Compiler design (2)

Lexical Analyzer :- The lexical analyzer is the f
phase of a compiler. Its main task is to read the
input characters and produce as output a seque
of tokens that the parser uses for syntax anal

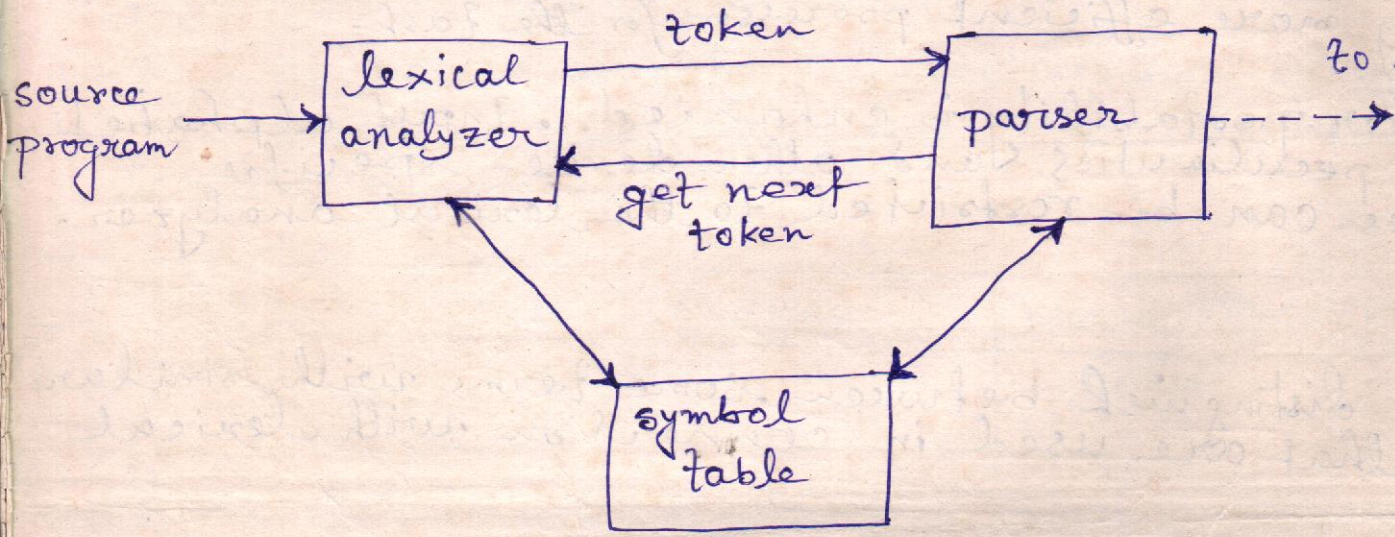


Fig. 1 :- Interaction of lexical analyzer with parse

The interaction, summarized schematically in Fig. 1,
commonly implemented by making the lexical analy
be a subroutine or a coroutine of the parser.
receiving a "get next token" command from the
parser, the lexical analyzer reads input chara
until it can identify the next token.

Issues in Lexical Analysis :- (1) Simpler design is perhaps the most important consideration. The separation of lexical analysis from syntax analysis often allows us to simplify one or the other of these phase

(2) Compiler efficiency is improved. A separate lexical analyzer allows us to construct a specialized and potentially more efficient processor for the task.

(3) Compiler portability is enhanced. Input alphabet peculiarities and other device-specific anomalies can be restricted to the lexical analyzer

We shall distinguish between some terms with similar meanings that are used in connection with lexical analysis.

* Lexemes :- A lexeme is a sequence of characters in the source program that is matched by the pattern for a token. For example, in the Pascal statement,

```
const pi = 3.1416 ;
```

the substring pi is a lexeme for the token "identifier".

* Tokens :- Tokens are classes of similar lexemes such as identifier, constant, operator.

* Pattern :- Pattern is an informal or formal description of a token, such as an identifier is a string of at most 8 characters, in which the first character is an alphabet, and the successive characters are either digits or ~~characters~~ alphabe

A pattern serves two purposes :-

(i) It is a precise description or specification of tokens.
(ii) This description can then be used to generate a lexical analyzer.

TOKEN	SAMPLE LEXEMES	Informal Description of Patt
) const	const	const
) if	if	if
i) relation	<, <=, =, <>, >, >=	< or <= or = or <> or >=
') id	pi, count, D2	letter followed by letters digits
') num	3.1416, 0, 6.02E23	any numeric constan
i) literal	"core dumped"	any characters befu " and " except "

Example :- The tokens 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, >

<num, integer value 2>

Input Buffering :- There are three general approaches to the implementation of a lexical analyzer.

(1) Use a lexical-analyzer generator, such as Lex compiler to produce the lexical analyzer from a regular expression-based specification. In this case the generator provides routines for reading and buffering the input.

(2) Write the lexical analyzer in a conventional system-programming language, using the I/O facilities of that language to read the input.

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

(a) Buffer Pairs :-

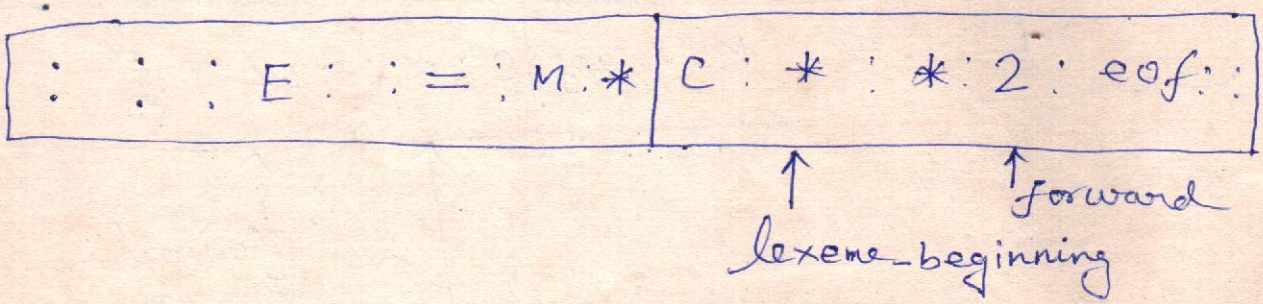


Fig 2 :- An input buffer in two halves.

```
if forward at end of first half then begin
        reload second half;
        forward := forward + 1
    end
    else if forward at end of second half then b
        reload first half;
        move forward to beginning of first half
    end
    else forward := forward + 1
```

Fig 3 :- Code to advance forward pointer

(b) Sentinels :-

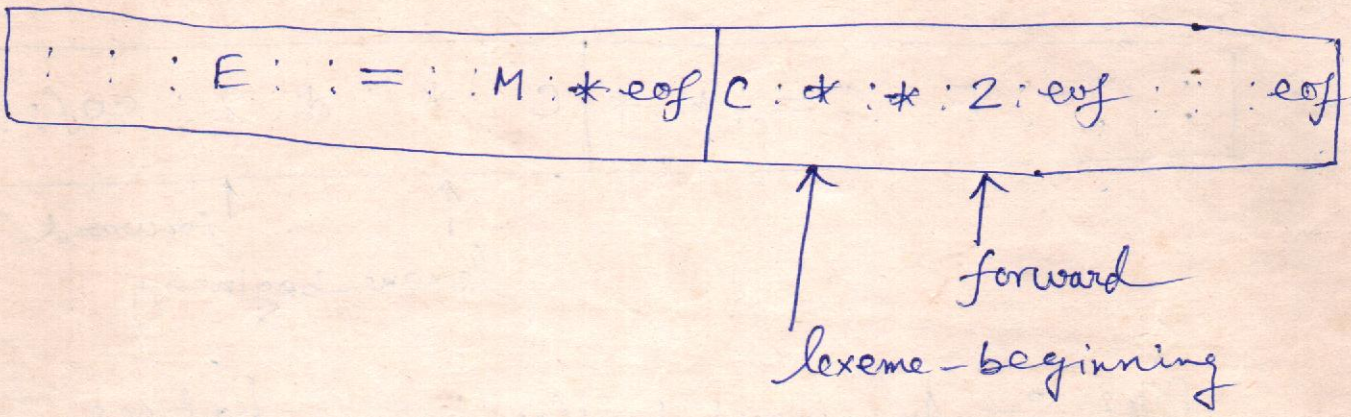


Fig 4 :- Sentinels at end of each buffer half.

```
forward := forward + 1 ;
if forward ↑ = eof then begin
    if forward at end of first half then begin
        reload second half;
        forward := forward + 1 ;
    end
else if forward at end of second half then begin
        reload first half;
        move forward to beginning of first half
    end
else /* eof within a buffer signifying end of in
        terminate lexical analyzer.
end.
```

Fig 5 :- Lookahead code with sentinels.

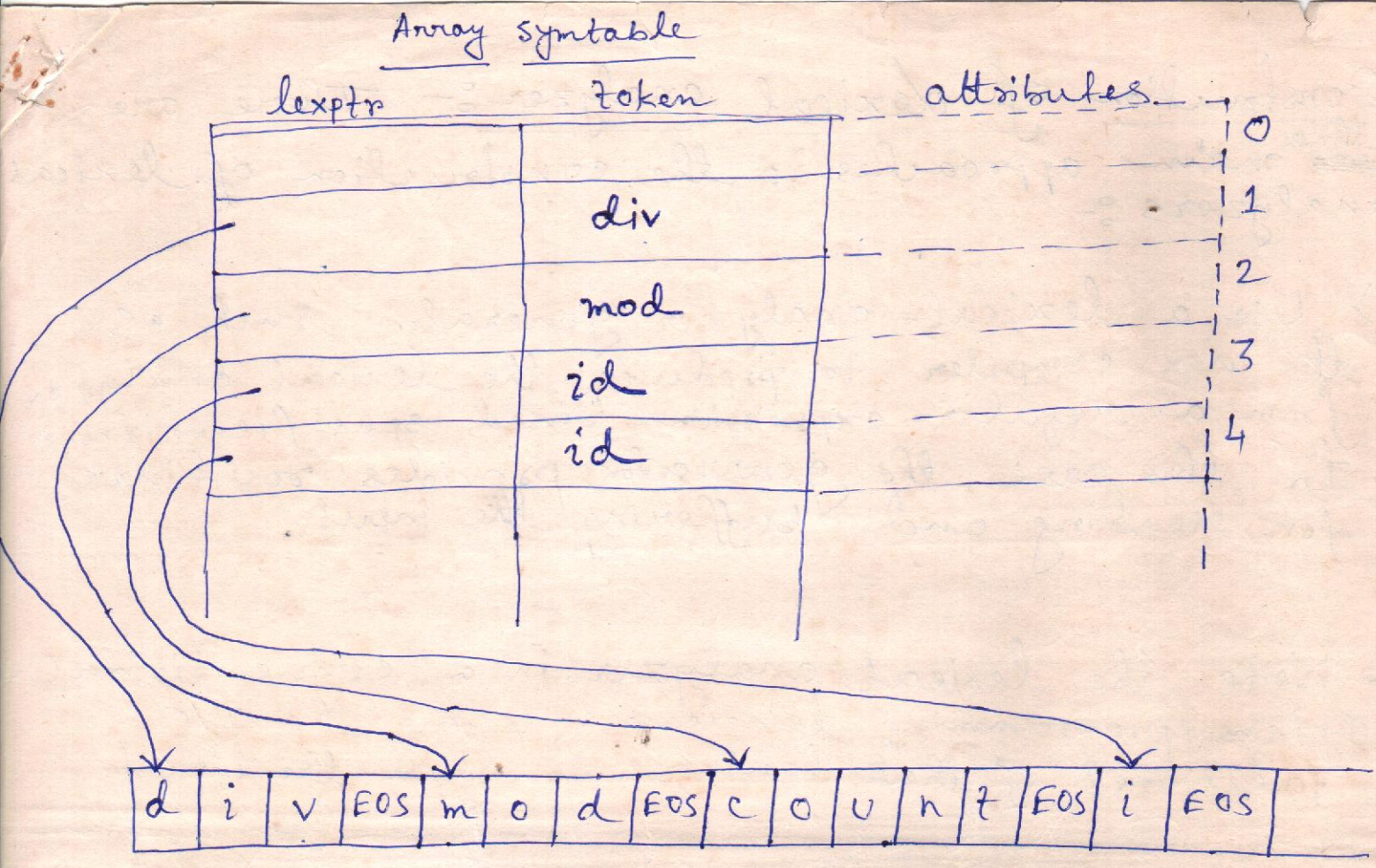


Fig 6 :- Symbol table and array for storing strings.

* lexemeBegin :- Pointer lexemeBegin, marks the beginning of the current lexeme, whose extent we are attempting to determine.

* forward :- Pointer forward scans ahead until a pattern match is found.

Construction of lexical analyzer :- There are three main approaches to the construction of lexical analyzers :-

1) Use a lexical-analyzer generator, such as the lex compiler to produce the lexical analyzer from a regular-expression-based specification. In this case, the generator provides routines for reading and buffering the input.

2) Write the lexical analyzer in a conventional systems-programming language, using the I/O facilities of that language to read the input.

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

Sometimes, lexical analyzers are divided into a cascade of two processes :-

(a) 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.

(b) Lexical analysis proper process is the more complex portion, where the scanner produces the sequence of tokens as output.

Lexical Analyzer :- Input Buffering :- Buffer ~~Pairs~~

Pairs :-

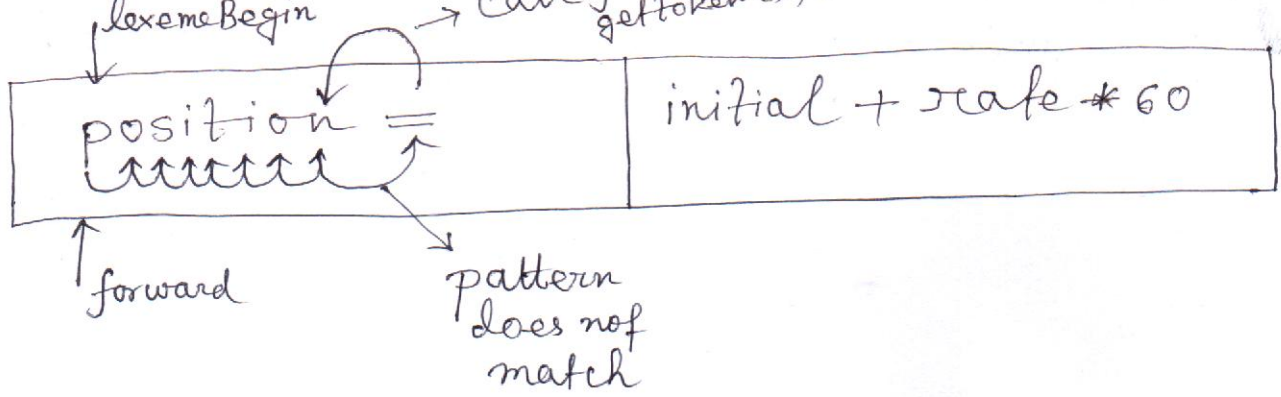


Fig.1 :- Using a pair of input buffers

* lexemeBegin :- Pointer lexemeBegin, marks the begining of the current lexeme, whose extent we are attempting to determine.

* forward :- Pointer forward, scans ahead until a pattern match is found.

* Note :- Install the reserved words in the symbol table initially. A field of the symbol table entry indicates that these strings are never ordinary identifiers, and tells which token they represent.

* installID() :- When we find an identifier, a call to installID places it in the symbol table if it is not already there and returns a pointer to the symbol table entry for the lexeme found. Of course, any identifier not in the symbol table during lexical analysis can not be a reserved word, so its token is id

* getToken() :- The function getToken examines the symbol table entry for the lexeme found, and returns whatever token name the

symbol table says this lexeme represents — either id or one of the keyword tokens that was initially installed in the table.

Recognition of Tokens :- Now, we must study how to take the patterns for all the needed tokens and build a piece of code that examines the input string and finds a prefix that is a lexeme matching one of the patterns.

stmt → if expr then stmt |
if expr then stmt else stmt | ∈

expr → term relop term | term

term → id | number

Fig. 1 :- A grammar for branching statements

The terminals of the grammar, which are if, then, else, relop, id, and number, are the names of tokens as far as the lexical analyzer is concerned

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

relop → < | > | <= | >= | = | <>

Fig. 2 :- Patterns for tokens.

In addition, we assign the lexical analyzer the job of stripping out white-space, by recognizing the "token" ws defined by :-

ws → (blank | tab | newline)+

LEXEMES	TOKEN VALUE	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

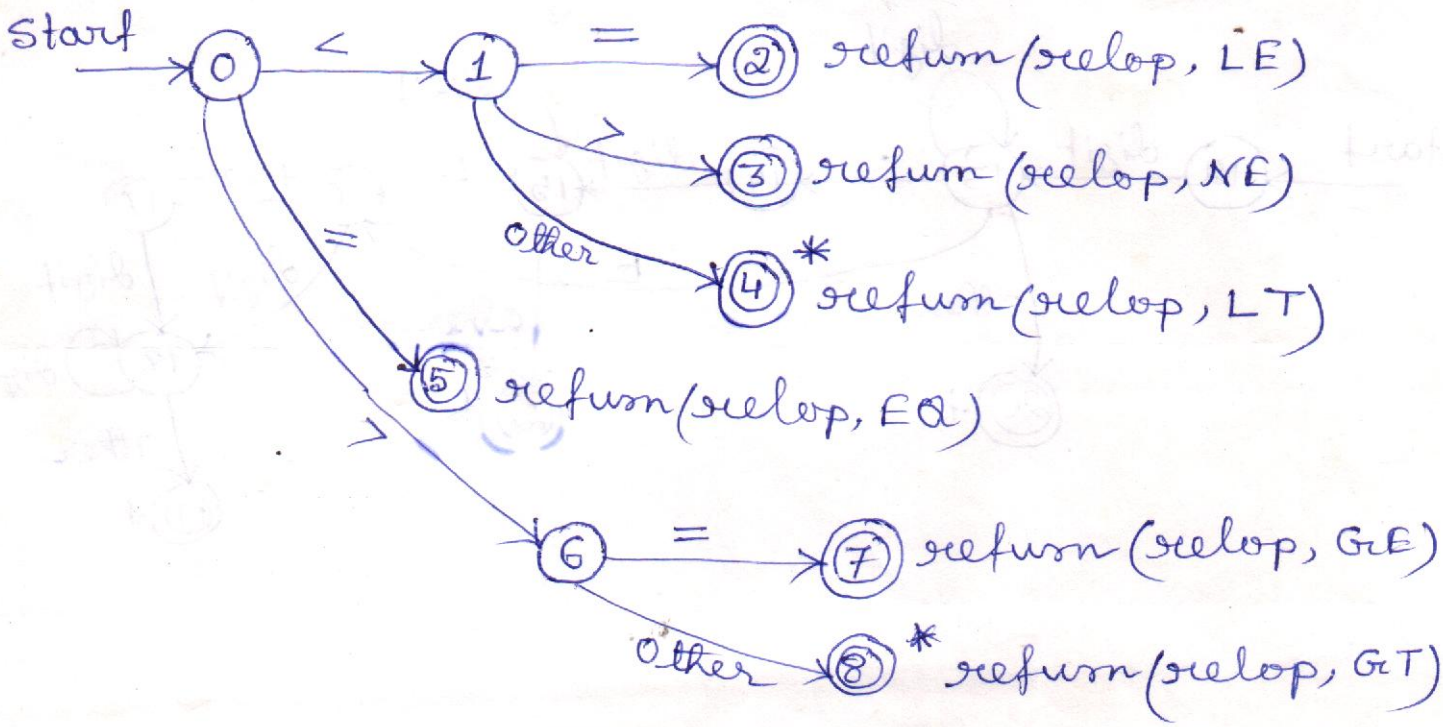
Fig3 :- Tokens, their patterns, and attribute values

Note :- Install the reserved words in the symbol table initially. A field of the symbol-table entry indicates that these strings are never ordinary identifiers, and tells which token they represent.

* installID() :- When we find an identifier, a call to installID places it in the symbol table if it is not already there and returns a pointer to the symbol-table entry for the lexeme found. Of course, any identifier not in the symbol table during lexical analysis cannot be a reserved word, so its token is id.

* getToken() :- The function getToken examines the symbol-table entry for the lexeme found, and returns whatever token name the symbol table says this lexeme represents — either id or one of the keyword tokens that was initially installed in the table.

Transition diagram for relop :-



Recognition of Reserved Words and Identifiers :-

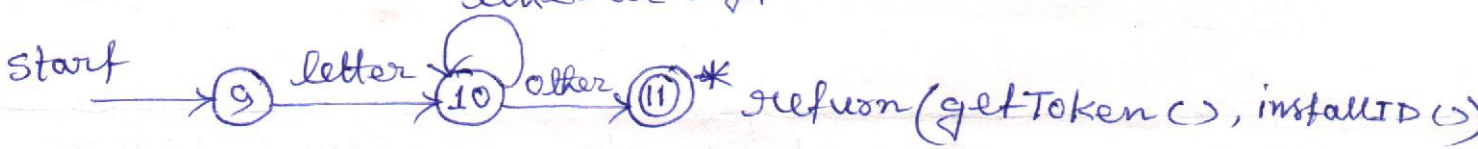
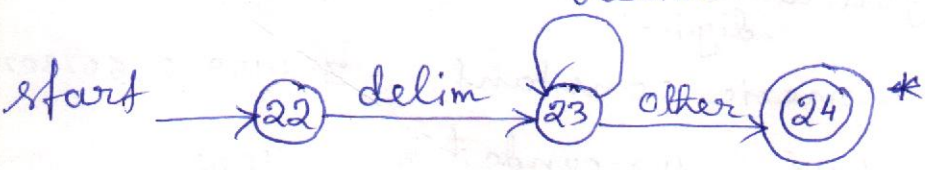
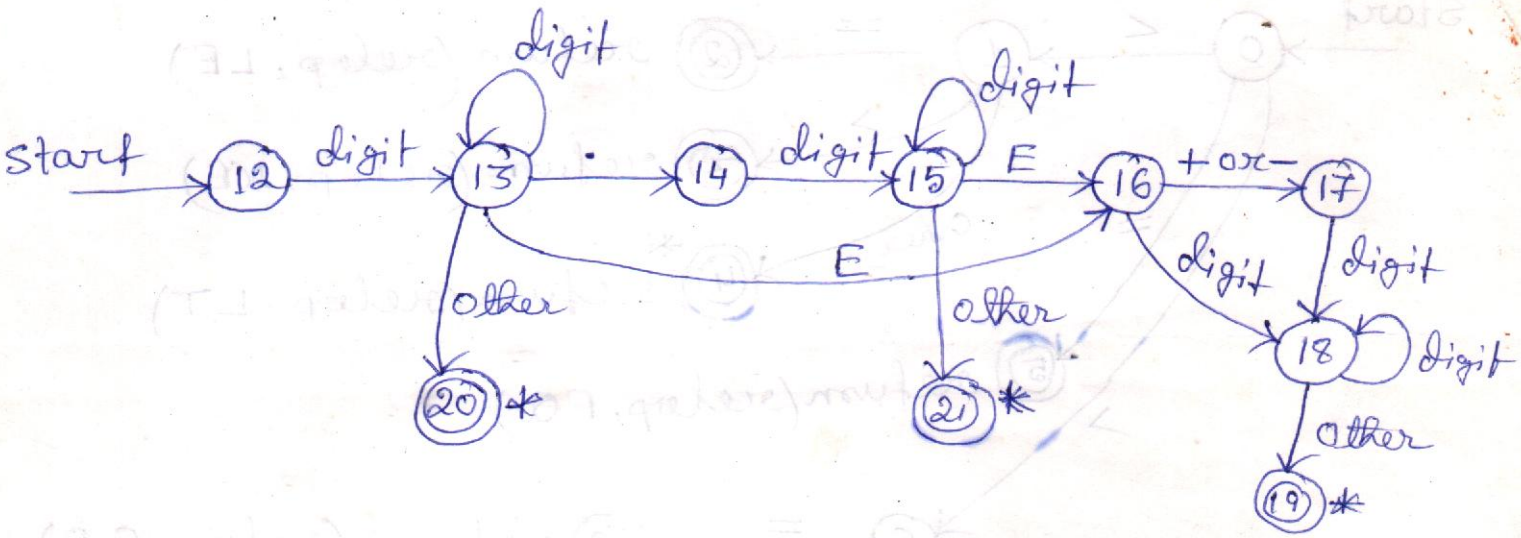


Fig. 4 :- A transition diagram for ids and keywords

A transition diagram for whitespace



A transition diagram for unsigned numbers



TOKEN	Informal Description	SAMPLE LEXEMES
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, 6.023e23
literal	anything but ", surrounded by "'s	"core dumped"