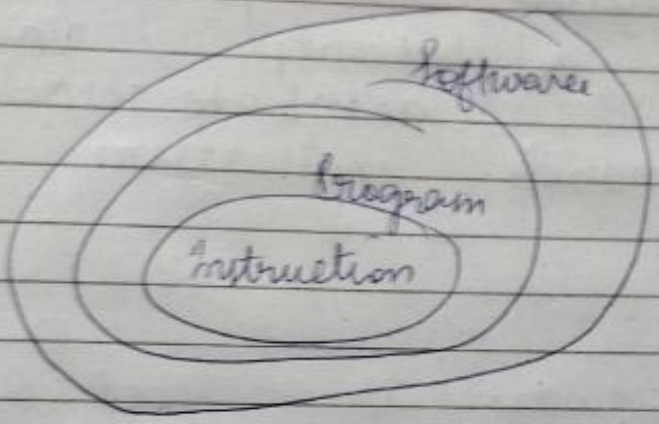
2-5 - Numericals
1 & 6 - Theory

Page No.
Date

SPCC (System programming & compiler construction)

Model - 1

Overview of system software



Types of software

① System s/w — linker, Assembler, OS, macOS, compiler, debugger, editor.
② Application s/w.

System software

System software is a software which is used to run the system from user point of view. For eg. OS (operating system), Assembler, compiler, macro and macroprocessor, 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

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

BI

Engineering & Science

Application Software	System software
① HLL	① LLL
② It is used for specific purpose.	② It is used for general purpose.
③ It executes	③

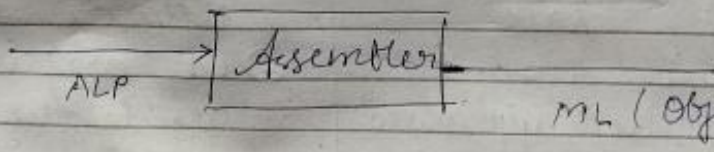
Application software	System software.
① Application software is build for specific task	① System software used for application software to run.
② HLL are used to write the application software.	② LL are used to write the system software
③ Without application software, system always run.	③ Without system software, system can't run.
④ Application software run as per user request.	④ System software run when the system is turned ON and stop when the system is turned off.

⑤ Specific purpose software	⑤ General purpose software
⑥ Example: Adobe reader, Photoshop, VLC media player	⑥ OS, Assembler, compiler, Macro processor, Loader, linker etc.

Types of System software's

① Assembler.

Assemble is a language translator that takes as input the Assembly language program and generate 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 of ALP.

IVM360/370 MP instruction

Page No.
Date

Definition of Macro:

```
MACRO
MACRONAME
=
=
=          } group of statements to be executed.
MEND
```

DIA:

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

Compiler

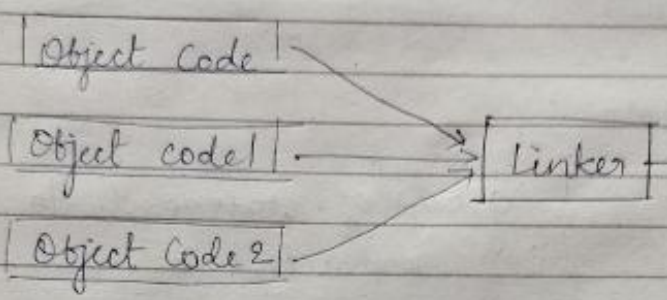
Compiler is a language translator that takes as input the source program i.e. is written in high level language and produces as output an assembly language or machine language.

Translator

Translator is a system program that converts program in one language to program in another language.

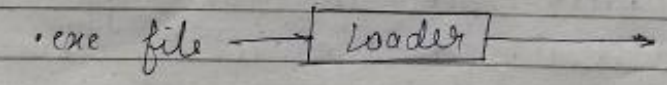
Eg. Assembler compiler

* 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.

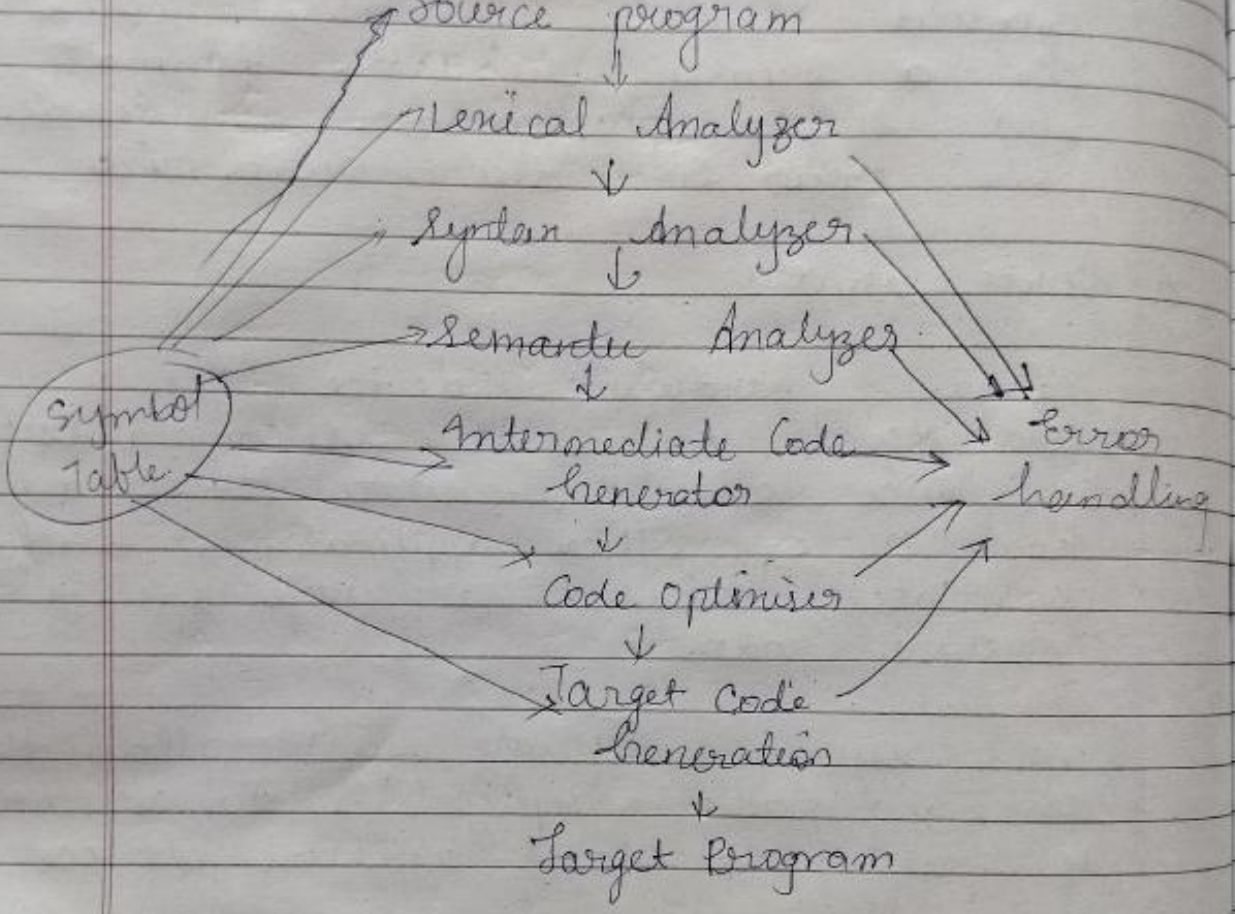
Debugger compiler is used to present compiler generated errors in a human readable format and helps programmer in finding the error in the source code.

* Device drivers.

Every hardware device can understand only its LLL commands; a device driver translate 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.

Phases of Compiler



Module 2: lexical analysis

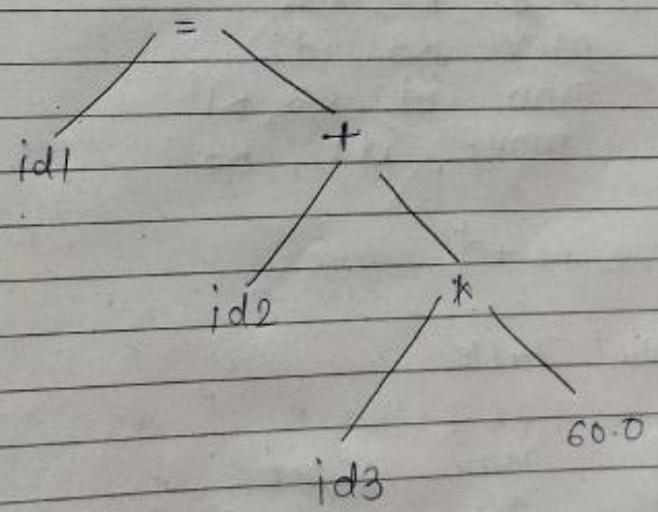
position = initial + rate * 60.

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 3CAF (3 code Address format)

$t_1 = id3 * 60.0$
$t_2 = id2 + t_1$
id1 ~~t_3~~ $= id1 + t_2$

Code optimizer

$t_1 = id3 * 60.0$
$id1 = id2 + t_1$

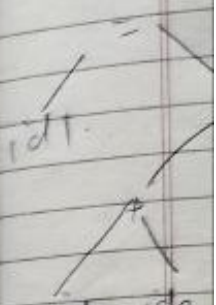
Target code generation

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

a = b + c + d.

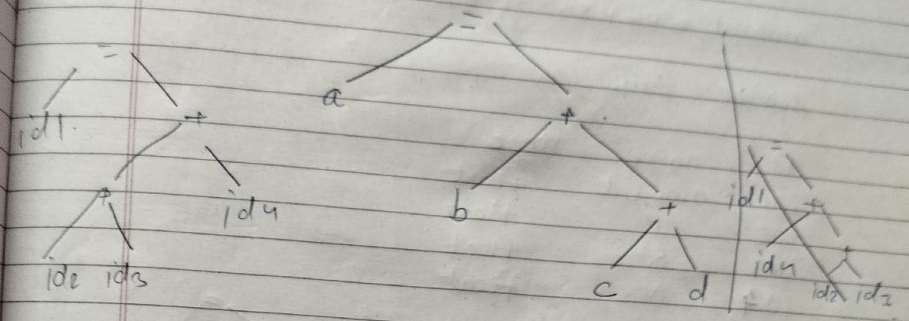
Symbol Table.

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



id1 = id2 + id3 * id4

Parse tree



ICG (3CAF)

$t_1 = id3$

$t_2 = t_1 + id4$

$id1 = t_2$

Optimized.

$t_1 = c + d$ $\quad t_1 = id2 + id3$

$a = b + t_1$ $\quad id1 = id4 + t_1$

Target code generation.

MOV R1, id2

MOV R2, id3

ADD R1, R2

MOV R3, id4

ADD R1, R3

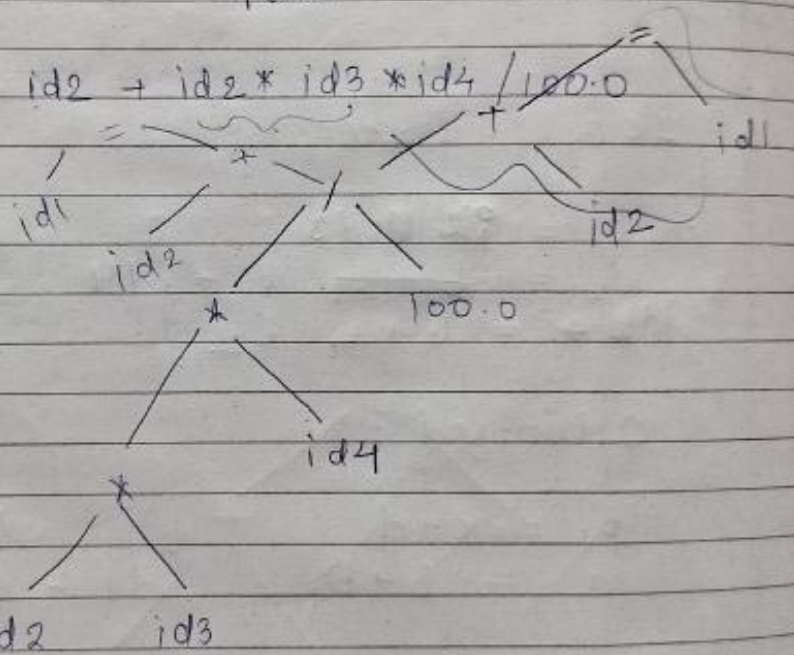
MOV id1, R1

amount = P + P*R*N / 100

Symbol table

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

id1 = id2 + id2 * id3 * id4 / 100.0



ICG

t_1 = id2 * id3

t_2 = t_1 * id4

t_3 = t_2 / 100.0

t_4 = id2 + t_3

id1 = t_4

Code optimizer

$t_1 = id2 * id3$

$t_2 = t_1 * id4$

$t_3 = t_2 / 100.0$

$id1\ t_4 = id2 + t_3$

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 ⇒ To generate tokes

Secondary fn

(1) Remove blank spaces

(2) Remove comments.

(3) Communicating with the symbol table.

(4) Correlate error LA invalid token error. Bypasses correct token.

SA

identifier

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

The secondary fn of lexical analyzer are
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. ~~Prog~~

* 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.

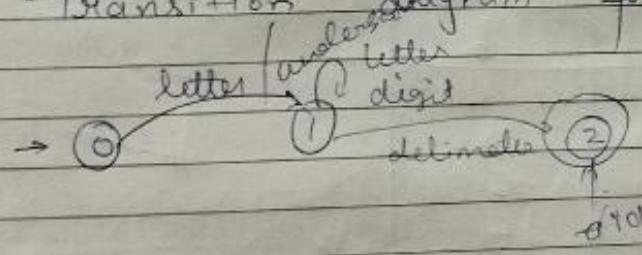
Finite automata is a recognizer for a language L that takes as an input string x and answers "Yes" if the x is a sentence of L and 'no' otherwise

Regular expression for an identifier

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

(a-z)(A-Z) (0-9)

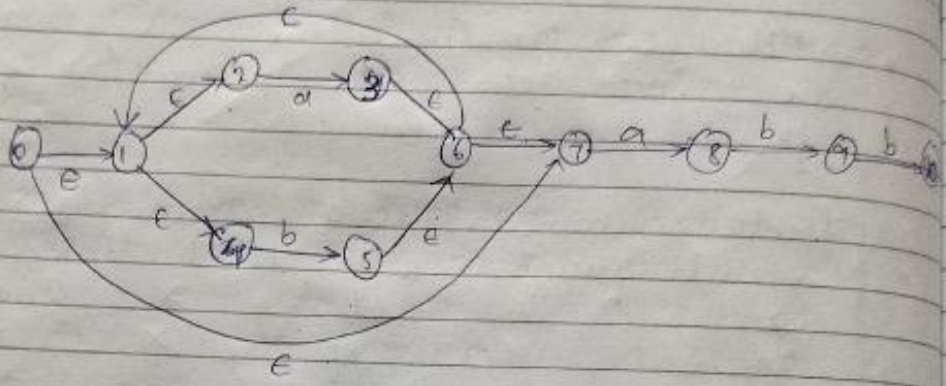
Transition diagram for an identifier



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

Design DFA for the regular expression

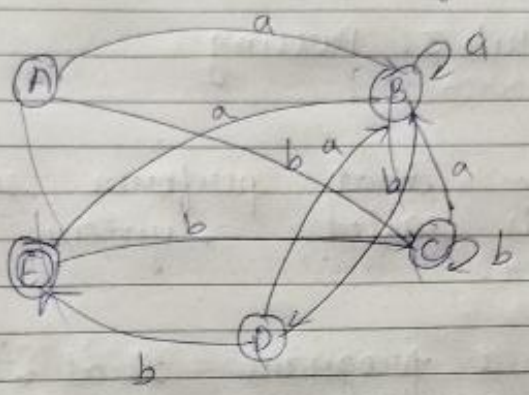
$r = (a+b)^* abb$



	x	y = ε-closure(x)	δ(y, a)	δ(y, b)
A	{0}	{0, 1, 2, 4, 7}	{3, 8}	{5}
B	{3, 8}	{3, 8, 6, 1, 2, 4, 7}	{3, 8}	{5, 9}
C	{5}	{5, 6, 1, 2, 4, 7}	{3, 8}	{5}
D	{5, 9}	{6, 1, 2, 4, 7, 5, 9, 10}	{3, 8}	{5, 10}
E*	{5, 10}	{5, 6, 1, 2, 4, 7, 10}	{3, 8}	{5}

	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
flex demo.l
gcc lex.yy.c -ll
./a.out
yytext



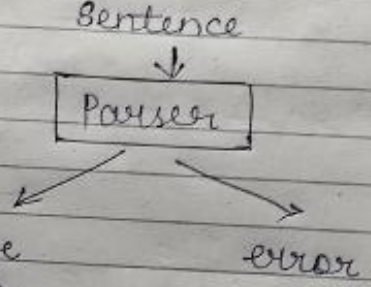
	a	b
AC	B	AC
B	B	D
D	B	E
E*	B	AC

Module 3: Parsing

Parser

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

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



Parsing techniques

top down parser	Bottom up parser
1) The parser builds the parse tree from top to bottom ie. from root to the leaves.	1) The parser builds parse tree from bottom to top ie. from leaves to top.
2) Recursive descent parser (RDP) Predictive parser (LL)	2) Predictive parser (LL1) Shift red Operator precedence

Simple LR
Canonical parse
look ahead parses.

* Recursive descent parser (RDP).

Design recursive descent parser for the following grammar.

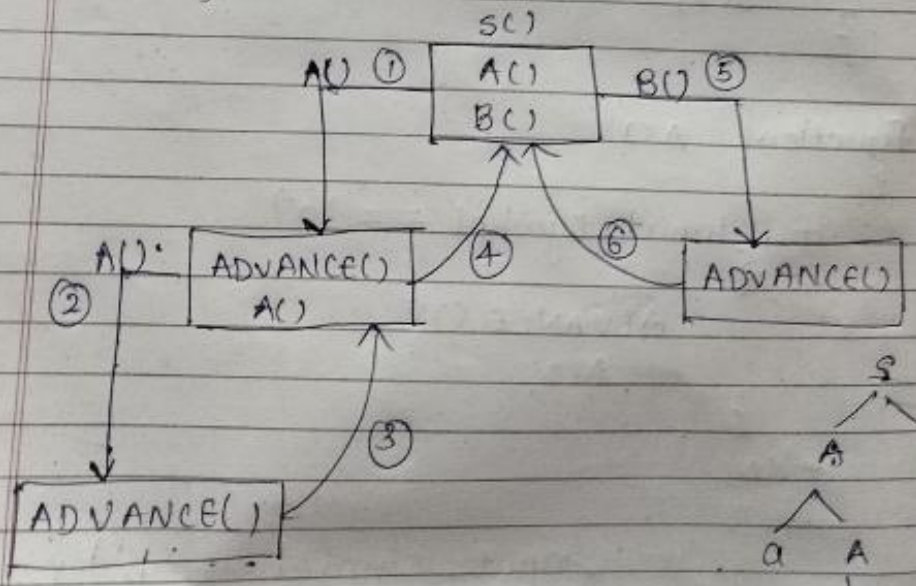
S → AB
A → aA | b
B → bB | a

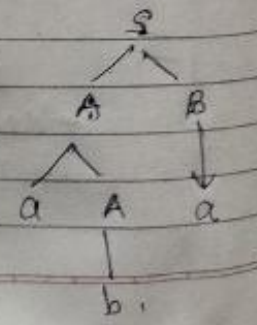
```
function S()
 {
     A();
     B();
 }

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
        B();              input symbol.
    }
    else {
        if (input_symbol == 'a') {
            ADVANCE();
        }
        else { ERROR(); }
    }
}
```





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

```
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

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

Elimination procedure

Eliminate the left recursive grammar.

$A \rightarrow \beta A'$

$A' \rightarrow \alpha A' \mid \epsilon$

or parser

form

sive

$E \rightarrow E + E \mid id$

$E \rightarrow id\ E'$

$E' \rightarrow +\ E\ E' \mid \epsilon$

```
function E() {
    if (input_symbol == 'id') {
            ADVANCE();
            E'();
    }
    else {
            ERROR();
    }
}

function E'() {
    if(inputsymbol == '+') {
            ADVANCE();    E(); E();
    if (input_symbol == 'E') { E();
            ADVANCE();
    }
    else { ERROR();
            if (input_symbol == 'ε') {
            ADVANCE   E'();
              ADVANCE();
            }
          else { ERROR(); }
        else { ERROR(); }
    }
}
```

~~else if (input symbol == 'ε')~~ {

}

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.

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

$A \rightarrow \beta A'$

$A' \rightarrow \alpha A' \mid \epsilon$

General format.

$A \rightarrow A\alpha_1 \mid A\alpha_2 \mid \ldots \mid A\alpha_n$

$A \rightarrow \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n$

$A \rightarrow \beta_1 A' \mid \beta_2 A' \mid \ldots \mid \beta_n A'$

$A' \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \ldots \mid \alpha_n A' \mid \epsilon$

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

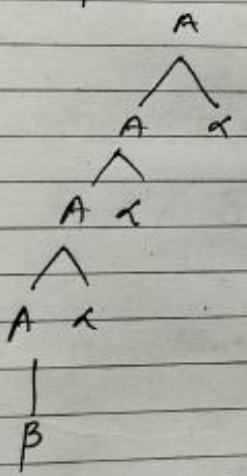
We have to remove or eliminate left recursion but preserve the language generated by grammar.

Grammar

left recursion

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

$A \rightarrow \beta$ or

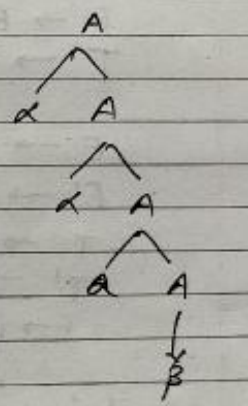


Language : $\beta\alpha^*$

Right recursion

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

$A \rightarrow \beta$ or



Language : $\alpha^*\beta$

There are two types of left recursion

(1) Direct left recursion

Eg. $B \rightarrow Ba$

(2) Indirect left recursion

$S \rightarrow Aa$

$A \rightarrow Sa$

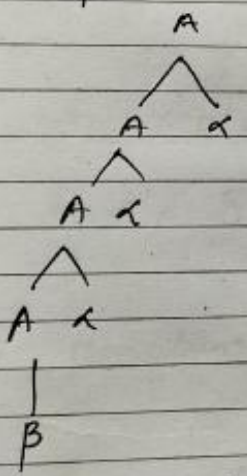
We have to remove or eliminate left recursion but preserve the language generated by grammar.

Grammar

left recursion

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

$A \rightarrow \beta$ or

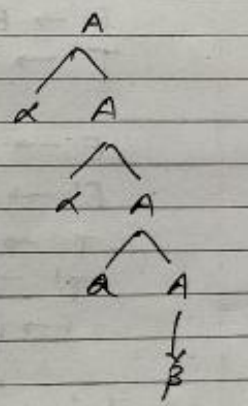


Language : $\beta\alpha^*$

Right recursion

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

$A \rightarrow \beta$ or



Language : $\alpha^*\beta$

There are two types of left recursion

(1) Direct left recursion

Eg. $B \rightarrow Ba$

(2) Indirect left recursion

$S \rightarrow Aa$

$A \rightarrow Sa$

Eliminate the left recursion from the following grammar.

$E \rightarrow E + T / T$
$T \rightarrow T * E | F$
$F \rightarrow id$

$E \rightarrow TE'$
$E' \rightarrow +TE$

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

$E \rightarrow TE'$
$E' \rightarrow +TE' | \epsilon$
$T \rightarrow$ ~~id FT'~~ FT'
$T' \rightarrow *ET' | \epsilon$
$F \rightarrow id.$

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

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

```
    else {
            error();
         }
}
  function T() {      ADVANCE();
            function   F();  T();
         }

  function T'(){
        if(input-symbol==='*') {
                    ADVANCE();
                    E();
                    T'();
         }
        else{
                NO ACTION();
         }
}

function
```

$A \rightarrow Ba \mid Aa \mid c$
$B \rightarrow Bb \mid Ab \mid d$

$A \rightarrow Aba \mid Aa \mid c$
$B \rightarrow Bb \mid Bab \mid d$

$A \rightarrow cA'$
$A' \rightarrow baA' \mid aA' \mid \epsilon$

$B \rightarrow dB'$
$B' \rightarrow bB' \mid abB' \mid d$

$S \rightarrow Aa \mid b$
$A \rightarrow Ac \mid Sd \mid \epsilon$

$S \rightarrow Sda \mid b$
$A \rightarrow Ac \mid Aad \mid \epsilon$

$S \rightarrow bS'$
$S' \rightarrow daS' \mid \epsilon$

$A \rightarrow cA'$
$A' \rightarrow cA' \mid adA' \mid \epsilon$

Page No.
Date

Left factoring

There are two different types of grammar :

① Deterministic grammar

It ~~define~~ ~~who~~ determines where we want to go exactly. In deterministic grammar there is exactly one unique prefix.

② Non-deterministic grammar not There is exactly one unique prefix.

$A \rightarrow \alpha\beta \mid \alpha\gamma$

The main cause of Non-deterministic grammar is repeatition 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

$A \rightarrow \alpha\beta \mid \alpha\gamma$

common prefix

Left factoring

There are two different types of grammar:

① Deterministic grammar

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

② Non-deterministic grammar

There is not exactly one unique prefix.

$A \rightarrow \alpha\beta / \alpha\gamma$

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 the left factoring comes into the picture. In left factoring we convert non-deterministic grammar to deterministic grammar.

Any production of the form

$A \rightarrow \alpha\beta / \alpha\gamma$

common prefix

Now to eliminate left factoring

$A \to \alpha A'$

$A' \to \beta \mid \gamma$

General format.

$A \to \alpha_1 \beta_1 \mid \alpha_1 \beta_2 \mid \ldots \mid \gamma_1 \mid \gamma_2$

$\Downarrow$

$A \to \alpha_1 A' \mid \gamma_1 \mid \gamma_2$

$A' \to \beta_1 \mid \beta_2$

Eliminate left factoring from the following grammar:

(i) $S \to asb \mid ab$

(ii) $A \to aB \mid abc$

(iii) $S \to iEtS \mid iEtSeS \mid a$

$E \to b$

(iv) $S \to$ ~~assb~~ $assbS \mid asasb \mid abb \mid b$

(i) $S \to asb \mid ab$

$S \to aS'$

$S' \to Sb \mid b$

(iii) $A \rightarrow aB \mid abc$

$A \rightarrow aA'$

$A' \rightarrow B \mid bc$

(iii) $S \rightarrow iEtS \mid iEtSeS \mid a$

$E \rightarrow b$

$S \rightarrow iEtS' \mid a$

$S' \rightarrow eS \mid e$

$E \rightarrow b$

$S \rightarrow iS' \mid a$

$S' \rightarrow EtS \mid EtSeS$

$E \rightarrow b$

(iv) $S \rightarrow aSSbS \mid aSaSb \mid abb \mid b$

$S \rightarrow aSS' \mid abb \mid b$

$S' \rightarrow SbS \mid aSb$

(v) 3AC (3 address code form).

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

$t_1 = a*b$

$t_2 = b-c$

$t_3 = t_1 + t_2$

$a = t_3$.

FIRST(α)

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

Rule 1:

$R_1 \rightarrow$

If $X \rightarrow a\alpha$ then first{X} = {a}

Rule 2:

If $X \rightarrow \epsilon$ then first{X} = {ϵ}

Rule 3:

If a is a terminal then first{a} = {a}

Rule 4:

If $X \rightarrow y_1 y_2 \cdots y_k$ then first{X} = first{y_1}

If y_1 contains ϵ then,

first{X} = first{y_1} - {ϵ} + first{y_2}

Note: If y_i contains ϵ then ~~FIRST{X}~~ FIRST{X} = ϵ where $j = 1, 2, 3$

FOLLOW(A).

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

Rule 1:

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

Rule 2:

If $X \rightarrow \alpha Y$

then FOLLOW{Y} = FOLLOW{X}

Rule 3:

If $X \rightarrow \alpha Y\beta$ then

FOLLOW{Y} = FIRST{β}

If FIRST{β} contains ∈ then

FOLLOW{Y} = FIRST{β} - {∈} + FOLLOW{X}

$X \rightarrow \alpha Y$ (B)

FOLLOW(A).

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

Rule 1:

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

Rule 2:

If $X \rightarrow \alpha Y$

then FOLLOW{Y} = FOLLOW{X}

Rule 3:

If $X \rightarrow \alpha Y\beta$ then

FOLLOW{Y} = FIRST{β}

If FIRST{β} contains ∈ then

FOLLOW{Y} = FIRST{β} - {∈} + FOLLOW{X}

$X \rightarrow \alpha Y$ (B)

Page No.
Date

Q. Find first & follow of the given grammar.

$S \rightarrow ABC$
$A \rightarrow a/\epsilon$
$B \rightarrow b/\epsilon$
$C \rightarrow c/\epsilon$

First {a} = {a}
First {b} = {b}
First {c} = {c}

$S = \{a, b\}$

4)

First {A} = {a, ϵ}
First {B} = {b, ϵ}
First {C} = {c, ϵ}

same question as above (B → b)

Follow(S) = {$}
Follow(A) = {b}
Follow(B) = {c, $}
Follow(C) = {$}

Q. $E \rightarrow TE'$
$E' \rightarrow +TE' / \epsilon$
$T \rightarrow (E) / id$

First {+} = {+}
First {(} = {(}
First {)} = {)}
First {id} = {id}

given

First {E} = { (, id}
First {E'} = {+, ∈}
First {T} = { (, id}

Follow {E} = {$,)} → as E is a start symbol
Follow {E'} = { $,)}
Follow {T} = { +, $,)}

4)

E → TE'
E' → +TE' | ∈
T → FT'
T' → *FT' | ∈
F → (E) | id.

First {+} = {+}
First {*} = {*}
First {(} = {(}
First {)} = {)}
First {id} = {id}

First {E} = { (, id}
First {E'} = { +, ∈}
First {T} = { (, id}
First {T'} = { *, ∈}
First {F} = { (, id}.

Follow {E} = { $,)}
Follow {E'} = { $,)}
Follow {T} = { +, $,)}
Follow {T'} = { +, $,)}
Follow {F} = { *, +, $,)}

above
b

Q. Design predictive parser for the following grammar.

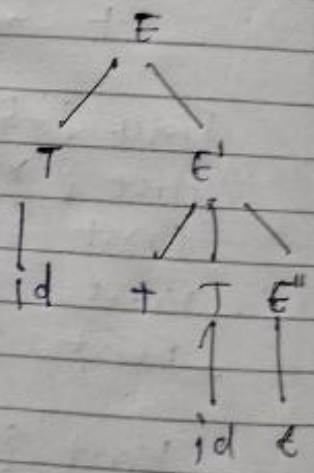
$E \to TE'$

$E' \to +TE' \mid \epsilon$

$T \to (E) \mid id$

NT \ T	id	+	(	)	$
E	$E \to TE'$		$E \to TE'$		
E'		$E' \to +TE'$		$E' \to \epsilon$	$E' \to \epsilon$
T	$T \to id$		$T \to (E)$		

Stack	Input
$ 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 $x = a \neq \$$ then pop 'x' & remove 'a'

- else if $M[x, a] \; x \rightarrow y_1 y_2 \cdots y_k$ then pop 'x' & push $y_k y_{k-1} \cdots y_1$
- else error ()

Q. Test weather the following grammar is LL(1) or not

$S \rightarrow AaAbB \mid BaBb$

$A \rightarrow \epsilon$

$B \rightarrow \epsilon$

Follow{A} = {ε}
Follow{B} = {ε}
Follow{S} = {ε, $}

First{A} = {ε}
First{B} = {ε}
First{S} = {a}
First{a} = {a}
First{b} = {b}

V \ T	$	a	b
A			
B			
S	.	a . .	

Shift Reduce Parser.

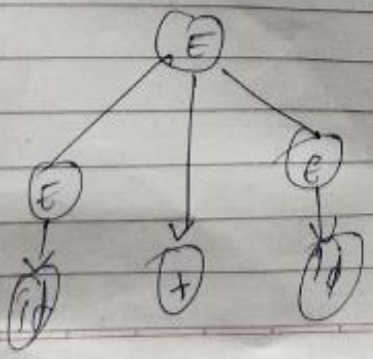
Actions in shift reduce parser:
- Shift — Shift the terminal on the SA
- Reduce — Reduce LHS by RHS.
- Accept — Completely scanned input and stack contains
- Reject

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→id
$E	+id$	Shift +
$E+	id$	Shift id
$E+id	$	Reduce using E→id
$E+E	$	Reduce using E→E+E
$E	$	Accepted



Handle

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

Handle Pruning

The process of selection or reducing string with variable.

* Operator precedence parser.

id ⇒ highest
$ ⇒ lower ↑ ⇒ power ~~Right to left scan~~.

Algo.

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

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

operator	precedence	Associativity
id	Highest	—
^		Right
/, *		left
+, -		left.
$	lowest	

$a \lessdot b$ CFG
$a \gtrdot b$ * Null.
$a \doteq b$ *.

Table / matrix

$a \lessdot b$ or $a \doteq b$ shift
$a \gtrdot b$ Reduce
else.

→ left associative
$3 \times 4 \times 5 \therefore >$

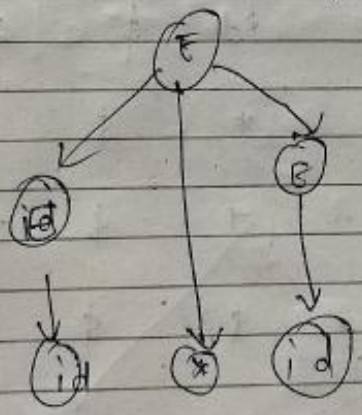
well formness of parenthesis ( )

Step1.	id	+	*	$
id	-	>	>	>
+	<	>	<	>
*	<	>	>	>
$	<	<	<	Accepted

id + id * id topmost stack terminal

Stack	Input	a	b	input terminal Action
$	id * id $	$	id	* shift id
$ id	* id $	id	*	> Reduce $\epsilon \to id$.
$ E	* id $	$	*	< shift *
$ E *	id $	*	id	< shift id
$ E * id	$	id	$	> Reduce. $\epsilon \to id$.
$ E * E	$	*	$	> Reduce. $E \to E * E$
$ E	$	$	$	Accept.

id

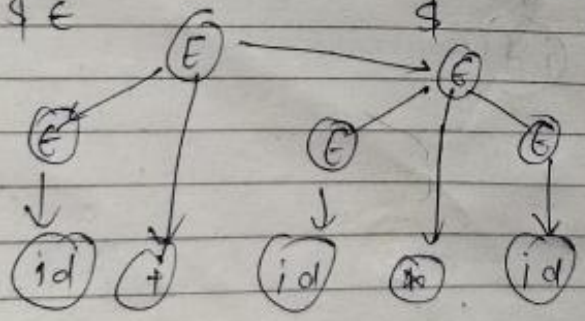


associativ

id + id * id.

Stack	Input	a	b	Actions.
$	id + id * id	$	id	> shift id
$id	+ id * id	id	+	> shift +
$id+	id * id	+	id	< Reduce E → id
$E+	id * id	+		

Stack	Input	a	b	Action.
$	id + id * id $	$	id	< shift id
$id	+ id * id $	id	+	> Reduce E → id.
$E	+ id * id $	$	+	< shift +
$E+	id * id $	+	id	< shift id
$E+id	id * id $	id	*	> Reduce E → id
$E+E	* id $	+	*	< shift *
$E+E*	id $	*	$	E → E + E
$E+E*	id $	*	id	< shift id
$E+E*id	$	id	$	> Reduce E → id
$E+E*E	$	*	$	> Reduce E → E * E
$E+E	$	+	$	> Reduce E → E + E
$E	$	$	$	Accept



∈ ⇒ Missing operator.

operator grammar is context free grammar
⇒ no null value & no two variable should be together.

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

	id	+	*	↑	$
id	−	>	>	>	>
+	<	>	<	<	>
*	<	>	>	<	>
↑	<	>	>	<	>
$	<	<	<	<	Accept

right to left.

~~$S \to SAS \mid a$~~
~~$A \to bSb \mid b$~~

Convert the following grammar into operator grammar.

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

$S \to SbSbS \mid SbS \mid a$
$A \to bSb \mid b$ → should not be written in final answer as A variable is not present in start variable.

2) $S \to SES$
$E \to + \mid - \mid * \mid /$

$S \to S+S \mid S-S \mid S*S \mid S/S \mid$

Q. With the help of following grammar explain the role 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"

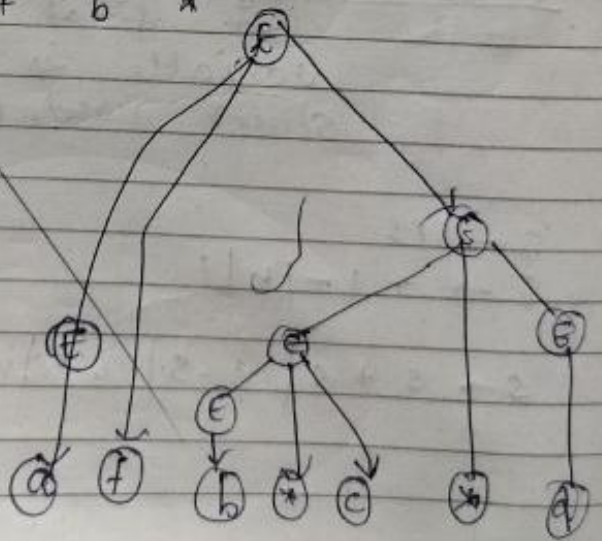
2-terminal +, *
4-identifier a, b, c, d.

$E \rightarrow E$

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

Operator precedence table.

	a	+	b	*
a				
+				
b				
*				
c				
d				
$				



$E \to E+E \mid E*E \mid a \mid b \mid c \mid d$.

Operator precedence table.

	a	b	c	d	+	*	$
a					>	>	>
b					>	>	>
c		E_1			>	>	>
d					>	>	>
+	<	<	<	<	>	<	>
*	<	<	<	<	>	>	>
$	<	<	<	<	<	<	Accept

Stack	Input @ a b		Action
$	a+b*c*d$	$ a	< Shift a.
$a	+b*c*d$	a +	> Reduce $E \to a$.
$E	+b*c*d$	$ +	< Shift +
$E+	b*c*d$	+ b	< Shift b
$E+b	*c*d$	b *	> Reduce $E \to b$.
$E+E	*c*d$	b+ *	< Shift *
$E+E*	c*d$	* c	< Shift c
$E+E*c	*d$	c *	> Reduce $E \to c$
$E+E*E	*d$	* *	> Reduce $E \to E*E$
$E+E	*d$	+ *	< Shift *
$E+E*	d$	* d	< Shift d
$E+E*d	$	d $	> Reduce $E \to d$
$E+E*E	$	* $	> Reduce $E \to E*E$
$E+E	$	+ $	> Reduce $E \to E+E$
$E	$	$ $	Accept

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

	id	(	)	+	*	$
id	E_1	-	>	>	>	>
(	<	<	=	<	<	
)	E_1	E_1	>	<	<	E_2
+	<	<	>	>	<	>
*	<	<	>	>	>	>
$	<	E_2	E_3	<	<	Accept

	id	+	*	(	)	$
id	-	>	>	-	>	>
+	<	>	<	<	>	>
*	<	>	>	<	>	>
(	<	<	<	<	=	-
)	-	>	>	-	>	>
$	<	<		<	-	Accept

SLR Parsing technique

LR0(0)
- left to right
- RHD in reverse.
- number of input look ahead.

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

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

S1) Augment the grammar.

$E' \to E$

Follow(E) = Follow(E')

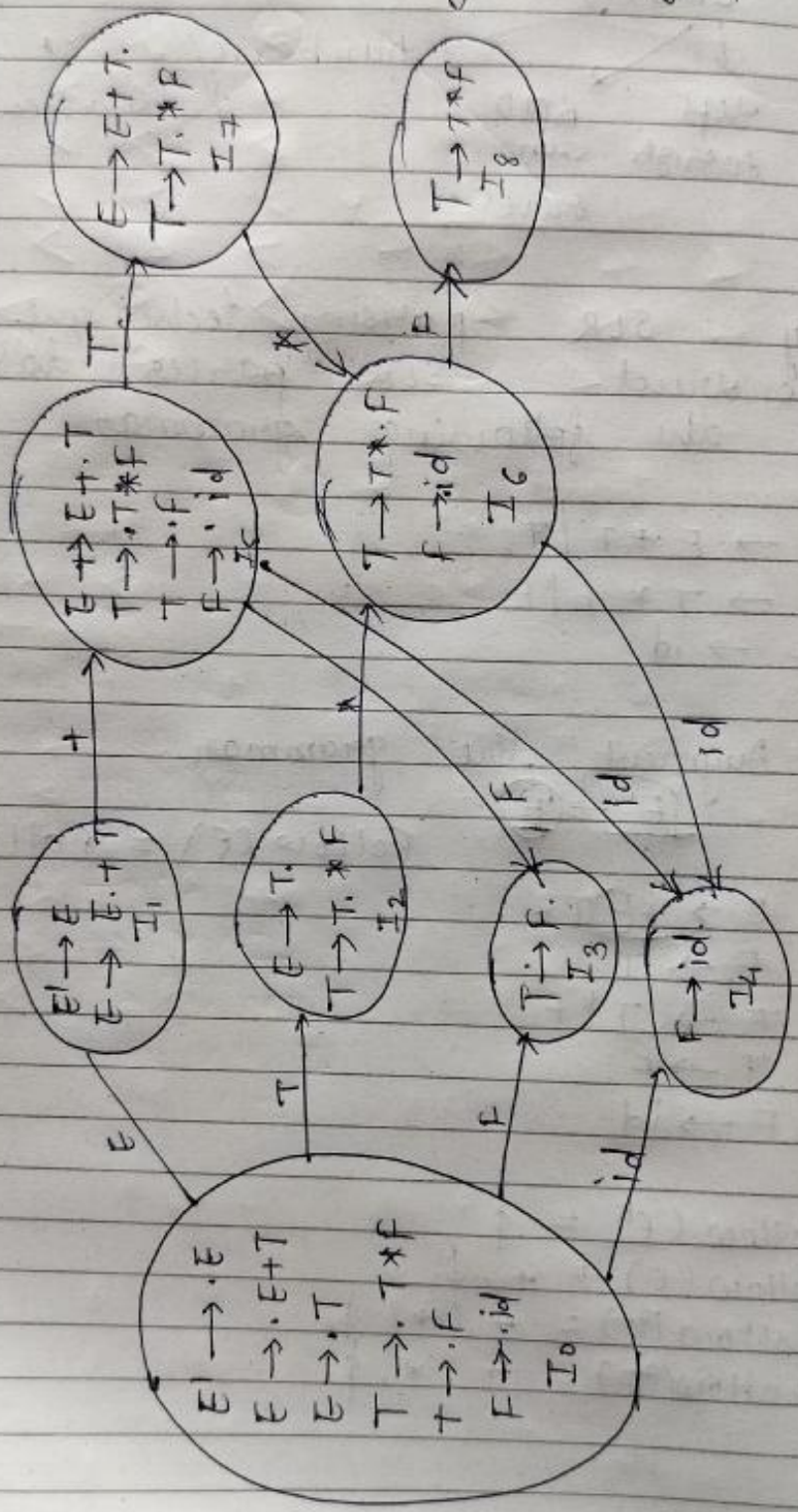
A → B

1. $E \to E + T$
2. $E \to T$
3. $T \to T * F$
4. $T \to F$
5. $F \to id$

Follow(E') = $
Follow(E) = +, $
Follow(T) = +, *, $
Follow(F) = +, *, $

Page No.
Date

Q2: How graph / go to graph.



Q3: LR(0) Parser table.

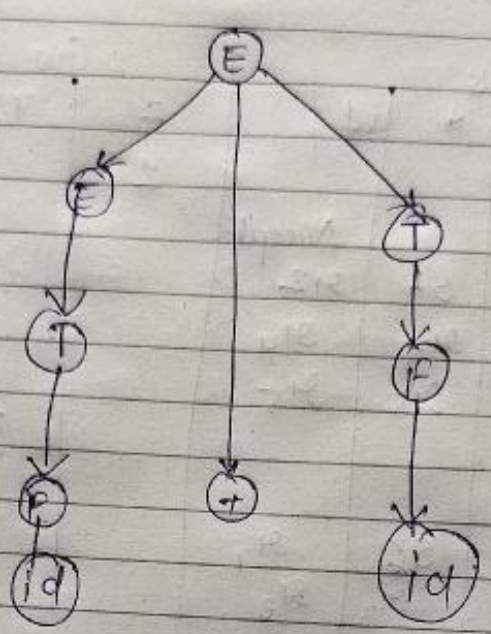
State	Actions				Go to		
	id	+	*	$	E	T	F
0	S4				1	2	3
1		S5		Accept			
→ 2		r2	S6	r2			
→ 3		r4	r4	r4			
→ 4		r5	r5	r5			
5	S4					7	3
6	S4						8
→ 7		r1	S6	r1			
→ 8		r3	r3	r3			

id + id.

$ stack	Input	Action.
$0	id + id $	shift id & goto 4.
$0 id 4	$ + id $	reduce using F→id
$0 F3	+ id $	reduce using T→F
$0 T2	+ id $	reduce using E→T
$0 E1	+ id $	shift + goto 5
$0 E1 + 5	id $	shift id go to 4.
$0 E1 + 5 id 4	$	reduce using F→id.
$0 E1 + 5 F 3	$	reduce using T→F
$0 E1 + 5 T 7	$	reduce using E→E+T
$ 0 E 1	$	Accept.

Goto. Action

Parse Tree.



Compute LR(0) item for the following production.

$S \rightarrow XYZ$.

$S1 \rightarrow \cdot S$

$S \rightarrow \cdot XYZ$

$S \rightarrow X \cdot YZ$

$S \rightarrow XY \cdot Z$

$S \rightarrow SYZ \cdot$

Page No.
Date

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

$S \rightarrow (S)S$

$S \rightarrow \epsilon$