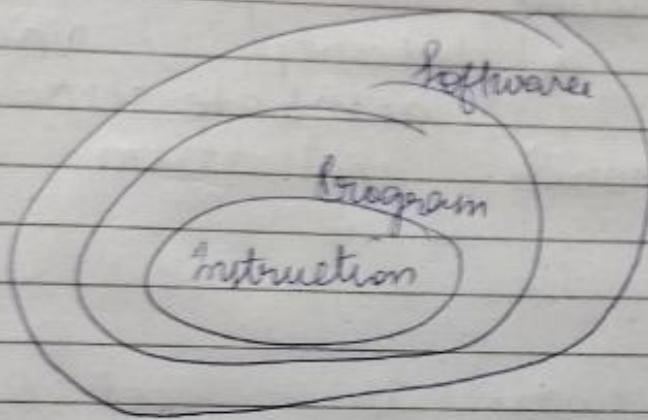


SPCC (System programming & computer construction)

Model - 1

Overview of system software



Types of software

- ① System s/w linker
Windows, OS, macOS, compiler, debugger, editor.
- ② Application s/w

System software

System software is a software which is used to run the system for user point of view. for
for eg. OS (operating system), Assembler, compiler, macro and microprocessor, linker, loader, editor, debugger etc.
Without system software the system

Can't run.

Application software

Application software is a software which is used for executing a particular application. For eg. Photoshop, VNC media player, acrobate reader, mobile application etc.

Software

Software is a set of program which are executed sequentially to get the desired output.

Program is a set of instruction which are executed sequentially to get the required output.

Instruction:

Instruction can be any programmable executable statements (having predefined syntax or command).

Programming software

Database

B3

Engineering & Science

Application software

- ① UI
- ② It is used for specific purpose.
- ③ It executes

System software

- ① UI
- ② It is used for general purpose.
- ③

Applications software

- ① Application software is build for specific task.
- ② UI are used to write the application software.
- ③ Without application software, system always run.
- ④ Application software run as per user request.

System software

- ① System software used for application software to run.
- ② UI are used to write the system software.
- ③ Without system software, system can't run.
- ④ System software run when the system is turned ON and stop when the system is turned off.

⑤ Specific purpose software

⑥ Example: Adobe reader, Photoshop, VLC media player

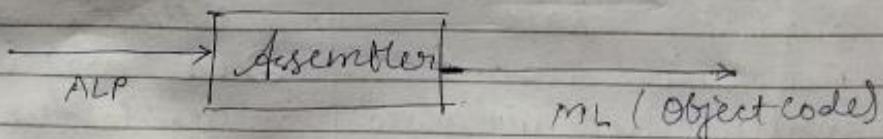
⑦ General purpose software

⑧ OS, Assembler, compiler, macro processor, loader, linker etc.

Types of System software's

① Assembler:

Assembler is a language translator that takes as input the Assembly language program and generates its machine language equivalent code or object code.



② Macro processor:

Macro is defined as the single line abbreviation for group of instructions for or code. It is an additional facility given to the ALP.

Definition of Macro

MACRO

MACRONAME

=
=

MEND

} group of statements to be executed.

DIA:

Macroprocessor is a program which is responsible for processing of MACRO (handles all macro calls and performs macro expansion).

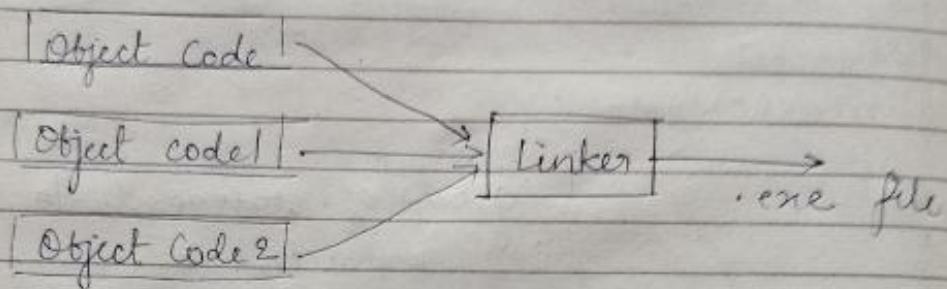
Compiler

Compiler is a language translator that takes as input the source program i.e. written in high level language and produces as output an assembly language or machine language.

Translator

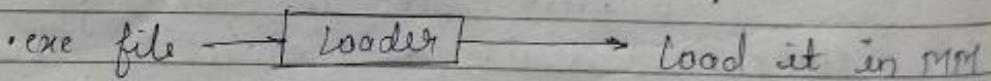
Translator is a system program that converts program in one language to program in another language.
e.g. Assembler computer

* Linker.



Linker A program that combines object files generated by compiler, assembler and other pieces of codes to create an executable file with .exe extension.

* Loader.



A program that takes .exe file from linker and load it into main memory and prepare it for execution by computer.

Editor

It is a tool which is used to write a particular code.

* Debugger.

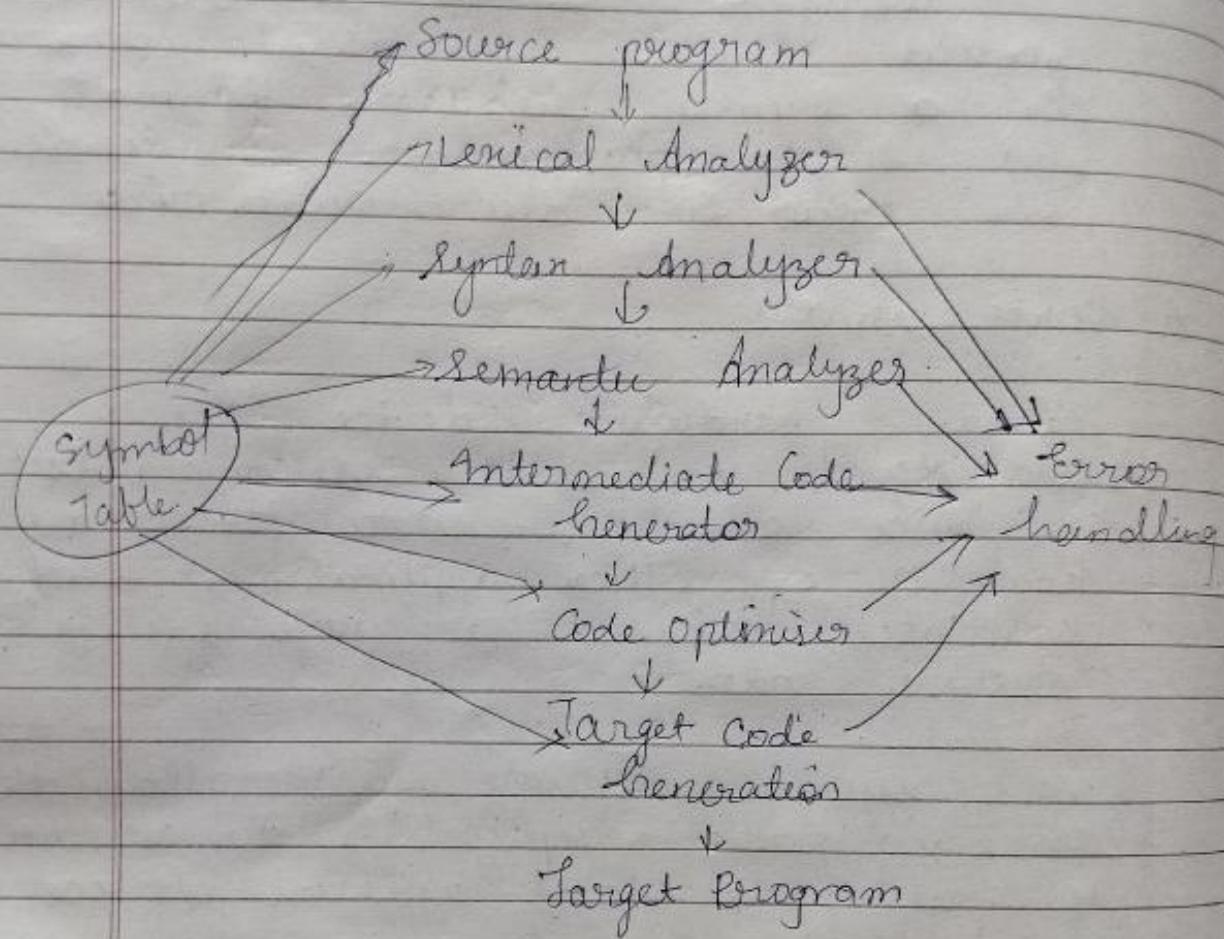
Debugger comes in used to present computer generated source in a human readable format and helps programmer in finding the errors in the source code.

* Device drivers.

Every hardware device can understand only its HL commands; a device driver translates HL commands from OS or other applications and translates it into hardware specific machine code.

A system software that allows the OS and other applications to communicate with specific hardware device.

Phase of Computer



Model 2: Lexical analysis

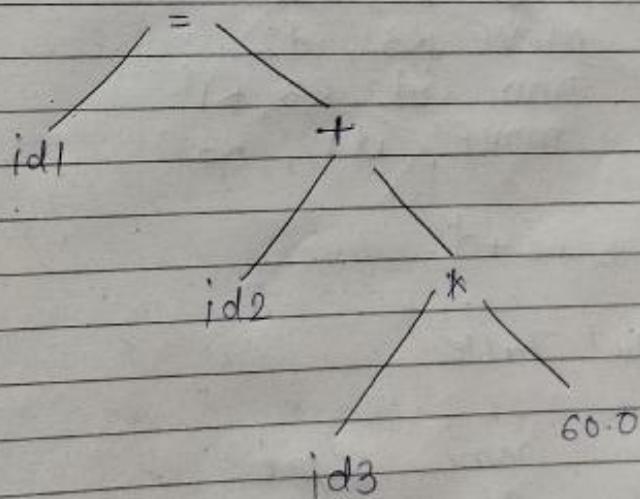
$$\text{position} = \text{initial} + \text{rate} * 60.$$

symbol table

Symb			
id	& name	type	
id1	position	float	
id2	initial	float	
id3	rate	float	-

$$id1 = id2 + id3 * 60.0$$

Syntax analyzer / semantic analyzer



ICG BCAF (3 code Address form)

$$t_1 = id_3 * 60.0$$

$$t_2 = id_2 + t_1$$

$$id_1 \leftarrow t_2 = id_1 + t_2$$

Code optimizer

$$t_1 = id_3 * 60.0$$

$$id_1 = id_2 + t_1$$

Target code generation

```

B MOVF R1, 60.0
MOVF R2, id3
MUL R1, R2
MOVF R3, id2
ADD id2 R3, R1
MOVF id1, R3

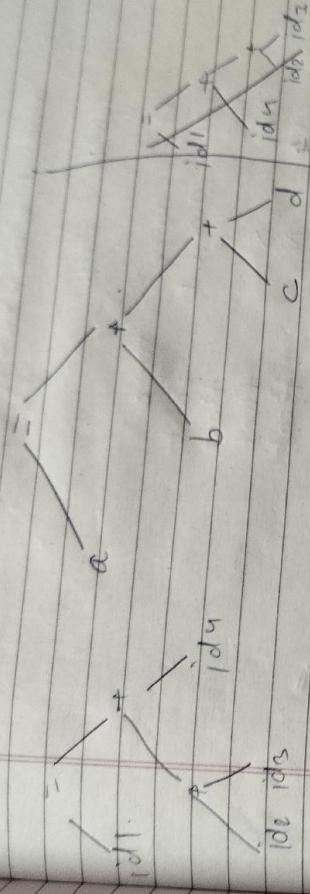
```

$$a = b + c + d.$$

Symbol Table

id	name	type
id1	a	float int
id2	b	float int
id3	c	float int
id4	d	float int

$$id_1 = id_2 + id_3 + id_4$$



TC L7 (3CAF)

$$\begin{aligned} t_1 &= \cancel{id_2} + id_3 \\ t_2 &= b + \cancel{t_1} \\ id_b &= \cancel{b} + t_1 \end{aligned}$$

Optimized.

$$\begin{aligned} t_1 &= x + \cancel{d} \\ a &= \cancel{b} + t_1 \end{aligned}$$

Target code generation

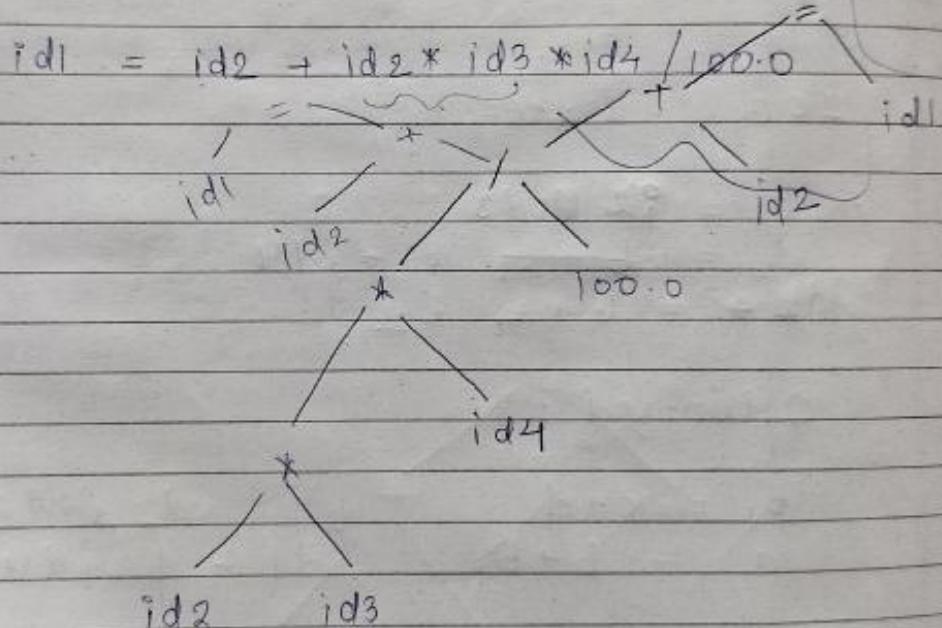
```

MOV R1, id2
MOV R2, id3
MOV ADD R1, R2
MOV R3, id4
ADD R1, R3
MOV id1, R1
  
```

$$\text{amount} = P + P \times R * N / 100$$

symbol table

id	name	type
id1	amount	float
id2	P	float
id3	R	float
id4	N	float



ICG

$$t_1 = id2 * id3$$

$$t_2 = id2 * t_1$$

$$t_3 = t_2 / 100.0$$

$$t_4 = id2 + t_3$$

$$id1 = t_4$$

Code optimizer

$$t_1 = id_2 * id_3$$

$$t_2 = t_1 * id_4$$

$$t_3 = t_2 / 100.0$$

$$id_1 + id_4 = id_2 + t_3$$

target code generation

MOVF R₁, id₂

MOVF R₂, id₃

MUL R₁, R₂

MOVF R₃, id₄

MUL R₁, R₃

MOVF R₄, 100.0

DIV R₁, R₄

~~ADD~~ MOV R₅, id₂

ADD R₅, R₁

MOVF id₁, R₅

lexical analyzer

Primary fn \Rightarrow to generate tokens
Secondary fn

① remove blank spaces

② remove comments

③ Communicating with the symbol table

④ Correlate error LA invalid token error
It parses correct token

SA

identifier

$r = (\text{letter}) (\text{letter/digit})^* \text{ delimiter}$

The secondary fn of lexical analyzer

- ① Remove comment
- ② Remove white spaces
- ③ Communication with symbol table i.e. storing information regarding an identifier or variable in the symbol table.
- ④ Correlate error (if the lexical analyzer finds a token invalid it generates an error and passes only valid tokens to the syntax analyzer when it demands)
- ⑤ Peing

* Program that performs lexical analysis is called Lexical analyzer.

* Regular expression is used to specify the programs.

* The advantage of using regular expression is that a recognizer for the token called.

finite automata could be easily constructed

- * While constructing the lexical analyzer we first design the regular expression for the token.

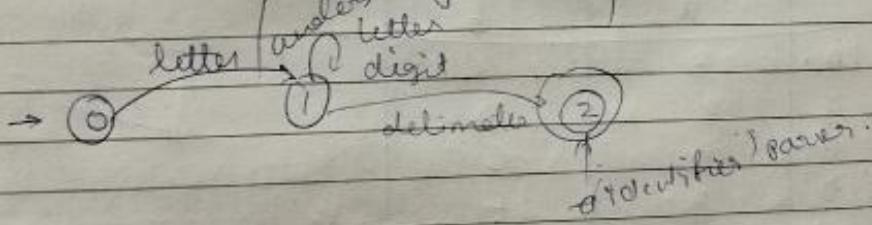
Finite automata is a recognizer for a language L that takes an input string x and answer "Yes" if the x is a sentence of L and "no" otherwise.

Regular expression for an identifier

$$r = (\text{letter}) (\text{letter/digit})^* \text{ delimiter}$$

↑ ↑
 $(a-z)(a-z)$ $(0-9)$

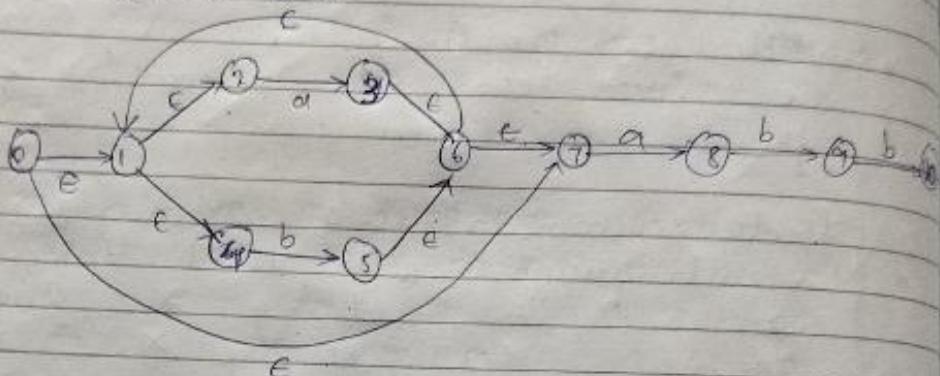
Transition diagram for an identifier



State 2 indicates that an identifier has been found and ~~takes~~ a token is generated and returned to the parser/syntax analyzer.

Design DFA for the regular expression

$$r = (a+b)^* a b b$$



x	$y = \epsilon\text{-closure}(x)$	$S(y, a)$	$S(y, b)$
A $\{0\}$	$\{0, 1, 2, 4, 7\}$	$\{3, 8\}$	$\{5\}$
B $\{3, 8\}$	$\{3, 8, 6, 12, 4, 7\}$	$\{3, 8\}$	$\{5, 9\}$
C $\{5\}$	$\{6, 1, 2, 4, 7\}$	$\{3, 8\}$	$\{5\}$
D $\{5, 9\}$	$\{6, 1, 2, 4, 7, 5, 9, 10\}$	$\{3, 8\}$	$\{5\}$
Ex $\{5, 10\}$	$\{5, 6, 1, 2, 4, 7, 10\}$	$\{3, 8\}$	$\{5, 10\}$

	a	b
A.	B	C
B.	B	D
C.	B	C
D.	B	E
E*	B	C

sudo apt-get install flex
gedit demo.l

len demo.l

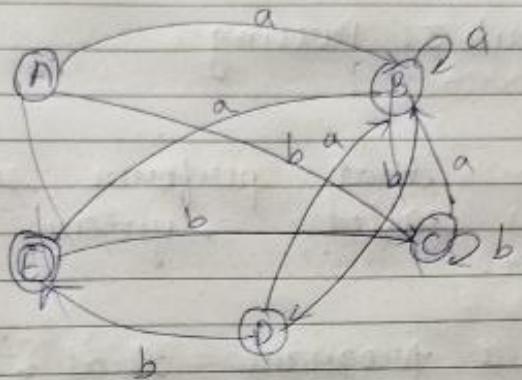
unzip

Page No.

90

.l/a.out

get



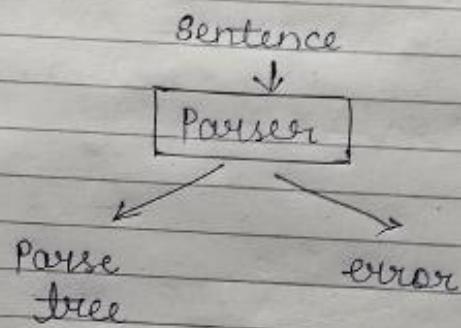
	a	b
A C	B	A C
B	B	D
D	B	E
G*	B	A C

Module 3: Parsing

4 Parser

Programme that perform syntax analysis is called syntax analyzer or parser.

Parser is a program that takes a input a sentence or string and if the grammar can derive the sentence it generates a parse tree else it generates an error.



Parsing techniques

- | | |
|---|--|
| 1) top down parser
The parser builds the parse tree from top to bottom i.e. from root to the leaves. | Bottom up parser
The parser builds parse tree from bottom to top i.e. from leaves to top.
↳ Predictive parser:
CLL) Shift red operator precedence |
| 2) Recursive descent parser (RDP)
Predictive parser (LL) | |

syntax
analyzer

takes as
and if
the
tree

parser
built the
from bottom
from
top:
syntax:
left reduce
precedence

Simple LR
Canonical parse
look ahead parser.

* Recursive descent parser (RDP).

Design recursive descent parser for
the following grammar.

$$\begin{aligned} S &\rightarrow AB \\ A &\rightarrow aA/b \\ B &\rightarrow bB/a \end{aligned}$$

function $S()$
{
 $A();$
 $B();$
}

function $A()$
{
 if (input_symbol == 'a')
 ADVANCE();
 $A();$
 }
 else

{
 if (input_symbol == 'b')
 ADVANCE();
}

```

else {
    ERROR();
}

```

```

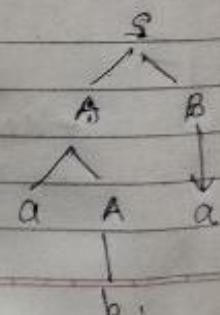
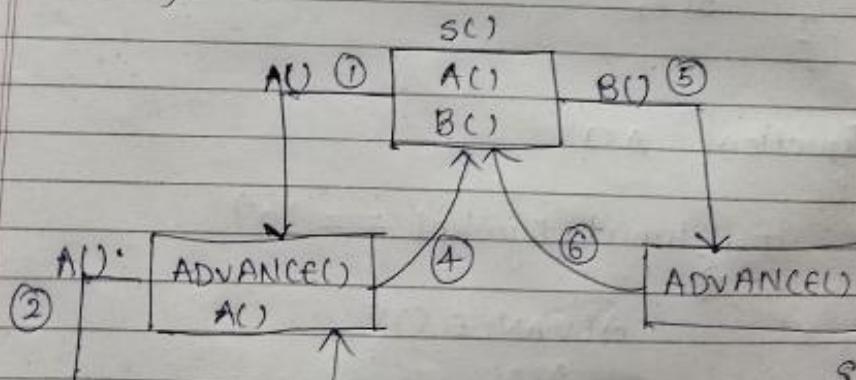
function B() {
    if (input_symbol == 'b') {
        ADVANCE(); // Points to the next symbol.
        B();
    }
}

```

```

else {
    if (input_symbol == 'a') {
        ADVANCE();
    } else {
        ERROR();
    }
}

```



$E \rightarrow E + E | id$
 $E \rightarrow E * E$
 $E \rightarrow id$.

function E() {
 if (input symbol == 'id')
 { ADVANCE();
 }
 else {
 E();
 if (input symbol == '+')
 { ADVANCE();
 E();
 }
 else {
 ERROR();
 }
 }
}

For id + id

not scanned as
 after scanning first id it will
 terminate the function. There
 will be no parse tree.

In the above example parser goes
 into an infinite loop because the
 given grammar is left recursive

and hence not suitable for parser design.

Left recursive

Any production of the form

$$A \rightarrow A\alpha | \beta$$

Elimination procedure

Eliminate the left recursive grammar.

$$A \rightarrow \beta A'$$

$$A' \rightarrow \alpha A' | e$$

or parser

form

utive

$$E \rightarrow E + E' | id$$

$$E' \rightarrow id E'$$

$$E' \rightarrow + E E' | e$$

function $E()$ {

if (input_symbol == 'id') {

ADVANCE();

$E'()$;

}

else {

ERROR();

}

function $E'()$ {

if (input_symbol == '+') {

ADVANCE(); $E()$; $E()$;

if (input_symbol == 'E') { $E()$;

ADVANCE();

else { ERROR();

if (input_symbol == 'E') {

ADVANCE(); $E'()$;

ADVANCE();

else { ERROR(); }

else { ERROR(); }

}

}

else if (input symbol == 'c') {

3
else {
 NOACTION();
 ERROR();
}

Left recursion

A production of the grammar is said to have a left recursion if the left most variable of its RHS is same as the variable of its LHS.

$$A \rightarrow A\alpha | \beta$$

$$\begin{aligned} A &\rightarrow \beta A' \\ A' &\rightarrow \alpha A' / \epsilon \end{aligned}$$

General format.

$$A \rightarrow A\alpha_1 | A\alpha_2 | \dots | A\alpha_n$$

$$A \rightarrow \beta_1 | \beta_2 | \dots | \beta_n$$

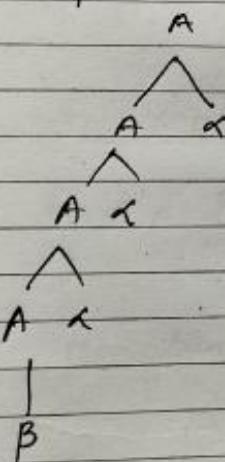
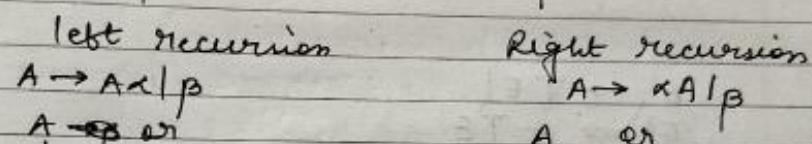
$$A \rightarrow \beta_1 A' | \beta_2 A' | \dots | \beta_n A'$$

$$A' \rightarrow \alpha_1 A' | \alpha_2 A' | \dots | \alpha_n A' | \epsilon$$

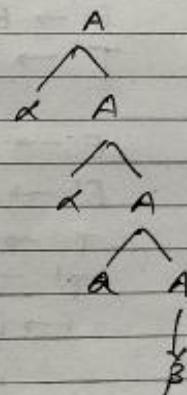
Top down parser cannot handle the grammar having left recursion because they are not suitable for parser design.

Ideally we have to remove or eliminate
left recursion but preserve the language
generated by grammar.

Grammar



language : β^*



language: $\alpha^* \beta$

There are two types of left recursion

- ① Direct left recursion
Eg. $B \rightarrow Ba$

② Indirect left recursion

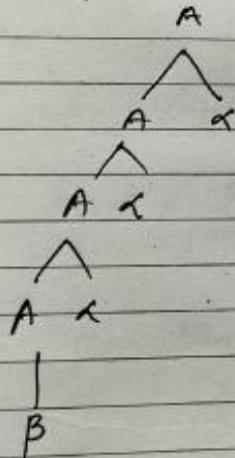
$$S \rightarrow Aa$$

We have to remove or eliminate recursion but preserve the language generated by grammar.

Grammar

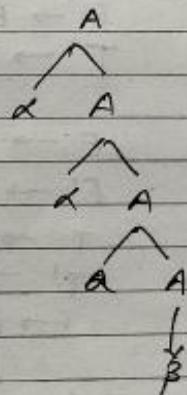
left recursion
 $A \rightarrow A\alpha | \beta$

$$A \rightarrow \underset{\downarrow}{\alpha} \text{ or } \beta$$



Right recursion
 $A \rightarrow \alpha A | \beta$

$$A \rightarrow \underset{\downarrow}{\beta} \text{ or } \alpha$$



language: $\beta\alpha^*$

language: $\alpha^*\beta$

There are two types of left recursion

- ① Direct left recursion

Eg. $B \rightarrow Ba$

- ② Indirect left recursion

$$\begin{aligned} S &\rightarrow Aa \\ A &\rightarrow Sa \end{aligned}$$

Eliminate the left recursion from the following grammar.

$$\begin{aligned} E &\rightarrow E + T / T \\ T &\rightarrow T * E / F \\ F &\rightarrow \text{id} \end{aligned}$$

$$\begin{array}{l} E \rightarrow TE' \\ E' \rightarrow + TE \end{array}$$

$$\begin{aligned} E &\rightarrow E + T / T \\ T &\rightarrow T * E / \text{id } F \\ E &\rightarrow TE' \\ E' &\rightarrow + TE' | \epsilon \\ T &\rightarrow FT' \\ T' &\rightarrow * ET' | \epsilon \\ F &\rightarrow \text{id}. \end{aligned}$$

function $E()$ {
 T();
 E'();
 }

function $E'()$ {
 if (input-symbol == '+') { ADVANCE;
 T();
 E'();
 } ADD;

from

```
else {  
    error();  
}  
function T() {  
    ADVANCE();  
    function FC();  
    T();  
  
function T'() {  
    if (input-symbol == 'x') {  
        ADVANCE();  
        FC();  
        T();  
    }  
    else if (  
        NO ACTION();  
    }  
    function
```

ADVANCE();

$$A \rightarrow Ba | Aa | c$$

$$B \rightarrow Bb | Ab | d$$

$$A \rightarrow A ba | Aa | c$$

$$B \rightarrow B b | B a b | d$$

$$A \rightarrow C A'$$

$$A' \rightarrow ba A' | a A' | e$$

$$B \rightarrow d B'$$

$$B' \rightarrow b B' | ab B' | d$$

$$S \rightarrow Aa | b$$

$$A \rightarrow Ac | Sd | e$$

$$S \rightarrow Sda | b$$

$$A \rightarrow Ac | Aad | e$$

$$S \rightarrow b S'$$

$$S' \rightarrow das' | e$$

$$A \rightarrow \cancel{c} A'$$

$$A' \rightarrow c A' | ada' | e$$

Left factoring

There are two different types of grammar :

- ① Deterministic grammar : It defines where the want to deterministic exactly. In deterministic grammar there is exactly one unique prefix -

- ② Non-deterministic grammar set There is exactly one unique prefix .

$$A \rightarrow \alpha B \mid \alpha P$$

The main cause of ↑ Non-deterministic grammar in repetition of prefixes. Due to this the parser is not able to determine to which production it wants to go. Therefore the left factoring comes into the picture. In left factoring convert non-deterministic grammar to deterministic grammar.

Any production of the form

$$A \rightarrow \alpha B \mid \alpha V$$

common prefix

Left factoring

There are two different types of grammar:

① Deterministic grammar

It defines where we want to go exactly. In deterministic grammar there is exactly one unique prefix.

② Non-deterministic grammar

There is exactly one unique prefix.

$$A \rightarrow \alpha\beta \mid \alpha\nu$$

The main cause of

→ Non-deterministic grammar is repetition of prefixes. Due to this the parser is not able to determine to which production it wants to go. Therefore left factoring comes into the picture. In left factoring convert non-deterministic grammar to deterministic grammar.

At any production of the form

$$A \rightarrow \alpha\beta \mid \alpha\nu$$

common prefix

Now to eliminate left factoring

$$A \rightarrow \alpha A'$$

$$A' \rightarrow \beta | \gamma$$

General format.

$$A \rightarrow \alpha, \beta, |\alpha, \beta_2| \dots |\gamma_1, \gamma_2$$

\Downarrow_1

$$A \rightarrow \alpha, \alpha' | \gamma_1, \gamma_2$$

$$\alpha' \rightarrow \beta_1, \beta_2$$

Eliminate left factoring from the grammar.

$$(i) S \rightarrow aSb | ab$$

$$(ii) A \rightarrow aB | abc$$

$$(iii) S \rightarrow iEts | iEtses | a$$

$$E \rightarrow b$$

$$(iv) S \rightarrow \cancel{aabb} assbs | aSasb | abb | b$$

$$(i) S \rightarrow aSb | ab$$

$$S \rightarrow aS'$$

$$S' \rightarrow Sb | b$$

(ii)

$$A \rightarrow aB \mid abc$$

$$A \rightarrow aA'$$

$$A' \rightarrow B \mid bc$$

(iii)

$$S \rightarrow i \cancel{Ets} \mid iEtSeS \mid a$$

$$\begin{array}{l} S \rightarrow iEtS \mid a \\ S \rightarrow eS \mid e \\ E \rightarrow b. \end{array}$$

~~$$\begin{array}{l} S \rightarrow iS \mid a \\ S \rightarrow EtS \mid EtSeS \\ E \rightarrow b \end{array}$$~~

(iv)

$$S \rightarrow assbs \mid asasb \mid abb \mid b$$

$$S \rightarrow ass \mid abb \mid b$$

$$S' \rightarrow sbs \mid asb$$

~~3ACF~~

3ACF (3 address code form).

$$a = (a+b) + (b-c)$$

$$t_1 = a \wedge b$$

$$t_2 = b - c$$

$$t_3 = t_1 + t_2$$

$$a = t_3.$$

FIRST(α)

It is defined as a set of terminals that begins with the string derived from α (α is some sentential form).

Rule 1:

$R_1 \rightarrow$

If $x \rightarrow a\alpha$ then $\text{first}\{x\} = \{a\}$

Rule 2:

If $x \rightarrow G$ then $\text{first}\{x\} = \{e\}$

Rule 3:

If a is a terminal then $\text{first}\{a\} = \{a\}$

Rule 4:

If $x \rightarrow y_1 y_2 \dots y_k$ then $\text{first}\{x\} = \text{first}\{y_1\}$

If y_j contains e then,

$$\text{first}\{x\} = (\text{first}\{y_1\} - \{e\}) + \text{first}\{y_2\}$$

Note: If y_j contains e then ~~FIRST~~
 $\text{FIRST}\{x\} = e$ where $j = 1, 2, 3$

terminal
being
sentential

FOLLOW(A)

FO α is defined as a set
of terminals that can appear
on RHS of variable A.

Rule 1:

If y is a start variable
then $\text{FOLLOW}(y) = \{\$y\}$

Rule 2:

If $x \rightarrow \underline{x}y$.

then $\text{FOLLOW}(y) = \text{FOLLOW}(x)$

Rule 3:

If $x \rightarrow \underline{x}y\beta$ then

$\text{FOLLOW}(y\beta) = \text{FIRST}\{\beta\}$

If $\text{FIRST}\{\beta\}$ contains ϵ , then

$\text{FOLLOW}(y) = \text{Follow}(x) \cdot \text{First}\{\beta\} - \{\epsilon\}$
+ $\text{Follow}(x)$

$x \rightarrow \underline{x}y \circledcirc$

terminal
being
sentential

FOLLOW(A)

FO α is defined as a set
of terminals that can appear
on RHS of variable A.

Rule 1:

If y is a start variable
then $\text{FOLLOW}(y) = \{\$ \}$

Rule 2:

If $x \rightarrow \underline{x} y$.

then $\text{FOLLOW}(y) = \text{FOLLOW}(x)$

Rule 3:

If $x \rightarrow x Y \beta$ then

$\text{FOLLOW}(Y\beta) = \text{FIRST}(\beta)$

If $\text{FIRST}(\beta)$ contains ϵ , then

$\text{FOLLOW}(Y) = \text{Follow}(x) \cdot \text{First}(\beta) - \{\epsilon\}$
+ $\text{Follow}(x)$

$x \rightarrow x Y \beta$

Q. find first & follow of the given grammar.

$$S \rightarrow ABC$$

$$A \rightarrow a/c$$

$$B \rightarrow b/c$$

$$C \rightarrow c/e$$

$$\text{First } \{a\} = \{a\}$$

$$S = \{a, b\}$$

$$\text{First } \{b\} = \{b\}$$

$$\text{First } \{c\} = \{c\}$$

4)

$$\text{First } \{\epsilon\} = \{\epsilon\}$$

$$\text{First } \{b\} = \{b, \epsilon\}$$

$$\text{First } \{c\} = \{c, \epsilon\}$$

$$\text{Follow}(S) = \{\$\}$$

same question as above
 $B \rightarrow b$

$$\text{Follow}(A) = \{b\}$$

$$\text{Follow}(B) = \{c, \$\}$$

$$\text{Follow}(C) = \{\$\}$$

$$\begin{aligned} E &\rightarrow TE' \\ E' &\rightarrow +TE' | e \\ T &\rightarrow (E) | id \end{aligned}$$

$$\text{First } \{+\} = \{+\}$$

$$\text{First } \{(\} = \{(\}$$

$$\text{First } \{) \} = \{) \}$$

$$\text{First } \{id\} = \{id\}$$

given

Page No. _____
Date _____

$$\text{First } \{E\} = \{ (, \text{id} \}$$

$$\text{First } \{E'\} = \{ +, \epsilon \}$$

$$\text{First } \{T\} = \{ (, \text{id} \}$$

$$\text{Follow } \{E\} = \{ \$,) \} \rightarrow \text{as } E \text{ is a symbol}$$

$$\text{Follow } \{E'\} = \{ \$,) \}$$

$$\text{Follow } \{T\} = \{ +, \$,) \}$$

4)

$$E \rightarrow TE'$$

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

$$T \rightarrow FT'$$

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

$$F \rightarrow (\epsilon) | \text{id} .$$

$$\text{First } \{+\} = \{ + \}$$

$$\text{First } \{* \} = \{ * \}$$

$$\text{First } \{(\} = \{ (\}$$

$$\text{First } \{) \} = \{) \}$$

$$\text{First } \{\text{id}\} = \{\text{id}\}$$

$$\text{First } \{E\} = \{ (, \text{id} \}$$

$$\text{First } \{E'\} = \{ +, \epsilon \}$$

$$\text{First } \{T\} = \{ (, \text{id} \}$$

$$\text{First } \{T'\} = \{ *, \epsilon \}$$

$$\text{First } \{F\} = \{ (, \text{id} \}$$

$$\text{Follow } \{E\} = \{ \$,) \}$$

$$\text{Follow } \{E'\} = \{ \$,) \}$$

$$\text{Follow } \{T\} = \{ +, \$,) \}$$

$$\text{Follow } \{T'\} = \{ +, \$,) \}$$

$$\text{Follow } \{F\} = \{ +, *, \$,) \}$$

d. Design predictive parser for the following grammar.

$$E \rightarrow TE'$$

$$E' \rightarrow +TE' \mid e$$

$$T \rightarrow (E) \mid id$$

λT	id	+	c)	\$
E	$E \rightarrow TE'$		$E \rightarrow TE'$		
E'		$E' \rightarrow +TE'$		$E' \rightarrow e$	$E' \rightarrow E$
T	$T \rightarrow id$		$T \rightarrow (E)$		

Stack	Input	Parse Tree
\$ E	id + id \$	
\$ E' T	id + id \$	
\$ E' id	id + id \$	
\$ E' \$	+ id \$	
\$ E' T +	+ id \$	
\$ E' T	id \$	
\$ E' id	id \$	
\$ E	\$	
\$	\$	

Predictive parser (LL(1) Parser)

- Let ' x ' be a stack top symbol & let ' a ' be the input symbol.
- if $x = a = \$$ then accept & break
- else if $x \neq a \neq \$$ then pop ' x ' & remove ' a '

- else if $M(x, y) \geq r \rightarrow y_1, y_2, \dots, y_k$ then pop 'r'
push y_k, y_{k-1}, \dots, y_1
- else error()

D. Test whether the following grammar is LLL(1) or not

$$\begin{array}{l} S \rightarrow AaAbB \mid Babb \\ A \rightarrow \epsilon \\ B \rightarrow \epsilon \end{array}$$

$$\begin{array}{l} \text{Follow}\{A\} = \{\epsilon\} \\ \text{Follow}\{B\} = \{\epsilon\} \\ \text{Follow}\{S\} = \{\epsilon, \$\} \end{array}$$

$$\text{First}\{A\} = \{\epsilon\}$$

$$\text{First}\{B\} = \{\epsilon\}$$

$$\text{First}\{S\} = \{a\}$$

$$\text{First}\{a\} = \{a\}$$

$$\text{First}\{b\} = \{b\}$$

$\checkmark T$	\$	a	b	
A				
B				
S	:	a..		

* Shift Reduce Parser.

Ans

Actions in shift reduce parser:

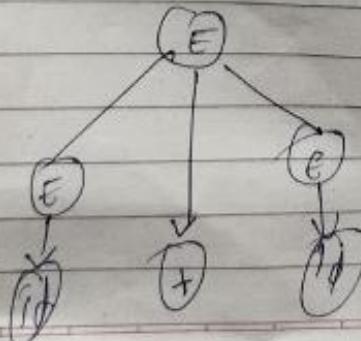
- Shift ~~take~~ shift the terminal on the stack
- Reduce Reduce LHS by RHS.
- Accept Completely scanned input and stack contains empty tokens
- Reject

Algorithm.

Design the following grammar using shift reduce parser method.

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

stack	input	Action
\$	id + id \$	shift id
\$ id	+ id \$	Reduce using $E \rightarrow id$
\$ E	+ id \$	shift +
\$ E +	id \$	shift id
\$ E + id	\$	Reduce using $E \rightarrow id$
\$ E + E	\$	Reduce using $E \rightarrow E + E$
\$ E	\$	Accepted



Handle

The string on the R.H.S. of the production can be changed with some variable on L.H.S.

Handle Pruning

The process of selecting or reducing string with variable.

* Operator precedence parser.

Id \Rightarrow highest power right to left scan.
\\$ \Rightarrow lower

Algo.

Start variable scan exit
if a = topmost stack terminal
 b = input string
 a < b a = b a > b.
 ;

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

operator	precedence	Associativity
id	Highest	-
^		Right
*		Left
+		Left
\$	lowest	

$a < b$ CFG
 $a > b$ * Null.
 $a = b$ #
 .

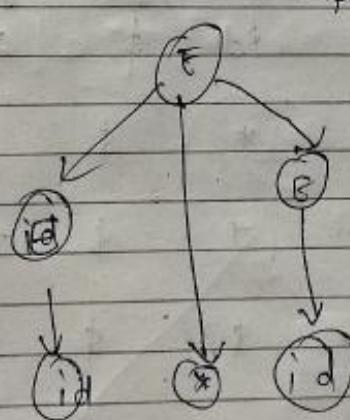
Table Matrix

$a \leq b$	$a \geq b$	$a = b$	$a \neq b$	$a > b$	$a < b$	shift	Reduce	→ left associ.
shift	shift	shift	shift	shift	shift	shift	shift	3x4x5: >
else								

wellformness of parenthesis ()

Step 1	id	+	*	\$
id	-	>	>	>
+	<	≥	<	>
*	<	>	>	>
\$	<	<	<	Accepted

stack	input	a	b	input terminal	action
\$	id * id \$	\$	id	*	shift id
\$ id	* id \$	id	*	*	> Reduce $E \rightarrow id$
\$ E	* id \$	*	*	*	< shift id
\$ E * id	\$	id	\$	*	> Reduce $E \rightarrow id$
\$ E * E	\$	*	\$	*	> Reduce $E \rightarrow E * E$
\$ E	\$	\$	\$		Accept



$\text{id} + \text{id} * \text{id}$

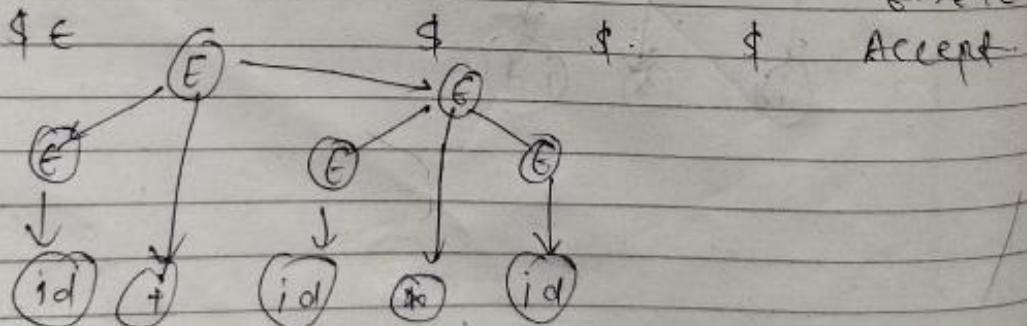
Page No.

Date

Stack	Input	a	b	Action
\$	$\text{id} + \text{id} * \text{id}$	\$	id	\rightarrow shift id
\$id	$+ \text{id} * \text{id}$	id	*	\rightarrow shift *
\$id+	$\text{id} * \text{id}$	+		\rightarrow id \leftarrow Reduce $E \rightarrow id$
\$E+	$\text{id} * \text{id}$	+		

Stack	Input	a	b	Action
\$	$\text{id} + \text{id} * \text{id}$	\$	id	\times shift id
\$id	$+ \text{id} * \text{id}$	id	+	\rightarrow shift Reduce $\epsilon \rightarrow id$
\$E	$+ \text{id} * \text{id}$	\$	+	\leftarrow shift +
\$E+	$+ \text{id} * \text{id}$	\$	id	\leftarrow shift id
\$E+id	$+ \text{id} * \text{id}$	id	*	\rightarrow Reduce $E \rightarrow id$
\$E+E	$+ \text{id} * \text{id}$	\$	+	\leftarrow shift *
\$E+E*	$+ \text{id} * \text{id}$	\$	id	$\rightarrow E \rightarrow E^*$
\$E+E*	$+ \text{id} * \text{id}$	*	id	\times shift id
\$E+E*id	$+ \text{id} * \text{id}$	\$	id	\rightarrow Reduce $E \rightarrow id$
\$E+E*E	$+ \text{id} * \text{id}$	\$	*	\rightarrow Reduce $E \rightarrow id$
\$E+E*E	$+ \text{id} * \text{id}$	\$	\$	\rightarrow Reduce $E \rightarrow E^*$

Stack	Input	a	b	Action
\$E+E	$+ \text{id} * \text{id}$	\$	+	\rightarrow Reduce $E \rightarrow E^*$



E1 \Rightarrow Mixing operators. operator grammar
contents free

Date _____
Page No. _____
operator grammar
 \Rightarrow null value is
two tokens
should be
together.

$$E \rightarrow E+E \mid E * E \mid E \uparrow E \mid id.$$

	id	*	*	\uparrow	\$
id	-	>	>	>	>
+	<	>	<	>	
*	<	>	>	<	>
\uparrow	<	>	>	<	>
\$	<	<	<	<	Accept

$$\begin{array}{l} S \xrightarrow{} SAS \mid a \\ A \xrightarrow{} bSb \mid b \end{array}$$

Convert the following grammar into operator grammar.

1) $S \rightarrow SAS \mid a$
 $A \rightarrow bSb \mid b$

$$\begin{array}{l} S \rightarrow SbSbS \mid SbS \mid a \\ A \rightarrow bSb \mid b \end{array}$$

should not be written
in final answer as A
variable is not present in
start variable.

2) $S \rightarrow SFS$
 $F \rightarrow + \mid - \mid * \mid / \mid 1$

$$S \rightarrow S + S \mid S - S \mid S * S \mid S / S \mid 1$$

Q. With the help of following grammar explain the rules of operator precedence parser.

$$E \rightarrow E + T \mid T$$

$$T \rightarrow T * V \mid V$$

$$V \rightarrow a \mid b \mid c \mid d$$

String to be parsed "a + b * c * d"

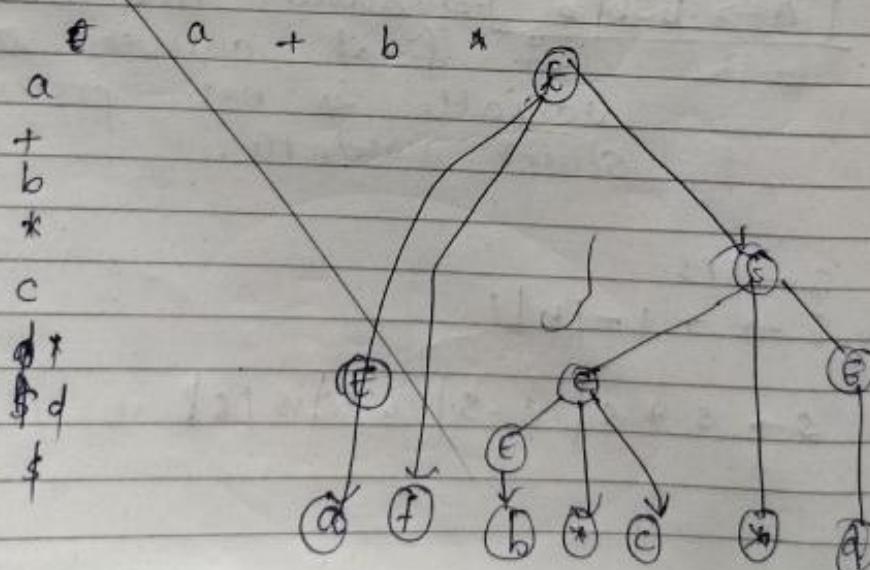
2-terminal +, *

Non-terminal a, b, c, d.

~~$E \rightarrow E$~~

~~$T \rightarrow T * a \mid a \mid T * b \mid b \mid T * c \mid c \mid T * d \mid d$~~

Operator precedence table.



$E \rightarrow E+E \mid E * E \mid a/b/c/d$

Page No.	
Date	

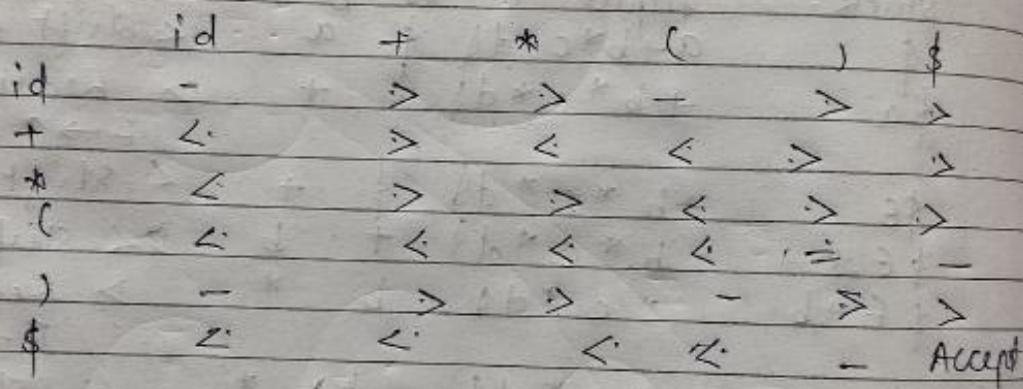
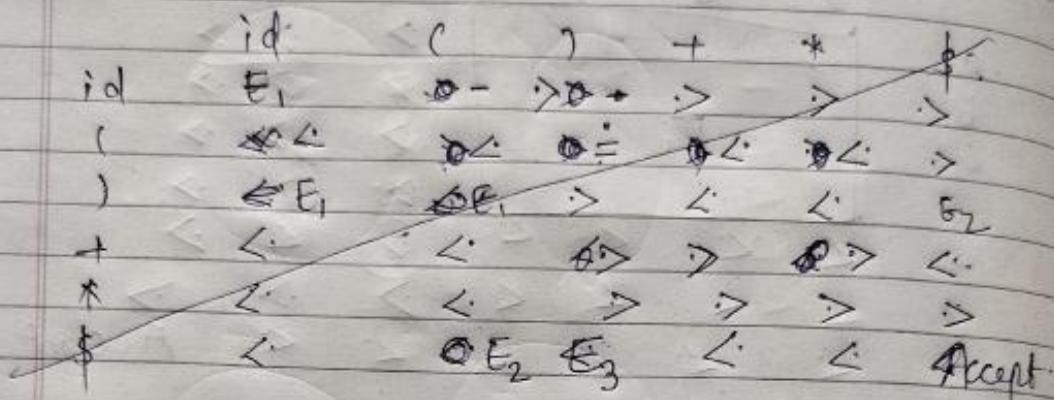
Operator precedence table.

	a	b	c	d	+	*	/
a					>	>	>
b					>	>	>
c					>	>	>
d					>	>	>
+	<	<	<	<	>	<	>
*	<	<	<	<	>	>	>
/	<	<	<	<	<	<	Accept

Stack.

Input	@	a	b	Action
\$				< Shift a.
\$ a		a		> Reduce. $E \rightarrow a$.
\$ E				< Shift +
\$ E +				< Shift b
\$ E + b				> Reduce $E \rightarrow b$.
\$ E, + E,				< Shift *
\$ E + E *				< Shift c
\$ E + E * c				> Reduce $E \rightarrow c$.
\$ E + E * E				> Reduce $E \rightarrow E * E$
\$				< Shift *
\$ E + E				< Shift *
\$ E + E *				> Reduce $E \rightarrow E * E$
\$ E + E * d				> Reduce $E \rightarrow E * E$
\$ E + E * E				> Reduce $E \rightarrow E * E$
\$ E + E				> Reduce $E \rightarrow E * E$
\$ c				> Reduce $E \rightarrow E * E$

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



SLR

Parsing technique
LR(0)

↓
 left RMD → number of input
 for right in look ahead,
 reverse.

Apply SLR parsing technique and
 construct SLR parser table
 for the following grammar.

$$E \rightarrow E + T \mid T$$

$$T \rightarrow T * F \mid F$$

$$F \rightarrow id$$

S1) Augment the grammar.

$$\underline{E' \rightarrow E}$$

$$\text{Follow}(E) = \text{follow}(E')$$

- ① $E \rightarrow \underline{E} + T$
- ② $E \rightarrow I$
- ③ $T \rightarrow I^* F$
- ④ $T \rightarrow F$
- ⑤ $F \rightarrow id$

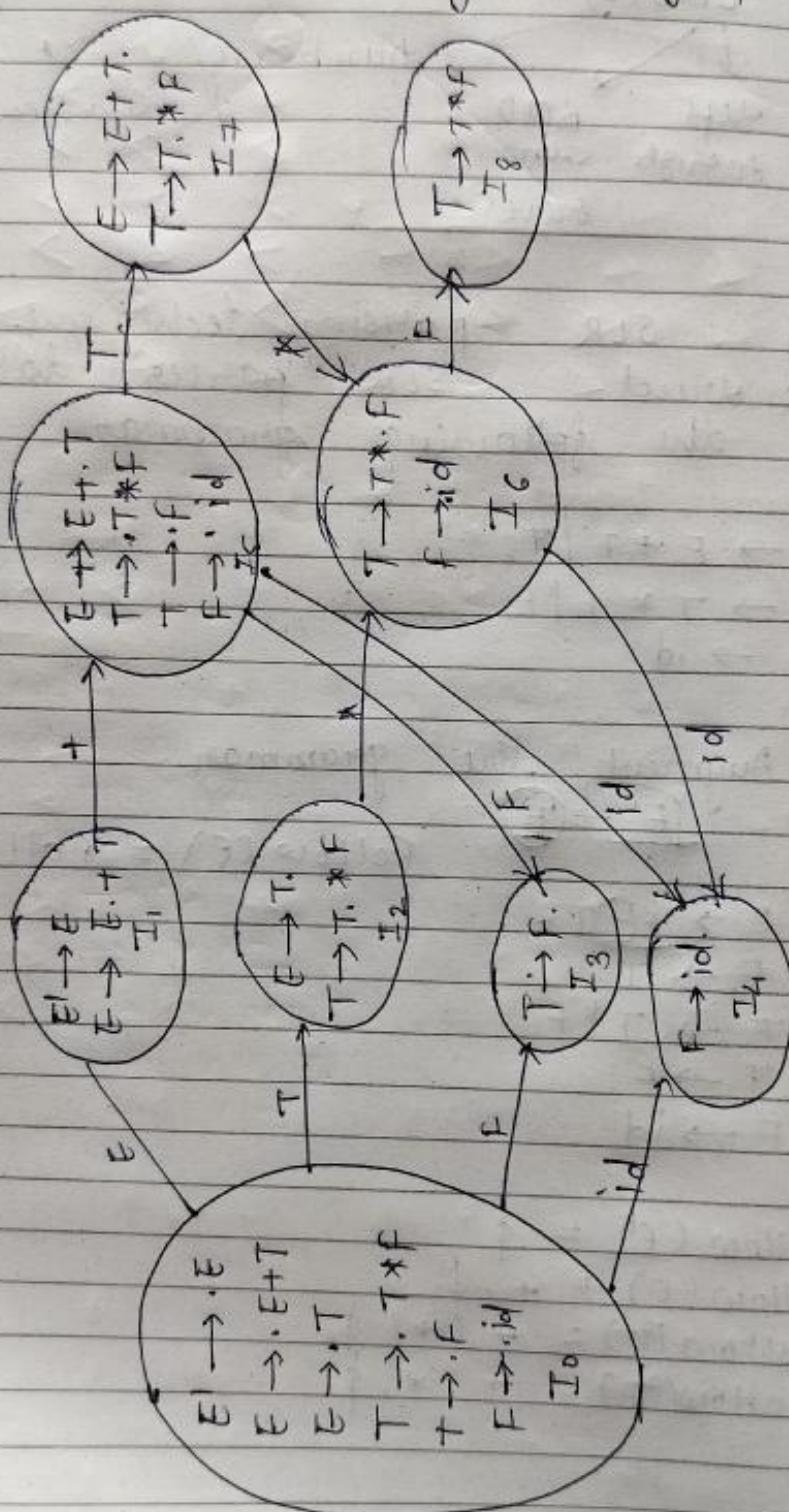
$$\text{Follow}(E') = \$$$

$$\text{Follow}(E) = +, \$$$

$$\text{Follow}(T) = +, *, \$$$

$$\text{Follow}(F) = +, *, \$$$

32 : Flow graph / go to graph.



53

Start

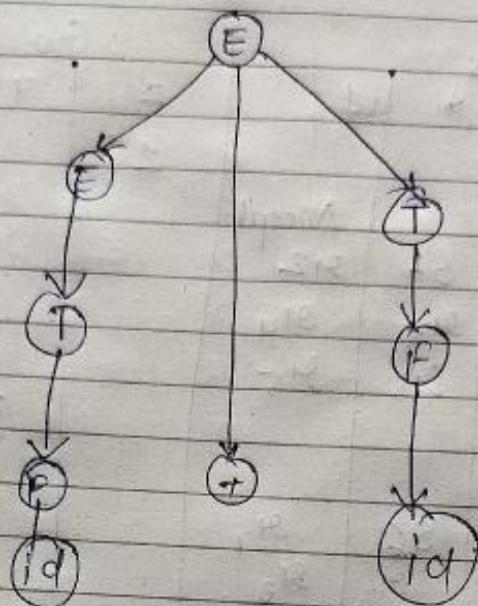
 $\alpha_2 \rightarrow$ $\alpha_2 \rightarrow$

53: LR(0) Parser Table.

State	Actions				Goto to		
	id	+	*	id \$	E	T	F
0	S4				1		
1		S5					
2				Accept			
3				S12	2		
4				S14			
5				S15			
6	S4						
7					7		
8						3	
Pd + id.							8

\$ Stack	Input	Action
\$0	id + id \$	shift id & goto 4.
\$0 id 4	\$ + id \$	reduce using F → id
\$0 F 3	+ id \$	reduce using T → F
\$0 T 2	+ id \$	reduce using E → T
\$0 E 1	+ id \$	shift + goto 5
\$0 E 1 + 5	id \$	shift id go to 4.
\$0 E 1 + 5 id 4	\$	reduce using F → Id.
Goto:		Action
8 0 E 1 + SFB	\$	reduce using T → P
\$ 0 E 1 + S T 7	\$	reduce using E → E + T
\$ 0 E 1	\$	Accept.

Parse Tree.



Compute LR(0) item for the following production.

$$S \rightarrow XYZ$$

$$S^1 \rightarrow \cdot S$$

$$S \rightarrow \cdot XY2$$

$$S \rightarrow X \cdot YZ$$

$$S \rightarrow X Y \cdot Z$$

$$S \rightarrow S Y Z$$

Apply SLR parsing technique and construct SLR parser table for the following grammar.

$$\begin{aligned} S &\rightarrow (S) S \\ S &\rightarrow \epsilon \end{aligned}$$