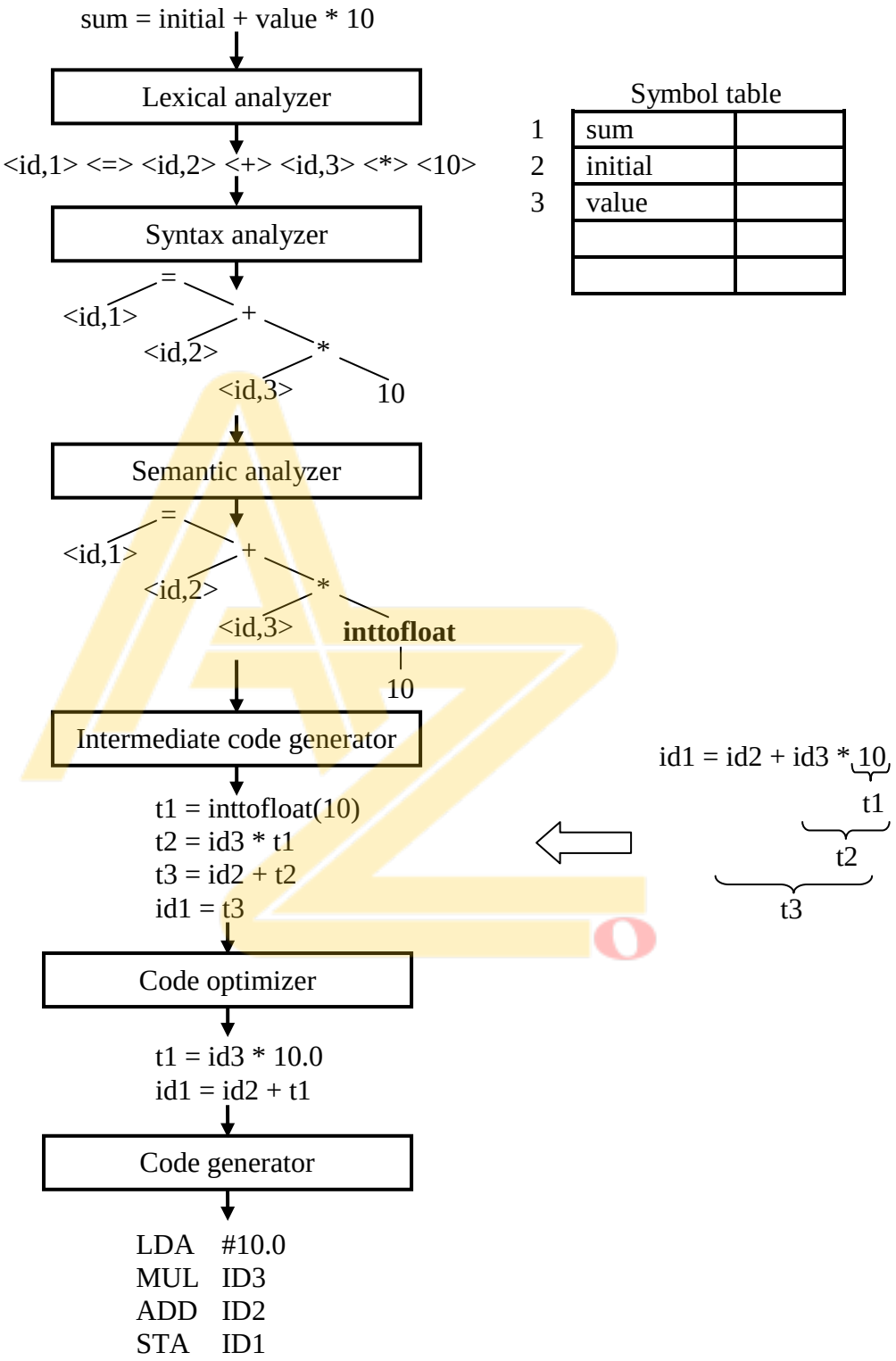
Definition: A multi-pass compiler is a type of compiler that processes the source code two or more times. The multi-pass compilers are slower, but much more efficient when compiling.

Multi-pass compilers are used for the following reasons:

- By changing the front-end of the compiler and retaining the back-end of the compiler for a particular target machine, it is possible to write compilers for different languages.
- On similar lines, by retaining the front-end and changing the back-end of the compiler, it is possible to produce a compiler for different target machines.
- If there is feature of a language that requires a compiler to perform more than one pass over the source, then multi-pass compiler is used.
- Splitting a compiler up into small programs is a technique used by researchers and designers interested in producing provably correct compilers. Proving the correctness of a set of small programs often requires less effort and time than proving the correctness of a larger, single, equivalent program.

Now, let us "Show the output of each phase of the compiler for the assignment statement: ***sum = initial + value * 10*** " The translation process is shown below:

1.16 💻 Lexical Analyzer



1.3 Compiler construction tools

In this section, let us see "What are compiler construction tools?" The compiler writer, like any software developer, can profitably use modern software environments containing tools such as editors, debuggers and so on. Some of the tools that help the programmer to build the compiler very easily and efficiently are listed below:

- **Parser generators:** They accept grammatical description of a programming language and produce syntax analyzers.
- **Scanner generators:** They accept regular expression description of the tokens of a language and produce lexical analyzers.
- **Syntax-directed translation engines:** They produce various routines for walking a parse tree and generating the intermediate code.
- **Code generator generators:** Using the intermediate code generated, they produce the target language. The target language may be object program or assembly language program.
- **Data flow analysis engines:** It gathers the information of how values are transmitted from one part of a program to other part of the program. Data flow analysis is key part of the code optimization
- **Compiler construction toolkits:** They provide a set of routines for constructing various phases of the compiler.

1.4 Lexical analysis

Now, let us see "What is lexical analysis?"

Definition: The process of reading the source program and converting it into sequence of tokens is called lexical analysis.

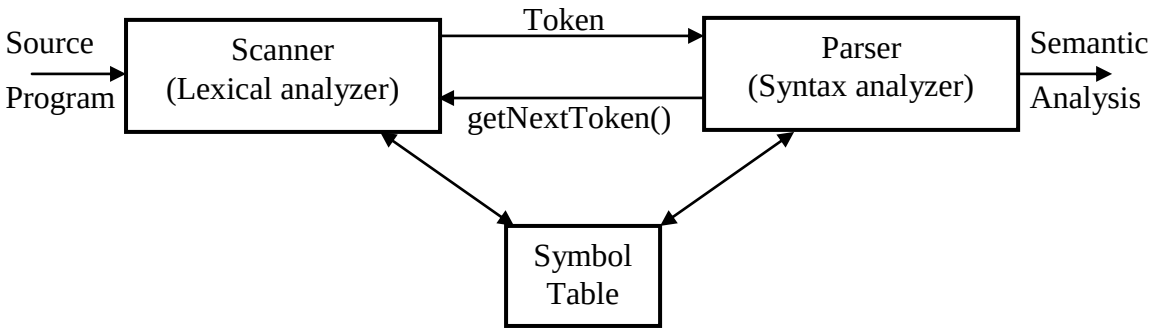
1.4.1 The role of lexical analyzer

Now, let us see "What is the role of lexical analyzer?" The lexical analyzer is the first phase of the compiler. The various tasks that are performed by the lexical analyzer are:

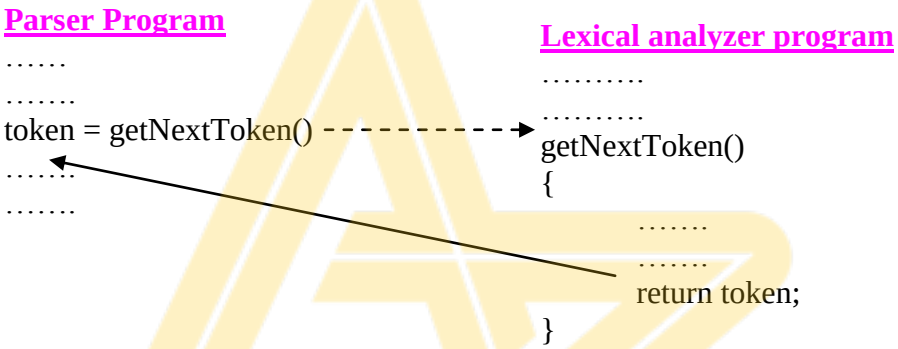
- Read a sequence of characters from the source program and produce the tokens.
- The tokens thus generated are sent to the parser for syntax analysis. The parser is also called syntax analyzer.
- During this process, lexical analyzer interacts with symbol table to insert various identifiers and constants. Sometimes, information of identifier is read from symbol table to assist in determining the proper token to send to the parser.

The interaction between the lexical analyzer and the parser is pictorially represented as shown below:

1.18 Lexical Analyzer



- The parser program calls the function getNextToken() which is the function defined in lexical analyzer (See the calling sequence below)



- The function getNextToken() of lexical analyzer returns the token back to parser for parsing.
- If the token obtained is an identifier, it is entered into the symbol table along with various attribute values and returns a token as a pair consisting of an integer code denoted by ID and a pointer to the symbol table for that identifier.
- The other actions that are performed by the parser are:
  - Removes comments from the program.
  - Remove white spaces such as blanks, tabs and newline characters from the source program and then tokens are obtained.
  - Keep track of line numbers so as to associate line numbers with error messages.
  - If any errors are encountered, the lexical analyzer displays appropriate error messages along with line numbers
  - Preprocessing may be done during lexical analysis phase

1.4.2 Lexical analysis verses parsing

Now, let us see "Why analysis portion of the compiler is separated into lexical analysis and syntax analysis phase?" The various reasons for which lexical analysis is separated from syntax analysis are shown below:

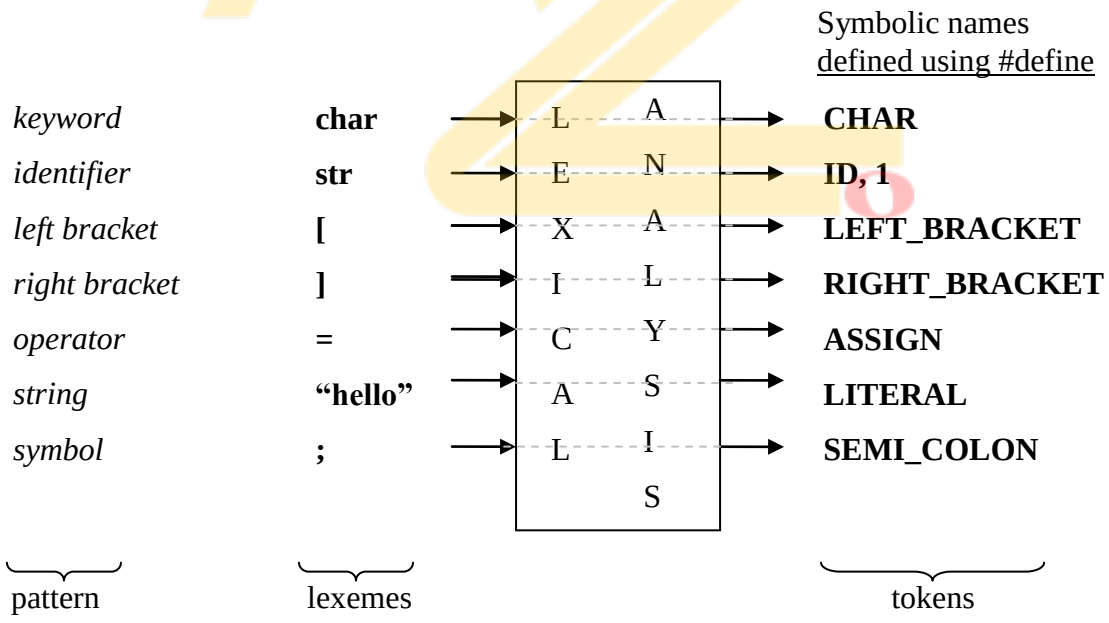
- **Simplicity of the design:** The separation of lexical analysis and syntax analysis allows us to simplify some of the tasks listed below:
  - A parser that deals with comments and white spaces would be more complex. So, to simplify this task and reduce the complexity of parser, removing of comments and white spaces is done by the lexical analysis
  - When designing a new language, the separation of first two phases results in cleaner overall language design
- **Compiler efficiency is improved:** A separate lexical analyzer allows us to use specialized techniques for lexical task and generation of tokens will be very easier and efficient. Also, specialized buffering techniques are used for reading the input characters which will speed up the compiler efficiently.
- **Portability enhancement:** By restricting the input-device-specific peculiarities, the compiler portability can be enhanced.

1.4.3 Tokens, patterns and lexemes

In this section, before proceeding further, let us see "What is the meaning of *patterns, lexemes* and *tokens*?" To explain the meaning of these three words, consider the following statement:

char str[] = "hello";

In the above statement, the *patterns*, *lexemes* and respective *tokens* are shown below:



1.20 💻 Lexical Analyzer

Observe that lexemes are input to lexical analyzer and lexical analyzer produces tokens. The description of a lexeme is a pattern. Now, let us study the meaning of token, pattern and lexemes in detail.

1.4.3.1 Tokens

Now, let us see "What is a token?"

Definition: A token is a pair consisting of *token name* and an optional *attribute value.* The *token names* are basically integer codes represented using symbolic names written in capital letters such as INT, FLOAT, SEMI_COLON etc (defined in the file token.h in next page). The attribute values are optional and will not be present for keywords, operators and symbols. The attribute values are present for all identifiers and constants.

- For every keyword there is a unique token name. For example, INT, FLOAT, CHAR, IF, WHILE etc
- For every symbol there is a unique token name. For example, SEMI_COLON, COLON, COMMA, LEFT_PARANTHESIS, RIGHT_PARANTHESIS, ASSIGN etc.
- For an identifier *sum*, the token is <ID, 1> where ID is the token name and 1 is the position of the identifier in the symbol table

Whenever there is a request from the parser, the lexical analyzer sends the token. So, tokens are output of the lexical analyzer and input to the syntax analyzer. The syntax analyzer uses these tokens to check whether the program is syntactically correct or not by deriving the tokens from the grammar. All the tokens are represented using symbolic constants defined using #define directive as shown below:

Note: A simple header file **"token.h"** which is shared by both lexical analyzer and syntax analyzer with various tokens and corresponding integer codes is shown below:

```
/* TOKENS with corresponding integer codes for keywords */
#define     IF          1
#define     ELSE        2
#define     INT         3
#define     CHAR        4
#define     DOUBLE      5
#define     FLOAT       6
……..
……..
……..
……..
```

```
/* TOKENS with corresponding integer codes for operators */
#define     PLUS          50
#define     MINUS         51
#define     SUBTRACT      52
#define     DIVIDE        53
.....
.....
/* TOKENS with corresponding integer codes for symbols */
#define     LEFT_BRACKET          70
#define     RIGHT_BRACKET         71
#define     LEFT_PARANTHESIS      72
#define     RIGHT_PARANTHESES     73
#define     SEMI_COLON            74
.....
.....
/* TOKENS with corresponding integer codes for identifiers and literals */
#define     ID            100
#define     LITERAL       101
```

1.4.3.2 Lexeme

Now, let us see "What is a lexeme?"

Definition: A sequence of characters in the source program that matches the patterns such as identifiers, numbers, relational operators, arithmetic operators, symbols such as #, [,], (,) and so on are called lexemes. In other words, a *lexeme* is a string of patterns read from the source file that corresponds to a token.

1.4.3.3 Patterns

Now, let us see "What is a pattern?"

Definition: The description of a lexeme is called pattern. More formally, a pattern is described as a rule describing set of lexemes. The various patterns are shown below:

- **Keyword:** The pattern *keyword* is a sequence of characters that form reserve words of a language. For example, **int**, **if**, **else**, **while**, **do**, **switch** etc are all reserve words. They are also called keywords
- **Identifier:** The pattern *identifier* is described a sequence of letters or underscores followed by any number of letters or digits or underscores. For example, *sum*, *i*, *pos*, *first*, *rate_of_interest* that represent variables in a program or that represent names of functions, structures etc. are all treated as *identifiers*.

- **Relational Operator:** The pattern *relational operators* which is described as the symbols that represent various relational operators of a language. For example: <, <=, > , >=, ==, != represent patterns identifying the relational operators.
- **Symbols:** The pattern *symbols* is described as set of symbols such as #, $, (,), [,], {, }, : and so on

---

Example 1.1: Identify lexemes and tokens in the following statement:

a = b * d;

Solution: The lexemes, patterns and tokens for the above expression are shown below:

- *a* is a lexeme matching the pattern *identifier* and returning the token <ID, 1> where ID is the token name and 1 is the position of identifier *a* in the symbol table
- = is a lexeme matching the *pattern* symbol '=' and returning the token ASSIGN
- *b* is a lexeme matching the pattern *identifier* and returning the token <ID, 2> where ID is the token name and 2 is the position of identifier *b* in the symbol table
- * is a lexeme matching the *pattern* symbol '*' and returning the token STAR
- *d* is a lexeme matching the pattern *identifier* and returning the token <ID, 3> where ID is the token name and 3 is the position of identifier *d* in the symbol table
- ; is a lexeme matching the pattern *semicolon* and returning the token SEMICOLON

Note: All capital letters in the above solution represent symbolic names of integers defined in the file "tokens.h" given in section 1.4.3.1 (page 1.20)

Example 1.2: Identify lexemes and tokens in the following statement:

printf("Simple Interest = %f\n", si);

Solution: The lexemes, patterns and tokens for the given **printf** statements are shown below:

- *printf* is a lexeme matching the pattern *identifier* and returning the token <ID, 1> where ID is the token name and 1 is the position of identifier *printf* in the symbol table
- The character '(' is a lexeme matching the pattern *symbol* and returning the token LEFT_PARANTHESES
- The sequence of characters "Simple Interest = %f\n" is a lexeme matching the pattern *string* and returning the token <LITERAL, 2> where LITERAL is the token name and 2 is the position of literal in the symbol table
- The character ',' is a lexeme matching the pattern *symbol* and returning the token COMMA

- *si* is a lexeme matching the pattern *identifier* and returning the token <ID, 3> where ID is the token name and 3 is the position of identifier *si* in the symbol table
- The character ';' is a lexeme matching the pattern *symbol* and returning the token SEMI_COLON

1.4.4 Attributes for tokens

We know that a token is a pair consisting of token name and optional attribute value. Now, let us see "What is the need for returning attributes for tokens along with token name?"

When more than one lexeme matches the pattern, the lexical analyzer must provide additional information about the type of lexeme matched, to subsequent compiler phases. For example,

the pattern for *identifier* matches both *sum* and *pos*

The lexical analyzer returns the token ID for both lexemes. For the token ID, we need to associate more information such as:

- lexemes (For example, *sum* and *pos*)
- the type of identifier (such as int, float, char, double etc.)
- the location at which the identifier found (say found in line number 10 and 20). This information is required to display appropriate error messages along with line numbers

All the above information must be kept in the symbol table. Thus, appropriate attribute value for the identifier is "pointer to symbol table entry for an identifier". But, w*e have used the position of identifier in the symbol table as the attribute value.*

It is very important for the code generator to know which lexeme was found in the source program to generate the actual code and to associate correct value to that variable. Thus, lexical analyzer returns token name and its attribute value. The token name is used by the parser to check for syntax errors, whereas the attribute value is used during translation into machine code.

Example 1.3: Give the token names and attribute values for the following statement:

E = M * C **2.

where ** in FORTRAN language is used as exponent operator.

Solution: The various token names and attribute values for the above statement are shown below:

<ID, 1000>
<ASSIGN>

<ID, 1010>
<STAR>
<ID, 1020>
<POWER>
<NUM, 2>

Symbol Table

	lexemes	type	line no.
1000	E	float	2
1010	M	float	2
1020	C	float	2

Example 1.4: Give the token names and attribute values for the following statement:

si = p * t * r / 100

Solution: Assume the variables *si*, *p*, *t*, and *r* are already declared and they are available in symbol table. The token names and associated attribute values for the above statement are shown below:

<ID, 1000>
<ASSIGN>
<ID, 1010>
<STAR>
<ID, 1020>
<STAR>
<ID, 1030>
<DIVIDE>
<NUM, 100>

Symbol Table

	lexemes	type	line no.
1000	si	float	2
1010	p	float	2
1020	t	float	2
1030	t	float	2

Note: In the above set of tokens, the second component of each pair such as 1000, 1010, 1020 and 1030 are the attribute values which contain the addresses to the symbol table.

1.5 Input buffering

Now, let us see "Why input buffering is required?" Input buffering is very essential for the following reasons:

- Since lexical analyzer is the first phase of the compiler, it is the only phase of the compiler that reads the source program character-by-character and consumes considerable time in reading the source program. Thus, the speed of lexical analysis is a concern while designing the compiler.
- Lexical analyzers may have to look one or more characters beyond the next lexeme before we have the right lexeme.

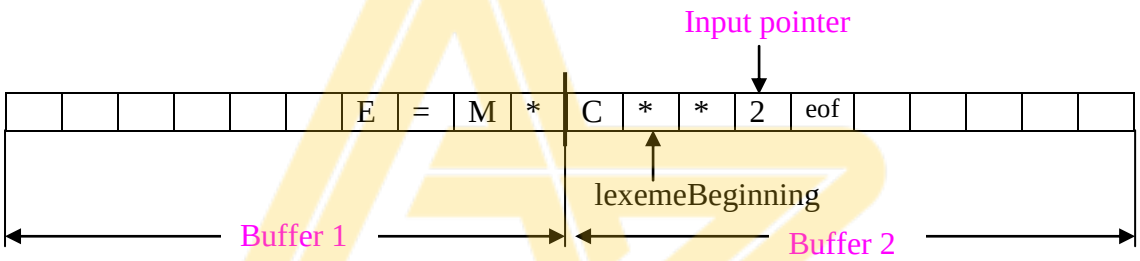
For this reason, we use the concept of input buffering where a block of 1024 or 4096 or more characters are read in one memory read operation and stored in the array to speed up the process.

Now, let us see "What is input buffering?"

Definition: The method of reading a block of characters (1K or 4K or more bytes) from the disk in one read operation and storing in memory (normally in the form of an array) for further processing and faster accessing is called *input buffering*. The memory (an array) where a block of characters read from the disk are stored is called *buffer*. Now, instead of reading character-by-character from the disk, we can directly read from the buffer to identify the lexemes for the tokens. This enhances the speed of the lexical analyzer as character-by-character reading from the disk consume considerable amount of time. The principle behind input buffering can be explained using:

- Buffer pairs
- Sentinel

Buffer pairs: In this input buffering technique two buffers are used as shown below:



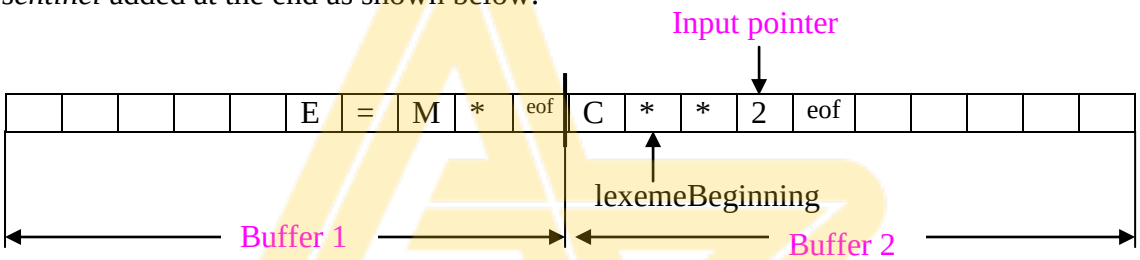
- The size of each buffer is N where N is usually the size of the disk block. If size of disk block is 4K, in one read operation 4096 characters can be read into the buffer using one system command rather than using one system call per character which consumes lot of time.
- If less than 4096 characters are present in the disk, then a special character represented by **eof** is stored in buffer indicating no-more characters are present in the disk file to read.
- Two pointers to the input are maintained as shown below:
  1) A pointer *lexemeBeginning* is used and it is pointing to the current lexeme.
  2) Another pointer called *input pointer* is used to scan ahead until the pattern is found.
- Now, all the characters starting from *lexemeBeginning* till the *input pointer* (excluding it) is the current lexeme. The token corresponding to this lexeme must be returned by the lexical analyzer.
- Once the token is found the pointer *lexemeBeginning* ponts to the input pointer to identify the next lexeme.
- As the *input pointer* moves forward, if end of buffer is encountered, read one more block whose size is N from the disk and load into other buffer.

Now, let us see "What is the disadvantage of input buffering with buffer pairs?" or "Explain the use of sentinels in recognizing the tokens" In input buffering with buffer pairs technique, each time we move the *input pointer* towards right, it indicates that we have not moved off the buffer. If we moved off the buffer, we have to load the other buffer. Thus, for each character read we have to do two tests:

- The first test is to determine the end of the buffer
- Second test is to determine what character is read.

It is possible to combine the buffer-end test with the test for the current character, if we extend the each buffer to hold a *sentinel* character at the end.

Sentinels: A *sentinel* is a special character that cannot be the part of the source program. So, the natural choice is to use a character **eof** which acts as *sentinel* as well as end of the input program. The input buffering technique with *sentinels* also uses two buffers with *sentinel* added at the end as shown below:



- The size of each buffer is N where N is usually the size of the disk block. If size of disk block is 4K, in one read operation 4096 characters can be read into the buffer using one system command rather than using one system call per character which consumes lot of time.
- Irrespective of number of characters stored in the buffer, last character of each buffer is *eof*.
- Note that **eof** retains its use as a marker for the end of the entire input. Any **eof** that appears other than at the end of a buffer means that the input is at an end.
- The use of two pointers *lexembeBeginning* and *input pointer* and the method of accessing *lexeme* remains same as in buffer pairs.
- The algorithm consisting of lookahead code with sentinels is shown below:

```
switch (*inputPointer++)
{
        case eof: If (inputPointer is at end of first buffer)
                {
                        reload the second buffer;
                        inputPointer = beginning of second buffer;
                        break;
                }
```

```
                    if (inputPointer is at the end of second buffer)
                    {
                        reload the first buffer;
                        inputPointer = beginning of first buffer;
                        break;
                    }

                    /* eof within a buffer indicates the end of the input */
                    /* So, terminate lexical analysis */
                    break;

            /* Cases for other characters */
    }
```

Observe from the above algorithm that instead of having two tests as in *buffer pair* technique, there is only one test i.e., testing the **eof** marker.

1.6 Specifications of tokens

The regular expressions are the notations for specifying the pattern. Even though they cannot express all possible patterns, they are very effective in specifying those types of patterns that are necessary for tokens. In the next section, let us see how to build lexical analyzers by converting regular expressions to finite automata. The definition of some of the terms are discussed here:

Definition: An alphabet is any finite set of symbols. The set of alphabets is denoted by the symbol $\sum$. Some of the alphabets are letters, digits and punctuations.

Ex 1: $\sum = \{0, 1\}$ is set of binary alphabet
Ex 2: $\sum = \{0, 1, 2\}$ is a set of ternary alphabet

Definition: A *string* over an alphabet is a finite sequence of symbols drawn from alphabets. The length of string *s* denoted by |s| gives the number of symbols in the string *s*. For example, let $\sum = \{a, b\}$. The various strings that are obtained from $\sum$ are shown below:

a, aa, ab, ba, bb, …….and so on

Definitions: A *language* is set of strings obtained from the alphabets denoted by $\sum$. For example, language consisting of one or more a's is shown below:

L = { a, aa, aaa, aaaa, ……..}

1.28 Lexical Analyzer

Definition: A regular expression is recursively defined as shown below:

1. ϕ is a regular expression denoting an empty language.
2. ε-(epsilon) is a regular expression indicates the language containing an empty string.
3. *a* is a regular expression which indicates the language containing only {a}
4. If R is a regular expression denoting the language L_R and S is a regular expression denoting the language L_S, then
    a. R+S is a regular expression corresponding to the language $L_R \cup L_S$.
    b. R.S is a regular expression corresponding to the language $L_R.L_S$.
    c. R* is a regular expression corresponding to the language L_R^*.
5. The expressions obtained by applying any of the rules from 1 to 4 are regular expressions.

The following table shows some examples of regular expressions and the language corresponding to these regular expressions.

Regular expressions	Meaning
a*	String consisting of any number of a's (or string consisting of zero or more a's)
a^+	String consisting of at least one a (or string consisting of one or more a's)
(a+b)	String consisting of either one *a* or one *b*
(a+b)*	Set of strings of a's and b's of any length including the NULL string.
(a+b)*abb	Set of strings of a's and b's ending with the string abb
ab(a+b)*	Set of strings of a's and b's starting with the string ab.
(a+b)*aa(a+b)*	Set of strings of a's and b's having a sub string aa.
a*b*c*	Set of string consisting of any number of a's(may be empty string also) followed by any number of b's(may include empty string) followed by any number of c's(may include empty string).
$a^+b^+c^+$	Set of string consisting of at least one 'a' followed by string consisting of at least one 'b' followed by string consisting of at least one 'c'.
aa*bb*cc*	Set of strings consisting of at least one 'a' followed by string consisting of at least one 'b' followed by string consisting of at least one 'c'.
(a+b)* (a + bb)	Set of strings of a's and b's ending with either *a* or *bb*

(aa)*(bb)*b	Set of strings consisting of even number of a's followed by odd number of b's
(0+1)*000	Set of strings of 0's and 1's ending with three consecutive zeros(or ending with 000)
(11)*	Set of strings consisting of even number of 1's
01* + 1	The language represented is the string 1 plus the string consisting of a zero followed by any number of 1's possibly including none.
(01)* + 1	The language consists of a string 1 or strings of (01)'s that repeat zero or more times.
0(1* + 1)	Set of strings consisting of a zero followed by any number of 1's
(1+ε)(00*1)*0*	Strings of 0's and 1's without any consecutive 1's
(0+10)*1*	Strings of 0's and 1's ending with any number of 1's (possibly none)
(a+b)(a+b)	Strings of a's and b's whose length is 2

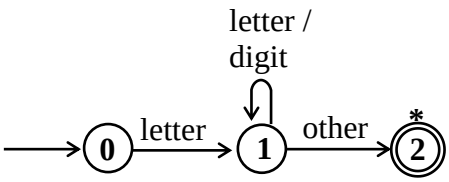
1.7 Recognition of tokens and construction of lexical analyzer

The finite automaton represented using transition diagram can be easily written using which the tokens can be obtained. Using this transition diagram, we can easily construct the lexical analyzer. Now, let us see how to write the lexical analyzers by looking into transition diagrams. The two functions that are used while designing the lexical analyzer are:

- **retract():** This function is invoked only if we want to unread the last character read. It is identified by having a an edge labeled other to the final state with * marked to it. This function un reads the last character read
- **install_ID():** Once the identifier is identified, the function install_ID() is called. This function checks whether the identifier is already there in the symbol table. If it is not there, it is entered into the symbol table and returns the pointer to that entry. If it is already there, it returns the pointer to that entry.

Example 1.5: Design a lexical analyzer to identify an identifier

The transition diagram to identify an identifier is shown below:



Observe the following points from the above transition diagram:

State 0: If input symbol is a letter goto state 1

State 1: If input symbol is a letter or digit remain in state 1. But, any character other than letter and digit go to state 2.

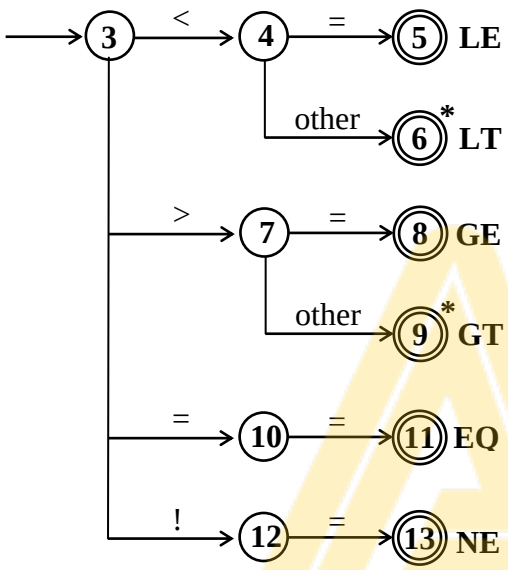
State 2: Since other character is read before coming to state 2, push back the other character using the function retract() and return the token ID and pointer to symbol table entry for the identifier recognized by calling the function install_ID();

So, the complete lexical analyzer to identify the identifier using the above procedure is shown below:

```
state = 0;

for (;;)
{
        switch (state)
        {
                case 0:
                        ch = getchar()

                        if (ch == letter)  state = 1;
                        else state = 3;          // Identify the next token

                        break;

                case 1:
                        ch = getchar()

                        if (ch == letter or ch == digit) state = 1;
                        else state = 2;

                        break;

                case 2: retract();                      // undo the last character read
                        return ( ID, install_ID() )

                case 3: /* Identify the next token */
        }
}
```

Example 1.6: Design a lexical analyzer to identify the relation operators such as <, <=, >, >=, == and !=

Note: Observe from the previous problem that, states 0, 1 and 2 are used to recognize the token corresponding to an identifier. But, state 3 onwards, other tokens are identified. Now, we start from state 3 to recognize other tokens such as relational operators as shown in following transition diagram.



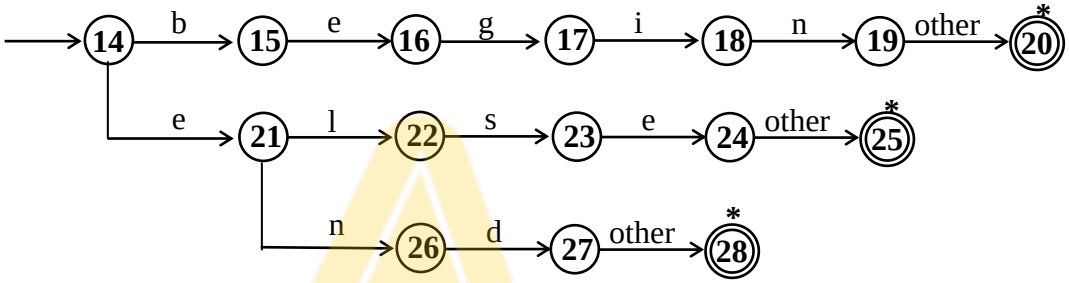
In the above transition diagram, observe that there are no transitions defined for states 3, 10 and 12 for symbols other than specified as labels. So, from state 3, 10 and 12 on any *other* symbol, we enter into state 14 which is used to identify some other token. The lexical analyzer for identifying relational operators is shown below:

```
state = 0;

for (;;)
{
        switch (state)
        {
                /* Case 0-2: Identify the identifier */
                case 3:
                        ch = getchar()

                        if (ch == '<')          state = 4;
                        else if (ch == '>')     state = 7;
                        else if (ch == '=')     state = 10;
                        else if (ch == '!')     state = 12;
                        else                    state = 14;

                        break;

                case 4:
                        ch = getchar()
```

```
                    if (ch == '=')  state = 5;
                    else            state = 6;

                    break;

                case 5: return LE;

                case 6: retract();              // unread the last character read
                    return LT;

                case 7:
                    ch = getchar()

                    if (ch == '=')  state = 8;
                    else            state = 9;

                    break;

                case 8: return GE;

                case 9: retract();              // unread the last character read
                    return GT;

                case 10:
                    ch = getchar()

                    if (ch == '=')  state = 11;
                    else            state = 14;

                    break;
                case 11: return EQ;

                case 12:
                    ch = getchar()

                    if (ch == '=')  state = 13;
                    else            state = 14;

                    break;

                case 13: return NE;

                case 14: /* Identify the next token */
            }
        }
```

Example 1.7: Design a lexical analyzer to identify the keywords **begin**, **else, end.** Sketch the program segment to implement it showing the first two states and one final state

Note: Observe from the previous problem that, states 0 to 13 are used to identify the identifier and relational operators. Let us identify the keywords **begin**, **else**, **end** using the following transition diagram starting from state 14 as shown below:

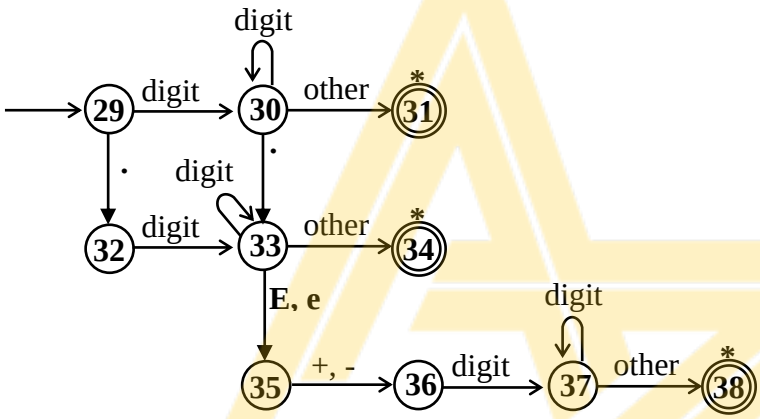


Similar to the previous two problems, we can write various cases to identify the tokens BEGIN, ELSE and END. The lexical analyzer is implemented by writing the code for first two states and one final state as shown below:

```
state = 0;
for (;;)
{
        switch (state)
        {
                /* Case 0-13: Identify other tokens */
                case 14:
                        ch = getchar()
                        if (ch == 'b')
                                state = 15;
                        else if (ch == 'e')
                                state = 21;
                        else
                                state = 29;      /* other token */

                case 15:
                        ch = getchar();
                        if (ch == 'e')
                                state = 16;
                        else
                                state = 29;      /* other token */
```

```
                    break;
            …….
            …….
            case 28: retract();
                    return END;
        }
}
```

Example 1.8: Design a lexical analyzer to recognize unsigned number. Sketch the program segment to implement it showing the first two states and one final state



Similar to the previous two problems, we can write various cases to identify unsigned integers and floating point numbers represented using decimal point and scientific method. The lexical analyzer is implemented by writing the code for first two states and one final state as shown below:

```
state = 0;
for (;;)
{
        switch (state)
        {
                /* Case 0-28: Identify other tokens */
                case 29:
                        ch = getchar()

                        if (ch == digit)
                                state = 30;
                        else if (ch == '.')
                                state = 32;
```

```
                    else
                            state = 39;       /* other token */
                    break;
            case 30:
                    ch = getchar();

                    if (ch == digit)
                            state = 30;
                    else if (ch == '.')
                            state = 33;
                    else
                            state = 31;
                    break;
            .......
            .......
            case 31: retract();
                     return (NUM, InstallNUM() );
        }
    }
```

Note: Here, Install_Num() is a function which checks whether the number is already in the numeric table. If it is not present, it is entered into the numeric table and returns the token NUM and pointer to that integer as the attribute value. Otherwise, it returns the pointer to that integer as the attribute value apart from returning the token NUM.

Exercises

1) What is a language processor? What are the various types of language processors?
2) What is a compiler? What are the activities performed by the compiler?
3) What is an interpreter? What are the differences between a compiler and interpreter
4) What is an assembler? What are the activities performed by the compiler?
5) What is hybrid compiler?
6) What are the cousins of the compiler? or Explain language processing system
7) Discuss the analysis of source program with respect to compilation process
8) What are the different phases of the compiler? or "Explain the block diagram of the compiler construction method

1.36 💻 Lexical Analyzer

9) Show the output of each phase of the compiler for the assignment statement: ***sum = initial + value * 10***

10) What is the difference between a phase and pass? What is a multi-pass compiler? Explain the need for multiple passes in compiler?

11) What are compiler construction tools?

12) What is lexical analysis? What is the role of lexical analyzer?

13) Why analysis portion of the compiler is separated into lexical analysis and syntax analysis phase?

14) What is the meaning of *patterns*, *lexemes* and *tokens*?

15) Identify lexemes and tokens in the following statement:
a = b * d;

16) Identify lexemes and tokens in the following statement:
printf("Simple Interest = %f\n", si);

17) What is the need for returning attributes for tokens along with token name?"

18) Give the token names and attribute values for the following statement:
E = M * C **2.
where ** in FORTRAN language is used as exponent operator.

19) Give the token names and attribute values for the following statement:
si = p * t * r / 100

20) Why input buffering is required? What is input buffering?

21) What is the disadvantage of input buffering with buffer pairs? Explain the use of sentinels in recognizing the tokens.

22) Design a lexical analyzer to identify an identifier

23) Design a lexical analyzer to identify the relation operators such as <, <=, >, >=, == and !=

24) Design a lexical analyzer to identify the keywords **begin**, **else, end.** Sketch the program segment to implement it showing the first two states and one final state

25) Design a lexical analyzer to recognize unsigned number. Sketch the program segment to implement it showing the first two states and one final state