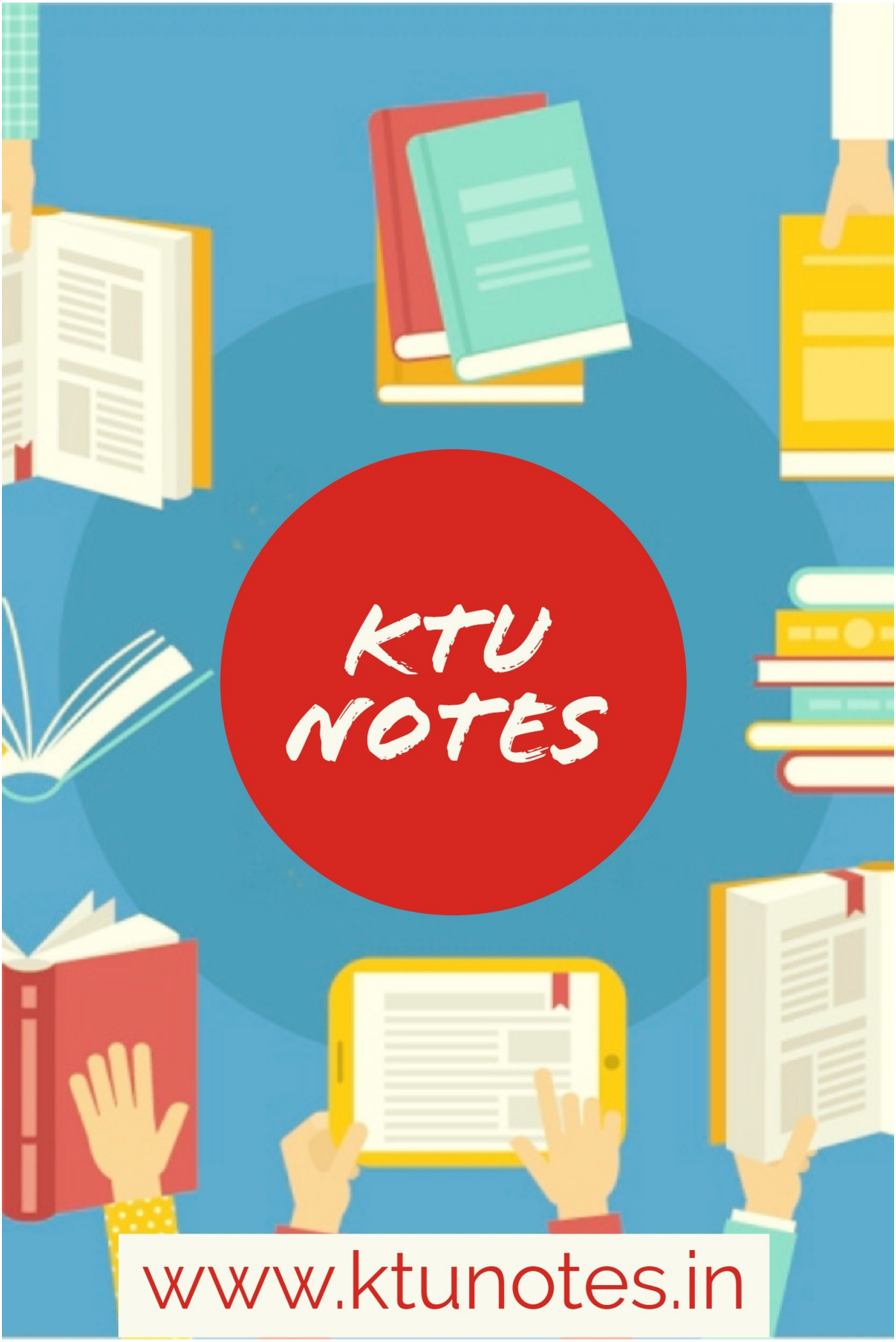

KTU
NOTES
www.ktunotes.in

Module 1

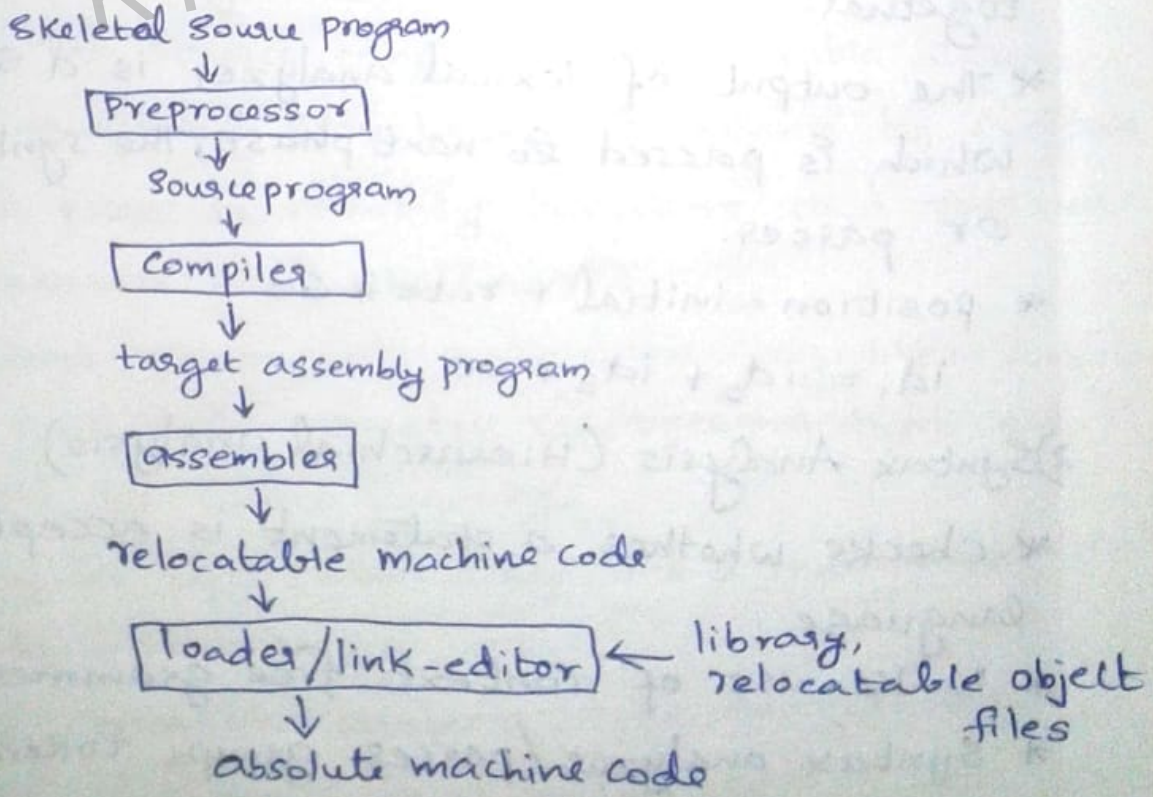
Introduction to compilers

Compiler is a program that reads source programming language and translate it to target language.

There are two parts of compilation: analysis & Synthesis

Analysis part breaks up the source program into constituent pieces and creates an intermediate representation of source program. Synthesis part constructs desired target program from the intermediate representation.

Actual sequence of program execution :-



Analysis phase ⇒ lexical analysis
Syntax analysis
Semantic analysis

Synthesis phase ⇒ Intermediate code generation
Code optimization
Code generation.

Analysis phase

1) Lexical Analysis (linear Analysis)

Tokens - Meaningful sequence of characters in source program. e.g: keywords, identifiers...

* Identifies whether given string or word is accepted in the language.
* Use regular expression for this.
* lexical analyzer / scanner seperate characters of the source language into groups that logically belong together.
* The output of lexical analyzer is a stream of tokens which is passed to next phase, the syntax analyzer or parser.
* position = initial + rate * 60

$id_1 = id_2 + id_3 * 60$

2) Syntax Analysis (Hierarchial analysis)

* checks whether a statement is acceptable in the language
* Make use of context free grammer.
* Syntax analyzer / parser groups tokens together into syntactic structures.

temp2 = id3 * temp1

temp3 = id2 + temp2

id1 = temp3

* primary difference b/w intermediate code & assembly code is that intermediate code need not specify registers to be used for each operation.

2) Code optimization

* Attempts to improve intermediate code so that the ultimate object program runs faster and takes less space.
* It is optional phase
* $id_1 = id_2 + id_3 * id_4$

If id_3 & id4 was multiplied beforehand in the program & if there is no modification for it, we can rewrite the above expression with solution of that particular expression.

temp1 = id3 * id4
id1 = id2 + temp1

} Reduce execution time.

* In the other case ($id_1 = id_2 + id_3 * 60$)

Into real operation can be eliminated if second step was rewritten as follows

temp2 = id3 * 60.0

Besides, 3rd step & 4th step can be combined as

id1 = id2 + temp2

* Compile time evaluation - Avoid repeated Computation of statement.

3) Code generation

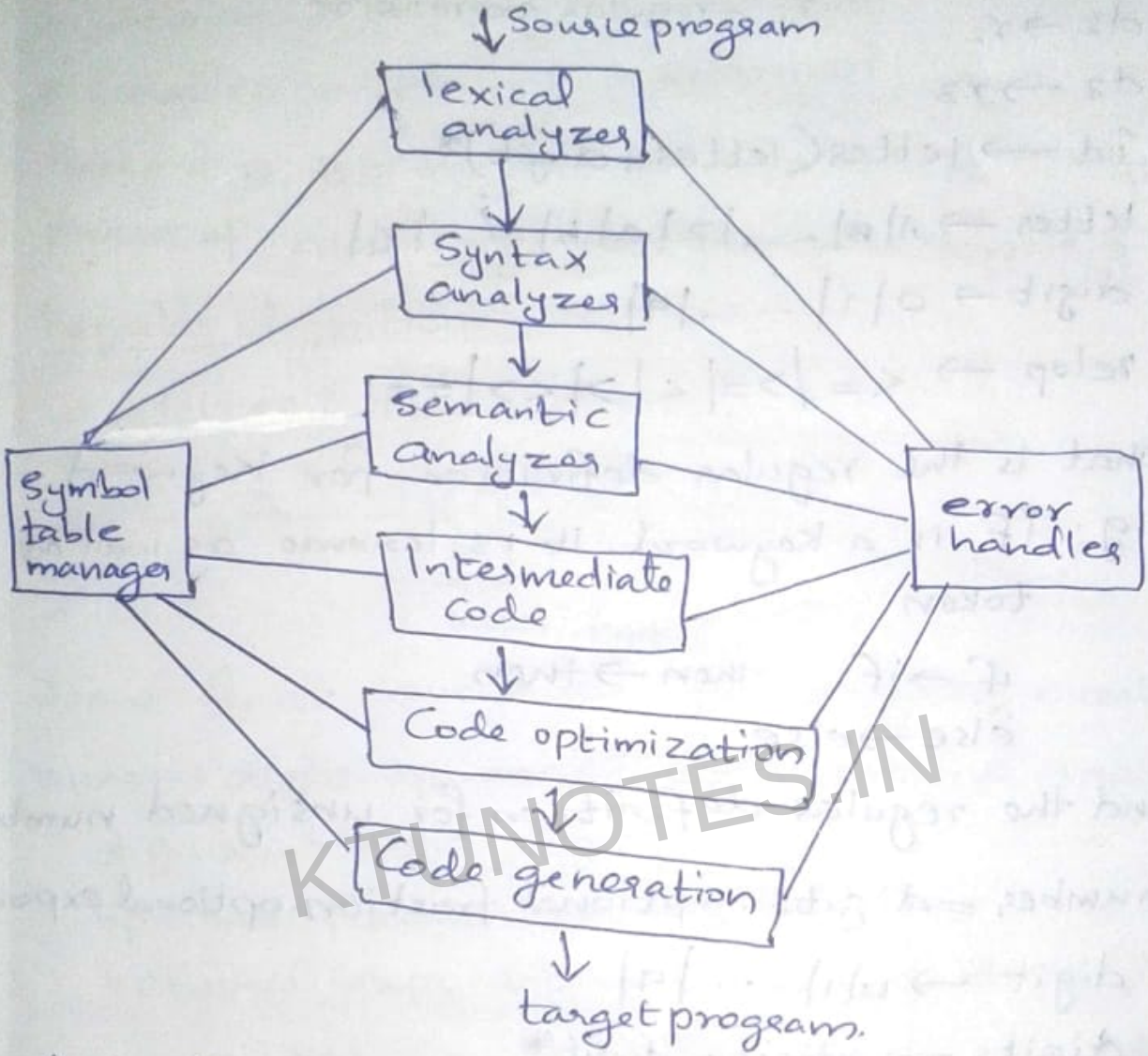
* generate target code is relocatable machine code or assembly code.
* Intermediate instructions are translated into sequence of machine instructions.
*
```
MOVF  id3, R1
MULF  #60.0, R1
MOVF  id2, R2
ADDF  R1, R2
MOVF  R2, id1
```

Table Management / Bookkeeping - portion of the compiler keeps track of the names used by the program & record essential information about each, such as its type (integer, real etc), name, value size etc. The data structure used to store this information is called symbol table. when an identifier in the source program is detected by lexical analyzer, identifier is entered into symbol table.

Error Detection & Reporting - Error handler is invoked when the flaw in source program is detected. Each phase can encounter errors. But a phase must deal with that error so that compilation can proceed, allowing further errors in the source program to be detected. Syntax and semantic analysis phases usually handle large

fraction of errors detectable by compiler.

Phases of a Compiler ⇒



Lexical Analysis

A class of meaningful sentences in a language is called Tokens. Lexeme is the instance of those classes. ie sequence of ilp characters that comprises a single token is called lexeme.

e.g: identifier id is a token

xyz, abc are instances or lexeme.

Patterns are the rules to represent token.

For identifier rule is letter (letter+digit)*

It is represented using regular definitions

$d_1 \rightarrow r_1$ $\quad d_i$ – distinct name

$d_2 \rightarrow r_2$ $\quad r_i$ – regular expression

$d_3 \rightarrow r_3$

id $\longrightarrow$ letter(letter+digit)*

letter $\rightarrow$ A|B|...|z|a|b|....|z|

digit $\rightarrow$ 0|1|....|9|

relop $\rightarrow$ <=|>=|<|>|<>|==

What is the regular definition for keyword

e.g: if is a keyword. It is lexeme as well as token

if $\rightarrow$ if $\quad$ then $\rightarrow$ then

else $\rightarrow$ else

? find the regular definition for unsigned number?

number = digits . optional fraction . optional exponent

digit $\rightarrow$ 0|1|...|9|

digits $\longrightarrow$ digit . digit*

optional fraction $\rightarrow$. digits / ϵ

optional exponent $\rightarrow$ E (-/+/ϵ) digits / ϵ

Role of Lexical Analyzer $\Rightarrow$

* first phase of compiler
* divided into a cascade of 2 phases
  - Scanning [scanner is responsible for doing simple tasks]
  - lexical analysis (lexical analyzer perform complex operations).

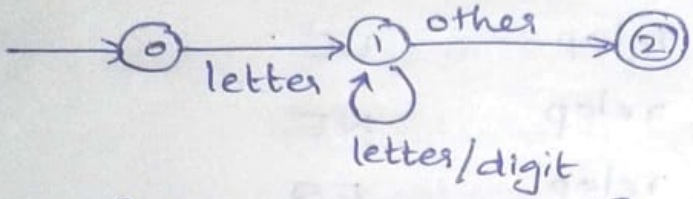
Issues in lexical analysis ⇒

1. Simpler design is most important consideration
2. Compiler efficiency is improved
3. Compiler portability is enhanced.

These are the reasons for seperating analysis phase of compiling into lexical analysis & parsing.

Regular definitions

id = letter (letter + digit)*



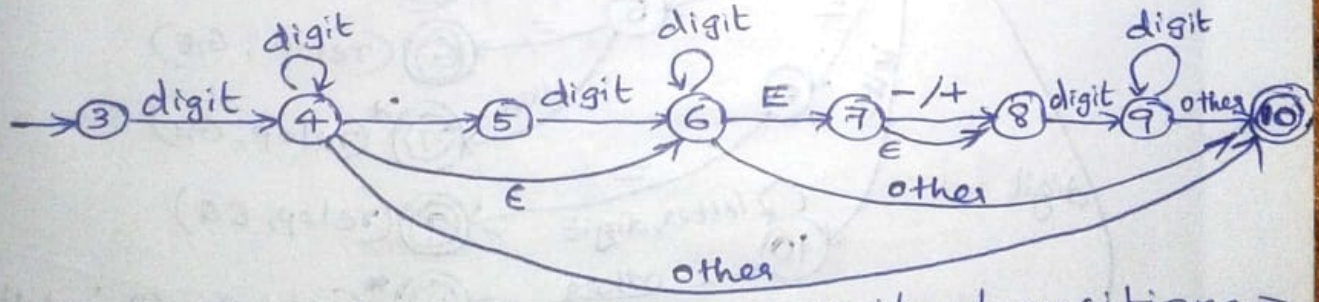
? Draw finite automata for unsigned number.

num → digits. optional fraction. optional exponent

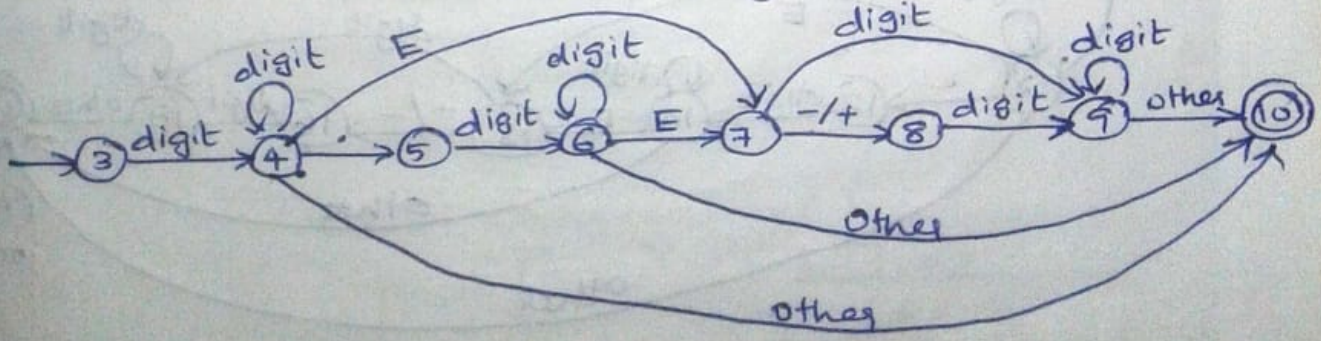
digits → digit digit*

optional fraction → . digits/ϵ

optional exponent → E (-/+/ϵ) digits/ϵ

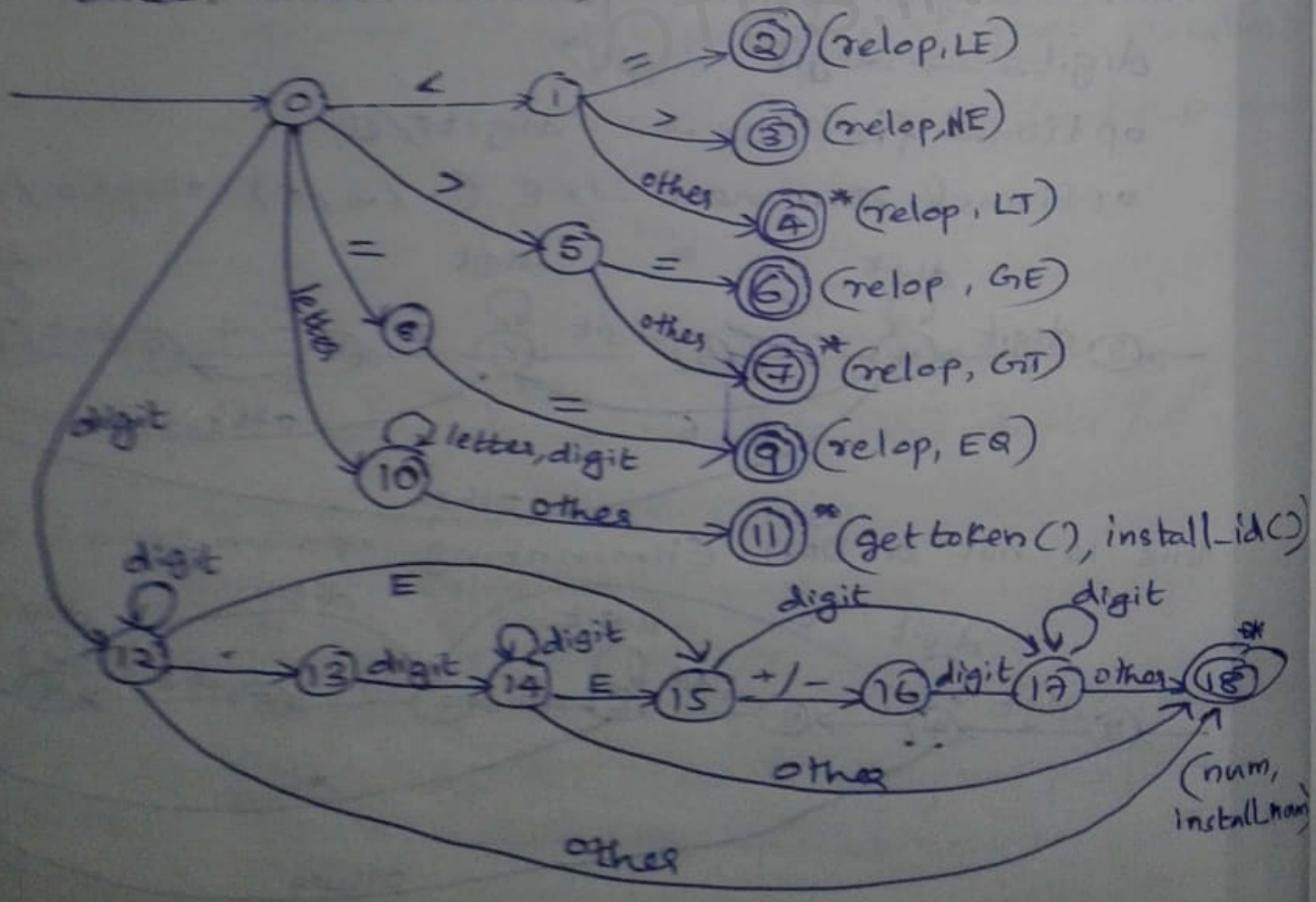


This is not a DFA. Eliminating epsilon transitions ⇒



Regulas expression	token	attribute value.
if	if	–
then	then	–
else	else	–
id	id	Pointer to symbol table entry
num	num	"
>	relop	GT
>=	relop	GE
<	relop	LT
<=	relop	LE
<>	relop	NE
==	relop	EQ

Combined automata ⇒



Final state gives token name and attribute value

x = abc * 10

↑ token beginning ↑ forward Pointer

retract: forward pointer retracts back one position

? Write a procedure for finding automata - next token()

```
token next token()
{ while(1)
  { switch (state)
    { case 0 : c = next char()
               if (c = blank || c = tab || c = newline)
                  state = 0
                  lexeme beginning ++
               else if (c == '<')
                  state = 1
               else if (c == '>')
                  state = 5
               else if (c == '=')
                  state = 8
               else if (isletter(c))
                  state = 10
               else if (isdigit(c))
                  state = 12
               else
                  state = fail()
               break;
      case 1 : c = nextchar()
               if (c == '=') state = 2
               else if (c == '>') state = 3
```

```
        else state = 4
        break;
Case 2: return (relop, LE)

Case 3: return (relop, NE)

Case 4: retract (1)
        return (relop, LT)
Case 5: c = nextchar()
        if (c == '=') state = 6
        else state = 7
        break;
Case 6: return (relop, GE)

Case 7: retract (1)
        return (relop, GT)

Case 8: c = nextchar()
        if (c == '=') state = 9
        break;
case 9: return (relop, EQ)

Case 10: c = nextchar()
        if (isdigit(c) || isletter(c)) state = 10
        else state = 11.
        break;
Case 11: retract (1);
        install_id()
        return (gettoken())
```

```
Case 12: c = nextchar()
         if (isdigit(c)) state = 12
         else if (c == '.') state = 13
         else if (c == 'E') state = 15
         else state = 18
         break;
Case 13: c = nextchar()
         if (isdigit(c)) state = 14
         break;
Case 14: c = nextchar()
         if (isdigit(c)) state = 14
         else if (c == 'E') state = 15
         else state = 18
         break;
Case 15: c = nextchar()
         if (isdigit(c)) state = 17
         else if (c == '-' || c == '+') state = 16
         break
Case 16: c = nextchar()
         if (isdigit(c)) state = 17
         break;
Case 17: c = nextchar()
         if (isdigit(c)) state = 17
         else state = 18
         break;
Case 18: retract (1)
         install num()
         return(num) }
}
```

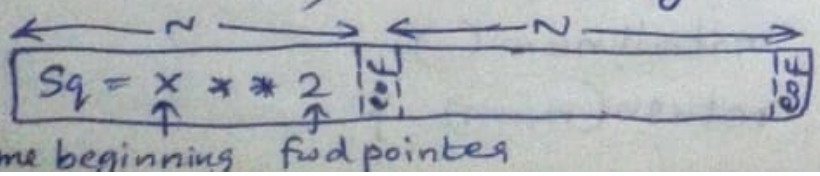
Transition diagrams depict the actions that take place when a lexical analyzer is called by the parser to get next token.

* indicates states on which input retraction must take place. If failure occurs while we are following transition diagram, then we retract the forward pointer to where it was in the start state and then activates next transition diagram. (forward pointer is retracted to the position marked by lexeme beginning pointer). If error occurs in all transition diagrams, then lexical error has been detected.

gettoken() and install_id() is used to obtain token and attribute value. Procedure install_id() has access to buffer, where identifier lexeme has been located. Installid() returns 0 if lexeme is keyword & return 1 if it is program variable. gettoken() similarly returns token if it is keyword and returns token id otherwise.

Input Buffering

* efficiency issues concerned with buffering of i/p.
* Two-buffer i/p scheme - useful when look ahead on i/p is necessary to identify tokens.



Sentinals is special characters that cannot be part of the source program. e.g: eof
We read N i/p characters into each half of buffer with one s/m read command rather than a read command for each i/p character. eof marks the end of source file. The string of characters b/w the two pointers is current lexeme. Initially both points to first character of next lexeme to be found. After current lexeme is processed both pointers are set to character immediately past the lexeme. If forward pointer is about to move halfway mark, right half is filled with N new i/p characters. If forward pointer about to move past right end of buffer, left half is filled with N new characters.

Using the sentinal characters 3 conditions are checked
1. Check end of 1st buffer
2. check end of 2nd buffer
3. check end of file

Lookahead code with sentinels ⇒

```
forward := forward + 1
if forward↑ = eof then
    begin
        if end of 1st buffer then
        begin
            reload 2nd buffer
        end forward := forward + 1
```

```
else if end of second buffer then
    begin
        reload first buffer
        move forward to beginning of 1st buffer
    end
else
    terminate lexical analysis
end.
```

Interaction of lexical analyser with parser