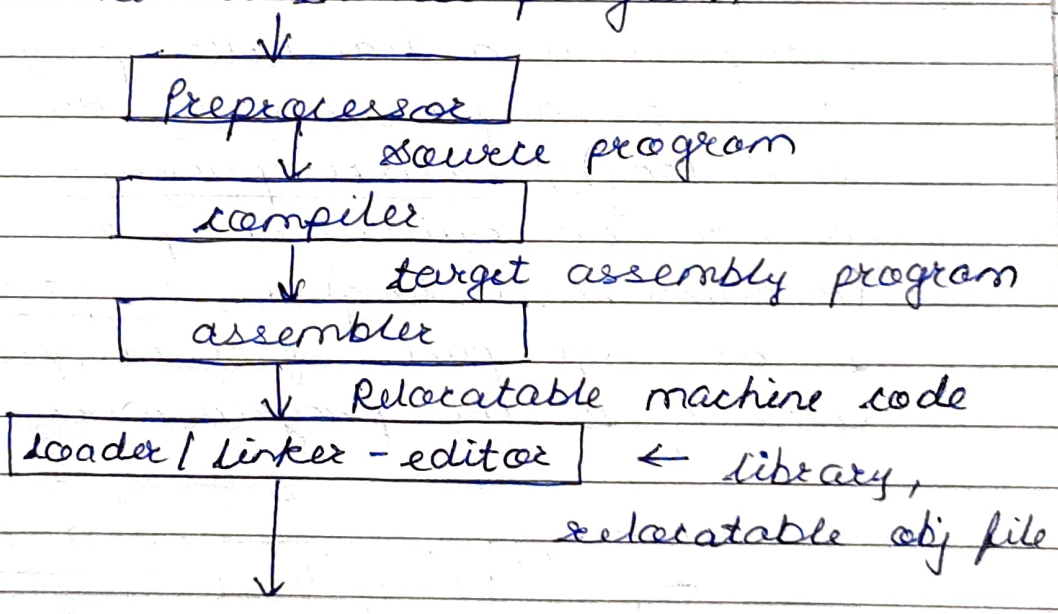
CC - Unit I

Language processors (1-7), Phases of compiler
Pass & phase , bootstraping (22)
CC tools , application of CC
programming lang. basics
lexical analysis , role & responsibility,
input buffering (25), specification of tokens,
recognition of tokens , LEX tool,
design of lexical analyzer generator. (24)

1] Overview of lang. processing system.



2] Preprocessor

A preprocessor produce input to compilers. They may perform the following functions.

① Macro processing - A preprocessor may allow a user to define macros that are short hands for longer constructs.

② file inclusion - A preprocessor may include header files into the program text.

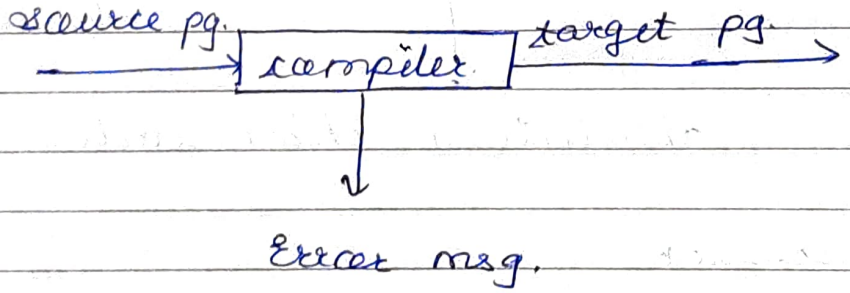
③ rational preprocessor - These preprocessors augment older languages with more modern flow-of-control and data structuring facilities.

④ language extensions - These preprocessor attempts to add capabilities to the language by certain amounts of built-in - macro.

3] Compiler

Compiler is a translator program that translates a program written in HLL (high level lang) i.e. source program and translates it into an equivalent program in MLL (medium level lang) i.e., target program.

As an imp part of a compiler is error showing to the programmer.



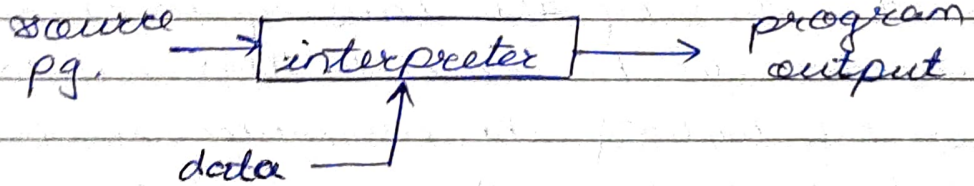
Executing a program written in HLL is basically of two parts. The source program must first be compiled translated into a object program. Then the results object program is loaded into a memory executed.

4] Assembler

Programmers found it difficult to write/read programs in machine lang. They begin to use a mnemonic (symbols) for each machine instruction, which they would subsequently translate into machine lang. Such a mnemonic machine lang is called an assembly lang. Programs known as assembler were written to automate the translation of assembly lang to machine lang. The input to an assembler program is called source program, the output is a machine lang translation (obj program).

5] interpreter

An interpreter is a program that appears to execute a source program as if it were machine language.



Languages such as BASIC, SNOBOL, LISP can be translated using interpreters. JAVA also uses interpreter. The process of interpretation can be carried out in following phases.

1. lexical analysis
2. syntax analysis
3. semantic analysis
4. direct execution

advantages :-

- Modification of user program can be easily made & implemented as execution proceeds.
- Type of object that denotes various may change dynamically.
- Debugging a program & finding errors is simplified task for a program used for interpretation.
- The interpreter for the lang. makes it machine independent.

disadvantages :-

- The execution of program is slower.
- Memory consumption is more.

6] Loader and link - editor :-

Once the assembler procedures an object program that program must be placed into memory and executed. The assembler could place the obj program directly in memory and transfer control to it, thereby causing machine language program to be execute. This would waste core by leaving the assembler in memory while the user's program was being executed. Also programmer would have to retranslate his program with each execution, thus wasting translation time. To overcome this problem of wasted translation time & memory system programmers developed another component called loader.

A loader is a program that places programs

into memory and prepares them for execution. It would be more efficient if subroutines could be translated into object form the loader could relocate directly behind the user's program. The task of adjusting programs they may be placed in arbitary core locations is called relocation. Relocation loaders perform four functions.

7] Translator :-

A translator is a program that takes input a program written in one lang and produces as output a program in another lang. It also performs error-detection. Any violation of HLL specification would be detected and reported to the programmers. Imp scale are-

① translating the HLL program input into an equivalent mLL program.

② Providing diagnostic messages wherever the programmer violates specification of HLL.

types of translators -
interpreter
compiler
preprocessor

list of compilers -

1. Ada compilers
2. ALGOL compilers
3. BASIC compilers
4. C# compilers
5. C compilers
6. C++ compilers
7. COBOL compilers
8. Java compilers

8] Phases of compiler

IMP

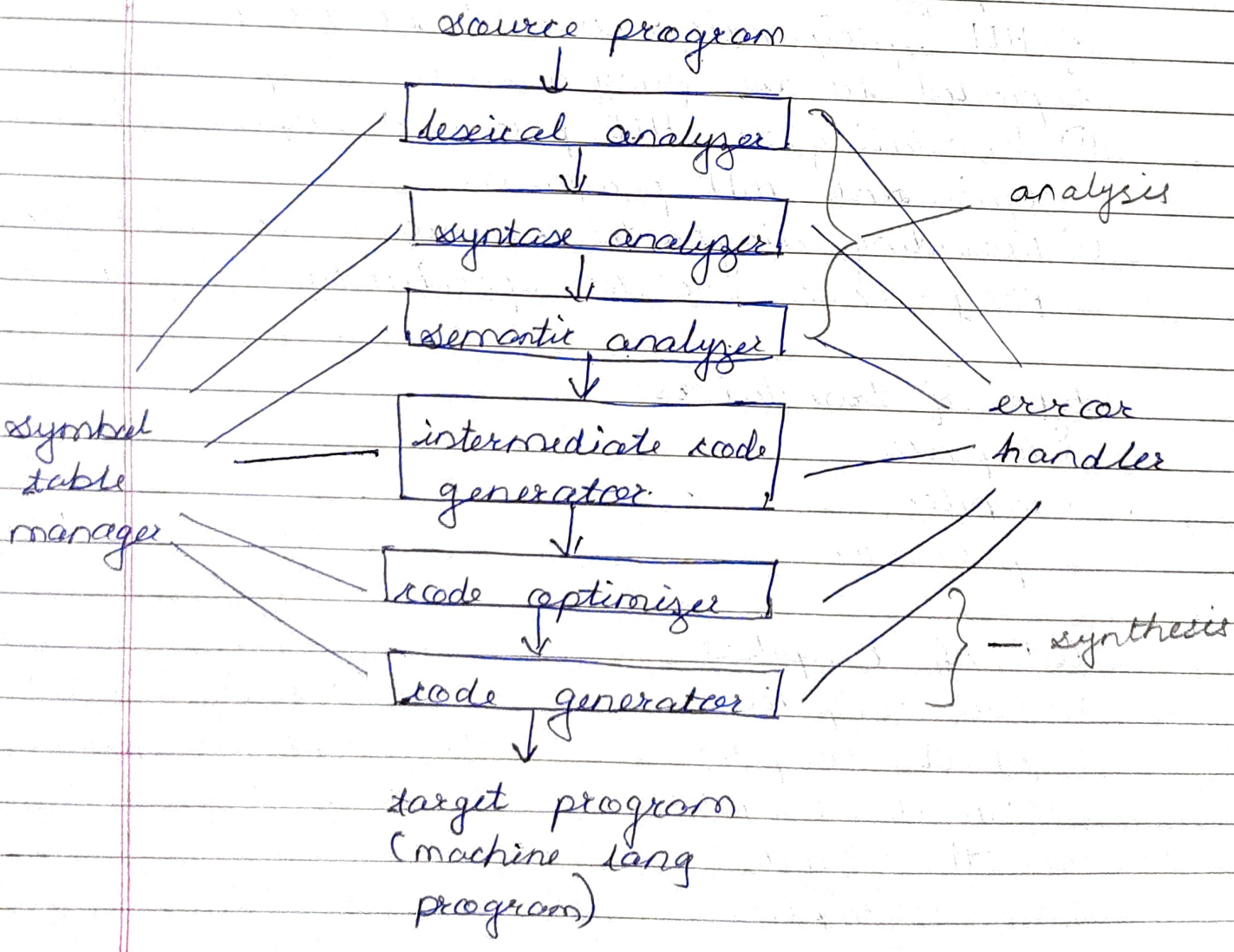
A compiler operates in phases. A phase is a logically interrelated operation that takes source program in one representation and produces output in another representation.

There are two phases of compilation.

a. analysis (machine independent / lang dependent)

b. synthesis (machine dependent / lang independent)

Compilation process is partitioned into ~~no.~~ no-of- sub processes called phases.



lexical analysis :-
LA or scanners read the source program one char at a time, carving the source program into a sequence of automatic units called tokens.

syntax analysis :- (parsing)
The second stage of translation is called syntax analysis/ parsing. In this phase exp, statements, declarations etc. are identified by using the results of lexical analysis. Syntax analysis is aided by using techniques based on formal grammar of programming lang.

intermediate code generations-
An intermediate representation of the final machine lang. code is produced. This phase bridges the analysis and synthesis phases of translation.

code optimisation-
This is optional phase described to improve the intermediate code so that the output runs faster and takes less space.

code generation :-
The last phase of translation is code generation. A number of optimisations to reduce the length of machine language program are carried out during this phase. The output of the code generator is the machine language

program of the specified program.

Table management or book-keeping :-
This is the portion to keep the names used by the program and records essential info about each. The data structure used to record this info is called symbol table.

Error handler :-
It is invoked when a flaw error in the source program is detected. The output of LA is a stream of tokens, which is passed to the next phase, the syntax analyzer or parser. The SA groups the tokens together into syntactic structure called as expression. Expression may further be combined to form statements. The syntactic structure can be regarded as a tree whose leaves are tokens called as parse trees.

The parser has two functions -
It checks if the tokens from lexical analyzer, occur in pattern that are permitted by the specification for the source language.
It also imposes on tokens a tree-like structure that is used by sub-sequent phases of compiler.

Eg - If a program contains the exp A+/B after lexical analysis this exp might appear to the syntax analyzer as the token sequence id+/id. On seeing the /, the syntax analyzer should detect an error situation,

t 3 -
tase
r

because the presence of these two adjacent binary operators violates the formulations rule of an exp.

Syntax analysis is to make explicit the hierarchial structure of the incoming token stream by identifying which parts of the token stream should be grouped.

Ex - (A/B*C has two possible interpretations)
1 - divide A by B and then multiply by C
2 - multiply B by C and then use the result to divide A.
Each of these two interpretations can be represented in terms of a parse tree.

intermediate code generation -
The intermediate code generation uses the structure produced by syntax analyzer to create a stream of simple instructions. Many styles of intermediate code are possible. One common style uses instruction with one operator and a small number of operands. The output of syntax analyzer is some representation of a parse tree. The intermediate code generation phase transforms this parse tree into an intermed-iate language representation of the source program.

CC

code optimization -
This is optional phase described to improve the intermediate code so that the output runs faster and takes less space. Its output is another intermediate code program that does the same job as the original but in a way that saves times / spaces.

1- Local optimization -
There are local transformations that can be applied to a program to make an improvement.
For ex, if A > B goto L2
Goto L3
L2:
This can be replaced by a single statement
if A < B goto L3.

Another imp local optimization is the elimination of common sub-expressions.
A := B + C + D
E := B + C + F
might be evaluated as -
T1 := B + C
A := T1 + D
E := T1 + F

2. Loop optimization -
Another imp source of optimization concerns about increasing the speed of loop. A typical loop improvement is to move a computation that produces the same result each time around the loop to a point, in the program just before the loop is entered.

code generator -
C produces the object code by deciding on the memory locations for data, selecting code to access each data and selecting the registers in which each computation is to be done. Many computers have only a few high speed registers in which computations can be performed ~~easily~~ quickly. A good code generator would attempt to utilize registers as efficiently as possible.

Error handling -
One of the imp functions of compiler is the detection & reporting of errors in the source program. The error message should allow the programmer to determine exactly where the errors have occured. Errors may occur in all or the phases of compiler. Whenever a phase of compiler discovers an error, it must report error to the error handler, which issues an appropriate diagnostic msg. Both of the table-management and error-handling routines interact with all phases of compiler.

Ex -

position := initial + rate * 60.

↓

[lexical analyzer]

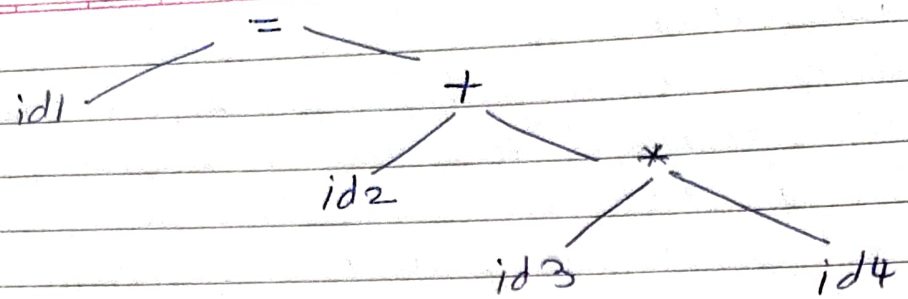
↓

tokens id1 = id2 + id3 * id4

↓

[syntax analyzer]

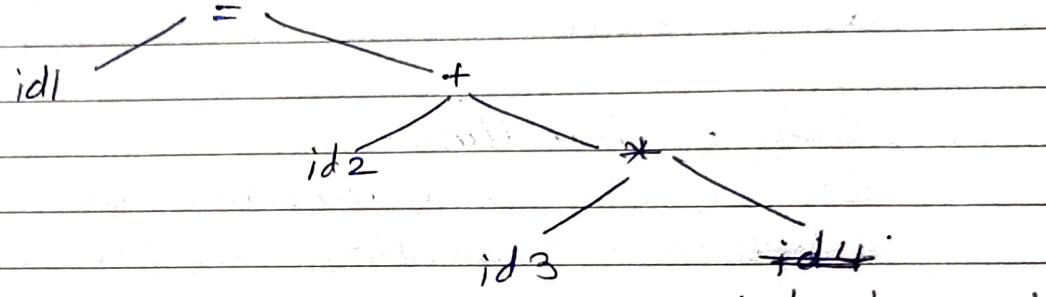
↓



↓

[semantic analyzer]

↓



↓

[intermediate code generator]

↓

```
temp1 := int to real (60)
temp2 := id3 * temp1
temp3 := id2 + temp2
id1 := temp3
```

↓

[code optimizer]

↓

```
temp1 := id3 * 60.0
id1 := id2 + temp1
```

↓

[code generator]

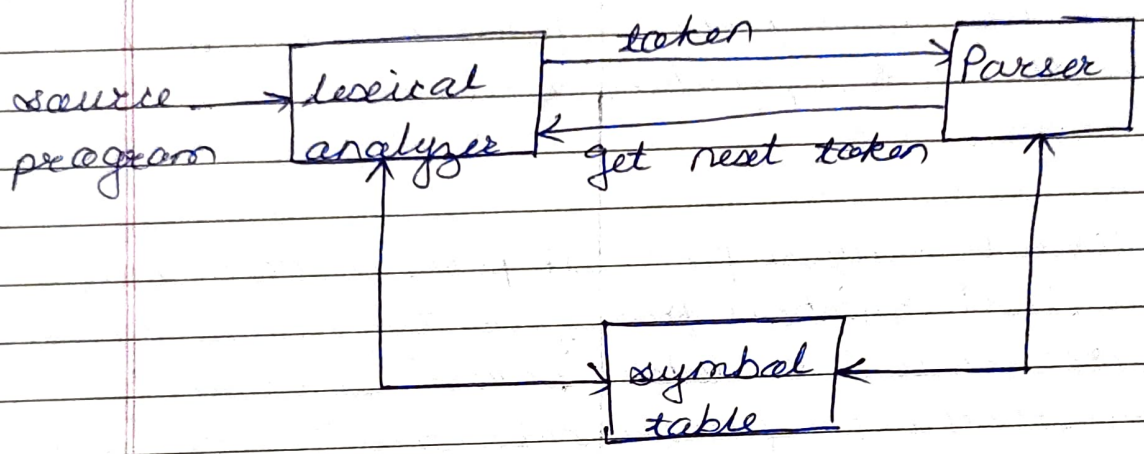
↓

```
MOV R2 * 60.0   M
MOV R2          A
```

9] Lexical analyzer :-

IMP

The LA is first phase of compiler. LA is called the linear analysis or scanning. In this phase the stream of characters making up the source program is read from left-to-right and grouped into tokens that are sequences of characters having a collective meaning.



upon receiving a 'get next token' command from the parser the lexical analyzer reads the input character until it can identify the next token. The LA return parser representation for the token it has found. The representation will be integer code, if the token is simple construct such as comma, parenthesis or colon.

LA may also perform certain secondary tasks as the user interface. One such task is striping out from the source program the commands & white spaces in the form of blank, tab & new line characters. Another is correlating error message from the compiler

with the source program.

(6) Lexical analysis vs parsing :-

Lexical analysis	parsing.
A scanner simply turns an input string (say a file) into a list of tokens. These tokens represent things like identifiers, parenthesis $, operators, etc.	A parser converts list of tokens into a like obj to represent how these token together form a cohesive / sentence.
The lexical analyzer pass individual symbols from source code into tokens. From there, the parser turns these whole tokens into sentences of your grammar	A parser does not give the nodes meaning beyond structural cohesion. The thing to do is extract meaning from this structure (called counter analysis).

(7) token :- Token is a sequence of characters that can be treated as a single logical entity.
Typical tokens are -

identifiers	special symbols
keywords	constants
operators	

pattern - A set of strings in the input for which the same token is produced as output. This set of string is described by a rule called a pattern associated with the token.

lexeme - A lexeme is a sequence of characters in the source program that is matched by the pattern for a token.

token	lexeme	pattern
const	const	const
if	if	if
relation	<, <=, =, <>, >=, >	< or <= or = or <> or >= or >
id	pi	any letter followed by letters & digit
num	3.14	any numeric constant
literal	"core"	any character b/w " and ".
		pattern.

A pattern is a rule describing the set of lexemes that can represent a particular token in source program.

17] Lexical errors -
Lexical errors are the errors thrown by the lexer when unable to continue. which means that there's no way to recognise a lexeme as a valid token for lexer. Syntax errors, will be thrown by scanner when a given set of already recognised valid tokens don't match any of the right sides of your grammar rules. Simple panic mode error handling system requires that we return to a high-level parsing function when a parsing or lexical error is detected.

Error - recovery actions are :-
- Delete one char from remaining input
- insert a missing char in remaining input
- Replace a char by another char
- Transpose two adjacent char.

[3] difference between compiler & interpreter.

1] A compiler converts high level instr^ into machine lang. while an interpreter converts high level lang. instr^ into an intermediate form.

2] Before execution, entire program is executed by the compiler whereas after translating the first line, an interpreter then executes it & so on.

3] List of errors is created by the compiler after the compilation process while the interpreter stops translating after the first error.

4] An independent executable file is created by the compiler whereas interpreter is required by an interpreted program each time.

5] The compiler produce object code whereas interpreter does not produce object code.

6] In the process of compilation the program is analyzed only once and the code is generated whereas source program is interpreted every time it is to be executed and every time the source program is analyzed. Hence interpreted

is less efficient than compiler.

Ex of interpreter - A UPS Debugger is basically a graphical source level Debugger but it contains built in C interpreter which can handle multiple source files.

Ex of compiler - Borland C compiler or Turbo C compiler compiles the programs written in C or C++.

14] specifications of tokens.
There are three specifications of tokens:-
(1) string
(2) lang.
(3) regular exp.

15] strings & lang.
An alphabet or character class is a finite set of symbols.
A string over an alphabet is a finite sequence of symbols drawn from that alphabet.
A language is any countable set of steings over some fixed alphabet.

In lang theory, the terms "sentence" and "word" are often used as synonyms for "string". The length of a string s is written as |s| is the number of occurences of symbols in s.
For ex, banana is a string of length six.
The empty string, denoted by ε is the string of length zero.

Operations on strings.

1] A prefix of string s is any string obtained by removing zero or more symbols from the end of string.
Ex - ban is prefix of banana.

2] A suffix of string s is any string obtained by removing zero or more symbols from the beginning of s.
Ex - nana is suffix of banana.

3] A substring of s is obtained by deleting any prefix and suffix from s.
Ex - nan is substring of banana.

4] The proper prefix, suffix and substring of a string s are those prefixes, suffixes & substrings resp. of s that are not ε or not equal to s itself.

5] A subsequence of s is any string formed by deleting zero or more not necessarily consecutive positions of s.
Ex - baan is subsequence of banana.

Operations on languages.
Let $L = \{0, 1\}$ and $S = \{a, b, c\}$

union = $L \cup S = \{0, 1, a, b, c\}$
concatenation = $L.S = \{0a, 0b, 0c, 1a, 1b, 1c\}$
Kleen closure = $L^* = \{\varepsilon, 0, 1, 00, 11, 010, 101, 11, \dots\}$
positive closure = $L^+ = \{0, 1, 00, 11, 010, 101, 1100, \}$
$L^+ = L^* - \{\varepsilon\}$

18] Regular expressions -

IMP

Each regular expression r denotes a language $L(r)$.

1. ε is a regular exp and $L(\varepsilon)$ is $\{\varepsilon\}$, i.e., the language whose sole member is the empty string.

2. If 'a' is a symbol in Σ, then 'a' is a regular exp, and $L(a) = \{a\}$ i.e., the language with one string of length one with 'a' in its one position.

3. The unary operator * has highest precedence and is left associative.
   ↑ same for (+).

4. Concatenation has second highest precedence and is left associative.

5. Suppose r and s are regular expressions denoting lang $L(r)$ & $L(s)$.
   Then,
   - $(r)|(s)$ is a regular exp denoting the lang. $L(r) \cup L(s)$.
   - ~~$(r).(r)$~~ $(r)(s)$ is a regular exp denoting the language $L(r)L(s)$.
   - $(r)^*$ is a regular exp denoting $(L(r))^*$.
   - (r) is a regular exp denoting $L(r)$.

6. . has lowest precedence & is left associative.

Regular definitions -

For notational convinience, we may wish to give names to regular exp & to define regular exp using these names as if they were symbols.

Identifiers are the set or string of letters & digits beginning with a letter.

Ex - $Ab^*|Cd?$ is equivalent to $(a(b^*))|(c(d?))$

letter - $A|B|\cdots|z|a|b|\cdots|z|$

digits - $0|1|2|\ldots|9$

Id - letter (letter/digits)*

Shorthands

Certain constructs occur so frequently in regular exp that it is convenient to introduce

1. One or more instances (+):

The unary postfix operator + means "one or more instances of".

If r is a regular exp that denote the language $L(r)$, then $(r)^+$ is a regular exp that denotes the lang $(L(r))^+$.

Thus the regular exp a^+ denotes the set of all strings of one or more a's.

* The operator + has the same precedence and
* associativity as the operator *.

2. Zero or one instance (?):

The unary postfix operator ? means "zero or one instance of".

The notation $r?$ is a shorthand for $r|\epsilon$.

If 'r' is a regular exp, then (r)? is a regular exp that denotes the language L(r) ∪ {ε}.

3. character classes:-
The notation [abc] where a, b & c are alphabet symbols denotes the regular exp a|b|c.
character class such as [a-z] denotes the regular exp a|b|c|...|z.
We can describe identifiers as being strings generated by the reg exp,
~~[A-Za-z]~~ [A-Z a-z][A-Z a-z 0-9]*

non regular set :-
A lang which cannot be described by any regular exp is a non-regular set.
Ex- The set of all strings of balanced parenthesis & repeating strings cannot be described by a regular exp. This set can be specified by a context-free grammar.

18] Recognition of tokens.
~~If~~ if → if
then → then
else → else
relop → <|<=|=|<>|>|>=
id → letter (letter | digit)*
~~num → digit+ (.digit)+ ?~~

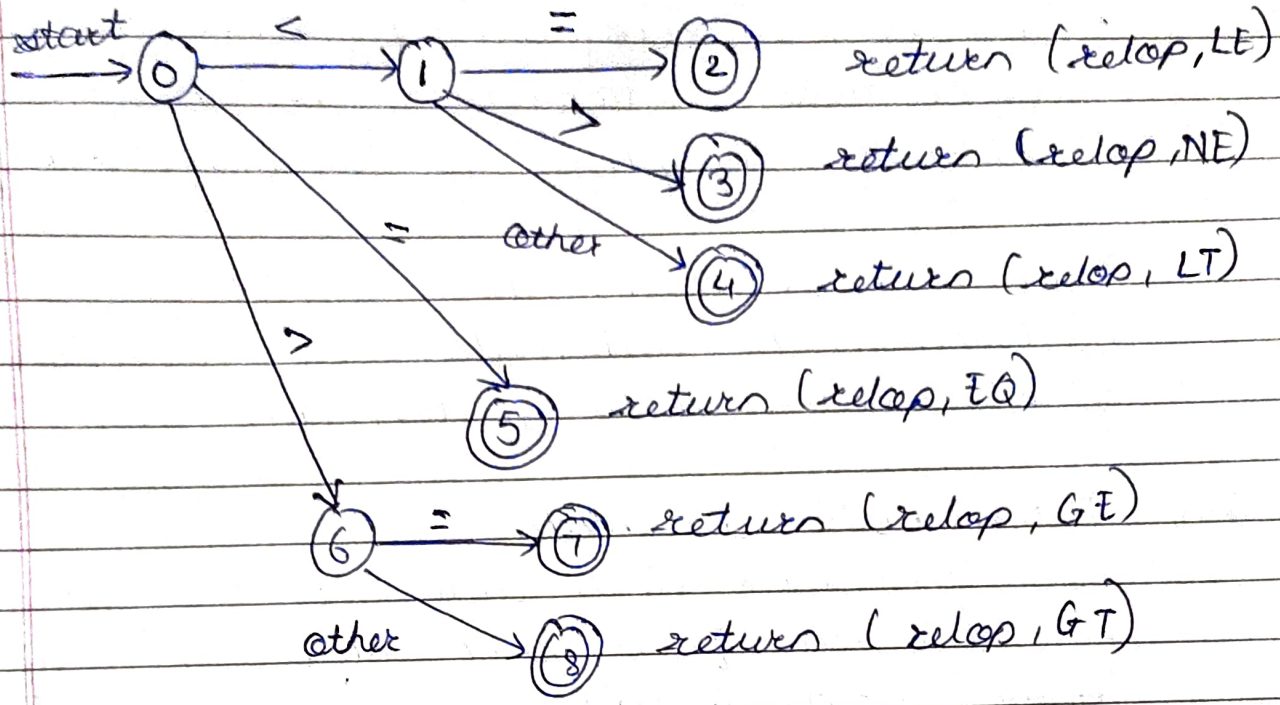
num → digit+ (.digit+)? (E(+|-)? digit+)?

lexeme	token name	attribute name.
any ws		
if	if	
then	then	
else		
any id	id	pointer to entry table
any num	number	pointer to table entry
<	relop	LT
<=	relop	LE
=	relop	ET
<>	relop	NE

19] Transition diagram.

Transition diagram has a collection of nodes or circles, called states. Each state represents a condition that could occur during process of scanning the input ~~taking~~ looking for a lexeme that matches one of several patterns. Edges are directed from one set of the transition diagram to another. Each edge is ~~directed~~ by labeled by a symbol or set of symbols

As an intermediate step in the constr^ of LA, we first produce a stylized flowchart called a transition diagram. Position in a transition diagram, are drawn as circles and are called as states.



20] Automata -

Automation is defined as a system where information is transmitted and used for performing some functions without direct participation of man.

1. An automation in which the ~~out~~ output depends only on the input is called automation without memory.
2. An automation in which the output depends on the input and state also is called as automation with memory.
3. An automation in which the output depends only on the state of machine is called a Moore machine.
4. An automation in which the output depends on the state & input at any instant of time is called a mealy machine.

- An automata has a mechanism to read input from input tape.

- Any lang is recognised by some automation. Hence these automation are basically language 'acceptors' or 'language recognizers'

Types of finite automata-
(1) Deterministic automata.
(2) non-deterministic automata.

2.] Deterministic automata.
A deterministic finite automata has at most one transition from each state on any ~~point~~ input. A DFA is a special case of NFA in which-
1. it has NO transitions on input ϵ
2. each input symbol has at most one transition from any state.

DFA is defined by 5 tuple notation.
$M = (Q, \Sigma, \delta, q_0, F)$
where Q = finite set of states (non-empty)
Σ = input alphabets
q_0 = initial state
δ = transition fun^ or mapping fun^, using this fun^ the next state can be determined.
F = set of final states.

The regular exp is converted into minimized DFA by:-
regular exp → NFA → DFA → minimised DFA

The finite automata is called DFA if there is only one path for a specific input from current state to next state.

22] Bootstrapping :-

When a computer is first turned on or restarted, a special type of absolute loader, called as bootstrap loader is executed. This bootstrap loads the first program to be run by the computer usually an operating system. The ~~boot~~ bootstrap itself begins at address 0 in the memory of the machine. It loads the operating system (or some other program) starting at address 80. After all 8 of the object code from device has been loaded, bootstrap program jumps to address 80, which begins the execution of the program that was loaded.

Such loaders can be used to run stand-alone programs independent of the OS or the system loader. They can also be used to load the OS or the loader itself into memory.

Loaders are of two types -

① linking loader
② linkage editor

Linkage loaders, perform all linking & relocation at load time.
Linkage editors, perform linking prior to load time & dynamic linking, in which the linking function is performed at execution time.