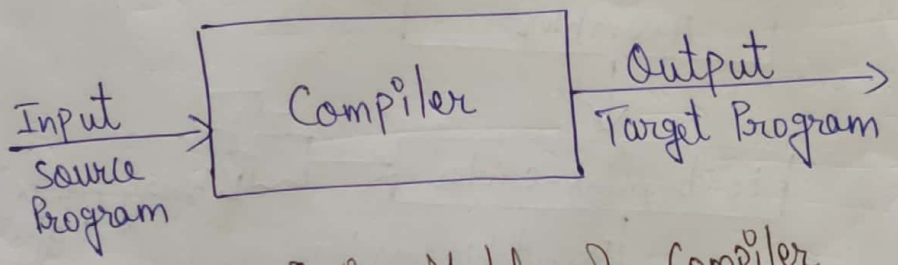
COMPILER DESIGN

UNIT I : Overview of Compiler & Lexical Analysis

*** Introduction :** Compiler is a program which takes one language (source program) as input and translates it into an equivalent another language (target program).

→ Compilers basically act as translators. The basic model of compiler can be represented as follows :



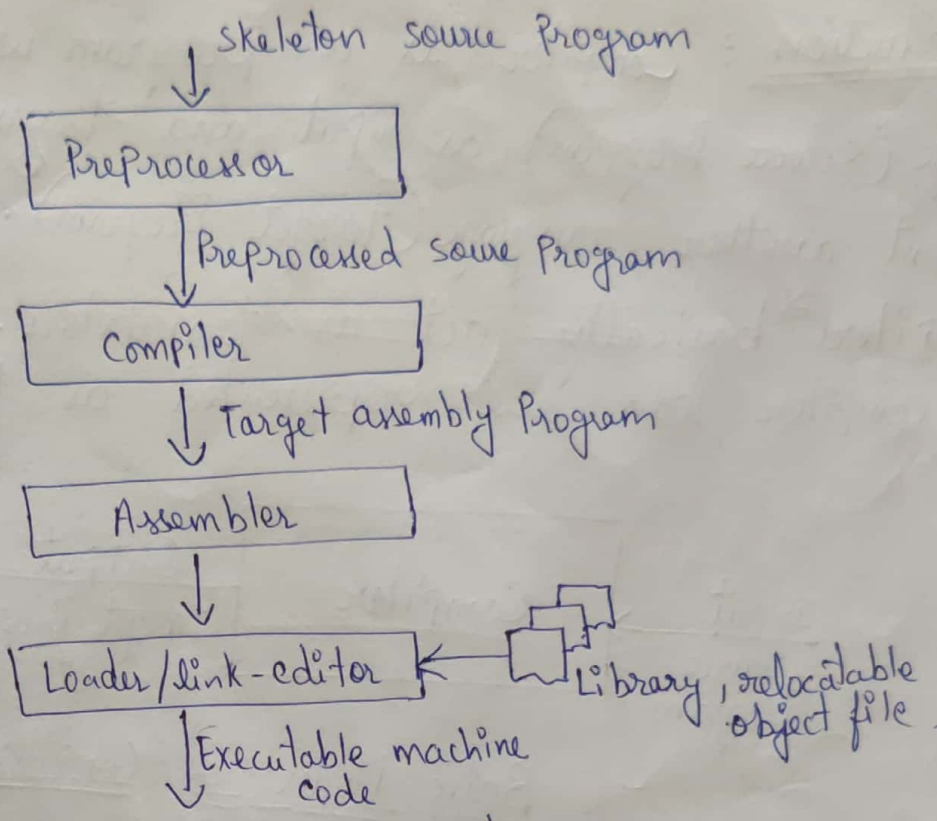
Basic Model of Compiler

→ The compiler takes a source program as higher level languages such as C, Pascal, Fortran & converts it into low level language or a machine level language such as assembly language.

*** Properties of a Compiler :**

- The compiler itself must be bug-free.
- It must generate correct machine-code
- The generated machine code must run fast.
- The compiler must be portable.
- It must give good diagnostics & error messages.
- It must have consistent optimization.

→ To create an executable form of your source Program only a compiler Program is not sufficient. we may require several other Programs to create an executable target Program



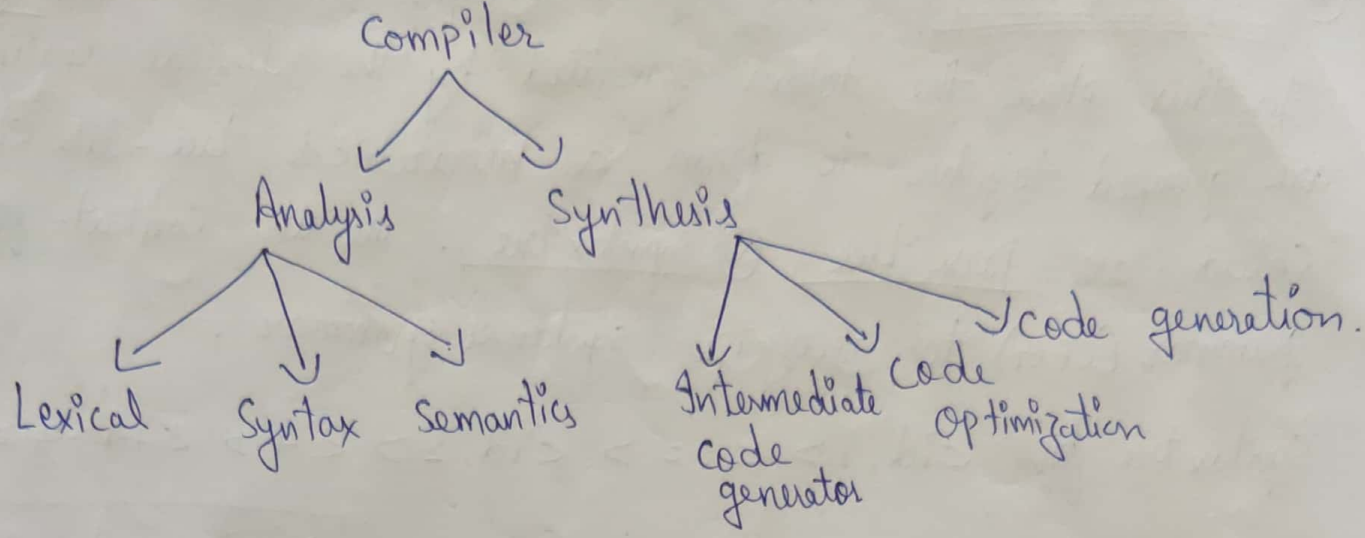
(a) Target Machine code.

Process of Execution of Program.

* Structure of a Compiler :-

The Process of Compilation is carried out in two Parts namely: "Analysis & Synthesis".

→ Analysis Part is the front end of the compiler.
→ Synthesis Part is the back end of the compiler.
→ Compilation Process operates as a sequence of phases, each of which transforms one representation of the source program to another.



Phases of compiler

(a) Lexical Analysis :

The lexical Analysis is also called "Scanning". The complete source code is scanned & the source program is broken up into group of strings called "lexemes". For each lexeme, the lexical analyzer Produces as output a "token" of the form : (token-name, attribute-value).

"token" → sequence of characters having a collective meaning

→ The blank characters which are used in the Programming statement are eliminated during this phase.

Eg : c := a+10 ;

→ these are mapped to the tokens like this.

c, a ⇒ identifiers
:= ⇒ assignment
+ ⇒ operator
10 ⇒ constant

tokens

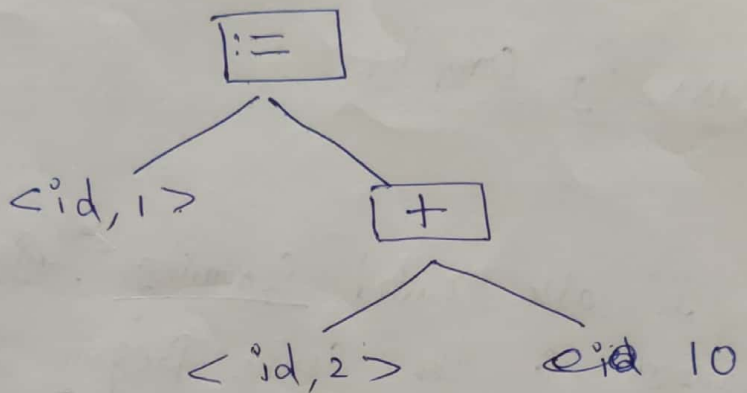
<id, 1> <:=> <id, 2> <+> <10> ;

→ we use regular expressions to recognize tokens.

(b) **Syntax Analysis** :- The syntax analysis is also called "Parsing". In this phase the tokens generated by the lexical analyser are grouped together to form a hierarchical tree-like structure, called as Parse tree (or) syntax tree. We use context-free grammar (CFG) to recognize syntax or grammar.

Syntax tree for <id,1> <:=> <id,2> <+> ~~<id,3>~~ 10 is :-

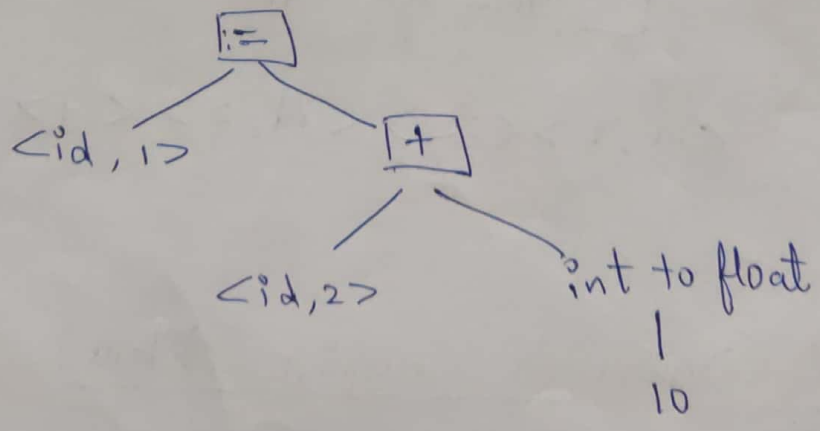
2 Parsing techniques in this phase
- Top-down
- Bottom-up



Parse tree (or) syntax tree.

(c) **Semantic Analysis** :-

Semantic Analysis determines the meaning of the source string. It also does "type checking" where the compiler checks that each operator has matching operands. For example, matching of Parenthesis in the expression, matching of else stmts, checking the scope of operation etc.



(d) Intermediate code generation :

The intermediate code is a kind of code which is easy to generate & this code can be easily converted to target code. Intermediate code is of many forms such as three address code, quadruple, triple, Posix etc. For the above example let us convert it into three address form (consists of instructions each of which has at most three operands).

Eg : $t_1 = \text{int to float}(10)$

$t_2 = id2 + t_1$

$id1 = t_2$

(e) Code optimization :

The code optimization phase attempts to improve the intermediate code. This is necessary to have a faster executing code or less consumption of memory. By optimizing the code the overall running time of the target program can be improved.

Eg : $t_1 = id2 + 10.0$ // code is optimized here. unnecessary variables are removed.

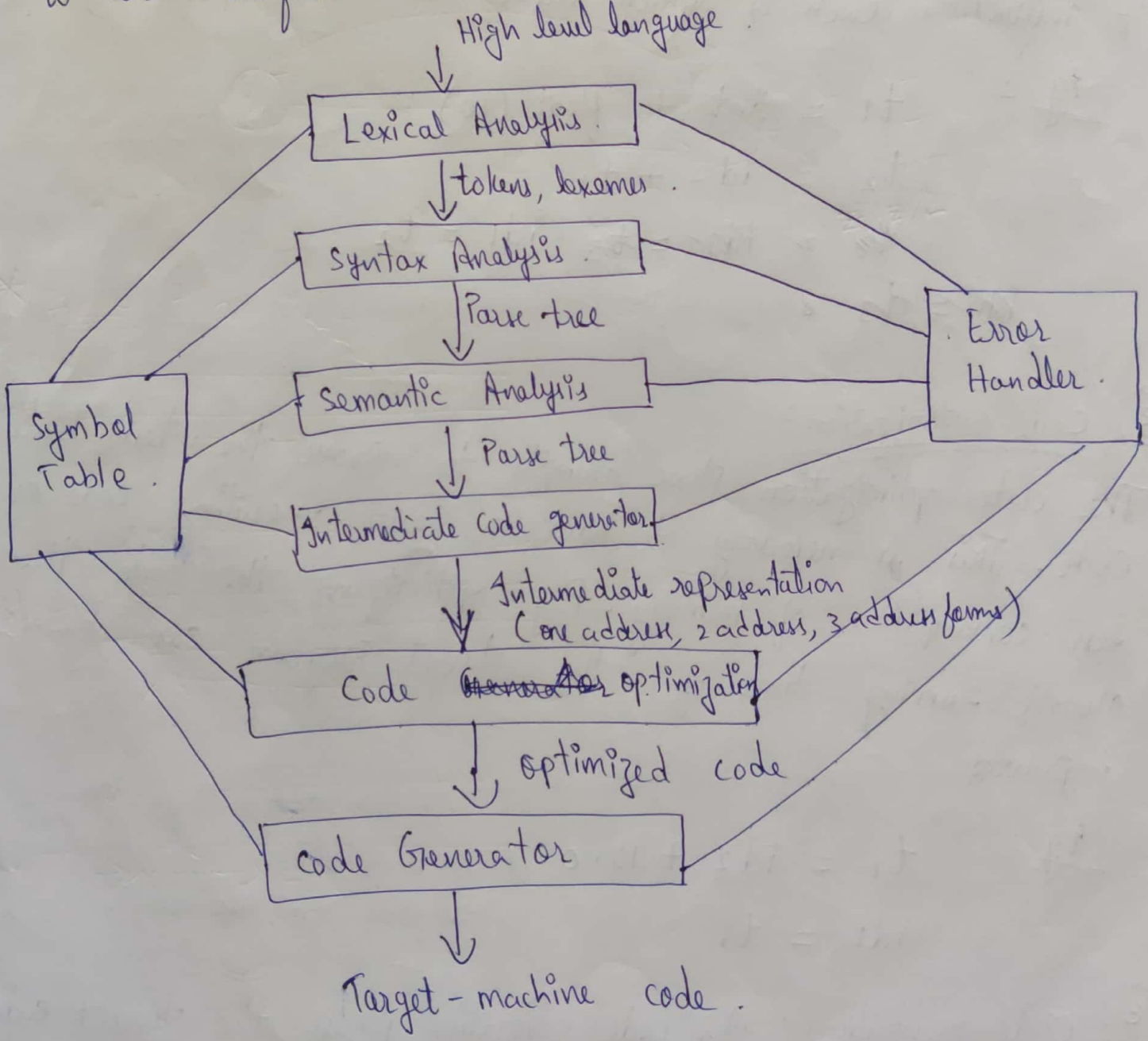
$id1 = t_1$

(c) Code Generation : In code generation phase the target code gets generated. The intermediate code instructions are translated into sequence of machine instructions.

Eg: Mov id2, R1
ADDF R1, #10.0, R1

Mov id2, R1
ADDF #10.0, R1
Mov R1, id3

we are moving value of id2 to 'R1' register. Then we add id2 & 10 & result is stored in 'R1'. Then value of R1 is assigned to id3 identifier.



Phases of Compiler

* Symbol Table

Symbol table is a data structure containing a record for each variable name, with fields for the attributes of the name. Symbol table also stores information about the subroutines used in the program.

→ It allows us to find the record for each identifier quickly & to store or retrieve data from that record efficiently.

* Error detection & Handling :

In compilation, each phase detects errors. These errors must be reported to error handler, whose task is to handle the errors so that the compilation can proceed. Errors are reported in the form of 'messages'.

→ Large no. of errors can be detected in syntax analysis phase. Such errors are called as "syntax errors".

* Pass : when several phases are grouped together, we call it as Pass.

* The Science of Building a Computer :

A compiler must accept all source programs that conform to the specification of the language. 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.

a) Modeling in Compiler Design & Implementation

The study of compilers is mainly a study of how we design the right mathematical models & choose the right algorithms, while balancing the need for generality & Power against simplicity & efficiency.

→ Some of most fundamental models are finite-state machines & regular expressions (for describing the lexical units of Programs), contex-free grammars (to describe syntactic structure).

b) The Science of Code optimization:

The term 'optimization' in compiler design refers to the attempts that a compiler makes to produce code that is more efficient than the obvious code.

→ Compiler optimizations must have the following design objectives:

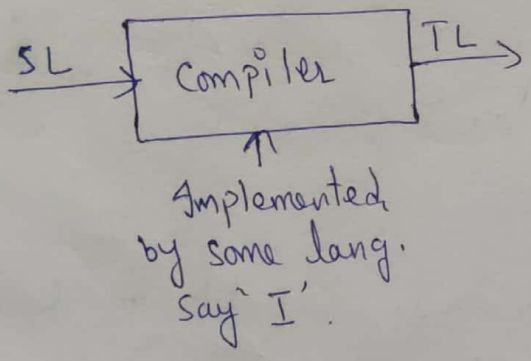
- The optimization must be correct, i.e Preserve the meaning of the compiled Program.
- The optimization must improve the Performance of many Programs.
- The compilation time must be kept reasonable
- The Engineering effort required must be manageable.

* Bootstrapping :

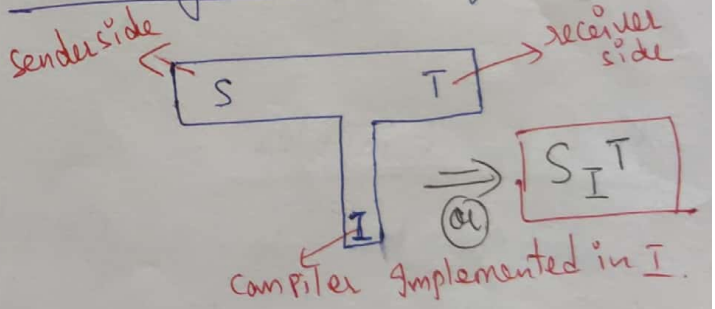
A Process by which simple language is used to translate more complicated Program, which in turn may handle an even more complicated Program.

→ There are three types of languages.

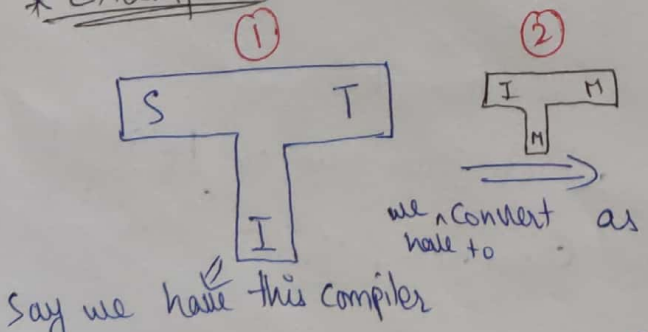
- Source language(SL) ; i.e the application program
- Target language(TL) in which machine code is written.
- Implementation language in which a compiler is written.

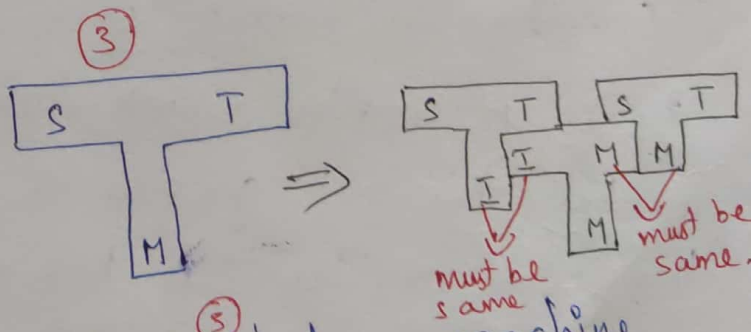


⇒ This can be represented by "T-diagram" as follows;



* Example :



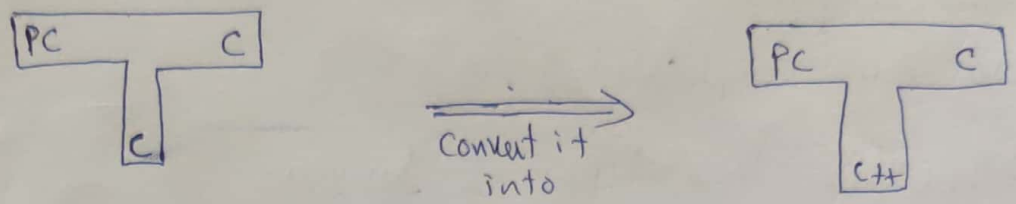


→ The Process of converting (1) 'I' into (5) 'M' lang. machine is Boot-strapping.

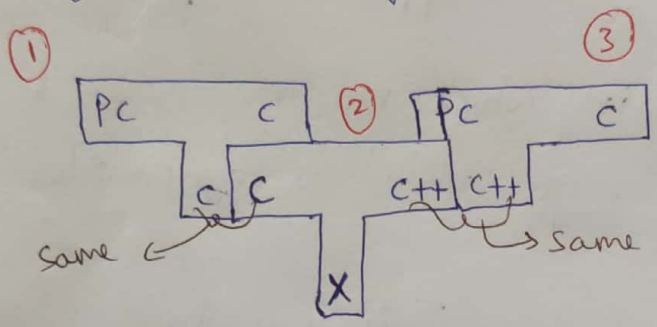
Example :

We have Pascal Translator written in → 'c' lang.
i/p is Pascal code(Pc), o/p – 'c'. create pascal translator in c++.

Sol :- Given :-



with help of Bootstrapping :-



X ⇒ can be anything like java or any other lang

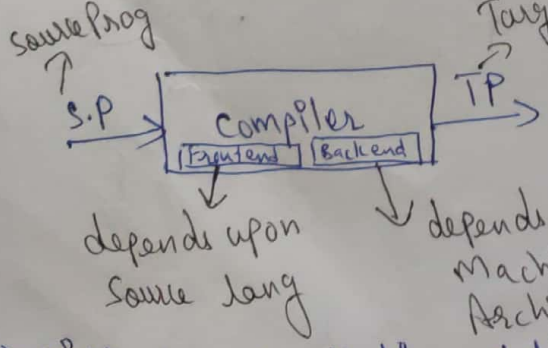
(or) $P_c{}^c + C_x{}^{c++} \Rightarrow P_{c++}{}^c$

* Cross Compiler :-

Cross compiler is a compiler which is capable of creating executable code for a platform other than the one on which compiler is running.

Eg :- A compiler which runs on windows 7 but this generates a code which can run on Android smartphone.

Then this compiler is known as Cross-compiler.



⇒ we need to change only the backend to implement Cross compiler.

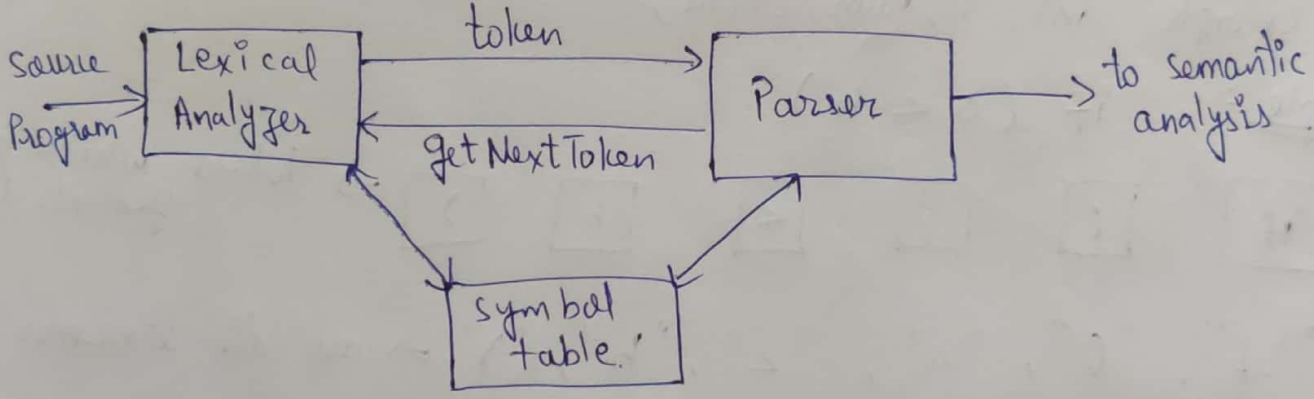
→ using cross compilation, platform independency can be achieved.

LEXICAL ANALYSIS

* Role of Lexical Analyzer :-

Lexical Analyzer reads the input source program from left to right one character at a time & generates the sequence of tokens. It is the first phase of a compiler.

→ Each token is a single logical cohesive unit such as identifier, keywords, operators & punctuation marks. Then the Parser to determine the syntax of the source program can use these tokens.



Interactions between the lexical analyzer & the Parser

→ As the lexical analyzer scans the source program to recognize the tokens, it is also called as "Scanner".

* Functions of lexical Analyzer :-

- It Produces stream of tokens
- It eliminates whitespaces (blank, newline, tab) & comments.
- It generates symbol table which stores the information about identifiers, constants encountered in the i/p.
- It keeps track of line no's.
- It reports the error encountered while generating the tokens.

* Tokens : A token is a pair consisting of a token name & an optional attribute value. It describes the class or category of input string. for example identifiers, keywords, constants are called tokens.

* Pattern : A pattern is a description of the form that the lexemes of a token may take. It is a set of rules that describe the token.

* Lexemes : Sequence of characters in the source program that are matched with the pattern of the token. For example int, i, num, choice;

Example : if (a < b)

if	(	a	<	b	)

⇒ tokens which are generated.

here "if", "(", "a", "<", "b", ")" ⇒ are all lexemes.

→ "if" is a keyword
→ "(" is openening Parenthesis
→ "a" is identifier → collection of letter.
→ "<" is an operater
→ "b" is identifier
→ ")" is closing Parenthesis.

* Attributes for tokens :

Eg :- Token : Lexemes

number — 0, 1, 2, ... 0, 6.2, 1, ...

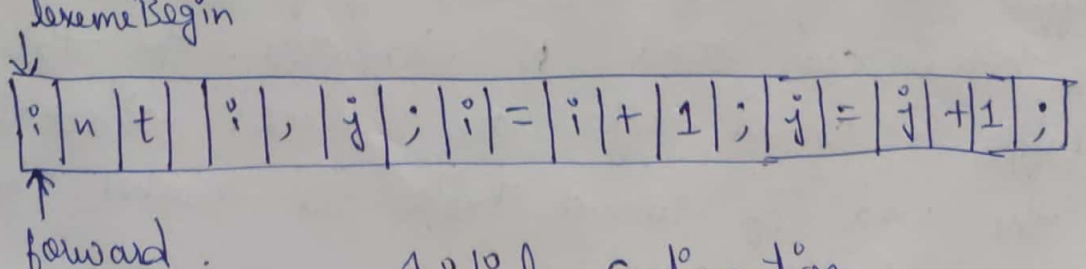
When more than one lexeme can match a pattern, the lexical analyzer must provide the subsequent compiler phases additional information about the particular lexeme that matched.

Eg :- Pattern for "token : Number" matches both '0' and '1'.

→ Lexical analyzer returns to the parser not only a token name but an attribute value that describes the lexeme represented by the token. Token name influences parsing decisions, while the attribute value influences translation of tokens after the Parse.

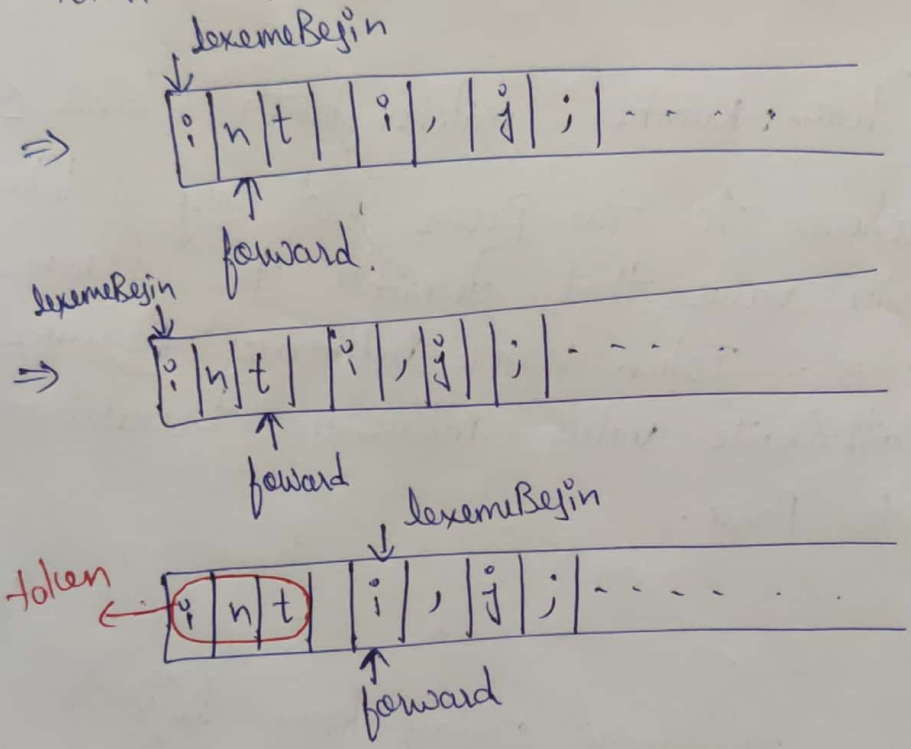
* Input Buffering :

The lexical analyzer scans the input string from left to right one character at a time. It uses two pointers: begin Pointer "lexemeBegin" and foward Pointer "foward" to keep track of the portion of the input scanned. Initially both the pointers point to the first character of the input string as:



Initial Configuration

The 'lexemeBegin' pointer remains at the beginning of the sting to be read & the 'forward' pointer moves ahead to search for end of lexeme. As soon as the blank space is encountered, it indicates end of lexeme. & then both 'lexemeBegin' & 'forward' is set at next token 'i'.



→ Reading i/p's from secondary storage is costly. so, a block of data is first read into a buffer & then scanned by lexical analyzer.

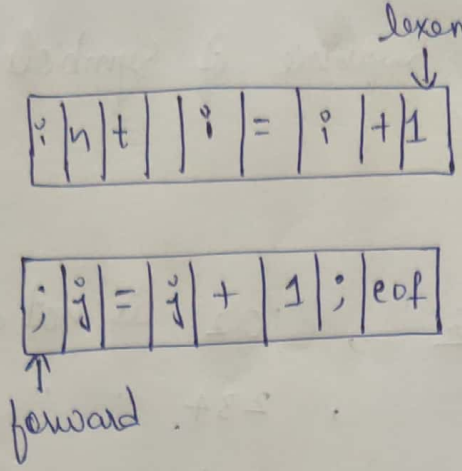
Input Buffering

→ Before one buffer scheme was used but now we are using two buffer scheme.

* Two Buffer Scheme

to store i/p string.

Two buffers each of size 'N' (usually size of disk block) are used A Each buffer can read 'N' characters. In this method, first buffer & second buffer are scanned alternately. When the end of current buffer is reached the other buffer is filled.



Two Buffer scheme

* Sentinel :
Both buffers have a special character "eof" at the end. when 'lexemeBegin' pointer encounters 'eof' at Buffer 1, then it starts filling up second buffer. Similarly when 'eof' is encountered at Buffer2, it starts filling Buffer1 & so on.

→ This 'eof' character introduced at the end is called "Sentinel" which is used to identify the end of buffer.

* Specifications of Tokens :

To specify tokens 'regular expressions' are used. when a pattern is matched by some regular expression then token can be recognized.

* Alphabet : Alphabet is a finite, non-empty set of symbols. It is represented by the symbol 'Σ'.

Eg : $\Sigma = \{0,1\}$, $\Sigma = \{a,b\}$
$\Sigma = \{a,b,c,\cdots\}$, $\Sigma = \{e_0, e_1, \cdots e_9\}$

* String : String is a finite sequence of symbols choosen from some alphabet.

Eg : $\Sigma = \{0, 1\}$.

0011 is a string of Σ, 102 is not a string of Σ.

1010 " " " " ", 234 " " " " " ".

→ length of string is denoted by $|s|$.

→ Empty string can be denoted by 'ϵ'.

* Language : Set of strings which belong to some alphabet 'Σ' is called as a Language (L).

Eg : $\Sigma = \{0, 1\}$.

$L = \{00, 01, 10, 11\}$

$L = \{0^n 1^n \mid n \geq 1\}$

* Operations on Language :

There are various operations which can be performed on a language as follows : below. Let 'L', M' be two languages.

Operation	definition & Notation.
• Union of L & M	$L \cup M = \{ s \mid s$ is in L or s is in M$\}$
• Concatenation of L & M	$LM = \{st \mid s$ is in L & t is in M$\}$
• Kleen closure of 'L'	$L^* = \bigcup_{i=0}^{\infty} L^i$
• Positive closure of 'L'	$L^+ = \bigcup_{i=1}^{\infty} L^i$

Definitions of operations on languages

Eg :- $L = \{A, B, C, \ldots Z, a, b, c, \ldots\}$, $D = \{0, 1, 2, \ldots 9\}$ are two languages.

- LUD is a set of letters & digits
- LD is a set of strings consisting of letters followed by digits.
- L^5 is a set of strings having length of 5 each.
- L^* is " " " " all strings including 'ϵ'.
- L^+ " " " " except 'ϵ'.

* Regular Expressions :- A Regular Expression is a notation to represent certain sets of strings in an algebraic fashion. This notation involves a combination of strings of symbols from some alphabet 'Σ', the symbol of null string 'ϵ', star operator (*) & plus operator (+).

* Examples :-

(1) write a Regular Expression (RE) for a language containing the strings of length two over $\Sigma = \{0, 1\}$.

Sol :- $RE = (0+1)(0+1)$.

(2) write a RE for lang. containing strings which end with 'abb' over $\Sigma = \{a, b\}$

Sol :- $(a+b)^*abb$.

(3) write a RE for a recognizing identifier.

Sol :- For denoting identifier, we will consider a set of letters

& digits because identifier is a combination of letters or letter & digits but having first character as letter always. Hence RE can be denoted as:

$$\boxed{RE = letter\ (letter + digit)^*}$$

where letter = (A, B, ... Z, a, b, ... z) & digit = (0, 1, 2, ... 9).

* Regular Definitions:

If 'Σ' is an alphabet of basic symbols, then a regular definition is a sequence of definitions of the form:

$$d_1 \longrightarrow r_1$$
$$d_2 \longrightarrow r_2$$
$$\vdots \qquad \vdots$$
$$d_n \longrightarrow r_n$$

where,

- Each 'di' is a new symbol, not in 'Σ' & not the same as any other of the d's &
- Each 'ri' is a regular expression over the alphabet $\Sigma \cup \{d_1, d_2, \ldots, d_{i-1}\}$.

→ Some of the notations used for writing the regular expressions are as follows:

(1) One or more instance : To represent one or more instances "+" sign is used. Eg : r^+.

(2) Zero or more instances : To represent zero or more instances "*" sign is used. Eg : r^*.

(3) Character classes : A class of symbols can be denoted by [ ]. Eg : [012] means 0 or 1 or 2.

* Examples :

(1) write Regular definition for 'C' language identifiers.

Sol : 'C' identifiers are strings of letters, digits & underscores.

Regular definition is :

letter_ → [A-z a-z]
digit → [0-9]
id → letter_ (letter_ | digit)*

(2) write Regular definition for Unsigned numbers.

Sol : unsigned nos (integer or floating point are strings like :
5080, 0.01234, 6.33458, 0.89E-4 etc.

Regular definition is :

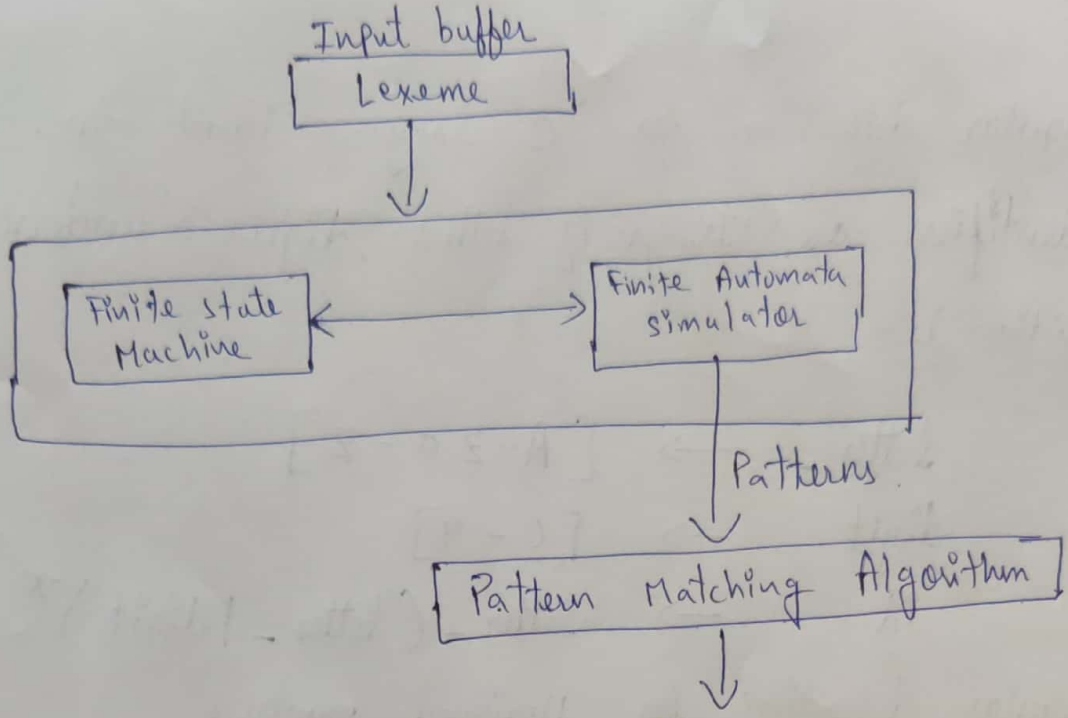
digit → [0-9]
digits → digit+
number → digits (. digits)? (E [+-]? digits)?

(? → zero or one instance)

* Recognition of Tokens :

The token is usually represented as: | Token type | Token value/attribute |

→ The token type tells us the category of token & token value gives us the information regarding token. During lexical analysis, Process, the symbol table is maintained. The token value can be a pointer to symbol table in case of identifiers & constants.



Block diagram of Lexical Analyzer

* Example :

digit → [0-9]
digits → digit +
number → digits (. digits)? (E [+-]? digits)?
letter → [A-Z a-z]
id → letter (letter | digit)*
if → if
then → then
else → else
relational operators ↳ relop → < | > | <= | >= | = | <>

Sample Regular definition

Lexemes	Token Name	Attribute Value
Any ws	—	—
if	if	—
then	then	—
else	else	—
Any id	id	Pointer to table entry.
Any number	number	Pointer to table entry.
<	relop	LT
<=	relop	LE
=	relop	EQ
<>	relop	NE
>	relop	GT
>=	relop	GE

Tokens, their patterns & attribute values

* Transition Diagrams:

In the middle of lexical analyzer Process we convert patterns into Transition diagrams. Transition diagrams have a collection of nodes or circles called states. Each state represents a condition that could occur during the Process of scanning the input looking for a lexeme that matches one of several patterns.

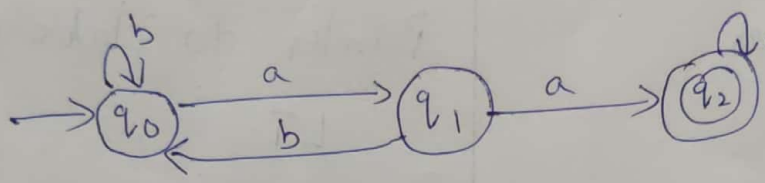
→ Edges are directed from one state to another. Each edge is labeled by a symbol or set of symbols.

- $\rightarrow (q_0)$ ⇒ initial state.
- (q) ⇒ final state.

* Examples :-

(1) Design a transition diagram for the language consisting of all the strings containing at least one pair of consecutive a's over $\Sigma = \{a, b\}$.

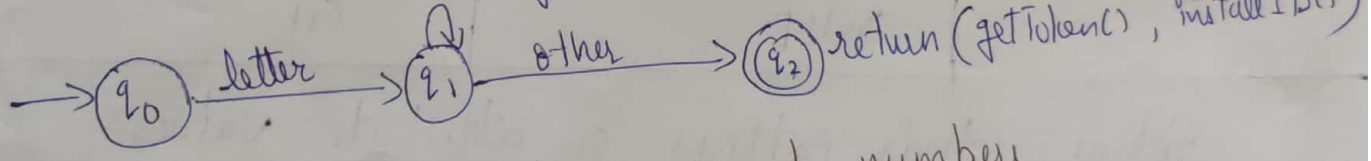
Sol :-



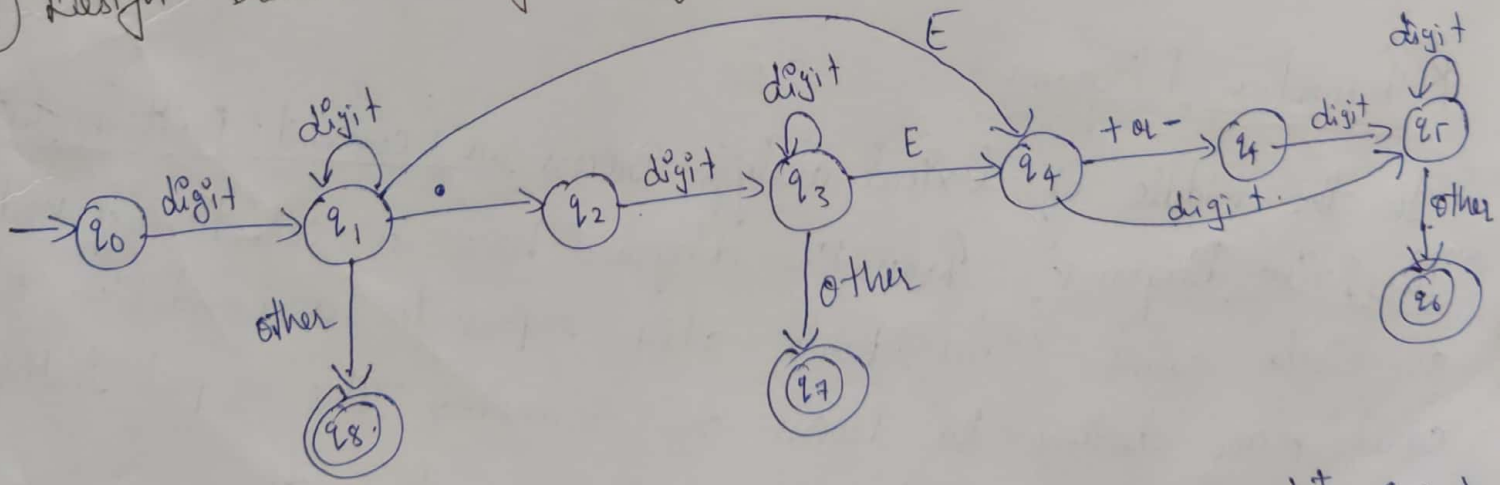
	a	b
$\rightarrow q_0$	q_1	q_0
q_1	q_2	q_0
q_2	q_2	q_2

(2) Design transition diagram for identifiers.

Sol :- RE for identifiers :- r.e = letter (letter | digit)*

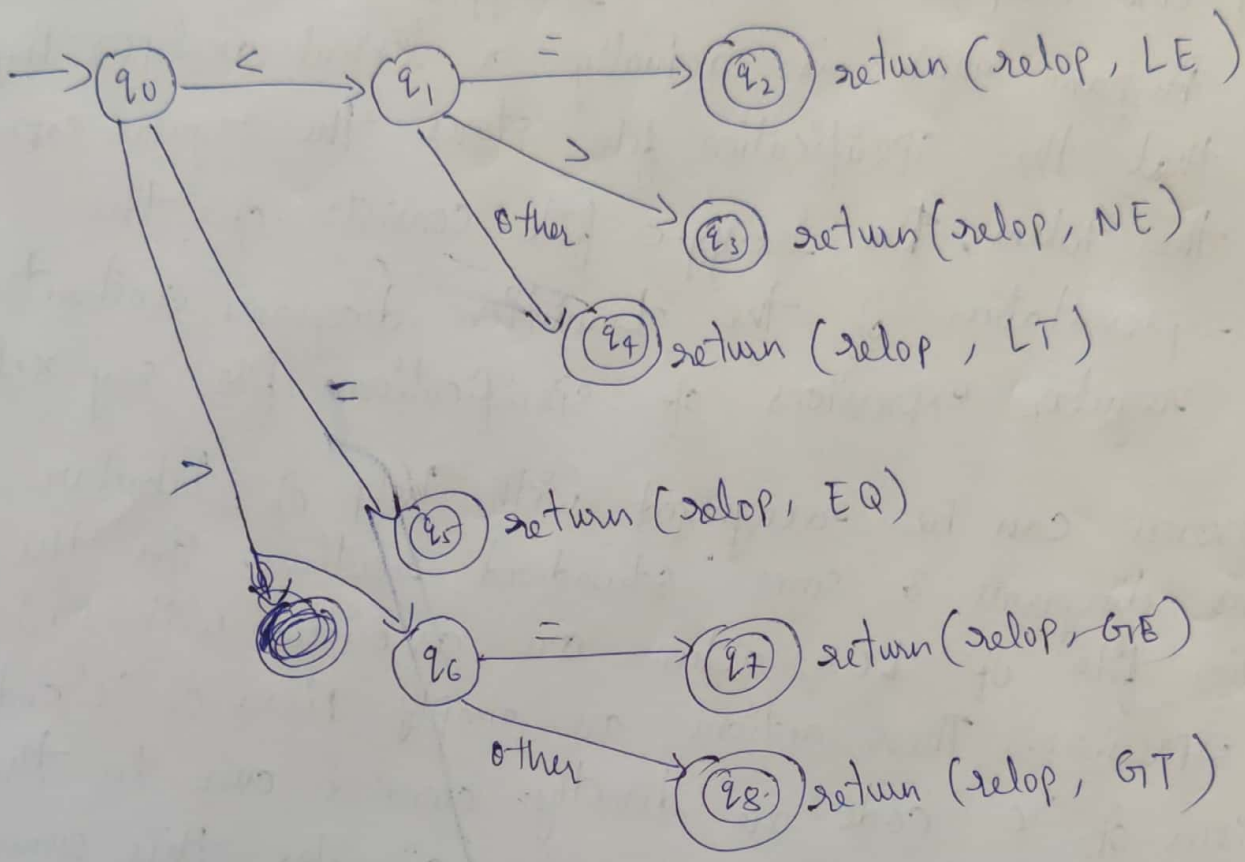


(3) Design transition diagram for unsigned numbers.



re = digit+ | (digit)+ (.) (digit)+ | (digit)+ (.) (digit)+ E (+|-) (digit+).

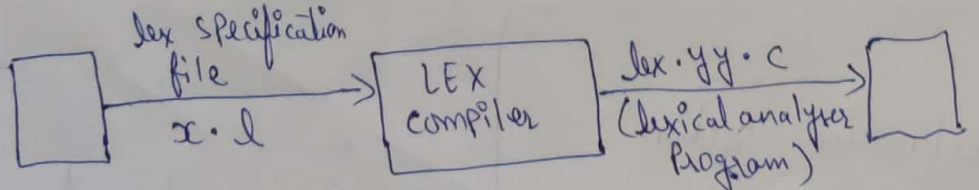
(4) Design transition diagram for relational operators (relop).

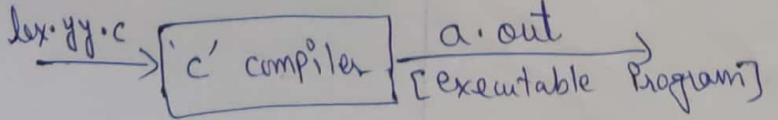


* LEX — Lexical Analyzer Generator:

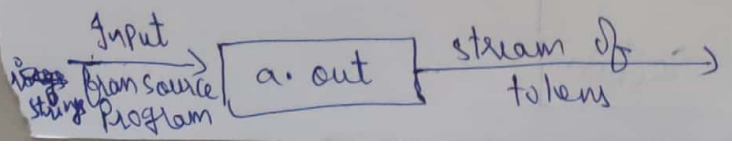
(a) FLEX

Regular expressions are used in recognizing the tokens. A tool called LEX or FLEX (recent implementation) allows one to specify a lexical analyzer by specifying regular expressions to describe patterns for tokens.





Generation of lexical analyser using LEX



→ Specification file can be created using a file with extension ".l" (dot 'l'). Say for example "x.l". This x.l is then given to LEX compiler to Produce lex.yy.c. This lex.yy.c is a 'c' Program which is actually a lexical analyzer Program we know that the specification file stores the regular expressions for the tokens. The lex.yy.c file consists of the tabular representation of the transition diagrams constructed for the regular expressions of specification file say x.l.

→ The lexemes can be recognized with help of tabular transitions diagrams & some standard routines. In the specification file of LEX, actions are associated with each regular expression. These actions are simply pieces of 'c' code. These pieces of 'c' code are directly carried over to the lex.yy.c. Finally the 'c' compiler compiles this generated lex.yy.c & Produces an object Program a.out. When some input stream is given to a.out then sequence of tokens get generated.

→ LEX Program consists of three parts:
a) Declaration section, b) Rule section & c) Procedure section.

```
%{
   Declaration section
%}
%%
Rule section
%%
Auxiliary Procedure section
```

LEX Program format

(a) Declaration section :- In this section, declaration of variables, constants can be done. Some regular definitions can also be written in this section. The regular definitions are basically components of regular expressions appearing in the rule section.

(b) Rule section :- It consists of regular expressions with associated actions. These translation rules can be given as :-

R_1	{action 1}
R_2	{action 2}
⋮	
R_n	{action n}

where each R_i is a regular expression & each action; is a program fragment describing what action is to be taken for corresponding regular expression. These actions can be specified by piece of 'C' code.

(c) Auxiliary Procedure section :- In this section, all the required Procedures are defined. Sometimes these Procedures are required by the actions in the rule section.

→ The lexical analyzer or scanner works in co-ordination with Parser. When activated by the Parser, the lexical analyzer begins reading its remaining input, character by character at a time. When the string is matched with one of the regular expressions R_i then corresponding action; will

get executed & this action; returns the control to the Parser.

* Example LEX Programming :

```
                                    // Program name : x.l
%{
%}                                  // Declaration section
%%                                  // Rule section
"Rama"  |
"seeta" |
"Geeta" |
"Neeta" |   Printf ("In Noun");
"sings" |
"dances"|
"eats"      Printf ("In Verb");
%%
main()                              // Auxiliary Procedure section.
{
   yylex();
}
int yywrap()
{
   return 1;
}
```

→ consists of RE's & actions.

→ This is a simple program that recognizes noun & verb from the string.

* Execution :

```
$ lex x.l ↵
$ cc lex.yy.c ↵
$ ./a.out ↵
```

o/p : ip1 : Rama eats ↵
o/p : Noun
Verb.

ip2 : Seeta sings ↵
o/p : Noun
Verb

* LEX Actions :

(1) BEGIN : It indicates the start state. The lexical analyzer starts at state '0'.

(2) ECHO : It emits the input as it is.

(3) yylex() : As soon as call to yylex() is encountered, scanner starts scanning the source Program.

(4) yywrap() : The function yywrap() is called when scanner encounters end of file. If yywrap() returns '0'(zero) then scanner continues scanning. when " " '1'(one) that means end of file is encountered.

(5) yyin : It is the standard file that stores input source Program.

*FLEX :-

Flex is a tool for generating scanners. A scanner, sometimes called a 'tokenizer' is a program which recognizes lexical patterns in text. The flex program reads user-specified input files or its standard input if no file names are given for a description of a scanner to generate. The description is in the form of pairs of regular expressions & 'C' code called rules. Flex generates a 'C' source file named "lex.yy.c" which defines the function yylex(). The file lex.yy.c can be compiled & linked to produce an executable.

→ when the executable is run, it analyzes its i/p for occurances of text matching with the regular expressions for each rule. whenever it finds a match, it executes the corresponding 'C' code.