Unit 1: Syllabus

- Introduction: Compiler Introduction; Analysis of source program; *Phases and Passes of Compiler*; Symbol table & its implementation; *Cousins of a Compiler*; Compiler Construction Tools; Bootstrapping: *Regular Grammar and Regular Expressions.*
- Lexical analysis: *Role of a Lexical Analyzer; Input Buffering;* Specifications of Tokens; Recognition of Tokens; LEX Tool and its Implementation

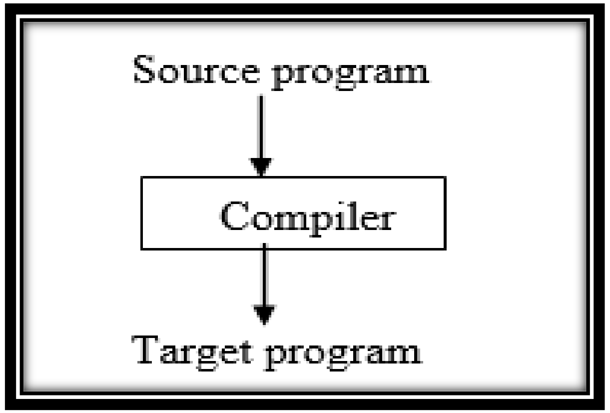
Unit 1

Language processor:

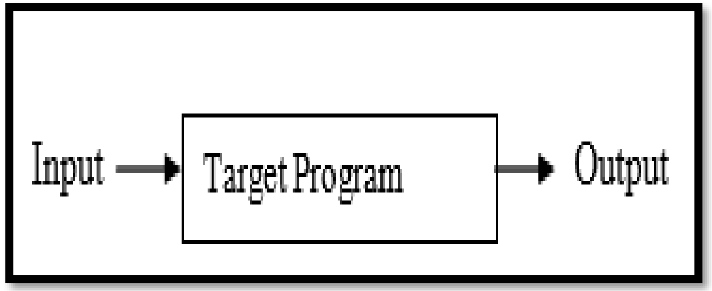
A language translator is a program which is used to translate an input program written in one programming language into another programming language (output program). Language processor is also called a **language translator.**

Compiler:

A compiler is a program that can read a program in one language - the *source* language - and translate it into an equivalent program in another language - the *target* language.

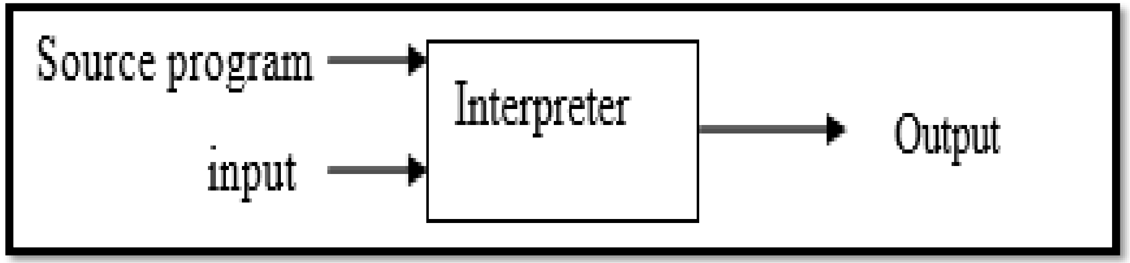


- If the target program is an executable machine-language program, it can then be called by the user to process inputs and produce outputs.

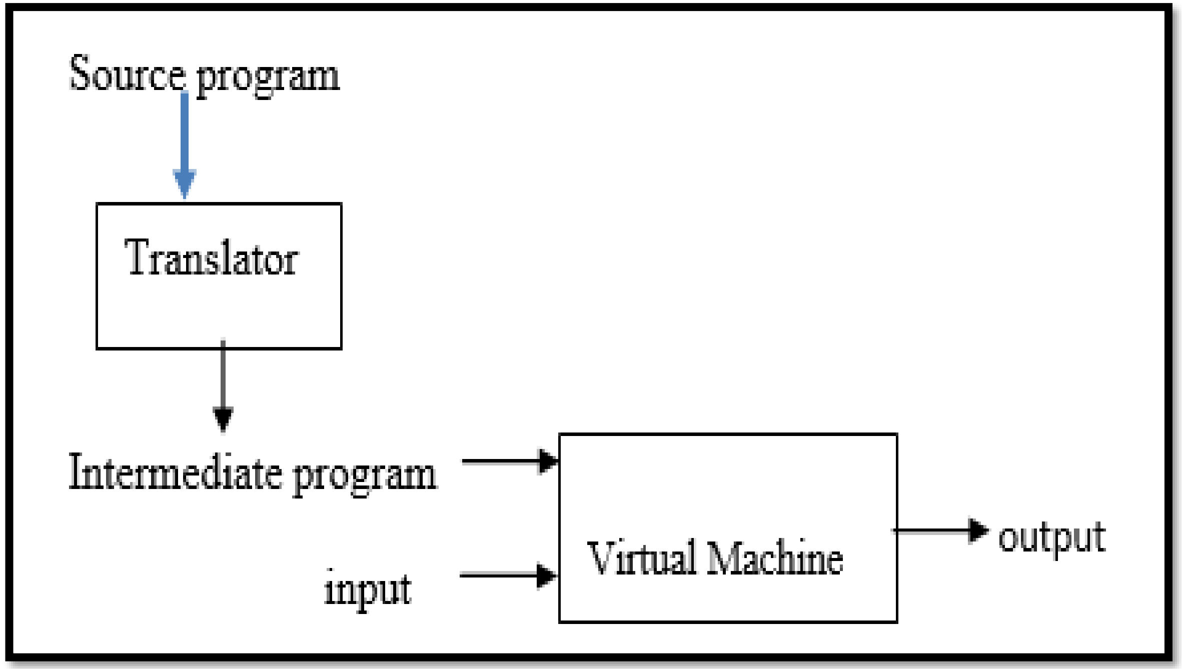


Interpreter:

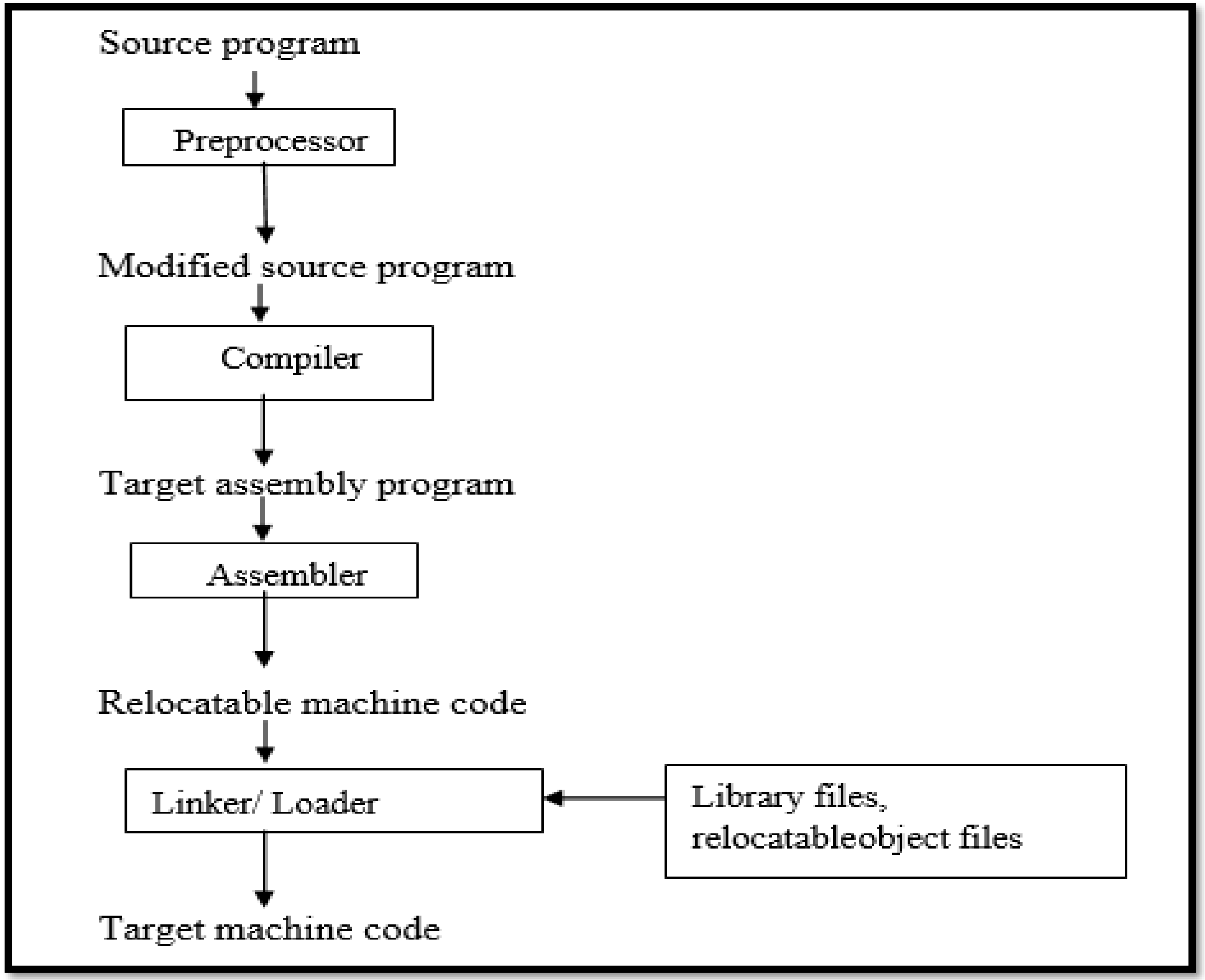
An *interpreter* is another common kind of language processor. Instead of producing a target program as a translation, an interpreter appears to directly execute the operations specified in the source program on inputs supplied by the user



Hybrid Compiler:



Language Processor:

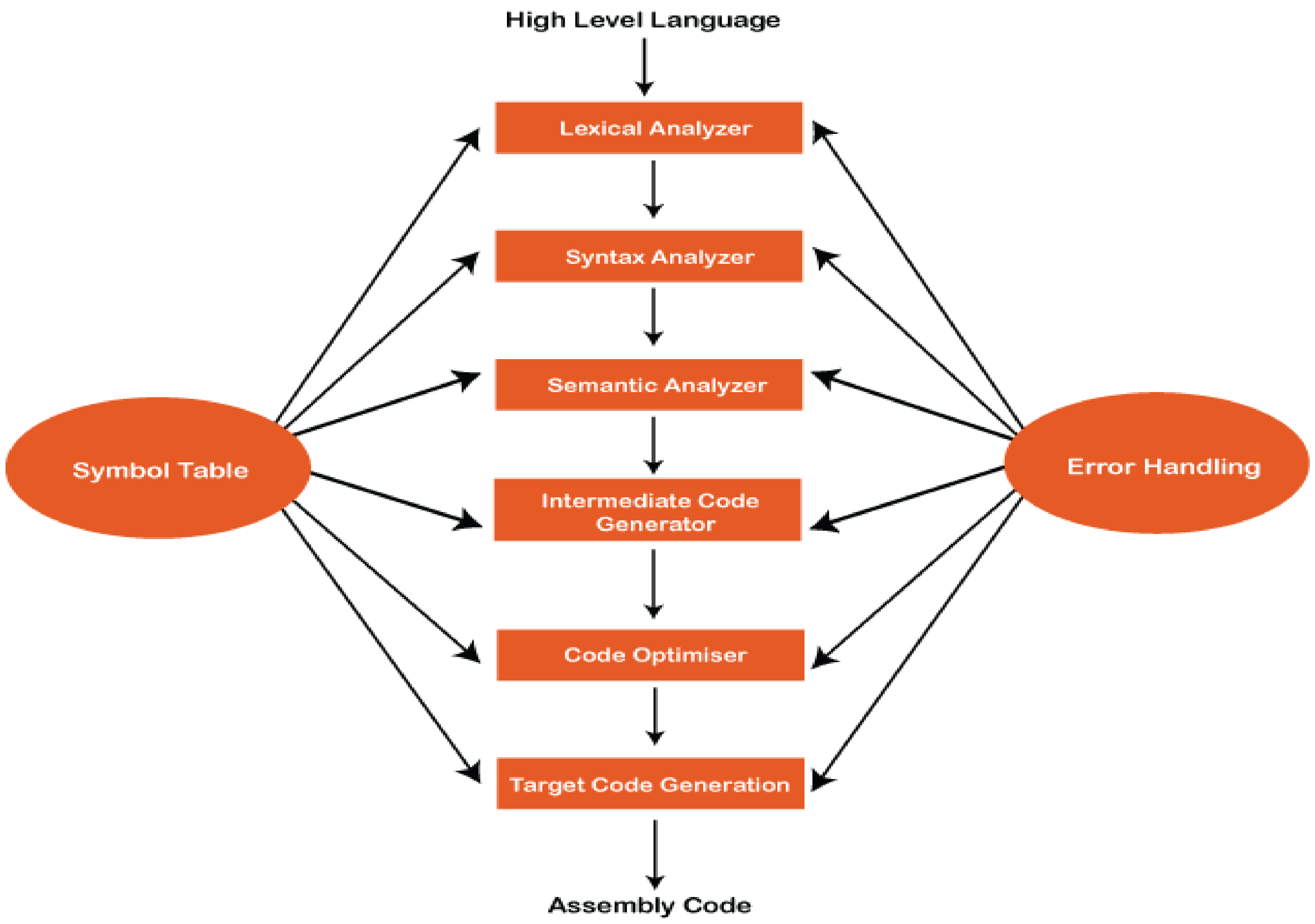


The structure of a compiler

A compiler as a single box that maps a source program into a semantically equivalent target program. There are two parts to this mapping: **analysis and synthesis.**

Analysis part: Breaks up the source program into constituent pieces and imposes a grammatical structure on them. It then uses this structure to create an intermediate representation of the source program.

Synthesis part: constructs the desired target program from the intermediate representation and the information in the symbol table. The analysis part is often called the front end of the compiler; the synthesis part is the back end.


High Level Language
Lexical Analyzer
Syntax Analyzer
Semantic Analyzer
Intermediate Code Generator
Code Optimiser
Target Code Generation
Assembly Code
Symbol Table
Error Handling

Phases of Compiler

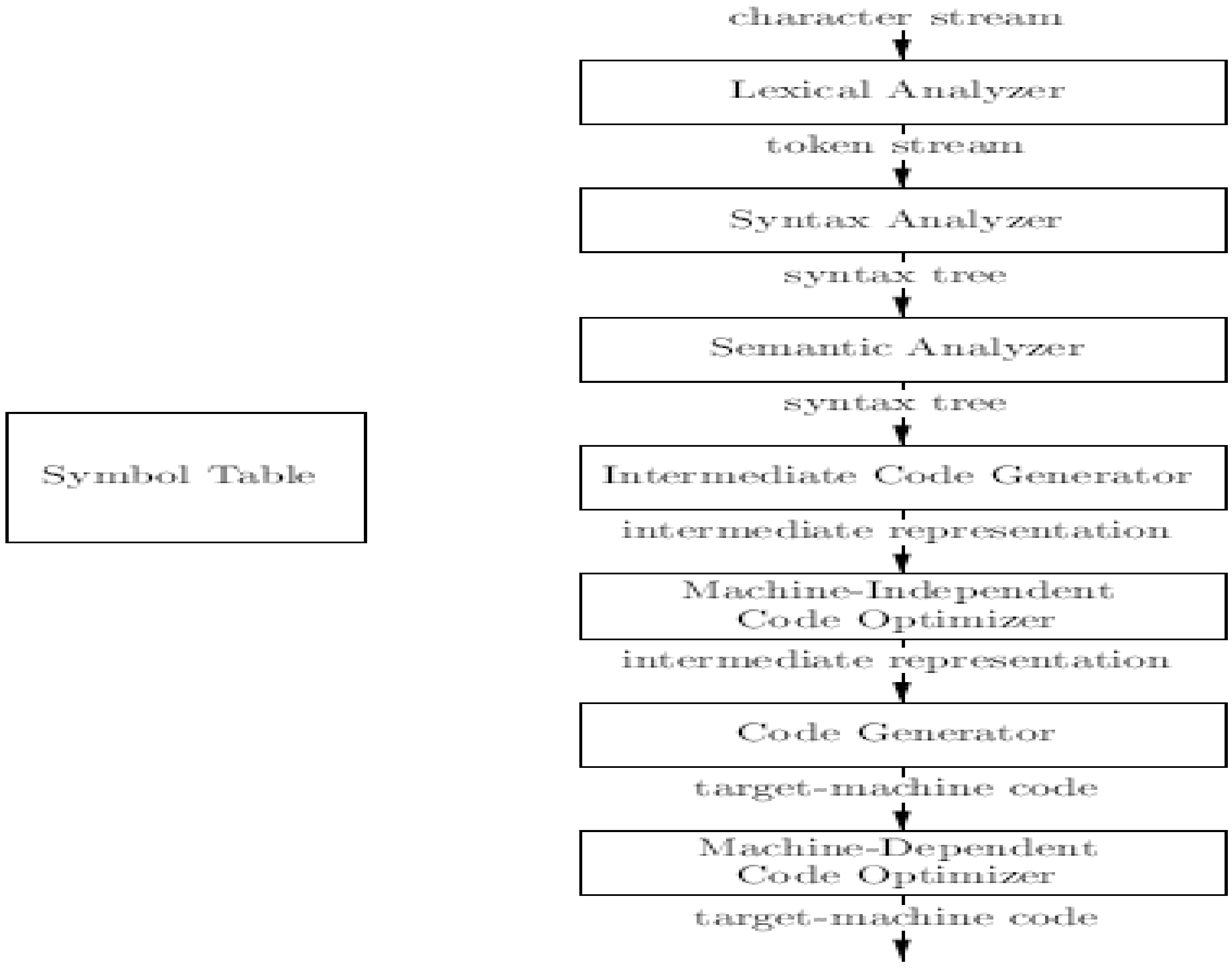


Figure 1.6: Phases of a compiler

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.
- **Token:** Token is a sequence of characters that represent lexical unit, which matches with the pattern, such as keywords, operators, identifiers etc.
- **Lexeme:** Lexeme is instance of a token i.e., group of characters forming a token.
- **Pattern:** Pattern describes the rule that the lexemes of a token take. It is the structure that must be matched by strings.
- Once a token is generated the corresponding entry is made in the symbol table.

Input: *stream of characters*

Output:*Token*

Token Template: <token-name, attribute-value>(eg.) c=a+b*5;

Lexemes and tokens

Lexemes	Tokens
C	Identifier
=	assignment symbol
A	Identifier
+	+ (addition symbol)
B	Identifier
*	* (multiplication symbol)
5	5 (number)

Hence, <id, 1><=>< id, 2>< +><id, 3 >< * >< 5>

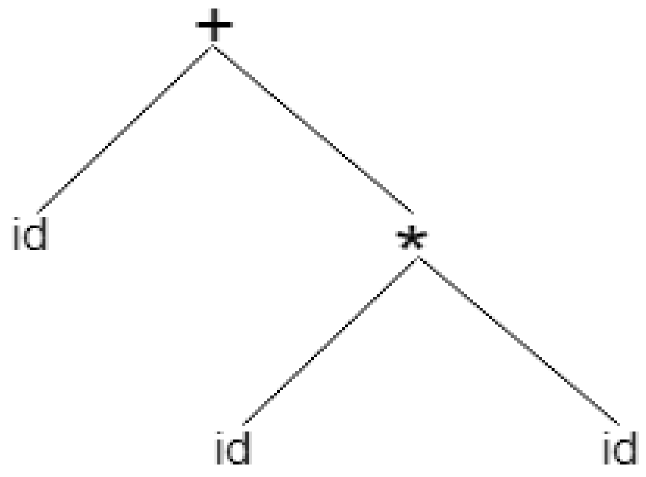
Syntax Analysis

- 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.

Input: *Tokens*

Output: *Syntax tree*

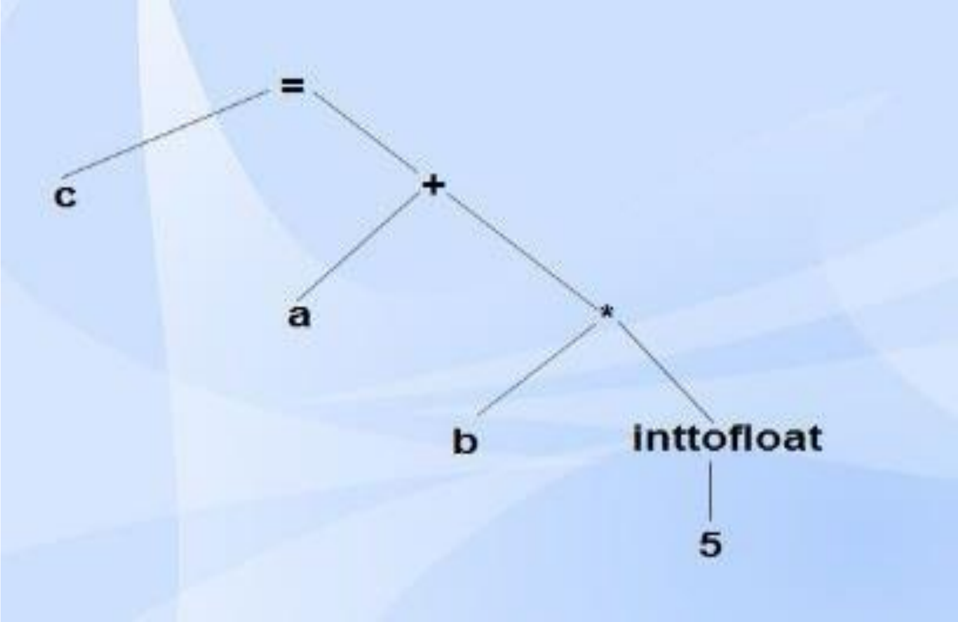
Example: id + id * id



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

Code Optimization

- Code optimization phase gets the intermediate code as input and produces optimized intermediate code as output.
- It results in faster running machine code.
- It can be done by reducing the number of lines of code for a program.
- This phase reduces the redundant code and attempts to improve the intermediate code so that faster- running machine code will result.
- During the code optimization, the result of the program is not affected.
- To improve the code generation, the optimization involves deduction and removal of dead code (unreachable code).

Code Generation

Code generation is the final phase of a compiler.

- It gets input from code optimization phase and produces the target code or object code as result.
- Intermediate instructions are translated into a sequence of machine instructions that perform the same task.
- The code generation involves
  - Allocation of register and memory.
  - Generation of correct references.
  - Generation of correct datatypes .
  - Generation of missing code.

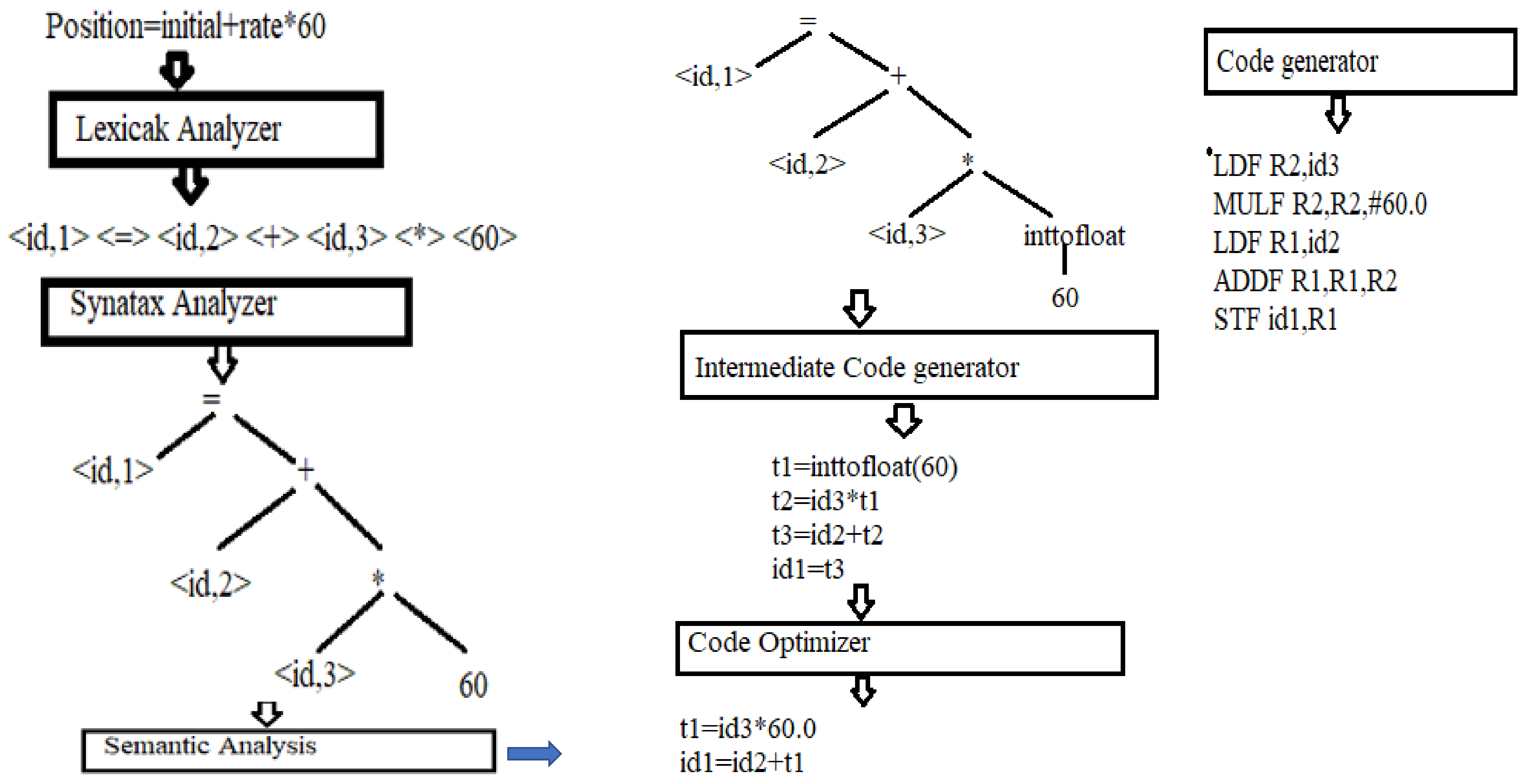
Symbol Table Management

- 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.
- If an identifier is detected in any of the phases, it is stored in the symbol table.

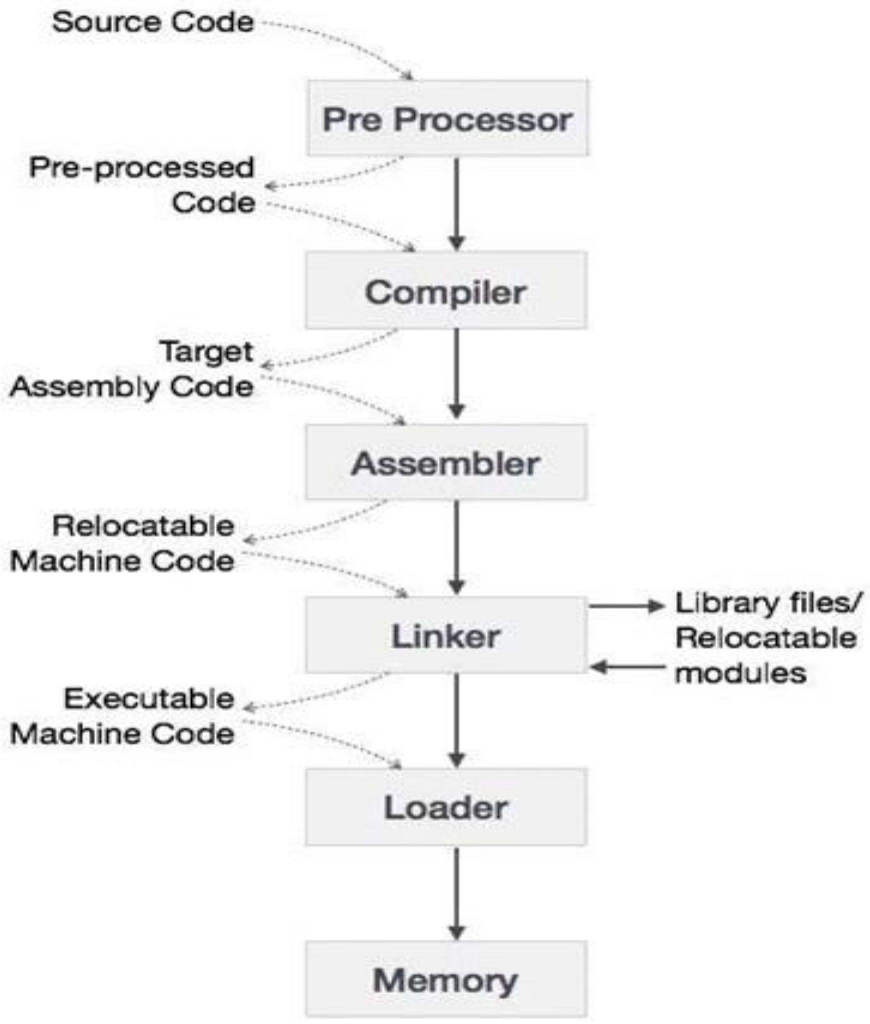
Example :

int a, b; float c; char z;

Symbol name	Type	Address
a	Int	1000
b	Int	1002
c	Float	1004
z	Char	1008


Position=initial+rate*60
Lexicak Analyzer
<id,1> <=> <id,2> <+> <id,3> <*> <60>
Synatax Analyzer
=
<id,1>
+
<id,2>
*
<id,3>
60
Semantic Analysis
=
<id,1>
+
<id,2>
*
<id,3>
inttofloat
60
Intermediate Code generator
t1=inttofloat(60)
t2=id3*t1
t3=id2+t2
id1=t3
Code Optimizer
t1=id3*60.0
id1=id2+t1
Code generator
LDF R2,id3
MULF R2,R2,#60.0
LDF R1,id2
ADDF R1,R1,R2
STF id1,R1

Cousins of a Compiler



Regular expressions

- Regular expression is a Meta language, use to describe the particular pattern of interest.
- Ex: - letter_ (letter_|digit)*

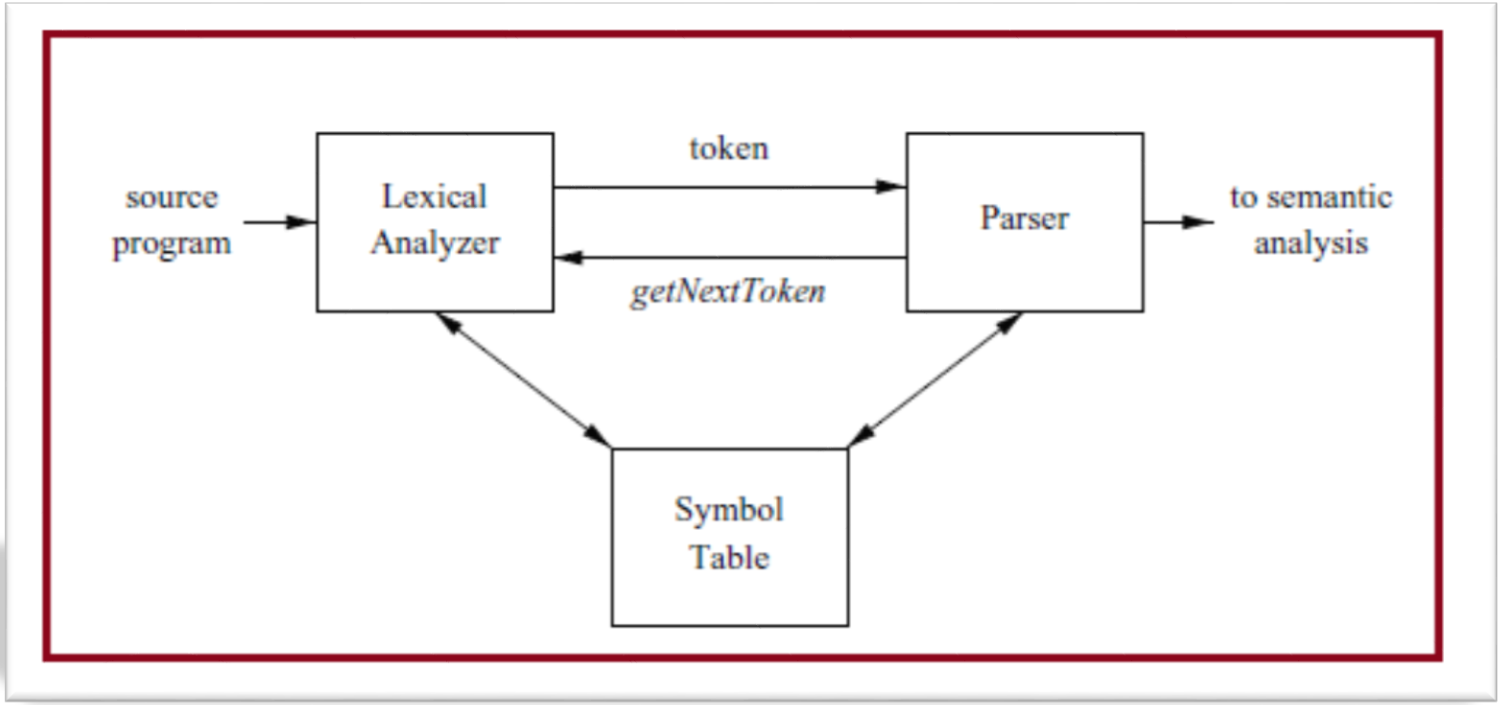
Algebraic laws for regular expression

LAW	DESCRIPTION
r|s = s|r	| is commutative
r| (s|t) = (r|s) |t	| is associative
r (st) = (rs) t	concatenation is associative
r(s|t) = rs |rt ; (s|t)r = sr|tr	concatenation distributers over |εr
= rε = r	ε is the identity for concatenation.
r*= (r|ε)*	ε is the guaranteed in a closure.
r** = r*	* is idempotent.

Lexical Analysis

Lexical Analysis is the first phase of the compiler also known as a scanner.

It converts the High level input program into a sequence of **Tokens**.



Lexical analyzers are divided into a cascade of two processes:

- Scanning consists of the simple processes that do not require tokenization of the input, such as deletion of comments and compaction of consecutive whitespace characters into one.
- Lexical analysis proper is the more complex portion, which produces tokens from the output of the scanner.

Lexical Analysis versus Parsing:

There are a number of reasons why the analysis portion of a compiler is normally separated into lexical analysis and parsing (syntax analysis) phases.

- Simplicity of design.
- Compiler efficiency is improved. A separate lexical analyzer allows us to apply specialized techniques that serve only the lexical task, not the job of parsing.
- Compiler portability is enhanced.

What is a token?

A lexical token is a sequence of characters that can be treated as a unit in the grammar of the programming languages.

Example of tokens:

- Type token (id, number, real, . . .)
- Punctuation tokens (IF, void, return, . . .)
- Alphabetic tokens (keywords)

A **pattern** is a description of the form that the lexemes of a token.

A **lexeme** is a sequence of characters in the source program that matches the pattern for a token and is identified by the lexical analyzer as an instance of that token.

Token	**Informal description**	**Sample lexems**
if	Characters i, f	**If**
Else	Characters e, l, s, e	**else**
Comparison	< or > or <= or >= or == or !=	**<=, !=**
id	Letter followed by letters and digits	**Pi, score, D2**
number	Any numeric constant	**3.14159, 0, 6.02e23**
literal	**Anything but “, surrounded by” ’s**	**“core dumped”**

Examples of Tokens

Exercise 1:

Count number of tokens :

```
int main()
{
        int a = 26, b = 21;
        printf(“Diff is :%d",a-b);
         return 0;
}
```

Answer: Total number of token:

Exercise 2:

Count number of tokens :

```
int min(int i);
```

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

Attributes for Tokens:

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

Example: The token names and associated attribute values for the Fortran 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>

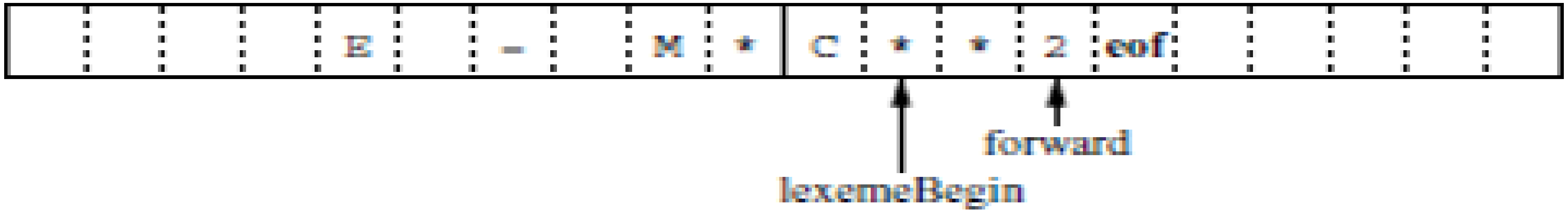
<exp op>

<number, integer value 2>

Input Buffering:

- Scanner performance is crucial:
  - This is the only part of the compiler that examines the entire input program one character at a time.
  - Disk input can be slow.
  - The scanner accounts for ~25-30% of total compile time.
- We need lookahead to determine when a match has been found.
- Scanners use *double-buffering* to minimize the overheads associated with this.

Buffer Pairs



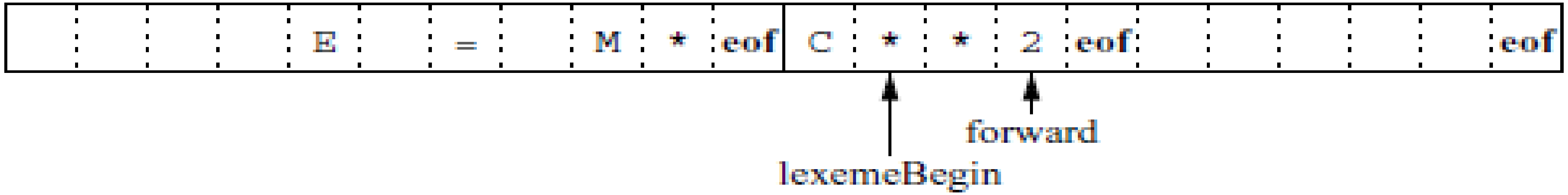
- Use two *N*-byte buffers (*N* = size of a disk block; typically, *N* = 1024 or 4096).
- Read *N* bytes into one half of the buffer each time. If input has less than *N* bytes, put a special EOF marker in the buffer.
- When one buffer has been processed, read *N* bytes into the other buffer ("*circular buffers*").

Two pointers to the input are maintained:

1. Pointer lexemeBegin, marks the beginning of the current lexeme, whose extent we are attempting to determine.
2. Pointer forward scans ahead until a pattern match is found;

Sentinels:

The **sentinel** is a special character that cannot be part of the source program, and a natural choice is the character eof.



Sentinels:

```
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 */
        break;
    cases for other characters
}
```

Specification of Tokens:

Regular expressions are an important notation for specifying lexeme patterns.

Strings and Languages:

- 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 symbol drawn from that alphabet.

- The length of the string S, **|S|** is the number of occurrences of symbols in S.
- Ex: banana is a string of length six.
- The empty string denoted ε, is the string of length zero.

Terms for Parts of Strings

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 example, ban, banana, and Ɛ 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 example, nana, banana, and Ɛ are suffixes of banana.
3. A **substrin**g of s is obtained by deleting any prefix and any suffix from s. For instance, banana, nan, and Ɛ are substrings of banana.

Regular Definitions:

If $\sum$ is an alphabet of basic symbols then a regular definition is a

sequence of the form $d_1 \rightarrow r_1$

$d_2 \rightarrow r_2$

………..

$d_n \rightarrow r_n$ where

Each d_i is a new symbol not in $\sum$ note the same as any other of the d's .

1. Each r_i is a regular expression over the alphabet $\sum U \{ d_{1,}\ d_{2,}\ d_{3..} d_{i-1} \}$ by restricting r_i to $\sum$ the previously defined d's .

Example-1:

C identifiers are strings of letters, digits, and underscores. Here is a regular definition for the language of C identifiers.

letter_ → A | B | …… | Z | a | b | …… | z | _

digit → 0 | 1 | …… | 9

id → letter_ (letter_ | digit)*

Example-2:

Unsigned numbers (integer or floating point) are strings such as 5280, 0.01234, 6.336E4, or 1.89E-4.

The regular definition

6.336E4

digit → 0 | 1 | …… |9

digits -> digit digit*

optionalFraction→ . digits | Ɛ

optionalExponent→ (E(+ | - | Ɛ) digits) | Ɛ

Number→digits optionalFraction optionalExponent

Extensions of Regular Expressions

1. One or more instances
2. Zero or one instance.
3. Character classes.

Example-1:letter_ → A | B | …… | Z | a | b | …… | z | _

digit → 0 | 1 | …… | 9

id →letter_ (letter_ | digit)*

Rewritten as

letter_ → [A-Za-z_]

digit→ [0-9]

id→ letter_ (letter_ | digit)*

Recognition of Tokens

A grammar for of branching statements and conditional expressions.

stmt → if expr then stmt

| if expr then stmt else stmt

|Ɛ

expr → term relop term

| term

term → id

| number

Example : Patterns for Tokens in the grammar

digit → [0-9]

digits -> digit+

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

letter → [A-Za-z]

id → letter (letter | digit)*

if → if

then → then

else → else

relop → < | > | <= | >= | = |<>

Tokens, their patterns, and attribute values

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

Example

C=a+b*5

<id, pointer to symbol table entry>

<relop, EQ>

<id, pointer to symbol table entry>

<assign-op,- >

<id, pointer to symbol table entry>

<multi_op - >

<num, pointer to symbol table entry>

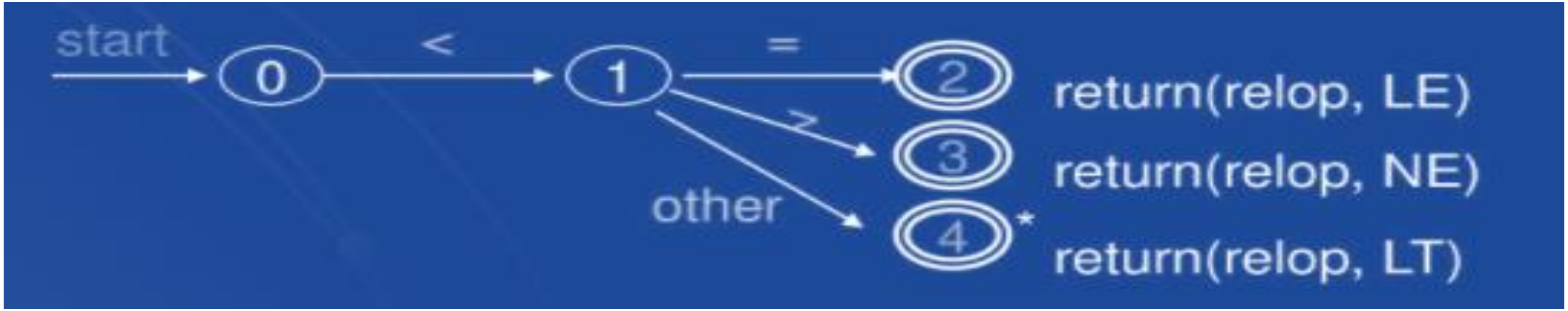
Transition Diagrams:

- The construction of a lexical analyzer first convert patterns into stylized flowcharts, called “**transition diagrams**.“
- **Nodes:** States, 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 of the transition diagram to another

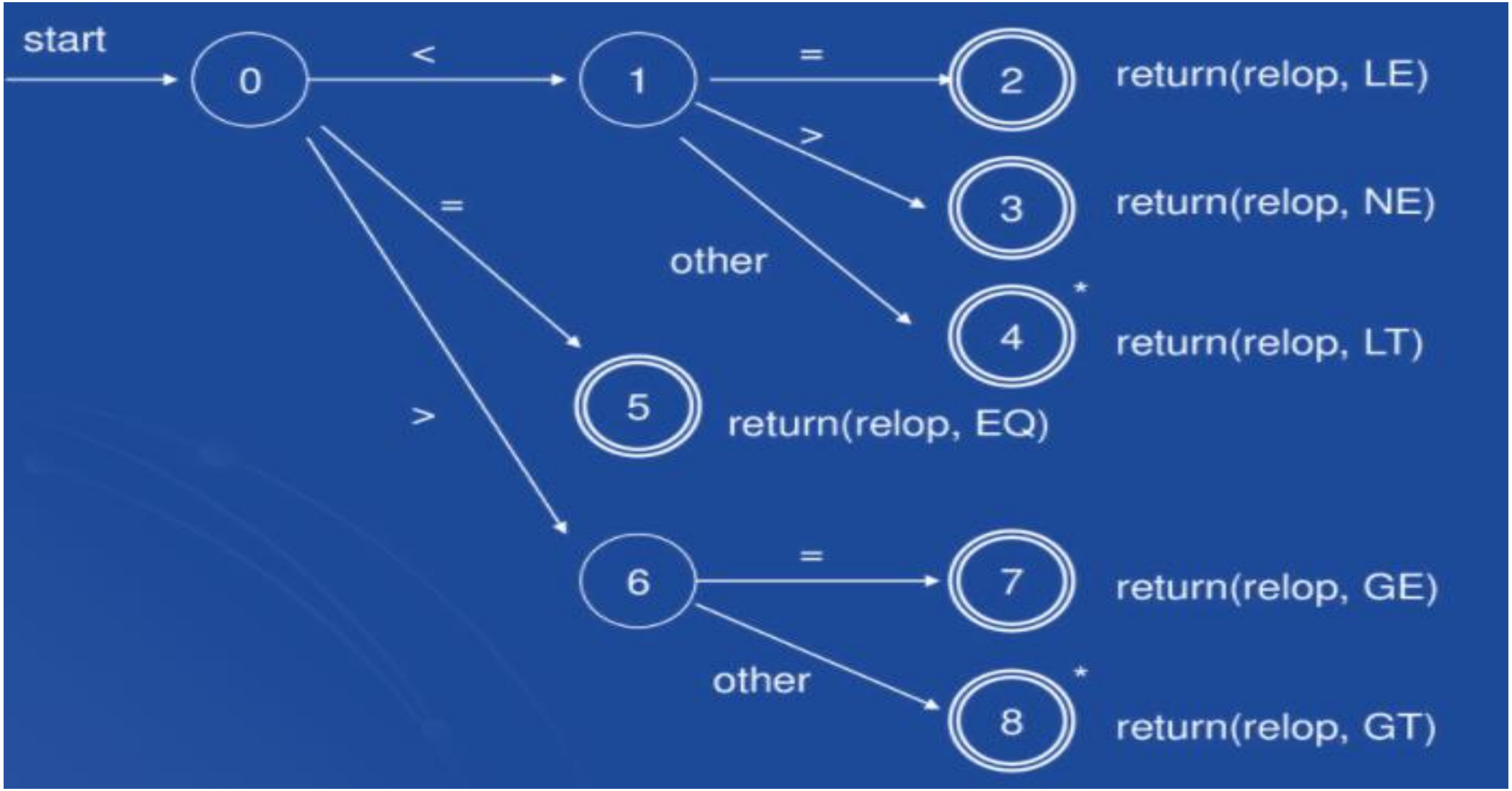
  →Labeled by a symbol or set of symbols.
- **Deterministic:** There is never more than one edge out of a given state with a given symbol among its labels.
- Certain states are said to be accepting, or final.

➢ It is necessary to retract the forward pointer one position. then we shall additionally place a * near that accepting state.

➢ Start state, or initial state; it is indicated by an edge, labeled ”start“, entering from nowhere.



Example Transition diagram for relop



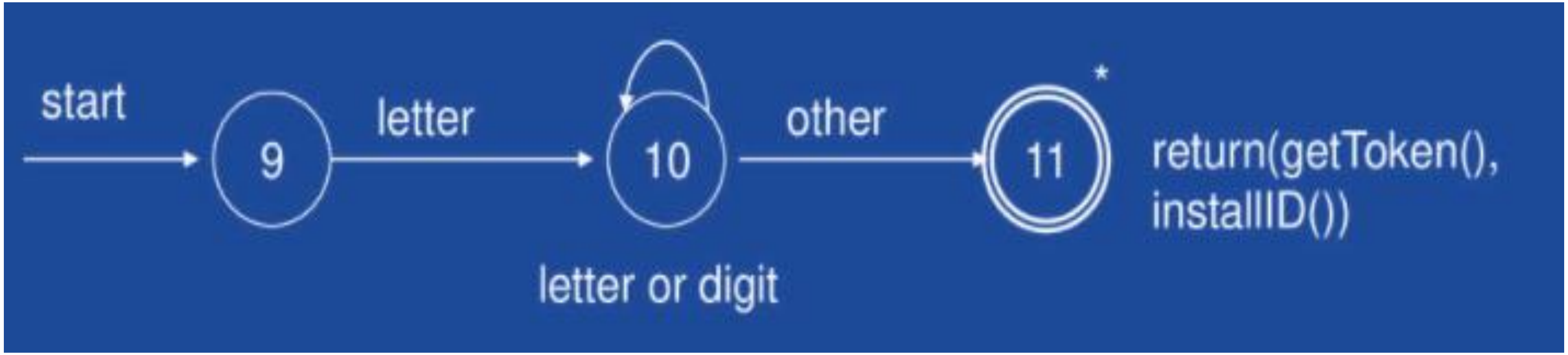
Recognition of Reserved Words and Identifiers

Problem: Key words look like identifiers.

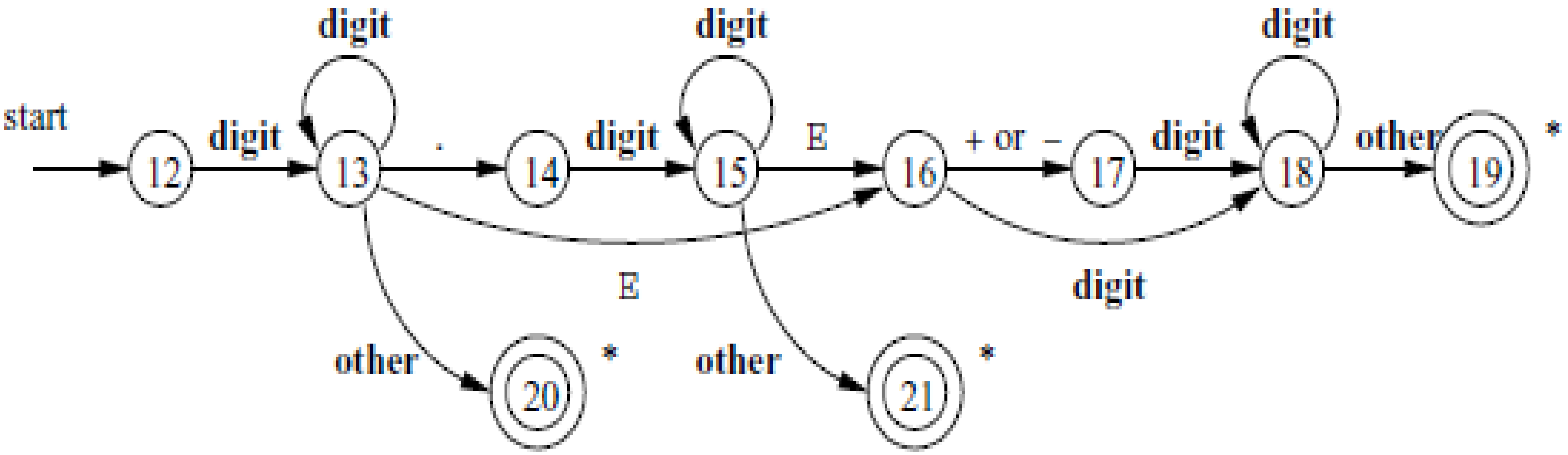
Solution:

- Install the reserved words in the symbol table initially
- Create separate transition diagrams for each keyword

Example for Identifiers and Keywords



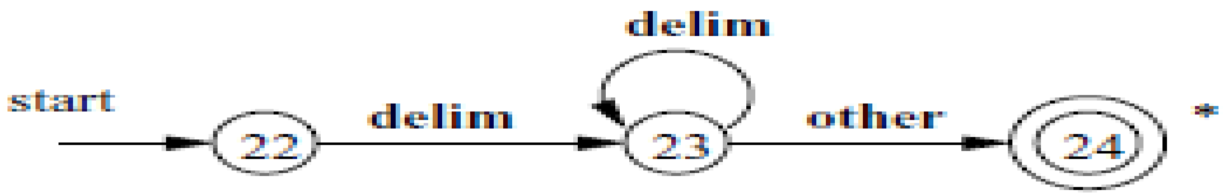
Transition diagram for unsigned numbers:



31.4

A transition diagram for unsigned numbers

Transition diagram for whitespace:



A transition diagram for whitespace