Compiler design

* **Compiler:** Compiler is a S/W which converts a program written in HLL to LLL

Source lang (HLL) → [Compiler] → LLL.
↑
Compilation error.

* **Language processing system**

HLL
↓
pre processor — removes #include #define
↓
Compiler — pure HLL. HLL – LLL
↓
Assembler — LLL – M/C.
↓ — relocatable m/c code
loader/linker — library prog
↓
machine code

Eg:-

C program (HLL)
⇓
C compiler translates prog into (LLC)/(Assembly lang)
⇓
Assembler translates (LLL) into (m/c) code.
⇓
linker is used to link all parts of prog together for execution
⇓

loader loads all of them into memory and then the program is executed.

Note: A compiler converts HLL to LLL.
Assembler converts LLL to MLL.

Statements {HLL}
⇓
Mnemonics {ALL}.
⇓
Binary no's. {M/c code}.

* PHASES OF COMPILER.

→ Phase :- is a logically interrelated operation that takes source program in one representation and produce output in another representation

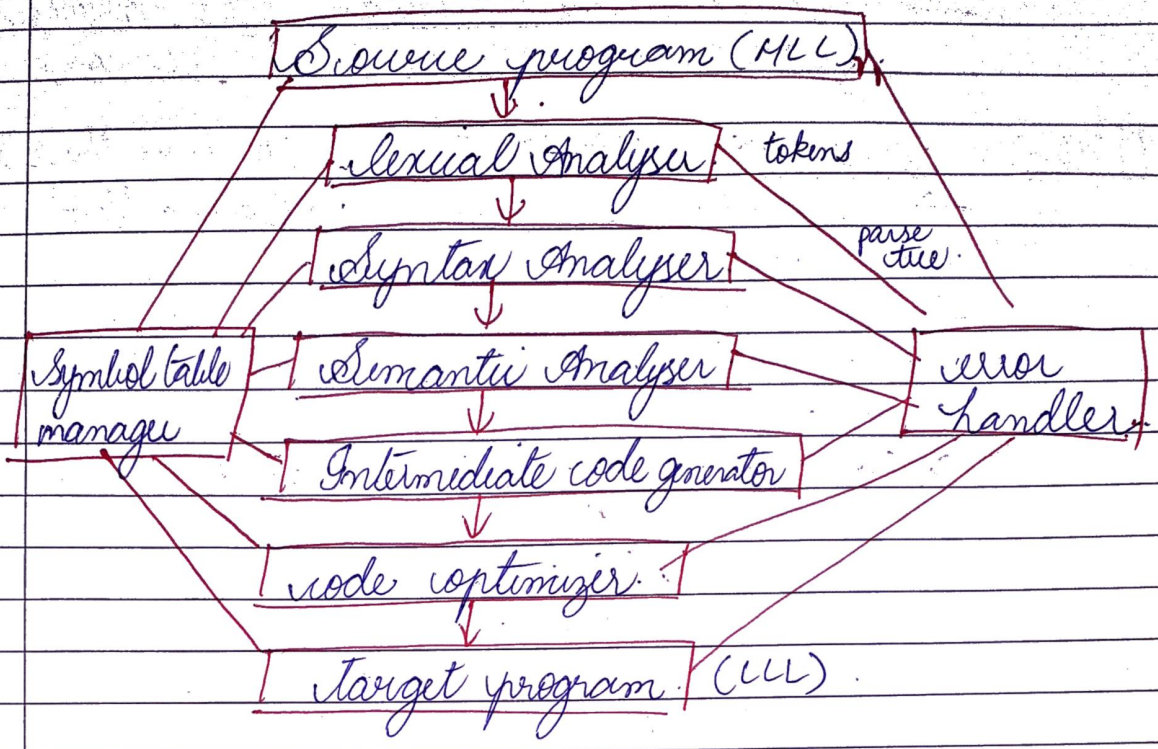
→ There are 2 major phases
a) Analysis phase - (M/c indep / lang dep)
b) Synthesis phase (M/c dep / lang indep)

→ There are 6 phases

→ All these phases are managed by Symbol table manager

→ If any error occurs in between these phases, the error handler is going to handle that error

→ The phases of compiler is also called Structure of the compiler



Phase 1: Lexical Analysis:

→ The LA reads the stream of char making up the source prog. & group the char into meaningful sequences called lexeme.

→ Lexical Analyser represents these lexemes in the form of tokens.

→ < token name, attribute value >

→ eg: new value := old value + 12

here,

the tokens are.

new value	:	identifier
=	:	Asst operator
old value	:	identifier
+	:	Add operator
12	:	number

→ The LA also truncates the white spaces and removes the errors

Phase 2 SYNTAX ANALYSIS (Analysis Phase)

→ Syntax Analysis is also called as parsing

→ It takes tokens which is produced by lexical as i/p and generates a parse tree / Syntax tree.

→ Eg: new val := (add val + 12)

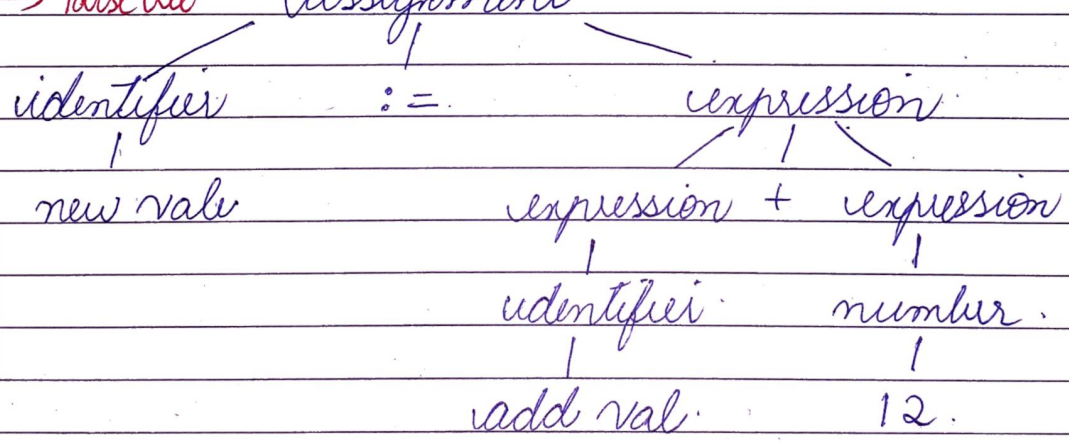
⇓

tokens

⇓

Syntax tree

→ Parse tree



Phase 3 SEMANTIC ANALYSIS- (Analysis Phase)

→ It checks whether the parse tree constructed, follows the rules of language

→ Eg: assignment of values is b/w compatible data types.

→ Also, the SA keeps track of identifiers, their types & expressions.

Eg: Sum = a+b.	Semantic record.
int a;	a: int
double sum;	sum: double
char b;	b: char

→ Here, the datatypes are mismatched as a is an integer & b is a character and if you add integer & char, you wont get the double.
→ Therefore we can say that it is syntatically correct but semantically incorrect.

Phase4 Intermediate Code Generator (Syn Phase)
→ The ICG is the representation of final m/c lang code is produced.
→ This phase bridges the analysis & synthesis phase of translation.
→ Eg: new val := old val + fact * 1.

⇓

id1 := id2 + id3 * 1.

⇓

Temp1 = int real (1);
Temp2 = id3 * temp1;
Temp3 = id2 + temp2;
id1 := temp3.

Phase5. Code Optimization

→ It is used to improve the intermediate code.
→ So that the o/p runs faster &

takes less space

→ eg: new val := old val + fact * 1.

```
Temp1 = id3 * 1.
id1 = id2 + temp
```

Phases

Code generation :

→ It translates the intermediate code into sequence of relocatable m/c code

→ id1 := id2 + id3 * 1.

```
MOV  R1, id3
MUL  R1, #1.
MOV  R2, id2.
ADD  R1, R2.
MOV  Id1, R1.
```

} Assembly level lang / Low level lang

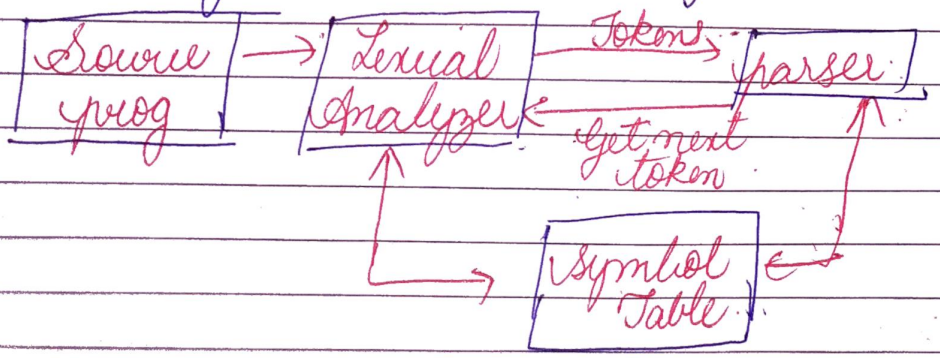
→ ∴ Here, the compiler converts HLL to LL

---

* LEXICAL ANALYSER :

→ LA is the first phase of the compiler also called as linear or scanning.

→ In this phase the stream of char making up the program is read from left to right and grouped into tokens that are sequences of character having collective meaning.



→ Upon receiving a "get next token" command from the parser, the lexical analyser reads the i/p char until it can identify the next token.

→ LA may also perform certain secondary tasks as user interface

→ One of such task is striping out from source program, the commands & white spaces in the form of blank, new line char's.

→ Another is co-relating error messages from the compiler to the source code

* CROSS COMPILER:-

→ A cross compiler is a type of compiler that generates machine code targeted to run on a system different that the one generating it.

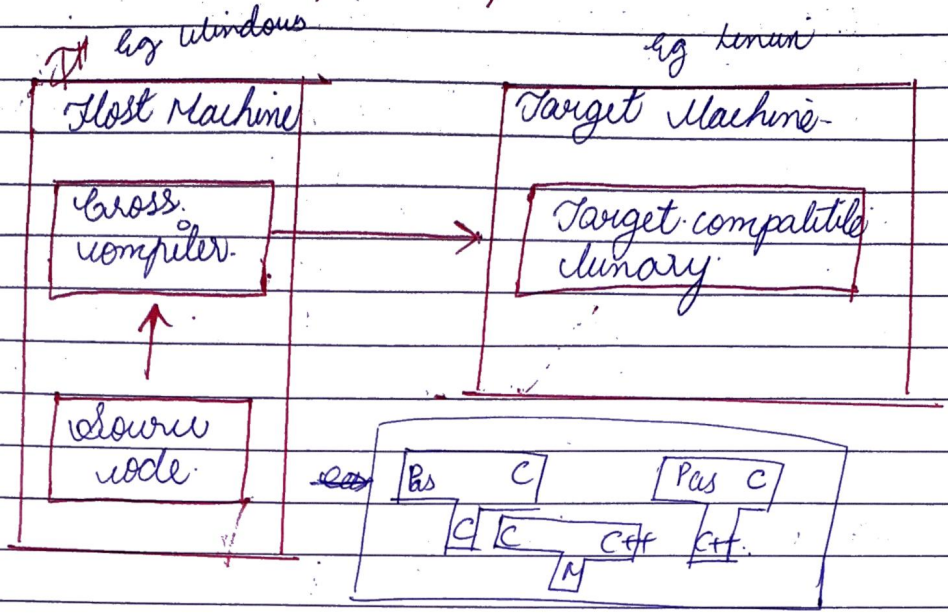
→ eg:- A compiler that runs on windows platform also generates a code that runs on linux platform is a cross compiler.

→ The process of creating executable code for diff m/c is also called "retargetting"

→ The cross compiler is also known as "retargeting compiler".

→ GNU GCC is an example for cross compiler

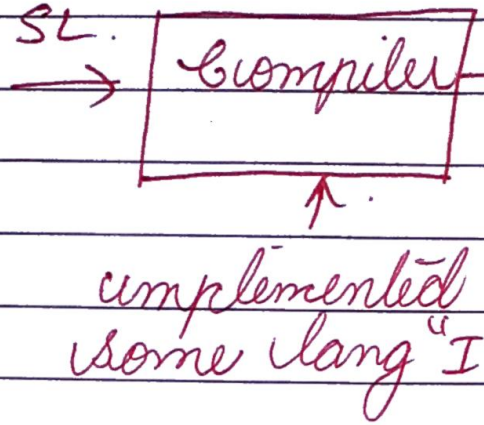
→ Cross compiler operation

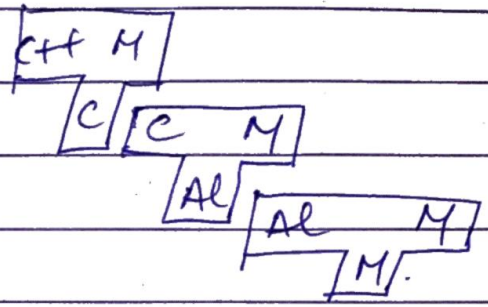


* Bootstrapping

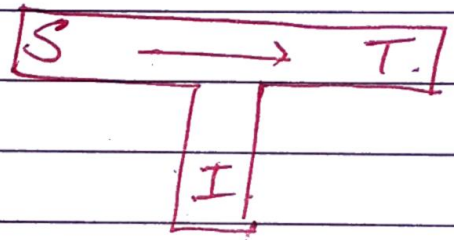
→ Bootstrapping is widely used in compilation development.
→ It is used to produce a self-hosting compiler.
→ A self-hosting compiler is a type of compiler that can compiles its own source code.
→ A compiler can be characterized by three languages.
1. Source language (S). (HLL)
2. Target language (T) (ALL)
3. Implementation language (I) (Intermediate)
→ Bootstrapping is the process by which simple language is used to translate more complicated program

which in turn may handle an even more complicated program so on.





→ In bootstrapping this compiler diagram is represented as 'T' diagram.



∴ T

(or) S_I.

letters ⇒ non terminals
+, -, %, * ⇒ terminals

__/__/__

Operator Precedence parsing

→ Any grammar G is called an operator precedence grammer if it meets the following two conditions.
1. There exist no production rule which contain ∈ on right hand side.
2. There exist no production rule which contains 2 non-terminal adjacent to each other on its right hand side.

⇒ A parser that reads & understands an operator precedence grammer is called operator precedence parser

⇒ Example: Which is not an operator precedence grammer?

$$E \rightarrow EAE \mid (E) \mid -E \mid id.$$
$$A \rightarrow + \mid - \mid * \mid / \mid \wedge.$$

→ Example2: Which is an operator precedence grammer?

$$E \rightarrow E+E \mid E-E \mid E*E \mid E/E \mid (E) \mid -E \mid id.$$

→ Operator precedence can only be established between the terminals of the grammer. It ignores the non terminals of the grammer.

*Parsing Actions Rules.

→ both end of the given i/p string, add $ symbol.
→ Now scan the i/p string from left to right until > is encountered

→ Scan towards the left over all the equal precedence until the first left most < is encountered.
→ Everything b/w left most < and right most > is handle.
→ $ on $ means parsing is successful

* There are three operator precedence relations.
a > b ⇒ Terminal a has higher precedence than b
a < b ⇒ Terminal a has lower precedence than b.
a = b ⇒ Terminal a and b have same precedence

* Precedence Table

	+	*	(	)	id	$	Rule
+	>	<	<	>	<	>	id, a, b, c ⇒ high
*	>	>	<	>	<	>	$ ⇒ low
(	<	<	<	=	<	×	+ > +
)	>	>	×	>	×	>	* > *
id	>	>	×	>	×	>	id ≠ id
$	<	<	<	×	<	×	$ ≠ $

* Example:

Role of Lexical Analyser

1. Lexical Analyser functions (OR) Role of Lexical Analyser. (SCANNER)
2. Seperation of Lexical Analyzer (PARSER) from Syntax Analyser
3. tokens, lexemes and patterns
4. Lexical errors

1. → Lexical Analyser reads the source program and produces a stream of tokens
→ It removes

INPUT BUFFERING

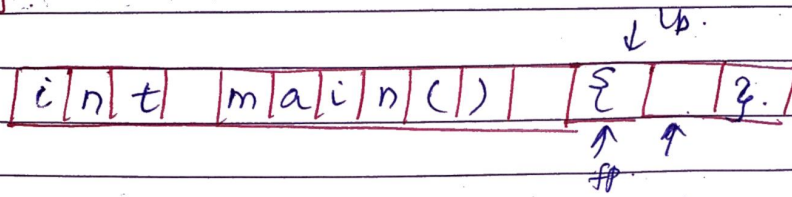
So in order to read each token, 2 pointers are used.
1. lexeme begin pointer
2. forward pointer

1. → Lexeme begin pointer points towards the beginning character of the current lexeme
2. → The forward pointer reads the 1st character, then moves ahead. Whenever the forward pointer reaches the blank space, it identifies as tokens end.
→ After that, the forward pointer ignores the blank space & plays

the lexeme begin & forward pointer in next character location.

eg
```
int main()
{

}
```



→ We know that the corresponding program is stored in hard disk so at a time we can reach only one character from the hard disk.

→ If the program size is very large, then we require too many system calls in order to read the data from the hard disk

→ So, we can overcome this problem with the help of Input buffering techniques

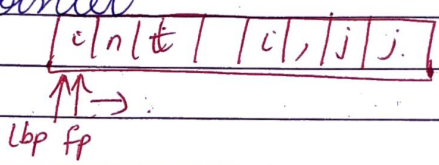
Buffering :- (USE).

Buffering means, instead of reading a single character from the hardisk, a block of characters are to be reached into the buffer simultaneously with the help of a single sys call.

Buffering :- (IMPLEMENTATION)

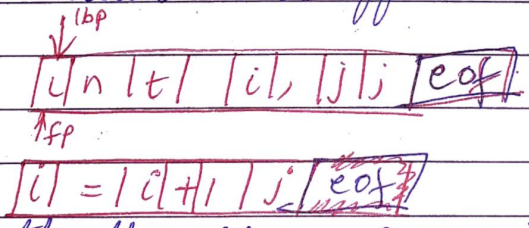
Buffering can be implemented in 2 ways
1. One buffer scheme.
2. Two buffer scheme.

1. → One buffer scheme means to only use one buffer inorder to store the given i/p string
→ Here also we use 2 pointers, i.e lexeme begin pointer & forward pointer



→ The problem here is if the i/p string size is larger than the capacity of the buffer then we have to override the buffer inorder to store the remaining i/p string.

2. → In a 2 buffer scheme, we can maintain 2 buffers.



→ Both the lbp & fp points towards the first beginning character of the first buffer
→ The first buffer is filled completely then we have to refill the second buffer

→ In order to determine whether the first buffer is completely filled, a special character called eof or sentinal character is inserted in the last position.

→ second buffer

→ eof is not the part of the source program

→ whenever the second buffer is completely filled then the remaining i/p string has to be filled in the 1st buffer.

→ Then the first buffer will be overwritten

Algo

```
switch (*forward++)
{
case 'eof':
    if (forward is at end of 1st buffer)
    {
        forward = begining of 2nd buffer;
        refill 2nd buffer
    }
else if (forward is at end of 2nd buffer)
    {
        forward = begning of 1st buffer;
        refill 1st buffer
    }
else
    break;
cases for other char;
}
```

SPECIFICATIONS OF TOKENS

→ Regular expressions are used to specify tokens

1. **Alphabets :-** It is a set of symbols
   $\Sigma = \{0,1\}$ It is a binary alphabet.
   $\Sigma = \{a, b \cdots z\}$ It is a set of lower case letters

2. **String :-** It is a finite set of symbols generated from Σ (Input alphabet)
   $\Sigma = \{a, b\}$
   a, b, ab, ba, aab .....

   a) **Length of a string :-** Total no of characters present in the string.
   $S = 1100$.
   $|S| = 4$.

   b) **Empty string :-** If lt of the string is 0 then we call it empty string
   $\epsilon = 0$.

   c) **Prefix of a string :-** It is any no. of leading symbols in the string
   let $S = abc$.
   prefix :- ϵ, abc, a, ab

   d) **proper prefix of a string :-** Except ϵ & input string
   let $S = abc$.
   proper prefix :- a, ab.

e) Suffix of a string :- It is any no of trailing symbols in the string.
let S = abc.
suffix = ∈, abc, c, bc

f) proper suffix of a string :- Except ∈ & string, the remaing
let S = abc
proper suffix = c, bc.

g) Substring :- It is obtained by deleting prefix & suffix from the string.
let S = banana.
Substring :- ∈, banana, nan, anan, nana

h) proper substring :- Except ∈ & string itself, the remaining
let S = banana.
proper substring nan, anan, nana.

i) Concatenation of 2 strings :-
Combining 2 strings.
let x = abc
y = de.
xy = abcde.
yx = deabc.

3. Language :- Set of strings which are generated from the alphabet
$\Sigma = \{a, b\}$.
$L = \{a, b, ab, ba, abb \ldots\}$

Operations on languages

a) Union of 2 languages ($L \cup M$) :- To write both the languages

$L \cup M = \{ s \mid s$ is in L or s is in $M \}$

$L = \{0, 1\}$ $M = \{00, 11\}$.

$L \cup M = \{0, 1, 00, 11\}$

b) Intersection of 2 languages ($L \odot M$)

To concatenate one string from L and other from M.

$L \odot M = \{ xy \mid x$ is in L and y is in $M \}$

$L = \{0, 1\}$ $M = \{00, 11\}$

$LM = \{000, 011, 100, 111\}$

c) Kleene Closure :- (L^*)

A set of strings which include empty strings

$$L^* = \bigcup_{i=0}^{\infty} L^i$$

$= L^0 \cup L^1 \cup L^2 \ldots$

$\Sigma = \{a\}$.

$\Sigma^* = \epsilon, a, aa$.

It is also called as star closure

d) positive closure (L^+)

A set of strings except null string

It starts from 1.

$$L^* = \bigcup_{i=1}^{\infty} L^i.$$

$L^1 \cup L^2 \ldots$

$\Sigma = \{a\}$

$\Sigma^+ = a, aa$:

4. Regular Expression :- a) A RE over ϵ can be defined as.

→ ϕ is a regular expression for empty set.

→ ϵ " " " " " null string $\{\epsilon\}$

→ If 'a' is a symbol in ∈ then 'a' is RE.
for {a}

→ If R & S are 2 RE then,
a) Union of 2 RE is a RE
b) Concatenation of 2 RE is a RE
c) Kleene closure of any RE is a RE.

*The above 3 are known as properties or components of a RE

b). Algebraic rules for RE / Identity Rules.

1. $R+S = S+R$
2. $(R+S)+\epsilon = R+(S+t)$
3. $(RS)t = R(St)$.
4. $(R+S)t = Rt+St = (S+R)t = St+Rt$
5. $\epsilon R = R.E = R$.
6. $R^* = (\epsilon+R)^*$
7. $R^{**} = R^*$

c). Meta Characters or patterns of RE.

meta char.	description
1. x	matches with char x.
2. .	matches with any char except newline
3. R*	0 or more occurances of R.
4. R+	1 or more " " "
5. R?	0 or 1 " " "
6. ^	matches with beginning of line
7. $	matches with end of line.
8. R1/R2	either R1 or R2.
9. [abc].	a, b, c

d). Regular definition - It is a name given to the RE & we can use that name later.
It contains production rules which is of the following form.

$d_1 \rightarrow r_1$
$d_2 \rightarrow r_2$
$d_3 \rightarrow r_3$

Ex①:- Regular expression for identifier

letter → a|b|c ... z |A|B|C|..|Z|.
digit → 0|1|2|... 9.
id → letter (letter / digit)*
(OR) id → [a-z A-Z] [a-zA-ZA 0-9]*

Ex②:- Regular Expression for digits (int no, floating point no)

digit → 0|1|... |9
digit → digit (digit)*
number → digits (.digits)? (E[+-]?digits)?

RECOGNIZATION OF TOKENS

→ Tokens are recognized with transition diagrams

~~Recognizer~~

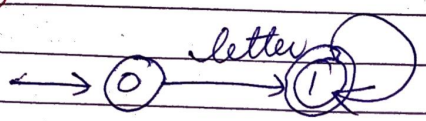
1. Recognization of Identifiers

Identifiers starts with a letter followed by any no of digits. In Pascal, there is no need to use _ as identifier

RE

letter → {a|b|...|z|...A|B|...|Z|.
digit → 0|1|...|9|.
id → letter (letter|digit)*

TD



abc12.

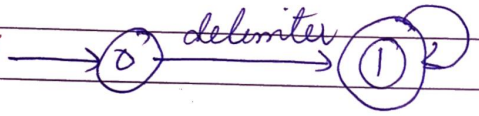
* In this way we can draw TD to recognize identifiers

2. Recognization of delimiter: A delimiter maybe blank space, gapspace.

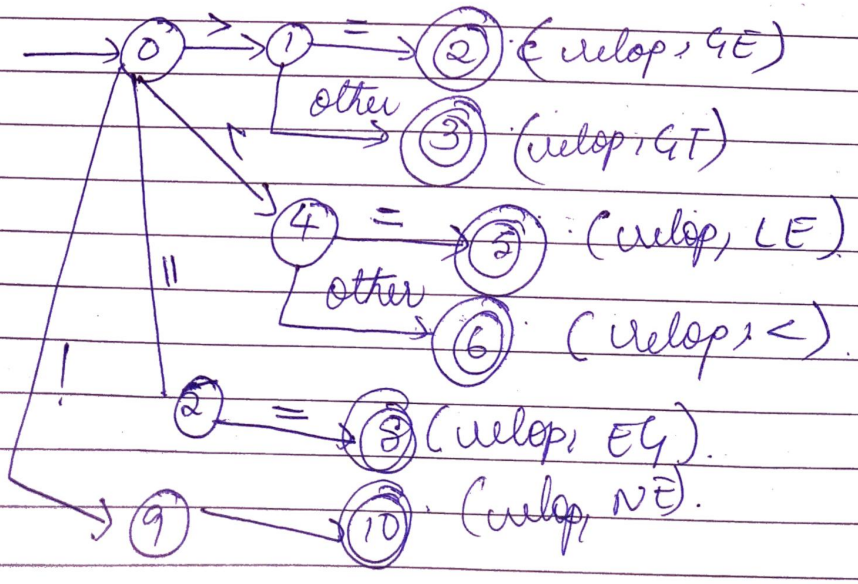
RE:

WS → delimiter (delimiter)*

TD:



3. Recognization of Relational Operators.
The relational operators are,
<, <=, >, >=, ==, !=.



lex.yy.c → [C compiler] → a.out

Input Stream → [a.out] → A sequence of tokens

→ Structure of a LEX program:

* Declarations %%
  translation rules
  %%
  Auxiliary functions

1. → Declaration section is useful inorder to declare C variables, constants.
→ It is useful for defining RE.
→ digit [0-9]
  letter [a-zA-Z]
→
```
%{
    variables, constants.
%}
```
→ Ex:
```
%{
    int a,b;
    float count=0;
%}
```

2. Translation rules.
→ Translation rules start with %% and ends with %%
→ Every rule is specified in the form of pattern followed by {action}
→ Here the patterns are RE & actions are C lang stmts

→ The pattern & action are seperated by multiple blank space characters.

pattern {Action}.

→ Eg :-

```
%.%
    pattern1  {Action1}.
    pattern2  {Action2}.
    pattern3  {Action3}.
%.%
```

→ If the string or token is found in / matches with pattern1 / then the corresponding action1 will be executed

3. Auxiliary functions :-

→ All the functions are defined in this section.

→ We have to use this AF inorder to define all the functions

→ Lex program to recognise identifiers, keywords, RO, numbers :-

lex.l

```
%{
    /* Lex program for recognize tokens*/
%}

letter [a-zA-Z]
digit [0-9]
id  {letter}({letter}|{digit})*
numbers {digit}+ (.{digit}+)?(E[+-]?{digit}+)?
```

```
%%
{id} {printf("%s is an identifier", yytext);}
if {printf("%s is a keyword", yytext);}
else {printf("%s is a keyword", yytext);}
"<" {printf("%s is less than Operator", yytext);}
"<="
">"
">="
"=="
"!=" {printf("%s is not equal to operator", yytext);}
{numbers} {printf("%s is a number", yytext);}
%%
```

→ **Compiler Construction Tools**

1.) **Scanner Generator :**
→ It is an other name for lexical Analysis, it produces stream of tokens.
→ Eg : Lex Tool

2) **Parser Generator :**
→ It is useful for the syntax analysis phase. It checks if the tokens syntax is correct or wrong
→ It accepts tokens as input & produces parse tree as output
→ Eg : YACC tool (yet another compiler compiler)

3) **Syntax Directed Translation Engine :**
→ It is useful during semantic analysis phase as well as intermediate code generation phase

→ It contains collections of routines (functions) which are useful to traverse the parse tree as well as it produces intermediate code

4). Data Flow Analysis Engine :-

→ It is useful during code optimization phase

→ So code optimization is done with the help of Data Flow Analysis Engine.

5) Code Generator :-

→ It accepts optimized Intermediate code as the input & produces machine code as the output.

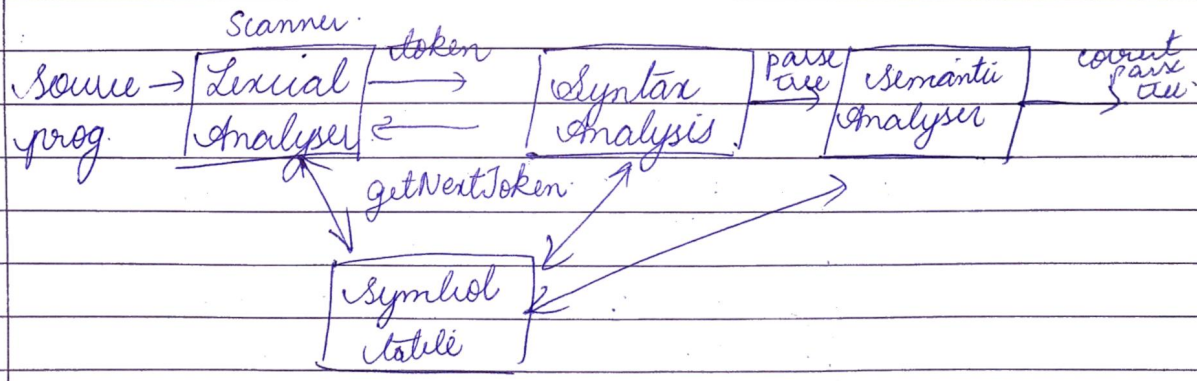
6) Compiler Construction Toolkits

→ It contains integrated collection of routines inorder to construct the compiler.

ROLE OF PARSER IN SYNTAX ANALYSIS.

OR

ROLE OF SYNTAX ANALYSIS



Lo1

Finite Automata :- I/p, O/p states transitions

Representations: * Graph like TD.
* Tabular like TD.
* Mathematical like TD

The mathematical model of finite automata.

$Q \rightarrow$ Finite set of states.
$\Sigma \rightarrow$ " " " i/p symbols
$\delta \rightarrow Q \times \Sigma \rightarrow Q$ transition function.
$q_0 \rightarrow$ start / Initial state.
$F \rightarrow$ Final / Accepting state.
$M = (Q, \Sigma, \delta, q_0, F)$

2 types of FA.

a) NFA :- No restriction on their edges & Σ as possible label.

b) DFA :- Exactly one have symbol of i/p leaving that state

Non Deterministic Finite Automata (NFA/NDFA)

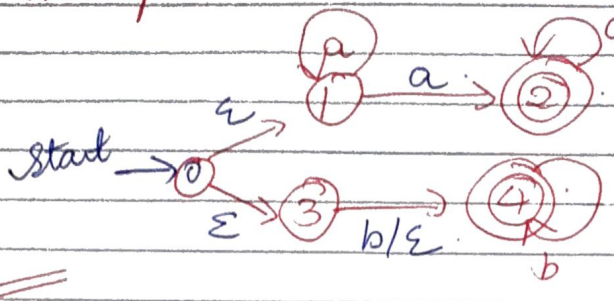
- Many paths for specific i/p from one state to other.
- Not every NFA is DFA, but NFA can be translated into DFA.

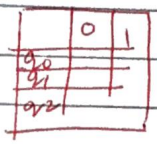
NFA consists of

* Finite set of states (S)
* Set of i/p symbols (Σ)
* Transition function ($\Sigma \cup \{\Sigma\}$)

* Start state (S0).
* Final state (SF).

Example:-

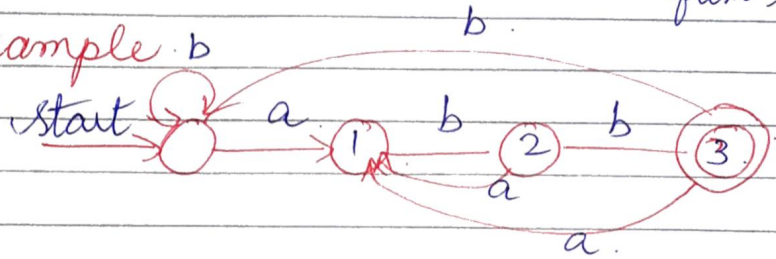


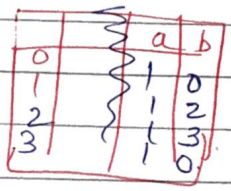


DFA:-

* There are no moves on i/p ε.
* There is exactly one edge out of state labelled with 'a' (ip symbol going out from each state is diff)

Example:





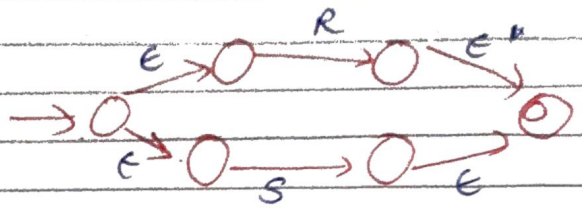
L02

Converting Regular Expression to NFA with ε

3 operations are,

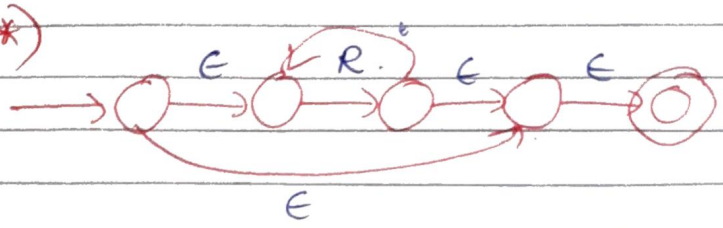
→ L + M. (union)
→ L . S. (concat)
→ L* (closure)

(R+S).



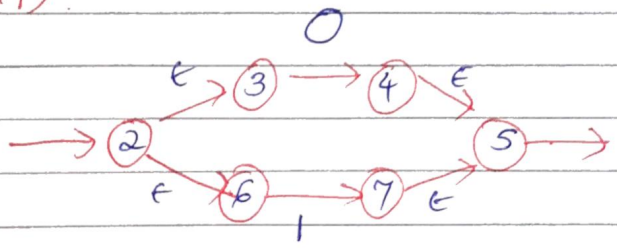
(R.S)

(R*)

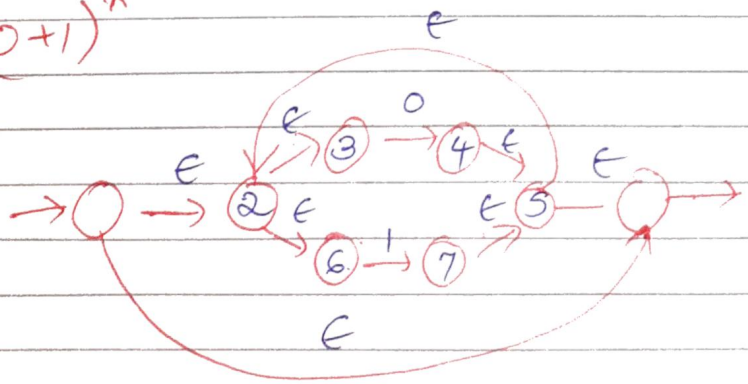


Ex:-

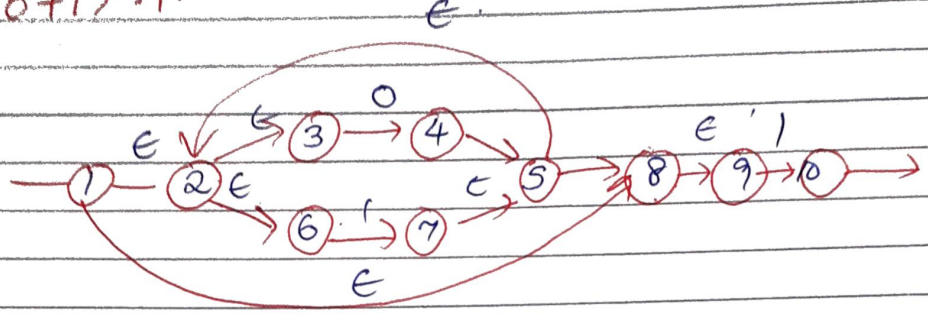
(0+1)



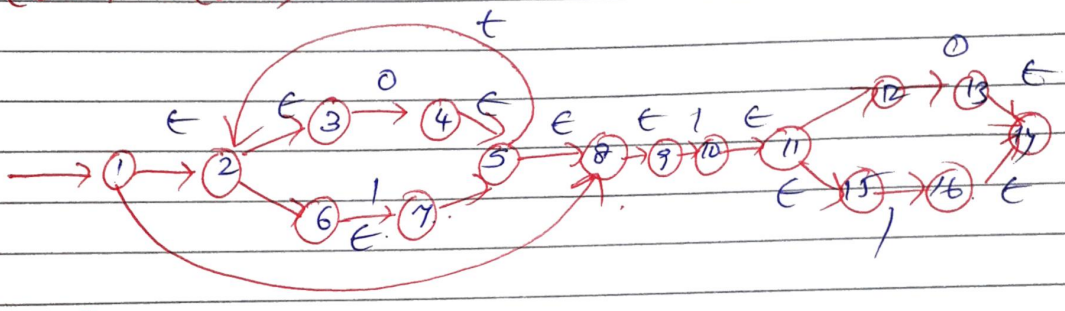
(0+1)*



$(0+1)^* \cdot 1$



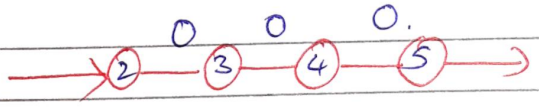
$(0+1)^* 1 (0+1)$



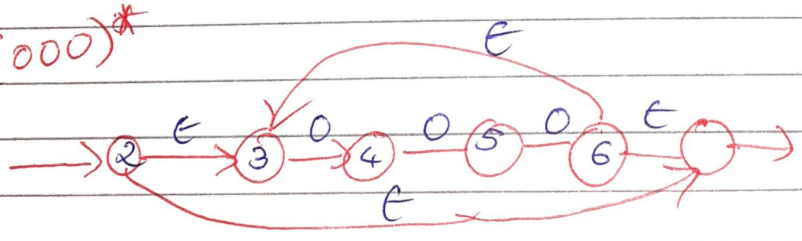
(ii) $(000)^* 1 + (00)^* 1$.

Soln.

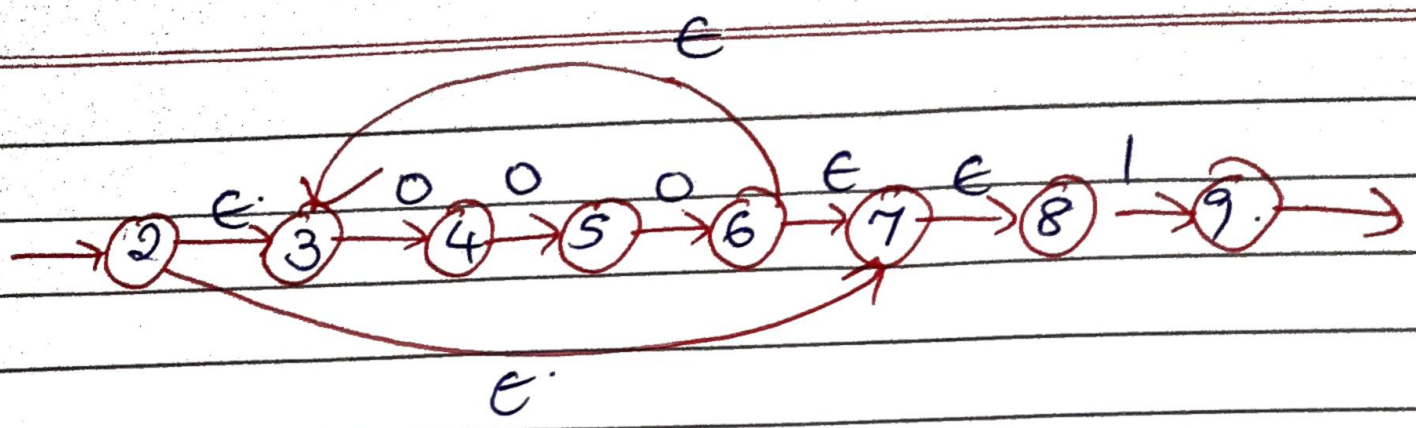
(000)



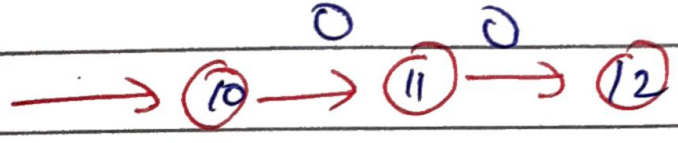
$(000)^*$



$(000)^* 1$



(00)



Converting ~~NFA~~ RE to NFA without ϵ

CD:- - - .

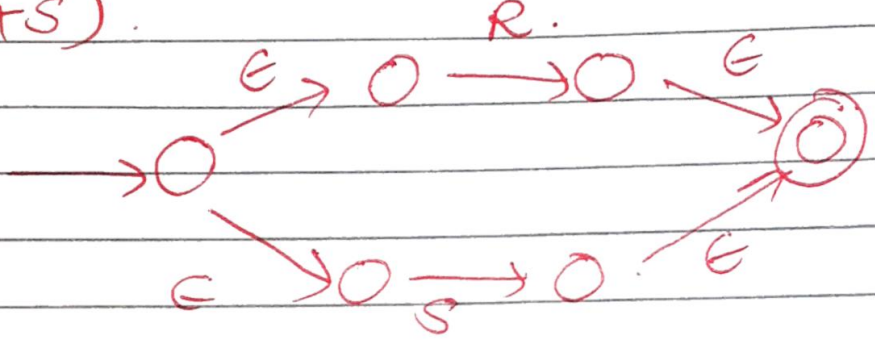
L03

CONVERSION OF RE TO NFA WITHOUT ∈.

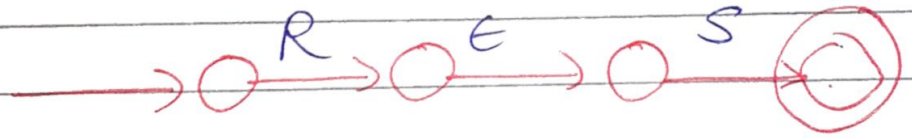
3 operations are
→ L+M
→ L.S.
→ L*.

∴ With NFA & ∈ nodes

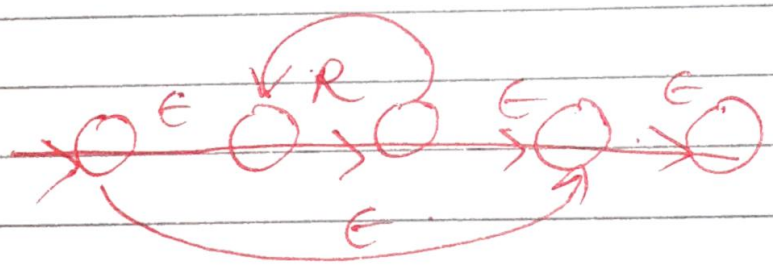
(R+S).



(R.S).



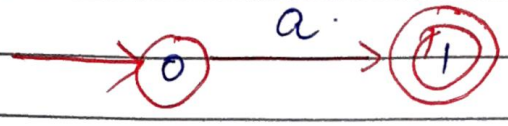
(R*).



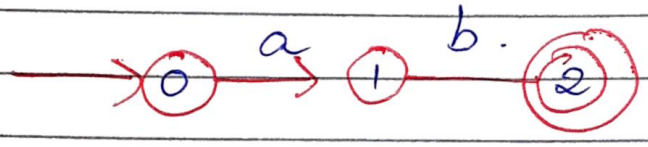
Now, we convert this RE to NFA without ∈-moves/more.

Steps / Operations.

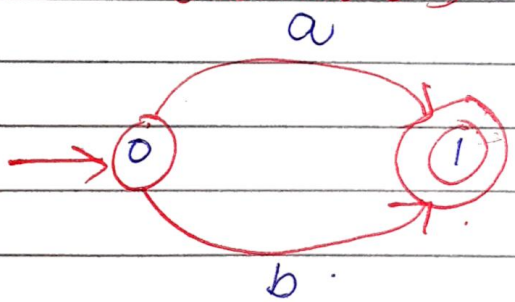
① a.



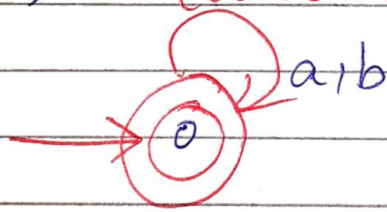
② a.b [concatenation].



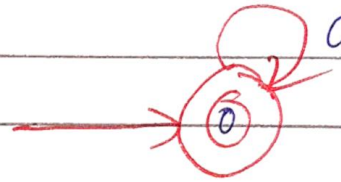
③ a+b. [union].



④ (a+b)* [closure].

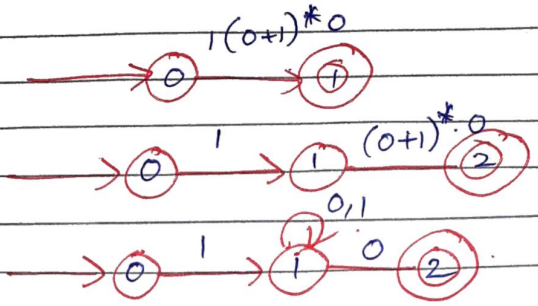


⑤ a*



Example

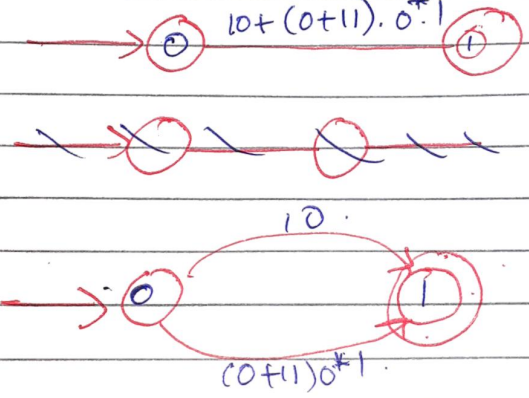
① $1(0+1)^*0$



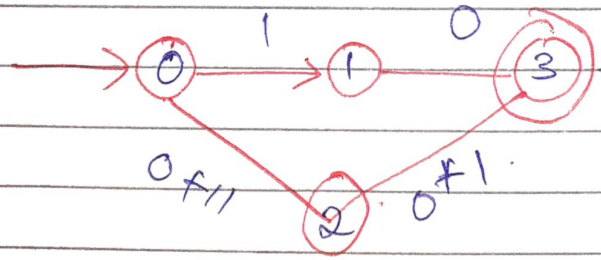
[concatenation]

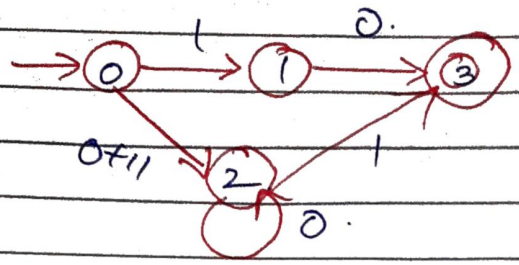
[closure].

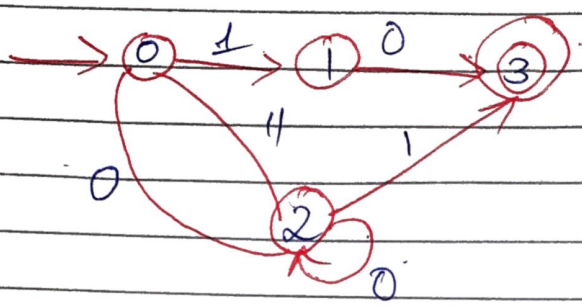
② $\underbrace{10}_{a} + \underbrace{(0+11)\,0^*1}_{b}$

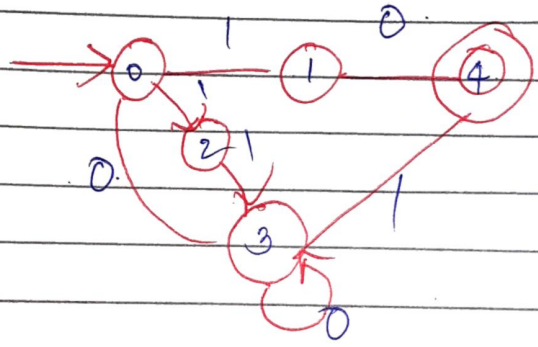


union





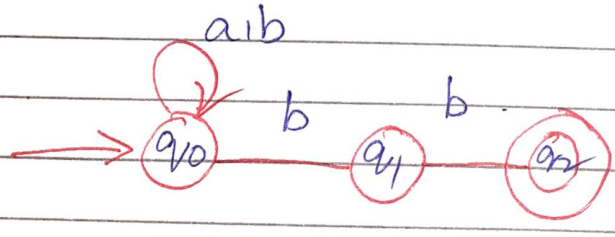




L04

NFA TO DFA.

Eg:-



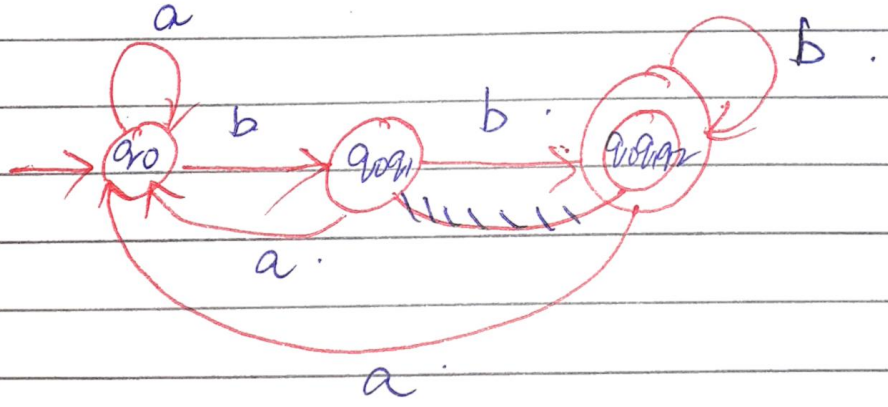
1). NFA Transition Table.

	δ	a	b
→	q_0	q_0	$\{q_0 q_0\}$
	q_1	—	q_2
*	q_2	—	—

2) DFA Transition Table

	δ	a	b
→	q_0	q_0	$[q_0 q_1]$
	$[q_0 q_1]$	q_0	$[q_0 q_1 q_2]$
*	$[q_0 q_1 q_2]$	q_0	$[q_0 q_1 q_2]$

3) DFA Transition Diagram.



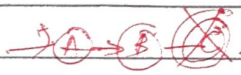
LOS

OPTIMIZATION OF DFA BASED PATTERN MATCHERS.

Steps involved in optimizing DFA are

1) Create Syntax Tree.
2) No of leaf nodes.
3) find the nullable of each node
4) " " firstpos " " "
5) " " lastpos " " "
6) " " Followpos " " "

Important States of an NFA:-



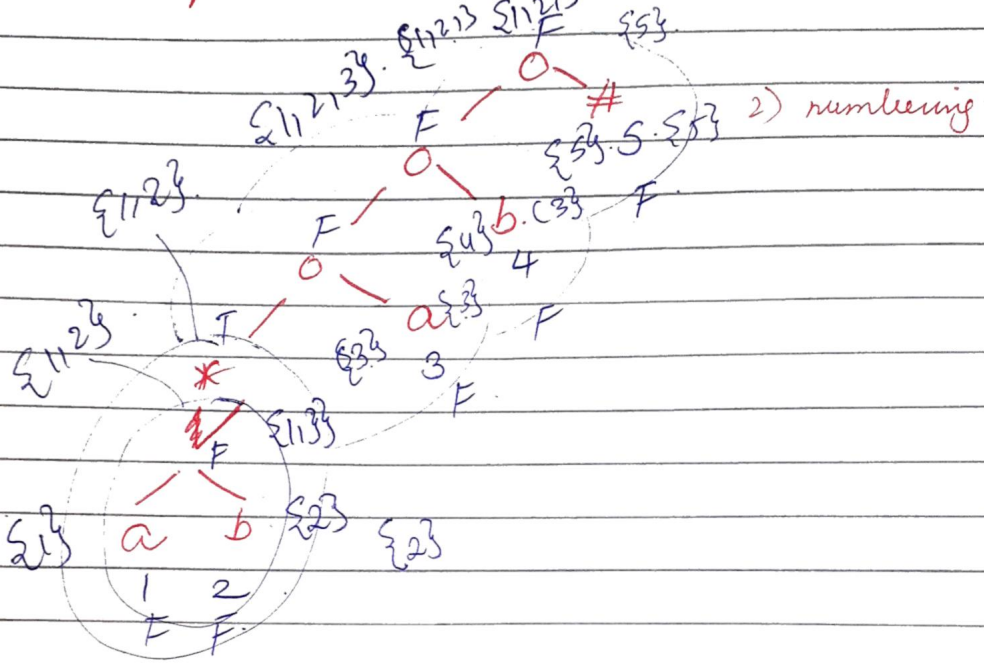
→ Roles played by various States in NFA.
→ non ε out-transitions States (imp)
→ When NFA has only one accepting State with no-out transitions then it is not an imp state.
→ So concentrate # (right endmarker) with RE to make the accepting state an important one

1) Syntax Tree.

→ Leaves correspond to operands & interior nodes corresponds to operators.

→ Interior node is called.
a) catnode if it is concatenation operator (dot)
b) Or node if it is union operator (|)
c) star node if it it is star operator (*)

Example:- (a|b)* as



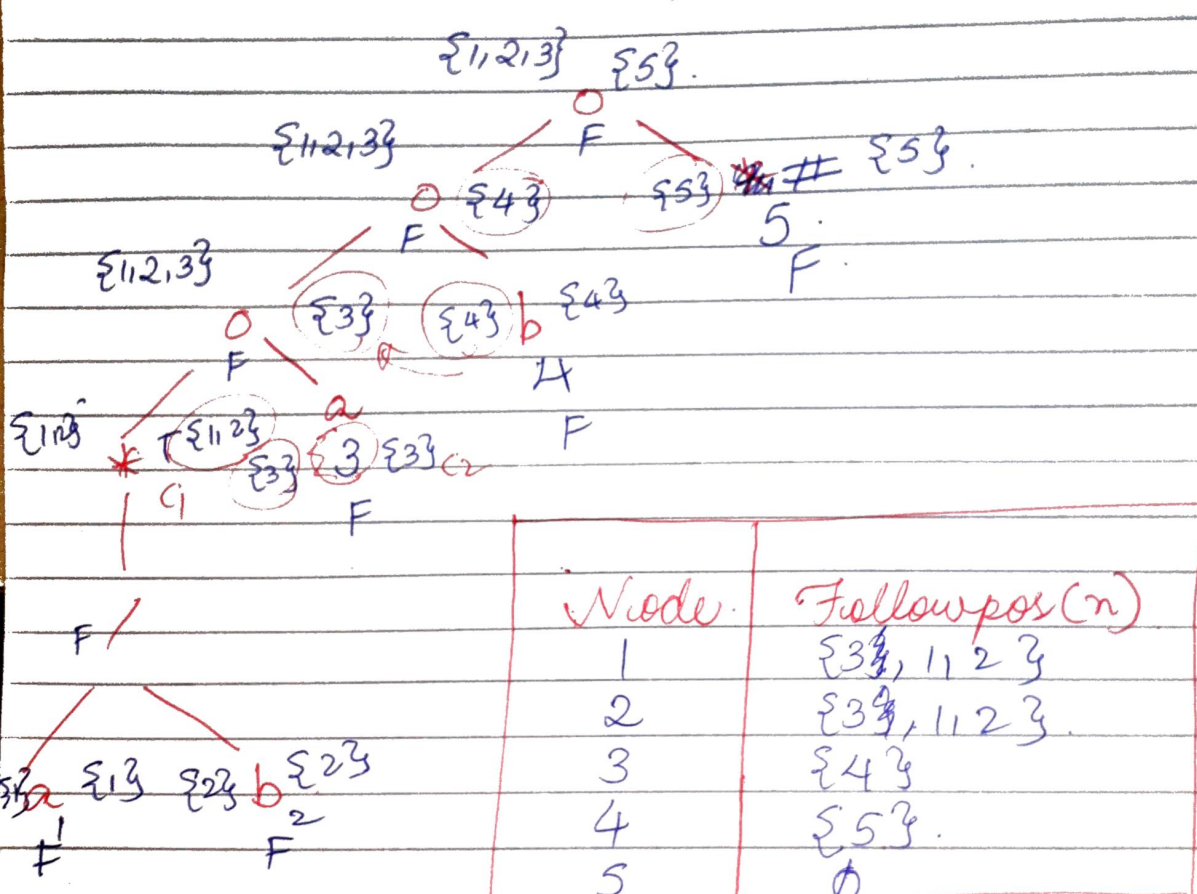
Note	Nullable	Firstpos(n)	Lastpos(n)
Leaf node ∈	True	ϕ	ϕ
Leaf with pos.	False	{i}	{i}
An or-node. $n = c_1 \mid c_2$	c_1 or c_2	firstpos(c_1) ∪ firstpos(c_2)	Lastpos(c_1) ∪ lastpos(c_2)
A cat-node. $n = c_1 c_2$	c_1 & c_2	if nullable(c_1)) firstpos(c_1) ∪ firstpos(c_2) else firstpos(c_1)	if (nullable(c_2) lastpos(c_1) ∪ las(c_2 else lastpos(c_2)
A star node $n = c_1^*$ A star node	True	firstpos(c_1)	laspos(c_1)

Followpos(n)

i) $n \rightarrow \cdot$ then
Firstpos(c2) → lastpos(c1)

(ii) $n \rightarrow *$ then
Firstpos(n) → lastpos(n)

Node Followpos(n).



Node	Followpos(n)
1	{3, 1, 2}
2	{3, 1, 2}
3	{4}
4	{5}
5	ϕ

Constructing Minimized DFA

Firstpos (Root node) = {1,2,3} → (A)

1) (A, a) = {1,3}
= FOP(1) ∪ FOP(3).
= {1,2,3,4} → (B)

(A, b) = {2}
= FOP(2).
= {1,2,3} → (A)

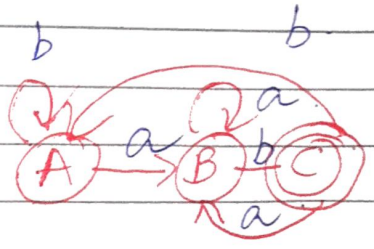
2) (B, a) = {1,3}
= {1,2,3,4} → (B)

(B, b) = {2,4}
= FOP(2) ∪ FOP(4). → (C)
= {1,2,3,5}

3) (C, a) = {1,3}
= {1,2,3,4} → (B)

(C, b) = {2}
= {1,2,3} → (A)

	a	b
→ A	B	A
B	B	C
* C	B	A



CONTEXT FREE GRAMMAR (CFG):-

→ $G = (V, T, P, S)$

where,

V is a set of variables or Non Terminals.
T is a set of terminal symbols.
P is a set of production rules. ($A \to \alpha$)
S is start symbol.

DERIVATION :-

→ In order to derive a string, we have to apply a seq of production Rules

→ 2 types

a) left most derivation
b) right most "

a) left most derivation :- At each step in derivation is applied to the left most variable or left most non terminal indicated by $\underset{lm}{\Rightarrow}$ or $\overset{*}{\underset{lm}{\Rightarrow}}$

Ex :- production rules

$E \to E+E$
$E \to E-E$
$E \to a/b$

input = a−b+a

LMD.

$E \underset{lm}{\Rightarrow} E+E$ ($\because E \to E+E$).
$\underset{lm}{\Rightarrow} E-E+E$ ($\because E \to a$)
$\underset{lm}{\Rightarrow} a-E+E$ ($\because E \to b$)
$\underset{lm}{\Rightarrow} a-b+E$ ($\because E \to a$)
$\underset{lm}{\Rightarrow} a-b+a$

b). Right most derivation :-
At each step in the derivation ; is applied to the right most variable or right most non terminal indicated by $\Rightarrow_{rm}$ or $\overset{*}{\Rightarrow}_{rm}$.

Ex :- Production rules.
$E \rightarrow E+E$.
$E \rightarrow E-E$ Input = a - b +.
$E \rightarrow a/b$.

RMD :-

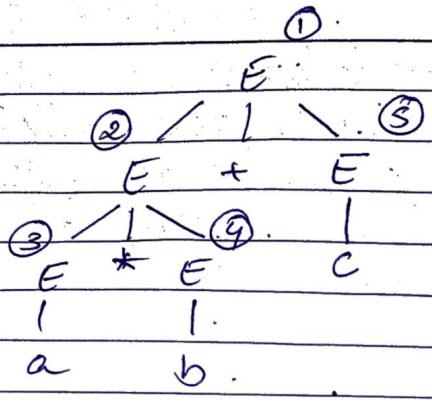
$E \Rightarrow E - E$.
$\Rightarrow$ 
$\Rightarrow E - E+E$ $(E \rightarrow E+E)$
$\Rightarrow E - E+a$ $(E \rightarrow a)$
$\Rightarrow E - b+a$ $(E \rightarrow b)$.
$\Rightarrow a - b - a$ //

PARSE TREE
It is a graphical representation for the derivation of the given production rule for a given CFG

Ex :- Production rule

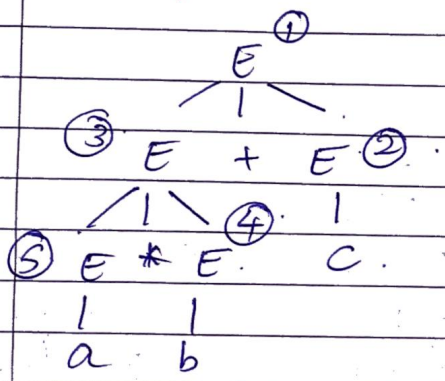
$E \rightarrow E+E$. Input = a * b + c.
$E \rightarrow E*E$
$E \rightarrow a/b/c$.

LMD/parse tree =



→ a*b+c.

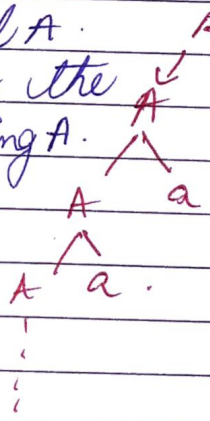
RMD / parse tree :-



⇒ a*b+c

RECURSION IN GRAMMAR :-

Left recursion :- The non-terminal A appears as the first symbol on the right hand side of the rule during A.
Ex :- A → Aa / a



Note :- Compiler never likes left recursion hence we always try to remove left recursion.

Note :-

A → Aα / β ⇓ A → βA' A' → αA' / ε

Formula.

eg:- $A \rightarrow Aa/a$.
$A \rightarrow aA'$
$A' \rightarrow aA'/E$

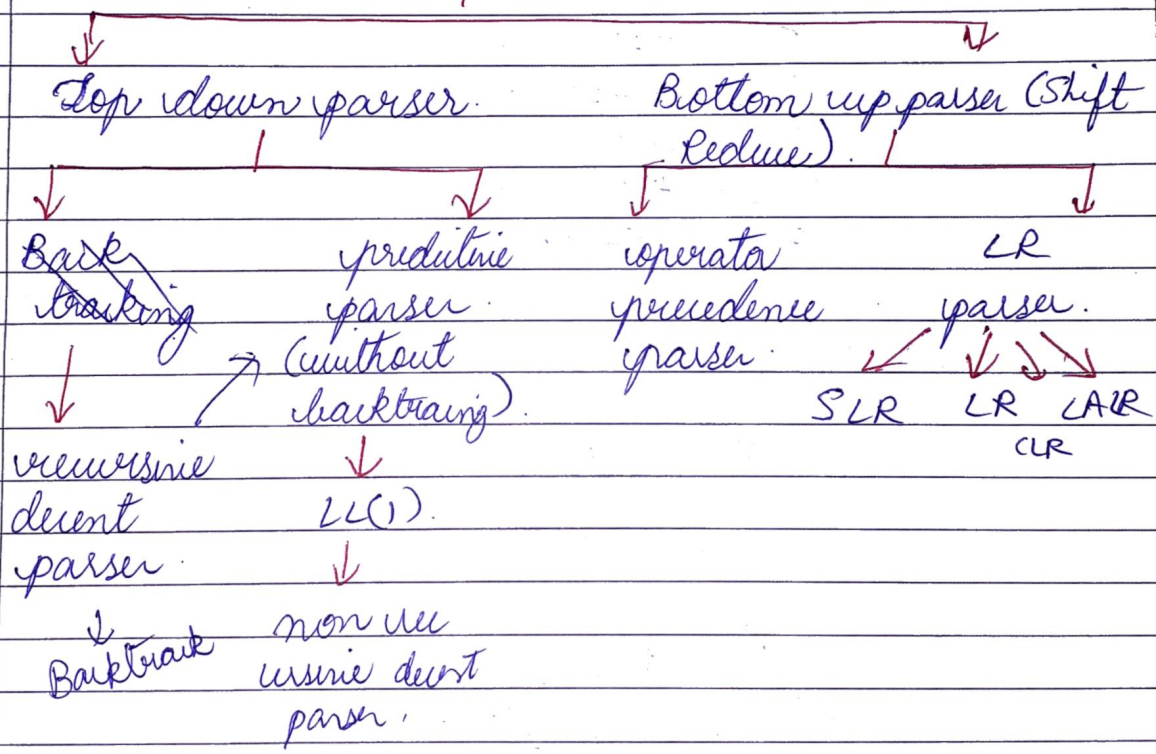
Right Recursion:-

→ The non terminal 'A' appears as the last symbol on the RHS of the rule defining A.

$A \rightarrow aA/a$

→ There is no problem for the compiler with right recursion.

TYPES OF PARSERS



Top Down Parser:-

→ Starting from root & process its children is known as top down parser

→ Top down parser are constructed for the grammar if its free from left recursion & ambiguity

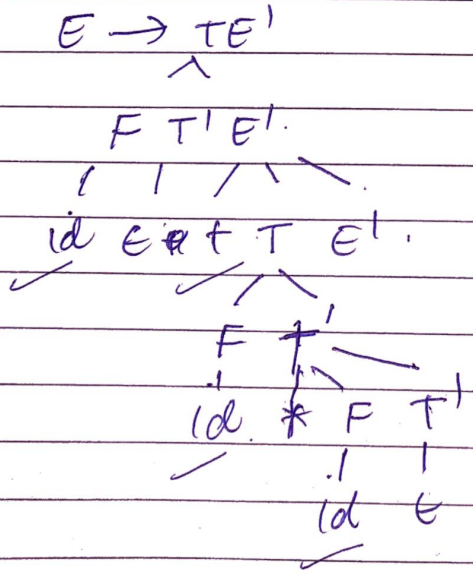
→ Top down parser always uses left derivation so it is imp to remove left recursion

Ex:- $E \rightarrow E+T/T$.
$T \rightarrow T*F/F$
$F \rightarrow (E)/id$.

Here we can see that the above grammar has left recursion

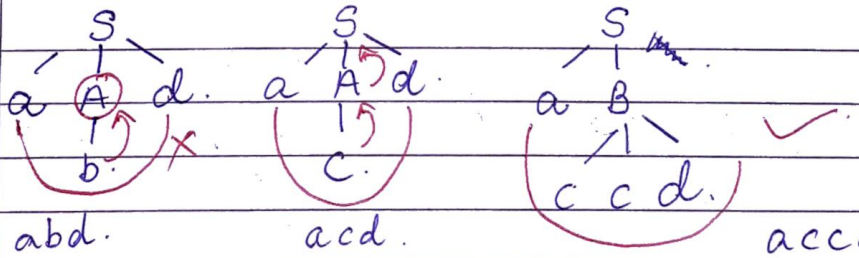
$E \rightarrow TE'$
$E' \rightarrow +TE'/\epsilon$.
$T \rightarrow FT'$.
$T' \rightarrow *FT'/\epsilon$.
$F \rightarrow (E)/id$.

drawing top down parser id+id*id.



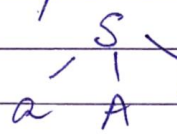
id + TE'.
id + FT' E'
id + id T' E'.
id + id * F T'
id + id * id T'
id + id * id E
id + id * id //.

Advantages Backtracking
Parser can be constructed easily by hand using top down methods.
→ Methods for performing top down pars
(i) Brute force method :- Here we try at a path, if we don't end up at result trackback & try again
Eg S → aAd/aB.
A → b/c.
B → ccd/ddc.
Strig :- accd.

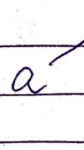


abd. acd. accd.

Steps :-
1). Select first production

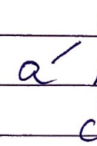


2) Choose first production of A



abdX

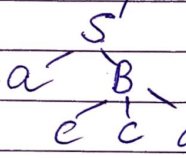
3) Choose second production of A.



acdX

4) Choose second production of S.

5) Choose 1st production of B



accd matched

(ii) Recursive Decent parsing

It is a top down parsing technique that constructs the parse tree from top & input is read from left to right.

Ex: $E \rightarrow E+T/T \longrightarrow ①$
$T \rightarrow T*F/F \longrightarrow ②$
$F \rightarrow E/id \longrightarrow ③$

string
id*(id+id)

Removing Left Recursion

① $E \rightarrow E+T/T$.

$E \rightarrow TE'$
$E' \rightarrow +TE'/\epsilon$

② $T \rightarrow T*F/F$.

$T \rightarrow FT'$
$T' \rightarrow *FT'/\epsilon$

c) $F \rightarrow (E) / id \rightarrow ⑥$

① $E \rightarrow TE'$

② $E' \rightarrow +TE' / E$

③ $T \rightarrow FT'$

④ $T' \rightarrow *FT' / E$

⑤ $F \rightarrow (E) / id$

recursive descent parser for grammar. (just in code format).

```
procedure E()①
 begin
        T();
        E Prime();
 end;
```

Note: E Prime ⇒ E'.
Advance is move pointer to 1 Step right (ip++)

```
Procedure E Prime()②
begin
 if input symbol = '+'
 then begin
  Advance();
  T();
  E Prime();
else
  begin
    return;
  end;
end;
```

```
Procedure T() ③
begin
    F();
    T Prime();
end;

Procedure T Prime() ①
begin
  if input-Symbol = '*' then
  begin
      Advance()
      F();
      T Prime();
  else
  begin
      return;
  end;
end;

Procedure F() ②
begin
    if input-Symbol = 'id' then
    begin
        Advance();
        E();
    end
    else if input-symbol = '('
    then
    begin
        Advance();
        E();
    end
```

```
if input-symbol = ')' then
    Advance();
else ERROR();
end;
end
```

First and Follow

First :- If α is any string of grammar symbols then First(α) the set of terminals that begins string derived from α

Eg :- ① $\alpha \rightarrow \epsilon$ (non terminal, terminal) First(α) = ϵ

② $\alpha \rightarrow a\beta$ First(α) = a
$\beta \rightarrow c$ First(β) = c

③ $Y \rightarrow y'$ First Y = First(Y').

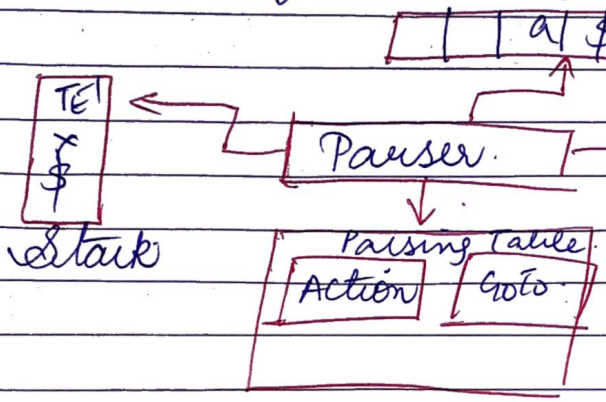
Eg :-
$S \rightarrow AaB$
$A \rightarrow b/\epsilon$
$B \rightarrow c$.

First(S) = First(First(A))
First(b, ϵ)
b, First($S \rightarrow aB$).
b, a. ✓

First(A) = b, ϵ
First(B) = c.

Top Down Parser - Predictive Parser / LL(1) parser / Non Backtracking

→ It is a type of Recursive decent with no backtracking
→ And it can be implemented non-recursively by using Stack DS.



→ With the help of LL(k) grammar, the predictive grammer is Back tracking free because this grammar contains some constrains
→ In Predictive parser, each step has atmost 1 production to choose

LL parser (Top-Down) Predictive Parser

→ It accepts LL grammar
→ It is denoted as LL(K) → no of look aheads.
- i/p from 1 to K.
- left most derivation

→ generally K = 1.
→ ∴ LL(K) = LL(1)

→ A grammer G is LL(1) If there are 2 distinct productions

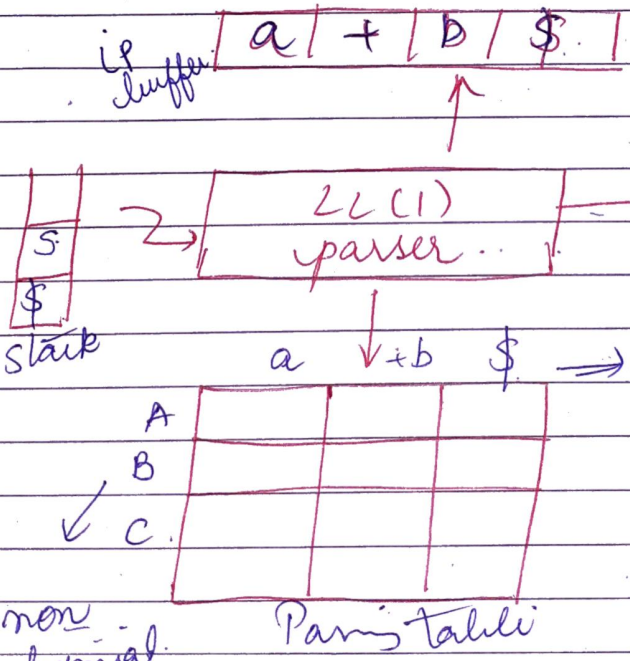
$A \to \alpha / \beta$.

1. For no terminal α & β derive strings beginning with a.
2. At most one of α & β can derive empty string
3. If $\beta \to \epsilon$, then α does not derive any string beginning with a terminal in FOLLOW(A)

→ LL(1) uses data structures like

a) i/p buffer.
b) stack
c) parsing table

→ Structure of LL parser



→ Construction of Predictive LL(1)

1. FIRST() LEADING()
   FOLLOW() / Trailing()

② Predictive parsing table using first & follow functions.
③ Parse the i/p string with the help of table.

Construction of LL(1) parser. / First & follow / Top down.

① First(), Follow()
② Parsing Table.
③ Stack implementation
④ parse the i/p string.

FINDING THE FIRST AND FOLLOW.

⇒ First & follow sets are needed. so that the parser can properly apply the needed production rule at the correct

FIRST FUNCTION :-

* First(X) is a set of terminal symbols that begin in strings derived from X.

Eg:- A → abc / def / g.

then, First(A) = {a, d, g}

RULES FOR CALCULATING FIRST FUNCTION:

Rule 01 :- For a production rule X → E
First(X) = {E}.

Rule 02 :- For any terminal symbol a.
First(a) = {a}.

Rule 03 :- For a production rule,
X → $Y_1 Y_2 Y_3$

Calculating First(X)

- If $\epsilon \notin$ First(Y1), then First(X) = First(Y1)
- If $\epsilon \in$ First(Y1), then First(X) = {First(Y1) − ϵ} ∪ First(Y2 Y3)

Calculating First(Y2 Y3)

- If $\epsilon \notin$ First(Y2), then First(Y2 Y3) = First(Y2)
- If $\epsilon \in$ First(Y2), then First(Y2 Y3) = {First(Y2) − ϵ} ∪ First(Y3).

Similarly we can expand the rule for any production rule.

$X \rightarrow Y_1 Y_2 Y_3 \ldots$

FOLLOW FUNCTION:-

* Follow(A) is a set of terminal symbols that appear immediately to the right of A.

Rules for calculation of follow function

Rule 1:- For the start symbol S, place $ in Follow(S).

Rule 2:- For any production rule $A \rightarrow \alpha B$, Follow(B) = Follow(A).

Rule 3:- For any production rule $A \rightarrow \alpha B\beta$.

- If $\epsilon \notin$ First(B), then Follow(B) = First(B).
- If $\epsilon \in$ First(B), then Follow(B) = {First(B) − ϵ} ∪ Follow(A).

Note:-

* ϵ may appear in the first function of a non-terminal
* ϵ will never appear in the follow

function of a non-terminal

* It is recommended to eliminate left recursion from grammar if present before calculating first & follow functions
* We will calculate the follow function of a non-terminal looking where it is present on RHS of a production Rule.

BOTTOM UP PARSING:-

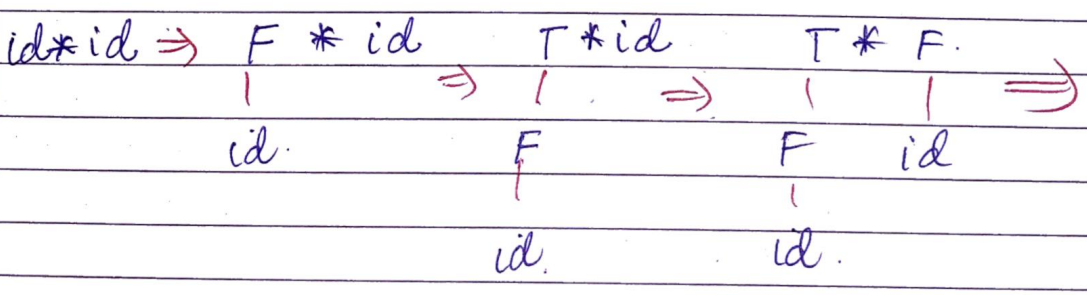
Construction of parse tree for an i/p string beginning at the leaves & working up towards the root.

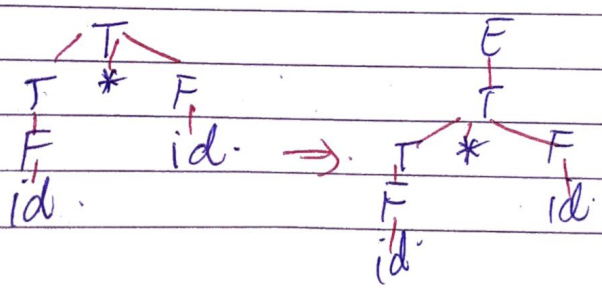
Example

$E \rightarrow E+T/T$
$T \rightarrow T*F/F$.
$F \rightarrow (E)/id$.

$w = id*id$.

$id*id \Rightarrow F*id \Rightarrow T*id \Rightarrow T*F \Rightarrow$





Reductions :

- The process of reducing a string 'w' to the Start Symbol of the grammar.
- At each step of reduction, a substring matching the body of production is replaced by non-terminal.
- The key decisions are
  * when to reduce
  * what production to apply.

Reductions in terms of sequence of strings.
id * id, F * id, T * id, T * F, T, E.

* Reduction is the reverse of a step in derivation
$E \Rightarrow T \Rightarrow T * F \Rightarrow T * id \Rightarrow F * id \Rightarrow id * id$

Handle pruning :

Handle → Substring that matches the body of productions.
Handle pruning → obtains the right most derivation in reverse order

Right Sentential form	Handle	Reducing Production
$id_1 * id_2$	id_1	$F \rightarrow id$
$F * id_2$	F	$T \rightarrow F$
$T * id_2$	id_2	$F \rightarrow id$
$T * F$	$T * F$	$T \rightarrow T * F$
T	T	$E \rightarrow T$

SHIFT REDUCE PARSING.

→ Shift reduce parsing is a process of reducing a string to the start symbol of a grammar.

→ It uses a stack to hold the grammar & an input to hold the string.

A string ——reduce to——→ the starting symbol

→ Shift reducing parsing performs 2 actions: Shift & reduce.

→ At shift action, the current symbol in the i/p string is pushed to the stack

→ At each reduction the symbol will be replaced by non-terminals

→ The Symbol is the right side of production & non-terminal is the left side of production.

Ex:

Grammer.	Input String
$S \to S+S$	$a_1 - (a_2 + a_3)$
$S \to S-S$	
$S \to (S)$	
$S \to a$	

Table.

Stack Contents	Input String	Actions
$	$a_1 - (a_2 + a_3)$$	Shift a_1
$$a_1$	$- (a_2 + a_3)$$	reduce by $S \to a$
$S	$- (a_2 + a_3)$$	shift -
$S-	$(a_2 + a_3)$$	Shift (
$S-(	$a_2 + a_3)$$	Shift a_2
$S-(a2	$+ a_3)$$	reduce by $S \to a$
$S-(S	$+ a_3)$$	shift +
$S-(S+	$a_3)$$	Shift a_3
$S-(S+a3	$)$$	reduce by $S \to a$
$S-(S+S	$)$$	Shift)
$S-(S+S)	$$	reduce by $S \to S+S$
$S-(S)	$$	reduce by $S \to (S)$
$S-S	$$	reduce by $S \to S-S$
$S	$	Accept

→ It is a form of bottom up parsing containing Stack & input buffer.
→ handle appears at the top of Stack.

SHIFT REDUCE PARSING CONT-----

STACK	INPUT.
$	w$

Four posible actions of shift-reduce parser:

1. Shift → Shift the next input symbol onto the top of the stack.
2. Reduce → replace the string with nonterminal.
3. Accept → Successful completion of parsing.
4. Error → Discover a syntax error

Primary options are.

1. Shift.
2. reduce

Conflicts during Shift-reduce parsing

1. Shift/reduce conflict
2. reduce/reduce conflict

Shift reduce conflict:

Whether to Shift the next input symbol or to reduce the current handle

A → Sa.
B → Sab.

Stack.	Input.
....	...
...	...
$Sa.	b--$.

CD- Cont.

Reduce-Reduce conflict:-

Whenever a handle appears on the stack, for that handle if there are several productions are available then the conflict occurs on which of the several reductions to apply.

$A \rightarrow Sa$.
$B \rightarrow Sa$

Stack	Input	Output
....	....	
....	....	
$\$Sa$	$b \cdots \$$	

INTRODUCTION TO L-R PARSING, SIMPLE L-R PARSING.

* The most common type of bottom up parsing → LR(K) parsing.
* L → left to right scanning.
  R → Rightmost derivation in reverse.
  K → No of i/p symbols of lookahead

Why LR Parsers?

* Table-Driven approach.
* LR-grammar → recognize the handles of right-sentential forms when it is on the stack
* recognize all programming language constructs

* Implemented efficiently as other
* Detect Syntactic error as soon as possible.
* Able to recognize the occurrence of right side of production - in a right-sided sentential form "Items" and "parser" states

Items and the LR(0) Automation:-
* Shift-reduce decisions are made by maintaining states.
* State → set of "items"
* LR(0) item of G is production of G of a dot at some position.

Eg:- A → xyz
A → .xyz.
A → x.yz.
A → xy.z.
A → xyz.

A → E
A → .

Canonical LR(0) Collection →

Collection of sets of
* augmented grammar.
* closure
* GOTO.

* Augmented grammar.

S → Aa.
S' → S.
S → Ar

Closure of Item Sets :-

For grammar G, I is a set of items and CLOSURE(I) can be constructed by two rules.

1. Add every items in I to closure(I).
2. If $A \to \alpha . B\beta$ is in CLOSURE(I) & $B \to \gamma$ is production, then add item $B \to . \gamma$ to CLOSURE(I).

Eg :-

$E \to E+T / T$.
$T \to T*F / F$
$F \to (E) / id$.

Augmented Grammar ⇒

$E' \to E$.
$E \to E+T / T$.
$T \to T*F / F$.
$F \to (E) / id$.

I is the set of one item $\{[E' \to . E]\}$, then CLOSURE(I) contains the set of items I_0.

$E' \to . E$
$E \to . E+T$.
$E \to . T$
$T \to . T*F$
$T \to . F$.
$F \to . (E)$.
$F \to . id$.

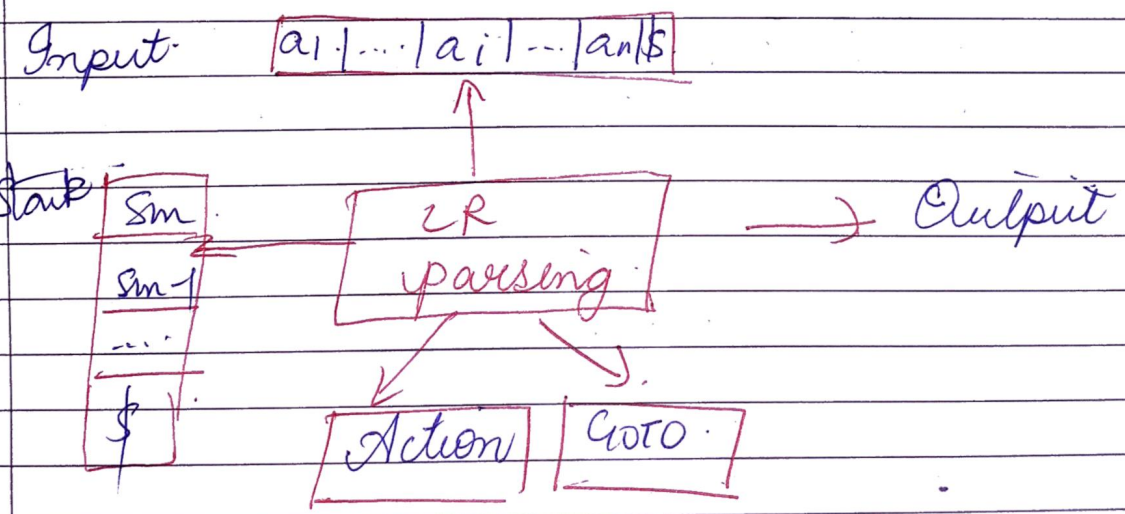
* In the closure of I, we have 2 classes.
1. Kernel items → Initial item, $S' \to .S$ & all items whose dots r not at left end

2. Non Kernel Items :- → Items with dots at left end except $S' \to .S$.

* GOTO (I, X)
  I → set of items.
  X → grammar symbol.
  * Define the transitions in LR(0)
  * Specifies transition from state under input X.

Eg :- $I \Rightarrow \{[E' \to E.], [E \to E.+T]\}$,
GOTO (I, +) contains
$E \to E+.T$
$T \to .T*F$
$T \to .F$
$F \to .(E)$
$F \to .id$

Model of LR Parser.



Structure of LR Parsing Table

Two parts
* Action
* GOTO

1. Action → four forms
   a) Shift c) Accept
   b) Reduce d) Error
2. GOTO function-
   GOTO (Ii, A) = Ij
   → maps a state i.e. non terminal A to state j

By using the example we are going to construct the SLR parsing table i.e.
Simple LR " "

Steps: ① Augmented grammar
② Canonical LR(0) collution
③ SLR parsing table

Example
E → E+T/T
T → T*F/F
F → (E) / id

① Augmented grammar
(0) E' → E
(1) E → E+T
(2) E → T
(3) T → T*F
(4) T → F
(5) F → (E)
(6) F → id

② Canonical LR(0) collection:

I_0 : $E' \to .E$
$E \to .E+T$
$E \to .T$
$T \to .T*F$
$T \to .F$
$F \to .(E)$
$F \to .id$

GOTO (I_0, E)
I_1 : $E' \to E.$
$E \to E.+T$

GOTO (I_0, T)
I_2 : $E \to T.$
$T \to T.*F$

GOTO (I_0, F)
I_3 : $T \to F.$

GOTO (I_0, C)
I_4 : $F \to (.E)$
$E \to .E+T$
$E \to .T$
$T \to .F$
$F \to .(E)$
$F \to .id$

GOTO (I_0, id)
I_5 : $F \to id.$

GOTO $(I_1, +)$
I_6 : $E \to E+.T$
$T \to .T*F$
$T \to .F$
$F \to .(E)$
$F \to .id$

GOTO $(I_2, *)$
I_7 : $T \to T*.F$
$F \to .(E)$
$F \to .id$

GOTO (I_4, E)
I_8 : $F \to (E.)$
$E \to E.+T$

GOTO (I_6, T)
I_9 : $E \to E+T.$
$T \to T.*F$

GOTO (I_7, F)
I_{10} : $T \to T*F.$

GOTO $(I_8,))$
I_{11} : $F \to (E).$

Construction of Simple LR parsing Table

STATE	ACTION						GOTO		
	id	+	*	(	)	$	E	T	F
0	S5			S4			1	2	3
1		S6				acc			
2		r2	S7		r2	r2			
3		r4	r4		r4	r4			
4	S5			S4			8	2	3
5		r6	r6		r6	r6			
6	S5			S4				9	3
7	S5			S4					10
8		S6			S11				
9		r1	S7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

FOLLOW(E) = {$, +,)}.
FOLLOW(T) = {$, +,), *}.
FOLLOW(F) = {$, +,), *}.

cont...

SLR PARSING.

w = id * id + id.

w = id * id + id.

STACK	SYMBOLS	INPUT	ACTION
0		id * id + id $	Shift
05.	id.	* id + id $	Reduce by F→id
03	F.	* id + id $	Reduce by T→F
02.	T	* id + id $	Shift
027.	T*	* id + id $.	Shift
0275	T* id.	+ id $	Reduce by F→id
02710	T* F	+ id $	Reduce by T→T*F
02	T	+ id $	Reduce by E→T
01	E	+ id $.	Shift
016.	E +	id $.	Shift.
0165.	E + id.	$.	Reduce by F→id
0163.	E + F.	$.	Reduce by T→F
0169	E + T	$	Reduce by E→E+T
01	E.	$.	accept

MORE POWERFUL LR PARSERS.

→ use one symbol of lookahead on the input.

* Canonical LR.
* Lookahead-LR or LALR.

1) Canonical LR uses lookahead.
   Symbols: LR(1) items = LR(0) item + lookahead.

2) LALR uses LR(0) sets of items.

Canonical LR(1) Items : LR(0) items + lookahead.
The general form of an item
[A → α . Bβ , a] ——→ (1).
A → αBβ is a production &.
a → terminal or right endmarker $

Difference of SLR & CLR.
1) CLR : reduce, it uses lookahead instead of finding follow
2) Closure : for lookahead, consider FIRST(βa) in (1). (LR(0) items + lookaheads)

Example :
S → CC.
C → cC / d.

∴ Augmented grammar :
S' → S.
(1) .S → CC.
(2) .C → cC.
(3) .C → d.

LR (1) Sets of Items.

A → α.Bβ, a.

I0 : S' → .S, $
S → .CC, $ — FIRST (E$).
C → .cC, c/d — FIRST (C$)
C → .d, c/d — FIRST (C$)

UNIT-3

SDD

SYNTAX DIRECTED DEFINATION.

* SDD is a ~~combo~~ of CFG with Semantic Rules.
* Attributes are associated with grammar symbols
* Semantic rules are associated with productions.
* If 'x' is a symbol and 'a' is one of its attributes then x.a denotes the value at node 'x'
* Attributes may be no's, strings, references, datatypes etc

Production	Semantic Rules
$E \rightarrow E+T$	$E.val = E.val + T.val$
$E \rightarrow T$	$E.val = T.val$

gram attri

TYPES OF ATTRIBUTES IN SDD

1) SYNTHESIZED ATTRIBUTE :-

* If a node takes value from the children node then it is a Synthesized Attribute
* Eg $A \rightarrow B,C,D$
  A is the parent node.
  B, C, D are the children nodes

A.S = B.S. } parent node A
A.S = C.S. } taking value from
A.S = D.S } children B, C, D

where S is an attribute associated with A, B, C, D

INHERITED ATTRIBUTE:-

* If a node takes value from its parent or siblings

Eg:- A → BCD.

C.i = A.i (parent node)
C.i = B.i (sibling)
C.i = D.i (sibling)

where i is an attribute associated with A, B, C, D

TYPES OF SDD:- — (2)

(1) S- ATTRIBUTED SDD
(or)
S- Attributed definations
(or)
S- Attributed grammar

(2)
L - Attributed SDD
(or)
L - Att df
(or)
L - Att gram.

① S-Attributed SDD.

* A SDD that uses only SYNTHESIZED Attributes is called S-Attributed SDD.
* Eg:- $A \rightarrow BCD$.
  $A.S = B.S$.
  $A.S = C.S$.
  $A.S = D.S$
* parent node takes values from the children nodes.
* Semantic actions are always placed at right end of production. Also called. Postfix SDD.
* Attributes are evaluated with bottom up parsing.

② L-Attributed SDD.

* A SDD that uses both synthesized & inherited Attributes
* Attributes are restricted to inherit from parent or left sibling only
* Eg:- $A \rightarrow XYZ$.
  $\{Y.S = A.S\}$ ✓.
  $\{Y.S = X.S\}$ ✓
  $\{Y.S \neq \cancel{A}.S\}$
* Semantic Actions are placed anywhere of RHS.
* Attributes are evaluated by traversing parse tree Depth first, Left to right

SA

SYNTAX DIRECTED TRASLATION SCHEME

* All operations of SDD can be implemented using SDT Schemes.
* SDT'S are implemented during parsing, without building a parse tree; SDD → SDT + actions

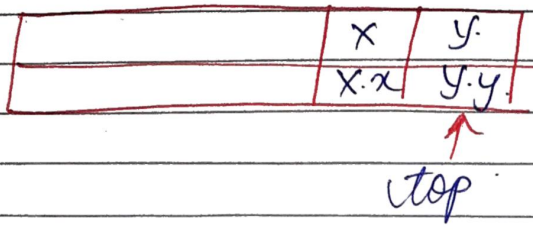
Two imp classes of SDD'S.
1) S-attributed, SR-Parsable
2) L-attributed, LL-parsable.

1) Postfix Translation Schemes.
* SDT in which each action is placed at the end of production (right)
* SDT'S with all actions at the right end of the production bodies. → postfix SDT'S.

$L \rightarrow En.$	{ Print (E.val); }
$E \rightarrow E_1 + T.$	{ E.val = E_1.val + T.val; }.
$E \rightarrow T$	{ E.val = T.val; .
$T \rightarrow T_1 * F$	{ T.val = T_1.val x F.val; }.
$T \rightarrow F$	{ T.val = F.val; }
$F \rightarrow (E)$	{ F.val = E.val; }.
$F \rightarrow digit$	{ F.val = digit. lexusval; }

2) Parser Stack Implementation of postfix SDT's.

* Implemented during LR parsing.



Grammar Symbols.
Synthsized attributes(s)

PRODUCTION.	ACTIONS.
$L \rightarrow En$.	{ Print (Stack [top-1].val); top = top - 1; }.
$E \rightarrow E_1 + T$.	{ Stack [top-2].val = Stack [top-2].val + Stack [top].val; top = top - 2; }.
$E \rightarrow T$.	
$T \rightarrow T_1 * F$	{ Stack [top-2].val = Stack [top-2].val x Stack [top].val; top = top - 2; }.
$T \rightarrow F$.	
$F \rightarrow E$.	{ Stack [top-2].val = Stack [top-1].val. top = top - 2; }
$F \rightarrow$ digit	

3) SDT's with actions inside productions.

* $B \rightarrow X \{a\} Y$, the action a is done after we recognized X, or all the terminals derived from X.
* If bottom up → perform as soon as 'X' appears on the top of parsing stack.
* If top down → perform actions just before ending the occurence of 'Y'

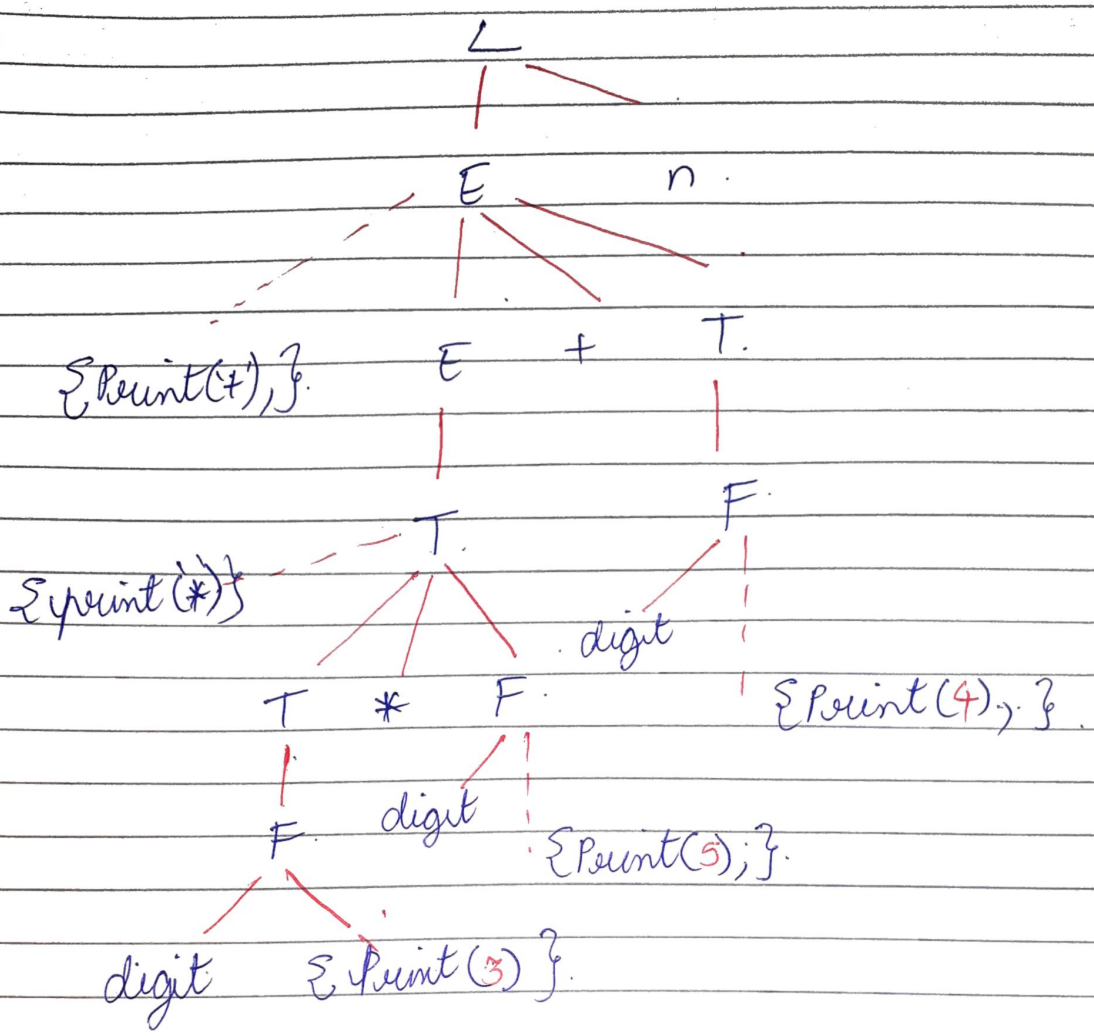
Production SDT for infix to prefix translation

1) $L \rightarrow En$.
2) $E \rightarrow \{Print('+');\} E_1 + T$.
3) $E \rightarrow T$.
4) $T \rightarrow \{Print('*');\} T_1 * F$.
5) $T \rightarrow F$.
6) $F \rightarrow (E)$
7) $F \rightarrow$ digit {Print(digit.lexval)}.
8)

* Ignore actions, parse I/p & produce parse tree
* Examine interior node, add additional children for actions
* perform preorder traversal

Rufi +* 354.

3 * 5 + 4



4) Eliminating Left recursion from SDTS.

Priniples;

* When transforming grammar, treat actions like they were terminal symbols.

$A \rightarrow A\alpha / B$.

$A \rightarrow \beta R$.

$R \rightarrow \alpha R / E$

$A \rightarrow A\alpha / \beta$.

Example

$E \rightarrow E_1 + T$.
$E \rightarrow T$.

$\alpha = +T$. {Printf('+');}.
$E \rightarrow TR$.
$R \rightarrow +T$. {Printf('+');} R.
$R \rightarrow \epsilon$.

$A \rightarrow A_1 y$ {$A.a = g(A_1.a, y.y)$}.
$A \rightarrow x$ {$A.a = f(x.x)$}.

$A \rightarrow xR$.
$R \rightarrow yR / \epsilon$

Parse Tree before elimination

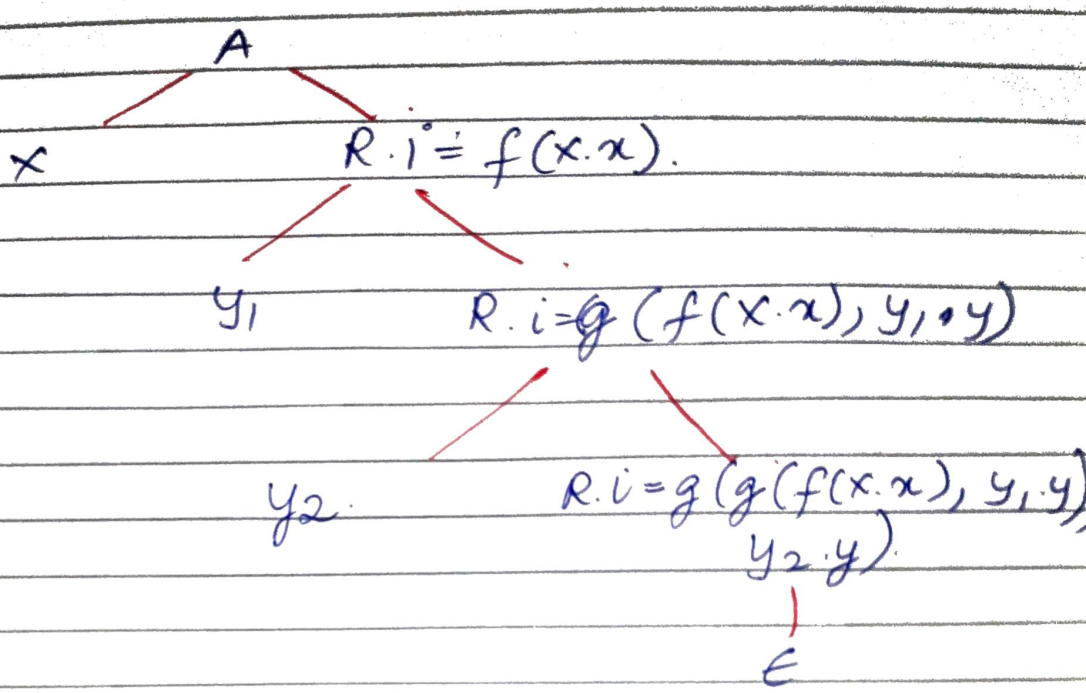
$A.a = g(g(f(x.x), y_1.y), y_2.y)$.

$A.a = g(f(x.x), y_1.y)$. y_2.

$A.a = f(x.x)$. y_1.

x.

Parse Tree After eliminating Left recursion



TRANSLATION OF BOOLEAN EXPRESSIONS.

(OR)

GENERATING 3-ADDRESS CODE FOR BOOLEAS.

(OR)

SDD OF THREE-ADDRESS CODE FOR BOOLEAN EXPRESSIONS.

(OR)

CONTROL FLOW TRANSLATION WITH BOOLEAN EXPRESSION.

* Boolean Expressions give either True & False
* True means 1
  False means 0
* Here we are taking 2 logic gates
  || and && &e !

Eg:- ~~PRODUCTION~~

a) For logical OR (||)
PRODUCTION.
$B \rightarrow B_1 \,||\, B_2$

SEMANTIC RULES :-
$B_1.true = B.true$
$B_1.false = newlabel()$.
$B_2.true = B.true$
$B_2.false = B.false$

$B.code = B_1.code \,||\, label(B_1.false) \,||\, B_2.code$

b) For logical AND
PRODUCTION :- $B \rightarrow B_1 \,\&\&\, B_2$

SEMANTIC RULES :-
$B_1.true = newlabel()$
$B_1.false = B.false$
$B_2.true = B.true$
$B_2.false = B.false$

$B.code = B_1.code \,||\, label(B_1.true) \,||\, B_2.code$

c) For logical NOT (!)

Production :- $B \rightarrow !B_1$

Syntax Rules-

$B_1.true = B.false$
$B_1.false = B.true$
$B.code = B_1.code$

Production :- $B \rightarrow true$

SR :- $B.code = generate('goto'\ B.true)$

Production :- $B \rightarrow false$

SR :- $B.code = generate('goto'\ B.false)$

//

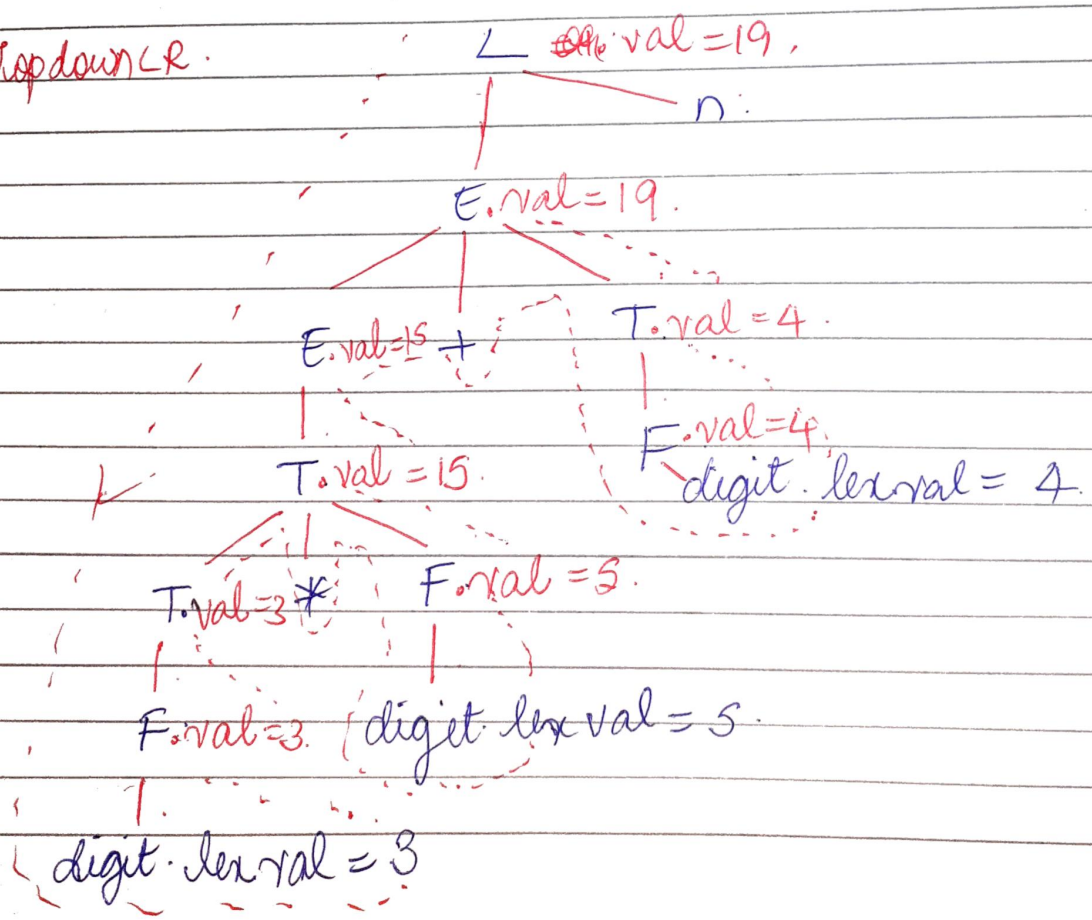
SDD OF SIMPLE DESK CALCULATOR.
(OR)
SDD OF EVALUATION OF EXPRESSION.
(OR)
ANNOTATED PARSE TREE FOR 3*5+4n.

Productions	Semantic Rules
$L \rightarrow En$	$L.val = E.val$
$E \rightarrow E_1 + T$	$E.val = E_1.val + T.val$
$E \rightarrow T$	$E.val = T.val$
$T \rightarrow T_1 * F$	$T.val = T_1.val * F.val$
$T \rightarrow F$	$T.val = T.val = F.val$
$F \rightarrow digit$	$F.val = F.val = digit.lexval$

Annotated parse Tree :-

A parse tree which contains values at each node is called as Annotated parse tree.

Top down LR.

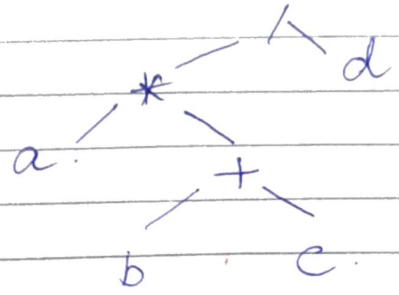


FORMS OF INTERMEDIATE CODE

We can represent Intermediate code in the form

1. Syntax tree (or) Abstract syntax tree
2. postfix notation
3) 3-address code representation.

1). ST-
* Parent node - operators
* leaf node - operands (const, var name)
* Eg:- a*(b+c)/d



2) Postfix
* If the operator appears after the operand.
* Eg :- Infix to postfix

Infix	Postfix
(a+b)*c	ab+c*
a+(b*c)	abc*+
(a-b)*(c/d)	ab-cd/*

3) Three Address code Representation

* In a 3AC instruction, each inst should contain atmost 3-address
* Also, the Right hand side must contain atmost 1 operator
* The 3AC is represented in 3 ways i.e Quadruple, Triple & Indirect Triple
* Eg = $a = b* - c + b* - c$.

1) Quadruple:-

* It contains 4 fields
  1) operator
  2) arg1
  3) arg2.
  4) result.

In $a = b* - c + b* - c$.

* Unary minus has highest priority
* The three address code for the above expression is:

$t_1 = - c.$
$t_2 = b * t_1.$
$t_3 = - c.$
$t_4 = b * t_3.$
$t_5 = t_2 + t_4$

Representing this 3 Address code in the form of Quadruples.

	op	arg1	arg2	Res
(0)	−	c		t1
(1)	*	b	t1	t2
(2)	−	c		t3
(3)	*	b	t3	t4
(4)	+	t2	t4	t5

* The major disadvantage is too many temporary variables are needed, ∴ It takes more memory

(2) TRIPLES.

* It contains only 3 field
  1) operator
  2) arg1
  3) arg2

	op	Arg1	Arg2
(0)	−	c	
(1)	*	b	(0)
(2)	−	c	
(3)	*	b	(2)
(4)	+	(1)	(3)

* The major adv of this approach is no need of temp variables

③ INDIRECT TRIPLE.

* A tuple with extra table
* The table contains pointers to the tuple.
* And the tuples info is available in TRIPLES ① ②

100	(0)
101	(1)
102	(2)
103	(3)
104	(4)

	op	Arg1	Arg2
0	−	c	
1	*	b	(0)
2	−	c	
3	*	b	(2)
4	+	(1)	(3)

THREE ADDRESS INSTRUCTION FORMS:

1). Assignment Instructions of the form
$x = y \ op \ z$
where, op is a binary operator.
Eg: $x = y + z$ (A.O)
$x = y > z$ (R.O)

2) Assignment of form $x = op \ y$;
where, op is a unary operator
Eg: $x = -y$ (Unary minus)
$x = ++y$ (Inc)
$x = !y$ (logical not)

3) Copy instructions of the form $x = y$ where value of y is assigned to x

eg
```
y = 10;
x = y;
```

4) Unconditional Jump goto L.

eg:-
```
goto L;
L: statements to be executed
```

5) a) Conditional jumps of the form if x relop y goto L.

b) Conditional jump of the form if x goto L1 or goto L2

```
if x relop y goto L.
if x > y goto L.
if x goto L1 else goto L2
```

6) Procedure calls & returns are implemented using the following

```
param x.
y = call p, n.
return y
```

7) Address & pointer asst of form

```
x = &y
x = *y
*x = y
```

8) Indexed copy Instruction of the form $x = y[i]$
$x[i] = y$

INTERMEDIATE CODE FOR FLOW OF CONTROL STATEMENTS

(or)

SDT of flow of control statements into 3-Address code

(or)

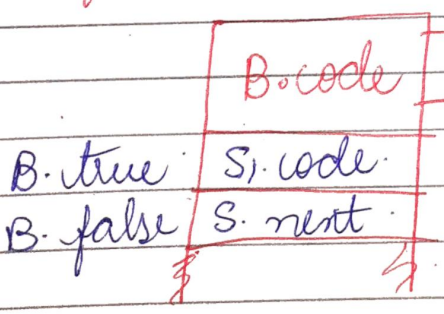
SDD. " " ".

* Control Statements
Transfering the flow from one state of the program to another state.
* In this, we are writing intermediate code for 3 concepts.
Simple if, if-else, while loop.
* We are translating those 3 control statements into 3-address code.
* Production : $S \rightarrow$ if (B) then S_1
here S and S_1 are some statements
B is a boolean expression

SIMPLE IF

* Production :- $S \rightarrow$ if (B) then S_1

* code for Simple if



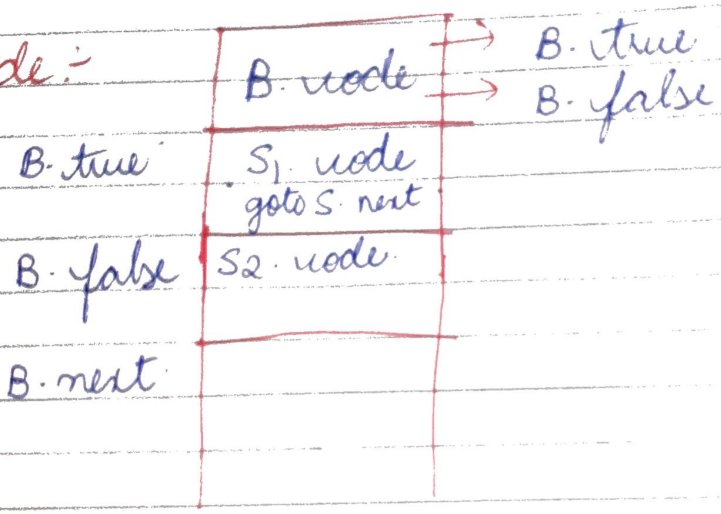
* Semantic Rule

B.true = newlabel()
S_1.next = S.next
B.false = S.next
S.code = B.code || Label(B.true) || S_1.code

② IF - ELSE

* Production : S → if (B) then S_1 else S_2

* code :



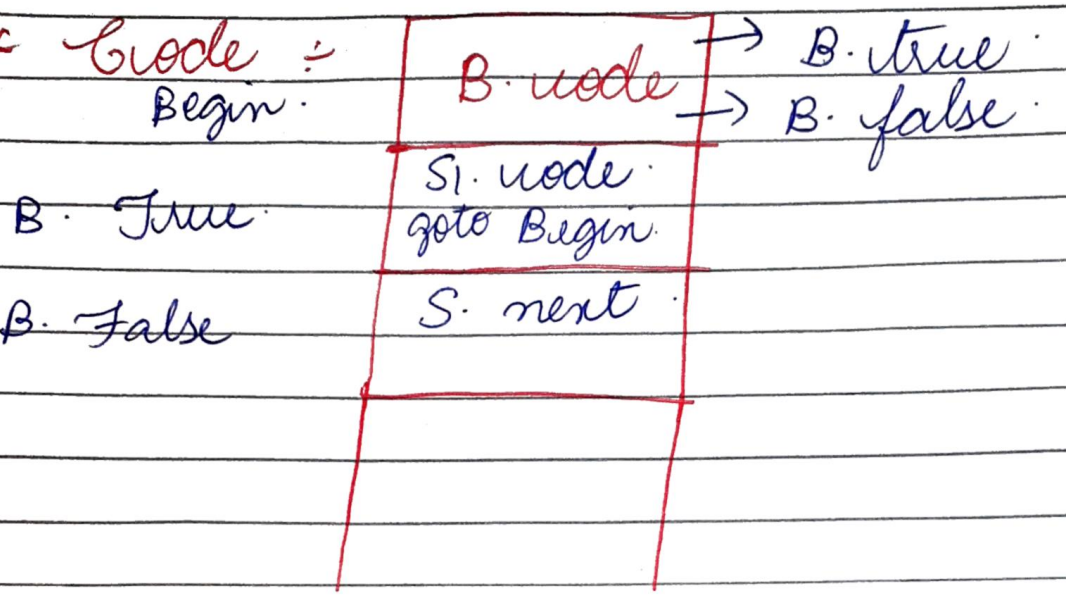
* Semantic Rules

B.true = newlabel()
S_1.next = S.next
B.false = newlabel()
S_2.next = S.next
S.code = B.code || label(B.true) ||
S_1.code || gen('goto' S.next) ||
label(B.false) || S_2.code

(3) WHILE LOOP:-

* Production :- while (B) then S_1

* Code :-

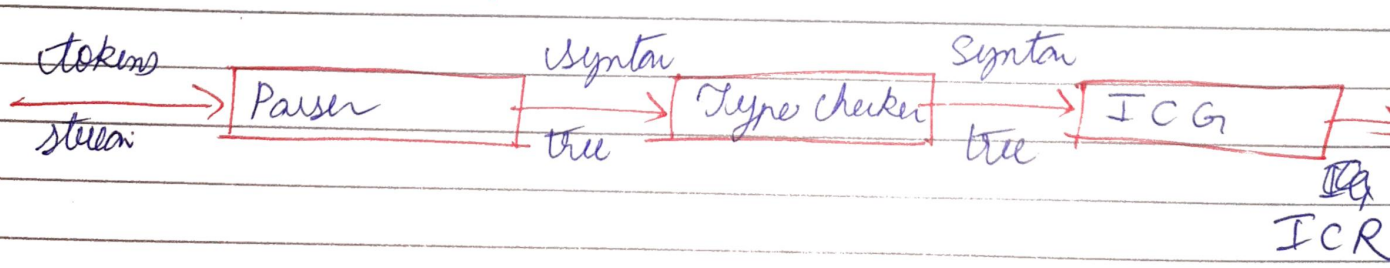


* Semantic Rules :-

Begin = new Label ().
B.true = new label ()
S_1.next = Begin
B.false = S.next.
S.code = label (Begin) || B.code || label (B.true) ||
S_1.code || gen ('goto' begin)

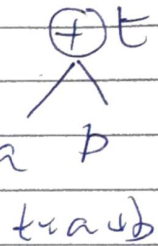
TYPE CHECKING

* The compiler has to follow the rules of a lang.
* The info abt data types is maintained & computed by the compiler.
* A type checker is a module of a compiler which is denoted to type checking tasks.
* Type checking is of 2 types.
  a) Static : Done at compile time.
  b) Dynamic: Done at run time.
* The main purpose of type checking is whether the program is correctly maintained or not is checked before execution.
* Static type checking is also useful to determine the amt of memory needed to store a variable.
* The design of Type checker depends on:
  a) Syntactic, Structure of lang constructs
  b) The type expression of a lang.
  c) The rules for assigning types to construct.
* Position of Type checker.



DAG

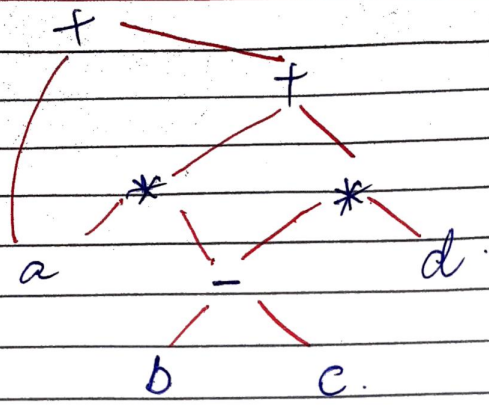
* Directed Acyclic Graph. (or)
  DAG representation of a basic block.
* The DAG represents the structure of a basic block.
* In a DAG, internal node represents operators.
* Leaf node represents the identifiers constants.
* Internal node also represents result of expressions.



Application of DAG

* Determining the common sub expressions.
* Determining which names are used inside the block & computed outside the block.
* Determining which statements of the block could have their computed value ~~xxx~~ outside the block.
* Simplifying one list of Quadruples by eliminating sub expressions.

Ex) Construction of DAG for
$a + a * (b - c) + (b + c) * a$



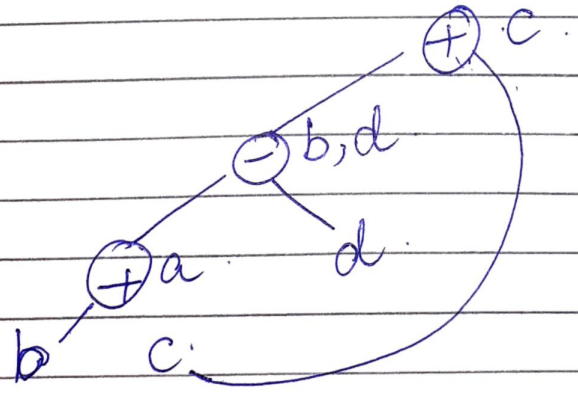
Ex 2). Construct DAG for the block.

$a = b + c$

$b = a - d$

$c = b + c$

$d = a - d$



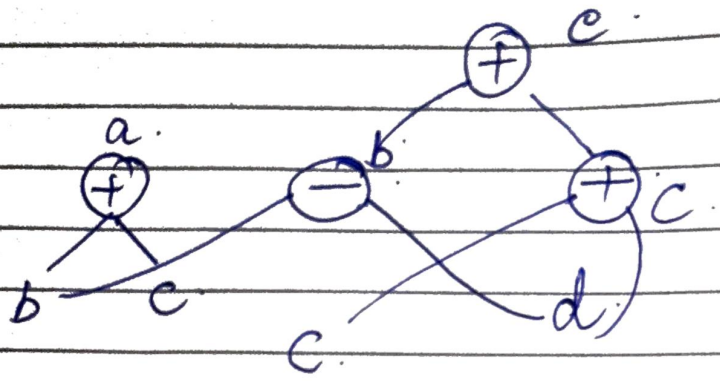
Ex 3:- Construct DAG for

$a = b + c$

$b = b - d$

$c = c + d$

$e = b + c$



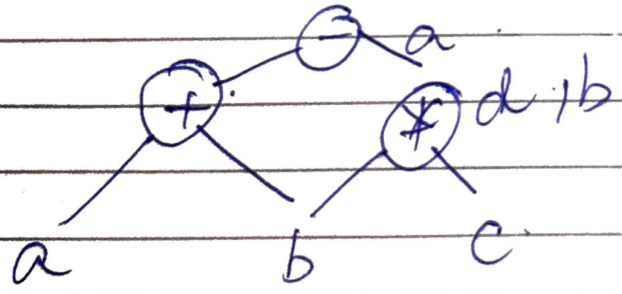
Ex (4)

$d = b * c$

$e = a + b$

$b = b + c$

$a = e - d$

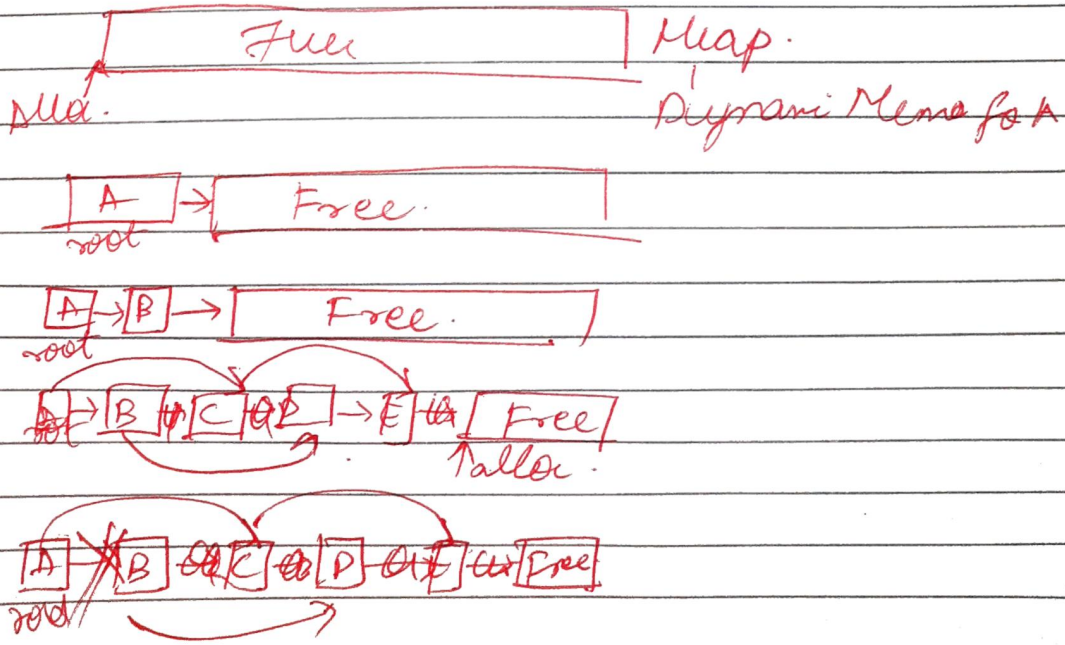


UNIT-④

INTRO TO GARBAGE COLLECTION.

* Data that can't be referenced is Garbage.
* Many high level programming lang remove the burden of manual memory management from programmer by offering automatic garbage collection which deallocates unreachable data.
* In Computer Science Garbage collection is a type of memory management that automatically frees unused objects i.e pointers in memory allowing resources to be used again.

Illustration of GC.



Now, from root A, B is not linked which makes as garbage value. Similarly its for D.
Hence GC works to remove B, D

* This garbage values are identified by considering root value & working on its link.
* This method is known TRACING.
* Garbage collection works on Four different Algo

① Mark Sweep.
② Mark Compact.
③ Copying Garbage Collector. } → TRACING.
④ Reference Counter.

TRACE BASED COLLECTION:-
— run periodically to find & unreachable objects & reclaim their space.

① Mark & Sweep Collector:-
* Straight forward, find all unreachable objects & put them on the list of free space.

Algorithm :-

1) add each obj referenced by root set to list unscanned & set its reached-bit to 1.
2) while (unscanned ≠ 0) {
3) remove some obj o from unscanned.
4) for (each obj o' referenced in o) {
5) if (o' is unreached) {
6) Set the reached bit to o' to 1.
7) put o' in unscanned.
8) free = 0;
9) for (each chunk of memory o in the heap) {
10) if (o is unreached, i.e its reached bit is 0) add o to free).
11) else set the reached-bit of o to 0;
}

free :- holds obj known to be free.
unscanned :- Reached, but successors not yet considered.

PEEPHOLE OPTIMIZATION :-

"This technique is applied to improve the performance of program by examining a short sequence of instructions in a window (peephole) & replace the instr" by a faster or short

Sequence of Instructions

Peephole Optimization Techniques

① Redundant Instruction Elimination:
Consider One Instruction

mov R0, a
mov a, R0

} mov R0, a

we can eliminate 2nd instruction
since 'a' is already in R0

② Removal of Unreachable code

Remove the statements which are unreachable i.e never executed

Ex:
```
i=0;
if(i==1)
{
    sum=0;
}
```
} i=0;

③ Flow of control Operations
using peephole optimization unnecessary jumps can be eliminated

Ex
```
goto L1.
L1: goto L2.
L2: goto L3.
L3: MOV a, R0
```

Multiple jumps can make the code inefficient. Above code can be replaced by

goto L3.

L4: goto L3.
L2: goto L3
L3: mov a, R0

④ Algebraic Simplification
Ex: x = x+0. (or) x = x*1.
One above statements can be eliminated because by executing those statements the result 'x' wont changes.

⑤ Use of Machine Idioms
It is the process of using powerful features of CPU & Instr. Ex: auto Inc/dec features can be used. to inc/dec variable

Ex: a = a-1.
sub a.

Register Allocation & Assignment

* Instructions with register operands are faster than memory operands.
* Efficient utilization of registers is important in generating good code.
* Various strategies for register allocation & assignment
  a) assign specific values in target program to certain registers
     * base addresses
     * arithmetic computations
     * top of the stack

① Global Register Allocation :-

* Keep frequently used value in a fixed register
* Assign some fixed no of registers to hold most active value in each inner loop.

② Usage Counts

* Count a savings of one for each use of x in loop L
* If x is allocated in a register then count a savings of two for each block in 'L'

$$\sum_{Blocks\ B\ in\ L} use(x,B) + 2 * live(x,B)$$

(3) Register Assignment for Outer loops.

* If an outer loop L_1 contains an inner loop L_2, the names allocated registers in L_2 need not be allocated in $L_1 - L_2$.
* If x is allocated to register L_2. Then load x on entrance of L_2 & store x on exit from L_2

(4) Register Allocation by graph Coloring

When a register is needed for computation, but all registers are in use, the contents of one register must be stored into memory location.

Two passes are used.

a) Target - machine inst are selected.
b) a register - interference graph is constructed, nodes are symbolic registers & edge connects 2 nodes.

ACTIVATION RECORD.

* Whenever a function or procedure call occurs, Activation Record gets created
* Model of AR or fields of AR.

Actual parameters.

Returned values.

Control or Dynamic Link.

Access or Static link.

Saved Machine ~~Status~~ Status.

Local Variables.

Temporary Variables.

① AP

* The parameter declared inside calling fun
* The actual parameter of the calling function are passed to the formal args of called func.

② RV :-
* Used to store the result of fun call

③ CL :-
It points to the activation rec of calling fun.

④ AL :-
It refers to the local data of called fun but found in another Activation record

⑤ SVS
Stores Address of next inst to be executed

⑥ LV :-
These variables are local to a fun

⑦ TV :-
Needed during expression evaluation.

A SIMPLE CODE GENERATOR ALGO.

* It generates target code for a sequence of instructions.
* It uses function getReg() to assign registers to variables.
* It uses 2 DS.

Register descriptor - used to keep track of which variable is stored in the register. Initially all registers are empty.

Address descriptor - used to keep track of locations where variables are stored. Locations may be register, memory address, stack.

* The following actions are performed by code generator for $x = y \text{ op } z$.
* Assume that L is the location where the output of y op z is saved

Algorithm

1) Call function getReg() to get location of L.

2) Determine the present location of 'y' by consulting address descriptor of y.

If y is not present in location L, then generate the instruction mov y, L to copy value of y to L

3) One present location of z is determined using step 2 & the instruction is generated as op z, L

4) Now L contains the value of y op z i.e Assigned to x, So y L is a register then update its descriptor that it contains value of x. Update address descriptor of x to indicate that it is stored in L.

5) If y, z have no future use, then update the descriptors to remove y & z

PRINCIPLE SOURCES OF CODE OPTIMIZATION (OR) CODE OPTIMIZATION TECHNIQUES

* Code optimization optimizes the code by removing unnecessary lines
* There are 5 techniques in code optimization

① Common Sub Expression Elimination

Common Sub expression is an expression which appears repeatedly in a program, which is computed previously but the values of variables in expression haven't changed

⊛ This technique replaces the redundant expression each time it is encountered

Unoptimized code	Optimized code
a = b+c —	a = b+c
b = a-d —	b = a-d
c = b+c —	c = b+c
d = a-d —	d = b
e = a-d —	

② Compile Time evaluation:-
Evaluation is done at compile time instead of runtime.

a) Constant folding:- Evaluate the expression & submit the result.
Ex:- area = (22/7) * r * r
Here 22/7 is calculated and result 3.14 is replaced
so, area = 3.14 * r * r

b) Constant propogation:- Here constant replaces a variable
Ex:- pi = 3.14, r = 5.
area = pi * r * r } area = 3.14 * 5 * 5

③ Code Movement / Code Motion
It is a technique which moves the code outside the loop if it won't have any difference, it executes inside or outside the loop.

Ex:- Unoptimized code.

```
for(i=0; i<n; i++)
{
    x = y+z;
    a[i] = 6*i;
}
```

Optimised code

```
x = y+z;
for(i=0; i<n; i++)
{
    a[i] = 6*i;
}
```

④ Dead code elimination

It includes eliminate those statements which are never executed or if executed the output is never used.

Ex 1. Unoptimized code

```
i=0;
if(i==1)
{
  a=x+i;
}
```

Optimized code

```
i=0;
```

Ex 2:

```
int add(int x, int y){
int z;
z=x+y;
return z;
printf("%d", z);
}
```

```
int add(int x, int y)
{
int z;
z=x+y;
return z;
}
```

⑤ Strength reduction

Replacement of These are expressions that consume more CPU cycles, time & memory. These expressions should be replaced with cheaper expressions without compromising the output.

Eg of the expression

```
b=a*2
 ↓
b=a+2
```

PARTIAL-REDUCNDANCY ELIMINATION.

* partial redundancy elimination is a compiler optimization that eliminates expression that are redundant on some but not necessarily all paths through the program.
* Partial-Redundancy elimination is an a form of common sub expression elimination

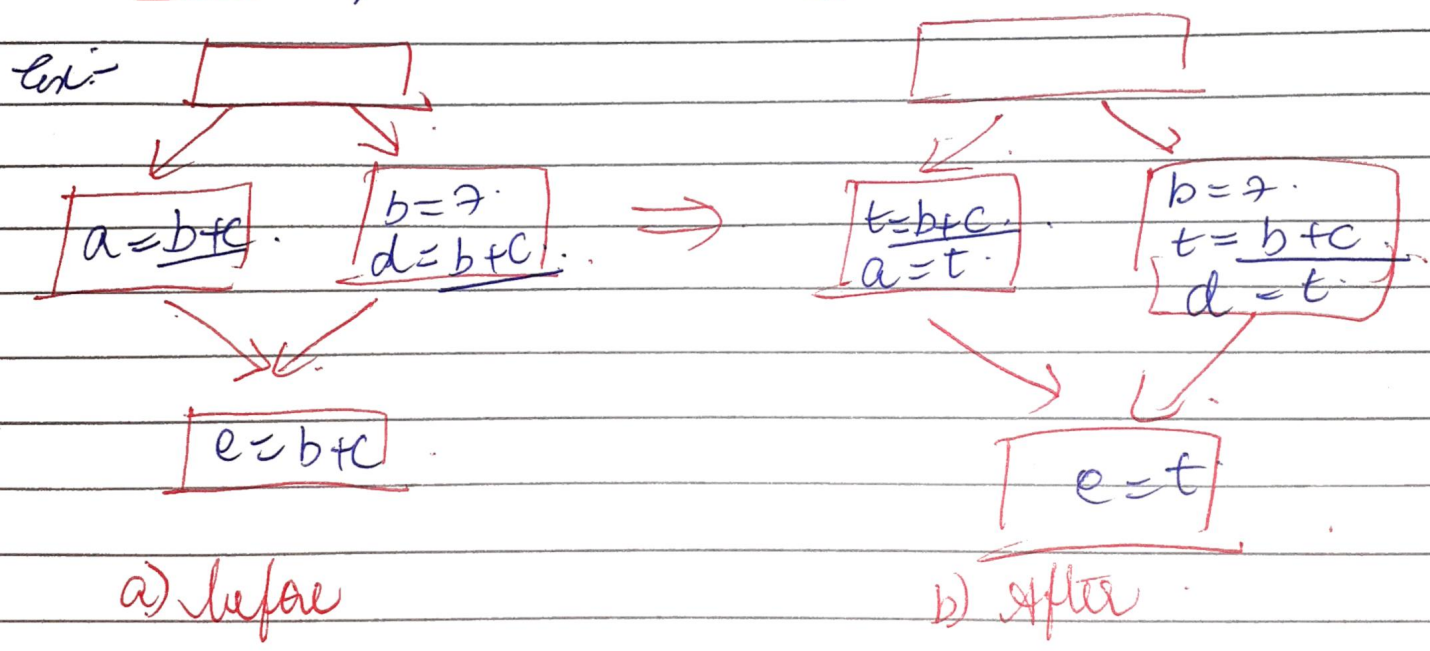


Fig:- common subexpression elimination

GLOBAL DATA FLOW ANALYSIS.

* It collects the inf about the entire program & distributes this info to each block in the flow graph
* The data flow can be collected at various points in the program by setting up & solving systems of equations
* A typical data flow eqn.

$$out[S] = gen[S] \cup [in[S] - kill[S]]$$

out [S] ⇒ Defination that reach B's exit.
gen [S] ⇒ " "within B that reach the end of B.
in [S] ⇒ that reaches B's entry.
kill [S] ⇒ that never reaches the end of B.

LOOPS IN FLOW GRAPH.

* Loop can be treated as control flow analysis
* Loops are important because program spend more time of their execution, so to improve the performance of loop is necessary
* Loop in flow graph consisting dominator & depth first ordering, back edge,

graph depth & reducibility Terms

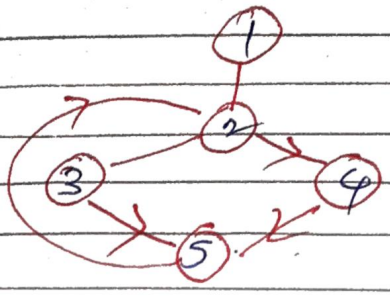
* CFG is a graph which may contain loops, known as SCC.
* Generally a loop is a directed graph whose nodes can reach all the other nodes along some path.
* This includes unstructured loops with multiple entry & multiple exit points
* Loops created by mapping high level source program to IR & are normal.
* Goto can create any loop; break creates additional exits

Terms

① Dominators :-

* A node d is said to dominate node n in a flow graph if every path to node n from initial node goes through d only.
* Every initial node dominates all the remaining nodes in a FG
* Every node also dominates itself

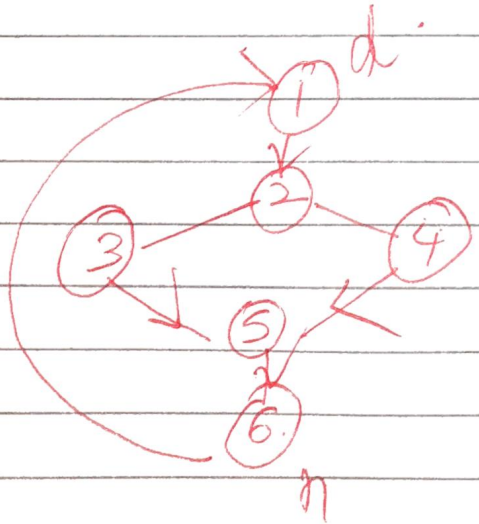
① — ② — ③



Ex:-
Node 1 dominates Node 2, 3, 4, 5 in addition to itself
Node 2 dominates 3, 4, 5 in addition to itself
Node 3 dominates itself
" 4 " " "
" 5 " ".

② Natural Loops:

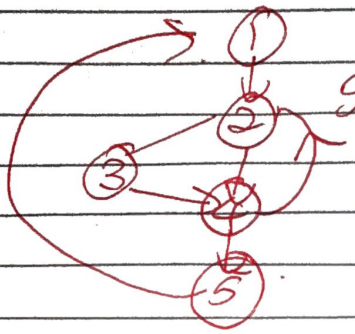
A Natural loop can be defined by a back edge $n \to d$ such that there exists a collection of all nodes that can reach to n without going to d.



Natural loop = d + {all nodes that can reach to n without going to d}

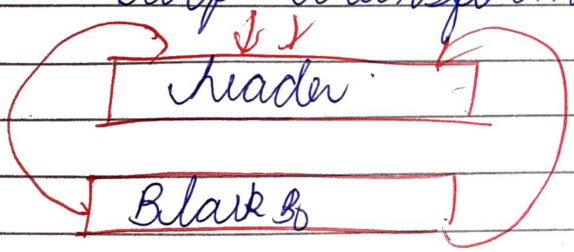
Natural loop $6 \to 1$ = {2, 3, 4, 5, 6, 1}

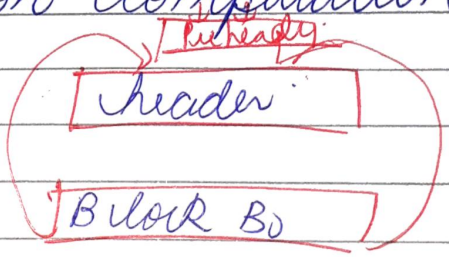
③ Inner loop:- Inner loop is a loop that can contain no other loop



Inner loop = 4 → 2.
= {2, 3, 4}

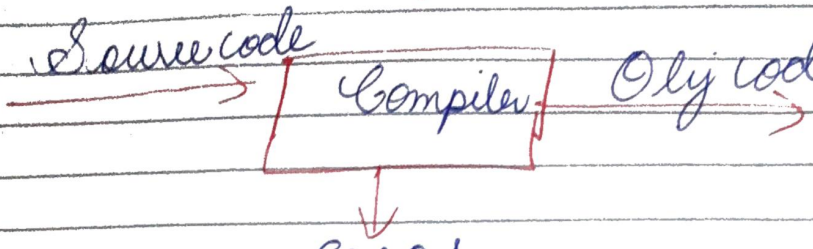
④ Pre-header:- The preheader is a new block created such that the successor of this block is a header block. It is added to facilitate loop transformation computation.





INP, UNIT-I.

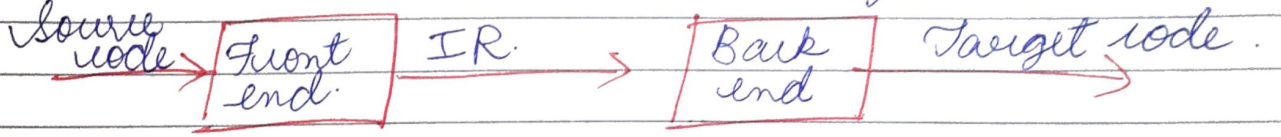
* Structure of a compiler



The compiler :-

* must generate correct code.
* must recognize errors
* analyses & synthesizis

Conceptual structure : 2 major phases



* FRONTEND
  - It performs analysis of source lang
  - Recognists legal & illegal programs & reports errors
  - produces IR & shapes the code for back end
  - much can be automated

* BACKEND
  - It doesnt target lang synthesis
  - chooses instr to implement each IR operation
  - Needs to conform with system interfaces

— automation has been less successful.

* Typically front end is O(n) while back end is NP-complete.

front end: analysis
back end: synthesis

LANGUAGE PROCESSOR :- HLL to LLL.

a) Compiler.
b) Interpiter.
c) Assembler.

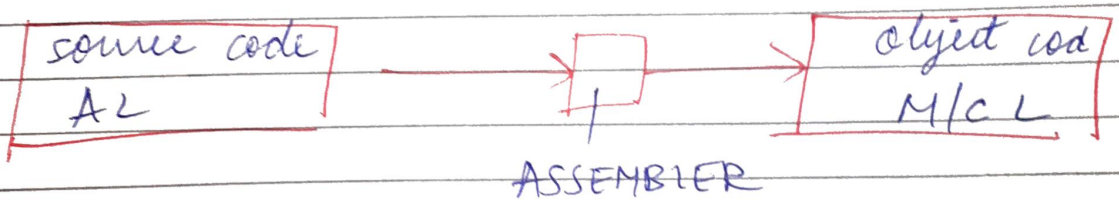
1) Compiler :-

* A lang processor that reads the complete source program written in HLL as a whole in one go and translates it into eq prog in M/C lang.
* Eg C/C++, Java.
* The source code is translated to obj code if its free of errors.

source code	→ compile →	obj code
HLL		M/C code

ASSEMBLER:-

* The Assembler is used to translate the program written in Assembly lang into machine code.
* Assembler is basically the first interface that is able to com" in humans with machine.
* The code written in Assembly lang is some sort of mnemonics like.
  ADD, MUL, SUB, DIV
* These ~~or~~ mnemonics also depend upon the architecture of the m/c



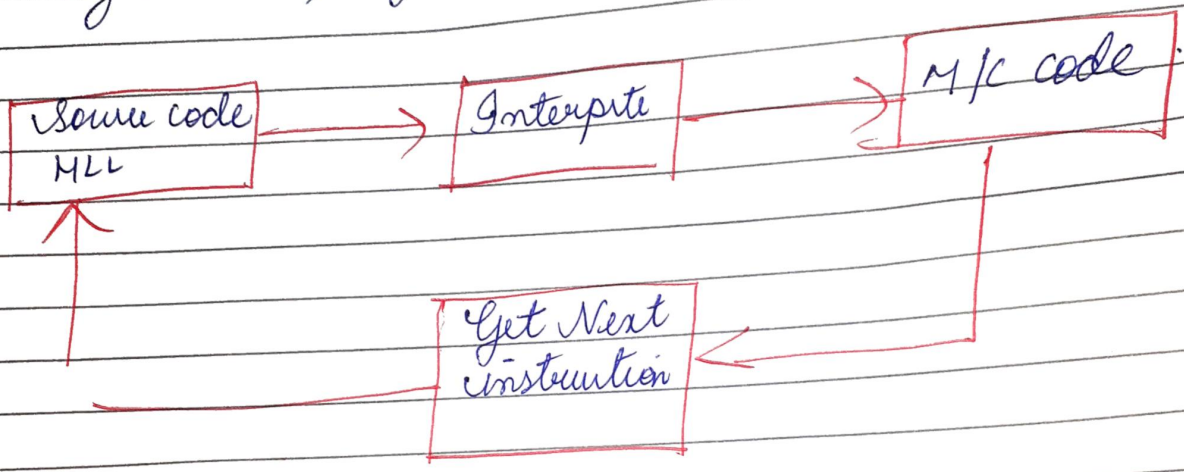
INTERPRETER

* The translation of single statement of the source program into m/c code is done & executes immediately before moving on to next line.
* If there is an error in the statement the interpeter terminats its

translating process at that stmt &
display a error msg.

* An interpreter directly executes instr written in programming lang or SL without previously rewriting them into m/c code

* Eg :- Perl, Python -



Tokens :- * Sequence of characters having a collective meaning

* It can be treated as a unit of grammar of the prog lang

* Eg :- identifier, keywords, operators, Special Symbols & constants.

* Eg of non tokens : comments, tabs, blanks, new line

Lexemes :- Lexeme is a sequence of characters in the source program that is matched for a token.

Patterns * A set of strings in the input for which the same token is produced as the output
* This set of strings is described by a rule called a pattern associated with the token.

LEX TOOL IN CD

(or)

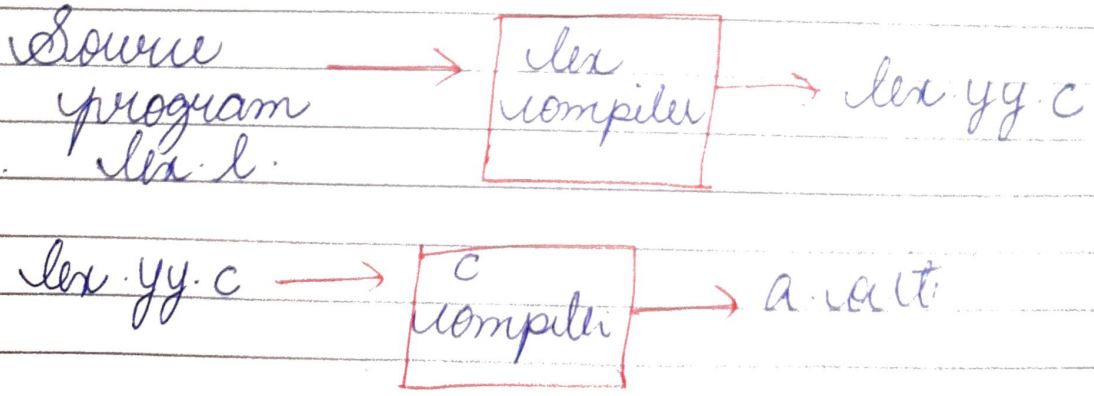
Language for specifying Lexical Analyser

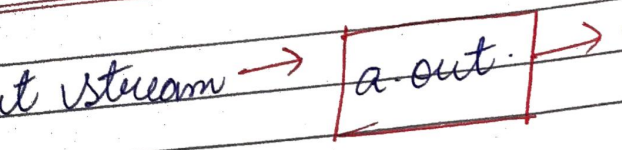
(or)

One Lexical Analyser Generator (LEX)

* LEX is a tool or a lang which is useful for generating lexical Analyser
* A lexical Analyser specifies regular expressions
* And a RE are useful for patterns of a tokens

Creating Lexical Analyser with LEX.





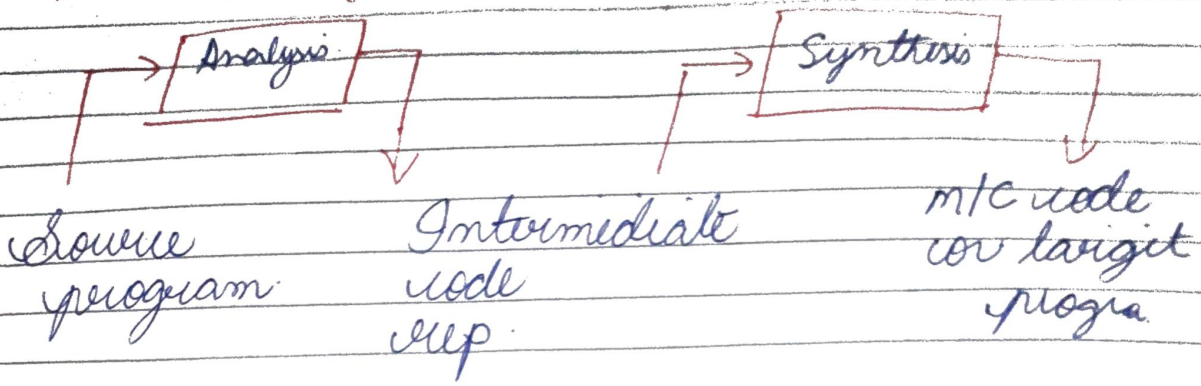
* ANALYSIS AND SYNTHESIS OF COMPILER

* A compiler operates in phases.
* A phase is a logically interrelated operation that takes sources program in one representation & produces output in another rep
* There are two parts of compilation.
  1) Analysis phase
  2) Synthesis phase

Analysis phase: It is also called m/c independent phase (or) lang dep phase. It consists of
1) Lexical Analysis or Scanning
2) Syntax Analysis or parsing
3) Semantic Analysis
4) Intermediate code generation

Synthesis Phase: It is also called m/c dependent phase or lang Independent phase. It consists of 2 phases

1) Code Optimization
2) Code Generation



→ The phases of the compiler can be grouped as

1) Front end
2) Back end

1)* Front end comprises of phases which are dependent on the input (source lang) & Independent on the target machine (target lang).
It includes
1) lexical Analysis
2) Syntax Analysis
3) Semantic Analysis
4) Intermediate code Gen

2)* Back end
Back end comprises of those phases of compiler that are dependent on the

Fig: Front end.

Lexical Analysis
↓
Syntax Analysis
↓
Semantic Analysis
↓
Intermediate code generation

Target machine & is independent on the source lang.

It includes.

Fig: Back end.

code optimization
↓
code generation.

Both front end & Back end includes both error handling & Symbol Table managmt.

WHY SEPERATE LEXICAL AND SYNTACTIC ANALYZER

* Everything can be described by a regular expression & also can be described by a grammer.

1) Seperating the syntactic structure of a lang into lexical & non-lexical parts provides a convenient way of modularizing the front end of a compiler into managable-sized components.
2) The lexical rules of a lang are frequently quite simple, & to describe them we do not need a notation as powerful as grammars
3) RE generally provide a more concise & easier to understand notation for tokens than grammars
4) More efficient lexical analysers can be constructed automatically from regular expressions than from arbitrary grammars
5) Simplicity in design
6) efficiency
7) Portability

ROLE OF LEXICAL ANALYSER

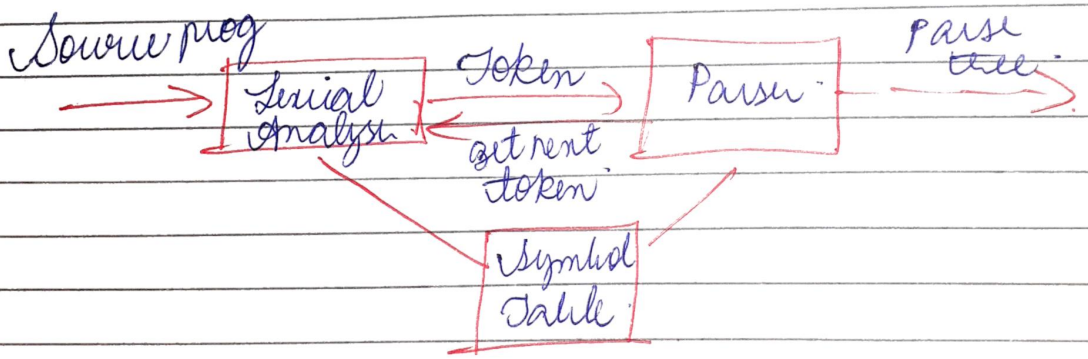
- Main task
  - → Scans & reads Input characters from source program.

- groups lexeme.
- generates tokens.

Other tasks
- Retrieves and updates symbol table.
- Eliminates white space in the comments.
- Reports error with line no.
- Expansion of macro pre processor

Two cascading process
- Scanning.
- Lexical Analysis

Interaction b/w lexical Analyser & Parser.



Example:-

```
int add (int a, int b)
// Adds two no's.
{
    int c;
    c=a+b;
    return(c);
}
```

S.no	Lexeme	Token
1	int	Keyword
2	add	Identifier
3	(	Punctuation
4	int	Keyword
5	a	identifier
6	,	P
7	int	keyword
8	b	identifier
9	)	P
10	{	P
11	int	keyword
12	c	identifier
13	;	P
14	c	identifier
15	=	O
16	a	identifier
17	+	O
18	b	identifier
19	;	P
20	return	Keyword
21	c	identifier
22	;	P
23	}	P

Issues in Lexical Analysis

* Design Simplicity
* Compiler efficiency is Improved
* Compiler portability is enhanced

ERROR RECOVERY STRATEGIES IN C.D.

Lexical error:

- Source code errors are difficult to identify.
- Example: `fi(a==y)`
  - Misspelled if statements but valid identifier.
- General lexical error: Illegal character / Unacceptable character.
- Panic mode recovery: Adequate strategy for lexical error.
  - Situation where lexical analyser can't proceed.
  - Deletes successive characters from the remaining part.
- Other possible recovery strategies:
  eg: `for(i=10; i>1; i++)`
  - Delete one character from the remaining input.
  - Insert a missing character into the remaining input.
  - Replace a character by another character.
  - Transpose 2 adjacent characters.

General Strategies

- Smallest no of transformations but quite expensive.

Unit2 - IMP

ELIMINATION OF LEFT FACTORING.

$$A \rightarrow \alpha B_1 | \alpha B_2 | \ldots \beta_1 | \gamma_2 \ldots$$

* If a grammar contains production rule like this, then we can say that the grammar contains left factoring
* Else, the top down parsers can't handle the grammar which contain left factoring
* ∴ We have to eliminate the left factoring

$$A \rightarrow \alpha B_1 | \alpha B_2 | \ldots \gamma_1 | \gamma_2 \ldots$$
$$A \rightarrow \alpha A' | \gamma_1 | \gamma_2$$
$$A' \rightarrow B_1 | B_2 | \ldots$$

* Ex 1 : $S \rightarrow iEtS | iEtSeS | a$.
$E \rightarrow b$

$$S \rightarrow \underbrace{iEtS}_{\alpha} | \underbrace{iEtSeS}_{\alpha \; B_2} | \underbrace{a}_{\gamma_1}$$

∴ $S \rightarrow iEtSS' | a$
$S' \rightarrow \epsilon | eS$
$E \rightarrow b$

* ∴ We have successfully eliminated left factoring.

Ex 2). $A \rightarrow aAB \mid aA \mid a$.
$B \rightarrow bB \mid b$.

$A \rightarrow \underset{\alpha}{a}\underset{\beta_1}{AB} \mid \underset{\alpha}{a}\underset{\beta_2}{A} \mid \underset{\alpha}{a}$.

$A \rightarrow aA'$.
~~$A' \rightarrow \alpha A'$~~
$A' \rightarrow AB \mid A \mid \epsilon$.

$B \rightarrow \underset{\alpha}{b}\underset{\beta_1}{B} \mid \underset{\alpha}{b}$.

$B \rightarrow bB'$.
$B' \rightarrow B \mid \epsilon$

∴ After eliminating left productions, the grammar is:

$A \rightarrow aA'$
$A' \rightarrow AB \mid A \mid \epsilon$.
$B \rightarrow bB'$.
$B' \rightarrow B \mid \epsilon$.

Ex3 $X \to X+X \mid X*X \mid D$
$D \to 1 \mid 2 \mid 3$.

$X \to \underbrace{X+X}_{\alpha\ \beta_1} \mid \underbrace{X*X}_{\alpha\ \beta_2} \mid \underbrace{D}_{\gamma}$.

$X \to XX' \mid D$.
$X' \to +X \mid *X$.
$D \to 1 \mid 2 \mid 3$. — no left factoring.

Ex4 $E \to T+E \mid T$.
$T \to \text{int} \mid \text{int} * T \mid (E)$.

$E \to \underbrace{T+E}_{\alpha\ \beta_1} \mid \underbrace{T}_{\alpha}$.

✓ $E \to TE'$
✓ $T' \to +E \mid \epsilon$

$T \to \underbrace{\text{int}}_{\alpha} \mid \underbrace{\text{int}}_{\alpha}\ \underbrace{*T}_{\beta} \mid \underbrace{E}_{\gamma_1}$.

✓ $T \to \text{int}\ T' \mid E$
✓ $T' \to *T \mid \epsilon \mid *T$

ERROR HANDLING IN COMPILER DESIGN.

Error Handling

- find errors.
- diagnosis.
- Error recovery.

Types of Errors.

1) Lexical errors: includes misspellings of identifies / keywords / operators

```
int xyz
print("xy3l")
```

2) Syntatic errors: includes mispelled semicolons or extra or missing braces.

```
@int
print("hi")
```

3) Semantic errors: Includes type mismatches between operator & operand.

```
int msg()
{
}
return float
```

Error Recovery

- Viable prefix property of parsers allows early detection of syntax errors

Error Recovery Strategies

① Panic Mode recovery.

* On discovering an error, the parser discards i/p symbol one at a time until one of designated set of Synchronizing tokens is found.
* The synchronizing tokens are usually delimiters such as ; or } whose role of source program is clear & unambiguous.
* While panic mode correction, often skips a considerable amt of i/p without checking it for additional errors.
* It has the advantage of simplicity & is guaranteed not to go into an infinite loop.

② Phase Level Recovery.

* On discovering an error, a parser may perform local correction on the remaining input.
* A typical local correction is to replace a comma by a semicolon, delete an extra ";" or insert a missing semicolon.

* Its major drawback is the difficulty it has in coping with situations in which the actual error has occured before the point of detection

② Statement Mode recovery

③ Error production
* If a user has knowledge of common errors that can be encounted then, these errors can be incorporated by augmenting the grammar with error productions that generate erroneous constructs
* If this is used, then during parsing appropriate error messages can be generated & parsing can be continued.
* The disadvantage is that, its difficult to maintain

④ Global Correction

* The parser examines the whole program & tries to find out the closest match for it which is error free
* The closest match has less no of insertions, deletions, & changes of tokens to recover from erroneous i/p

* Dure to high T.C & S.C this method is not implemented practically.

Semantic Errors.

These errors are detected in Semantic phase.

Recovery :-

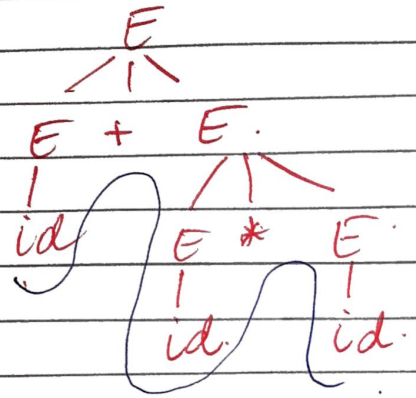
1) If the error Undeclared Identifier is encountered then, to recover from this a symbol table entry for the corresponding identifier is made.
2) If data types of 2 operands are incompatible then, automatic type conversion is done by the compiler.

* UNAMBIGUOUS GRAMMAR.

* A grammar is said to be unambiguous grammar if it generate more than one parse tree for the corresponding i/p string.
* Eg:- E+E | E*E | (E) | id

L.M.D.

$E \rightarrow E + E$
$\rightarrow id + E$
$\rightarrow id + E * E$
$\rightarrow id + id * E$
$\rightarrow id + id * id$



R.M.D.

$E \rightarrow E + E$
$\rightarrow E + E * E$
$\rightarrow E + E * id$
$\rightarrow E + id * id$
$\rightarrow id + id * id$