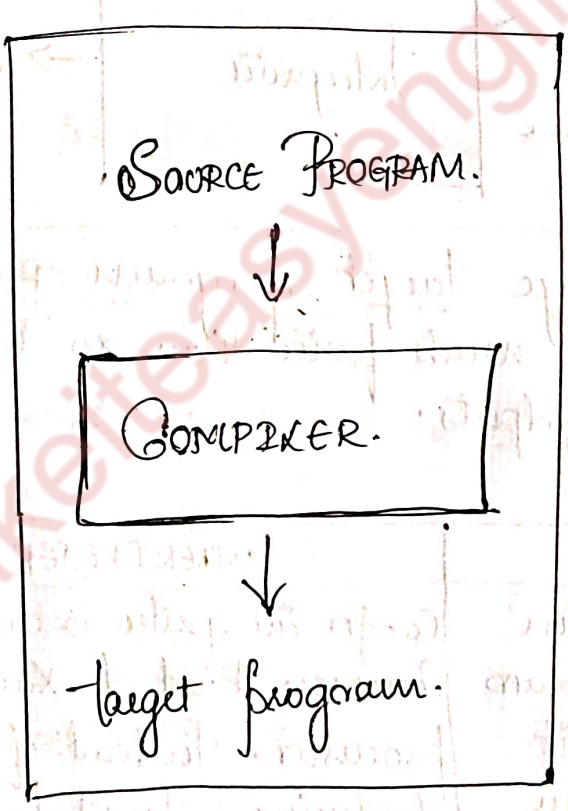
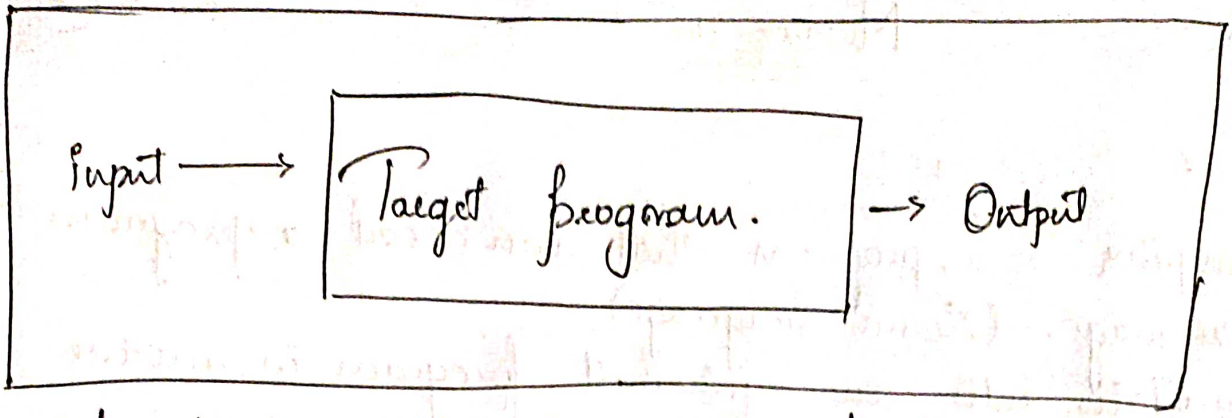
MODULE-II

LANGUAGE PROCESSORS:

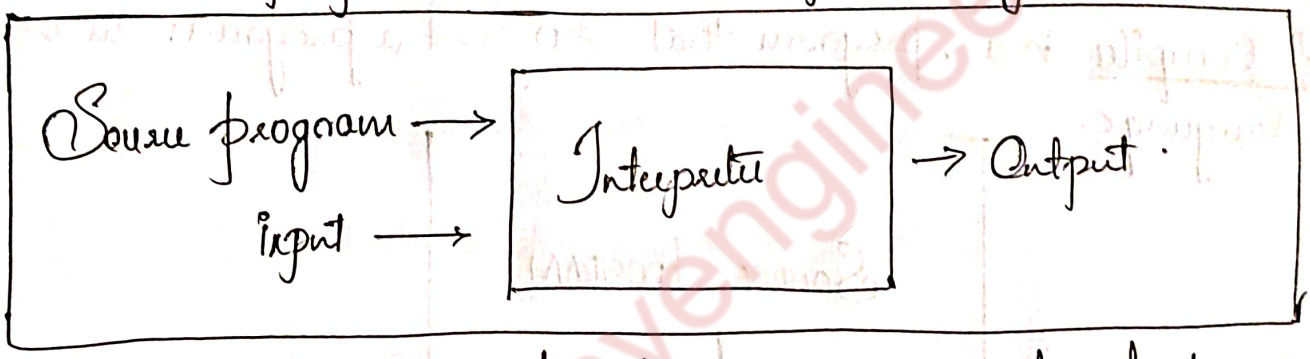
- A Compiler is a program that can read a program in one language - (Source language).
- It translates into an equivalent program in another language - (Target language).
- An Important role of the Compiler is to report any errors in the Source program that it decides or detects during translation process.
- A Compiler is a program that can read a program in one language.



- If the target program is an executable machine-language program.
- It can then be called by the user to process inputs and produce Outputs.



- An Interpreter is another Common kind of Language processor
- Instead of producing a target program as a translation an Interpreter appears to directly execute the operations specified in the Source program on inputs Supplied by the user.

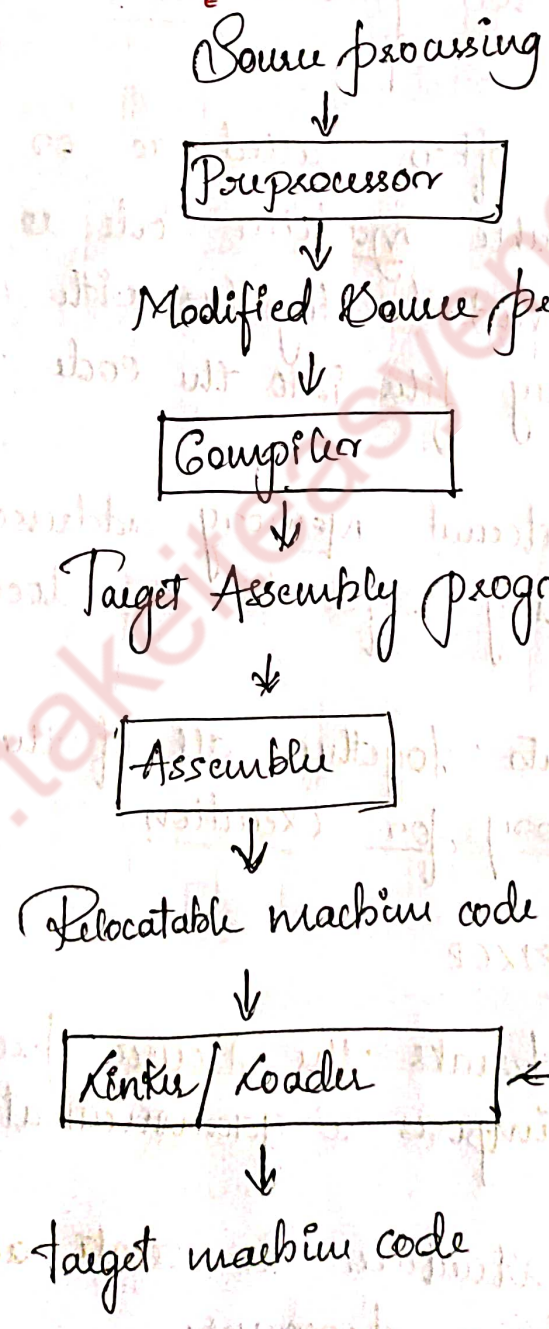


- The machine-language target program produced by a Compiler is usually much faster than an Interpreter at mapping inputs to outputs.

COMPILER	INTERPRETER.
A Compiler is a program that can read a program in a language and it is translate it into an equivalent program in another language.	An interpreter is another common kind of language processor. Instead of producing a target program as a translation. It appears directly to execute the operations.
The Machine-language target program produced by a Compiler is usually much faster than Interpreter.	Interpreter is slower in Mapping inputs to outputs.

⊕ Error diagnostics is less than Interpreter	⊕ An Interpreter is slower. However, can usually give better error diagnostics than a Compiler, because it executes the Source program statement by statement.

LANGUAGE PROCESSING



- A source program may be divided into modules stored in a separate files.
- The task of collecting the source program is sometimes entrusted to a separate program called as Preprocessor.
- The preprocessor may also expand shorthands, called macros into source language statements.
- The modified source program is then fed to a compiler.
- The compiler easier to produce as o/p and easier to debug.
- The assembly language is then processed by a program called an assembler. that produces relocatable machine code as its o/p.
- Large programs are often called as or compiled in pieces, so the relocatable machine code as its output.
- May have to be linked together with other relocated object files and library files into the code that actually runs on the machine.
- The linker resolves external memory addresses, where the code in one file may refer to the location in another file.
- The loader then puts together all of the executable object files into memory for execution.

THE STRUCTURE OF A COMPILER

- The Analysis part breaks the source progeam into constituent pieces. and imposes a grammatical structure on them.
- It then uses this structure to create an intermediate representation of the source program.

↳ If the analysis part detects that the Source program is either Systematically / Syntatically ill formed or Symantically Unsound.

↳ Then it must provide informative messages, So the user can take Corrective action.

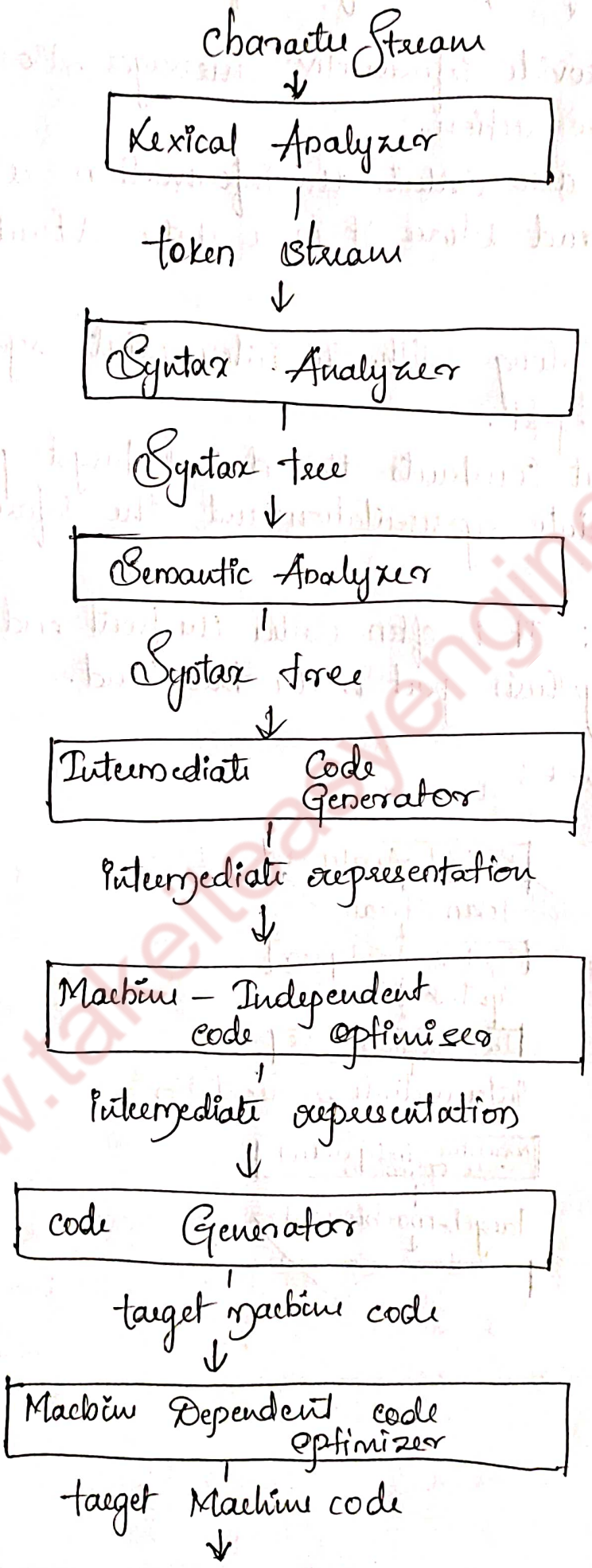
↳ The analysis part also Collects the information about the Source program and Stores it in a data Structure called a Symbol table.

↳ Which is passed along with the intermediate representation to the Synthesis part.

↳ The Synthesis part Constructs the desired target program from the intermediate representation and the information in the Symbol table.

↳ The analysis part: It is often Called the front end of the Compiler the Synthesis part is the back end..

PHASES OF COMPILER



Lexical Analysis:

- The first phase of a Compiler is called lexical analysis or Scanning.
- The Lexical analyzer reads the Stream of characters making up the Source and groups the characters into meaningful Sequences called lexemes.
- For each, lexeme the Lexical analyzer produces as output a token of the form that it passes on to the Subsequent phase, Syntax analysis, and the Second Component attribute value points to an entry in the Symbol table for this token.
- Information from the Symbol-table entry is needed for Semantic analysis and code generation.

For Example,

Suppose a Source program Contains the assignment Statement.

position = initial + rate * 60

Which the representation of the assignment Statement after lexical analysis as the Sequence of tokens.

(id, 1) (=) (id, 2) (+) (id, 3) (*) (60)

Syntax Anarysis:

- The Second phase of the Compiler is Syntax analysis or parsing
- The parser uses the first Components of the tokens produced by the Lexical analyzer to Create a tree-like intermediate representation that depicts the grammatical Structure of the token Stream.
- A typical representation is a Syntax Tree in which each interior node represents an Operation and the children of the node represent the arguments of the Operations.

SEMANTIC ANALYSIS

- The Semantic analyzer uses the Syntax tree and the Information in the Symbol table to check the Source program for Semantic Consistency with the language definition.
- It also gathers type information and saves it to either the Syntax tree or the Symbol table, for subsequent use during Intermediate-code generation.
- An important part of Semantic analysis is type checking, where the Compiler checks that each Operator has matching Operands.

For Example:

- Many programming language definitions require an array index to be an integer.
- the Compiler must report an error if a floating point number is used to index an array.
- The Language Specification may permit Some type Conversions called co-ercions.

INTERMEDIATE CODE GENERATION:

- After Syntax and Semantic analysis of the Source program, Many Compilers generate an explicit low-level or machine like Intermediate representation. ~~Should have two~~
- which we can think of as a program for an abstract machine.
- This Intermediate representation should have two important properties:
  - (i) It should be easy to produce and
  - (ii) It should be easy to translate into the target machine

- An intermediate form called three-address code.
- Which consist of a sequence of assembly like instructions with three operands per instruction.
- Each operand can act like a register
- The output of the intermediate code generator consist of a sequence of assembly-like instructions with three operands per instructions.
- Each operand can act like a register.
- The output of the intermediate code generator consists of the three-address code sequence.
- There are several points worth nothing about three-address instructions.
- First, each three address assignment instruction tpe at most one operator on the right side.
- Thus, these instructions fix the order in which operations are to be done.
- The Multiplication precedes the addition in the source program
- Second, the Compiler must generate a temperory name to hold the value computed by a three-address instruction.
- Third, some "three-address instructions" like the first and last in the sequence

CODE OPTIMISATION:

- The Machine ~~dependent~~ independent code-optimisation phase attempts to improve the intermediate code so that better target code will result.
- Usually better means faster, but other objectives may be desired, such as shorter code, or target code that consumes less power.

For Example:

- A straightforward algorithm generates the intermediate code, using an instruction for each operator in the tree representation that comes from the semantic analyzer.

$t_1 = id_3 * 60.0$

$id_1 = id_2 + t_1$

CODE GENERATION:

- The code generator takes as input an intermediate representation of the source program and maps it into the target language.
- If the target language is machine code, registers or memory locations are selected for each of the variables used by the program.
- Then, the intermediate instructions are translated into sequences of machine instructions that perform the same task.
- A crucial aspect of code generation is the judicious assignment of registers to hold variables.

SYMBOL - TABLE MANAGEMENT

- The symbol table is a data structure containing a record for each variable name, with fields for the attributes of the name.
- The data structure should be designed to allow the compiler to find the record for each name quickly.

Position = initial + rate * 60

↓

Lexical Analyser

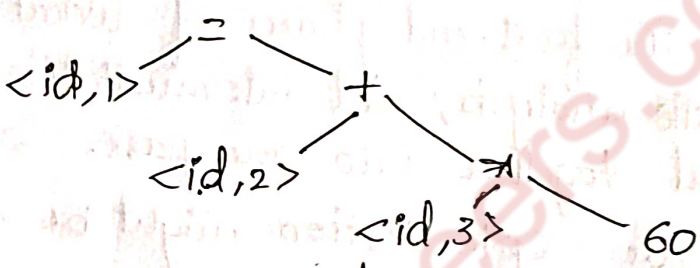
↓

<id,1> <=> <id,2> <+> <id,3> <*> <60>

↓

Syntax Analyser

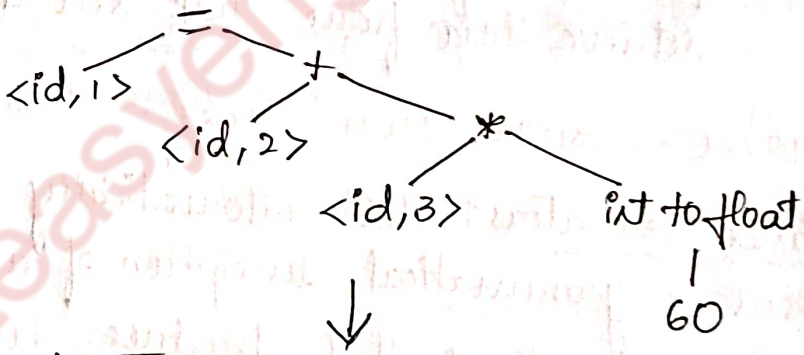
↓



↓

Semantic Analyser

↓



1	position	...
2	initial	...
3	rate	...

SYMBOL TABLE

↓

Intermediate Code Generator

t_1 = int to float (60)
t_2 = id_3 * t_1
t_3 = id_2 + t_2
id_1 = t_3

↓

Code Optimizer

t_1 = id_3 * 60.0
id_1 = id_2 + t_1

↓

Code Generator

LDF R_2, id_3
MULF R_2, R_2, #60.0
LDF R_1, id_2
ADDF R_1, R_1, R_2
STF id_1, R_1.

The Grouping Of Phases into Passes

- The discussion of phases deals with the logical organisation of a compiler.
- In an implementation, activities from several phases may be grouped together into a pass that reads an input file and writes an output file.

For Example:

- The front-end phases of lexical analysis, syntax analysis, semantic analysis, and intermediate code generation might be grouped together into one pass.
- Code optimization might be an optional pass.
- Then there could be a back-end pass consisting of code generation for a particular target machine or retrieve data from that record quickly.

Compiler-Construction Tools

1. Parser Generators: that automatically produce syntax analyzers from a grammatical description of a programming language.
2. Scanner Generators: that produce lexical analyzers from a regular-expression description of the tokens of a language.
3. Syntax-directed translation: engines that produce collection of routines for walking a parse tree and generating intermediate code.
4. Code-generator generators: that produce a code generator from a collection of rules for translating each operation of the intermediate language into the machine language for a target machine.
5. Data-flow analysis engines: that facilitate the gathering of information about how values are transmitted from one part of a program to each other part.

The Evolution Of Programming Languages:

The move to higher-level languages:

- The first step towards more people-friendly programming languages was the development of mnemonic assembly language in early 1950's.
- Initially, the instructions in an assembly language were just mnemonic representations of machine instructions.
- Later, macro instructions were added to assembly languages so that a programmer could define parameterized shorthands for frequently used sequences of machine instructions.

Impacts On Compilers:

- Compilers can help promote the use of high-level languages by minimizing the execution overhead of the programs written in these languages.
- Compilers are also critical in making high-performance computer architectures effective on users applications.

The Science Of Building A Computer

- A compiler must accept all source program 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.

Implementations of high-level programming Languages:

- A high-level programming language defines a programming abstraction.
- The programmer expresses an algorithm using the language, and the Compiler must translate that program to the target language.
- Optimizing Compilers include techniques to improve the performance of Generated code, thus offsetting the in-efficiency introduced by high-level

Optimizations for Computer Architectures

- The rapid evolution of Computer Architectures has also led to an insatiable demand for new Compiler technology.
- Almost all high-performance Systems take advantage of the same two techniques: parallelism and memory hierarchies.
- Parallelism can be found at several levels:
  - At instruction level
  - The processor level.

Parallelism:

- All modern microprocessors exploit instruction-level parallelism.
- Programs are written as if all instructions were executed in Sequence.
- The hardware dynamically checks for dependencies in the Sequential instructions stream and issues them in parallel when possible.
- In Some Cases, the machine includes a hardware Scheduler that can change the instruction ordering to increase the parallelism in the program.

Memory Hierarchies:

- A memory Hierarchy Consist of Several levels of Storage with different speeds and sizes, with the level closest to the processor being the fastest but Smallest.
- The average memory-access time of a program is reduced if most of its accesses are satisfied by the faster levels of the hierarchy.
- Both parallelism and the existance of a memory hierarchy improve the potential performance of a machine, but they must be harnessed effectively by the Compiler to deliver real performance on an application.

Design of new Computer architectures:

- In the early days of Computer architecture design, Compilers were developed after the machines were built.
- Since programming in highlevel languages is the norm, the performance of a Computer System is determined not by its raw speed but also by how well Compilers can exploit its features.

Program Translations:

- While we normally think of Compiling as a translation from a highlevel language to the machine level.
- The same technology can be applied to translate between different kinds of languages.

Software productivity Tools

- Programs are arguably the most Complicated Engineering artifacts ever produced.
- They Consist of many details, every one of which must be Correct before the program will work Completely.

LEXICAL ANALYSIS :

- Lexical analysis read characters from left to right and groups into tokens.
- A Simple way to build lexical analyzer is to Construct a diagram to illustrate the Structure of tokens of the Source program.
- This approach makes it easier to modify a lexical analyzer since we have only to rewrite the affected patterns, not the entire program.

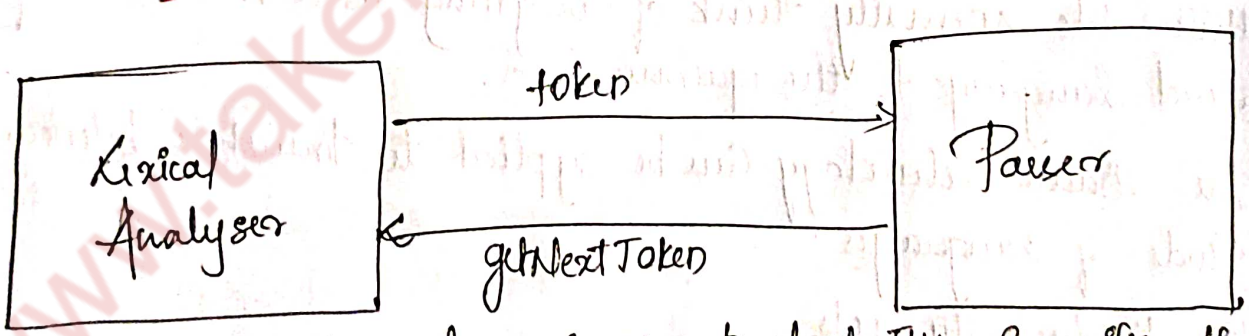
Three Important approaches for implementing lexical analyzer are:

(i) Use lexical analyzer generator (LEX) from a regular expression based Specification that provides routines for reading and buffering the input.

(ii) Write lexical analyzer in Conventional language using I/O facilities to read input.

(iii) Write lexical analyzer in assembly language and explicitly manage the reading of input.

THE ROLE OF THE LEXICAL ANALYSER.



- Since the Lexical analyzer is the part of the Compiler that reads the Source text.
- It may perform Certain other tasks besides identification of lexemes.
- One Sub task is Stripping out Comments and whitespace (blank, newline, tab and perhaps other characters that are used to Seperate tokens in the input).

- Another task is correlating error messages generated by the Compiler with the source program.
- For instance, the lexical analyzer may keep track of the number of newline characters seen, so it can associate a line number with each error message.
- In some Compilers, the lexical analyzer makes a copy of the source program with the error messages inserted at the top. appropriate positions.
- If the source program uses a macro-preprocessor, the expansion of Macros may also be performed by the lexical analyzer.
- It is the first phase of a Compiler. It reads source code as input and sequence of tokens as output.
- This will be used as input by the parser in Syntax analysis
- Upon recieving 'getNextToken' from parser, lexical analyzer searches for the next token.

TOKENS PATTERNS AND LEXEMES :

- Token is a terminal symbol in the grammar for the source language.
- When the character sequence 'pi' appears in the source program, a token representing identifier is returned to the parser.
- A token name is an abstract symbol representing a kind of lexical unit,
  Eg: a particular keyword, or a sequence of input characters denoting an identifier.
- The token names are the input symbols that the parser processes.
- We often refer to a token by its token name.

- Pattern is a rule describing the set of lexemes that can represent a particular token in source programs.
- A pattern is a description of the form that the lexemes of a token may take.
- In the case of a keyword as a token, the pattern is just the sequence of characters that form the keyword.
- For identifiers and some other tokens, the pattern is a more complex structure that is matched by many strings.

LEXEME:

- Lexeme is a sequence of characters in the source program that is matched by the pattern for a token
- A lexeme is a sequence of characters and is identified by the lexical analyzer as an instance of that token

In programming languages, the following classes cover most or all of the tokens:

1. One token for each keyword. The pattern for a keyword is the same as the keyword itself.
2. Tokens for the 1 operators, either individually or in classes such as the token comparison
3. One token representing all identifiers.
4. One or more tokens representing constants, such as numbers and literal.
5. Tokens for each punctuation symbol, such as left and right parenthesis, comma and semicolon.

Ex: Identify the lexemes and write the pattern and token for $a=b*10$.

Lexemes are $a=b$, $*$, 10

pattern for identifiers = letter followed by letter or digit (or)

Letter (Letter | digit)*

Pattern for numbers = any digits 0-9 (or) digit (+)

Token for identifiers is <id, pointer to Symbol table entry>

Token for numbers = <Num, Value>

ATTRIBUTES OF TOKENS:

- A token has only a single attribute - a pointer to the Symbol-table entry in which the information about the token is kept.
- The token names and associated attributes values for the Statement.

  E = M*C**2.

  are written below as a sequence of pairs.

<id, pointer to Symbol-table entry for E>
<assign_op>
<id, pointer to Symbol-table entry for M>
<mult_op>
<id, pointer to Symbol-table entry for C>
<exp_op>
<number, integer value 2>

LEXICAL ERRORS:

- It is hard for a lexical analyser to tell, without the aid of other components, that there is a source-code error.

  For instance, if the string fi is encountered for the first time in C program in the context:

  fi (a == f(x)) . . . .

- A lexical analyzer cannot tell whether fi is a misspelling of the keyword if or an undeclared function identifier.
- Since fi is a valid lexeme for the token id, the parser and
- but some other phase of the compiler.
- Probably the parser in this case - handle an error due to transposition of the letters.

- However, suppose a situation arises in which the lexical analyser is unable to proceed because none of the patterns for tokens matches any prefix of the remaining input.
- The simplest recovery strategy is "panic mode" recovery.
- We delete successive characters from the remaining input, until the lexical analyser can find a well-formed token at the beginning of what input is left.
- This recovery technique may confuse the parser, but in an interactive computing environment it may be quite adequate.

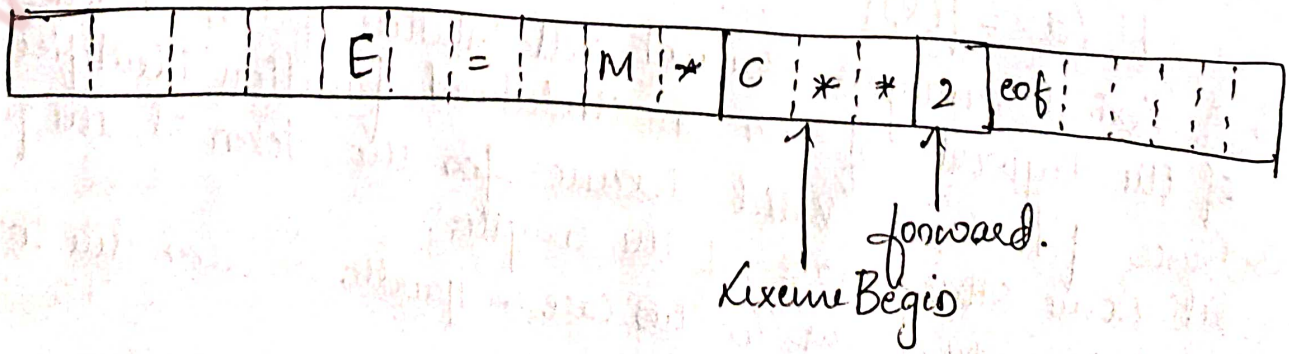
Other possible error-recovery actions are:

1. Delete one character from the remaining input.
2. Insert a missing character into the remaining input.
3. Replace a character by another character.
4. Transpose two adjacent characters.

INPUT BUFFERING:

- Before discussing the problem of recognizing lexemes in the input, let us examine some ways that the simple but important task of reading the source program can be speeded.
- This task is made difficult by the fact that we often have to look one or more characters beyond the next lexeme before we can be sure we have the right lexeme.

Buffer pairs:



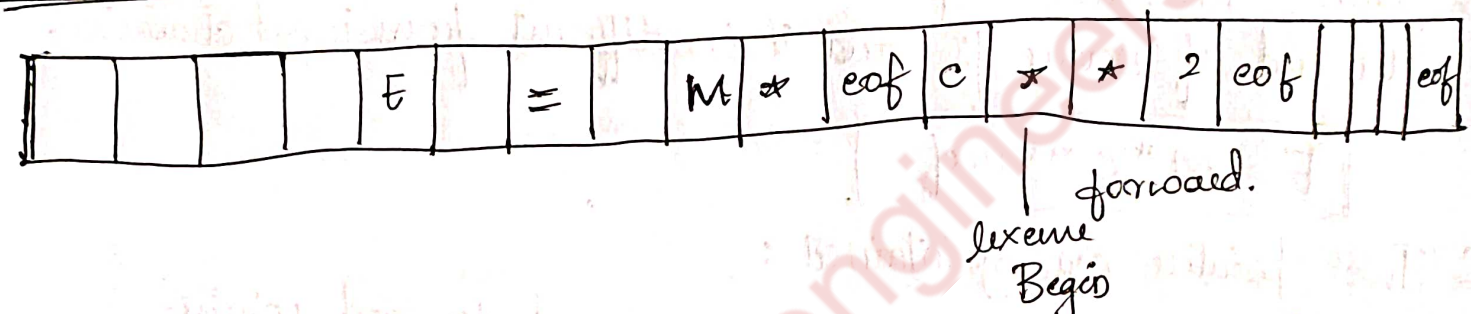
- Because of the amount of time taken to process characters and the large number of characters that must be processed during the compilation of a large source program.,
- Specialised buffering techniques have been developed to reduce the amount of overhead required to procus a single input character.
- Kook ahead of characters in the input is necessary to identify tokens.
- Specified buffering techniques have been developed to reduce the large amount of time consumed in moving characters.
- A buffer (array) divided into two N-characters halves.
- Where N = number of characters on one disk block 'eof' marks the end of source file and it is different from input character.

$$E = M * C ** 2 eof.$$

- Two pointers are maintained :
  - Beginning of the lexeme pointer and forward pointer.
- both pointers point to the first characters of the next lexeme to be found.
- Forward pointer scans ahead until a match for a pattern is found.
- Once the next lexeme is determined, processed and both pointers are set to the character immediately past the lexeme
- If the forward pointer moves halfway mark, the right N half is filled with new characters.
- If the forward pointer moves right end of the buffer then left N half is filled with new characters.
- The disadvantage is look ahead is limited and it is impossible to recognise tokens, when distanced between the two pointers is more than the length of the buffer.

- Advancing forward requires that we first test whether we have reached the end of one of the buffers.
- If so, we must reload the other buffer from the input, and move forward to the beginning of the newly loaded buffer.
- As long as we never need to look so far ahead of the actual lexeme that the sum of the lexeme's length plus the distance we look ahead is greater than N, we shall never overwrite the lexeme in its buffer before determining it.

SENTINALS :



- It is an Extra key inserted at the end of the array.
- It is Special, dummy character that can't be part of source program.
- With respect to buffer pairs, the code for advancing forward pointer is

Code with Sentinals

```
Switch (*forward ++)
  {
   case eof:
        if (forward is at end of first buffer)
        {
          reload second buffer;
          forward = beginning of second buffer:
        }
      else if (forward is at end of second buffer)
        {
          reload first buffer:
          forward = beginning of first buffer:
        }
      else /* eof within a buffer marks the end of input */
          terminate lexical analysis;
      break
  Cases for the other characters.
  }.
```

SPECIFICATION OF TOKENS

- An Alphabet is any finite set of symbols.
- Typical examples of symbols are letters, digits, and punctuation
- The set {0,1} is the binary Alphabet
- A String over an Alphabet is a finite sequence of symbols drawn from that Alphabet
- In language theory, the terms "sentence" and "word" are often used as synonyms for "string".

The length of string S, usually written |s|, is the number of occurances of symbols in S.

For Example:

Banana is a string of length six, The empty string denoted ε, is the string length of zero.

The following string-related terms are commonly used:

1. A prefix of string S is any string obtained by removing zero or more symbols from the end of S.

For Ex:

ban, banana, and ε are prefixes of banana.

2. A suffix of string S is any string obtained by removing zero or more symbols from the beginning of S.

For Ex:

nana, banana and ε are suffixes of banana.

3. A substring of s is obtained by deleting any prefix and any suffix from s.

For Instance,

banana, nan, and ε are substrings of banana.

4. The proper prefixes, suffixes, and substrings of a string s are those, prefixes, suffixes and substrings respectively. 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 consequitive positions of s.

for Ex: baan is a subsequence of banana.

String (s): sentence/word: finite set/sequence of symbols.

closure of L.

- $|S|$ = number of symbols in String S
  eg: S = banana, then $|S| = 6$
  Prefix (S): S = banana, ban
  Suffix (S): S = banana, nana
  Substring (S): S = banana, ba, na, na
  ϵ : empty string, $S = \epsilon$ then $|S| = 0$, ϕ = empty set.
  Language: set of Strings.
  If X and Y are strings then xy is Concatenation.

$$S\epsilon = \epsilon S = S$$
$$S^0 = \epsilon$$
$$S^1 = S$$
$$S^2 = SS$$
$$S^3 = SSS$$
$$S^i = S^{i-1}S \quad (\text{if } i > 0)$$

- A Language is any Countable set of Strings Over some fixed alphabet.
- This difinition very broad.
- Abstract languages like 0, the empty set, or $\{\epsilon\}$, the set Containing Only the empty string, are languages Under this definition.
- C programs and the set of all grammatically Correct English Sentences, although the latter two languages are difficult to Specify exactly.

$L_1 \cup L_2$: Union operation, where L_1 = Set of alphabet $\{A..Z, a..z\}$ and L_2 = Set of digits $\{0..9\}$
$L_1 L_2$: Concatenation
L_1^4: exponentation : set of Strings with 4 letters.

$$L^0 = \epsilon$$
$$L^i = L^{i-1}L$$
L^* = all Strings with ϵ : Kleen closure of L.

D : Set of all strings of digits of one or more.

Operations On Languages

1. LUD is the set of letters or digits – strictly speaking the language with 62 strings of length one, each of which strings is either one letter or one digit
2. LD is the set of strings of length two, each consisting of one letter followed by one digit
3. L^4 is the set of all 4-letter strings
4. L^* is the set of all strings of letters, including ϵ, the empty string
5. $L(LUD)^*$ is the set of all strings of ~~one or more digits~~.
6. letters and digits beginning with a letter.
7. D^+ is the set of all the strings of one or more digits.

Regular Expressions

- It is a notation which allows defining the sets precisely.
  Eg: $L(LUD)^*$ its regular expression is:
  letter (letter/digit)*
  Regular expression over alphabet has following data or rules:
- Is a regular Expression, the set containing empty string
  a is a regular expression.
- If a belongs to – this is {a} set containing the string a.
- Suppose r and s are regular expression.

denoting the languages L(r) and L(s) then

(r/s) is a regular expression denoting $L(r) \cup L(s)$

- rs is a regular expression denoting $L(r)\,L(s)$
  $(r)^*$ is a regular expression is said to be a regular set*, Concatenation and | has highest to lowest precedence with left associative
- If two regular expressions 'r' and 's' denote the same language we say 'r' and 's' are Equivalent and write r=s.

Recognition Of Tokens

- So, Let's see how to take the patterns for all the needed tokens and build a piece of code that examines that input string and finds a prefix that is a lexeme matching one of the patterns.
- Consider the following grammar, fragment/regular definitions:

```
stmt → if expr then stmt
expr → term relop term
term → id | num

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

- Keywords cannot be used as identifiers
  Num represents unsigned int and real numbers.
- The regular definition, ws,

```
delim → blank|tab|newline
ws → delim+
```

- If 'ws' was found, the lexical analyzer does not return a token to the parser.
- Our Goal is to Construct a lexical analyzer to produce a pair consisting of token and attribute-value as Output using the Transition table

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.	

TRANSITION DIAGRAMS

- These are the flow charts, as an intermediate step in the construction of a lexical analyzer.
- This takes actions when a lexical analyzer is called by the parser to get the next token.
- We use transition diagram to keep track of information about characters that are seen as and when the forward pointer scans the input.
- Lexeme beginning pointer points to the character following the last lexeme found.

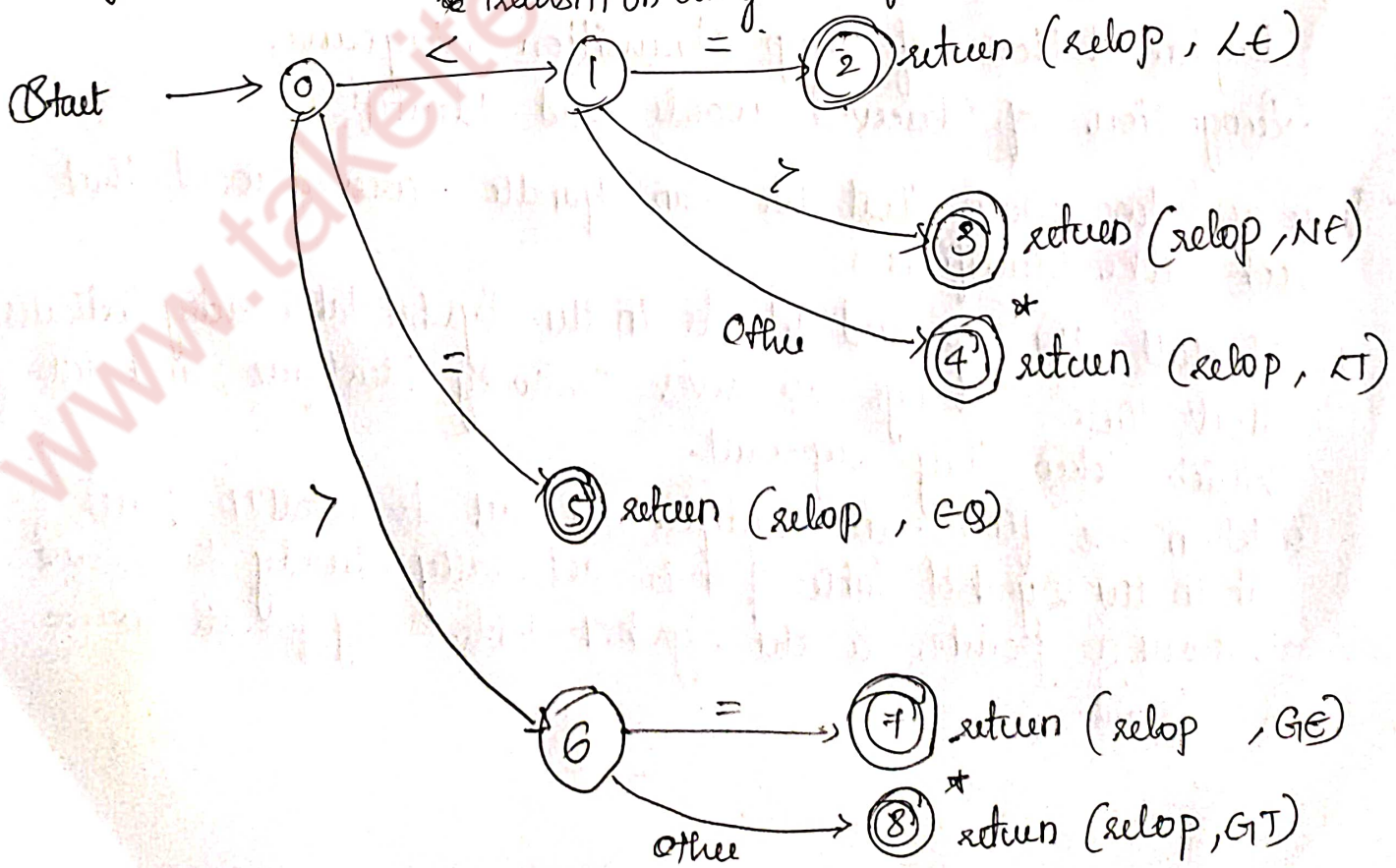
$$E = M * C ** 2 \text{ eof}$$

- Transition diagrams have a collection of nodes or circles. called states.

Some Important Conventions about Transition diagrams are:

- Certain states are said to be accepting, or final. These states indicate that a lexeme has been found, although the actual lexeme may not consist of all positions b/w the lexemeBegin and forward pointers.
- We always indicate an accepting state by a double circle, and if there is an action to be taken.
- Typically returning a token and an attribute value to passes.
- We shall attach that action to the accepting state
- If it is necessary to retract the forward pointer one position.
- Then we shall additionally place a * near that accepting state.
- One state is designated the start state, or initial state; it is indicated by an edge, labelled "start" entering from nowhere.
- The transition state always begins in the start state before any of symbols have been read.

Transition diagram of RELOP



Transition Diagram for Relop

```
TOKEN getRelop ()
{
  TOKEN retToken = new (RELOP);
  while (1)
  {
    or failure occurs
    Switch (state)
    Case 0: c = nextChar();
    if (c == '<') state = 1;
    else if (c == '=') state = 5;
    else if (c == '>') state = 6;
    else fail();
     break;
     Case 1: ...
     Case 8: retract();
     retToken . attribute = GT;
     return (retToken);
```

Implementation of relop transition diagram.

Recognition of Reserved words and Identifiers

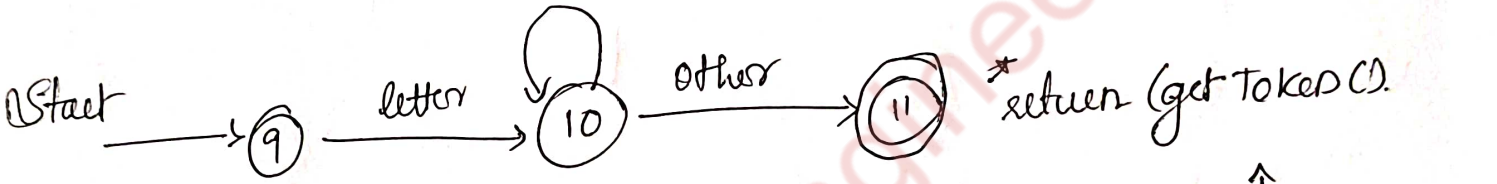
There are two ways that we can handle reserved word that look like identifiers:

1. Install the reserved words in the Symbol table entry indicates that these strings are never ordinary identifiers, and tells which token they represent.
   & when we find an identifier, a call to installID places it in the Symbol table if it is not ~~already~~ already there and returns a pointer to the Symbol-table entry for the lexeme found.

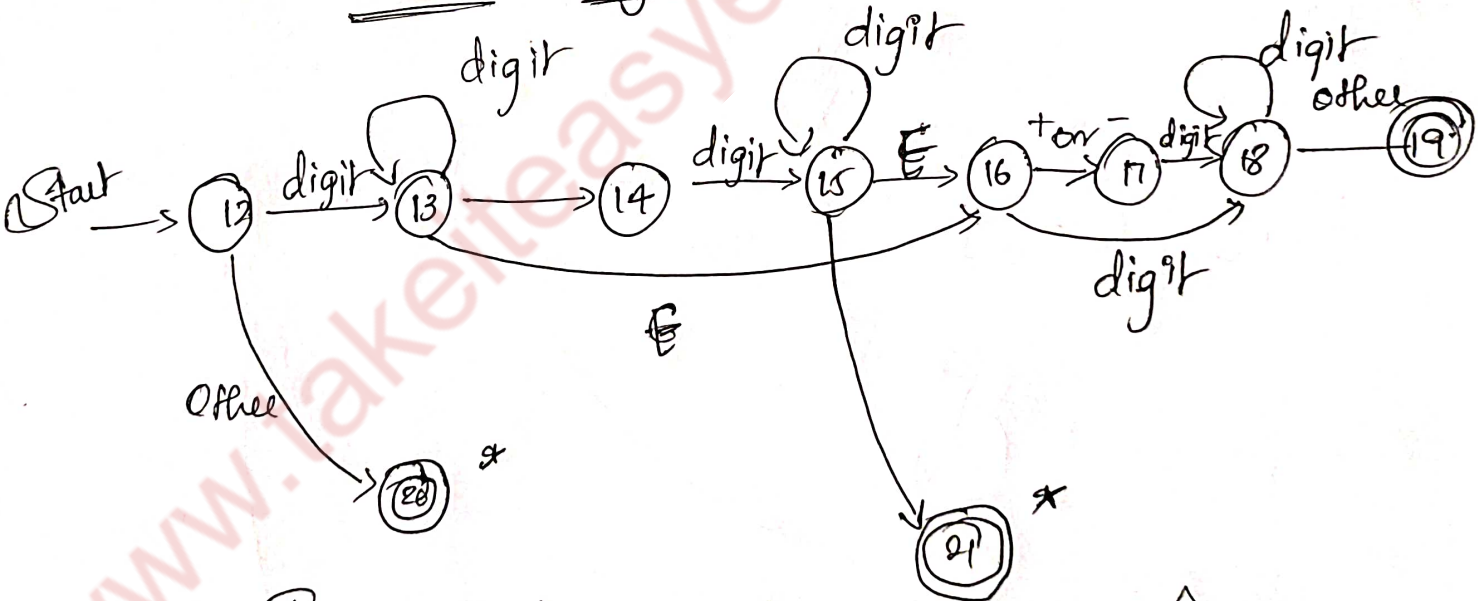
- If any identifier not in the Symbol-table entry for the lexeme found
- either id or one of the keyword tokens that was initially installed in the table.

v. Creati Separati Transition diagram for each keyword; an example for the keyword then.

- It is necessary to check the identifier has ended, or else we would return token then in situations where the correct token was id, with a lexeme like the next value that has then as a proper prefix.



Transition diagram of unsigned number ↑



Transition diagram of White Space ↑.