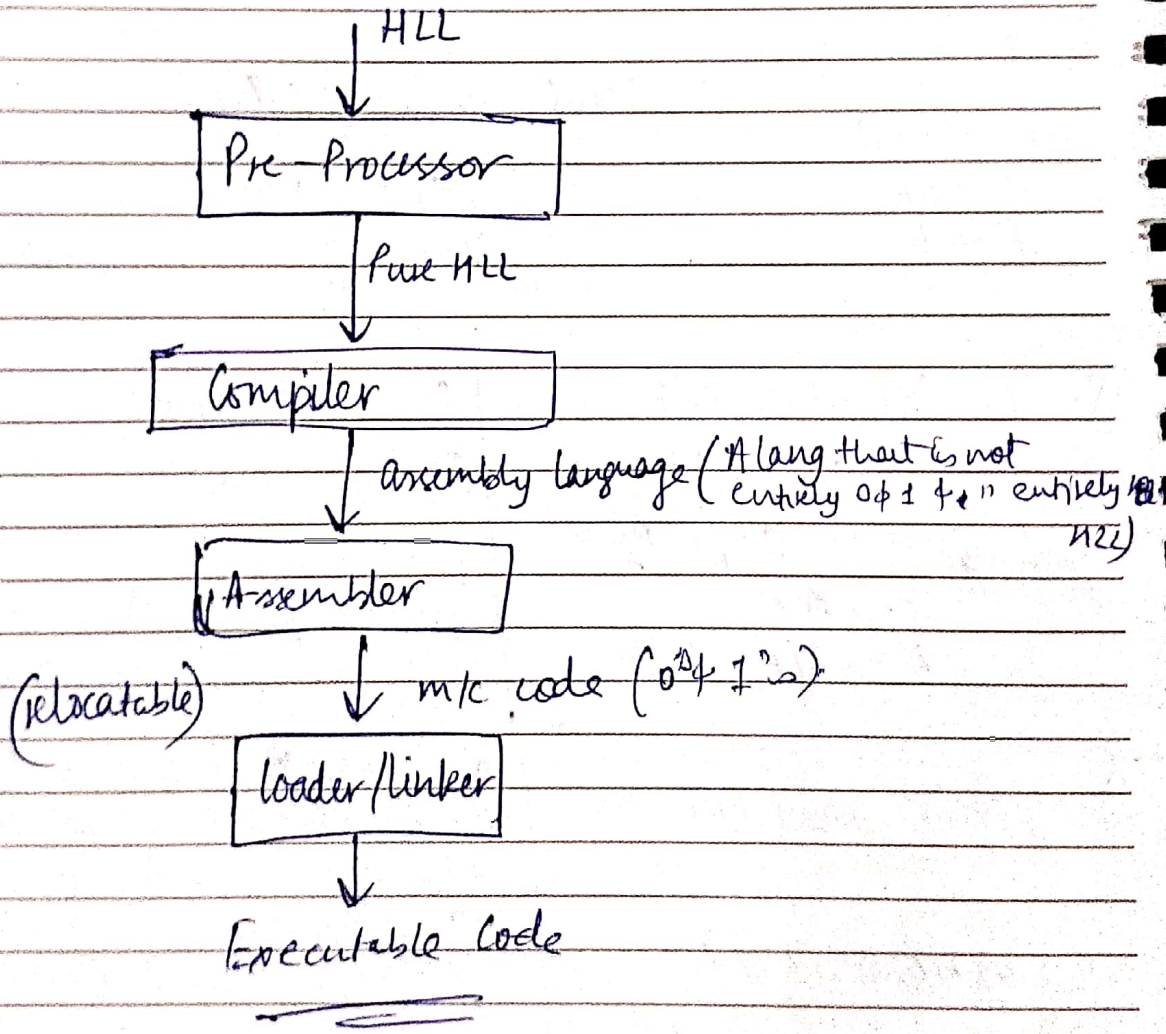
Harman Singh Jolly
IT-E
35451203116

SYSTIMAX®
SOLUTIONS

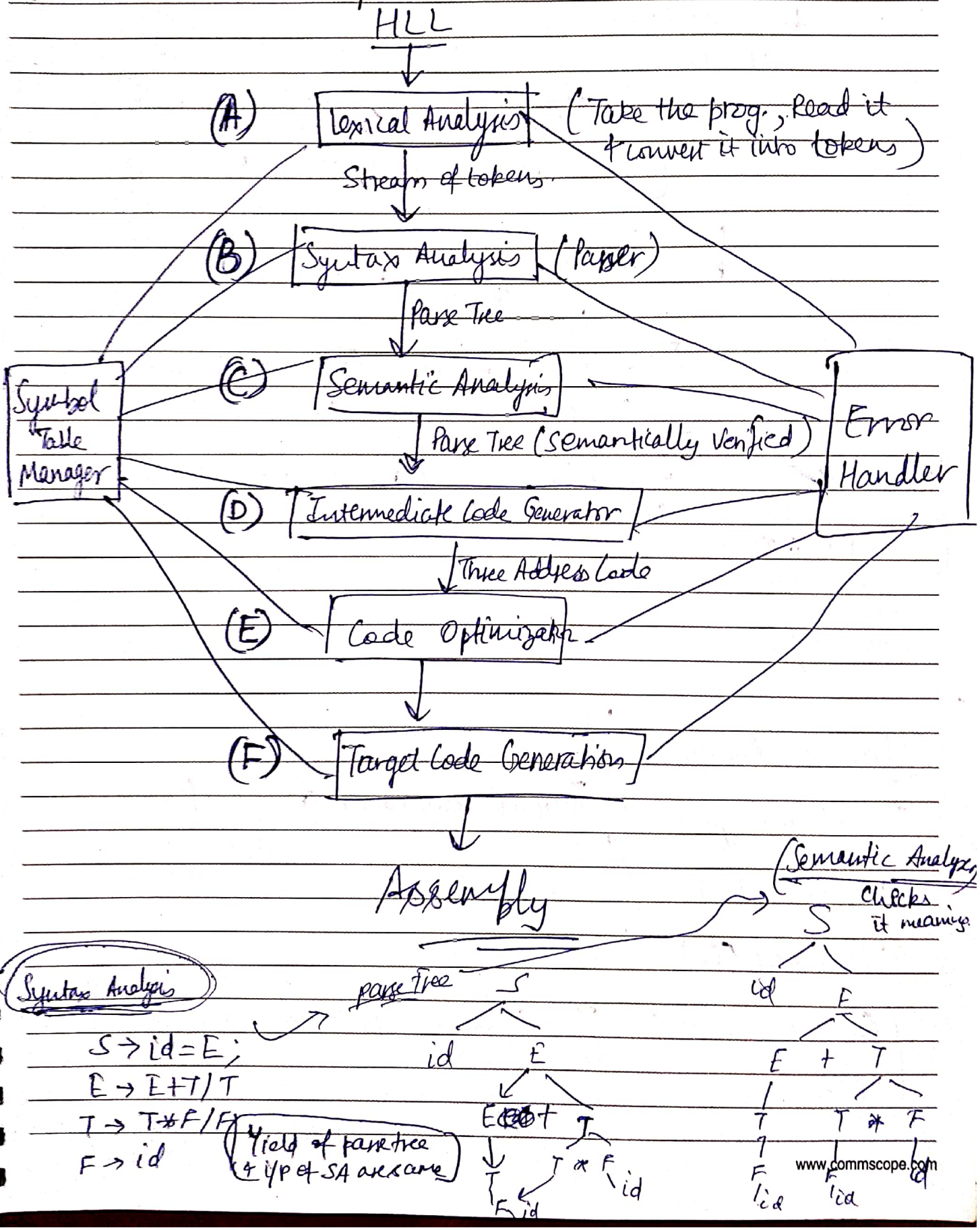
Compiler Design

Compiler - A program that reads a program written in one language, and then translates into an equivalent program in another language.

It also reports errors present in the source program as a part of its translation process.



Stages of Compiler / Phases of Compiler



Syntax Analysis

$S \to id = E;$

$E \to E+T / T$

$T \to T*F / F$

$F \to id$

Yield of parse tree & i/p of SA are same

Parse Tree

(Semantic Analyzer checks it meaning)

www.commscope.com

SYSTIMAX®
SOLUTIONS

ICG

$x = a+b*c;$

$t_1 = b*c;$
$t_2 = a+t_1;$
$x = t_2;$

Code Optimizer (Reduces lines)

$t_1 = b*c;$
$x = a+t_1;$

O/p is given to TCG

Target code Generator (Write the code which assembler can understand)

As assembler is dependant upon the platform ∴ depending on what assembler we use, we have to write code

```
mul R1, R2
add R0, R2
mov R2, x
```

a = R0 → Memory in Assembler
b = R1
C = R2

Lexical Analysis
Syntax Analysis
Semantic "
Intermediate CG

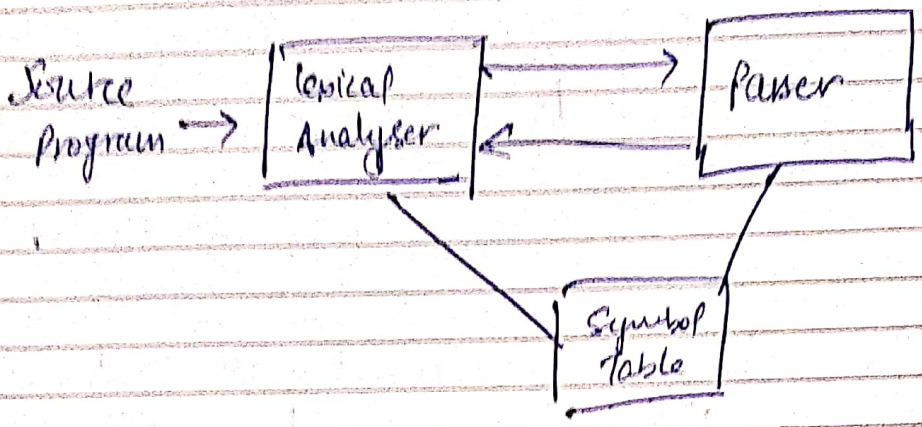
They all come under analysis part or frontend

Code Optimization
T Code Generation

Synthesis part or backend

(A) Lexical Analysis

→ Eliminates comments & blankspaces & throwing error.



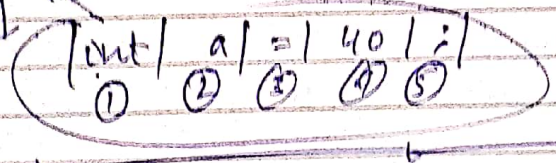
LA converts the source program into a stream of valid words, known as tokens

→ Token of LA are terminal symbols of parser

Token - A string of char which logically belong together

Pattern - Set of strings for which the same token is produced.

Lexeme - Sequence of chars matched by a pattern to form a corresponding token

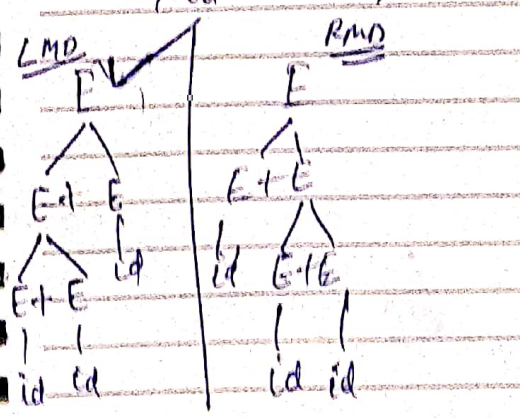


(1) Ambiguous & Unambiguous

Conversion of Ambiguous to Unambiguous.

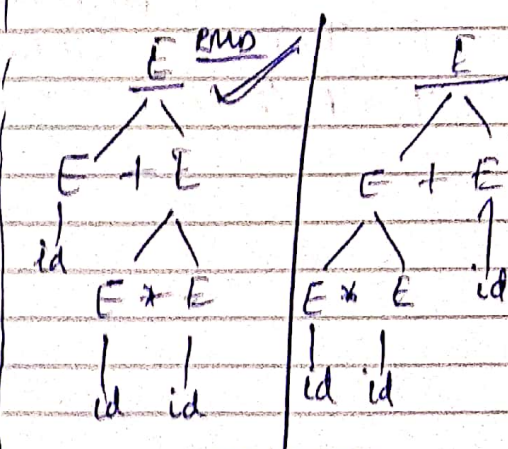
$E \to E+E$
$/E*E$
$/id$

(id+id*id)



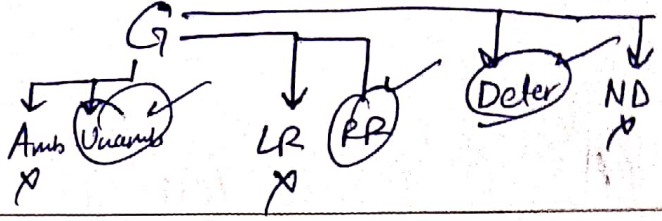
Associativity

Q. (id+id*id)



Precedence

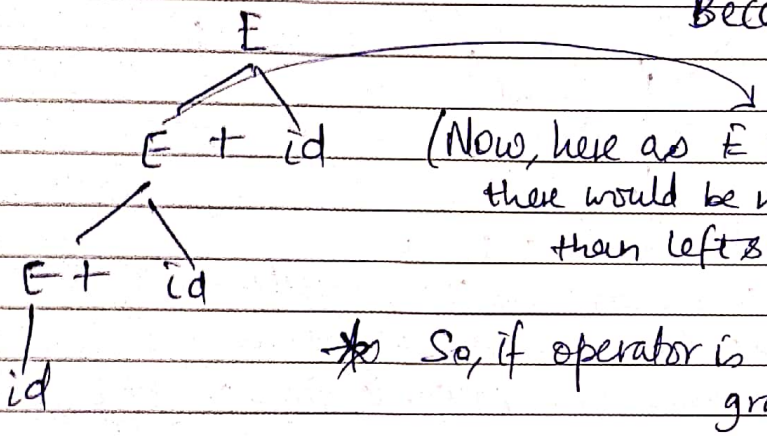
Hence, take care of these to make unambiguous grammar



So, if we keep on going in left direction, it will be left associative.
& in " " " " " " right " " " " right ".

Eg- $E \to E+id/id$

Associatively



Because (+) is left associative hence, we need it to grow in left side only.
(Now, here as E is written left side, hence there would be no place to go to other than left side).

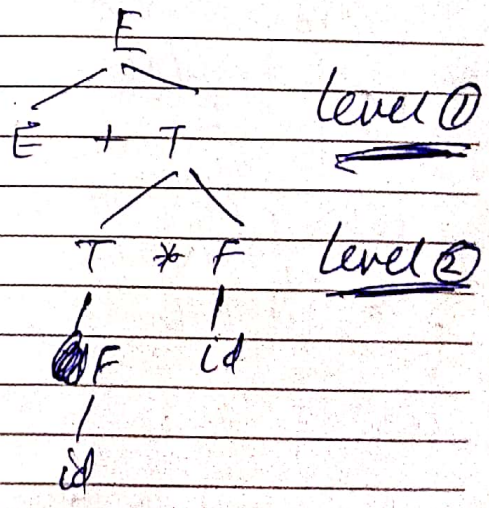
* So, if operator is left associative, then grammar has to be left recursive.
(Left most symbol in RHS = LHS)

* & If operator is right associative, then grammar has to right recursive.
Right most symbol in RHS = LHS.

Precedence

Right precedence operator should be written at least level.
So, if we have +, * in same grammar so we have to put + above *.

Eg
$E \to E+T/T$
$T \to T*F/F$
$F \to id$
} Write left Recursive always.



id+id*id

Q. Make it unambiguous.

$A \to A \$ B / B$

$B \to B \# C / C$

$C \to C @ D / D$

$D \to d$

$\Rightarrow$

$E \to E \$ B / B$

$B \to B \# C / C$

$C \to C @ D / D$

$D \to d$

Q.

$F \to E * F$

$/ F + E$ ② same level only, unless the vertex doesn't change

$/ F$

$F \to F - F$

$/ id$

↑ → Right associative

Ans → As (−) is written at a further away level, so its given the highest precedence

→ (*) is left associative & (+) is right associative here because of ②

hence $(*) \Rightarrow (*) \mid (+) \Leftarrow (+)$ and the (−)

② Left Recursive & Right "

Recursion

Left

$A \to A\alpha / \beta$

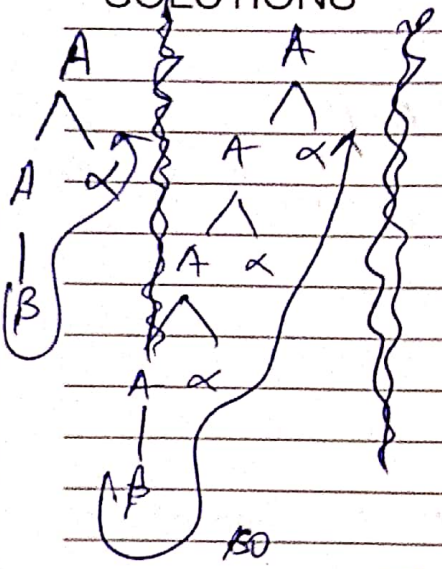
Right

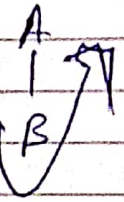
$A \to \alpha A / B$

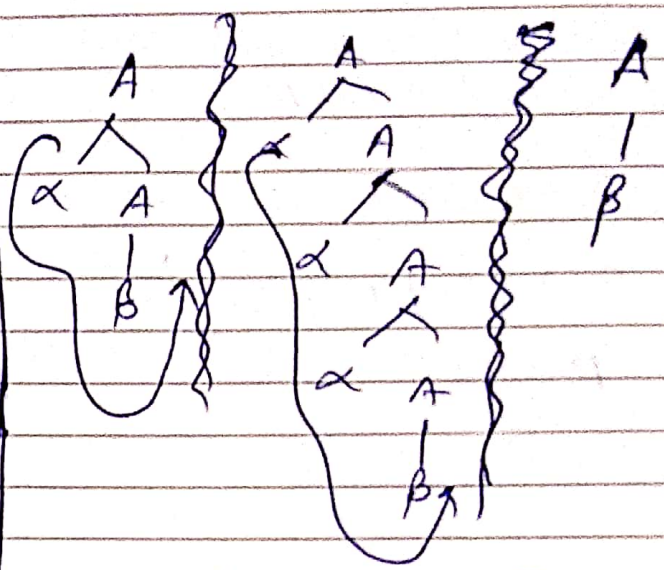
P.T.O.

SYSTIMAX®
SOLUTIONS

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







So

```
A() -②
{
  A() -①
  α
}
```

It makes it look it an infinite loop ② calls ① everytime it enters

$L = \beta\alpha^*$

```
A() -②
{
  α  -③
  A() -①
}
```

Now ③ acts like a check cond, & hence no infinite loop.

$L = \alpha^*\beta$

So, we want to eliminate left recursion without changing the language

hence as

$L = \beta\alpha^*$ (Left Recursive Grammar)

$G \to \begin{bmatrix} A \to \beta A' \\ A' \to \epsilon / \alpha A' \end{bmatrix} \iff A \to A\alpha / \beta$

So, [Right Recursive Grammar] $\iff$ [Left Recursive Grammar]

$S \to S+S$
$/ S*S$
$/ 2$

$2+2*2$

Q. Convert LRG to RRG

$E \Rightarrow E+T/T$

$A \quad \alpha / \beta$

Follow this

$A \rightarrow \beta A'$

$A' \rightarrow \epsilon / \alpha A'$

Ans.

$E \rightarrow TE'$

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

Q.

LRG $\Rightarrow S \rightarrow SOS/S/01$ { $S \rightarrow SOS/S/01$

Convert it into RRG.

Ans

$S \rightarrow \underbrace{S}_{A}\ \underbrace{OS/S}_{\alpha}/\underbrace{01}_{\beta}$] LRG

$A \rightarrow 01A'$

$A' \rightarrow \epsilon / OSA'/SA'$] RRG

Q. $A \rightarrow A\alpha_1/A\alpha_2/A\alpha_3 \cdots /\beta_1/\beta_2/\beta_3 \cdots$ LRG

Ans

$A \rightarrow \beta_1 A'/\beta_2 A'/\beta_3 A'$

$A' \rightarrow \alpha_1 A'/\alpha_2 A'/\alpha_3 A' \cdots$] RRG

SYSTIMAX®
SOLUTIONS

(3) Deterministic & Non-deterministic

Top-down Parsers fail

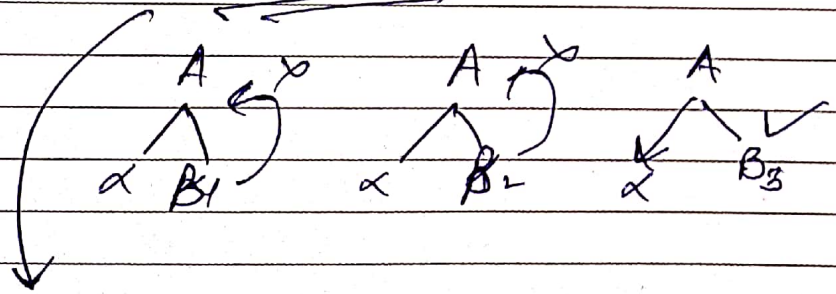
Eg $A \to \alpha\beta_1 / \alpha\beta_2 / \alpha\beta_3$.

So here A after reaching α can go to β_1 ✓
β_2 ✓
β_3 ✓

So, Non-Deterministic exists hence it becomes an issue.

So to remove non-deterministic, we use left factoring.

Eg $A \to \alpha\beta_1 / \alpha\beta_2 / \alpha\beta_3$



So, these trees make backtracking & then again make decision.
hence, eliminate this by left factoring / like
↳ or eliminating non-determinism

$A \to \alpha A'$
$A' \to \beta_1 / \beta_2 / \beta_3$

Q. Convert ND to Deterministic

$S \to iEtS$ ①
$/ iEtSeS$ ②③
$/ a$
$E \to b$ ✓

→ common from both the above.

$S \rightarrow iEtSS'/a$ → for ③

$S' \rightarrow$ [illegible] / eS → for ① → for ②

$E \rightarrow b$ Final

$E \rightarrow iEtSS'/a$

$S' \rightarrow \epsilon/eS$

$E \rightarrow b$

Left factoring - Procedure that is used to convert non-deterministic grammar to deterministic grammar is called

Q.

$S \rightarrow aSSbS$
$/aSaSb$
$/aabb$
$/b$

Ans

$S \rightarrow aSS'/b$

Final

$S \rightarrow aS'/b$

$S' \rightarrow SSbS$
$/SaSb$
$/bb$ so

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

$S' \rightarrow SY'/bb$
$Y' \rightarrow SbS$
$/aSb$

$S \rightarrow aS'/b$
$S' \rightarrow SY'/bb$
$Y' \rightarrow SbS$
$/aSb$

SYSTIMAX
SOLUTIONS

Q. $S \to bSSaaS \mid bSSaSb \mid bSb \mid a$

Ans.

$S \to bSS' \mid a$

$S' \to SaaS \mid SaSb \mid b$ } $S' \to SaY' \mid b$

$Y' \to aS \mid Sb$

$S \to bSS' \mid a$

$S' \to SaY' \mid b$

$Y' \to aS \mid Sb$

(B) Syntax Analysis (Parser) [Ambiguous ✗]

- Top Down (TDP)
  - TDP with full Backtracking → Bruteforce
  - TDP without Backtracking [LR ✗, NDX] (Grammar should be LR)
    - Recursive descent
    - Non-Recursive descent (LL(1))
- Shift Reduce Parser (SR Parser)
  - Bottom Up (BUP)
    - Operator Precedence Parser [Ambiguous]
    - LR Parser
      - LR(0)
      - SLR(1)
      - LALR(1)
      - CLR(1)

Q. Difference (LMD / RMD)

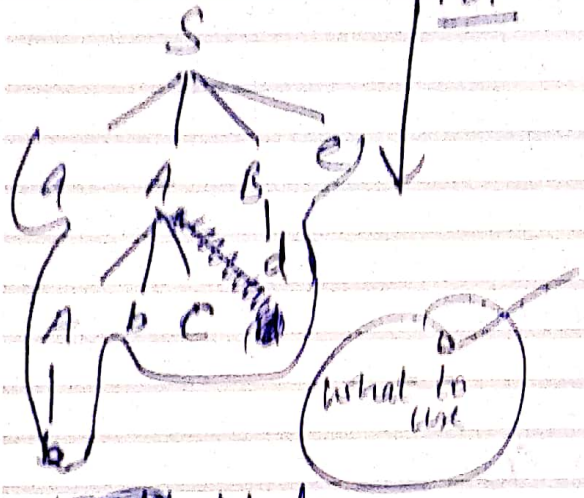
TDP

BUP

Q. Check whether string = abbcde can be constructed using
S → aABe
A → Abc / b
B → d.

(LMD)

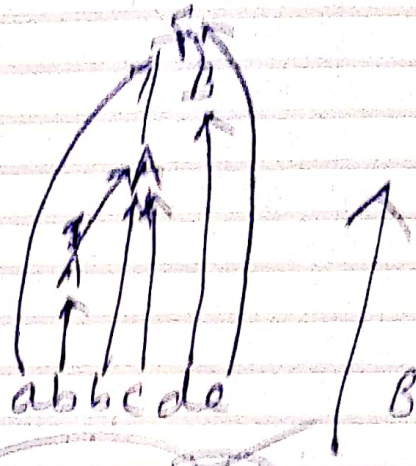
TDP



what to use

aAbcde → aBde → abbcde.

(Reverse of RMD)



BUP

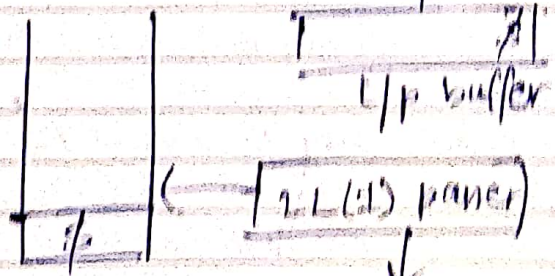
When to reduce

LL(1) Parser

(Left-to-right, Left-most derivation, (1))

no. of lookahead
how many symbols will you see to a decision can be made

LL(1)



LL(1) parsing table

* LR ✗
* ND ✗

Imp

Using this everything is done.

www.commscope.com

Video-6

SYSTIMAX® SOLUTIONS

For finding follow(), search for that Variable in the right side and then find the first of the symbol in right side.
Substituting ∈, and then that variable vanishes.

LL(1) parsing table - Before this, we should learn 2 things:
First() & FOLLOW()

Q. Find FIRST() & FOLLOW()

i) $S \to ABCDE$
$A \to a/\epsilon$
$B \to b/\epsilon$
$C \to c$
$D \to d/\epsilon$
$E \to e/\epsilon$

ii) $S \to Bb/Cd$
$B \to aB/\epsilon$
$C \to cC/\epsilon$

iii) $E \to TE'$
$E' \to +TE'/\epsilon$
$T \to FT'$
$T' \to *FT'/\epsilon$
$F \to id/(E)$

iv) $S \to ACB/CbB/Ba$
$A \to da/BC$
$B \to g/\epsilon$
$C \to h/\epsilon$

Ans:

i)

	FIRST	FOLLOW
S	{a, b, c}	{$}
A	{a, ∈}	{b, c}
B	{b, ∈}	{c}
C	{c}	{d, e, $} ①
D	{d, e}	{e, $}
E	{e, ∈}	{$}

(Check if variable occurs in RHS).

* FOLLOW() of start symbol is always $ & then check if it occurs in RHS.

* Whenever a Variable is occurring at the RHS & there is nothing after it, then its follow() is automatically of LHS.

Eg $S \to ABCD(E)$ → Here follow(E) is the S itself as nothing follows E but S only so follow(E) = {$}

* Whenever every Variable at RHS goes to ∈, then the follow becomes the Variable itself. like the case ①.

* For find First(), always leave the topmost production and find the first of the derived

Eg. FIRST(S) is done by finding FIRST(A) and then doing FIRST(S)

Watch Video-6 (Ravindrababu).

ii)

	First	Follow
S	{a, b, c, d}	{$}
B	{a, ε}	{b}
C	{c, ε}	{d}

iii)

	FIRST	FOLLOW
E	{id, (}	{$,)}
E'	{+, ε}	{$,)}
T	{id, (}	{+, *, $,)}
T'	{*, ε}	{+, $,)}
F	{id, (}	{*, +, $,)}

iv)

	FIRST	FOLLOW
S	{d, g, h, ε, b, a}	{$}
A	{d, g, h, ε}	{h, g, $}
B	{g, ε}	{$, a, h, g}
C	{h, ε}	{g, $, b, a}

v)

S → aABb
A → c/ε
B → d/ε

	First	Follow
S	{a}	{$}
A	{c, ε}	{d, b}
B	{d, ε}	{b}

SYSTIMAX®
SOLUTIONS

8.28 , 1:05:32
8:30

vi)

	First	Follow
$S \to aBDh$	{a}	{$}
$B \to cC$	{c}	{g, f, h}
$C \to bC/\epsilon$	{b, ϵ}	{g, h, f}
$D \to EF$	{g, ϵ, f}	{h}
$E \to g/\epsilon$	{g, ϵ}	{f, h}
$F \to f/\epsilon$	{f, ϵ}	{h}

Now, that we know how to find first() & follow(), then

Parsing Table is made :-

Eg.

	First	Follow
$E \to TE'$	{id, (}	{$,)}
$E' \to +TE'/\epsilon$	{+, ϵ}	{$,)}
$T \to FT'$	{id, (}	{+,), $}
$T' \to *FT'/\epsilon$	{*, ϵ}	{+,), $}
$F \to id/(E)$	{id, (}	{*, +,), $}

Terminals

Variables	id	+	*	(	)	$
E	$E \to TE'$			$E \to TE'$		
E'		$E' \to +TE'$			$E' \to \epsilon$	$E' \to \epsilon$
T	$T \to FT'$			$T \to FT'$		
T'		$T' \to \epsilon$	$T' \to *FT'$		$T' \to \epsilon$	$T' \to \epsilon$
F	$F \to id$			$F \to (E)$		

first()

Fill the of production in the table & whenever you see ϵ use follow() to fill it.

Q. $S \to (S) / \epsilon$

	First	Follow
	{(, ϵ}	{), $}

	(	)	$
S	S→(S)	S→ϵ	S→ϵ

S → A
A → aB / Ad
B → b
C → g

S → A
A → aBA'
A' → dA' / ϵ
B → b
C → g

	First()	Follow()
S	{a}	{$}
A	{a}	{$}
A'	{d, ϵ}	{$}
B	{b}	{d, $}
C	{g}	

Parsing Table

Contd.

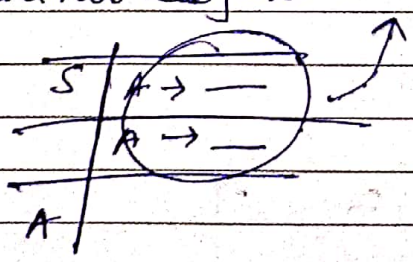
Q. $S \to AaAb / BbBa$
$A \to \epsilon$
$B \to \epsilon$

	First()	Follow()
S	{a, b}	{$}
A	{ϵ}	{a, b}
B	{ϵ}	{b, a}

	a	b	$
S	S→AaAb	S→BbBa	
A	A→ϵ	A→ϵ	
B	B→ϵ	B→ϵ	

Parsing Table

* There can be possibility that each row doesn't have only one answer but multiple answers and not just one. In this case, we can't make LL1 parser or table



SYSTIMAX®
SOLUTIONS

Q.

$S \to aSbS$
$/ bSaS$
$/ \epsilon$

Ans

	FIRST()	FOLLOW()
$S \to aSbS$ / $bSaS$ / ϵ	{a, b, ε}	{$, b, a}

	a	b	$
$S \to aSbS / bSaS / \epsilon$	$S \to aSbS$	$S \to bSaS$	$S \to \epsilon$
	$S \to \epsilon$	$S \to \epsilon$	

As same variable S has 2 answers for a & for b hence, its not LL(1).

Q.

$S \to aABb$
$A \to c / \epsilon$
$B \to d / \epsilon$

	FIRST()	FOLLOW()
$S \to aABb$	{a}	{$}
$A \to c / \epsilon$	{c, ε}	{d, b}
$B \to d / \epsilon$	{d, ε}	{b}

	a	b	c	d
S	$S \to aABb$			
A		$A \to \epsilon$	$A \to c$	$A \to \epsilon$
B		$B \to \epsilon$		$B \to d$

Q.

$S \to A / a$
$A \to a$

	FIRST()	FOLLOW()
$S \to A / a$	{a}	{$}
$A \to a$	{a}	{$, a}

	a	$
S	$S \to a$ (also $S \to a$)	
A	$A \to a$	

{S → A} ∩ {S → a}

Hence, NOT LL1

Q. $S \to aB / \epsilon$
$B \to bC / \epsilon$
$C \to cS / \epsilon$

	FIRST	FOLLOW()
$S \to$	{a, ϵ}	{$}
$B \to$	{b, ϵ}	{$}
$C \to$	{c, ϵ}	{$}

	a	b	c	$
S	S→aB	S→bC	S→cB	S→$
B				B→$
C				C→$

Q. $S \to AB$
$A \to a / \epsilon$
$B \to b / \epsilon$

	FIRST()	FOLLOW()
$S \to AB$	{a}	{$}
$A \to a / \epsilon$	{a, ϵ}	{b, $}
$B \to b / \epsilon$	{b, ϵ}	{$}

	a	b	$
S	S→AB		
A	A→a	A→ε	A→$
B		B→b	B→ε

Q. $S \to aSA / \epsilon$,
$A \to c / \epsilon$

	FIRST()	FOLLOW()
$S \to aSA / \epsilon$	{a, ϵ}	{c, $}
$A \to c / \epsilon$	{c, ϵ}	{$, c}

	a	c	$
S	S→aSA	S→ε	S→ε
A		A→c	A→ε

Q. $S \to A$
$A \to Bb / Cd$
$B \to aB / \epsilon$
$C \to cC / \epsilon$

	FIRST()	FOLLOW()
$S \to A$	{a, b, c, d}	{$}
$A \to Bb / Cd$	{a, c, ϵ}	{$}
$B \to aB / \epsilon$	{a, ϵ}	{b}
$C \to cC / \epsilon$	{c, ϵ}	{d}

www.commscope.com

4

Q.

	FIRST()	FOLLOW()
$S \to aAa/\epsilon$	$\{a, \epsilon\}$	$\{a, \$\}$
$A \to abS/\epsilon$	$\{a, \epsilon\}$	$\{a\}$

✓

Q.

	FIRST()	FOLLOW
$S \to iEtSS'/a$	$\{i/a\}$	$\{\$, e, \epsilon\}$
$S' \to eS/\epsilon$	$\{e, \epsilon\}$	$\{\$, e\}$
$E \to b$	$\{b\}$	$\{t\}$

✓

LL(1) finishes

Shift Reduce Parsing – It constructs parse tree for an input string beginning from the leaves node and working up towards the root.

Operator Precedence Parser

Operator grammar – Grammar that consists of operator only and no two variables are alongside each other but have an operator in between.

$E \to E+E/E*E/id$ ✓

$E \to EAE/id$ ✗
$A \to +/*$

11 only

$S \to SAS/a$
$A \to bSb/b$ → ✗

But we have to make it operator grammar,

$S \to SbSbS/SbS/a$
$A \to bSb/b$ ✓

Video-9

CommScope

Even though grammer is ambiguous, Operator precedence parses it and is able do so by operator relation table. So, we construct operation relation table,

Eg $E \to E+E / E*E / id$.

	(i) id	(ii) +	(iii) *	(iv) $
(1) id	—	⋗	⋗	⋗
(2) +	⋖	⋗	⋖	⋗
(3) *	⋖	⋗	⋗	⋗
(4) $	⋖	⋖	⋖	—

Compare (i) & (i), as same hence NA.

compare (i) & (ii), so (i) is given more priority hence, ⋗.

Now using this table, Operator Precedence Parser will pass input

⋖ · ⇒ Push into stack.
⋗ ⇒ Pop from the stack

* Comparing is always done by stack top first and then with exp.

So, exp ⇒ (2) id + id * id $

(1) | $ | (i) id |

Now compare (1) with & (2), $ & id. As, they are smaller than so push id in stack.

* Now compare (i) and (2) & i.e. id and +. Here id is ⋗ +, so pop id.

SYSTIMAX SOLUTIONS

id + id * id $

* Next we check ① $ and + ② ↑. As ↑ > $, push + in stack

$	id	+

* Now compare + and id.

id + id * id $

$	id	+	id

id + id * id $

By * Comparing + and *

id + id * id $

$	id	+	id	*

id + id * id $

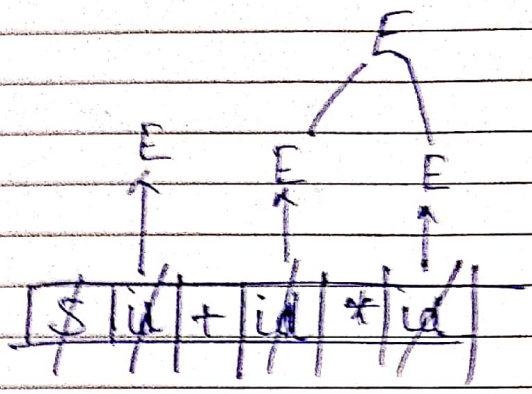
$	id	+	id	*	id

id + id * id $

$	id	+	id	*	id

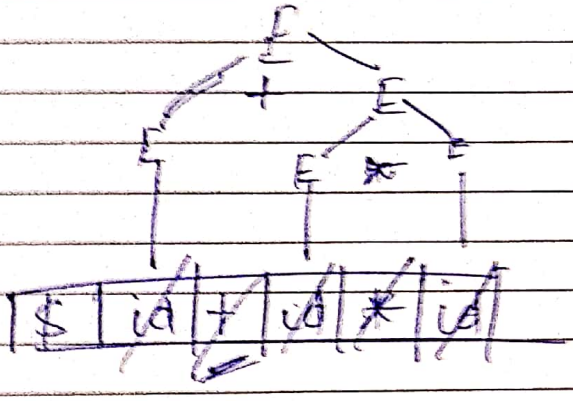
P.T.O

id + id * id $



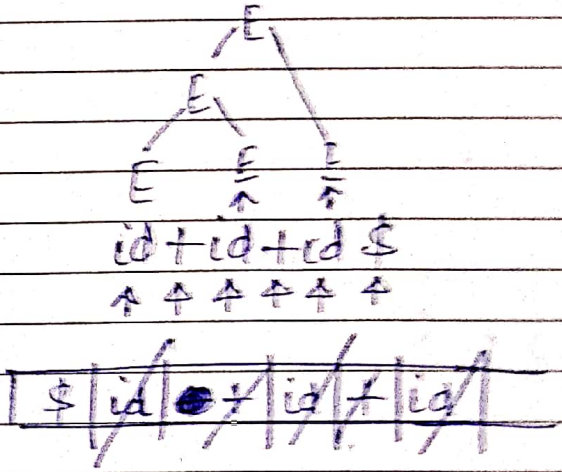
Now, according to stack * and id are compared & * is popped

id + id * id $



* Now, + and $ are being compared and pointer + is popped.

Q. id + id + id $



* As here, there are four operators hence table is 4x4, operator relation table. But if there are n operators, then n^2 relation table, hence we ~~make~~ operator function table.

SYSTIMAX®
SOLUTIONS

Q.

P → SR/S

R → bSR/bS

S → WbS/W

W → L*W/L

L → id

As the Grammar has two variables Where Br. Not operator Grammar, so we need to make it.

Ans

P → SbSRR/SbS/S

S → WbS/W → As it right recursive so *<·*.

W → L*W/L

L → id

P → SbP/SbS/S

For * and b, as * is lower to b no it gets the most precedence.

	id	*	b	$
id	-	·>	·>	·>
*	<·	<·	·>	·>
b	<·	<·	·>	·>
$	<·	<·	<·	-

LR Parsers

- LR(0)
- SLR(1) — Simple LR
- LALR(1) — Look ahead LR
- CLR(1) — Canonical LR

LR(0), SLR(1) → canonical collection of LR(0) items

LALR(1), CLR(1) → canonical collection of LR(1) items

LR(0)

Eg

Q- $S \to AA$
$A \to aA/b$

Ans: Whenever any grammar is given, we add one more production to grammar,

$S' \to S$, So, $S' \to S$, $S \to AA$, $A \to aA/b$ → Augmented grammar

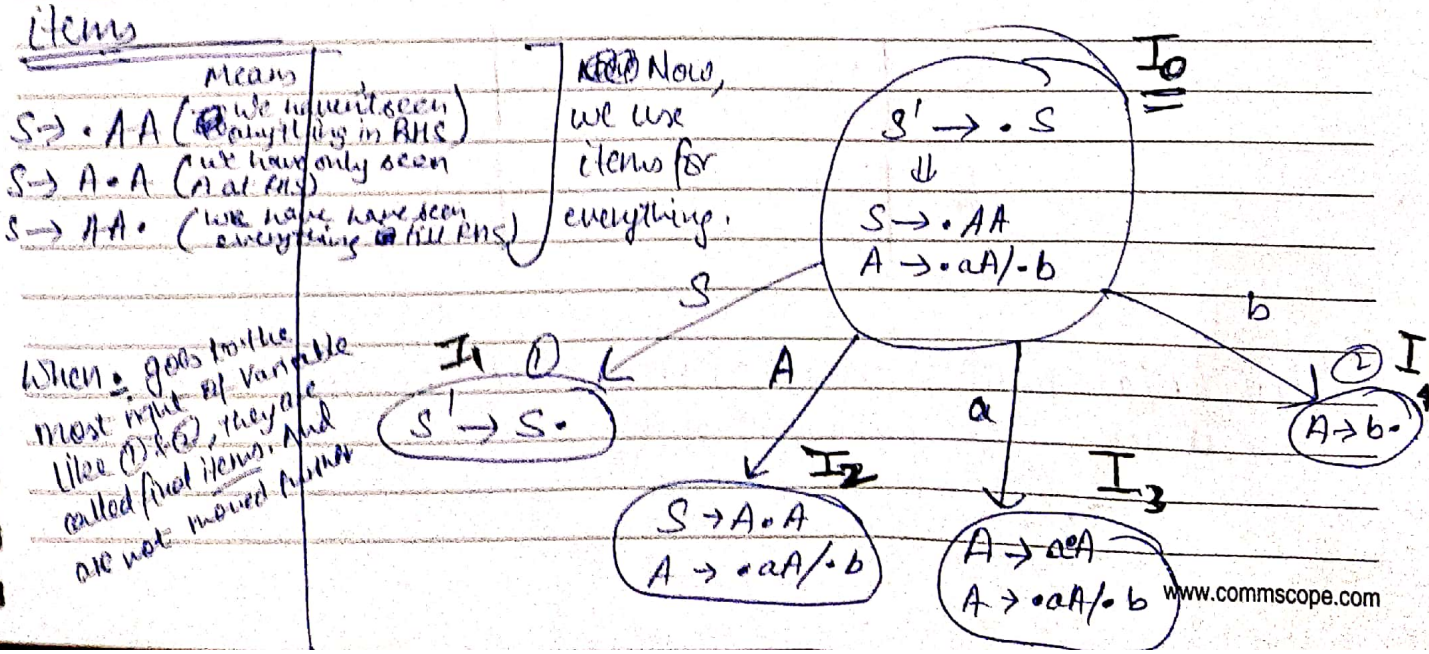
Items

Means

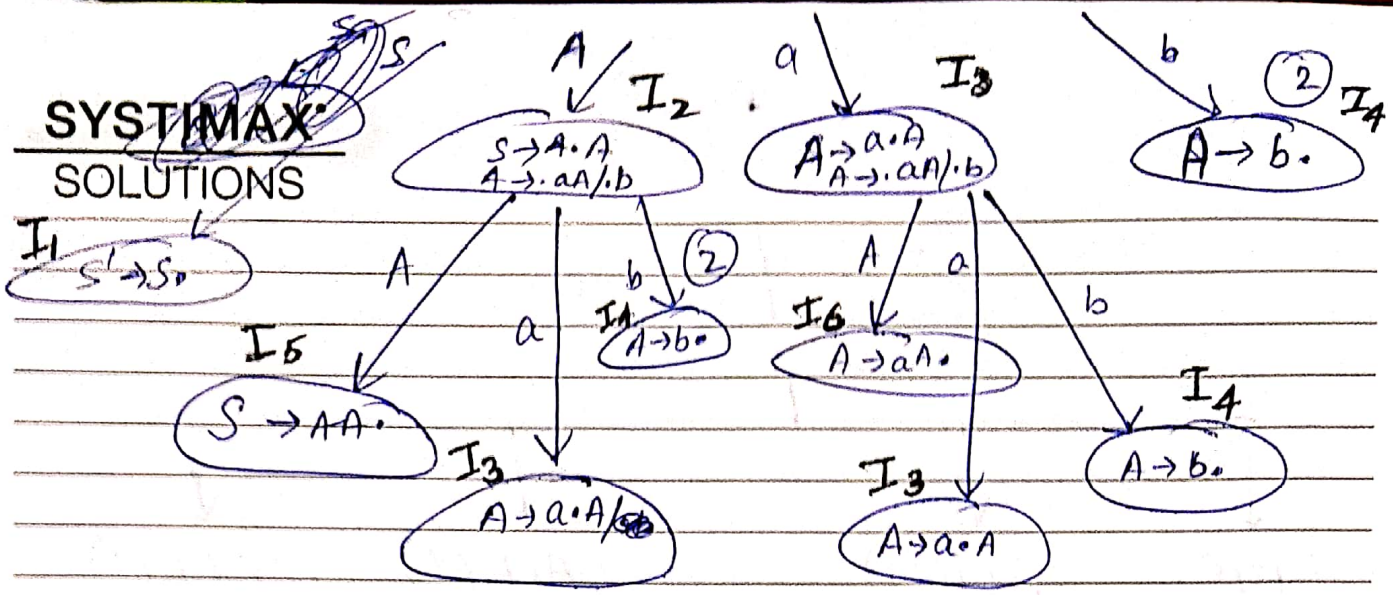
$S \to \cdot AA$ (we haven't seen anything in RHS)
$S \to A \cdot A$ (we have only seen A at RHS)
$S \to AA \cdot$ (we have seen everything in RHS)

Now, we use items for everything.

When . goes to the most right of variable like (2) & (3), they are called final items. And are not moved further



SYSTIMAX SOLUTIONS



	(Terminals) Action			(Variables) Goto	
	a	b	$	A	S
I_0	S_3 (shift move)	S_4		I_2	I_1
I_1			accept		
I_2	S_3	S_4		I_5	
I_3	S_3	S_4		I_6	
I_4					
I_5					
I_6					

For transition in Variable fill Goto.

For transition in Terminals fill Action.

$A \to I_1$ is the augmented production so we accept it.

$I_4, I_5, I_6 \Rightarrow$ (Final items)

Find its corresponding states, like I_4 comes from $A \to b$.

I_5 comes from $S \to AA$

I_6 comes from $A \to aA$

$S \to AA$ (1)

$A \to aA/b$ (2) (3)

	reduced move		
so I_4	r_3	r_3	r_3
I_5	r_1	r_1	r_1
I_6	r_2	r_2	r_2

Now LR(0) parsing table has been made, so now we can use it check for parsing a string.

Eg Check string : aabb , for $S \rightarrow AA$
$A \rightarrow aA / b$.

Ans

aabb$

a a b b $
↑

0	a

I_0 th state.

So, 0 on a is S_3 that is shift 3

0	a	3

a a b b $
↑

0	a	3

So, now 3 on a is S_3 that is shift 3

0	a	3	a	3

a a b b $
↑

0	a	3	a	3

So, now 3 on b is S_4 that is shift 4

0	a	3	a	3	b	4

SYSTIMAX®
SOLUTIONS

a a b b $

0	a	3	a	3	b	4

Now, 4 on b is r③ that is reduce production ③ (A → b)
so whenever we see a reduce, we don't reduce the current pointer but one pointer before.

Reducing Move

Now, A → b
See the number of symbols in RHS, i.e. 1 and reduce 2x symbols from stack.
and add the LHS in the stack.

0	a	3	a	3	b	4	A

Now, corresponding to the previous state, 3 (here) see on which state does A go on 3, that is 6.

0	a	3	a	3	b	4	A	6

A ? reduction of b.

a a b b $
↑

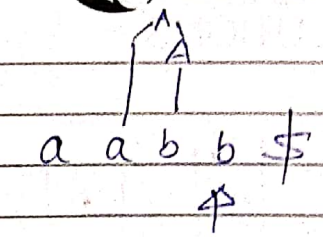
pointer doesn't increase in reduce still remains same

0	a	3	a	3	b	4	A	6

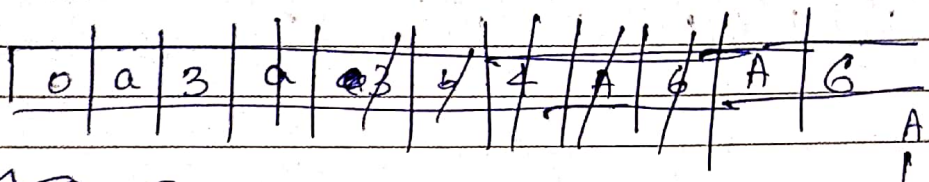
Now, we see b on 6, is reduce ② or r②
so take production ② (A → aA).

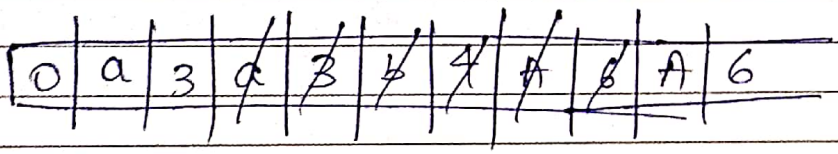
Reducing Move

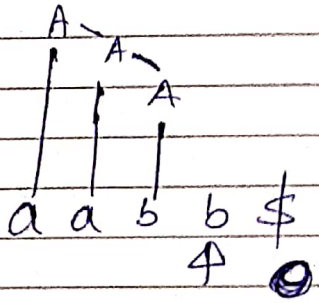
RHS of A → aA contains two symbols hence 4x symbols are popped & push LHS.



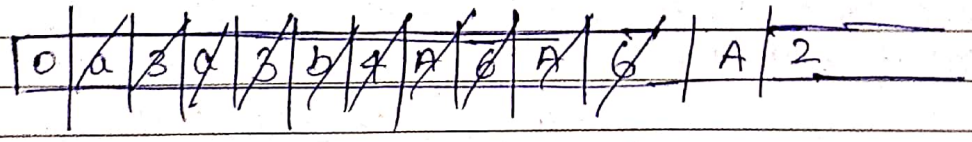
now corresponding to previous state, 3 (here) see on which state does A go on 3, that is 6.



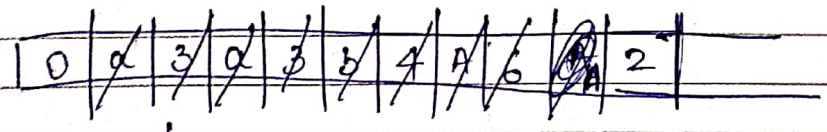




Now, b on 6 is r_2 that is reduce (2), see production (2) (A → aA) and reduce it.
RHS of (A → aA) has two symbols, hence 4x popped.

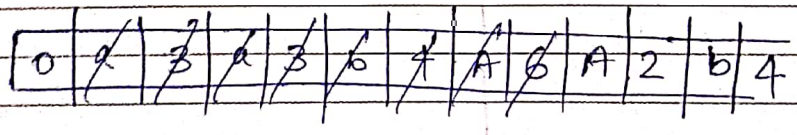


a a b b $



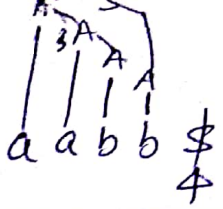
Now, b on 2 gives s_4 that b shift 4

a a b b $



Now, $ on 4 give r_3 that reduce (3), A → b.

SYSTIMAX®
SOLUTIONS

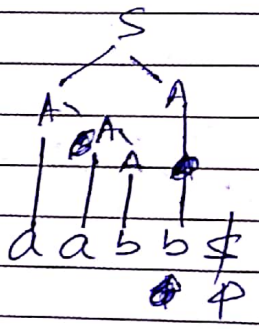


0	a	3	a	3	b	4	A	6	A	2	b	4	A	5

Now, we see 5 on $, r_1 S → AA, 4x popped.

0	a	3	a	3	A	4	A	6	A	2	b	4	S	1

so 1 on $ is accepted ✓



SLR(1)

Shift moves and Goto remain but while reduction, we fill it with the follow( ) of the production

So, for I_4 ($A \to b.$, State 4), hence find follow of LHS (here) | state. and fill it here.

	a	b	$
I_4	r_3	r_3	r_3
I_5			r_1
I_6	r_2	r_2	r_2

follow(A) → {a, b, $}

But of I_5 ($S \to AA.$, State 1), so follow(S) = {$}, hence

I_6 ($A \to aA.$), so follow(A) = {a, b, $}, hence

So, complete table is

	Action			Goto	
	a	b	$	A	S
I_0	S_3	S_4		2	1
I_1			accept		
I_2	S_2	S_4		5	
I_3	S_3	S_4		6	
I_4	r_3	r_2	r_2		
I_5			r_1		
I_6	r_1	r_1	r_1		

& then parsing continues same LR(0)

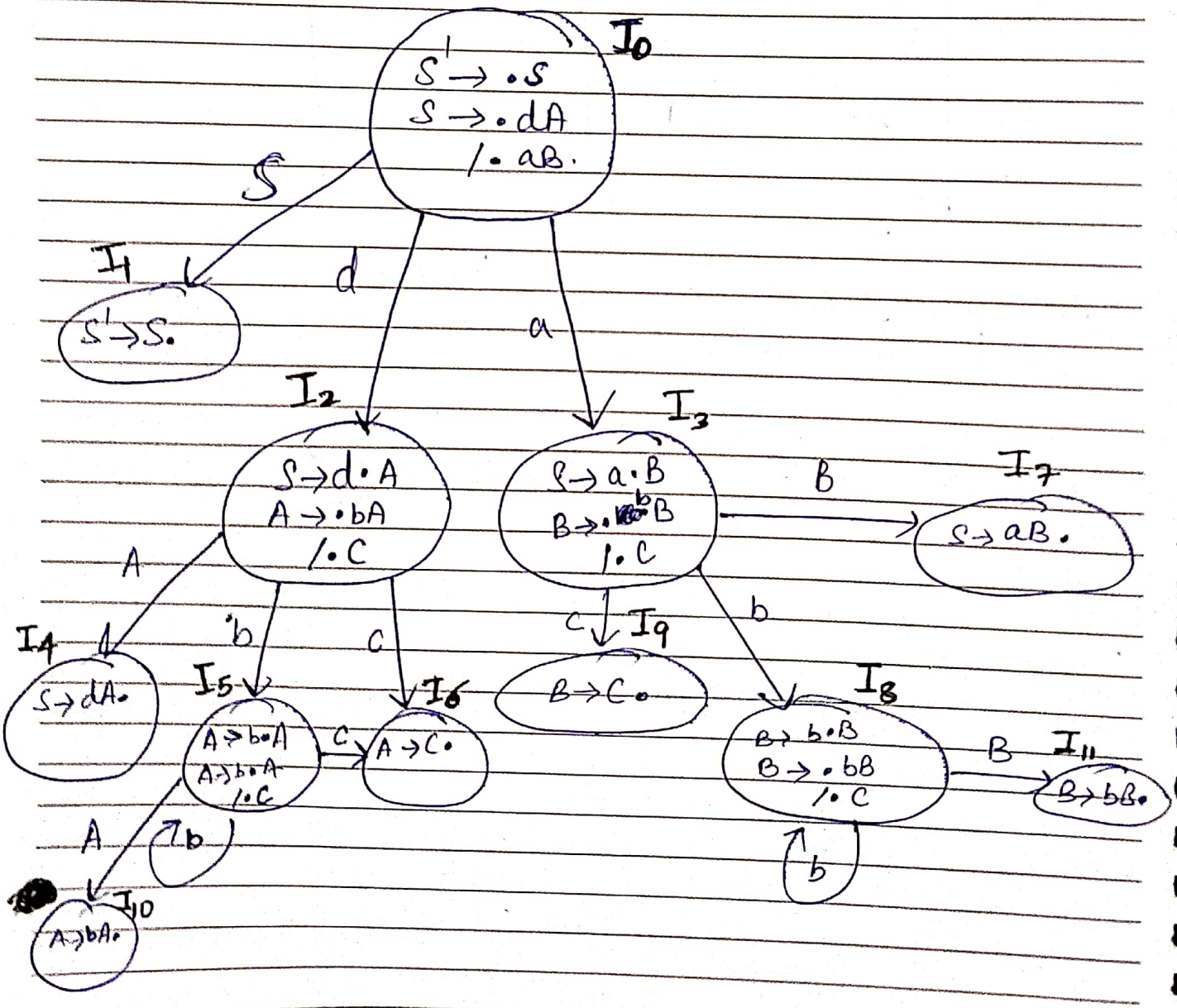
** Sometimes, shifting and reducing can occur in same row known as shift-reducing conflict.

Eg

$S \rightarrow dA$
$\quad / aB$

$A \rightarrow bA / c$
$B \rightarrow bB / c$

$S' \rightarrow S$ (1)
$S \rightarrow dA$ (2)
$\quad / aB$ (3)
$A \rightarrow bA / c$ (5)
$B \rightarrow bB / c$ (7)



	Action					Goto		
	a	b	c	d	$	A	B	S
I_0	S_3		S_2					I_1
I_1					accept			
I_2		S_5	S_6			I_4		
I_3		S_8	S_9				I_7	
I_4	r_2	r_2	r_2	r_2	r_2			
I_5		S_5	S_6			I_{10}		
I_6								
I_7								
I_8		S_8					I_{11}	
I_9								
I_{10}								
I_{11}								

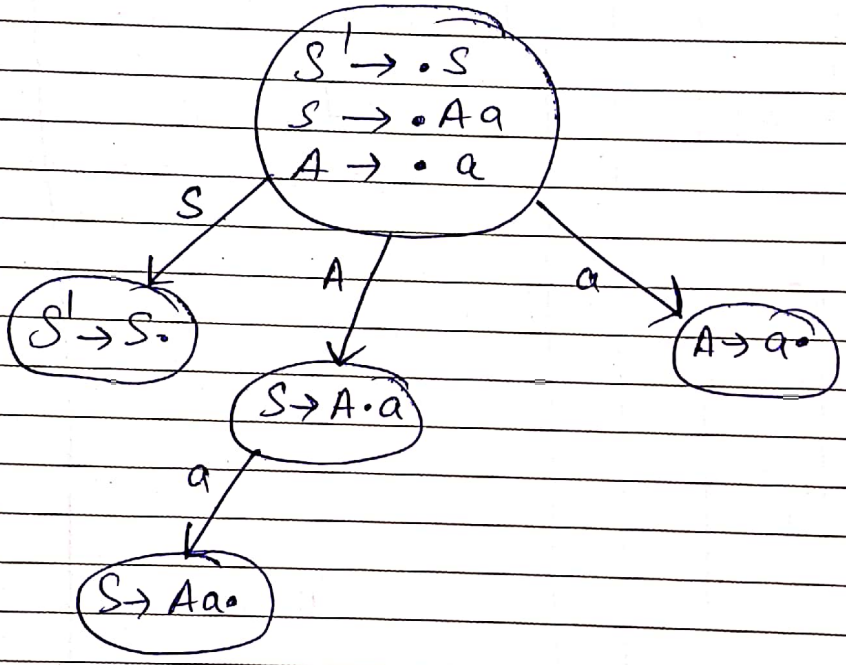
SYSTIMAX®
SOLUTIONS

L
S
C
a

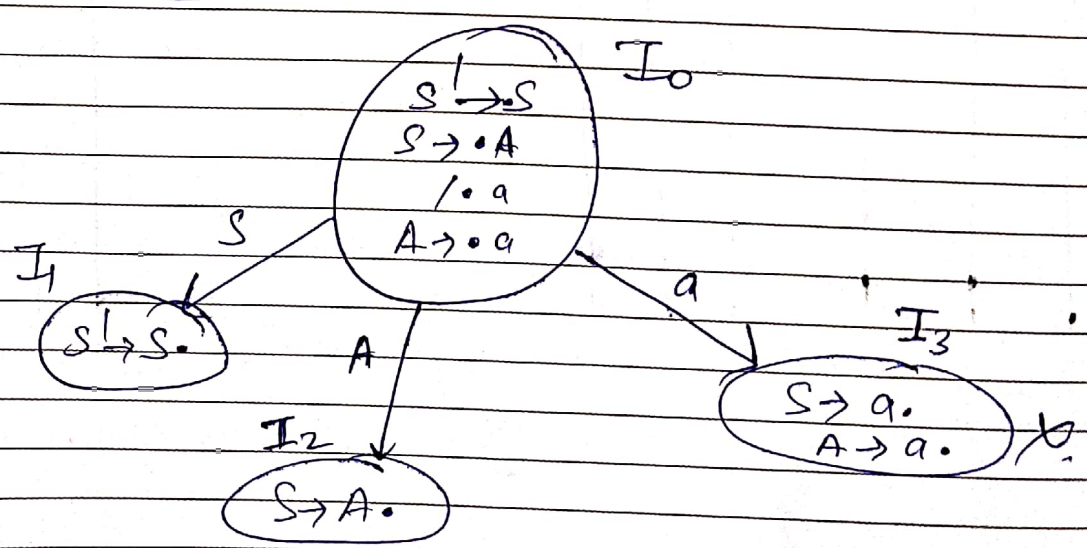
Q.

$S \to Aa$
$A \to a$

$S' \to S$ (1)
$S \to Aa$ (2)
$A \to a$ (3)



Q. $S \to A/a$
$A \to a$



LR(1) items ⇒ LR(0) + lookahead

CommScope

$S \to AA$
$A \to aA/b$

$S' \to .S$
$S \to .AA$
$A \to .aA/b$
} LR(0) item.

convert into LR(1) by adding lookahead.

lookahead.

$S' \to .S, \$$ → At the first of all the symbols present after S, (here) and if no, symbol present then straightaway add the previous lookahead.

$S \to .AA, \$$

$A \to .aA, a/b$
$/.b, a/b.$

→ Here, next symbol is $A\$$, hence FIRST($A\$$) is (a, b)

* lookaheads dont change

I_0

SO,
① $S' \to .S, \$$
② $S \to .AA, \$$
③ $A \to .aA, a/b$
$/.b, a/b.$

S → I_1: $S' \to S., \$$

A → I_2: $S \to A.A, \$$
$A \to .aA, \$$
$/.b, \$$

A → I_5: $S \to AA., \$$

a → I_3: $A \to a.A, a/b$
$A \to .aA, a/b$
$/.b, a/b$

a → I_6: $A \to a.A, \$$
$A \to .aA, \$$
$/.b, \$$

a (self-loop on I_6)

b → I_4: $A \to b., a/b$

A → I_8: $A \to aA., a/b$

A → I_9: $A \to aA., \$$

b → I_4: $A \to b., \$$

b → I_7: $A \to b., \$$

a (self-loop on I_3)

b → I_4: $A \to b., a/b$

www.commscope.com

SYSTIMAX® SOLUTIONS

CLR(1) Parsing Table

	Action			Goto	
	a	b	$	S	A
I_0	S_3	S_4		I_1	I_2
I_1					
I_2	S_6	S_7			I_5
I_3	S_3	S_4			I_8
I_4	r_3	r_3			
I_5			r_1		
I_6	S_6	S_7			I_9
I_7			r_3		
I_8	r_2	r_2			
I_9			r_2		

* Shifting of I_n remain same as before
* Reducing moves are used to find in the same way as before but placed in the lookahead columns.
* New states $[I_3, I_6]$, $[I_4, I_7]$, $[I_8, I_9]$ are same but have different lookaheads, so merging is possible.

By doing this $[I_3, I_6]$ ⇓ I_{36}, $[I_4, I_7]$ ⇓ I_{47}, $[I_8, I_9]$ ⇓ I_{89}

When merging of these states is done we get,

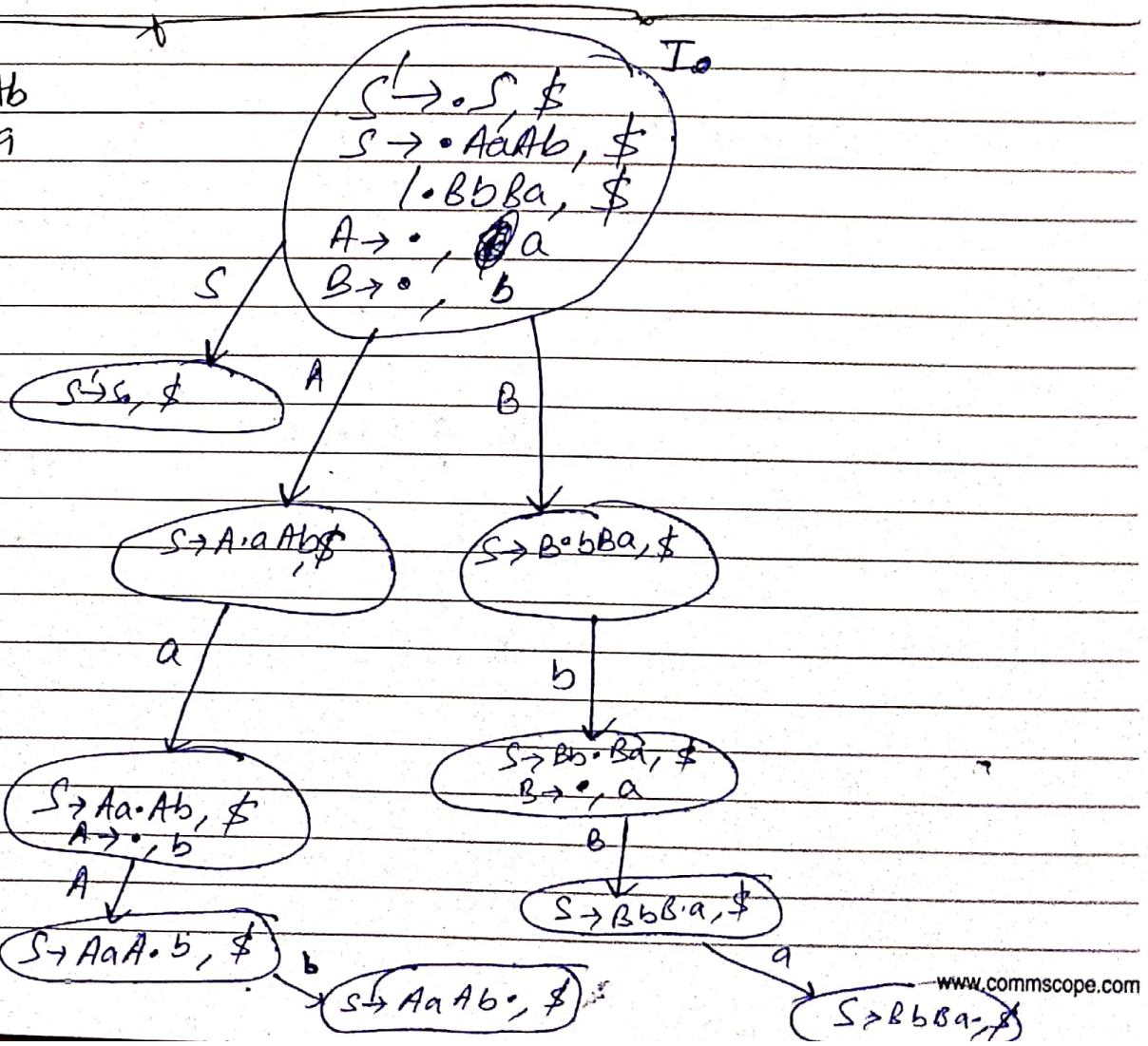
LALR(1) Parsing table.

LALR(1) Parsing table.

	Action			Goto	
	a	b	$	S	A
I_0	S_{36}	S_{47}		I_1	I_2
I_1					
I_2	S_{36}	S_{47}			I_5
I_{36}	S_{36}	S_{47}			I_{89}
I_{47}	r_3	r_3	r_3		
I_5			r_1		
I_{89}	r_2	r_2	r_2		

Eg

$S \to AaAb$
$\quad | BbBa$
$A \to \epsilon$
$B \to \epsilon$

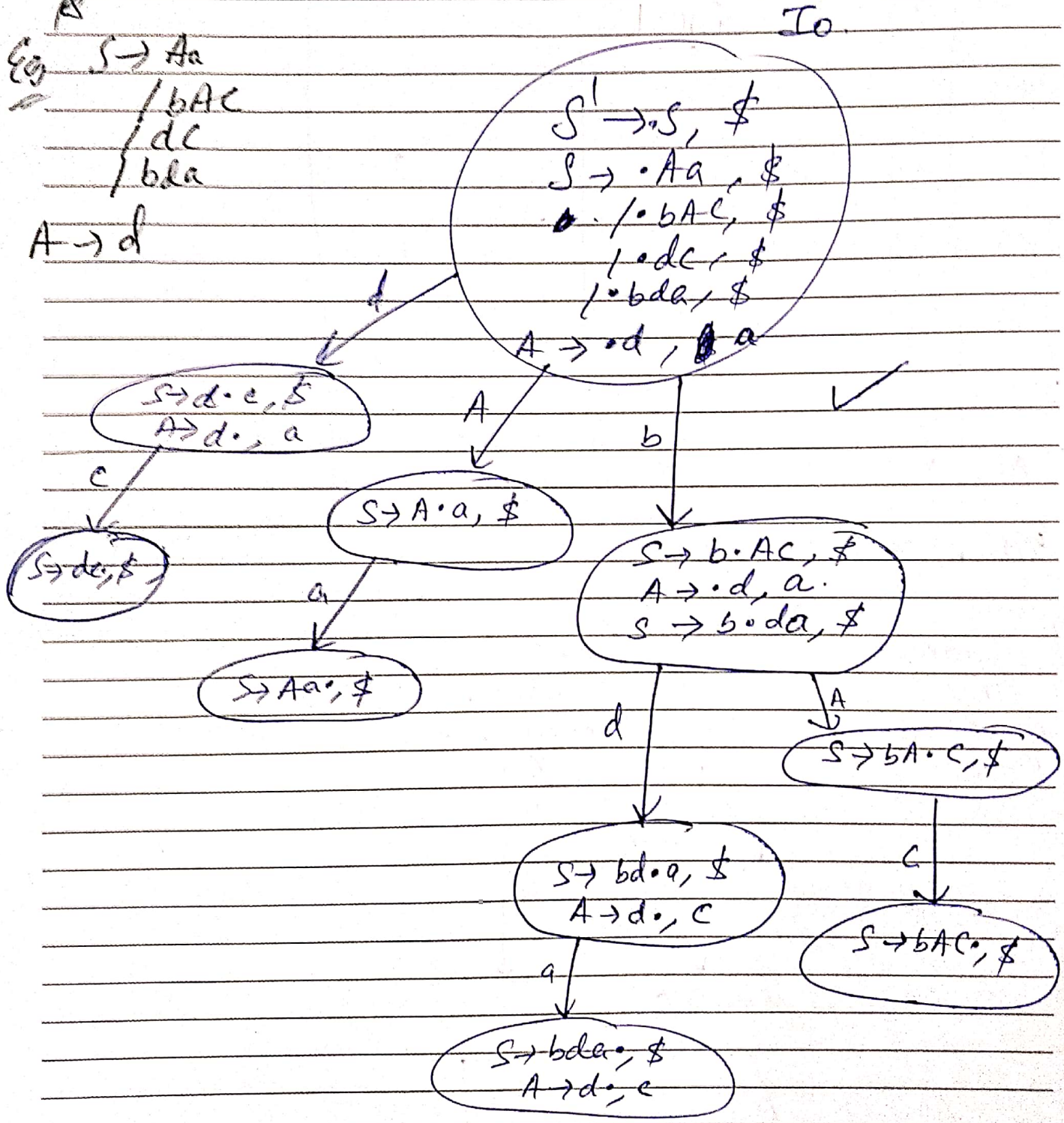


SYSTIMAX®
SOLUTIONS

Eg

$S \to Aa$
$\quad / bAC$
$\quad / dC$
$\quad / bda$

$A \to d$



Cross Compiler

A cross compiler is a compiler capable of running or creating executable code for a platform other than the one on which compiler is running.

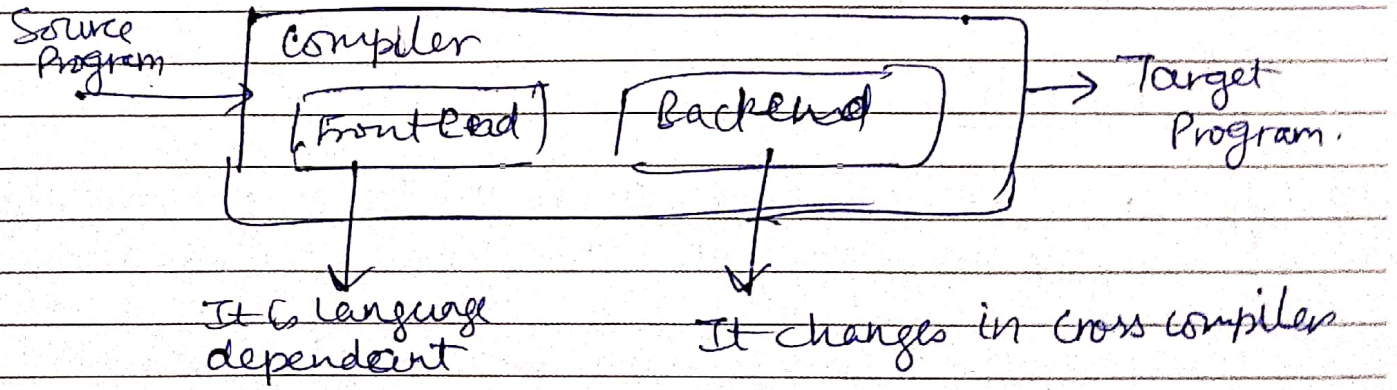
creating more and more compiler for the same source language but for different machines is called, retargetting, & retargetable compiler is called cross compiler

C++

GCC → old m/c (8 bit memory)

→ Intel 7c (64 bit memory)

C++ → (cross compiler) (Only Backend changes).

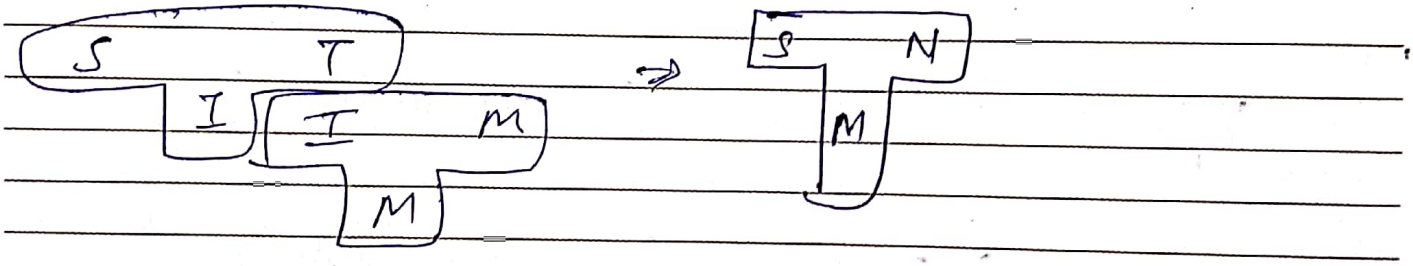


SYSTIMAX®
SOLUTIONS

Bootstraping

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, know as Bootstraping.

T Diagram



The process illustrated by T-diagram is called Bootstrapping.

WORKING

1) An i/p file called lex.L is written in the lex language which describes the lexical analyser to be generated

2) The lex compiler then transforms lex.L to a C program, in a file that is always named lex.yy.c

3) The later file is compiled by th C compiler into a file called a.out as always.

4) The C-compiler output is a working lexical analyser that can take a stream of i/p chars & produce a stream of tokens.

[Training]

⇒ It is a tool which generate lexical analyzer.

LEX (xyz.l)

→ lexical analyser is the first phase of compiler which takes input as source code and generate o/p as tokens.

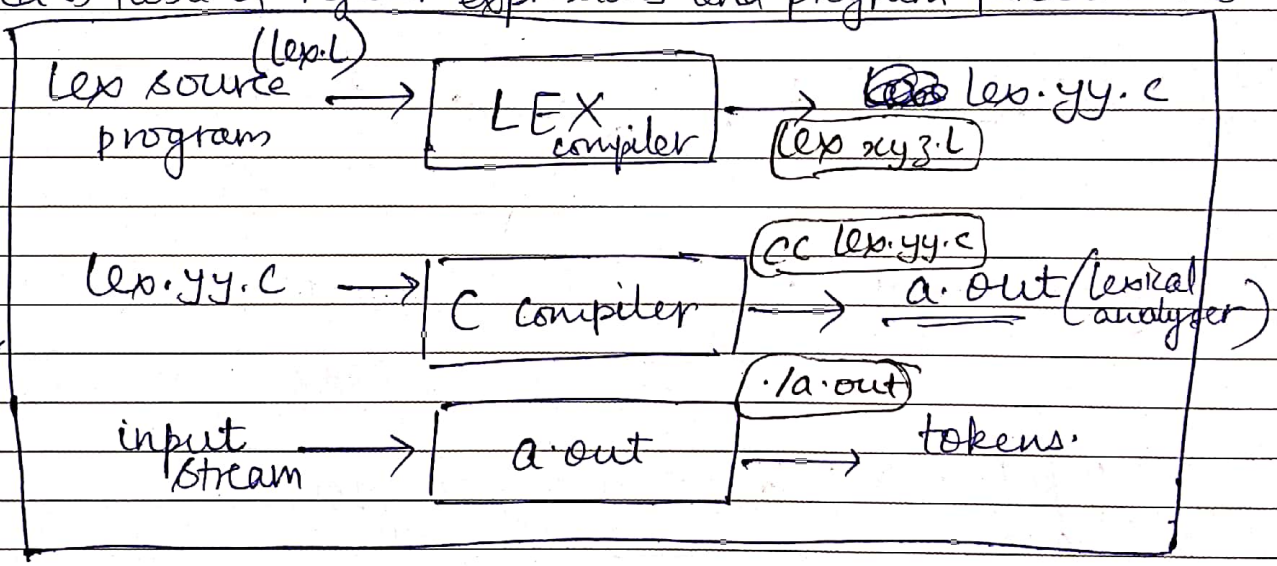
⇒ Break the input stream onto more usable tokens.

a = b + c * d ↓

id = id + id * id.

→ Tokens are terminals of a language, (+, *, =).

→ It is table of regular expressions and program fundamentals



lex source is seperated into 3 sections by %%. delimiters.

```
{definations}
%%
{transition rules}
%%
{user subroutines}
```

Eg

```
[Defination]
digit [0-9]
letter [a-zA-Z]
--%%--[Translation]
{letter}({letter}|{digit})*
printf("id: %s\n", yytext);
\n
printf("new line\n");
%%[Auxillary function]
main(){
    yylex();
}
```

```
{letter}({letter}|{digit}).
printf("id: %s\n", yytext);
```

www.commscope.com

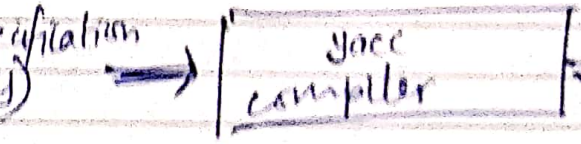
SYSTIMAX SOLUTIONS

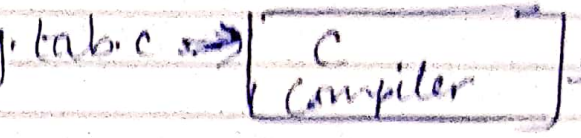
YACC [Syntax analyser or Parser]

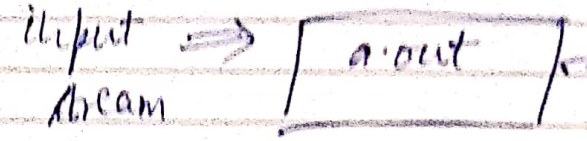
→ YACC stands for "yet another compiler compiler"

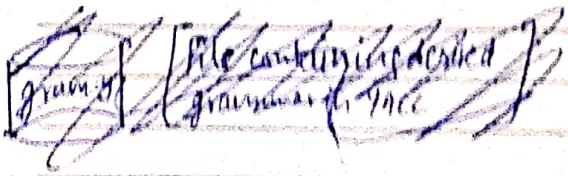
→ It is a tool which generates LALR parser

→ Syntax analyser (parser) is the second phase of compiler which takes input as token and generates syntax tree.









YACC is also divided into 3 sections by %% delimeter function:

Definitions
%% %% %%

Rules
%% %% %%

Supplementary Code
%% %% %%

CommScope

Definition

Where we configure various parser features such as defining token codes, establishing operator precedence and associativity and setting up variables used to communicate between the scanner and the parser.

Rules

The required productions section is where we specify the grammar rule.

Supplementry Code Section

It is used for ordinary C code that we want copied verbatim to the generated C file, declarations are copied to the top of the file, user subroutines to the bottom.

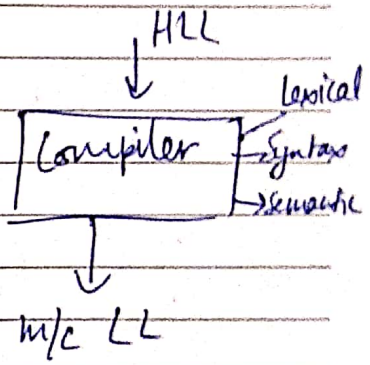
SYNTAX DIRECTED TRANSLATION

Grammar + Semantic rules = SDT.

SDT for evaluation of expression :-

Top Down
Left to Right

$E \to E+T$	$\{E.value = E.value + T.value\}$.
$/\ T$	$\{E.value = T.value\}$
$T \to T*F$	$\{T.value = T.value * F.value\}$.
$/\ F$	$\{T.value = F.value\}$.
$F \to num$	$\{F.value = num.Lvalue$



Q. 2+3*4.

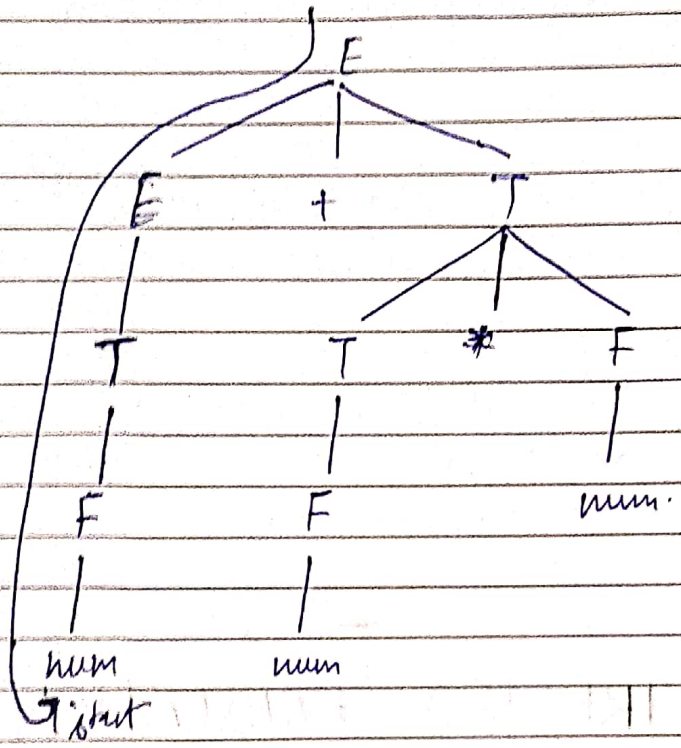
(Make Parse Tree)

Contd. After 2 pages.

SYSTIMAX®
SOLUTIONS

* All non-terminal get the attribute.

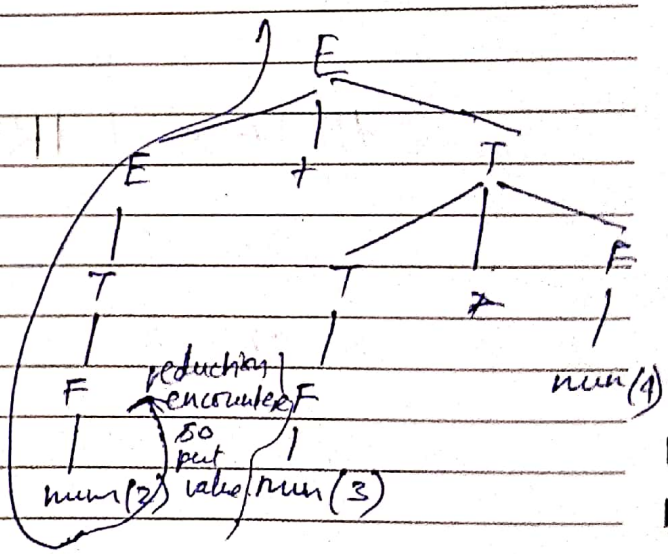
2 + 3 * 4.



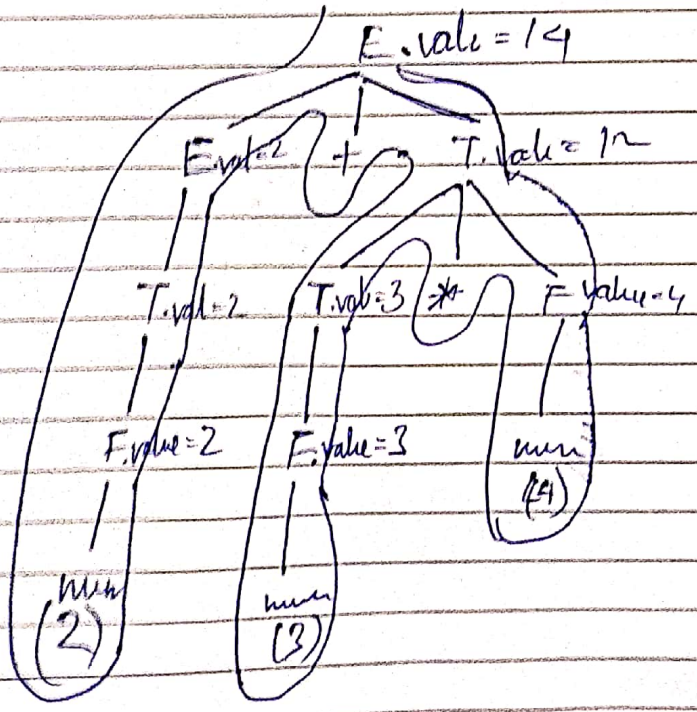
(1) Traverse this tree Top-Down and left-to Right

(2) Whenever you see reduction (T → F) then you put/replace it with values.

(3) Now, for reducing:- Go to production and carry out the action.



• value = attribute



Infix to Postfix Expression

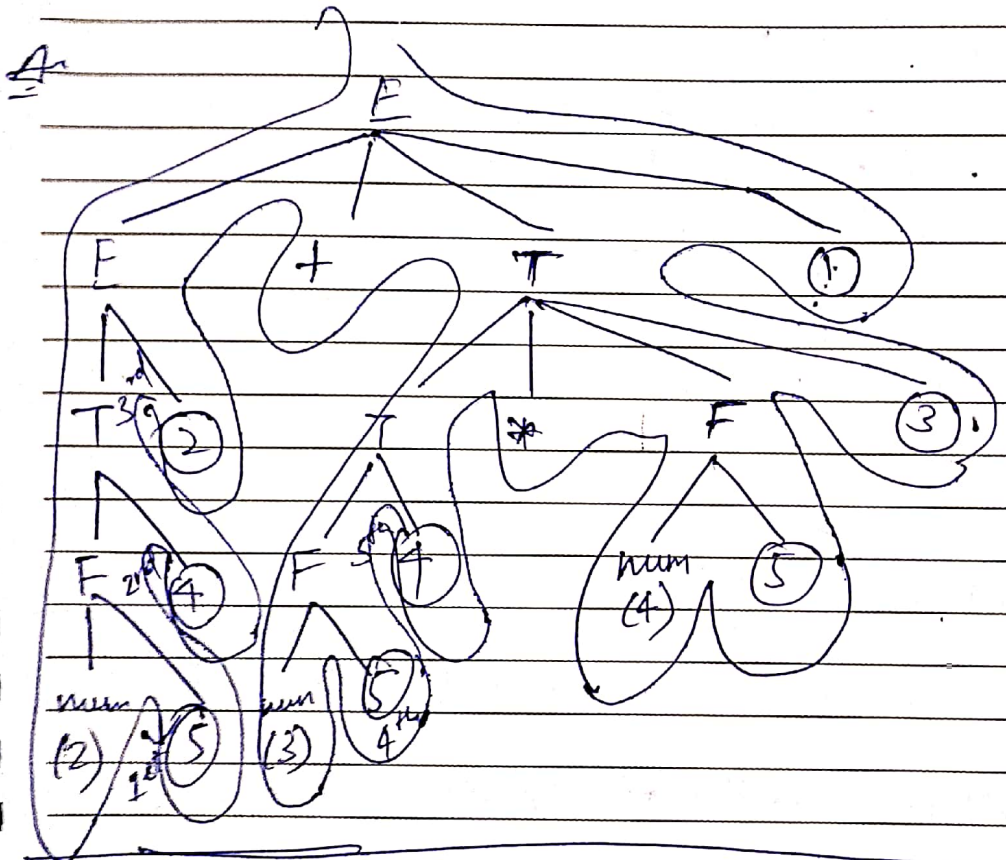
CommScope

2+3 * 4.

$E \to E + T$ / T { } (2) — { printf("+"); } (1) → semantic action

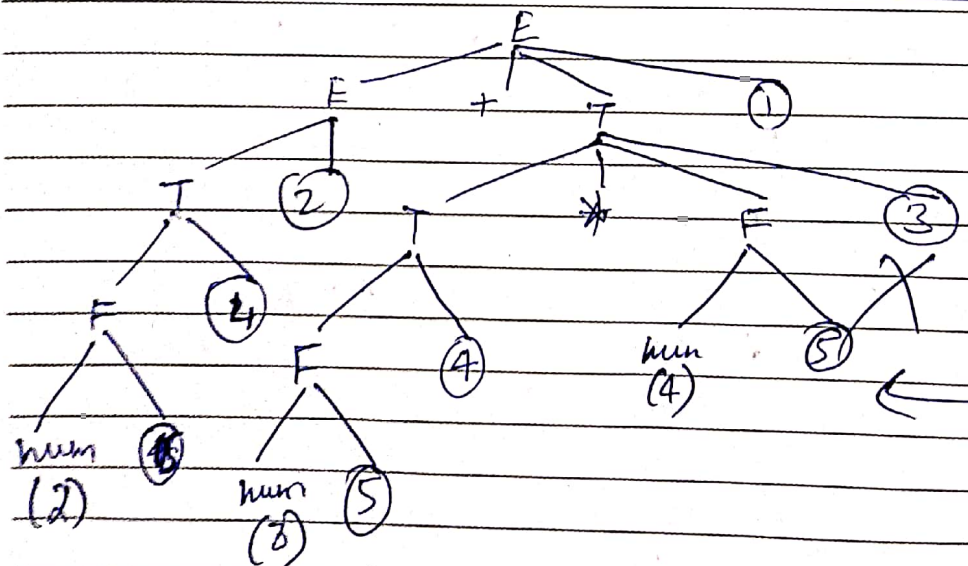
$T \to T * F$ / F { } (4) — { printf("*"); } (3)

$F \to num$ — { printf(num.lval); } (5)



Top-Down
left to Right

Output ⇒ 234*+

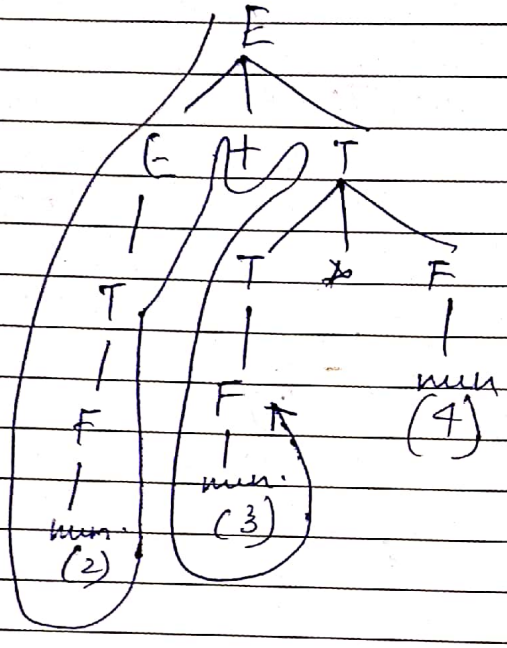


Incase of Button-up parser, they are mainly focused on reduction hence this will not work

SYSTIMAX®
SOLUTIONS

Button-Up try to reduce the terminal to variables, RHS to LHS

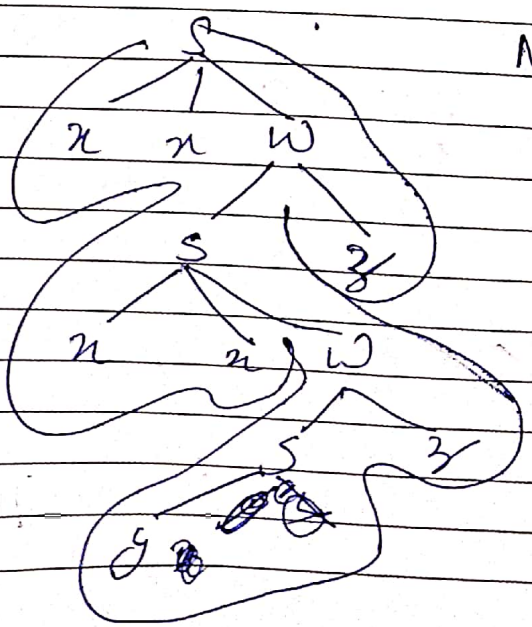
So when ever they reduce → they see production & do the action.



2 3 4 * +

Q. S → x x w {Printf (1);}
/ y {Printf (2);}
W → S z {Print (3);} for the string: x x x y 3 z

So, derive the parse tree w.r.t to the string first.



Button-Up.

Now, apply rules:
See reduction & perform action.

2 3 1 3 1

Q.

$E \to E_1 * T \quad \{E.val = E_1.val * T.val;\}$
$\quad / T \quad \{E.val = T.val;\}$.

$W = 4-2-4*2$

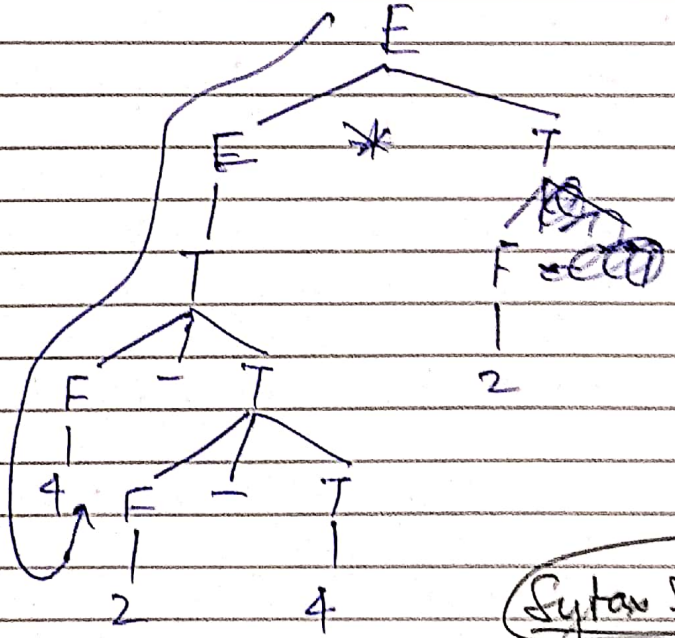
$T \to F - T_1 \quad \{T.val = F.val - T_1.val\}$
$\quad / F \quad \{T.val = F.val;\}$.

$F \to 2 \quad \{F.val = 2;\}$
$\quad / 4 \quad \{F.val = 4;\}$

$* \Rightarrow$ Left Recursive
$- \Rightarrow$ Right Recursive.

Ans. $W = \left(\left(4-\left(2-4\right)^{1}\right)^{2} * \quad 2\right)^{3}$.

Straightforward $\Rightarrow$ $4-(-2) \Rightarrow 4+2 \Rightarrow 6 \Rightarrow 6 * 2 \Rightarrow 12$



(1) Now as (−) minus is lower in the question, hence its priority is more than that of (*) multiplication.
(2) And, $E \to E * T$ is Left-Recursive & $T \to F - T$ is Right Recursive hence, the rightmost (−) will be taken into account first then the (−) & then *.

Syntax Directed Translation Scheme

A SDT is a context free grammar with program fragments embedded with production body. The prog. fragments are called semantic action.

We focus on the use of SDT's to implement two imp classes of SDD's:-

1) The underlying grammar is LR-parsable and the SDD is S-attributed.
2) The underlying grammar is LL parsable and the SDD is L-attributed.

SYSTIMAX SOLUTIONS

Types of Semantic Action performed by S.A.

(1) Scope Resolution = main
{
int a;
int a; → (Same scope cannot have 2 declarations of same variable)
}

(2) Type checking = int a;
a = "hello"; → (LHS is int & RHS is string so not allowed)

(3) Array Bound Checking = int a[5];
a[10]; → (Not possible)

Hence,

* Syntax Directed Translation arises, it is a mechanism to perform check on whether it is meaningful or not.

→ SDT is done by attaching rules to productions in a grammar.

→ Every node of the AST (abstract syntax tree) is attached with certain attribute set.

Eg if E → E+T {E.val = E.val + T.val}.

So for int a = 3 → ⟨Datatype, "int"⟩, ⟨current, "a"⟩ } so we have defined int in 2 attributes.

I → ID/D
D → [0-9] | I.value = I.value * 10 + D / I.value = D.value.

Attribute → Synthesized attribute
→ Inherited attribute

Hence, to do semantic analysis and code generation, we 2 things i.e. (tools)

① Semantic Rules (SDD definition)

(Syntax Directed Definition)

(For postfix translation)	Semantic Rule
$E \to E_1 + T$	$E.val = E_1.val \,\|\|\, T.val \,\|\|\, '+'$

→ Only involves synthesized attributes hence S-attributed

→ More Augmented CFG that specifies the values of attributes by associating semantic rules with grammar production.

② Semantic Action

(Syntax Directed Translation)

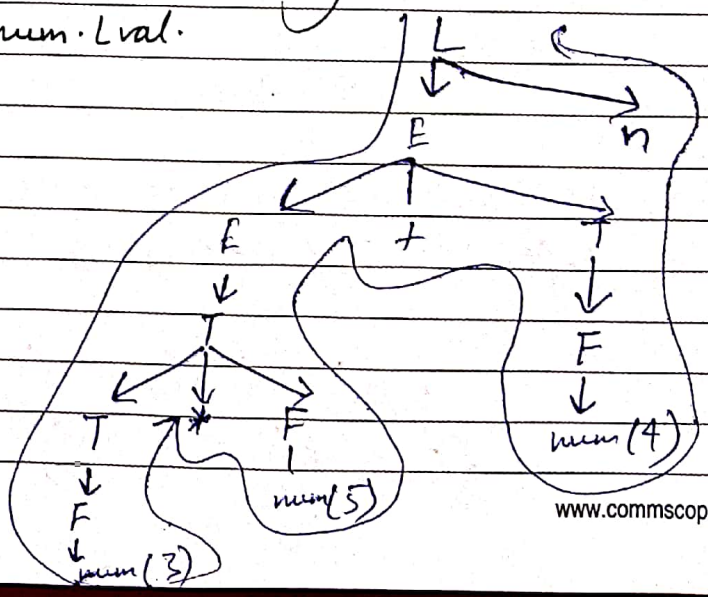
	Semantic Action
$E \to E_1 + T$	{Print '+'}

→ It embeds program fragments called semantic actions with production body.

Eg

	Rules
$L \to E_n$	$L.val = E.val$
$E \to E_1 + T$	$E.val = E_1.val + T.val$
$/T$	$E.val = T.val$
$T \to T_1 * F$	$T.val = T_1.val * F.val$
$/F$	$T.val = F.val$
$F \to (E)$	$F.val = E.val$
$/num$	$F.val = num.Lval$

for $3*5+4n$



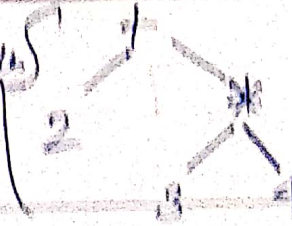
Ex $3*5+4n$.

$15+4n$

$19n$

SYSTIMAX SOLUTIONS

(Recall: Give any No. of examples)



Abstract syntax tree

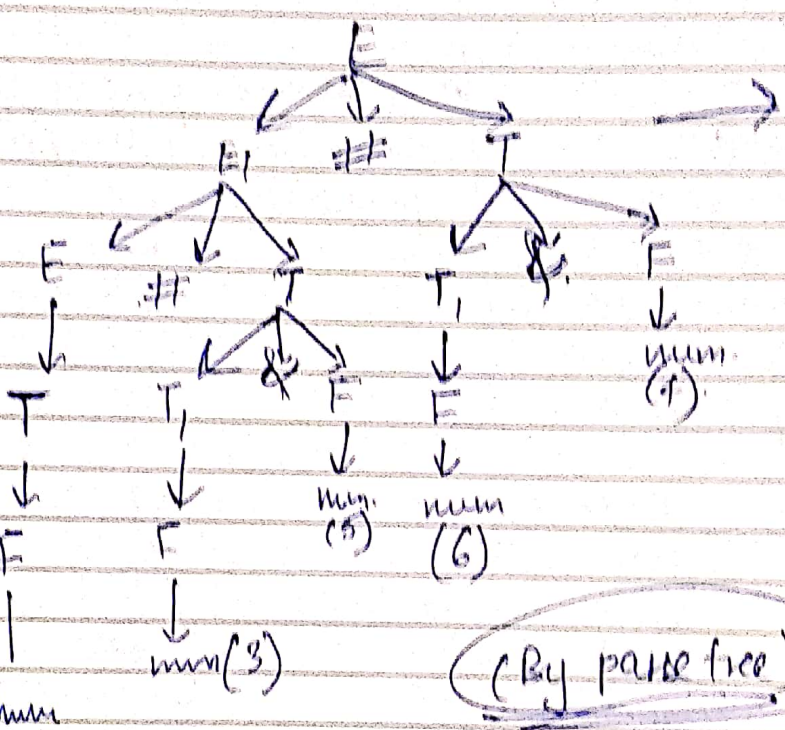
Q.

E → E1 # T { E.val = E1.val * T.val }
/ T { E.val = T.val }.

T → T1 & F { T.val = T1.val + F.val }
/ F { T.val = F.val }

E → num { E.val = num.lval }.

2 # 3 + 5 # 6 & 1.



→ Concrete syntax tree.

(By parse tree).

By Easier Method

represents *
& represents +

So,

2 * 3 + 5 * 6 + 1

(1) Now, & has more precedence to # so it is lower in order
(2) # is left-recursive and & is also left-recursive.

So,

2 * (3 + 5) * (6 + 1)
((2 * 8) * 10)
160.

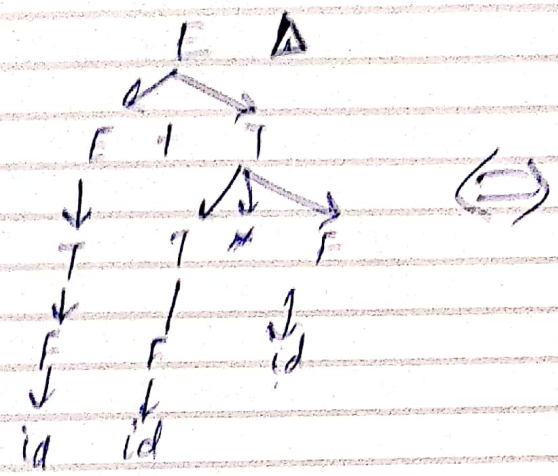
Intermediate Code

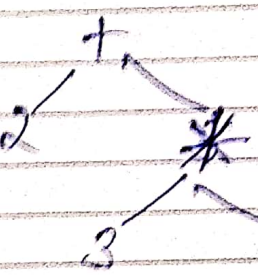
Video - 42 (Revision)

CommScope

Everytime making concrete syntax tree becomes tedious and time taking hence, we can also represent it in the form of syntax tree or AST (Abstract Syntax Tree).

So, we discuss SDT to build syntax tree:





There are various ways in which we can represent intermediate code of one of popular intermediate code is syntax tree.

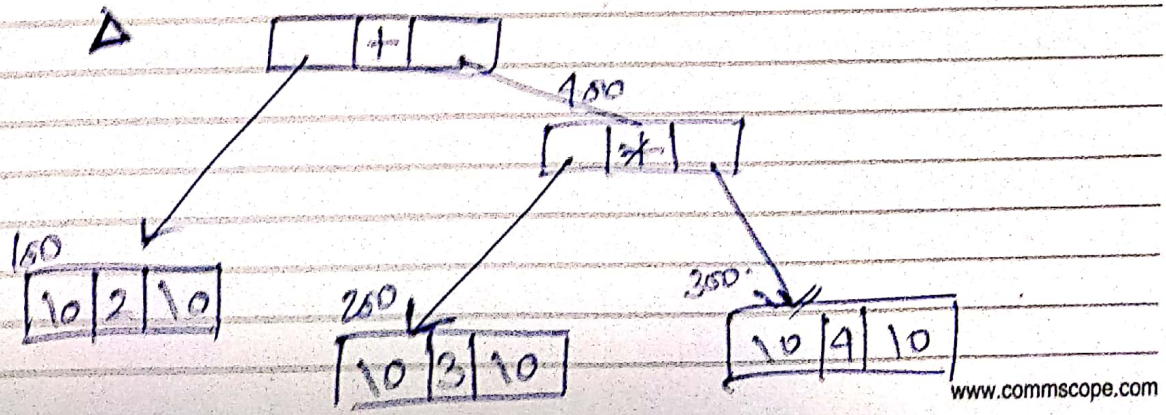
So,

E → E1 + T { E.nptr = mknode(E1.nptr, '+', T.nptr); }
/ T { E.nptr = T.nptr; }

T → T1 * F { T.nptr = mknode(T1.nptr, '*', F.nptr); }
/ F { T.nptr = F.nptr; }

F → id { F.nptr = mknode(null, id.name, null); }

nptr → node pointer
mknode → function that creates node with the parameter attributes



SYSTIMAX®
SOLUTIONS

dval ⇒ decimal value

$N \to L$	$\{N.dval = L.dval\}$
$L \to L_1 B$	$\{L.dval = L_1.dval * 2 + B.dval\}$
$/ B$	$\{L.dval = B.dval\}$.
$B \to 0$	$\{B.dval = 0\}$
$/ 1$	$\{B.dval = 1\}$.

This is basically for converting binary number to decimal no.

For 11.01 (cases)

$N \to L_1 . L_2$ $\{N.dval = L_1.dval + \frac{L_2.dval}{2^{L_2.\epsilon}}\}$

$L \to L_1 B$ $\{L.C = L_1.C + B.C;\ L.dval = L.dval * 2 + B.dval\}$

$/ B$ $\{L.C = B.C;\ L.dval = B.dval;$

$B \to 0$ $\{B.C = 1;\ B.dval 20\}$

$/ 1$ $\{B.C = 1;\ B.dval = 1\}$

SDT to generate Three Address Code → Intermediate Representation

$S \to id = E$ $\{gen(id.name = E.place);\}$

$E \to E_1 + T$ $\{E.place = newTemp(); gen(E.place = E_1.place + T.place);\}$

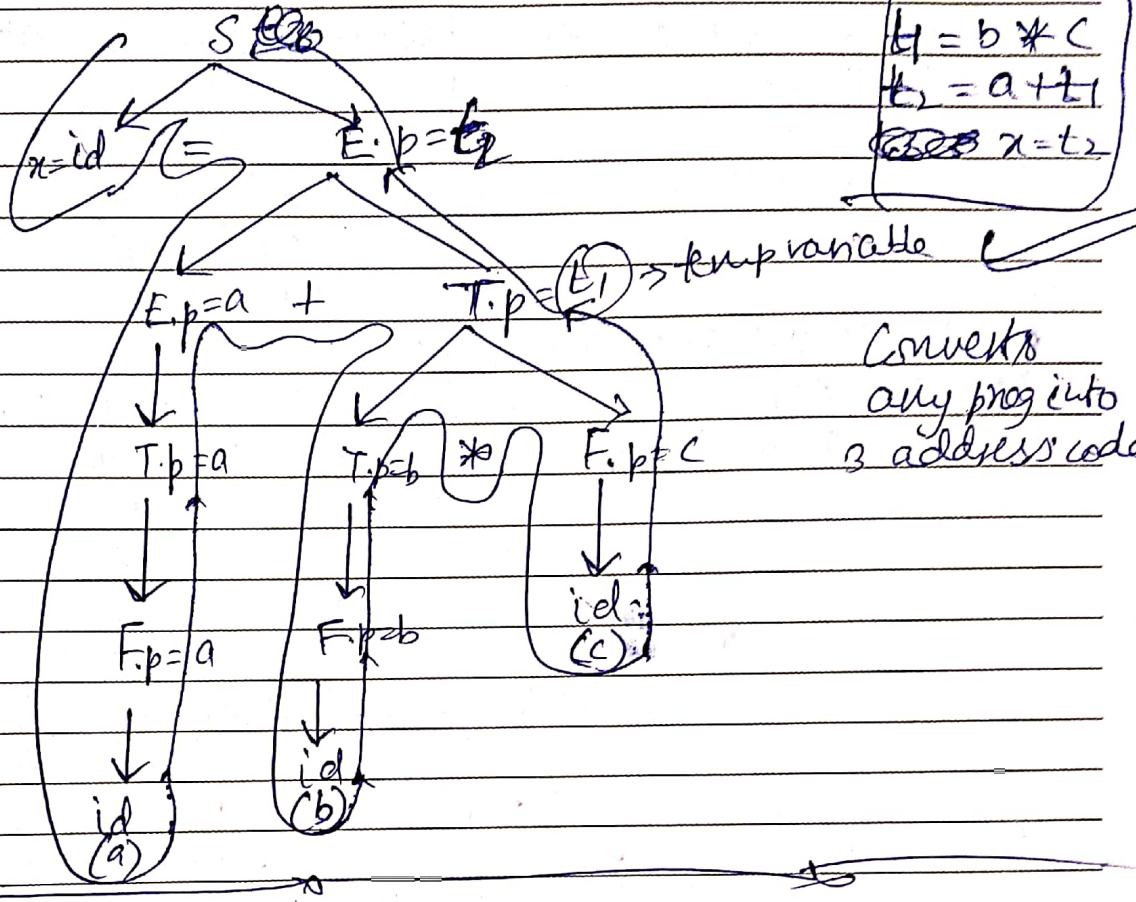
$/ T$ $\{F.place = T.place;\}$

$T \to T * F$ $\{T.place = newTemp(); gen(T.place = T_1.place * F.place);\}$ → Temp variable

$/ F$ $\{T.place = F.place;\}$

$F \to id$ $\{F.place = id.name;\}$.

So, 3 address code of $x = a + b * c$



Imp

* $A \to BCD$ & like $A.S = f(B.S, C.S, D.S)$ (attribute) so, if Parent is taking values from its children, then its synthesized attributes.

Eg ⇒ $A.S = B.S, C.S, D.S$

* $A \to BCD$ & $C.S = A.S$ so if children is taking values from its parent or siblings, then its inherited attributes.

Eg ⇒ $C.S = A.S, B.S, D.S$

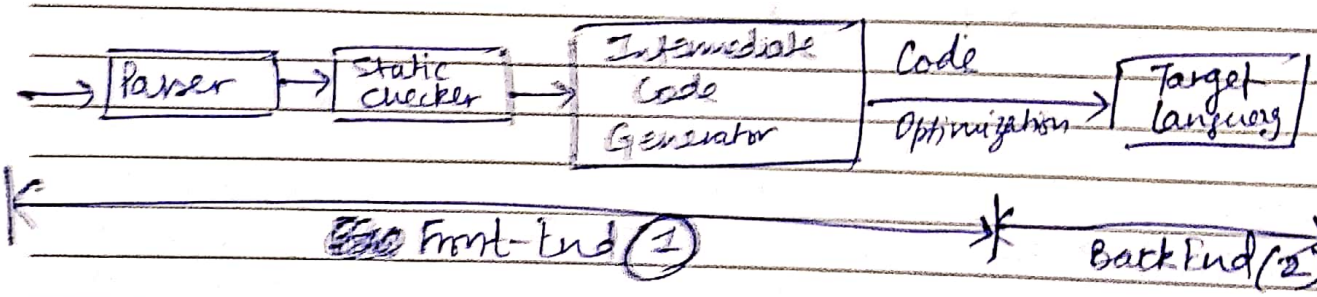
SDT

S-attributed	L-attributed
1) Uses only synthesized attributes	2) Uses both inherited and synthesized. Each inherited attribute is restricted to inherit either parent or left siblings only
Eg $A \to XYZ$ {A.S = X.S; Y.S; Z.S;}	Eg $A \to XYZ$ {Y.S = A.S; X.S; Z.S;}

SYSTIMAX SOLUTIONS

Less Restricted	More Free
2) Semantic actions are placed at right end of production.	2) Semantic actions can be placed anywhere in RHS.
Eg $A \to XYZ \{\}$	Eg $A \to \{\} B C$ / $D \{\} E$ / $F G \{\}$
3) Attribute are evaluated during BUP (Bottom-up parsing)	3) Attributes are evaluated during depth-first & left-to-right.

Intermediate Code Generation

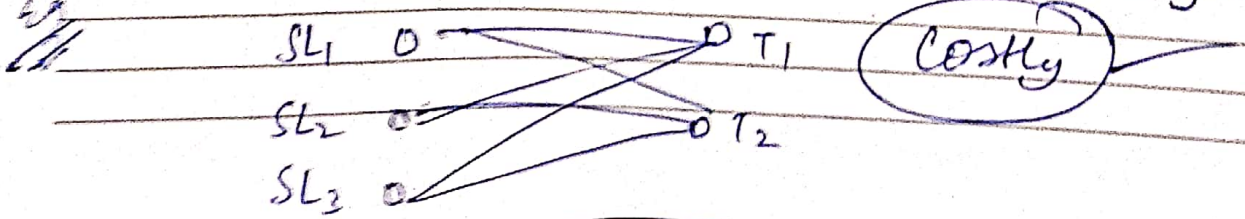


→ A language b/w the source and target

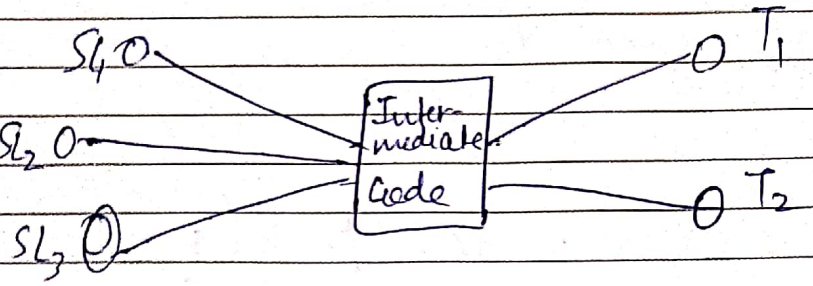
⇒ Intermediate language ⇒ High level language Assembly.

(SL) Source language (1) —Intermediate Code→ Target language (2)

So, basically source-language and target language can be different and hence generating target-language as per every source language is very costly.



Eg



So, this intermediate is expressed via Three address code.

3 Address Code

In 3 address code, at most three addresses are used to represent any statement

General form

$$a = b \text{ operator } c$$

a, b & c are operands that can be names, constants & operator is the operator itself (op)

Three Address Instructions Forms.

1. Assignment statements ⇒ $x = y$ op z (3 operands)
   $x = op\ y$ (2 ")
2. Copy statement ⇒ $x = y$
3. Conditional Jump ⇒ if x relop y goto L (3 operands)

relational operator

so if $x > y$ goto L
if $x < y$ goto L.

SYSTIMAX®
SOLUTIONS

(4) Unconditional jump: goto L

(5) Procedure Calls: param x call p.
return y

```
p(x)
{
  ...
  return(y)
}
```

Eg
a = b+c+d (operators are + and =)

a = ((b+c)+d)

+ has higher precedence than = and is left associative.

$t_1 = b+c$
$t_2 = t_1 + d$
$a = t_2$ ✓

Eg
(−)(a*b) (+)(c+d) (−)(a+b+c+d)
T_1 T_2 T_3.

Binary Operator

Unary Operator

And Unary operator have higher precedence

so, $t_1 = a*b$, $t_2 = (-)t_1$ } T_1

$t_3 = c+d$, $t_4 = t_2 + t_3$ } T_2

$t_5 = a+b$, $t_6 = t_3 + t_5$ } T_3.

$t_7 = t_4 - t_6$

Eg. if A < B
then 1
else 0

Ans (1) if (A < B) goto 4
(2) $T_1 = 0$
(3) goto (5)
(4) $T_1 = 1$
(5)

Eg. if a<b and c<d then t=1 else t=0

(1) if (a<b) goto (3)
(2) goto (4)
(3) if (c<d) goto (6)
(4) t=0
(5) goto (7)
(6) t=1
(7)

SYSTIMAX SOLUTIONS

Implementation of 3 address Code

Three address code can be implemented as a record with the address fields. There are 3 representations used for three address code:-

(1) Quadruples (2) Triples (3) Indirect-Triples.

(1) Quadruples

Each instruction is divided into 4 fields:-

op (operator), arg1 (argument1), arg2 (argument2), result (result)

→ arg1 & arg2 are the two operands used.
→ result is used to store a result of an exp.

Eg Translate the Expression into Quadruple
a + b * c / e ↑ f + b * a

(1) Always first generate 3 address code

(0) $T_1 = c \uparrow f$
(1) $T_2 = b * c$
(2) $T_3 = T_2 / T_1$
(3) $T_4 = b * a$
(4) $T_5 = a + T_3$
(5) $T_6 = T_5 + T_4$

location	op	arg1	arg2	Result
(0)	↑	c	f	T_1
(1)	*	b	c	T_2
(2)	/	T_2	T_1	T_3
(3)	*	b	a	T_4
(4)	+	a	T_3	T_5
(5)	+	T_5	T_4	T_6

	op	arg1	arg2	result
x = op y	op	y		x
param t1	param	t1		
conditional or unconditional				

(2) Triples

The use of temp. variables is avoided and instead references to instructions are made. Each instruction is divided into 3 parts

op (operator), arg 1 (argument), arg 2 (argument)

Eg. Same as b4

	op	arg	arg	Result
(0)	↑	e		
(1)				
(2)				
(3)				
(4)				
(5)				

location	op	arg 1	arg 2	Result
(0)	↑	e	f	(T_1)
(1)	*	b	c	(T_2)
(2)	/	T_2	T_1	(T_3)
(3)	*	b	a	(T_4)
(4)	+	a	T_3	(T_5)
(5)	+	T_5	T_4	(T_6)

	op	arg	arg
(0)	↑	e	f
(1)	*	b	c
(2)	/	(1)	(0)
(3)	*	b	a
(4)	+	a	(2)
(5)	+	(4)	(3)

(1) Replace the result column with respective location values in the arguments.

(2) This reduces space of saving results, no need to manage temporary compiler.

(3) Not good for code optimization or optimization compiler.

SYSTIMAX®
SOLUTIONS

(3) Indirect Triples

It is an enhancement over triples representation. Uses an additional instruction array to list the pointers to the triples in the desired order.

Thus, it uses pointer instead of position to store results which enables the optimizers to freely reposition the expression to produce an optimized code.

	Statement
35	(0)
36	(1)
37	(2)
38	(3)
39	(4)
40	(5)

Desired order should be maintained

Execution order

location	Op	Arg	Arg
(0)	↑	e	f
(1)	*	b	c
(2)	/	(1)	(0)
(3)	*	b	a
(4)	+	a	(2)
(5)	+	(4)	(3)

Q. $a = b * -c + b * -c$, Make 3 address code & then translate them to Quadruple, Triples & Indirect Triples

$T_1 = (-)\ c$
$T_2 = b * T_1$
$T_3 = (-)\ c$
$T_4 = b * T_3$
$T_5 = T_2 + T_4$
$a = T_5$

	Op	arg1	arg2	result
(0)	-	c		T_1
(1)	*	b	T_1	T_2
(2)	-	c		T_3
(3)	*	b	T_3	T_4
(4)	+	T_2	T_4	T_5
(5)	=	T_5		a

Don't learn again

(Same As Three Addres Code)

Translation of Assignment Statements (SDT)

In SDT, assignment statement mainly deals with exp. The exp. can be real, integer, array or records.

$S \rightarrow id = E$	{ p = look_up(id.name); (Symbol Table)
	if p ≠ nil then
	Emit (p = E.place) // Generate TAC (Three address Code).
	else
	error;
	}

$E \rightarrow E_1 + E_2$	{ E.place = newtemp(); Emit (E.place = E_1.place '+' E_2.place) }
$/ E_1 * E_2$	{ E.place = newtemp(); Emit = (E.place = E_1.place '*' E_2.place) }.
$/ E_1$	{ E.place = E_1.place }

/ id	{ p = look_up (id.name)).
	if (p ≠ nil) then
	Emit (p = E.place)
	else
	error;
	}

⇒ The emit function is used for appending the TAC (Three Address Code) to the output file.

⇒ The p returns the entry for id.name in the symbol table.

⇒ newtemp() is a function used to create new temporary variable.

Eg

$S \rightarrow id = E$ { gen (id.name = E.place); }.

$E \rightarrow E_1 + T$ { E.place = newtemp(); gen (E.place = E_1.place + T.place); }

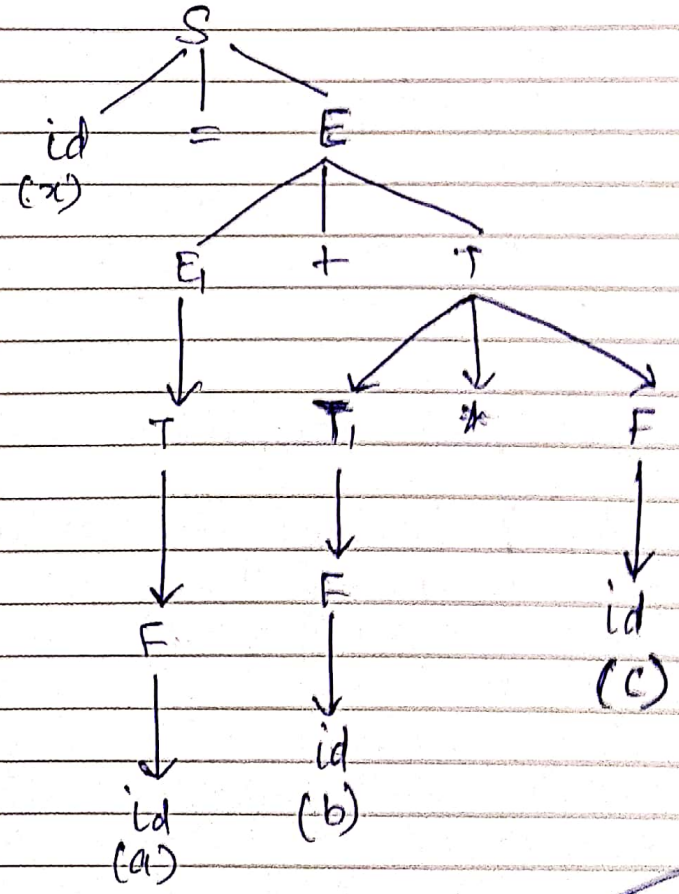
/T { E.place = T.place; }

$T \rightarrow T_1 * F$ { T.place = newtemp(); gen (T.place = T_1.place * F.place); }

/F { T.place = F.place; }

$F \rightarrow id$ { F.place = id.name; }.

for $x = a + b * c$



$t_1 = b * c$

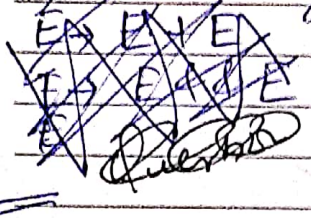
$t_2 = a + t_1$

$x = t_2$

Boolean Expressions (SDT)

Boolean expressions have 2 primary purposes:-

1) Used for computing logical values.
2) Used as conditional expressions using (if-then-else) or (while-do).
3) && and || are left-associated
4) Not has higher precedence than && and lastly ||.



Grammar

$E \rightarrow E_1$ OR E_2 { E.place = newtemp(); Emit(E.place "=" E_1.place "OR" E_2.place) }

$E \rightarrow E_1$ AND E_2 { E.place = newtemp(); Emit(E.place "=" E_1.place "AND" E_2.place); }

$E \rightarrow$ Not E_1 { E.place = newtemp(); Emit(E.place "=" "Not" E_1.place) }

$E \rightarrow (E_1)$ { E.place = E_1.place }

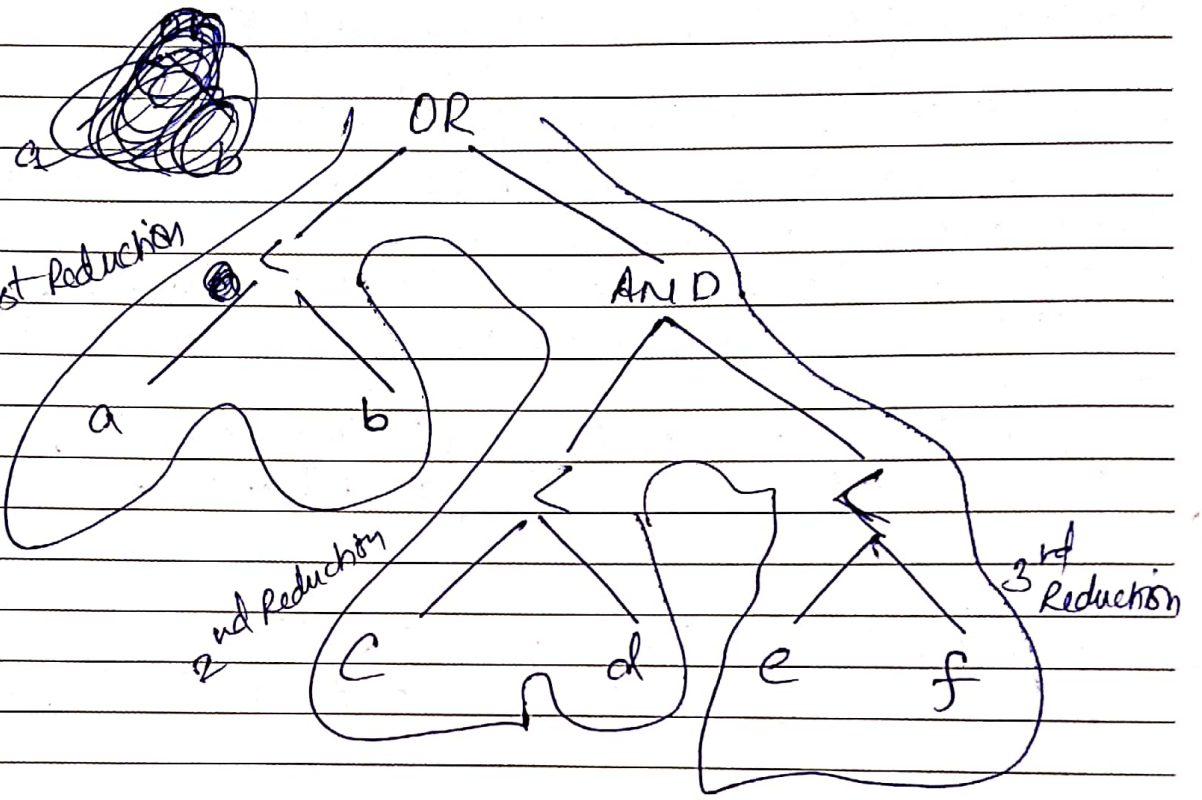
$E \rightarrow id_1$ relop id_2 { E.place = newtemp(); Emit(if id_1.place relop id_2.place goto nextstate+3); Emit(E.place "=" 0); Emit(goto nextstate+2); Emit(E.place "=" 1); }

E → True { E.place = newtemp(); Emit(E.place = "1"); }

E → False { E.place = newtemp(); Emit(E.place = "0") }

SYSTIMAX®
SOLUTIONS

Eg a<b or c<d and e<f



1st Reduction

100: if a<b goto(103)
101: t = 0
102: goto(104)
103: t = 1

2nd Reduction

104: if c<d goto(107)
105: $t_2 = 0$
106: goto(108)
107: $t_2 = 1$

3rd Reduction

108: if e<f goto (111)
109: $t_3 = 0$
110: goto(112)
111: $t_3 = 1$
112: $t_4 = t_2$ && t_3
113: $t_5 = t$ || t_4.

(It uses logical rules to reason about the behaviour of a program at run time)

Type Checking (SDT)

for int:

$E \to E_1 + E_2$ { if ((E_1.type == E_2.type) && (E_1.type = int))
then E.type = int
else
error;
}

$E \to E_1 == E_2$ { if ((E_1.type == E_2.type) && (E_1.type = int/bool))
then E.type = bool
else
error;
}

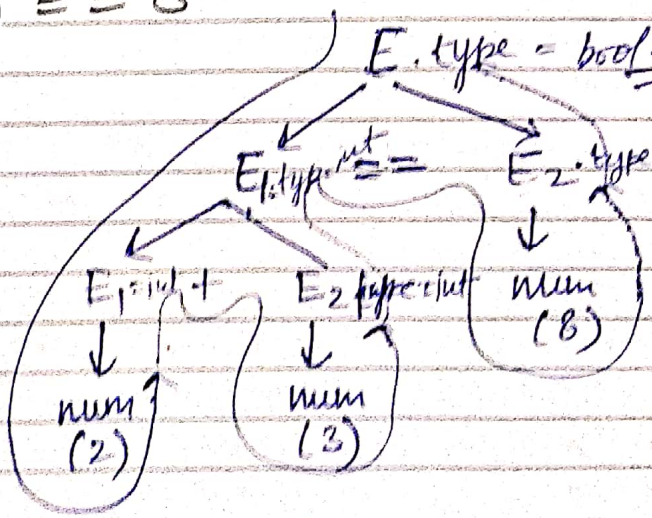
$E \to (E_1)$ { E.type = E_1.type ; }

$E \to$ num { E.type = int ; }

$E \to$ True { E.type = bool ; }

$E \to$ False { E.type = bool ; }

Q. (2+3) == 8.



hence, type checked.

SYSTIMAX SOLUTIONS

Type Expression & Type System

TE
- Basic Type {int, bool, char}
- Type Name {arrays, pointer, record}
- {Type constructor}

Array: T = T[]
so, array(I, T) = T[]
arr[1...10] of int = arr [1...10, int]
Type constructor

TS ⇒ Collection of rules for assigning type expression.
⇒ Concise formulation of the semantic checking rules.

- Basic type: Eg: int, char, float
- Type Construction: Eg: Array, String, Record
- Type Equivalence
  - name equivalence: Eg: char a, b; a = 'A'; b = 'a';
  - structural equivalence: Eg: Used in pointer

Type Conversion / Type Casting

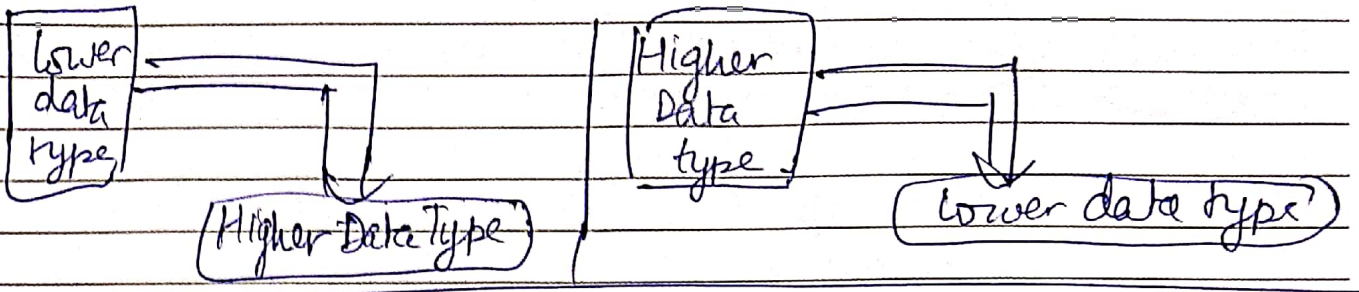
A type cast is basically a conversion from one type to another. There are two types of conversions:

Implicit conversion

⇒ If a compiler can convert one data type into another data type automatically
⇒ There is no data loss
Eg: int a = 20; float b = a; // Implicit

Explicit

⇒ There may be a data loss.
⇒ If a compiler doesn't convert automatically and data of one type is converted explicitly to another type
Eg: float to int



Symbol Table

<symbol name, type, attribute>
Eg <static, int, result>;

→ It is used by compilers to remember information (ie. identifiers) appearing in the source language program.

→ lexical analyser and parser fill up the entries in symbol table

→ Symbol table stores name, type, location, scope and other attributes.

Eg
```
{
    int x, y;
    float f1;
    msg();
}
msg()
{ int a, b;
  ----
}
```

SNo	Name	type
1	x	int
2	y	int

It stores information variables, procedures, functions info, constants, labels or structures.

Usage of Symbol Table

→ Semantic Analysis
→ Code generation
→ Optimization
→ Error detection

Representation of Symbol Table

(1) Fixed length
(2) Variable length

SYSTIMAX®
SOLUTIONS

Features of Symbol Table

Operations 1) lookup 2) Insert 3) Modify
4) Delete

1) Used to check-up whether it is defined or not. It lookup for the statements being defined or not and incase it is not defined, then it defines it in symbol table.

2) The insertion of an entry in symbol table is done through a insert operation.

3) Modify is used change the instructions stored in the symbol table. Suppose, we are to change the type of a variable, so then it is used

4) When program is runned completely then the variables dont need to stored or kept in symbol table. Hence, they are deleted.

Issues in Symbol Table

1) The format of storing. Array, Tree or what data structure

2) Access Method. Which search method to use? Linear, Binary etc

3) location of Storage – Primary or secondary. If the storage of symbol table exceeds then switch to secondary memory

4) Scope - Scope of variable. If x & y are declared in function & in main, causes confusion

Implementation of Symbol Table

Suppose, { int x; ... }

The data structure should be designed to allow the compiler to find name quickly

S.No	Name	Type
1	x	int

For simple scope variables

1) **Linear Table** = It is a simple array of records with each record corresponding to an identifier of the program.

int x, y;
msg( );

Lookup [O(n)]
Modify [O(n)]
~~Location~~ [O(1)]

SNo	Name	Type	location
1	x	int	offset of x
2	y	int	offset of y
3.	msg( )	func(n)	offset of offset

2) **ordered list** = It is a variation of linear table. List may be sorted in some fashion. Modification to the ordered list is done known as 'self-organizing list'.

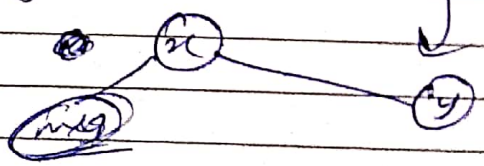
* Suppose, the most frequently used identifier to be accessed first

id	Name
first	

Lookup [O(log n)]
Modify [O(log n)]

3) **Tree** - Tree organisation may speed up access. Each entry is represented by node.

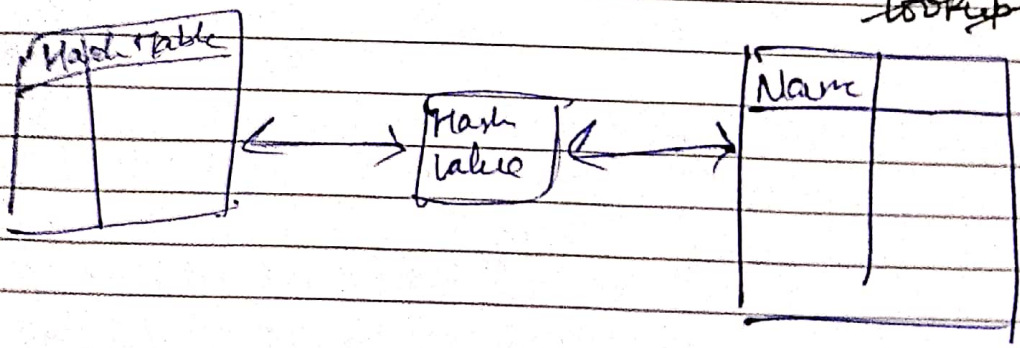
Lookup
O(log n)



on the basis of order of access.

4) **Hash Table** - Hash Table organization where we want to minimize access time.

~~Lookup O[log n]~~



SYSTIMAX® SOLUTIONS

Scoped Symbol Table

The scope of symbol defines the region of the program in which a particular definition of symbol is valid.

```
{
  int x = 20;
  cout << x; // 20
}
main()
{
  int x = 30;
  cout << x; // 30
}
```

Scopes

1. Global ⇒ Visible throughout the program.
2. Local scope (function) ⇒ Visible throughout the function/block.
3. File wide scope ⇒ Visibility in one file.

Scoping can be broadly classified into 2 categories:-

1. Dynamic or Runtime Scoping

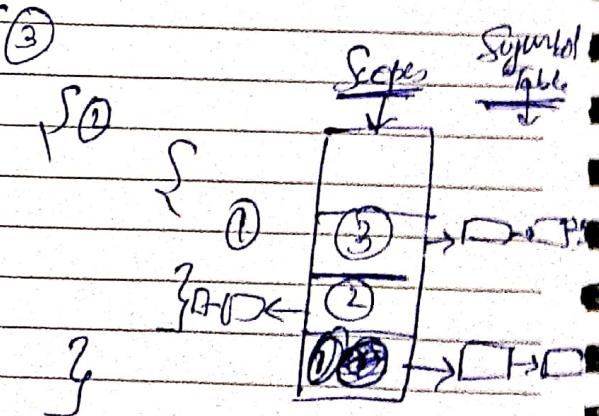
When different functions call the same function with different scopes or parameters at runtime.

```
p2(x)
{
  int a = x;
  a = 3; p1(3);   // diff. symbol table made
}
p3(x)
{
  int b = x;
  b = 4; p1(4);   // diff. symbol table made
}
```

Diff values for same function

2. Static or Lexical Scoping

- Nested Lexical Scoping
- One Table per scope
- One Table for all scope.



Either one table per scope (1, 2 or 3) or one table for all scopes (1, 2 or 3)

Storage Run-Time Administration (Imp)

① Stack Allocation Scheme

② Block structured language.

SL → Intermediate Code → machine code

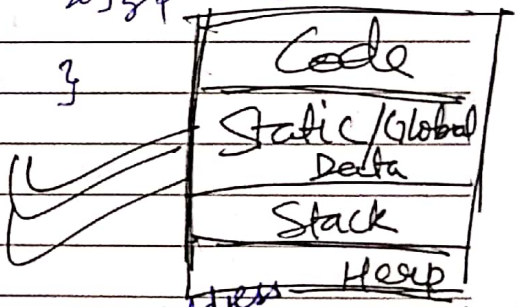
↓ Runtime

Runtime environment. { like memory, saving etc

Issues with Runtime Environment

1) linkage among procedures and functions like linking b/w ①, ② & ③
2) Parameter passing in functions
3) Interface for input/output devices
4) Interfacing to operating system
5) Mechanism of accessing variables.
6) Allocation and layout of storage

```
main() ①
{
   msg();
}
msg() ②
{ xyz();
}
xyz{ ③
}
```

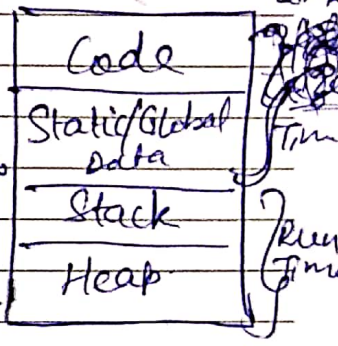


Elements in Run-time environment

1) Memory Organisation

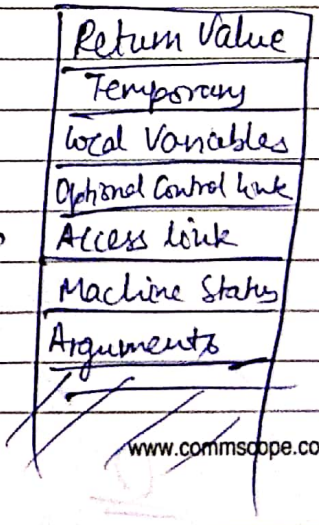
Code → Fixed size & holds target code
Static/Global Variables → Fixed size & holds global/local variables
Stack (LIFO) → Dynamic Memory allocation
Heap (alloc, malloc)

layout: ↓ lower Address … ↑ higher Address



2) Activation Record

Memory block used to call another block. Of code uses an activation code in the form like this. Each function uses its own activation Record.



3) Procedure call & Return – Checks calling & returning of all functions and it should also follows

4) Parameter Passing – All parameters are passed or not and how they are dealt with.

SYSTIMAX SOLUTIONS

(1) Stack Allocation Space

Memory layout:

Code
Data (2)
Static
Heap

, hence Storage → Static (compile time), Dynamic (Run time)

(Only for local data) — Static (static)

(For local + non-local data) — Heap (dynamic)

(2) Data

Control Stack (Runtime Stack) – A stack representing procedural calls, return and flow of control is called control stack. It shows the flow of control of the program.

Activation Tree

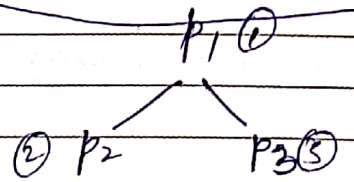
```
main()
{
  p1();
}
p1()
{
  p2();
  p3();
}
```

→ Now, the control stack has the flow of control of the functions and the order in which they are called or activated is stored in activation tree.

→ So, during execution of prog, activation of procedure can be represented by activation tree.

→ The sequence of procedure calls corresponds to pre-order traversal. (Root-left-Right) ($P_1 - P_2 - P_3$)

Eg



→ But sequence of returns corresponds to post-order traversal. (left-Right-Root) ($P_2 - P_3 - P_1$)

(3) Stack (Control stack) (LIFO)

Activation Record – It is a block of memory on control stack used to manage information for every single execution of a procedure.

Actual Parameters (1)	Return Values (2)	Control Link (3)	Access Link (4)	Saved Machine Status (5)	Local Data variables (6)	Temporaries variables (7)

(3) → Dynamic, (4) → Static

LIFO

No of functions = No of activation Records

CommScope

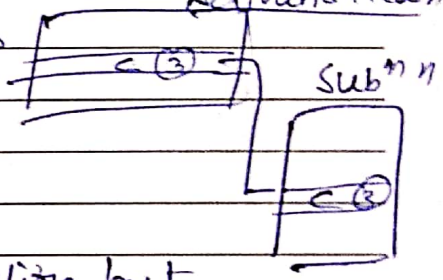
(1) The parameters that are declared inside the calling function

(2) Used to store the result of a function call. Whenever a function is called, it goes to func. definition. Returns value to the calling function

(3) It points to the activation record of the calling function. So, in order to store the add. of calling function, we use it

Activation Record

```
add()
{
  sub();
}
```



(4) It refers to the local data of called function but found in another Activation record.

(5) Whenever we call a function, we need to store the address of the next instruction to be executed. It stores address of next instruction to be executed.

(6) Variables that are local to the corresponding function.

(7) To evaluate an expression, so then temp variables are needed.

~~Allocation Strategies~~ (2) Block-Structured Lang

Storage

(1) local (Handeled by Activation Record)

(2) Non-local (handeled by Scope Information)

(A) Static (Block Structured Storage)

(B) Dynamic (Non-Block Structured Storage)

www.commscope.com

SYSTIMAX®
SOLUTIONS

(1) A()
{
int a;
}

So, a is local data/variable of (A).

return [?]
Local [a]

(2) test()
{ int a, b; //Now works as global.
{ int x, y; } Non-local Data.
{ int c, d; }
}

(A) scope test()

```
{
  int p, q;
B1 B2 { int p;
        { int r; } B3
      }
      { int q, s, t; } B4
}
```

r (B_3)	s (B_4)
P (B_2)	q (B_4)
q (B_1)	
P (B_1)	

For accessing r, B_3 is taken first

→ It uses the static (unchanging) relation b/w block of codes.

→ It refers to the identifier with the name that is declared in the closest enclosing scope of prog. test

```
{
  int a;
  { print(a); //Prints value.
}
```

(B)

→ It refers to the identifier associated with the most recent activation record.

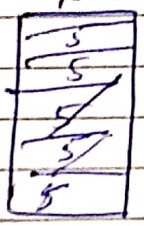
→ Uses actual sequence of calls that are executed in the dynamic

Eg

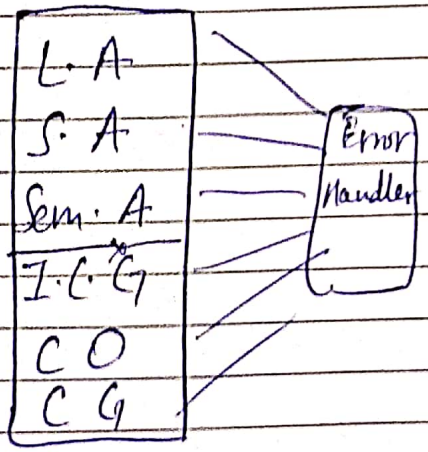
```
int r=5;
void show()
{ printf("%d", r); }

void small()
{ int r=20;
  show();
}
main()
{
  show();
  small();
  print(\n);
  show(); small();
}
```

Static Scope	Dynamic Scope
5 5	5 20
5 5	5 20

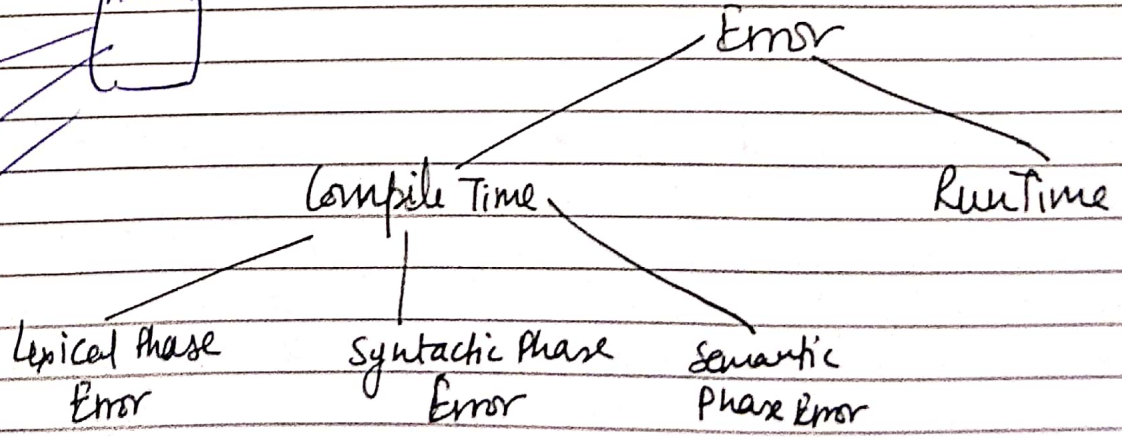
AR



ERROR DETECTION



⇒ Error Detection +
Error ~~Detection~~ Reporting +
Error Recovery



SYSTIMAX®
SOLUTIONS

Error Recovery Method	(1) Lexical Phase Error	(2) Syntactic Phase Error	(3) Semantic Phase Error
Panic Mode	✓	✓	✗
Phrase Mod	✗	✓	✗
Error Prod.	✗	✓	✗
Global "	✗	✓	✗
Using Symbol Table	✗	✗	✓

(1) Lexical Phase Error

i) Exceeding length of identifier → int char... very long ✗.
ii) Appearence of illegal character → int a, $b; ✗.
iii) Unmatched string or comment → " " but "hello ✗.

Eg
```
void main()
{
int a, @, $%= variable declaration */ ✗ (comments not complete).
a=10; ✓
print("%d", a); ✗
```

Recovery → Panic Mode ⇒ int a, $b, %b
As soon as $ is seen all attached characters are neglected till a comma is seen.

(2) Syntactic Phase Error

i) Missing Parenthesis → printf("%d", d ; ✗
ii) Missing operator → a+b c ✗
iii) Mis-spelled keyword → switch( ) ✗
iv) colon in place of semi-colon → int a = 1 : ✗
v) Extra blank space → /* comment * / ✗

Recovery:
Panic Mode ⇒ similar as lexical phase error
Phase level Recovery ⇒ when paser encounters an error, it performs necessary action on remaining i/p & o/p. It passes rest of i/p.

Error Production : Add extra grammar production and make an augmented grammar and parse the input.

Global Corrections: The parser examines the whole program & tries to find out closest match for it which is error. Has high complexity so not implemented.

(3) Semantic Phase Error –

i) Incompatible type of operands. int a; float b; int c = a+b;
ii) Undeclared variable → print(d) :-
(iii) Not matching actual argument with formal argument

Eg int a[10], b;

a = b; b.

Recovery Using Symbol Table
→ Declared variables are seen from symbol & undeclared are rejected.
→ Array is stored & int is stored