Lex Program
- Declaration
- Transaction Rules - define pattern & action
- Auxiliary function / Main function

* Module 3: Parsing
- The program that performs Syntax analysis is called the syntax analyzer or parser.
- Parser is a program that takes ~~as~~ as input a sentence/string and if the grammar can derive the sentence/string then it generates a parse tree, else it generates an error.
- Parser has two techniques
  - Top Down (root to leaf)
  - Bottom Up (leaf to root)

Parser (parsing techniques)
- Top Down
  - Recursive Descent Parser (RDP)
  - ~~Predictive~~ Predictive Parser (LL(1))
    - Left to right
    - LMD
- Bottom Up
  - ~~Shift~~ Reduce Parser
    - Shift Reduce Parser
    - Operator Precedence
    - Canonical parser
    - Simple LR
    - Look Ahead LR Parser (LALR)

Sentence → Parser → Parse tree (or) ERROR!

Top Down Parser – The parser builds the parse tree from top to bottom i.e. root to leaves.
Bottom Up – leaves to root

1) Recursive Descent Parser
- Top Down
- Recursive – The parser is implemented with functions which may work recursively
- Descent – Top down approach
- It will generate a parse tree only if the following conditions are satisfied –
  (i) i/p is completely scanned
  (ii) the error fn. wasn't called / no infinite loop.

Design Recursive Descent Parser:

$S \rightarrow AB$
$A \rightarrow aA \mid b$
$B \rightarrow bB \mid a$

- the parser will create functions for the variables (capitals)
- conditions are written on terminals.
- ADVANCE() fn. checks the next input symbol.

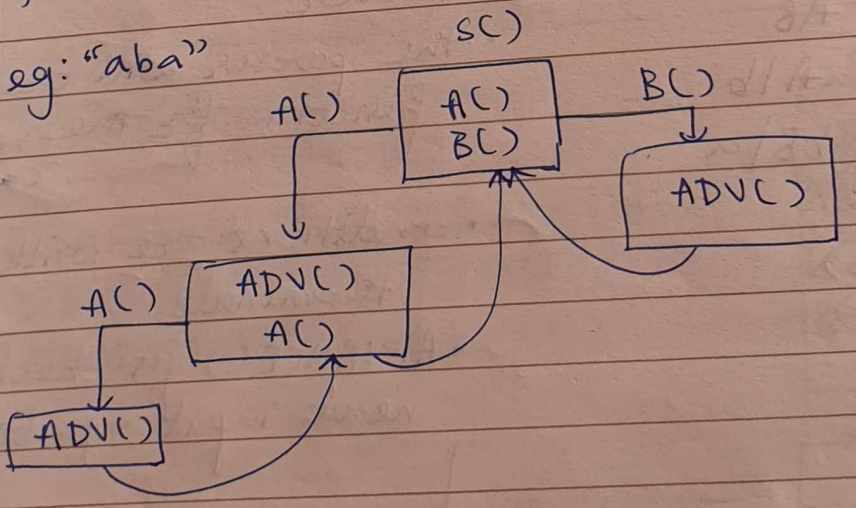
sol.

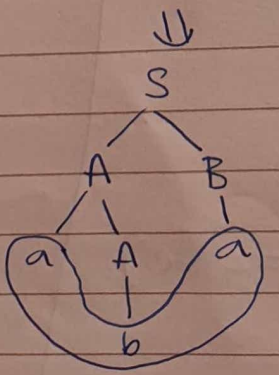
```
function S()
{
    A();
    B();
}

function A()
{
  if (input_symbol == 'a')
  {
    ADVANCE();
    A();
  }
  else
  {
    if(input_symbol=='b')
    {
      ADVANCE();
    }
    else {
      ERROR();
    }
  }
}
```

```
function B()
{
    if (input_symbol == 'b')
    {
        ADVANCE();
        B();
    }
    else if (input_symbol == 'a')
    {
        ADVANCE();
    }
    else
    {
        ERROR();
    }
}
```

⇒ eg: "aba"





Q $E \rightarrow E + E \mid id$ (doesn't work :)

to make it work ⟷ {left recursive}

(for a production of the form

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

⇓

$A \rightarrow \beta A'$

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

∴ $E \rightarrow E+E \mid id$ becomes

$E \rightarrow idE'$

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

```
function E()
{
  ~~if(input_symbol~~
    E();
```

```
function E(){
{
  if(input_sym == 'id')
  {
    ADVANCE();
  }
  else
  {
    E();
    ~~ADVANCE~~
    if(input_sym == '+')
    {
      ADVANCE();
      E();
    }
    else
    {
      ERROR();
    }
  }
}
```

In the above parser example, the parser goes into an infinite loop because the given grammar is left recursive & hence not suitable for Parser Design.

Left Recursive – Any production of the form
$A \rightarrow A\alpha | \beta$

Eliminating left recursive Grammar:
$A \rightarrow \beta A'$
$A' \rightarrow \alpha A' | \epsilon$

∴ $E \rightarrow idE'$
$E' \rightarrow +EE' | \epsilon$

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

A certain grammar / production is said to be left recursive if the left most variable of the RHS is the same as the left most variable of the LHS.

```
        ADVANCE();
        E();
        E'();
    }
    else
    {
        no_action;
    }
}
```

- When we change the production to remove left recursion we should preserve the original grammar (production)
- Grammar input in a parser should be Right recursive

- A production of the Grammar is said to have left recursion if the left most variable of its RHS is the same as the variable in its LHS

eg. $A \rightarrow A\alpha \mid \beta \quad (A = \beta\alpha^*)$

$\Downarrow$

$A \rightarrow \beta A'$

$A' \rightarrow \alpha A' \mid \in$

General form: $A \rightarrow A\alpha_1 \mid A\alpha_2 \mid \ldots \mid A\alpha_n$

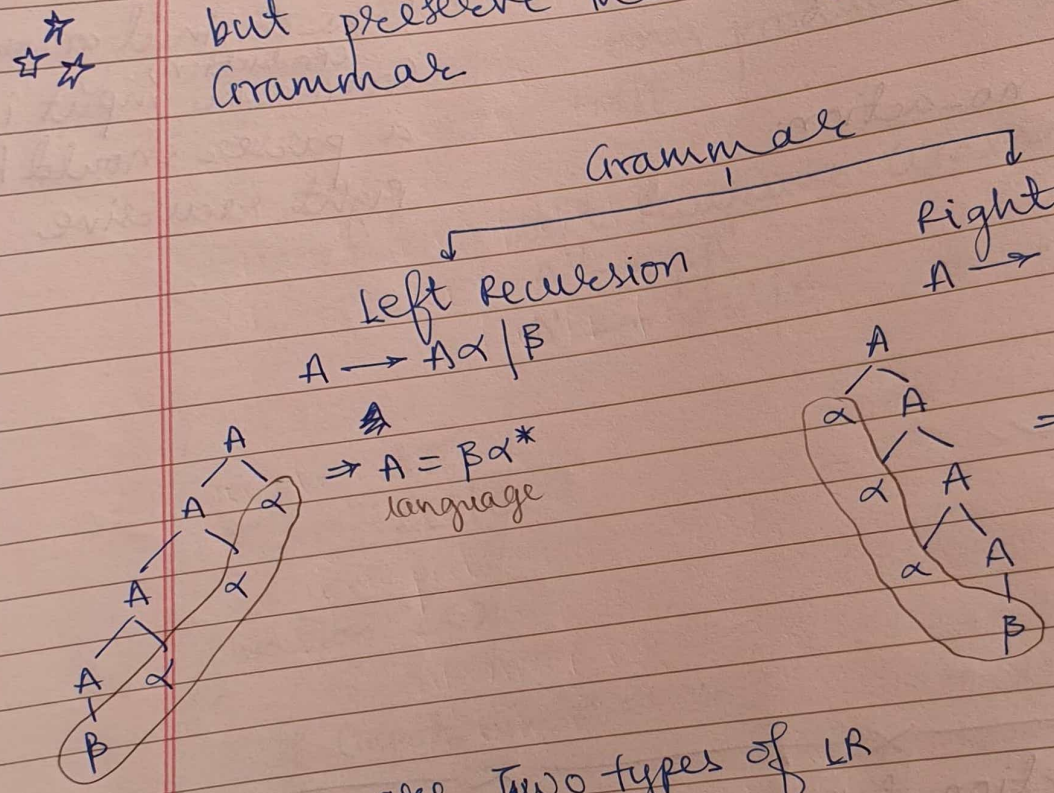
$A \rightarrow \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n$

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

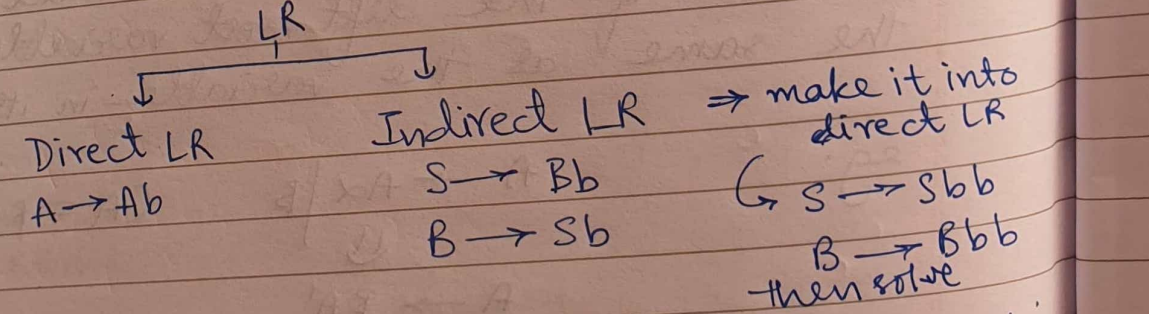
$A' \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \ldots \mid \alpha_n A' \mid \in$

gayatri

- Top down parser cannot handle grammar having left recursion because they are not suitable for parser design.
- We have to remove/eliminate left recursion but preserve the language generated by Grammar



- There are Two types of LR



Q. Eliminate ~~the~~ LR from the following grammar:

E → E+T | T
T → T*E | F
F → id

$E \rightarrow TE'$

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

$T \rightarrow FT'$

$T' \rightarrow *ET' \mid \epsilon$

$F \rightarrow id$

```
function E()
{
    T();
    E'();
}
function E'()
{
    if(input_symbol == '+')
    {
        ADVANCE();
        T();
        E'();
    }
    else
    {
        no_action;
    }
}
function T()
{
    F();
    T'();
}
function T'()
{
    if (input_symbol == '*')
    {
        E();
        T'();
    }
    else
    {
        no_action;
    }
}
function F()
{
    if(inputsym == 'id')
    {
        ADVANCE();
    }
    else
    {
        ERROR();
    }
}
```

Q. $A \to Ba \mid Aa \mid c$ ⇒ make it Direct LR first
$B \to Bb \mid Ab \mid d$

~~$A \to BaA' \mid cA'$~~
~~$A' \to$~~

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

$A \to cA'$
$A' \to baA' \mid aA' \mid \epsilon$
$B \to dB'$
$B' \to bB' \mid abB' \mid \epsilon$

```
function A()
{
    if(input_sym == 'c')
    {
        ADVANCE();
        A'();
    }
}
```

```
function A'()
{
    if (input_sym == 'b')
    {
        ADVANCE();
        if (inp_sym == 'a')
        {
            ADVANCE();
            A'();
        }
    else if (input_sym == 'a')
    {
        ADVANCE();
        A'();
    }
}
```

```
        else
        {
            ERROR(); no-action
        }
}
function B()
{
    if (inp_sym == 'd')
    {
        ADVANCE();
        B'();
    }
    else
    {
        ERROR();
    }
}
function B'();
{
    if (inp_sym == 'b')
    {
        ADVANCE();
        B'();
    }
    else if (inp_sym == 'a')
    {
        if (sym ADVANCE();
        if (inp_sym == 'b')
        {
            ADVANCE();
            B'();
        }
        else
        {
            ERROR();
        }
    }
    else
    {
        no_action;
        ERROR();
    }
}
```

```
function A'()
{
    if (inp_sym == 'b')
    {
        ADVANCE();
        if (inp_sym == 'a')
        {
            ADVANCE();
            A'();
        }
        else
        {
            ERROR();
        }
    }
    else if (inp_sym == 'a')
    {
        ADVANCE();
        A'();
    }
    else
    {
        no_action;
    }
}
```

Q $S \rightarrow Aa \mid b$ $\Rightarrow$ $S \rightarrow Sda \mid b$
$A \rightarrow Ac \mid Sd \mid \epsilon$ $A \rightarrow Ac \mid Aad \mid \epsilon$

$\rightarrow$ $S \rightarrow bS'$
$S' \rightarrow daS' \mid \epsilon$
$A \rightarrow A'$
$A' \rightarrow cA' \mid adA' \mid \epsilon$

$\Rightarrow$ $S \rightarrow bS'$
$S' \rightarrow daS' \mid \epsilon$
$A \rightarrow cA \mid adA \mid \epsilon$

Grammar

Deterministic
- Determines where we want to go exactly
- used in Parser

Non-Deterministic
$A \rightarrow \alpha\beta \mid \alpha\gamma$

Left factoring -
There are two types of grammar - DG & NDG.
- Deterministic Grammar determines where we want to go exactly.
- In DG, there is exactly 1 unique prefix
- In NDG, there is not exactly 1 unique prefix

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

The main cause of NDG is repetition of prefixes. Due to this, the parser is unable to determine to which production it wants to go. Therefore, left factoring comes into the picture.
In left factoring, NDG is converted to DG

Any production of the form $A \rightarrow \alpha\beta \mid \alpha\gamma$ (α = common prefix)

To eliminate (common prefix) left factoring is used –

$A \rightarrow \alpha A'$

$A' \rightarrow \beta \mid \gamma$

General :

~~$A \rightarrow \alpha_1\beta_1 \mid \alpha_2\beta_2 \mid \ldots \mid \gamma_1 \mid \gamma_2$~~

~~$A \rightarrow \alpha_1 A' \mid \alpha_2 A'$~~

~~$A' \rightarrow \alpha$~~

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

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

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

Q Eliminate left factoring

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

$\Rightarrow S \rightarrow aS'$

$S' \rightarrow Sb \mid \epsilon b$

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

$\Rightarrow A \rightarrow aA'$

$A' \rightarrow B \mid bc$

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

$E \rightarrow \epsilon b$

$\Rightarrow S \rightarrow iEtSS' \mid a$

$S' \rightarrow eS \mid \epsilon$

$E \rightarrow b$

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

~~$S \rightarrow aS' \mid b$~~

~~$S' \rightarrow SSbS \mid SaSb \mid bb \mid b$~~

~~$S \rightarrow aSS' \mid aS'' \mid b$~~

$S' \rightarrow$

$S \rightarrow aS' \mid aS'' \mid aS''' \mid b$
$S' \rightarrow SSbS$
$S'' \rightarrow SaSb$
$S''' \rightarrow bb$

$S \rightarrow aSS' \mid aS'' \mid b$
$S' \rightarrow SbS \mid aSb$
$S'' \rightarrow bb$

$S \rightarrow aA' \mid b$
$A' \rightarrow SS' \mid S''$
$S' \rightarrow SbS \mid aSb$
$S'' \rightarrow bb$

* first (α)

- can be applied on all terminals & variables.
- It is defined as a set of terminals that begins with the string derived from α. α is some sentential form

R1] $X \rightarrow a\alpha$
FIRST{X} = {a}

R2] $X \rightarrow \epsilon$
FIRST{X} = {ϵ}

R3] FIRST{a} = {a}
where a is a terminal.

R4] $X \rightarrow Y_1 Y_2 Y_3 \ldots Y_k$
FIRST{X} = FIRST{Y_1}
if Y_1 contains ϵ,
FIRST{X} = FIRST{Y_1} − {ϵ} + FIRST{Y_2}

note: if Y_j contains ϵ then
FIRST{X} = ϵ
where $j = [1, k]$

FOLLOW(A) ← only for variables

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

R1] if Y is a start variable,
FOLLOW(Y) = {$}

R2] X → αY
FOLLOW(Y) = ~~FOLLO~~ FOLLOW(X)

R3] X → αYβ
FOLLOW(Y) = FIRST{β}

if FIRST{β} contains ∈ then
~~FOLLOW(Y) = FOLLOW(X)~~
FOLLOW(Y) = FIRST{β} - {∈} + FOLLOW(X)

14/02/24

Q2 Find first & follow of the given grammar.

S → ABC
A → a | ∈
B → b
C → c | ∈

~~First~~

Q1. Find first -

S → ABC
A → a | ∈
B → b | ∈
C → c | ∈

First(S) ≠ ∈
First(A) = First(B) ≠ First(C) ≠ ∈

first{a} = {a}
first{b} = {b}
first {c} = {c}

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

first{S} = {a, b, c, ε}

2. first{A} = {a, ε}
first {B} = {b}
first {C} = {c, ε}
first {a} = {a}
First {b} = {b}
first {c} = {c}
First {S} = {a, b}

~~follow{A} = follow{S} = {$}~~
~~follow {B} =~~
follow {A} = first {B}? = {b}
follow {B} = first {C}? = {c, $}
follow {C} = follow{S} = {$}
follow{S} = {$}

Q3. E → TE'
E' → +TE' | ε
T → (E) | id

first {+} = {+}
first {(} = {(}
first {)} = {)}
first{id} = {id}

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

follow {E} = { $,) }
follow {E'} = follow {E} = { $,) }
follow {T} = First {E'} = { +, $ }

* look at all instances of the variable being in RHS

follow {E} = { $ } + First {) } = { $,) }
follow {E'} = follow {E} + follow {E'} = { $,) }

follow (T) = ~~First {E'} =~~
follow {E'} =

{ not completed }

Q
E → TE'
E' → +TE' | ϵ
T → F,T'
T' → *FT' | ϵ
F → (E) | id

variable/term.	first
+	+
*	*
(	(
)	)
id	id

First {F} = { (, id }
First {T'} = { *, ϵ }
First {T} = First {F} = { (, id }
First {E'} = { +, ϵ }
first {E} = first {T} = { (, id }

gayatri
- LR|LF
- First & Follow of variables
- Parse table
- Algorithm
- example

Date Page 27

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

Follow (T) = First {E'} + Follow (E)
= {+, $,)}

Follow (E) = {$} + first {)}
= {$,)}

Follow (T') = Follow (T) = {+, $,)}

Follow (E') = Follow (E)
= {$,)}

14/02/24 ~~Module 3 Parser~~

Q. Design predictive parser for the following gramma

$E \rightarrow TE'$

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

$T \rightarrow (E) | id$

(i) Elimination of LR & LF

(ii) First {E} = {(, id} Follow (E) = {$,)}
first {E'} = {+, ε} follow (E') = {$,)}
first {T} = {(, id} follow (T) = {+, $,)}

(iii) Predictive parser table

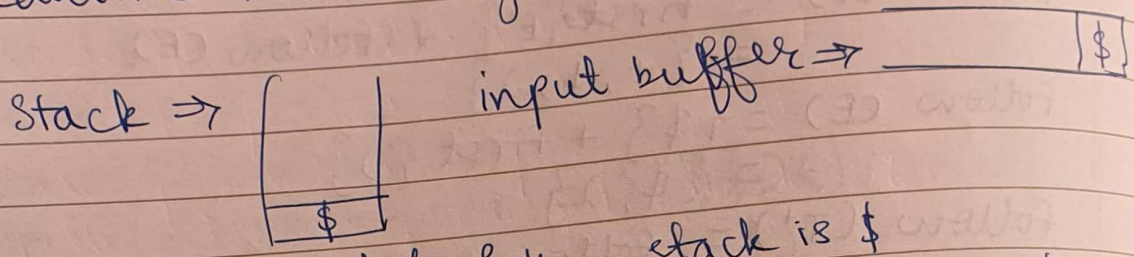
V \ T	id	+	(	)	$
E	E→TE'		E→TE'		
⇒ E'		E'→+TE'		E'→ ε	E'→ ε
T	T→id		T→(E)		

First of E' contains ε, so look at its FOLLOW

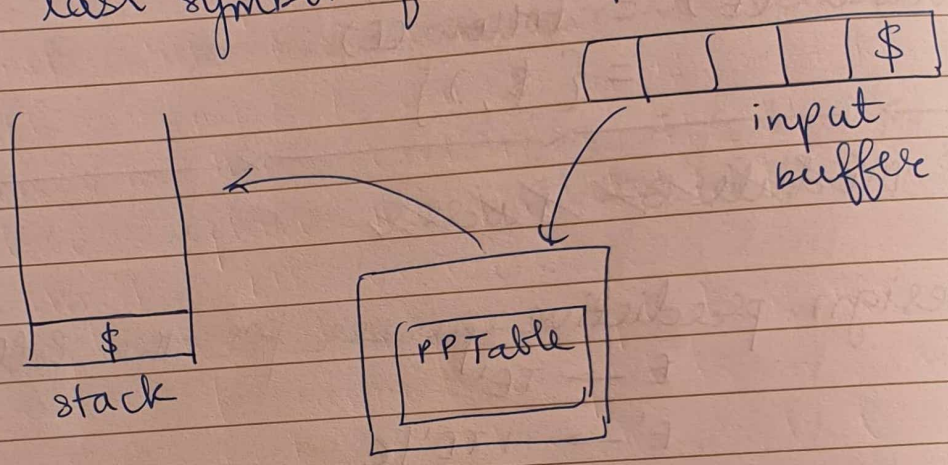
Follow (E') contains), $, so we write the ε production of E' in) & $

* Since each entry of the PP table ~~is~~ ~~one~~ consists of a single production, this is an LL(1) Parser

(iv) Predictive Parser algorithm



* the bottom symbol of the stack is $
* the last symbol of the input buffer is $



(i) Let 'x' be a stack top symbol & 'a' be the input symbol

(ii) if $x = a = \$$ then accept & break

(iii) elif $x = a \neq \$$ then pop x & remove a

(iv) elif $M[x, a]$ $x \rightarrow y_1 y_2 \cdots y_k$ then pop x & push $y_k \cdot y_{k-1} \cdots y_1$

(v) ~~ERR~~ else ERROR().

Q. Test if the following grammar is LL1 or not

$S \rightarrow AaAb \mid BaBa$
$A \rightarrow \epsilon$
$B \rightarrow \epsilon$

(i) ~~LR/LF~~

$S \rightarrow$

(i) LR/LF ✓

(ii) First {B} = {ε}
First {A} = {ε}
~~first {S} = {a}~~
First {S} = {a}

Follow (A) = {a, b}
follow (B) = {a, b}
follow (S) = {$}

(iii) Table

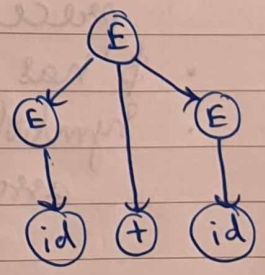
V \ T	a	b	$
A	A→ε	A→ε	
B	B→ε	B→ε	
S	S→AaAb S→BaBb		

* Not an LL(1) Parser

16/2/24 Shift reduce Parser — Handle, Handle pruning, Actions in SR parser, algorithm

Q 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	$	accept

* Handle is a string on the RHS of a production which can be reduced ~~using the~~ by replacing it with the LHS variable in a process called pruning to ~~proo~~ get to the start variable.
* Handle pruning is a process used to reduce a handle by replacing it with the LHS of the production.
* Actions in SR parser -
~~(i) Shift - till the string~~
(i) Shift 0 or more terminals in the stack - till the string in the stack is a handle.
(ii) Reduce - replace the handle in the stack with its LHS variable
(iii) Accept - the string is accepted if the input gets scanned completely and the stack only has the start variable (and the $s of c). The string is then valid
~~is~~
(iv) Reject - the string is invalid.

* Operator Precedence Parser
. An identifier is always ~~dep~~ given higher precedence than other symbols
. $ has the lowest precedence
. Symbols with the same precedence follow associativity

symbol	Precedence	Associativity
$	Lowest	-
+, -	-	Left
*, /	-	Left
^	-	Right
id	Highest	-

operator grammar is a CFG that satifies –
(i) no null production is present
(ii) no two variables are present together

Date
Page 31

* Steps to design & Operator Precedence Parser.

(i) Design Operator Precedence Relation Table (?)

(ii) Algorithm

(a) if input has been completely scanned and the stack only has the start variable (& $),
accept & break

(b) else

let a = topmost stack terminal
b = input terminal

if a $\lessdot$ b or a = b (precedence)
→ SHIFT

else if (a $\gtrdot$ b)
→ REDUCE

else ERROR()

(iii) example

E → E+E | E*E | id

eg: id+id

stack	input	action
$	id+id$	shift
$id	+id$	reduce
$E	+id$	shift (?)
$E+	id$	

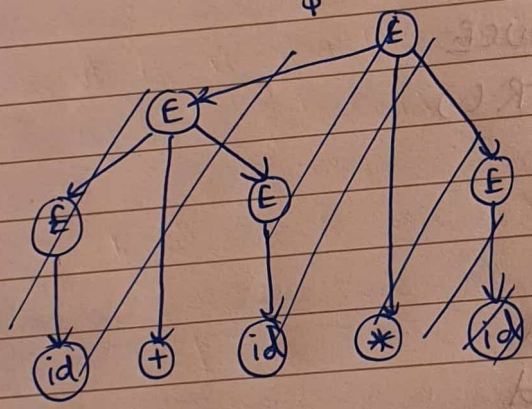
eg: E → E+E | E*E | id

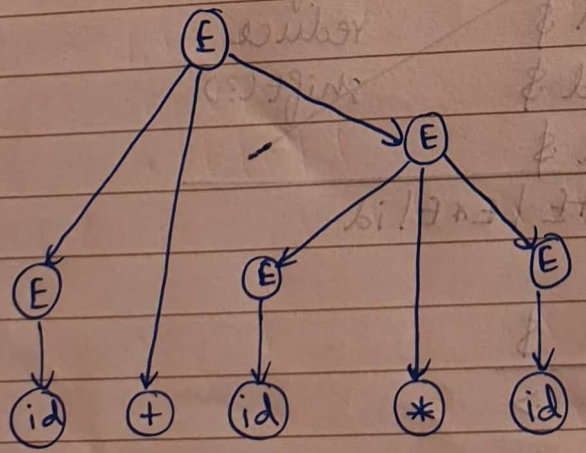
Step1:

	id	+	*	$
id	-	$\gtrdot$	$\gtrdot$	$\gtrdot$
+	$\lessdot$	$\gtrdot$	$\lessdot$	$\gtrdot$
*	$\lessdot$	$\gtrdot$	$\gtrdot$	$\gtrdot$
$	$\lessdot$	$\lessdot$	$\lessdot$	Accept

step2:

Stack	Input	top most stack term (a)	input term (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	*	Reduce E→id
$E+E	*id$	+	*	shift *
$E+E*	id$	*	id	shift id
$E+E*id	$	id	$	Reduce E→id
$E+E*E	$	*	$	Reduce E→E*E
$E+E	$	+E	$	Reduce E→E+E
$E	$	$	$	Accept





E_1 - missing operator (for id id)

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

input = ~~id+id*id~~ id↑id+id

step1

	id	↑	*	+	$
id	-	⋗	⋗	⋗	⋗
↑	⋖	⋖	⋗	⋗	⋗
*	⋖	⋖	⋗	⋗	⋗
+	⋖	⋖	⋖	⋗	⋗
$	⋖	⋖	⋖	⋖	Accept

Stack	input	a	b	Action
$	id↑id+id$	$	id	shift id
$id	↑id+id$	id	↑	Reduce $E \rightarrow id$
$E	↑id+id$	$	↑	shift ↑
$E↑	id+id$	↑	id	shift id
$E↑id	+id$	id	+	Reduce $E \rightarrow id$
$E↑E				

* Operator Grammar is a CFG that-
(i) contains no null productions
(ii) does not contain any productions where variables occur together

Q Converting Grammar to Operator Grammar
(i) $S \rightarrow SASb \mid a$
$A \rightarrow bSb \mid b$

$\Rightarrow S \rightarrow SbSbS \mid SbS \mid a$

*since A is unreachable from the start variable it is useless, do not write it

(ii) $S \rightarrow SES \quad E \rightarrow + \mid *$

$\Rightarrow S \rightarrow S+S \mid S*S$

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

Sol.

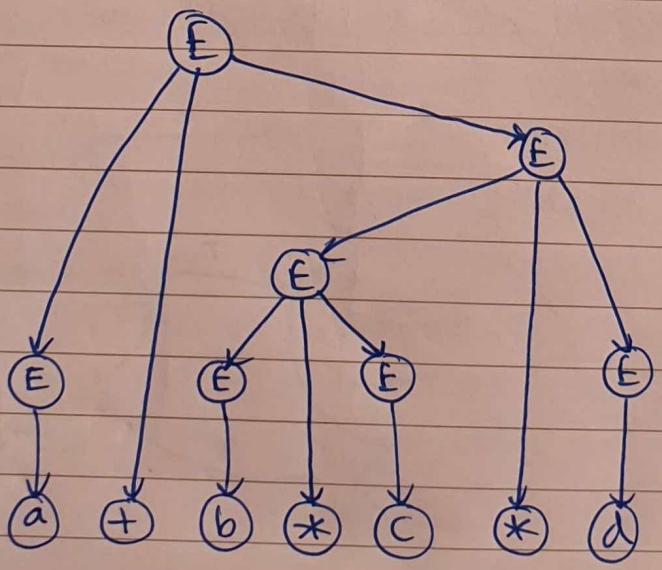
	a	b	c	d	+	*	$
a	–	–	–	–	⋗	⋗	⋗
b	–	–	–	–	⋗	⋗	⋗
c	–	–	–	–	⋗	⋗	⋗
d	–	–	–	–	⋗	⋗	⋗
+	⋖	⋖	⋖	⋖	⋗	⋖	⋗
*	⋖	⋖	⋖	⋖	⋗	⋗	⋗
$	⋖	⋖	⋖	⋖	⋖	⋖	Accept

$E \rightarrow T \rightarrow V \rightarrow a|b|c|d$

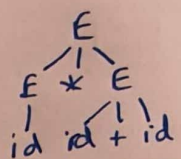
$E \rightarrow E+E \mid E*E \mid a|b|c|d$

&

Stack	Input	a	b	Action
$	a+b*c*d$	$	a	shift a
$a	+b*c*d$	a	+	Reduce E→a
$E	+b*c*d$	$	+	shift +
$E+	b*c*d$	+	b	shift b
$E+b	*c*d$	b	*	Reduce E→b
$E+E	*c*d$	+	*	shift *
$E+E*	c*d$	*	c	shift c
$E+E*c	*d$	c	*	Reduce E→c
$E+E*E	*d$	*	*	Reduce E→E*
$E+E	*d$	+	*	~~reduce~~ shift *
$E+E*	d$	*	d	shift d
$E+E*d	$	d	$	Reduce E→d
$E+E*E	$	*	$	Reduce E→E*
$E+E	$	+	$	Reduce E→E+
$E	$	$	$	Accept



$(a+b)+c$



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

input = ~~E*(E+E~~ id*(id+id)

stack	input	a	b	Action
$	id*(id+id)*$	$	id	shift id
$id	*(id+id)$	id	*	reduce E→id
$E	*(id+id)$	$	*	shift *
$E*	(id+id)$	*	(	shift (

	id	+	*	(	)	$
id	E_1	⋗	⋗	E_1	⋗	⋗
+	⋖	⋗	⋖	⋖	⋗	⋗
*	⋖	⋗	⋗	⋖	⋗	⋗
(	⋖	⋖	⋖	⋖	=	E_2
)	E_1	⋗	⋗	E_1	⋗	⋗
$	⋖	⋖	⋖	⋖	E_2	Accept

*

* SLR

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

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

{ closure, LR(0), complete item }

S1. Augmente the Grammar & list the follows

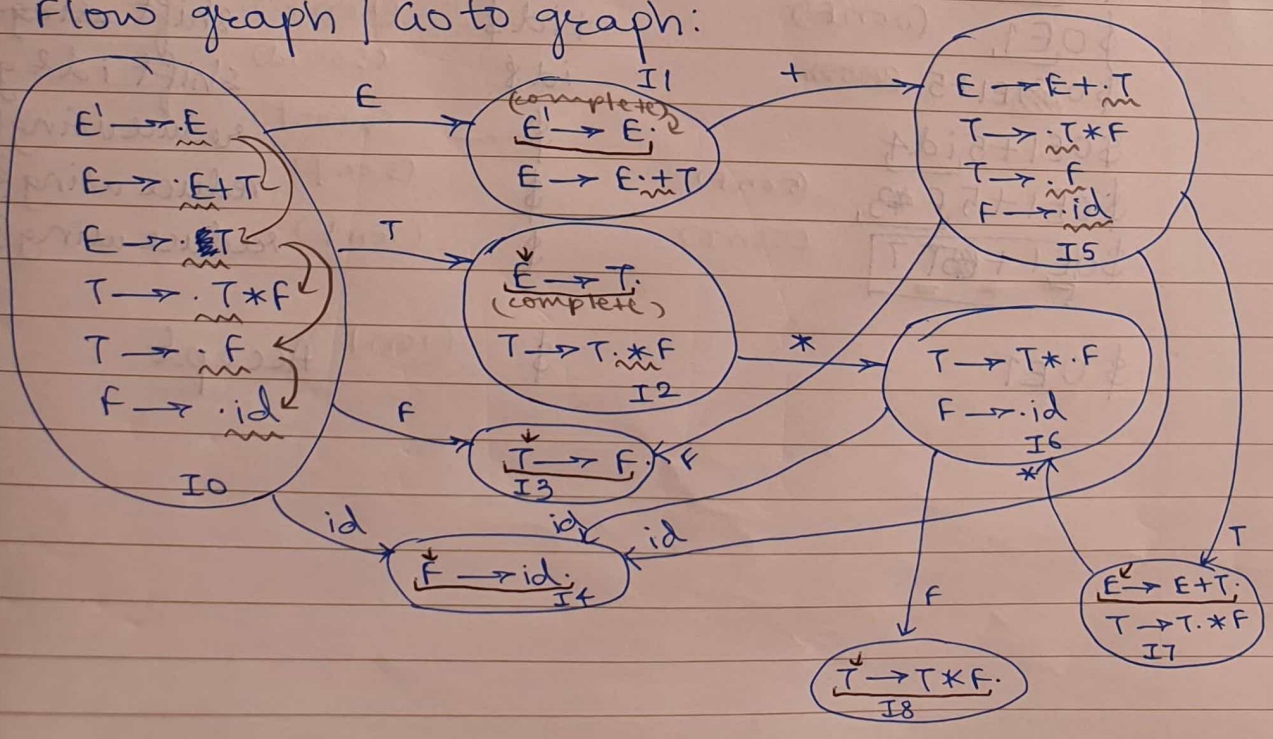
$E' \rightarrow E$

~~① E → E+T~~
~~② T → T*F | F~~
~~③ F → id~~

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

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

S2 Flow graph / Goto graph:



S3 LR(0) Parser Table

States	Actions: id	+	*	$	Goto: 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		r1			
8		r3	r3	r3			

FOLLOW(E) r2 → 2
FOLLOW(T) r4 → 3
FOLLOW(F) r5 → 4
FOLLOW(E) r1 → 7
FOLLOW(T) r3 → 8

S4 eg: id+id

stack	input	Action
$0 ← state	id+id$	(0 on id) shift id & goto 4
$0id4	+id$	(4 on +) reduce using ⑤ F→id
$0F3 (0 on F)	+id$	(3 on +) reduce using ④ T→F
$0T2 (0 on T)	+id$	(2 on +) reduce using ② E→T
$0E1 (0 on E)	+id$	(1 on +) shift & goto 5
$0E1+5	id$	(5 on id) shift id & goto 4
$0E1+5id4	$	(4 on $) reduce using ⑤ F→id
$0E1+5F3 (5 on F)	$	(3 on $) reduce using ④ T→F
$0E1+5T7 (5 on T)	$	(7 on $) reduce using ① E→E+T
$0E1	$	(1 on $) Accept

Q Compute LR(0) item (the flow diagram) for the following production $S \rightarrow XYZ$

$S' \rightarrow .S$ x not required for single production

$S \rightarrow .XYZ$
$S \rightarrow X.YZ$
$S \rightarrow XY.Z$
$S \rightarrow XYZ.$

Q Apply SLR parsing techniques and construct SLR parser table for the following grammar –

$S \rightarrow (S)S$
$S \rightarrow \epsilon$