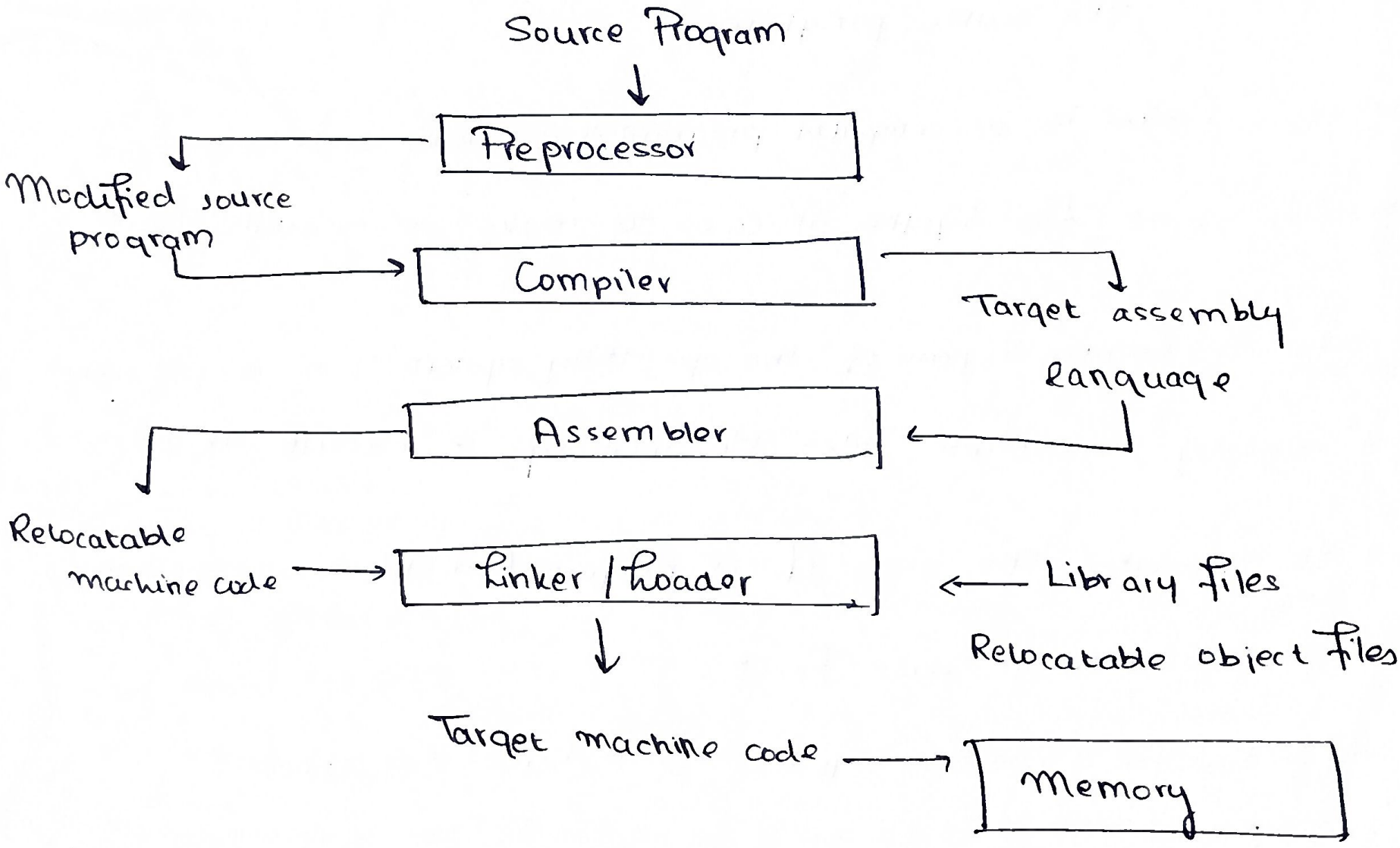
# UCS2702 COMPILER DESIGN

## Unit I

### Introduction to Compilers

Language processors - phases of compiler - role of lexical analyzer - Input buffering - specification of tokens - recognition of tokens - conversion of RE to DFA; Lexical analyzer generator: Structure of lex program - lookahead operator and conflict resolution

### * Language Processors



**Translators** : Source language → Target language

**Compiler** : High level language → Target / Machine / Assembly lang

FORTRAN, PASCAL, C/C++

(Compiler also gives error msgs)

**Language Processors** : Assembly language → Machine language

**Preprocessor** : High-level language → another high level language

**Interpreter** : Input → intermediate code

Instead of output, performs operations implied by the source program

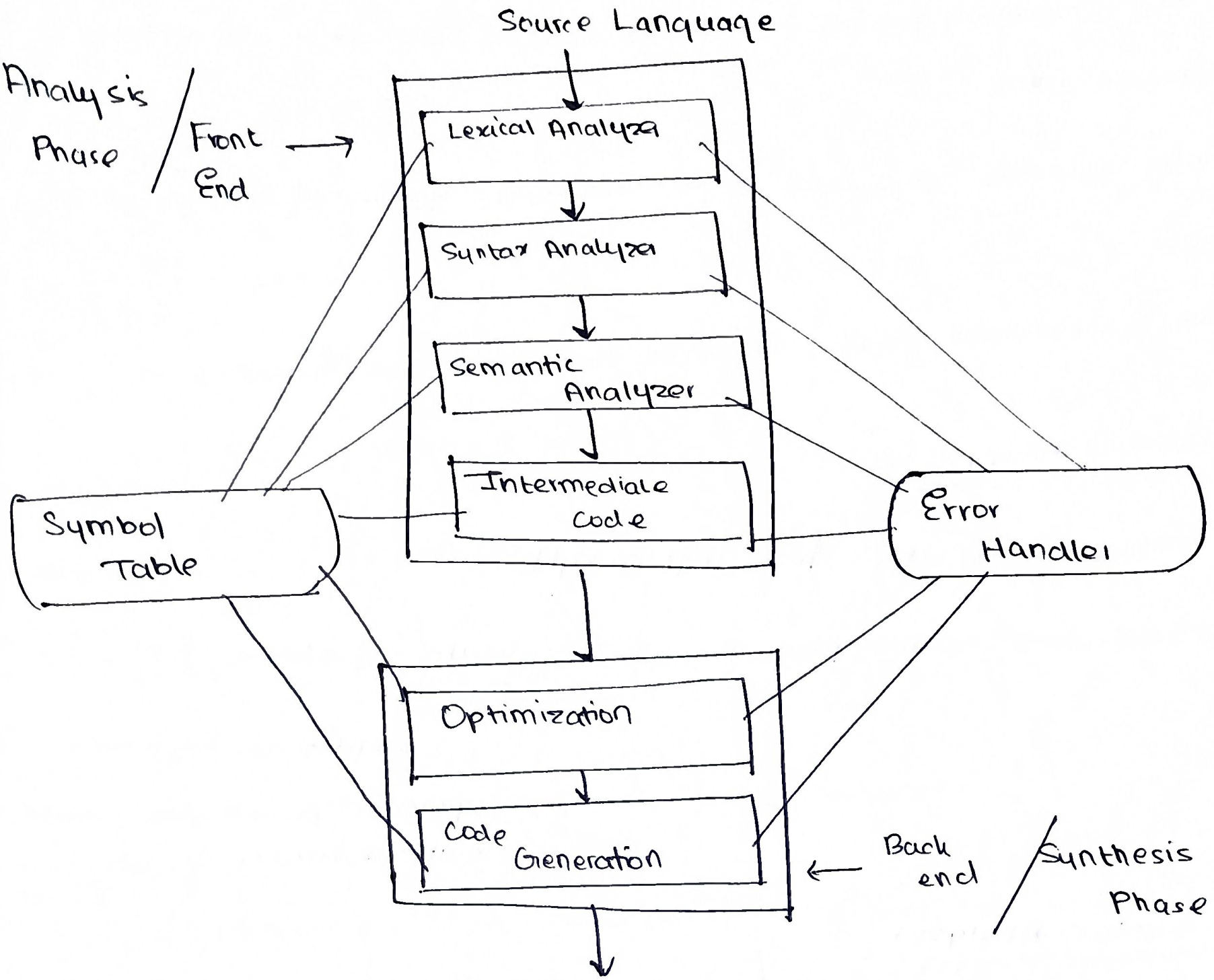
**Linker**: Linker is a computer program that links and merges various object files together in order to make an executable file.

**Loader** :→ Loader is part of the operating system & is responsible for loading executable files into memory & execute them.

→ It calculates the size of a program (instructions and data) & creates memory space for it

→ Initializes various registers to initiate execution

# * Phases of a Compiler




There are 2 main phases of a compiler

## ① Analysis Phase

- an intermediate representation is created from the given source program
- Lexical analyzer, syntax analyzer, semantic analyzer & intermediate code generator are the parts of this phase

## (2) Synthesis Phase

- the equivalent target program is created from this intermediate representation

- Code generator and code optimizer are the parts of this phase.

The individual stages in more detail are as follows:

A. Lexical Analyzer

→ reads characters from left to right

→ converts source program into a stream of tokens
↓
identifiers, keywords, operators, punctuation symbols, multi-character operators

B. Syntax Analyzer

→ converts a stream of tokens into a parse tree

Performs 2 functions:

(i) Checks the token if they occur in patterns specified by the source language

(ii) Construct a tree-like structure that can be used by the subsequent phases (Parser also known as hierarchical analysis)

The hierarchical structure of programs is usually described by recursive rods as follows:

1. Any identifier is an expression
2. Any number is an expression
3. If expression1 & expression2 are expressions, then so are:
   a. Expression1 + Expression2
   b. Expression1 * Expression2
   c. (Expression1)
4. If identifier1 is an identifier and expression2 is an expression then:

   Identifier1 := Expression2 is a statement

5. If Expression1 is an expression & Statement2 is a statement, then:
   a. while (Expression1) doStatement2
   b. if (Expression1) then Statement2 are statements

C. Semantic Analyzer - checks for more static semantic errors

parse tree → modified / annotated parse tree

D. Intermediate Code Generator - variety of intermediate representations

## Properties of 3 Address Code

→ at most one operator other than the assignment operator

→ Compiler must generate a temporary name to hold the value computed by each instruction.

→ Some 3 address instruction can have less than 3 operands

E) Code Optimizer

Tries to run code to

→ run faster

→ be smaller

→ consume less energy

F) Code Generator - converts optimized code into target program

## Other components

① **Symbol Table** → keep track of names declared in the program

→ separate levels for each scope

→ linear list → slow, but easy to implement

Hash Table → complex to implement, but fast

② **Error Handler**

- detection of errors
- reporting errors
- recovery of errors

## * Applications of Compilers

→ Techniques used in a lexical analyzer can be used in text editors, information retrieval system & pattern recognition programs

→ Parser - query processing system such as SQL

→ Many software having a compler front-end may need techniques used in compiler design

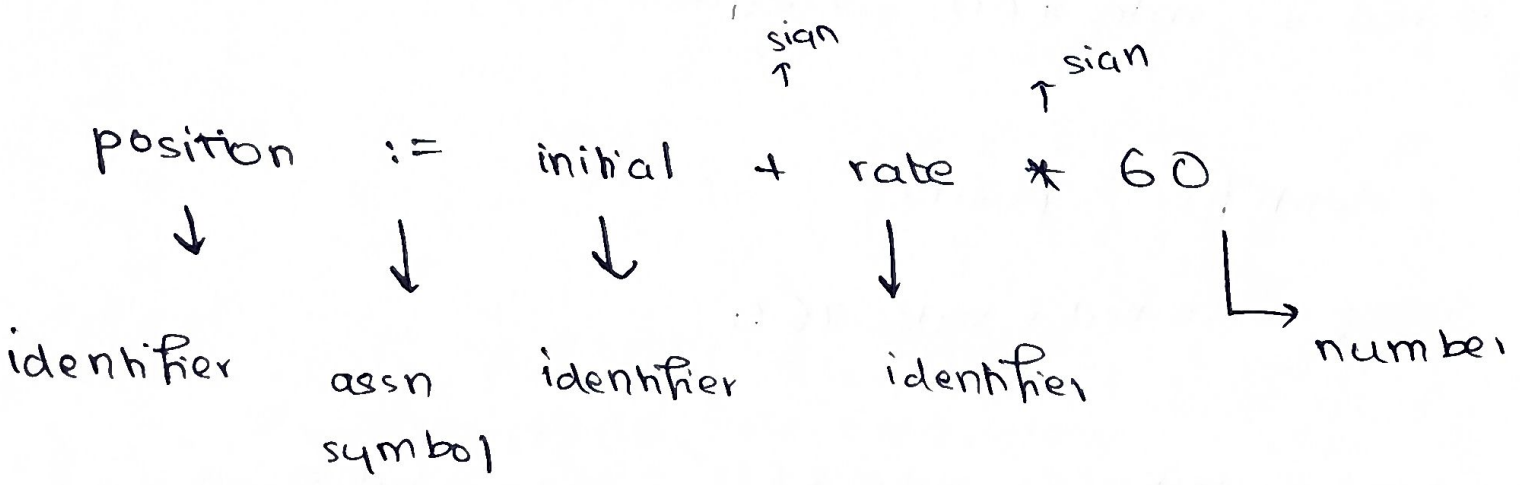
→ Most techniques used in compiler design can be used in NLP systems

(-X-)

Example: For the given expression, show its transformation through each stage

position := initial + rate * 60

Step1 : Lexical analyzer → L→R parse - get tokens




Step2 : Syntax Analyzer → check patterns, construct parse tree

Referring the rules on pg. (5)

position := initial + rate * 60

By Rule (1) position, initial, rate } => expression

By Rule (2) 60 } => expression

By Rule (3) a exp1 = rate

exp2 = 60

rate * 60 => expression

By rule (3) b exp1 = rate * 60

exp2 = initial

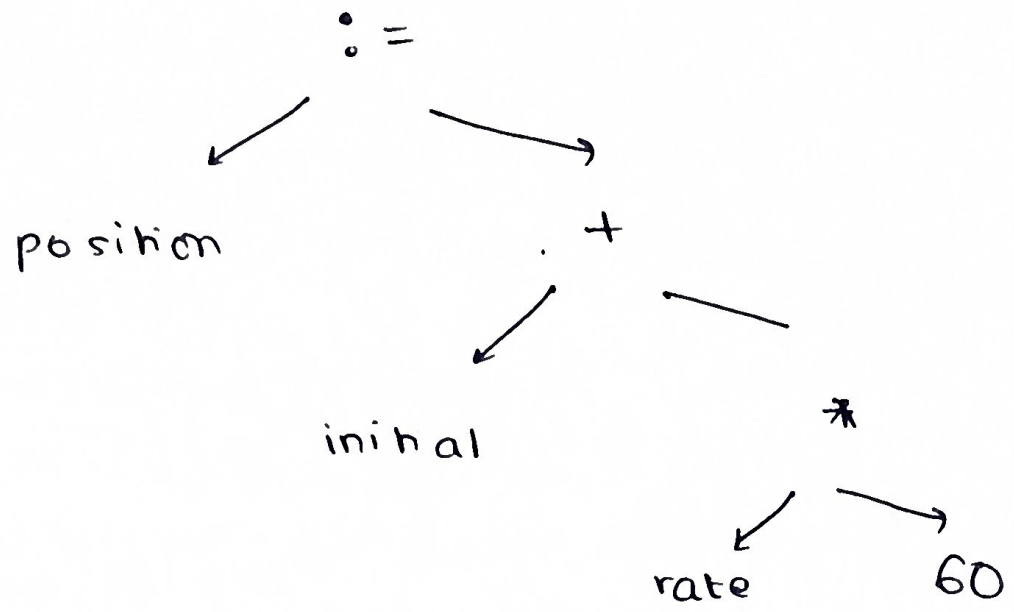
initial + rate * 60 => expression

By rule (4) identifier = position

exp = initial + rate * 60

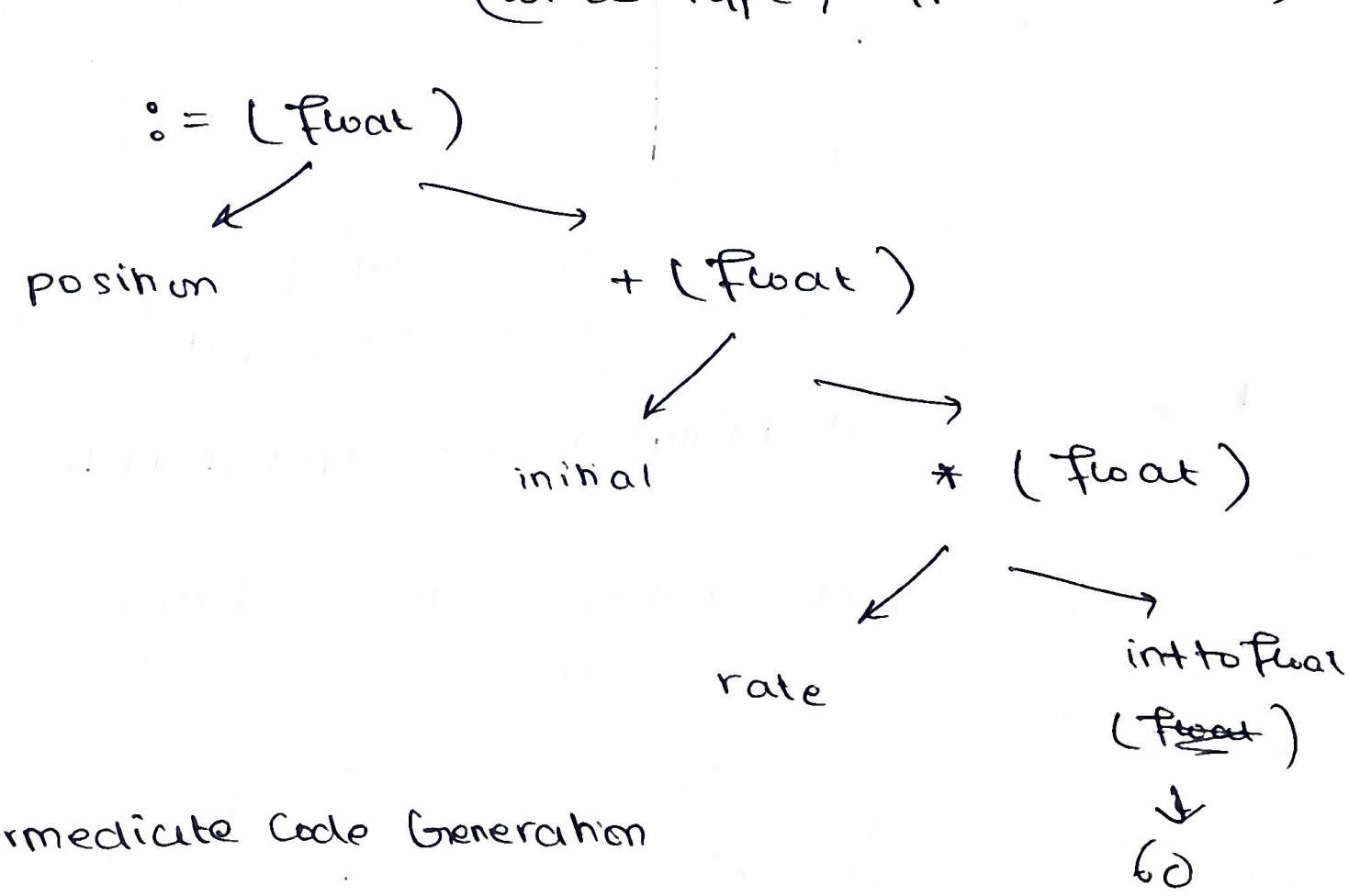
=> position := initial + rate * 60 => statement

Now constructing a parse tree




Step 3 : Semantic Analyzer

→convert to parse tree with annotations
(write type / type conversions)




Step 4: Intermediate Code Generation
(use the properties of 3 address code)

temp1 := int to float (60)
temp2 := rate * temp1
temp3 := inital + temp2
position := temp3

Step5 : Code Optimizer

temp1 : id3 * 60.0 → rate

id1 : id2 + temp1 → initial

position

Step6 : Code Generator

```
MOVF   id3, R2      # Move value of id3 (with rate value) into Reg R2
MULF   #60.0, id3
MOVF   id2, R1      # Move value of id2 (w/ initial value) into reg R1
ADDF   R2, R1
MOVF   R1, id1
```

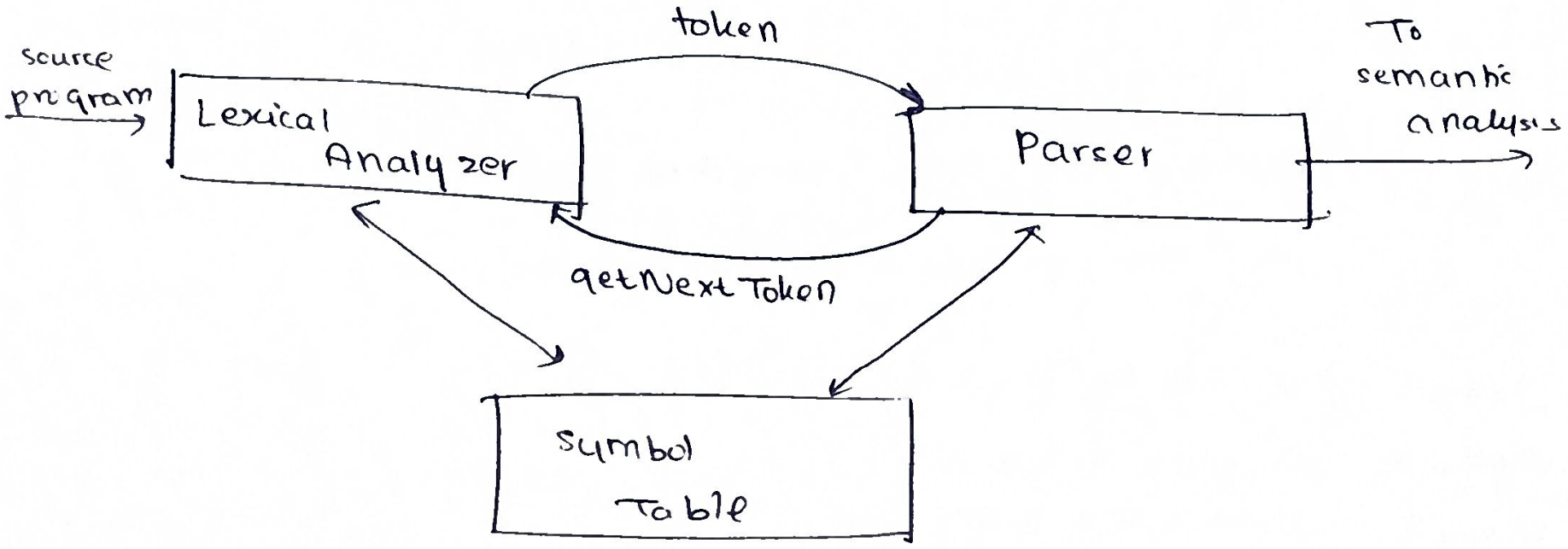
Move vars to registers R1 & R2

o/p stored in second term

→ move result of addition stored in R1 to id1

## * Role of Lexical Analyzer and Input Buffering

A lexical analyzer & parser work as follows:



## * Why separate lexical analysis & parsing?

① Simplicity of Design → Removing white space by lexical analyzer allows for easy implementation of parsing

② Improving Compiler Efficiency → Large amount of time is spent in reading the source program. Buffering techniques are used for reading input characters

③ Enhancing Compiler Portability → The representation of special symbols can be isolated in lexical analyzer.

## * Tokens, Patterns and Lexemes

① Token - A token is a pair of a token name and an optional token value.

② Pattern - A pattern is a description of the form that the lexemes of a token may take.

③ Lexeme - A lexeme is a sequence of characters in the source program that matches the pattern for a token.

eg.

| Token | Lexemes |
|---|---|
| if | if |
| else | else |
| comparison | <=, <, > |
| id | pi, score, variable |
| number | 1, 3.14, -6 |

## * Attributes for Tokens

eg. $E = M * C ** Q$

- <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

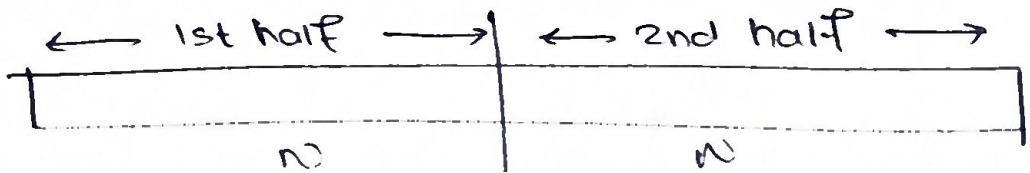
→ Sometimes, the lexical analyzer needs to look ahead some symbols to decide about the token to return.

→ A two-buffer scheme is introduced to handle large lookahead safely

### Buffer Pairs

→ The buffer is divided into two N-character halves.

N is the no. of characters 1024 or 4096



## Buffer Pair Algorithm

There are two pointers- lexeme-beginning & forward

lexeme-beginning is moved only once a token has been recognized. It then moves to the next character to parse.

forward moves one step forward for every character

Algorithm

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 forward := forward + 1

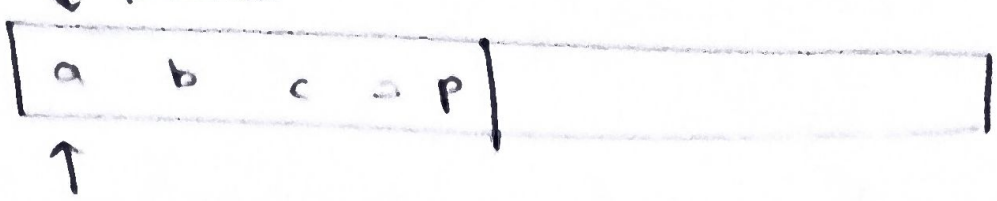
example consider abc = pqr * xyz

divide the characters into half

First half of buffer = abc = p

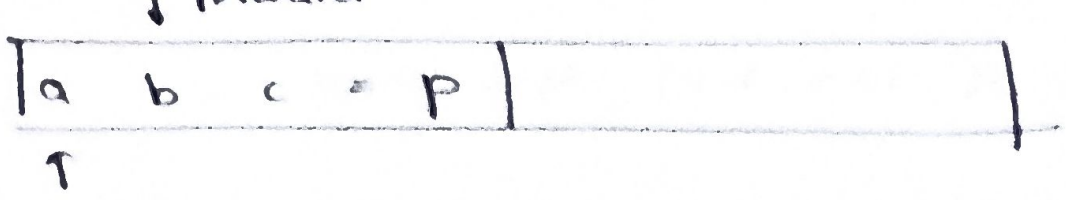
second half of buffer = qr * xyz

(i)



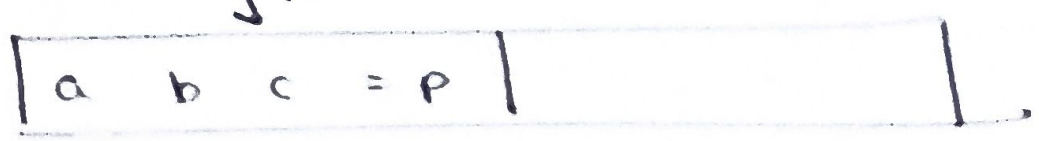

lexeme beginning

(ii)




lexeme beginning

(iii)



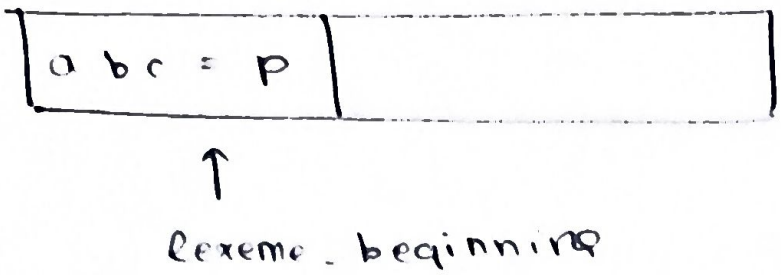

lexeme beginning

(iv)

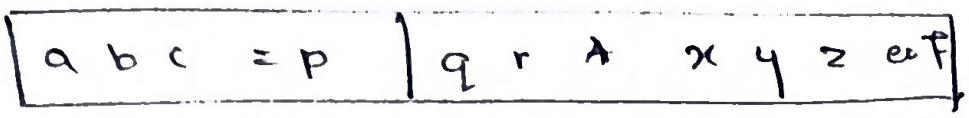

lexeme beginning = => assignment operator

=> abc → identifier

(v)




(vi) now load 2nd half




continue pointer movement

* Sentinels

→ Sentinels are used to mark the end of input in the buffer without requiring explicit checks after each character.

Algorithm

move forward pointer & check character at location

```
Switch ( * forward ++ ) {

    case eof :

        if (forward is at the end of first buffer) {

            reload second buffer

            forward := beginning of second buffer;

        }
```

else if ( forward is at the end of second buffer ) {

reload first buffer

forward := beginning of first buffer

}

else {

# means eof within buffer => marks end of input

terminate lexical analysis

break ;

}

case for other characters

## * Lexical Errors

→ Some errors are out of power of the lexical analyzer to recognize :

eg fi (a == f(x))

→ However, it may be possible to recognize errors like,

d = 2r

Such errors are recognized when no pattern for tokens matches a character sequence.

## * Error Recovery

→ Panic mode : successive characters are ignored until we reach a well-formed token

→ Delete one character from the remaining input

→ Insert a missing character into the remaining input

→ Replace a character by another character

→ Transpose two adjacent characters

Example : Tokenize the following expression

$$a + b / 2.6$$

a : Identifier

+ : Operator

b : Identifier

/ : Operator

2.6 : Numeric literal

## * Specification of Tokens

① Alphabet or Character Class

→ $\Sigma$ is a finite set of symbols

→ $\{0,1\}$ is a binary alphabet

② String

→ A finite sequence of symbols from $\Sigma$

→ $|s|$ = length of string

→ $\epsilon$ = empty string $|\epsilon| = 0$

## ③ Language

→ A language is a specific set of strings over some fixed alphabet $\Sigma$.

### More Specifications

① Prefix of s - A string obtained by removing 0 or more trailing symbols of s

② Suffix of s - A string formed by deleting 0 or more leading symbols of s

③ Substring of s - A string obtained by removing the suffix and prefix from s

④ Proper prefix and proper suffix - Any prefix or suffix other than the string itself

⑤ Subsequence of s - Any string formed by deleting zero or more not necessarily contiguous symbols from s

banana → baaa

### * Language Operations

① Union : $L \cup M = \{s \mid s \in L \text{ or } s \in M\}$

② Concatenation: $LM = \{xy \mid x \in L \text{ & } y \in M\}$

③ Kleene closure: $L^* = \bigcup_{i=0 \ldots \infty} L^i$

④ Positive closure : $L^+ = \bigcup_{i=1 \ldots \infty} L^i$

## * Regular Expressions

1. $\epsilon$ is a regular expression, $L(\epsilon) = \{\epsilon\}$
2. If a is a symbol in $\Sigma$, then a is a regular expression $L(a) = \{a\}$
3. $(r)|(s)$ is a RE denoting the language $L(r) \cup L(s)$
4. $(r)(s)$ is a RE denoting the language $L(r)L(s)$
5. $(r)^*$ is an RE denoting $(L(r))^*$
6. $r$ is an RE denoting $L(r)$

Operator Precedence

*
.
|

## * Recognition of Tokens

Step1 : ID the starting point, to understand the grammar the tokens denote

eg. stmt → if expr then stmt | if expr then stmt else stmt | $\epsilon$

expr → term relop term | term

term → id | number

Step2: Formalize the patterns

eg.

digit → [0-9]

digits → digit+

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

letter → [A-Z a-z _]

id → letter ( letter | digit )*

if → if  else → else

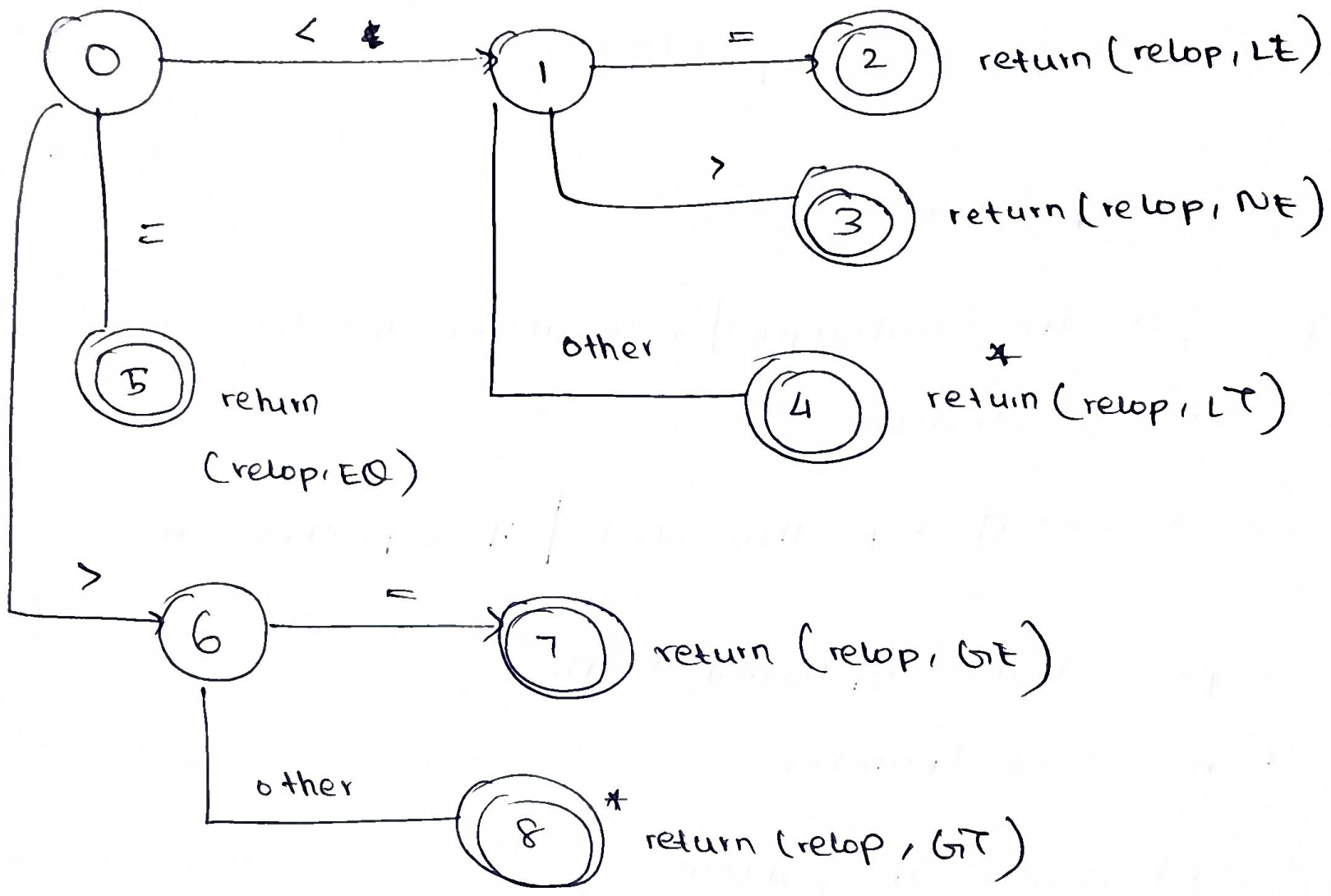
then → then  relop → < | <> | = | <= | >= | <>

Step 3 | Handle white spaces |

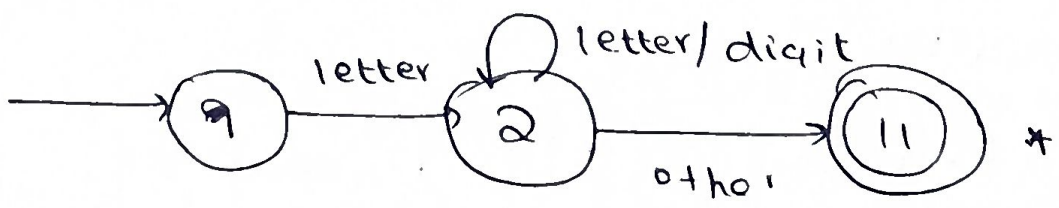
ws → ( blank | tab | newline )+

Step 4 | Draw a transition diagram |

Example 1 : Draw a transition diagram for relational operators

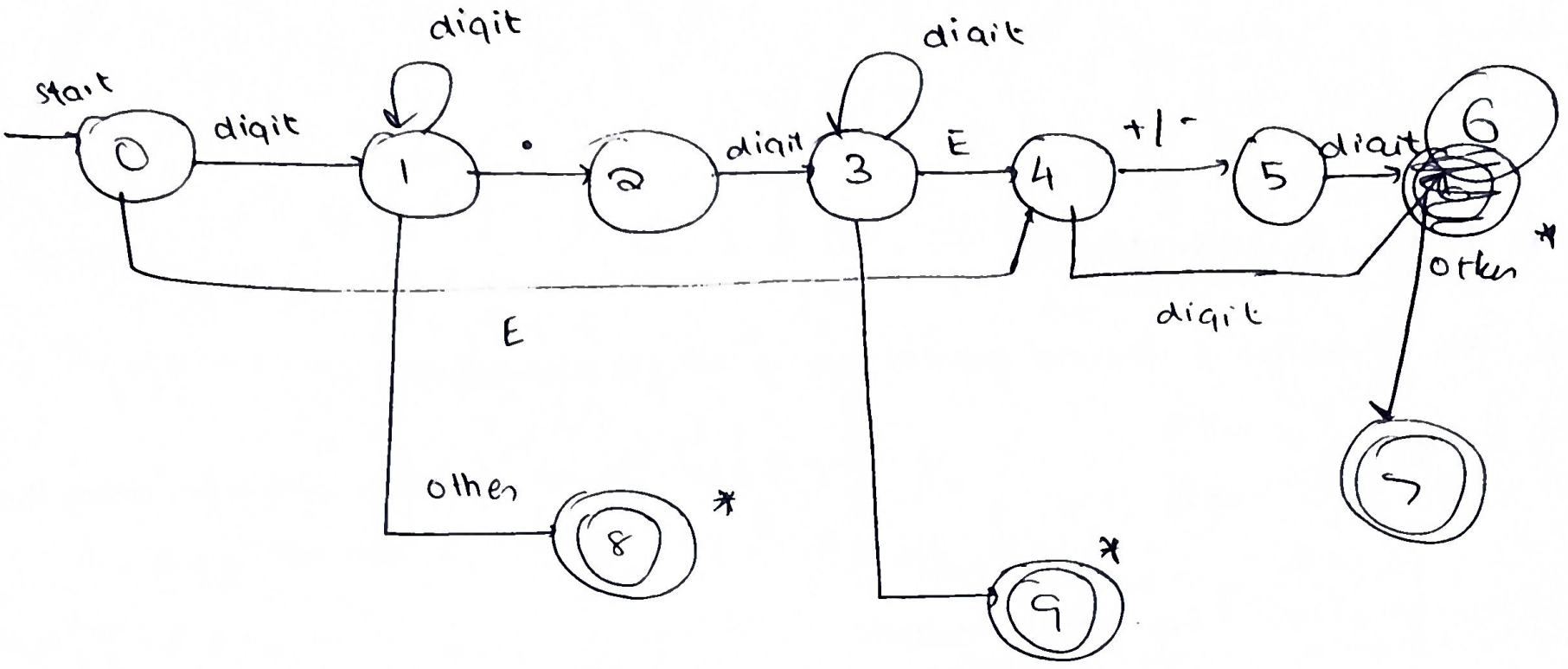



Example 2 : Transition diagram for reserved words and identifiers

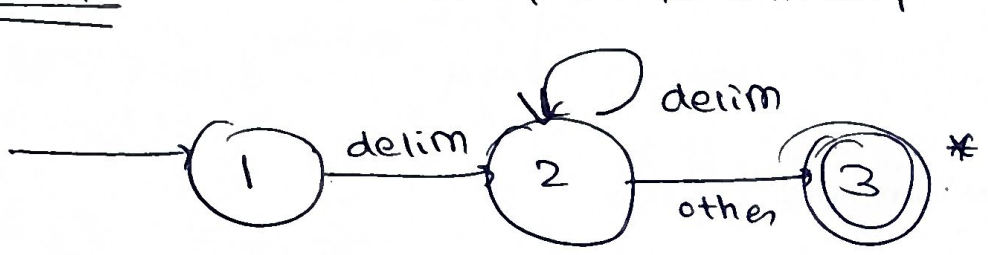



return (getToken(),
installID())

**Example 3** Transition diagram for unsigned numbers

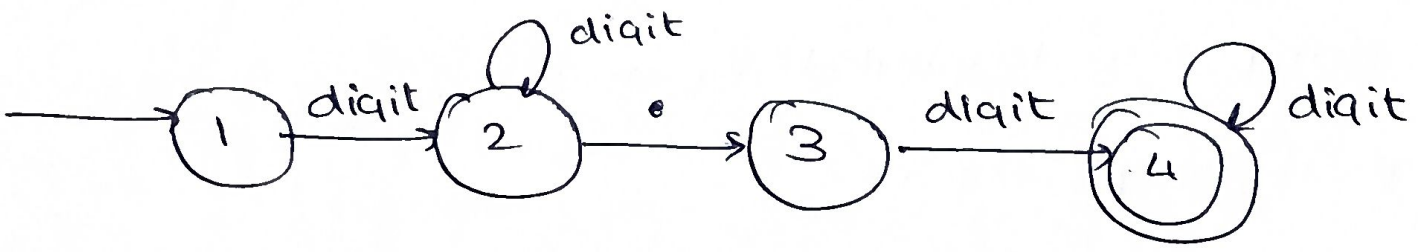



**Example 4** Transition diagram for whitespaces

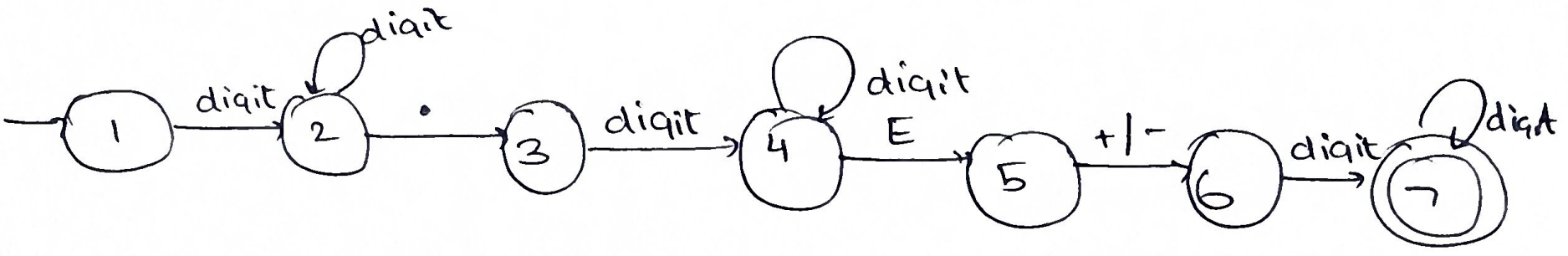



**Example 5** Draw the transition diagram for the following

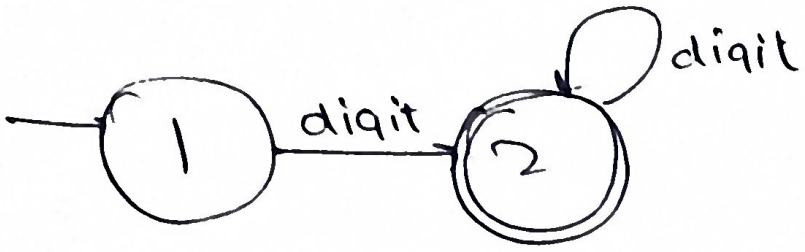
(i) ~~3e4e5~~. 3.4.56




(ii) 3.43E-5



(ii) 3456




## * Finite Automata

→ RE = outline, Finite automata = implementation

→ 2 types → NFA
→ DFA

↳ have finite memory, encode only the current state

### Deterministic Finite Automata

- one transition per input per state
- No $\varepsilon$- moves

$(S, \Sigma, \delta, s_0, F)$

- $\delta$: mapping $S \times \Sigma \to S$
- $S$: Finite set of sets
- $\Sigma$: alphabet
- $s_0$: start
- $F$: Final state(s)

### Non-deterministic finite automata

- can have multiple transitions for one input in a given state
- can have $\varepsilon$- moves

## * Simulating a DFA

→ Input string $x$ is terminated by an EOF, for a DFA D

→ Output yes if accepts, else no

```
S := s0
a := nextchar()
while a ≠ eof do
        S := move(S, a)
        a := nextchar()
end do
```

if S is in F, then

return 'yes'

else

return 'no'

## * Nondeterministic Finite Automata

An NFA is a 5-tuple

$(S, \Sigma, \delta, s_0, F)$

- S → finite set of states
- $\Sigma$ → alphabet
- $\delta$ → mapping from $S \times \Sigma \rightarrow$ a set of states
- $s_0$ → start state
- F → set of final states

## * Direct Conversion from RE to DFA

Step 1 : Augment the regular expression r with a special end symbol #

Step 2 : Construct a syntax tree for (r)#

alphabets = leaf node

operator = inner node

Step 3 : Number each alphabet including #

Step 4 : Traverse the tree to construct the functions nullable, firstpos, lastpos & followpos

## Terms

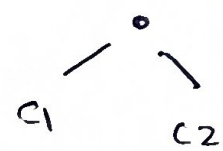
(i) nullable(n) : the subtree at node n generates languages including the empty string

(ii) firstpos(n): set of positions that can match the first symbol of a string generated by a subtree at node n

(iii) lastpos(n): the set of positions that can match the last symbol of a string generated be the subtree at node n

(iv) followpos(i): the set of positions that can follow position i in the tree

## * Annotating the tree

| Node n | nullable(n) | firstpos(n) | lastpos(n) |
|---|---|---|---|
| leaf ε | true | $\phi$ | $\phi$ |
| leaf i | false | {i} | {i} |
|   | nullable(c1) or nullable(c2) | firstpos(c1) ∪ firstpost(c2) | lastpos(c1) ∪ lastpos(c2) |
|   | nullable(c1) and nullable(c2) | if nullable(c1) then firstpos(c1) ∪ firstpost(c2) else firstpos(c1) | if nullable(c2) then lastpost(c1) ∪ lastpos(c2) else last pos(c2) |
|   | true | firstpost(c1) | lastpos(c1) |

## Followpos (n)

for each node n in the tree do

if n is a cat-node with the left child $c_1$ and right child $c_2$ then

for each i in lastpos($c_1$) do

followpos(i) := followpos(i) ∪ firstpos($c_2$)

end do

else if n is a star-node

for each i in lastpos(n) do

followpos(i) := followpos(i) ∪ firstpos(n)

end do

end if

end do

→ Calculate DTrans for each unique state starting from start

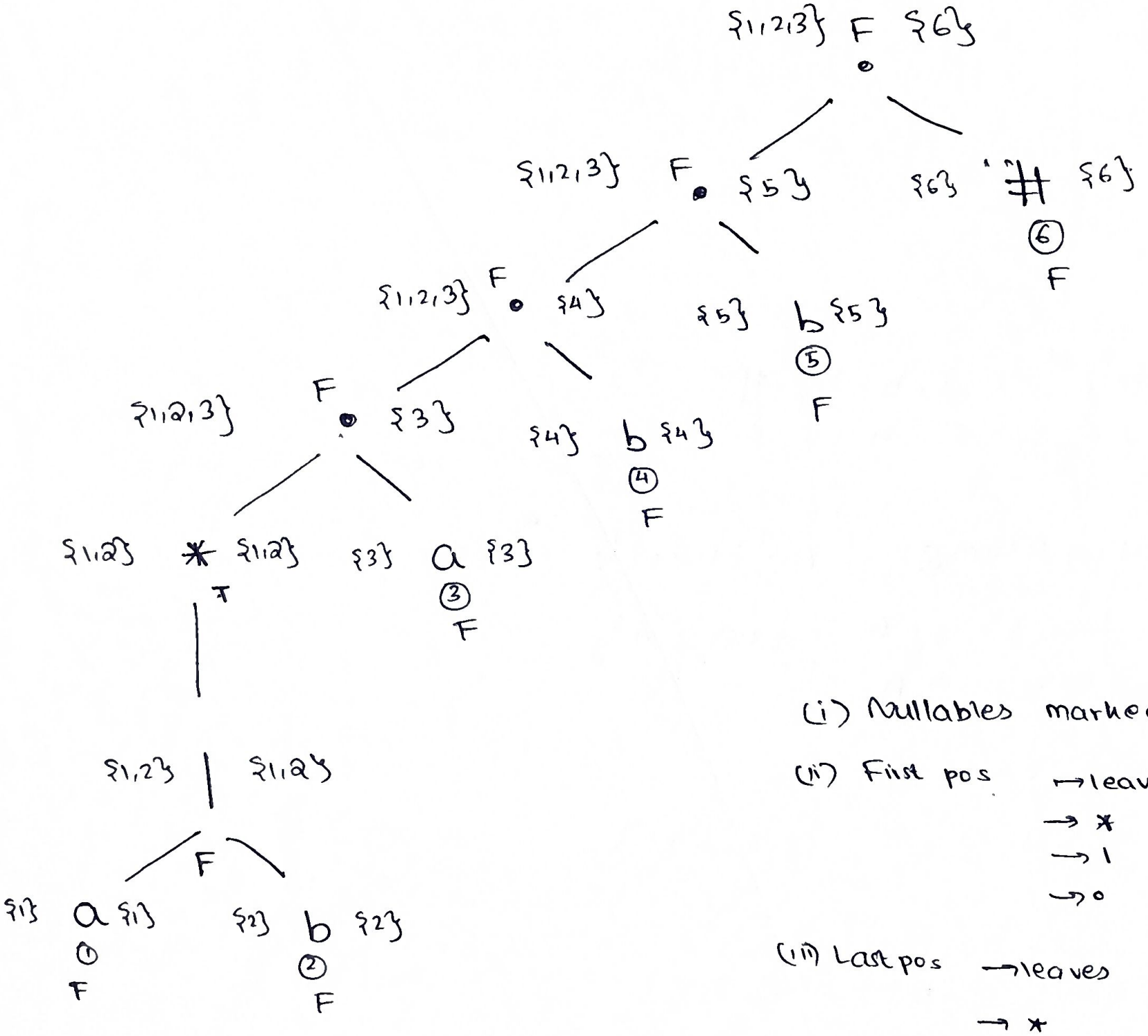
(Algo pg 91)

Convert the following Regular expressions into a DFA (2)

① $(a|b)^*abb$

$\Rightarrow (a|b)^* \cdot a \cdot b \cdot b \cdot \#$

Ans




(i) Nullables marked

(ii) First pos → leaves
→ *
→ |
→ .

(iii) Last pos → leaves
→ *
→ |
→ .

## Calculating Followpos

(i)

| Node | | | Followpos |
|---|---|---|---|
| a | 1 | (*) | 1,2,3 |
| b | 2 | (·) | 1,2,3 |
| a | 3 | (·) | 4 |
| b | 4 | (·) | 5 |
| b | 5 | (·) | 6 |
| # | 6 | | - |

cat-node: Followpos(i) :=

Followpos(i) ∪ Firstpos(c2)

Star node: Followpos(c) :=

Followpos(i) ∪ Firstpos(n)

$$(a/b)^* \cdot a \cdot b \cdot b \cdot \#$$
$$1\ \ 2 \qquad 3 \quad 4 \quad 5 \quad 6$$

## Calculating DTrans

A = {1,2,3}

DTrans(A,a) = B = {1,2,3,4}

DTrans(A,b) = A

DTrans(B,a) = B

DTrans(B,b) = C {1,2,3,5}

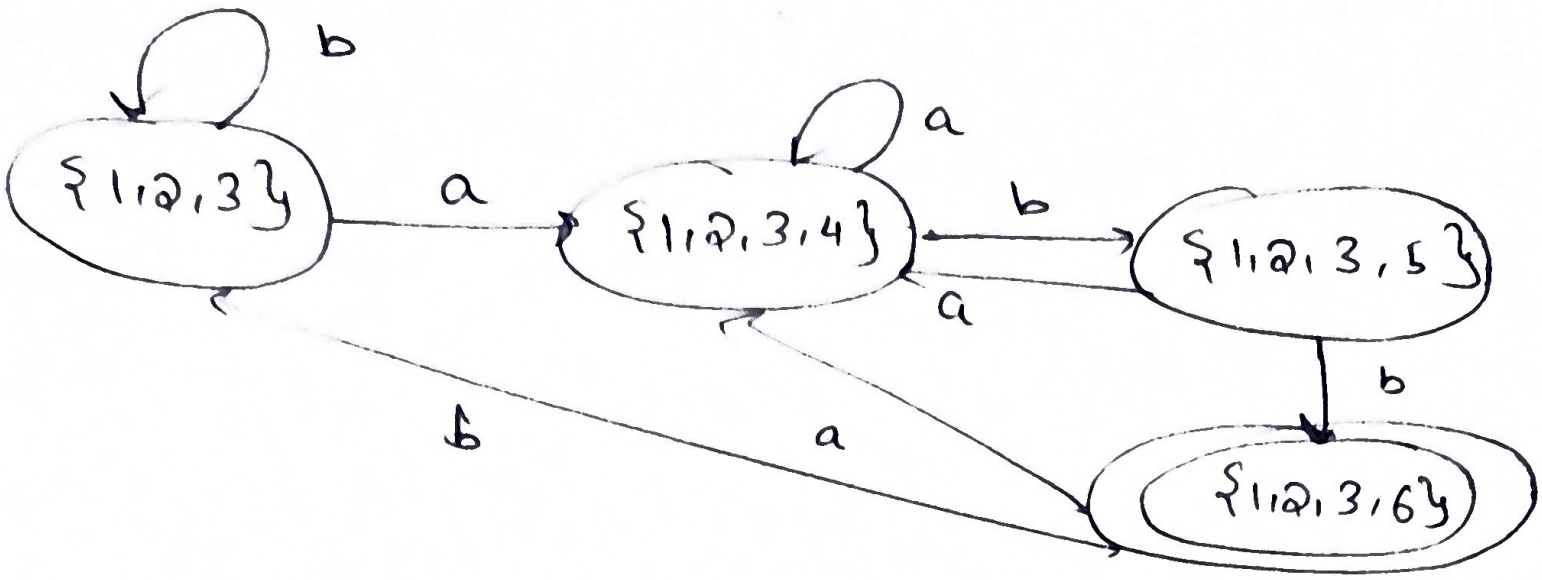
DTrans(C,a) = B

DTrans(C,b) = D {1,2,3,6}

DTrans(D,a) = B

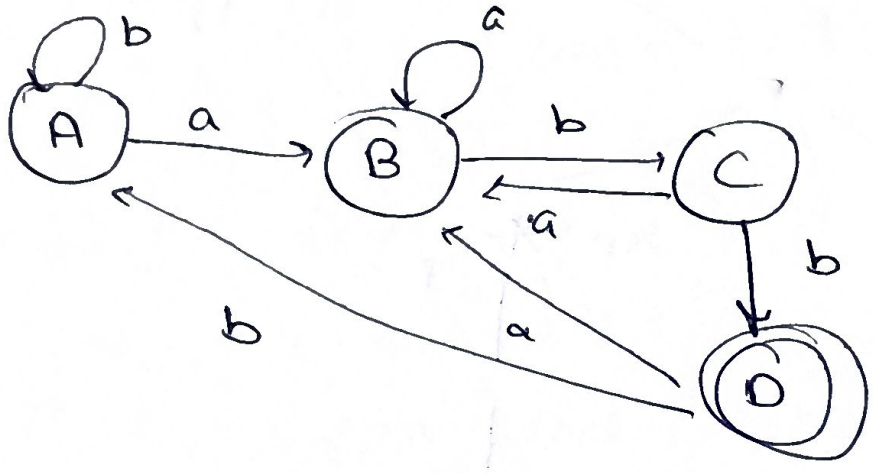
DTrans(D,b) = A

# Drawing a DFA




a 1 123
b 2 123
a 3 4
# 6 -

## Final DFA

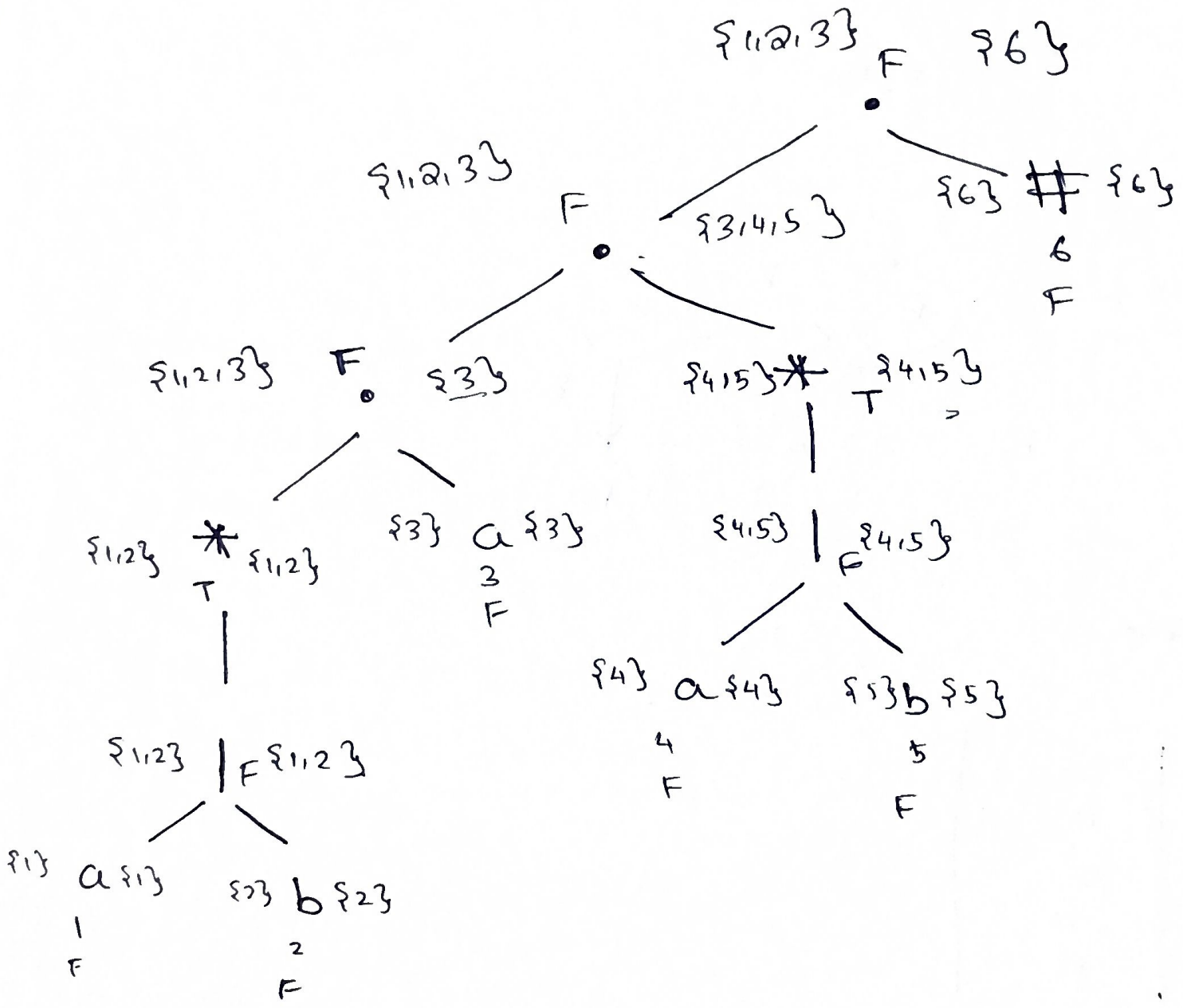



## Transition Table

| | S | a | b |
|---|---|---|---|
| → | A | B | A |
| | B | A | C |
| | C | B | D |
| * | D | B | A |

② $(a|b)^* a (a|b)^*$

$(a|b)^* \cdot a \cdot (a|b)^* \cdot \#$



## Calculating Followpos

$$(a|b)^* \; a \; (a|b)^* \; \#$$
$$1 \quad 2 \quad 3 \quad 4 \quad 5 \quad 6$$

| Node name | Node no | Followpos |
|---|---|---|
| a | 1 | 1,2,3 |
| b | 2 | 1,2,3 |
| a | 3 | 4,5,6 |
| a | 4 | 4,5,6 |
| b | 5 | 4,5,6 |
| # | 6 | – |

## Calculating DTrans

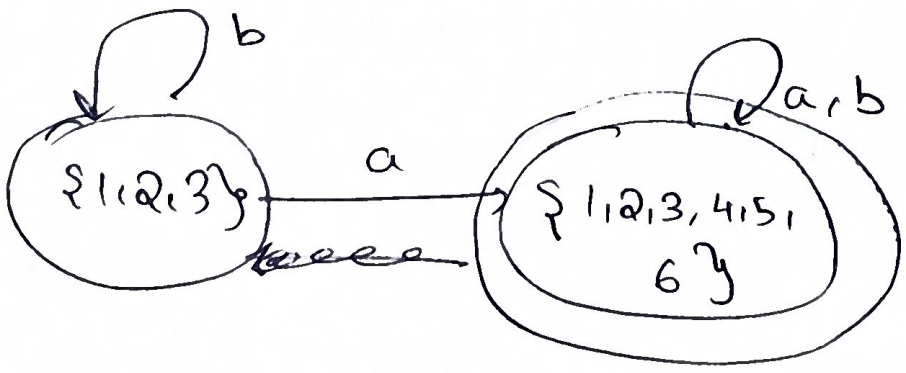
DTrans (A,a) = {1,2,3,4,5,6} = B

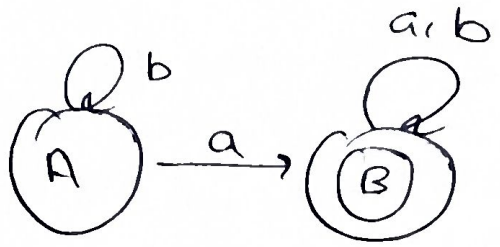
DTrans (A,b) = {1,2,3} = A

DTrans (B,a) = B

DTrans (B,b) = B

## Constructing DFA



# * Lex- Lookahead Operator and Conflict Resolution

## Conflict Resolution

→ Conflict arises when several prefixes of input matches one or more patterns. This can be resolved by the following:

(i) Always prefer a longer prefix than a shorter prefix

(ii) If two or more patterns are matched for the longest prefix, then the first pattern listed in the lex program is preferred.

eg. for patterns:

a → $P_1$
abb → $P_2$
a*b+ → $P_3$

For string abb, $P_2$, $P_3$ match
$P_2$ is chosen as it was listed first in the lex program

## Lookahead Operator

→ An additional operator that is read by Lex in order to distinguish additional patterns for a token

→ Lexical analyzer is used to read one character ahead of valid lexeme and then retracts to produce a token.

→ At times, it is needed to have certain characters at the end of input to match with a pattern.

→ In such cases / is used to indicate end of part of pattern that matches the lexeme.

eg. If language keywords are not reserved :

IF (I,J) = 5 &

IF (condition) THEN

results in a conflict whether to produce IF as an arrayname or a keyword.

This can be resolved as :

IF / \(.*/) {keyword}

IF \(.*/) {arr name}

eg0 {IDENT} /ab

I/p = bracadabra

o/p = bracad for yytext

eg3 {IDENT}

I/p = bracadabra

o/p = bracadabra for yytext

eg4 Rule: a*b/cc

I/p : aaabcc

o/p = aaab for yytext

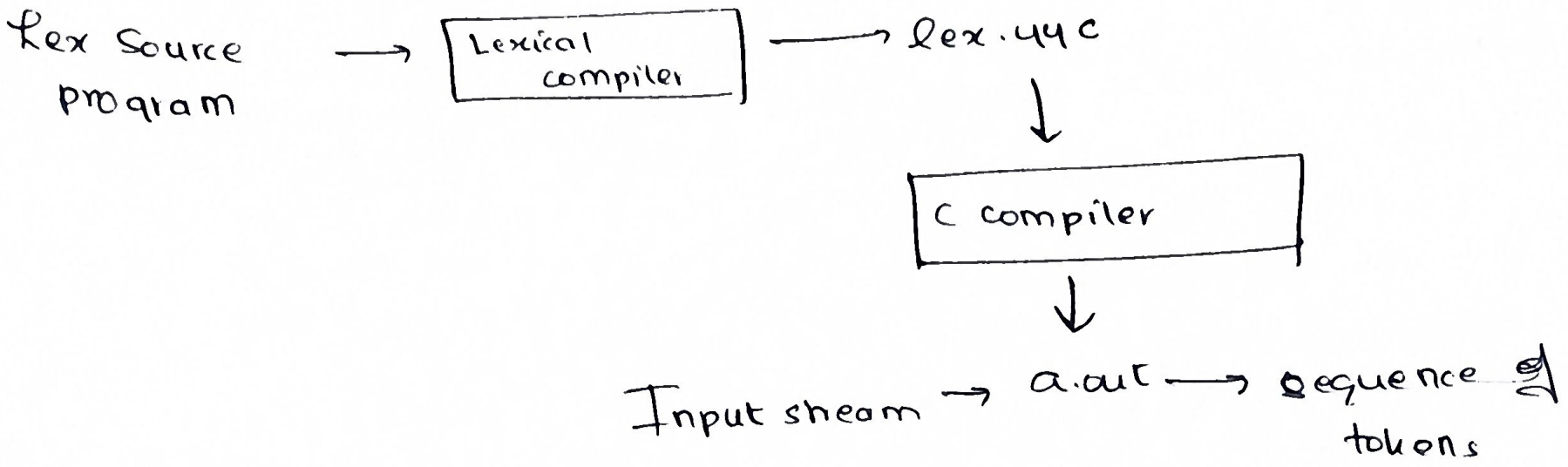
eg5 Rule: x* | xy

Ip = xxxy

o/p = xxx for yytext

## * Introduction to LEX

Lexical Analyzer Generator




## * Lex Specification

Has 3 parts :

(i) regular definitions, C definitions

(ii) Translation rules

(iii) User-defined auxiliary procedures

## * Sample lex Specification

```
%{
  #include <stdio.h>
%}

[0-9]+      {printf("%s\n",yytext);}

.|\n        {}

main()
{
  yylex();
}
```

compile it by:

```
lex   spec.l
acc   lex.yy.c   -ll
./a.out  spec.l
```

Ex1 Lex program to count no. of lines, words & characters

```
%{
    #include <stdio.h>
    int ch = 0,  wd = 0, nl = 0;
%}

delim    [\t]+
%%
\n          { ch++ ; wd++; nl++ }
{delim}     { ch += yyleng; wd++; }
.           { ch++ ; }
%%
main() {
        yylex();
        printf("%8d %8d %8d \n", nl, wd, ch);
}
```

Example: LEX Program to recognize digits, letters & identifiers

```
%{
    #include <stdio.h>
%}

digit     [0-9]
letter    [A-Za-z]
id        {letter} ( {letter} | {digit} )*

%%

{digit}+      {printf ("number: %s\n" yytext); }
{id}          {printf ("ident:  %s \n" , yytext); }
.             {printf("other: %s \n" , yytext); }

%%

main() {
      yylex();
}
```