UNIT - I

Introduction and Lexical Analysis: Language Processors, the structure of a compiler, the science of building a compiler, phases of a compiler. Lexical Analysis: The role of the lexical analyzer, identifying tokens, Transition diagrams for recognizing tokens, Input buffering, The lexical analyzer generator Lex, Finite automata, Conversion from regular expressions to automata, design of a lexical analyzer generator, Optimization of DFA-based pattern matchers.

Language Processors

Preprocessor:

A preprocessor produce input to compilers. A preprocessor takes the skeletal source program as input and produces an extended version of it, which is the resultant of expanding the Macros, manifest constants if any, and including header files etc in the source file.

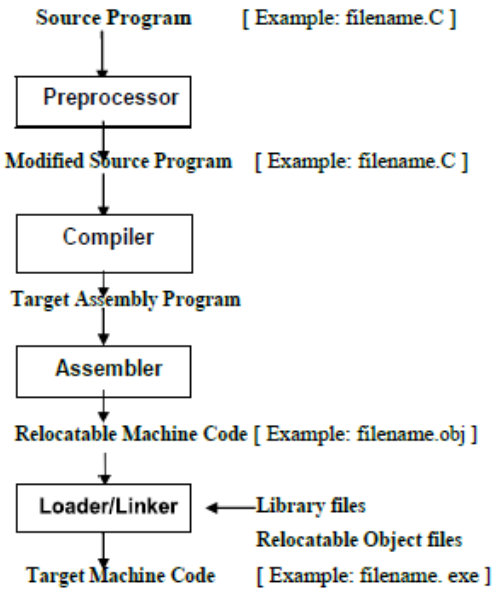
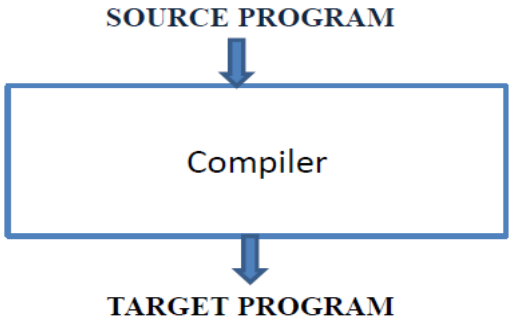


Fig: Process of Execution of a Program

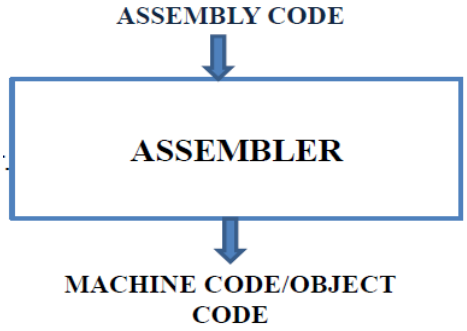
Compiler:

Compiler Is a translator that takes as input a source program written in high level language and converts it into its equivalent target program in machine language. The source code is translated to object code successfully if it is free of errors. The compiler specifies the errors at the end of compilation with line numbers when there are any errors in the source code.



Assembler:

The Assembler is a program that takes as input an assembly language program and converts it into its equivalent machine language code.



Loader / Linker: This is a program that takes as input a relocatable code and collects the library functions, relocatable object files, and produces its equivalent absolute machine code.

Specifically,

Loader translates the relocatable machine code, altering the relocatable addresses, and placing the altered instructions and data in memory at the proper locations.

Linker is used to make a single file from several files of relocatable machine code.

Interpreter: An interpreter is another commonly used language processor. The translation of single statement of source program into machine code is done by language processor and executes it immediately before moving on to the next line is called an interpreter.

- If there is an error in the statement, the interpreter terminates its translating process at that statement and displays an error message.
- The interpreter moves on to the next line for execution only after removal of the error.
- An Interpreter directly executes instructions written in a programming or scripting language without previously converting them to an object code or machine code.

Comparison Between Compiler and Interpreter:

Aspect	Compiler	Interpreter
Execution	Converts the entire program into machine code before execution.	Translates and executes code line by line.
Output	Typically produces an executable file that can be run independently.	Does not produce an independent executable file.
Speed	Generally faster execution since the code is precompiled.	Slower execution because it translates code on-the-fly.
Debugging	More challenging to debug, as errors are detected after compilation.	Easier to debug, as errors are reported as they occur during interpretation.
Memory Usage	Requires more memory to store the compiled code and data.	Typically uses less memory because it doesn't store compiled code.
Portability	Platform-specific, as the compiled code is generated for a particular platform.	More portable since it can be run on any platform with the appropriate interpreter.
Examples	C, C++, Java, Rust.	Python, Ruby, JavaScript, Perl.

The Science of building a Compiler

A compiler must accept all source programs that conform to the specification of the language; the set of source programs is infinite and any program can be very large, consisting of possibly millions of lines of code. Any transformation performed by the compiler while translating a source program must preserve the meaning of the program being compiled. Compiler writers thus have influence over not just the compilers they create, but all the programs that their compilers compile. This leverage makes writing compilers particularly rewarding; however, it also makes compiler development challenging.

Modelling in compiler design and implementation: The study of compilers is mainly a study of how we design the right mathematical models and choose the right algorithms. Some of most fundamental models are finite-state machines and regular expressions. These models are useful for de-scribing the lexical units of programs (keywords, identifiers, and such) and for describing the algorithms used by the compiler to recognize those units. Also

among the most fundamental models are context-free grammars, used to describe the syntactic structure of programming languages such as the nesting of parentheses or control constructs. Similarly, trees are an important model for representing the structure of programs and their translation into object code.

The science of code optimization: The term "optimization" in compiler design refers to the attempts that a com-piler makes to produce code that is more efficient than the obvious code. In modern times, the optimization of code that a compiler performs has become both more important and more complex. It is more complex because processor architectures have become more complex, yielding more opportunities to improve the way code executes. It is more important because massively parallel computers require substantial optimization, or their performance suffers by orders of magnitude.

Compiler optimizations must meet the following design objectives:

1. The optimization must be correct, that is, preserve the meaning of the compiled program,

2. The optimization must improve the performance of many programs,

3. The compilation time must be kept reasonable, and

4. The engineering effort required must be manageable.

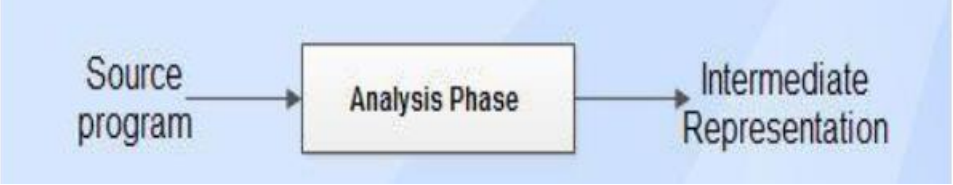
Phases of a compiler:

The compilation process contains the sequence of various phases. Each phase takes source program in one representation and produces output in another representation. Each phase takes input from its previous stage.

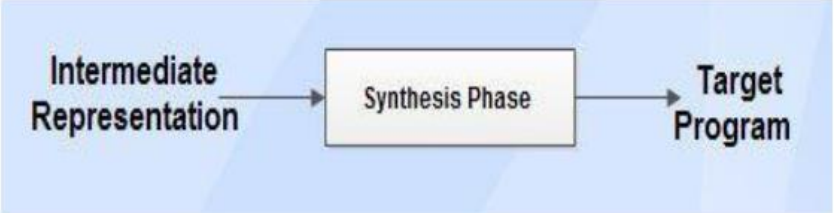
There are two parts of Compilation:

1. Analysis (Front End)
2. Synthesis (Back End)

The Analysis part breaks the source program into constituent pieces and creates an intermediate representation of source program.



The Synthesis part Construct the desired target program from the Intermediate representation.



The different phases of compiler are:

A compiler can have many phases and passes.

Pass: A pass refers to the traversal of a compiler through the entire program.

Phase: A phase of a compiler is a distinguishable stage, which takes input from the previous stage, processes and yields output that can be used as input for the next stage.

1. Lexical analysis
2. Syntax analysis
3. Semantic analysis
4. Intermediate code generator
5. Code optimizer
6. Code generator

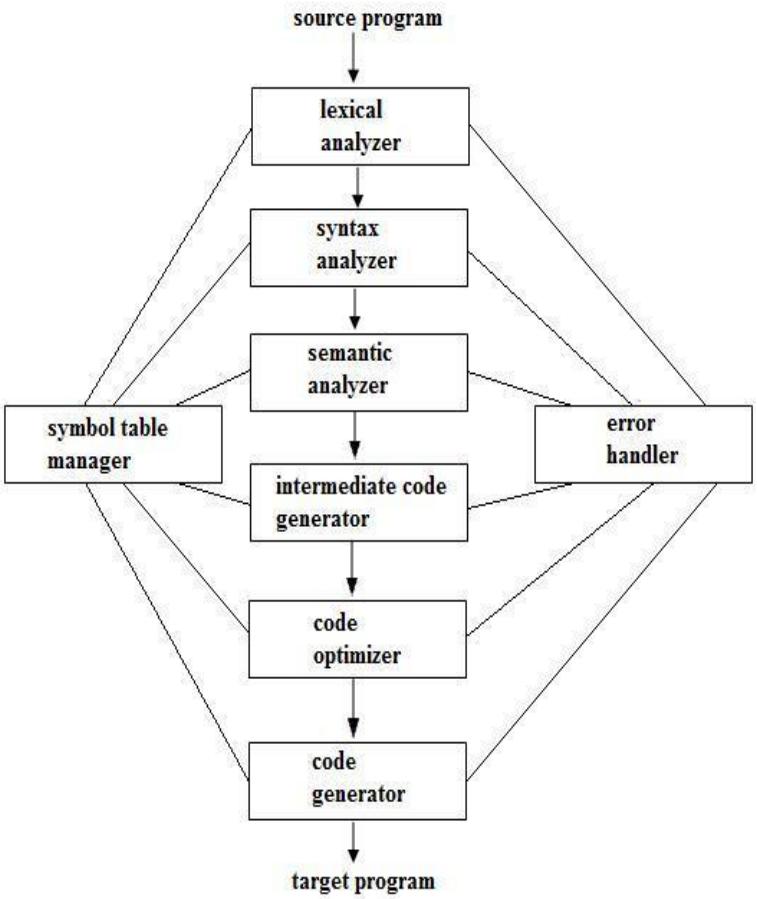


Fig. Phases of Compiler

All of the above mentioned phases involve the following tasks:

- Symbol table management.
- Error handling.

Lexical Analysis:

- Lexical analysis is the first phase of compiler which is also termed as scanning.
- Source program is scanned to read the stream of characters and those characters are grouped to form a sequence called lexemes which produces token as output.
- Once a token is generated the corresponding entry is made in the symbol table.

Syntax Analysis:

- Syntax analysis is the second phase of compiler which is also called as **parsing.**
- Parser converts the tokens produced by lexical analyzer into a tree like representation called **parse tree**.
- A parse tree describes the syntactic structure of the input.
- Syntax tree is a compressed representation of the parse tree in which the **operators** appear as **interior nodes** and the **operands** of the operator are the **children of the node for that operator**.

Semantic Analysis:

- Semantic analysis is the third phase of compiler.
- It checks for the semantic consistency.
- Type information is gathered and stored in symbol table or in syntax tree.
- Performs type checking.

Intermediate Code Generation:

- Intermediate code generation produces intermediate representations for the source program which are of the following forms:
- Postfix notation
- Three address code
- Syntax tree

Most commonly used form is the three address code. **Three address code** is a type of intermediate **code** which is easy to generate and can be easily converted to machine **code**. It makes use of at most **three** addresses or operands and one operator to represent an expression and the value computed at each instruction is stored in temporary variable generated by **compiler**.

Code Optimization:

- Code optimization phase gets the intermediate code as input and produces optimized intermediate code as output.
- It can be done by reducing the number of lines of code for a program.
- During the code optimization, the result of the program is not affected.

Code Generation:

- Code generation is the final phase of a compiler.
- It gets input from code optimization phase and produces the target code /object code as result.
- Intermediate instructions are translated into a sequence of machine instructions or assembly code that perform the same task.

Symbol Table Management:

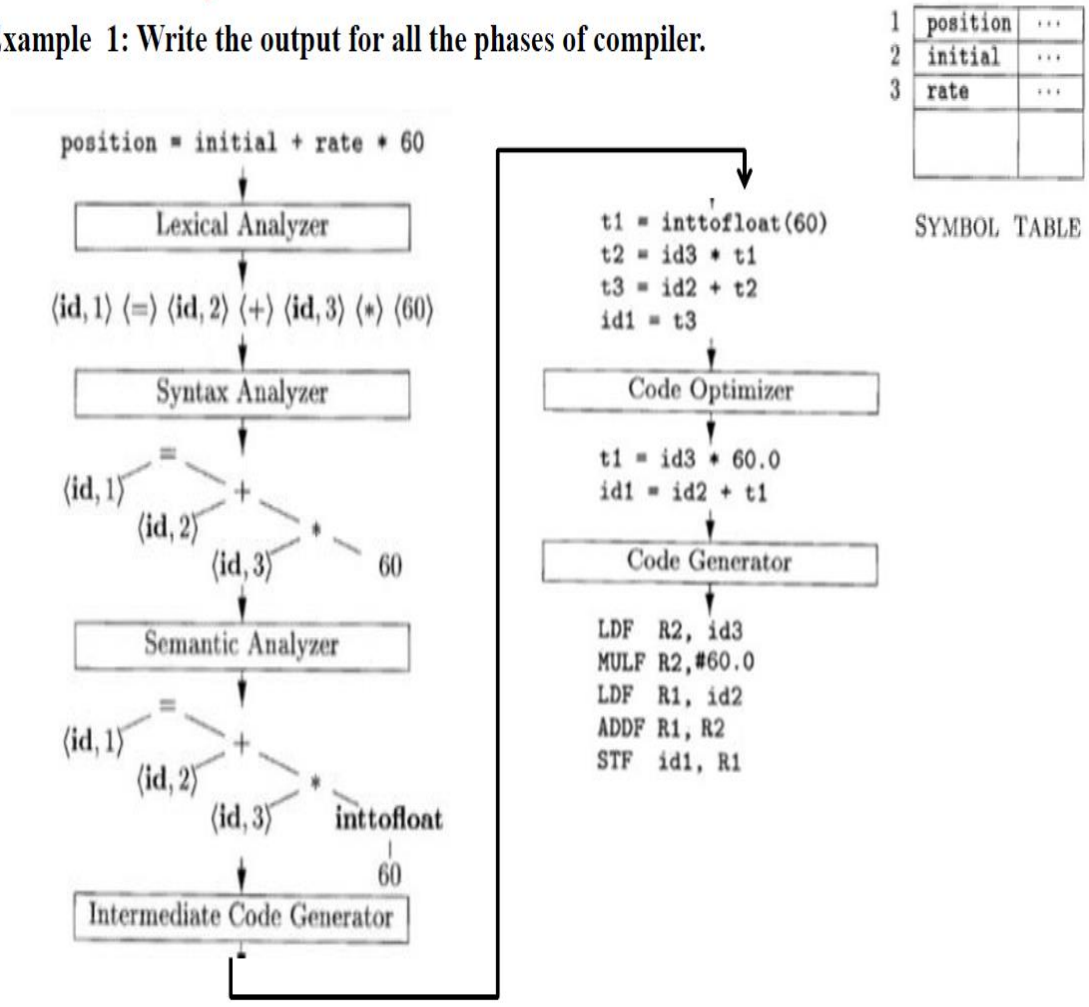
- Symbol table is used to store all the information about identifiers used in the program.
- It is a data structure containing a record for each identifier, with fields for the attributes of the identifier.
- It allows finding the record for each identifier quickly and to store or retrieve data from that record.
- Whenever an identifier is detected in any of the phases, it is stored in the symbol table.

Error Handling:

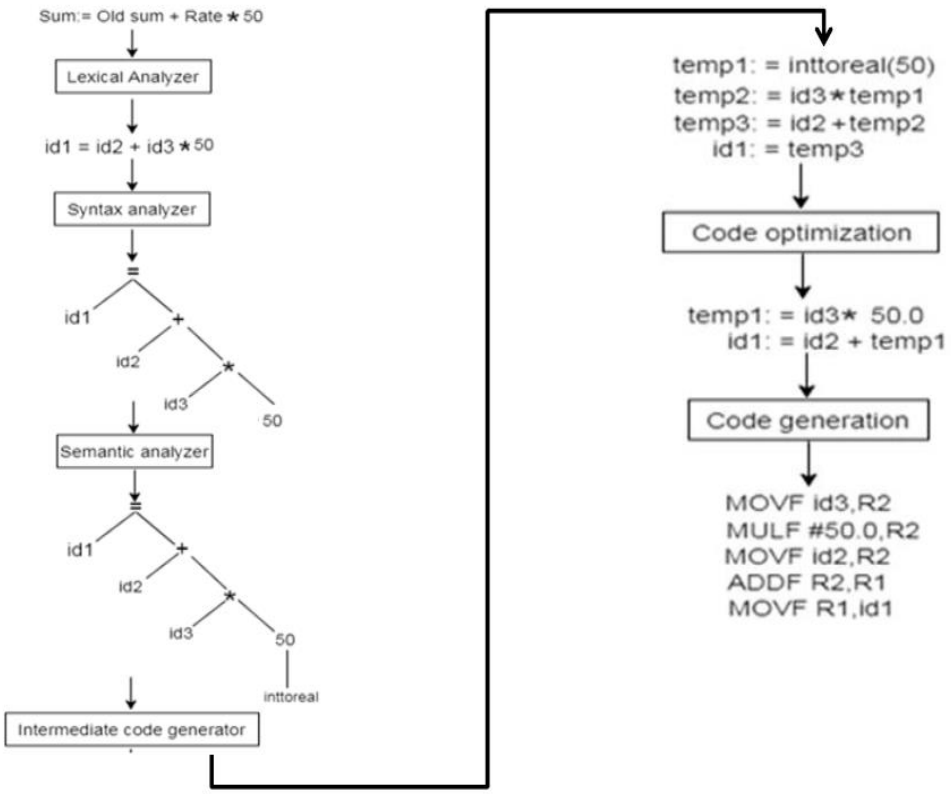
- Each phase can encounter errors. After detecting an error, a phase must handle the error so that compilation can proceed.
- In lexical analysis, errors occur in separation of tokens.
- In syntax analysis, errors occur during construction of syntax tree.
- In semantic analysis, errors may occur at the following cases:
  - When the compiler detects constructs that have right syntactic structure but no meaning
  - During type conversion.

Phases of Compilation

Example 1: Write the output for all the phases of compiler.



Example 2:

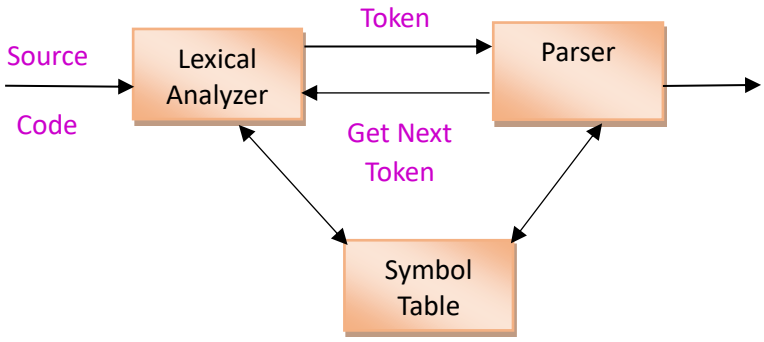


Lexical Analysis:

As the first phase of a compiler, the main task of the lexical analyzer is to read the input characters of the source program, group them into lexemes, and produce as output tokens for each lexeme in the source program. This stream of tokens is sent to the parser for syntax analysis. It is common for the lexical analyzer to interact with the symbol table as well.

The role of the Lexical Analyzer:

Lexical Analyzer is the first phase of compiler. The Lexical Analyzer works in two phases: In first phase it performs scan and in the second phase it does lexical analysis; means it generates the series of tokens. The Lexical Analyzer reads the input source program from left to right one character at a time and generates the sequence of tokens. Each token is a single logical cohesive unit such as identifier, keywords, operators, and constants. Then the parser to determine the syntax of the source program can use these tokens. The role of Lexical Analyzer in the process of compilation is shown below:



As the Lexical Analyzer scans the source program to recognize the tokens it is also called as Scanner. Apart from token identification Lexical Analyzer also performs following functions:

Functions of Lexical Analyzer:

1. It produces stream of Tokens.
2. It eliminates blank and comments.
3. It generates symbol table which stores the information about identifiers, constants, encountered in the input.
4. It keeps track of line numbers.
5. It reports the errors encountered while generating the tokens.

Lexical Errors:

- Typical Lexical phase errors are:

- Exceeding length of the identifier or numeric constants.
- Appearance of illegal characters.
- Unmatched String.

Identifying Tokens

Lexeme: Sequence of characters in the source program that are matched with pattern of the token is called Lexeme.

Token: Token describes the class or category of input string.

Pattern: Pattern is the set of rules that describes the token.

Example:

Identify the lexemes that make up the tokens in the following program segment. Indicate corresponding token and pattern.

```
void swap (int i, int j)
{
        int t;
        t=i;
        i=j;
        j=t;
}
```

Sol.

Lexeme	Token
void	keyword
swap	identifier
(	operator
int	keyword
i	identifier
,	operator
int	keyword
j	identifier
)	operator
{	identifier
int	keyword
t	identifier
t	identifier
=	operator

Lexeme	Token
i	identifier
i	identifier
=	operator
j	identifier
j	identifier
=	operator
+	operator
}	operator

Patterns:

1. Identifier:

a) Identifier is a collection of letters.

b) Identifier is a collection of alphanumeric characters.

c) The first character of identifier must be a letter.

2. Operator:

a) Operator can be arithmetic, logical, relational operators.

b) The parenthesis are considered as operators.

c) Comma is treated as separation operator.

d) Assignment is denoted by operator.

3. Keyword:

a) Keyword are special words to which some meaning is associated with.

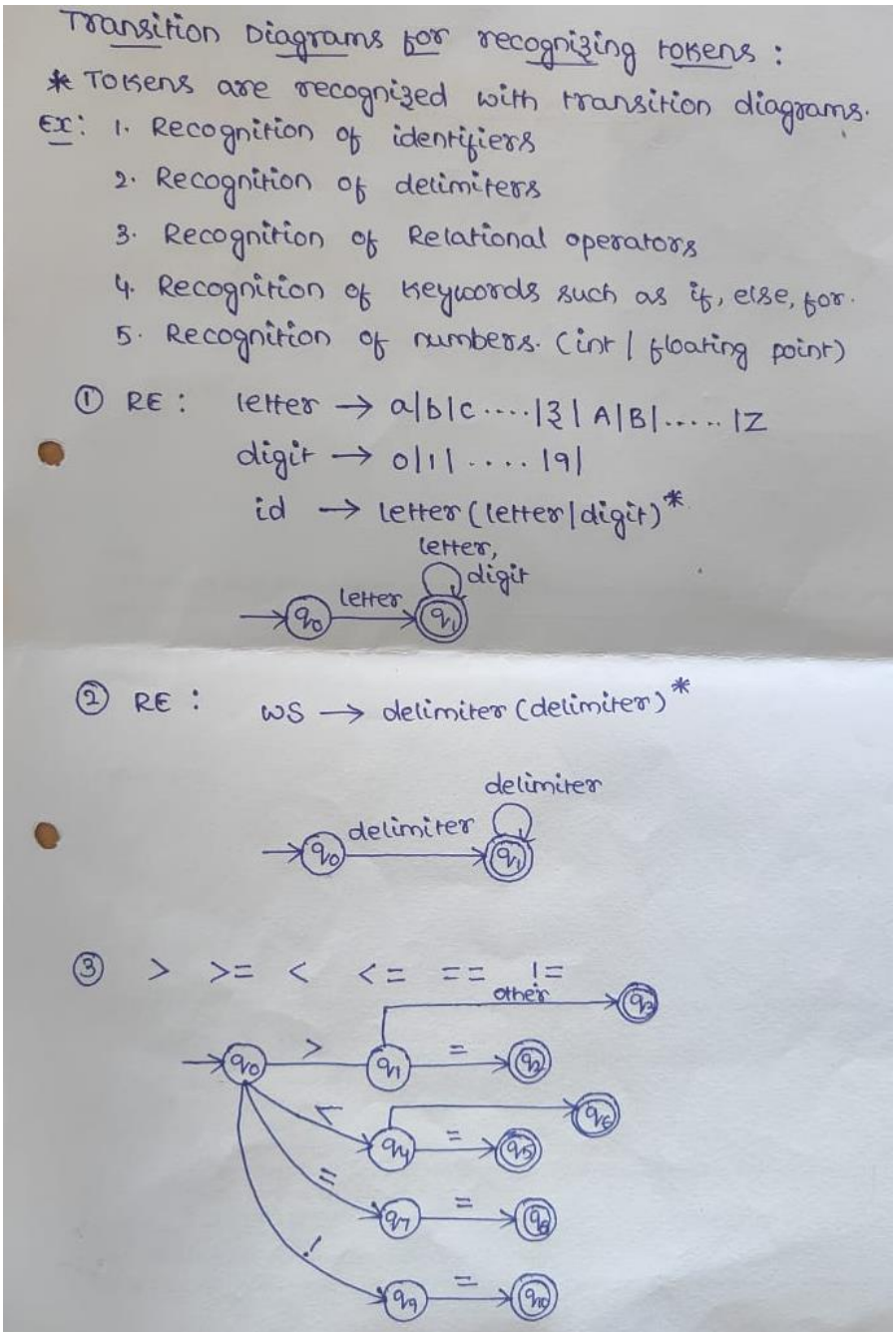
b) int, void are keywords for denoting data types.

Recognition of Tokens:

The token is usually represented by a pair token type and token value.

Token Type	Token Value

Transition diagrams for recognizing tokens:



Input Buffering:

Input buffering in compiler design is a technique where the compiler reads the source code into a buffer (a temporary storage area) to improve efficiency during lexical analysis. Instead of reading characters one by one, which can be slow, the compiler reads larger chunks of code into the buffer, reducing the number of system calls to read from the source file. This speeds up the lexical analysis phase, which identifies tokens (keywords, identifiers, etc.) in the code.

The Lexical Analyzer scans the input string from left to right one character at a time. It uses two pointers begin_ptr (bp) and forward_ptr (fp) to keep track of the portion of the input scanned. Initially both the pointers point to the first character of the input string as shown below:

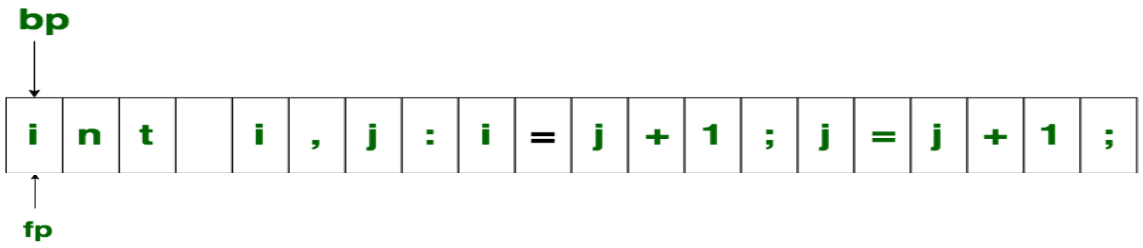


Fig. Initial Configuration

The forward_ptr moves ahead to search for end of lexeme. As soon as the blank space is encountered, it indicates end of lexeme. In above example as soon as forward_ptr (fp) encounters a blank space the lexeme “int” is identified.

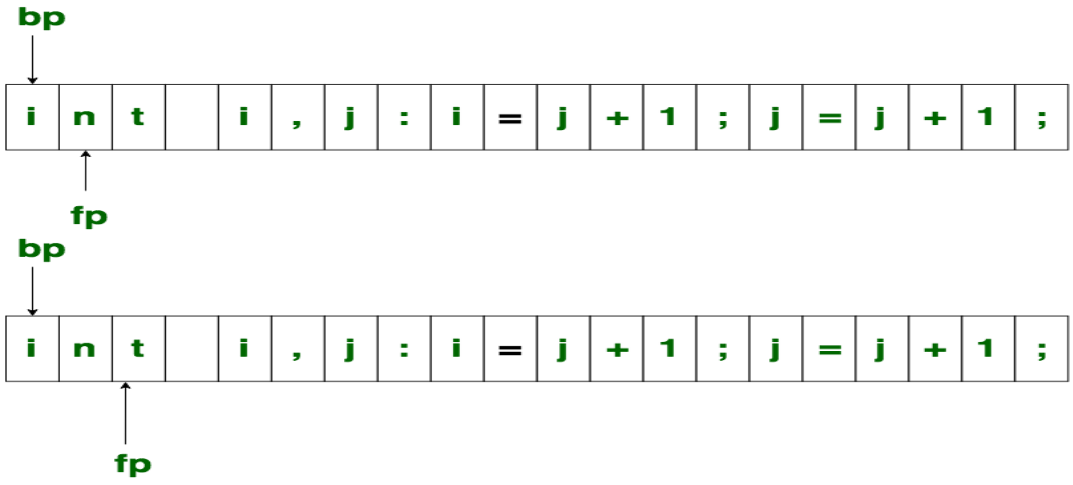


Fig. Input Buffering

The fp will be moved ahead at white space. When fp encounters white space, it ignores and moves ahead. Then both the begin_ptr (bp) and forward_ptr (fp) are set at next token i.

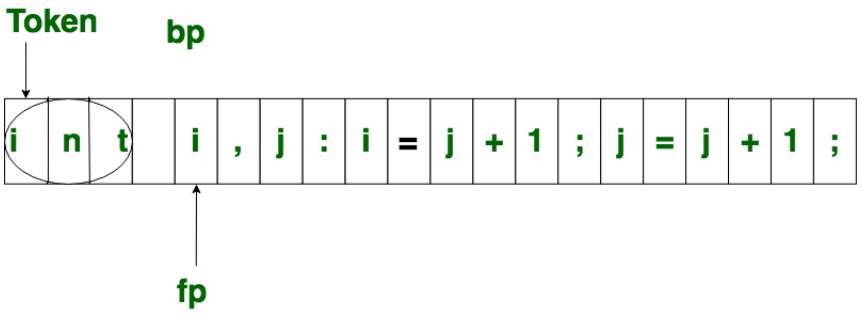


Fig. Input Buffering

The input character is thus read from secondary storage. But reading in this way from secondary storage is costly. Hence buffering technique is used. A block of data is first read into a buffer, and then scanned by lexical analyzer. There are two methods used in this context: one buffer scheme and two buffer scheme.

1. **One Buffer Scheme**

   In this one buffer scheme, only one buffer is used to store the input string. But the problem with this scheme is that if lexeme is very long then it crosses the buffer boundary, to scan rest of the lexeme the buffer has to be refilled, that makes overwriting the first part of lexeme.

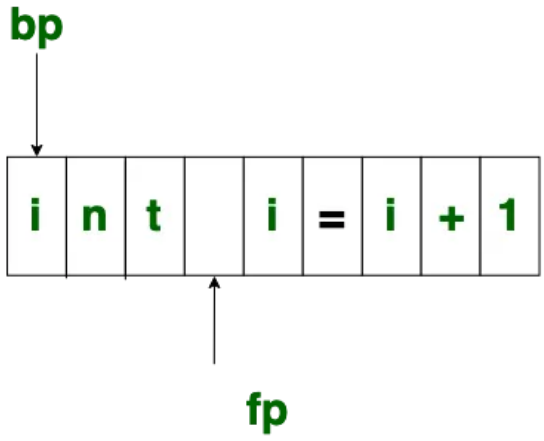


Fig. One buffer scheme storing input string

2. **Two Buffer Scheme**

   To overcome the problem of one buffer scheme, in this method two buffers are used to store the input string. The first buffer and second buffer are scanned alternately. When end of current buffer is reached the other buffer is filled. The only problem with this method is that if length of the lexeme is longer than length of the buffer then scanning input cannot be scanned completely.

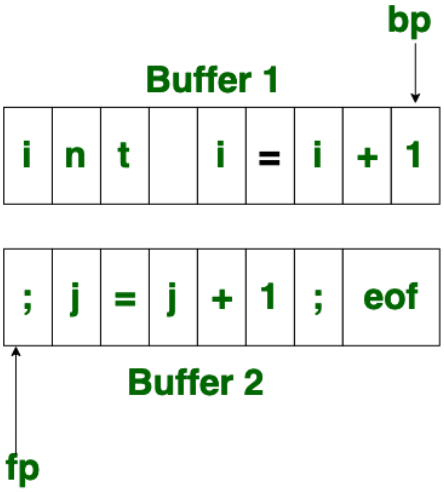


Fig. Two buffer scheme storing input string

Initially both the bp and fp are pointing to the first character of first buffer. Then the fp moves towards right in search of end of lexeme. As soon as blank character is recognized, the string between bp and fp is identified as corresponding token. To identify the **boundary** of first buffer **end of buffer** character should be placed at the end of first buffer. Similarly end of second is also recognized by the end of buffer mark present at the end of second buffer. When fp encounters first **eof,** then one can recognize end of first buffer and hence filling up of second buffer is started. In the same way when second **eof** is obtained then it indicates end of second buffer. Alternatively, both the buffers can be filled up until end of the input program and stream of tokens is identified. This **eof** character introduced at the end is called **sentinel** which is used to identify the end of buffer.

The Lexical Analyzer Generator Lex

- Lex is a program that generates lexical analyzer.
- It is a Unix utility.
- The lexical analyzer is a program that transforms an input stream into a sequence of
- tokens.
- Lex specifies tokens using Regular Expression.

The function of Lex is as follows:

- Firstly lexical analyzer creates a program called lex specification file, lex.l in the Lex language. Then Lex compiler runs the lex.1 program and produces a C program lex.yy.c.
- Finally, C compiler runs the lex.yy.c program and produces an object program a.out.

- a.out is lexical analyzer that transforms an input stream into a sequence of tokens.

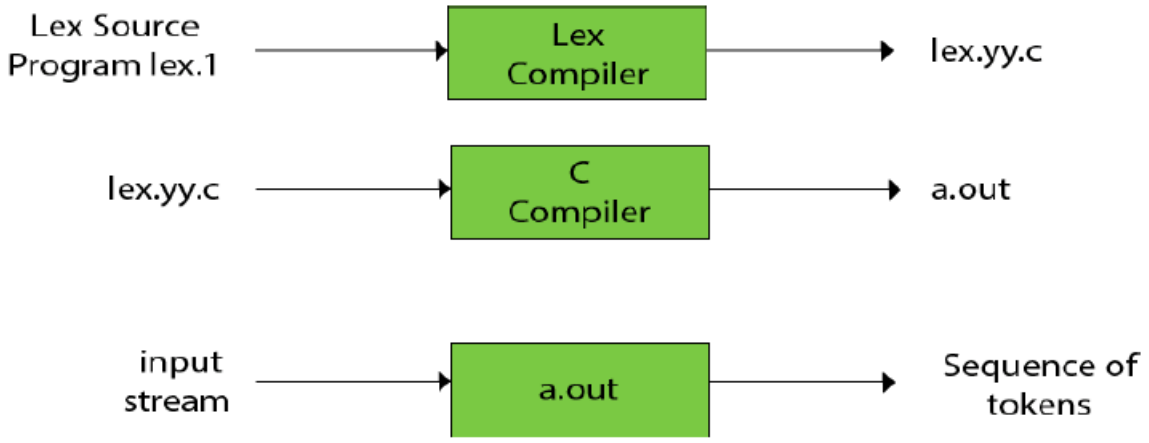


Fig. Generation of Lexical Analyzer using LEX

The structure of LEX programs

The Lex Program Consists of three parts. They are:

1. Declaration Section
2. Rule Section and
3. Auxiliary Functions or Procedure Section

Syntax:

```
%{
Declarations
%}
%%
Rules
%%
Auxiliary Functions
```

1. Declaration Section

- The declarations section consists of two parts, **auxiliary declarations** and **regular definitions**.
- **The auxiliary declarations** are copied as such by LEX to the output *lex.yy.c* file. This C code consists of instructions to the C compiler and are not processed by the LEX tool.
- The auxiliary declarations (which are optional) are written in C language and are enclosed within ' %{ ' and ' %} ' .
- It is generally used to declare functions, include header files, or define global variables and constants.

2. **Rule Section**

- Rules in a LEX program consists of two parts :

  1. The pattern to be matched

  2. The corresponding action to be executed

- Patterns are defined using the regular expressions and actions can be specified using C Code.

The Rules can be given as

R1 {Action1}

R2 {Action2}

.

.

.

Rn {Action n}

Where Ri is RE and Action i is the action to be taken for corresponding RE.

3. **Auxiliary Functions or Procedure Section**

- All the required procedures are defined in this section.

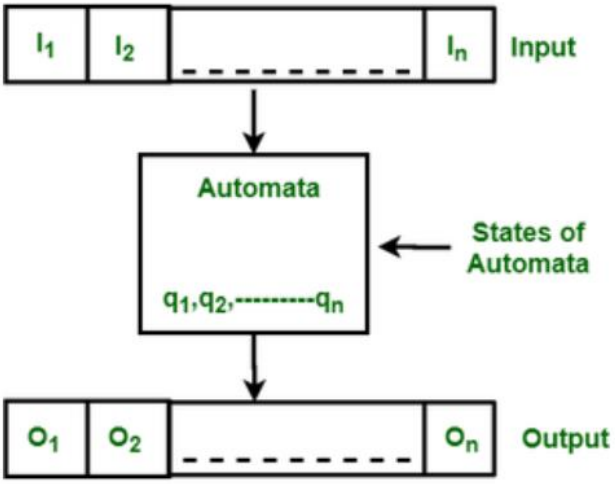
Example: /*lex program to count number of words*/

```
%{
#include<stdio.h>
#include<string.h>
int i = 0;
%}
/* Rules Section*/
%%
([a-zA-Z0-9])*    {i++;} /* Rule for counting number of words*/
"\n" {printf("%d\n", i); i = 0;}
%%
int yywrap(void){}
int main()
{
   // The function that starts the analysis
   yylex();
   return 0;
}
```

Finite automata:

Finite automata are abstract machines used to recognize patterns in input sequences. They consist of states, transitions, and input symbols, processing each symbol step-by-step. If the machine ends in an accepting state after processing the input, it is accepted; otherwise, it is rejected. Finite automata come in deterministic (DFA) and non-deterministic (NFA), both of

which can recognize the same set of regular languages. They are widely used in text processing, compilers, and network protocols.



Formal Definition of Finite Automata

A finite automaton can be defined as a tuple: { Q, Σ, q, F, δ }, where:

- Q: Finite set of states
- Σ: Set of input symbols
- q: Initial state
- F: Set of final states
- δ: Transition function

Types of Finite Automata

There are two types of finite automata:

- Deterministic Finite Automata (DFA)
- Non-Deterministic Finite Automata (NFA)

1. Deterministic Finite Automata (DFA)

A DFA is represented as {Q, Σ, q, F, δ}. In DFA, for each input symbol, the machine transitions to one and only one state. DFA does not allow any null transitions, meaning every state must have a transition defined for every input symbol.

DFA consists of 5 tuples {Q, Σ, q, F, δ}.

Q : set of all states.

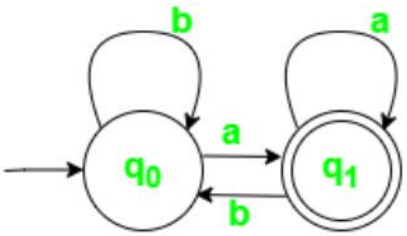
Σ : set of input symbols. (Symbols which machine takes as input)

q : Initial state. (Starting state of a machine)

F : set of final state.

δ : Transition Function, defined as δ : Q X Σ --> Q.

Example:

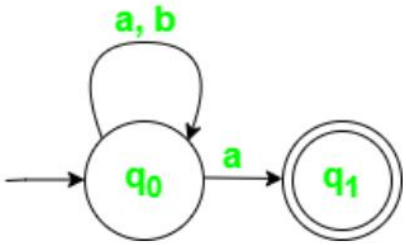


2. Non-Deterministic Finite Automata (NFA)

NFA is similar to DFA but includes the following features:

- It can transition to multiple states for the same input.
- It allows null (ϵ) moves, where the machine can change states without consuming any input.

Example:



Conversion from regular expressions to automata