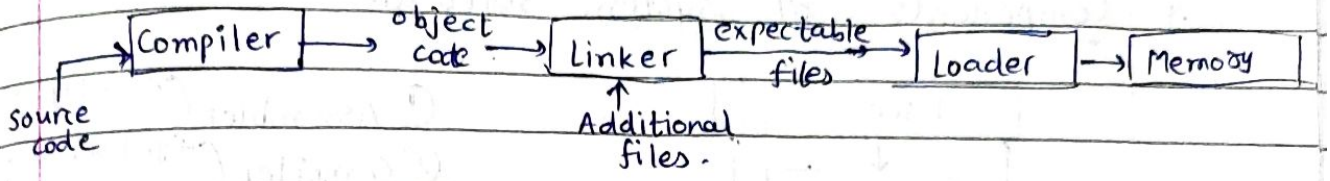
Language Translator

(Jeffrey D. Ullman, Ravi Sethi, Alfred V. Aho.)
CBS Publication)



Q. Meanings = High level language (User Oriented languages)
= Assembly language, Machine level language, Translator, Linker, Loader

Q* Real time application of language translator. (2 marks)
↳ Healthcare

- Computer → S/w^D + H/w^D

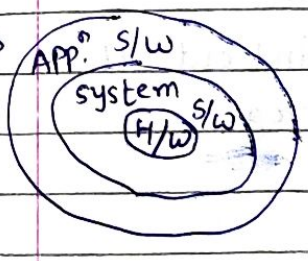
Assembler → Assembly to low
compiler → compiles whole prog. (obj file).
interpreter → compile line by line

CH-1

⁂ Language Translation overview

* need and component of system

- Application software : browser, word, bank managing syster
- System Software : OS, compiler, Assembler, interpreter.

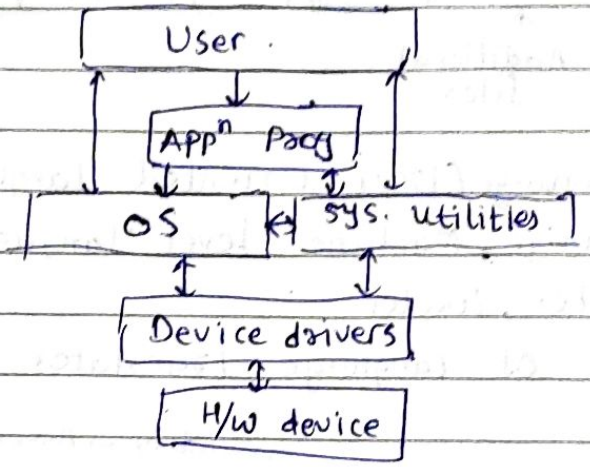
→ 

Q* (2 marks) System software is barrier b/w H/w & App^n S/w.

- Also sys. S/w will ~~conte~~. convert high level lang. to machine lang.
- Also helpful to store database
- Manage f^n storing data, retriving files & Shedding task
- To make effective execution of each prog.
- To mak effective utilization of all the resources

* Eg. 

* Components of system software



① Assembler (Assembly → Machine lang)
② Compiler (High level → Machine lang)
③ Interpriter (line by line High level → Machine lang)
④ editor (edit particular file/prog as per requirements)
⑤ Linker
⑥ Loader
⑦ Debugger
⑧ Macro
⑨ OS
⑩ device driver

→ Assembly lang → Assembler → Machine lang

→ High level → Compiler → Binary x Machine

→ prog. statement → Interpreter → Machine lang → statement execution

→ Editor Types = line editor, screen, word processor, structure, etc.

.c
.Lib
→ Linker → .exe (Integrete files which are require to execute)

→ Loader → Mem.

source code → Assembler → obj code → linker → executable code → loader → Mem.

→ Debugger is a computer prog. used to find out the errors which are also called as bugs in source prog.

→ Macro (Code reusability)

source code → macro processor → expand code → compiler Assembler → obj.

→ OS is the system prog. that enable user to communicate with computer H/W
- It has other prog. to run & control them.

eg: Linux, Windows, Unix.

Types - Single user, Multi tasking, multiprocessor, distributed, network

→ device driver → It is a system prog. used to control no. of devices which are attched to the computer, It tells us that how device will work or certain commands which are generated by the user

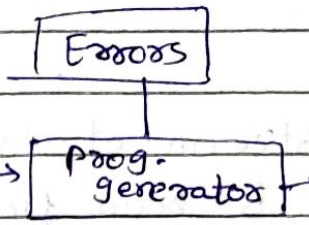
eg: mouse driver, KB, Wifi, Bluetooth

* Language processing Activities.

D.→ Language processor : Convert source code into machine code.

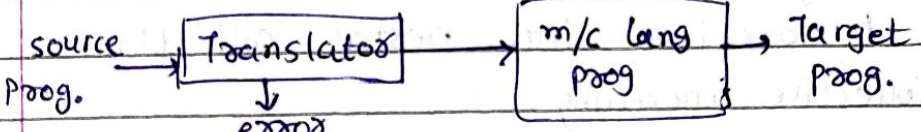
D.→ Language Processing : Activity which is carried out by the language processors

- prog. generation activity
- program execution activites
  - Prog. Translation
  - Prog. Interpretation



9/12/22

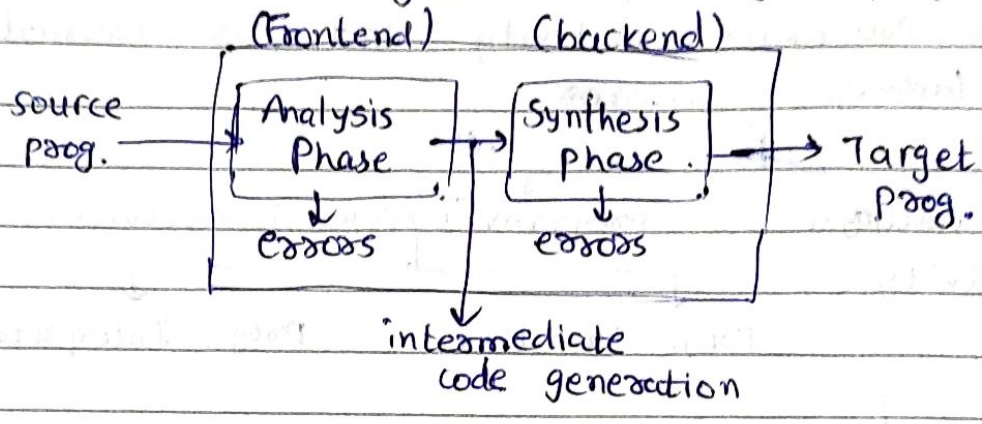
* Program Translation.



→ writing a Prog.
- Java file
- Compiler → class file
- Interpreter
  - Fetch
  - Decode
  - Execute

* Fundamental of language processing.

lang: processing = Analysis of source prog. + synthesis of target prog.



*→ Analysis phase has 3 steps

1] Linear analysis (Lexical analysis) - in which the stream of char making a the source prog. is read from left to right & grouped into tokens that are sequence of char having a collective meaning.

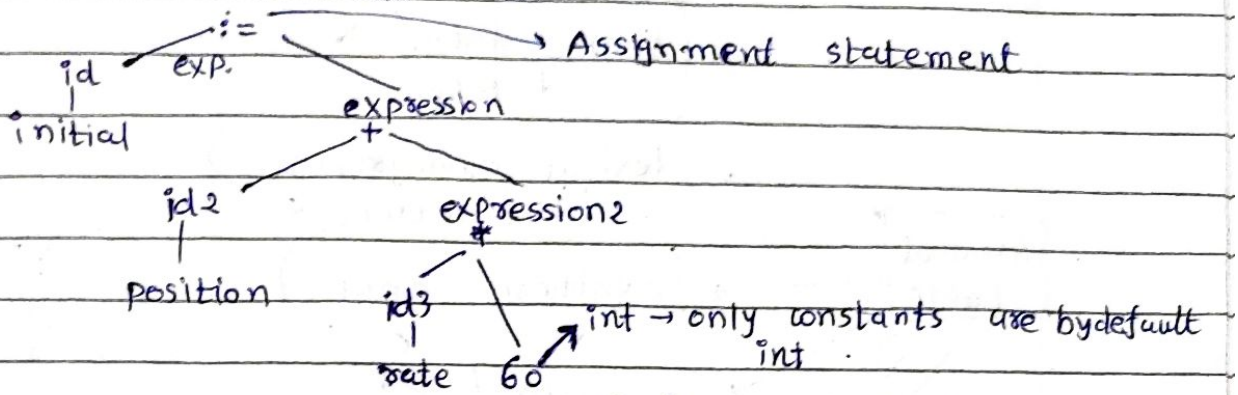
2] Heirachical analysis (Csyntext analysis) - In which char or tokens are grouped heirarchical into nested collections with collective meaning.

3] Symentic analysis (tight checking) - In which certain checks are performed to ensure that the components of a prog. fit together meaningful

eg. initial := position + rate * 60

1] initial → identifier
:= → assignment symbol
position → identifier
+ → Plus sign (symbol)
rate → identifier
* → mul. sgn (Symbol)
60 → int (constant)

2) initial = position + rate * 60
id1 = id2 + id3 * 60



3) all id's are real no.

* (Intermidiate code generation) So only here tight checking

→ temp1 = inttoreal(60)
temp2 = rate * temp1
temp3 = position + temp3
initial = temp3

→* Synthesis has 2 activities
1) Code optimizer , 2) Code generator

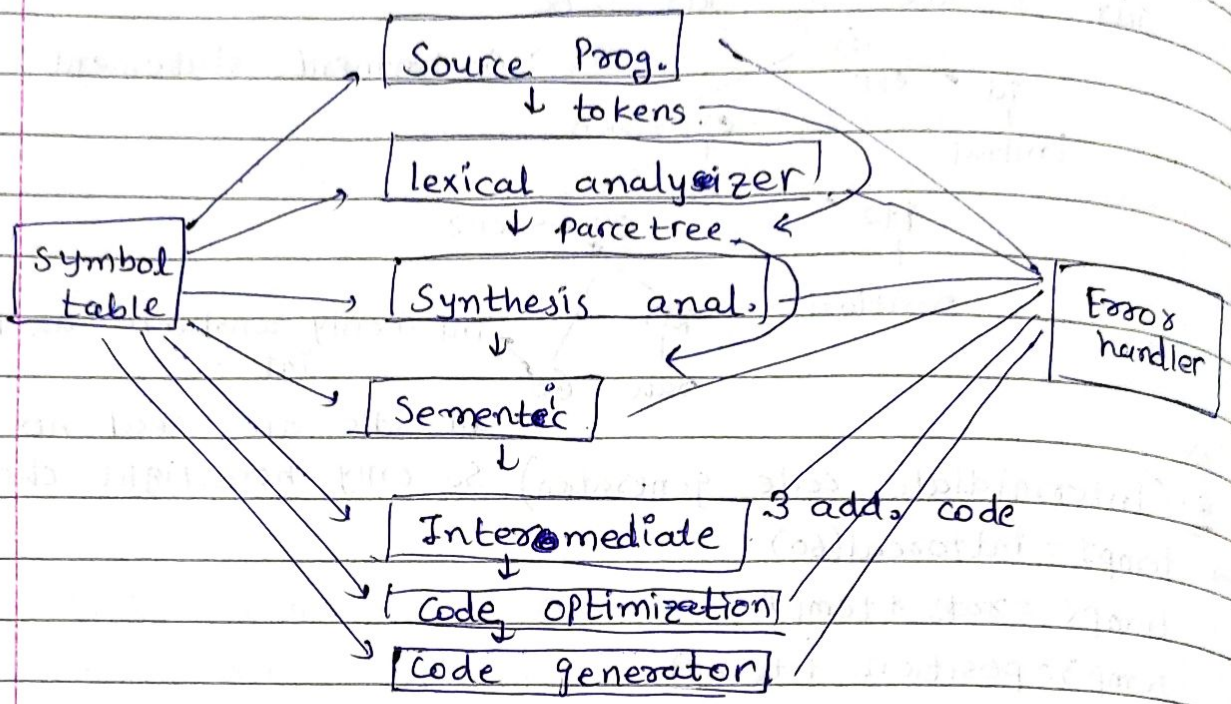
(1) optimization

Intermidiate Code ⟹ temp1 = id3 * 60.0
id1 = id2 + temp1

(2) Code generation

```
MOVF  R1, id3
MULF  R1, #60.0
MOVF  R2, id2
ADDF  R1, R2
MOVF  id1, R1.
```

* Phases of Compiler.

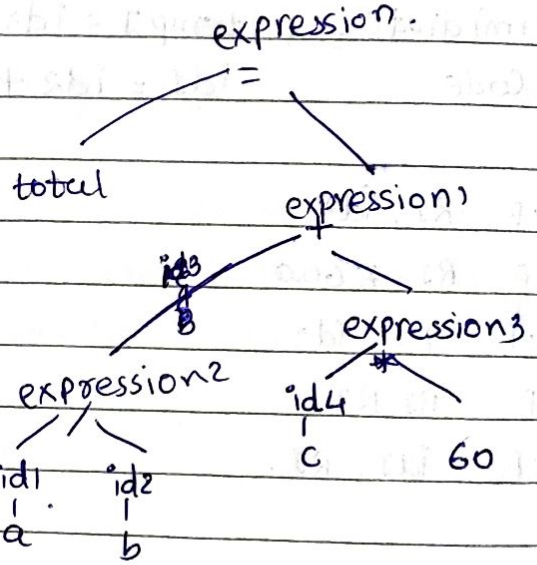


Q. Total = a/b + C*60

1) Token generation

Total → identifier
= → assignment symbol
a → identifier
/ → divide sign (symbol)
b → identifier
+ → Plus sign (symbol)
C → identifier
* → mul. sign (symbol)
60 → int (constant)

2) parse tree



3) Intermediate code generator.

temp1 = a/b → temp4 = ~~int~~ inttoreal(60)
temp2 = C* ~~60~~ temp4
temp3 = temp1 + temp2
Total = temp3

→ Code optimization.

```
temp2 = C * 60.0
Total = temp1 + temp2.
```

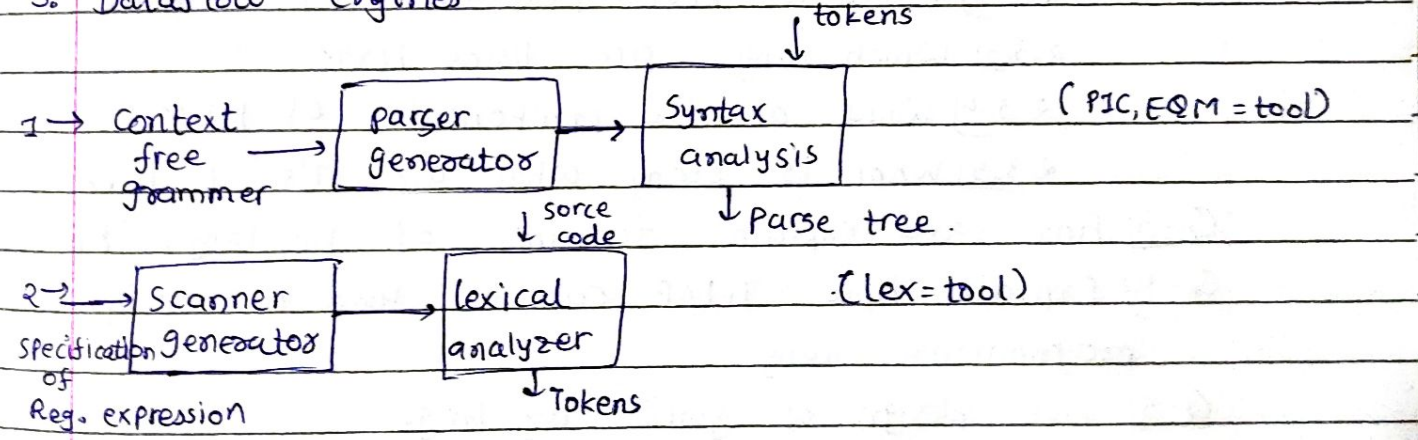
→ Code generation.

```
MOVF  R1, a
MOVF  R2, b
DIVF  R1, R2
MOVF  R2, C
MULF  R2, #60.0
~~MOVF~~ ~~Total~~ total, ADD R1, R2
MOVF  id1 , R1
```

* Compiler construction tools.

*→ Rules.

1. Parser generator.
2. Scanner generator.
3. Syntext directed translation engines. (generate intermediate code)
4. Autometic code generators.
5. Dataflow engines



13/12

Lab 1. Aim = Study compiler and various related tools.

Q-1] Write a simple prog. in C/C++ lang. and observe compiling, linking & loading process using GCC compiler (Find difference in size of files generated) (Find which files generated after each step

Q-2] Study & ans. the following Que.

2.1] Discuss following

compiler, interpreter, Likers, loaders

2.2] GCC

2.2.1] What is GCC, explain it's features

2.2.2] In which lang. is GCC written

2.2.3] What standard optimizations are supported by GCC

2.2.4] Explain following options of GCC

-O, -S, -c, -save, -g, -v, temps, -c -static

size <exefilename>

objdump -hrt <objfile>

2.3] LLVM

2.3.1] Understand & discuss following keyword

OpenCL, OpenGL, openmp, clang, LLVM, Bison.

2.3.2] What is LLVM, explain it's feature

2.3.3] Which lang. are using LLVM

2.3.4] What are the components of LLVM

2.3.5] What is clang, what are it's features

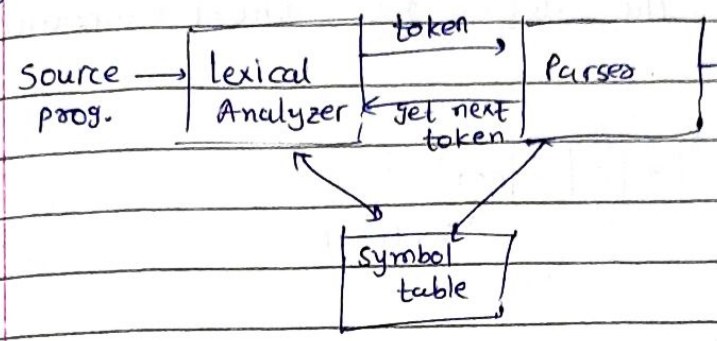
Q-3] Find and explain features of toy lang. "Bhailang"

Q-4] Explain how IFLAP can be used to design programming lang.

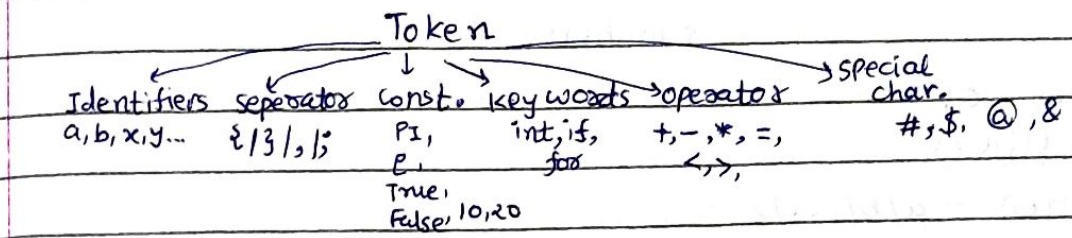
Q-5] Give design of your toy lang.

14/12

(*) The role of lexical analyzer.



→ How lexical Analyzer works?
(will give token)
lexeme → Tokens
↓
How it works
↳ int a,b → after each char. it will check for tab, space, new line, so lexeme will stop. ~~cattere~~ for (i, n, t ↵)



Q.
```
int main(){
    int a=20, b=30;
    if(a<b)
        return b;
    else
        return a;
}
```

special char =
Identifier = a, b
Seperator = { } , ( ) , ; , (2) (2) (3)
const. = 20, 30
operator = <, = (2)
keyword = int, if, else, main, return (2)

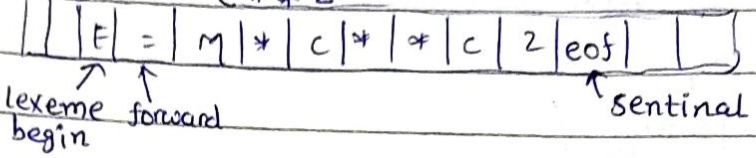
→ printf("i=%d, &i=%f", --)
b/w "" is one lexeme

→ It eliminates comments & white space (tab, space, new line)
→ if no relation b/w char, found token and return (a*b)
→ How it will handle error
1) exceeding length.
2) Unmatched string. → /* --- */
3) illegal char.

* Sentinals :-

- buffers - It stores the data for short amount of time

eg. E = M * C * * 2 .



* Recognization of token.

1- Recognization of identifiers,
2- " " delimeter
3- " " relational operators,
4- key words
5- numbers

* Identifiers

Recog. of

- letters → a/b/.../z
  A/B/../z
- digit → 0/1/../9

RE → id → letter(letter/digit)*

** → delim → blank / tab / newline

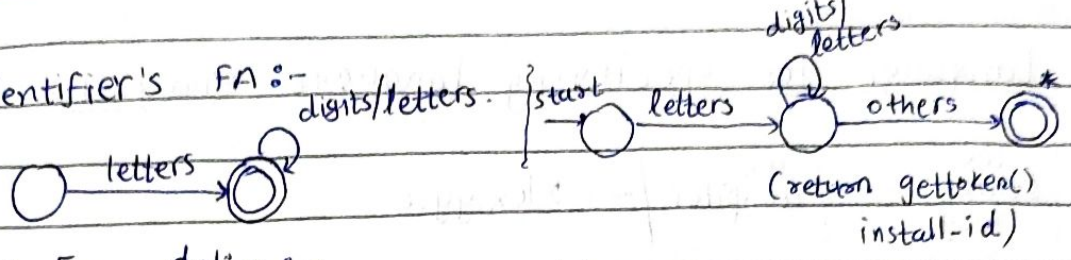
RE → ws → delem+

Q:- The tokens and associated attribute values for the statement E = M * C * * 2

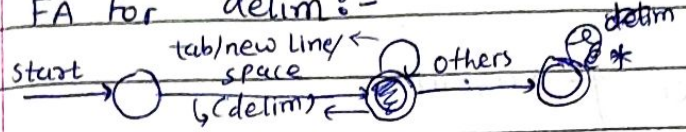
E <id, pointer to sym. table entry for E>
= <assignment op, assignment op. or blank>
M <
* < mult-op, >
C <
** < exponential-op, >
2 < constant/Num, >

15/12

→ Identifier's FA :-

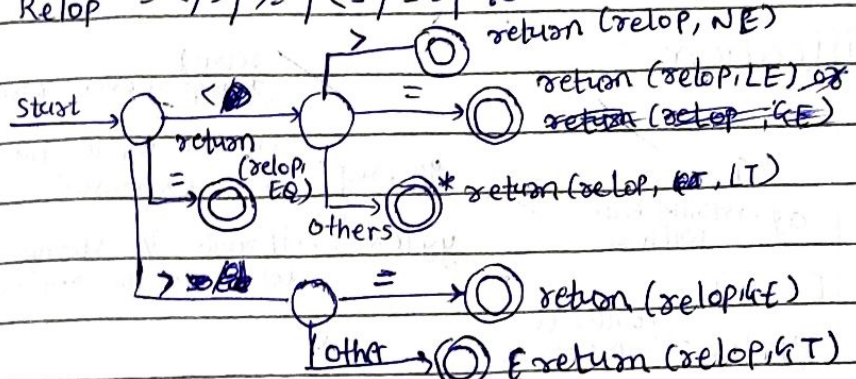


→ FA For delim :-

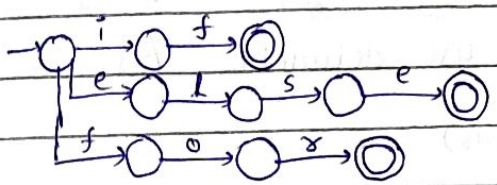


*→ Recognization of Relational Algebra.

Relop = </>/>=/<=/==/!=

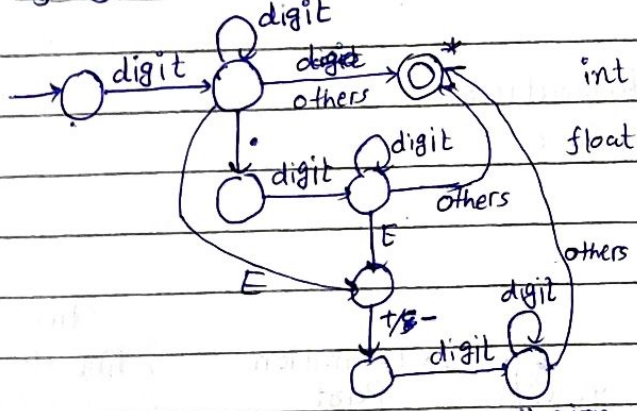


*→ Recognization of keywords (if, else, for ...)



*→ Recognization of Numbers

num → digit+ (.digit)*



int = digit (digit) other

floating: digit (digit) . digit (digit) other

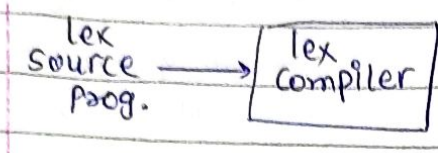
→ num → (digit)+ (.(digit)+)?

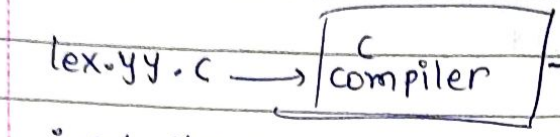
num → (digit)+ (.digit)+ ? (E(+/-)? digit+)?

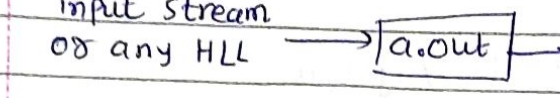
16/12

* Language For specifying Language Analyzer.

Lex :-







* Lex Specifications.

abc	^a	return 0 if I/P is over else 1
[a-z]	a$ → string ends with a	yywrap() → it call by lex for when I/P was exhausted
[a-z]*	[0-9]+	yylex() → it reads I/P string & generates tokens to the reg. expression
[A-Z, a-z]+	yytext → Pointer to input string	

→ declaration

after declaration we will use delimeters. %{ ..... %}

%{ ..... %}

%%

transition rule → Pattern (actions)

%%

auxiliary procedures

```
%{
   #include <stdio.h>
   /* ..... */ ← comment.
%}
%%
"hii" { printf ("How are you!"); } → transition Part
.* { printf ("Wrong String"); }
%%
```

declaration Part

other (else like)

(like if-else)

hii → How are you

will print

or wrong string will be print

```
main()
{
    printf("Enter i/p"); } i/p read
    yylex();
}
int yywrap(); } generate token until EOF it will return 0, at EOF return 1.
{ return 1;}
```

h | i | i | EOF

2012 Q. Find lexemes, tokens & patterns of the following que.

→ int x=5;

lexemes	token	Pattern
int	keyword → int ← keyword	keyword
x	identifier	letter(letter/digit)
=	operator	=, <, >
5	constant	constant
;	separator	Seperator

Q Find out lexical errors

~ Int x; → Not lexical error.

⇒ lexical error Rules

1. - exiding length of identifier → int x=12468102526209985;
2. - Appear of illegal char. → printf("6th - I"); $
3. - unmached string → String s="abc / s=abc" (not both "")
4. - Identifiers start with letters → int 6a=25
5. - Replacing a char with an incorrect char → int x=1$2
6. - White space b/w char → int a b=25

~ int x,y; → Not error

~ float m=5.7 → Error → 5

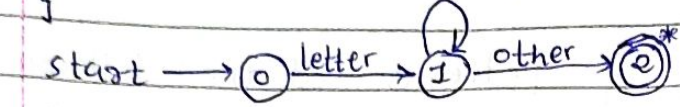
~ int 1sal; → Error → 4

~ Int A=#12; → Error → 5

~ String s="success" → Error → 3

→
```
#include<stdio.h>
main(){
  int x=5, y=6;
  char *a;
  a=&x;
✓ x = &abi
  printf("%d", x);
}
```

eg-1

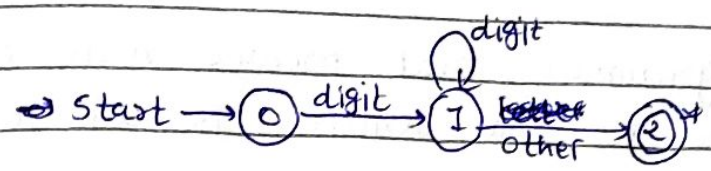



```
int start=0,
token nexttoken(){
  while(1){
    switch(state){
      case 0:
        c=nextchar();
        if(isletter()) state=1
        else state=fail
        break;
      case 1:
        c=nextchar();
        if(isletter())
            state=1;
        else if(isdigit()) state=1;
        else state=2;
        break;
      case 2:
        retract(1);        // retract is a look ahead pointer | char
        install_id();      // It is used bcoz, delimeter is not a
                           // part of identifier.
        return gettoken(1);   // it is used to access the buffer & it
                              // called as attribute val.
                  → // to obtain a token
    }
  }
}
```

```
int fail(){
  switch(start){
    case 0:
      start=0;
      break;

default:
/* compiler error */
}
return start;
}
```

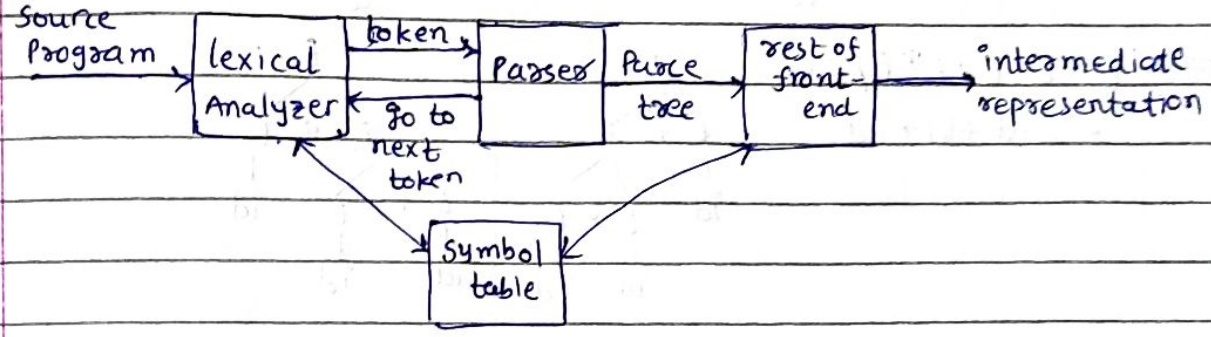
eg=2



21/13

(2) * Syntext Analysis

→ Role of the parser



* Context free grammar.

- Context free grammar is a four tuples.

→ $G = (V, T, P, S)$

- V: Finite set of symbol called as non terminal or variable. (It may be in uppercase) (usually S is used for starting symbol, Some italic names are also allowed)

eg. expr stmt

$S \rightarrow 0A$

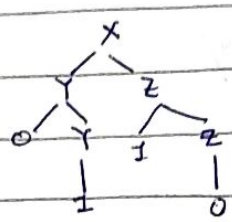
$A \rightarrow 0B$

$B \rightarrow \epsilon$

- T: set of symbol that are called as terminal (lowercase letters, operator symbols → +/-, punchuation symbols, digits, bold phase string → id, if)
- P = set of productions. ($\alpha \rightarrow \beta$)
- S: It is member of V, called as start symbol.

⇒ $X \rightarrow YZ$
$Y \rightarrow 0Y/1$
$Z \rightarrow 1Z/0$

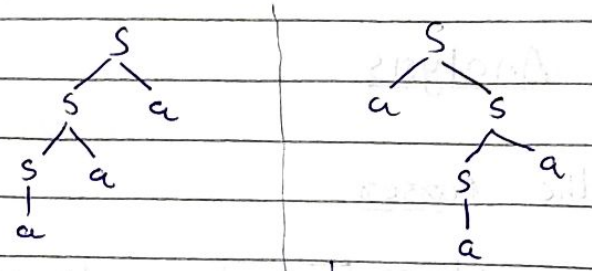
ω: 0110



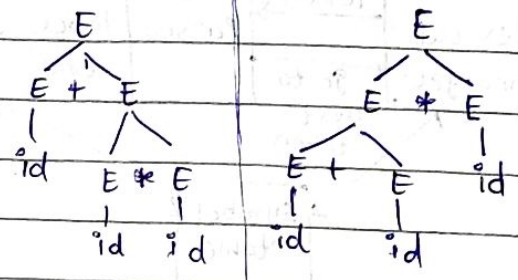
→ No other parce tree is possible for ω
→ So it is non-ambiguous

→ Ambiguity: A grammar that produce more one parce tree for same sentence is said to be ambiguous grammar.

⇒ $S \rightarrow Sa/aS/a$
ω: aaa



⇒ $E \rightarrow E+E/E*E/id$
ω: id + id * id



Rules:- (to remove ambiguity)

→ $ and ↑ is right associated, remaining all are left associative

→ Priority: $ < # < @

→ or < and < not.

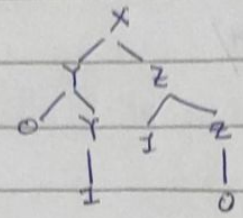
→ $- < + < / * < / < ↑ < $

$+ < * < \bigcirc^*$

- T = set of symbol that are called as terminal (lowercase letters, operator symbols → +/-, punchuation symbols, digits, bold phase string → id, if)
- P = set of productions. ($\alpha \to \beta$)
- S = It is member of V, called as start symbol.

⇒ $X \to YZ$
$Y \to 0Y/1$
$Z \to 1Z/0$

ω: 0110

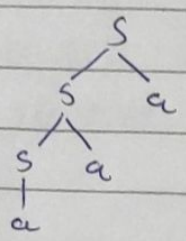


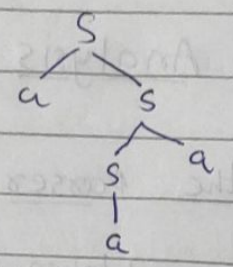
→ No other parce tree is possible for ω

→ So it is non-ambiguous

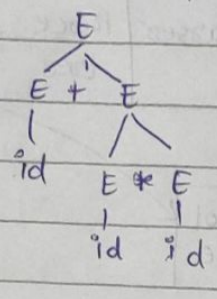
→ Ambiguity: A grammar that produce more one parce tree for same sentence is said to be ambiguous grammar.

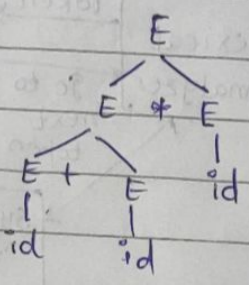
⇒ $S \to Sa/aS/a$
ω: aaa





⇒ $E \to E+E/E*E/id$
ω: id + id * id





Rules:- (to remove ambiguity)

→ $ and ↑ is right associated, remaining all are left associative

→ Priority:
$\$ < \# < @$

→ or < and < not.

→ $- < + < \not * < / < \uparrow < \$$

$+ < * < \bigcirc^*$

eg. $E \to E+E / E*E / id \longrightarrow E \to E+T / T$

$T \to E T*F / F$

$F \to id$

$E \to E*E / E+E / id \to E \to E+F / F$ (+ has low priority so, start with E+E)

$F \to F*T / T$

$T \to id$

eg. $E \to E\$E / id \longrightarrow E \to E\$T / T \curvearrowright E \to T\$E / T$ ($ is right associative so T will go to left

$T \to id$

eg. $A \to A\$A / A\#A / A@A / d \longrightarrow A \to T\$A / T$

$T \to T\#P / P$

$P \to P@N / N$

$N \to d$

22-12

① bExp → bExp or bExp / bExp and bExp / NOT bExp / True / False

② $R \to R+R / RR / R^* / a / b / c$ ✓ Ans → $R \to R+M / M$

$M \to MS / S$

$S \to S^* / a / b / c$

$E \to$

$T \to$

$F \to$

$E \to E+T / T$ ×

$T \to TF / F$ ×

$F \to F^* / a / b / c$ ×

Ans → bExp → bExp or T / T

T → T and F / F

F → F not M / M

M → True / False

* Left Recursion

→ Algorithm

input will take some grammar G, then o/p equvallent grammar with no recursion.

→ Method

If we have left recursive pair of production

$A \to A\alpha / \beta$, where β doesn't begin with A

then we can eleminate left recursion by

replacing this pair of production with

$A \to \beta A'$

$A' \to \alpha A' / \varepsilon$

eg. $A \to A\alpha / \beta / \gamma$

$\Rightarrow A \to \beta A' / \gamma A'$

$A' \to \alpha A' / \varepsilon$

There are 2 type of left recursion

1. direct left recursion
2. indirect left recursion

1. eg $A \to AB/d / Aa/a$

$B \to Bc/b$

→ No ambiguty

Ans: $A \to [illegible] aA'$

$A' \to aA' / BdA' / \varepsilon$

$B \to bB'$

$B' \to cB' / \varepsilon$

eg: $A \to AaB/x$

$B \to BCb/y$

$C \to Cc/\varepsilon$

→ No ambiguty.

Ans: $A \to xA'$

$A' \to aBA' / \varepsilon$

$B \to yB'$

$B' \to CbB' / \varepsilon$

$C \to cC' / \varepsilon$ } $C \to c'$, $C' \to cC' / \varepsilon$

23/12

* Indirect Left recursion

$S \to Aa/b$

$A \to Ac/Sd/\varepsilon$

→ $A \to Ac / Aad / bd / \varepsilon$.

$A \to bdA' / A'$

$A' \to cA' / adA' / \varepsilon$

* Left factory - Algo. for removing. I/p: Take grammar g. will

O/p: Eqivalent non left factory grammar

method: For each non-terminal A find the prefix α common two or more of its alternatives. Replace one of the A production

~ $A \to \alpha\beta_1 / \alpha\beta_2 / \alpha\beta_3 / \alpha\beta_n / \gamma$

- where γ represents all alternatives that do not begin with α

$A \to \alpha A' / \gamma$

$A' \to \beta_1 / \beta_2 / \beta_3 / \dots \beta_n$

- Repeatedly apply, this transformation until those two alternatives for a common prefix

eg. $S \to iEts / iEtses / a$
$E \to b$

Ans → $S \to iEtSS' / a$
$S' \to \varepsilon / es$
$E \to b$

eg. $A \to xByA / xByAzA / a$
$B \to b$

Ans → $A \to xByAA' / a$
$A' \to \varepsilon / zA$
$B \to b$

eg. $A \to aAb / aA / a$

Ans → $A \to aA'$
$A' \to Ab / A / \varepsilon$
$A' \to AA'' / \varepsilon$
$A'' \to b / \varepsilon$

eg. $A \to ad / a / ab / abc / b$

Ans → $A \to aA' / b$
$A' \to d / \varepsilon / b / bc$
$A' \to bB' / d / \varepsilon$
$B' \to \varepsilon / c$

eg. Ambiguity → left recursion → left factory

$E \to E+E / E*E / id$
$E \to E+T / T$
$T \to T*F / F$
$F \to id$

$E \to +TE'$
$E' \to +TE' / \varepsilon$
$T \to FT'$
$T' \to *FT' / \varepsilon$
$F \to id$

$A \to A\alpha / \beta$
$A \to \beta A'$
$A' \to \alpha A' / \varepsilon$

27/12

Parser

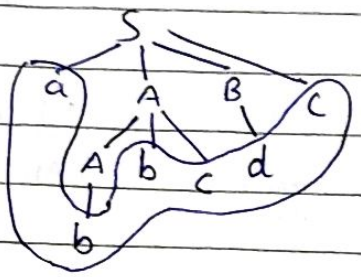
- Top-Down Parser
  - Brute force method
  - Recursive decent Parser
  - Non-Recursive decent Parser
- bottom-up Parser

* Top-Down Parser.

Generates a parse tree for the given input stream with the help of grammar productions by expanding the non-terminals.

- In this, if we start from start symbol & ends on the terminal. It uses left most derivation.

eg. $S \rightarrow aABc$
$A \rightarrow Abc/b$
$B \rightarrow d$
string = abbcde

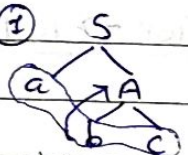


$S \rightarrow aABc$
$\rightarrow aAbcBc$
$\rightarrow abbcBc$
$\rightarrow abbcdc$

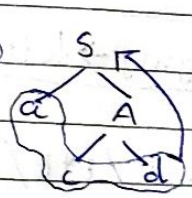
- at every point we have to ~~design~~ decide what is the next production we should use

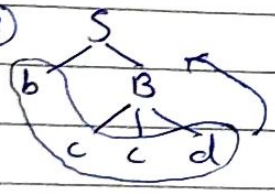
* Brute force Parser

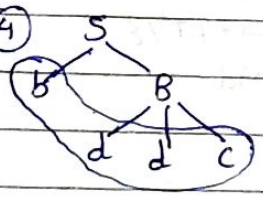
eg. $S \rightarrow aA/bB$
$A \rightarrow bc/cd$
$B \rightarrow ccd/ddc$
string = bddc

(1)


Not string generated, so backtrack

(2)


(3)


(4)


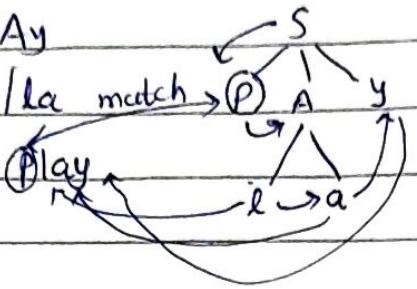
drawback → Time complexity
2^n

* Recursive decent Parser

eg. $S \to PAy$

$A \to a / la$

String = Play



eg. $E \to iE'$

$E' \to +iE' / \varepsilon$

① EC)

② E'C)

③ match(token t)

→ eg. i+i$

→ EC){

```
if(lookahead == 'i'){
    match('i');
    E'();
}
if(lookahead == '$'){
    print("sucess");
}
else
    return;
}
```

→ E'C){

```
if(lookahead == '+'){
    match('+');
}
if(lookahedd == 'i'){
    match('i');
    E'();
}
else return;
}
```

eg. $S \to iEtS / iEtSeS / a$

$E \to b$

→ Ambiguous

$S \to iEtS \to iEtiEtSeS$

$\hookrightarrow iEtSeS \to iEtiEtSeS.$

eg. $S \to ABC$

$A \to oA / \varepsilon$

$B \to 1B / \varepsilon$

$C \to 1C / \varepsilon$

S(), A(), B(), C(), match(token)