Input

↓

SA ← (G)

Syntax Analyzer

① SYNTAX ANALYZER

$G = (V, T, P, S)$

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

$V = \{E\}$
$T = \{+, *, id\}$
$S = \{E\}$

$id + id * id$

LMD

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

RMD (Right Most Recursion)

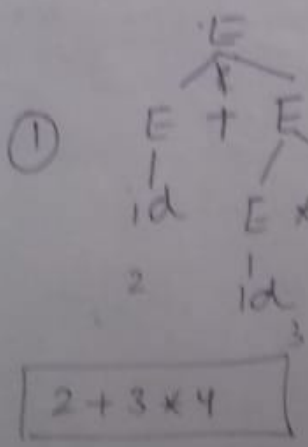
$E \to E+E$
$E \to E+E*id$
$E \to E+id*id$
$id+id*id$

LMD (Left Most Recursion)

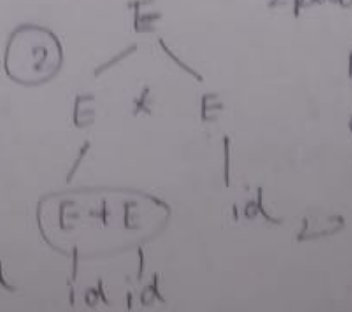
$E \to E*E$
$\to E+E*E$
$\to id+E*E$
$\to id+id*E$
$\to id+id*id$

RMD $E \to E*E$
$\to E*id$
$E+E*id$
$E+id*id$
$id+id*id$

More than one LMD or RMD it is ambiguous



2 parse tree



Parser gets confused which one to generate

2 + 3 * 4

multiplication will be just as right until

Q. Find ambiguous if using ① then generate 14 (No algorithm)

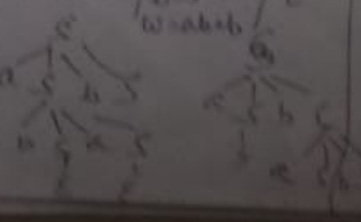
② Remove ambiguity

$S \to aS/Sa/a$

$w = aa$

$S \to aSbS/bSaS/\varepsilon$

$w = abab$



$R \to R+t/Rt/t^*/a/b$

(2)

$E \to E+T$

$E \to E+id / id$

$E \to E+T / T$

$T \to T \times F / F$

$F \to id$

$E+T$

$T \times F$

$E+id$

$E+id$

Ambiguity to remove ambiguity instead of

$E \to E+E /$ Not taking associativity

$E \times E /$ and precedence

id

$id+id \times id$

In order to define associativity we define recursion

In order to define Precedence we define layer

$(id+id)+id$

$A \to A$

$P \to level$

left associativity

$E \to E+id$

$E \to \cdot$

On more operators

$2 \uparrow 3 \uparrow 2$

$2^{(3^2)}$

Right associativity

Higher precedence operator should be at last and define after two Then it should grow up

LA

$E \to E+T / T$

LA

$T \to T \times F / F$

RA

$F \to G \uparrow F / G$

$G \to id$

Level

$+ \quad * \quad \uparrow$

All + will provide fail

More precedence will be ↑

operator close to start will have least precedence More will be at last more than same operator it is left associative for + and × right for ↑

$A \to A \& B / B$

$B \to B \| C / C$

$C \to C @ D / D$

$D \to a$

Left associativity & left associativity

Recursion (4)

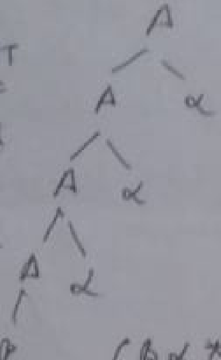
(Top) down Parser

LR — RR

$A \to A\alpha / \beta$ $\quad$ $A \to \alpha A / \beta$

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

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



$(\beta\alpha^*)$

$(\alpha^*\beta)$

$A \to \beta\alpha^*$
$A \to \beta A'$
$A' \to \varepsilon / \alpha A'$

$A \to \beta A'$
$A' \to \varepsilon / \alpha A'$
$A \to \alpha A / \beta$

Remove Left Recursion

Left Recursion

① $S \to (L) / x$
$L \to L, S / S$
$A \to A\alpha / \beta$

$\alpha = , S$
$\beta = S$

~~S → S01S / 01~~

$S \to S0S1S / 01$
$A \to A\ \alpha\ / \beta$

$S \to 01S'$
$S' \to \varepsilon / 0S1SS'$

$A \to A\alpha / \beta$
$A \to \beta A'$
$A' \to \varepsilon / \alpha A'$

G: A, U — No pars; G: LR, RR — $A \to A\alpha$

$A \to \beta A'$ $A' \to \alpha A' / \varepsilon$	$L \to SL'$ $L' \to , SL' / \varepsilon$	② $A \to A\alpha_1 / A\alpha_2 / A\alpha_3 / \beta_1 / \beta_2 / \beta_3$ $A \to \beta_1 A' / \beta_2 A' / \beta_3 A'$ $A' \to \alpha_1 A' / \alpha_2 A' / \alpha_3 A'$

OP can pree ambiguous

iEtiEtSeS

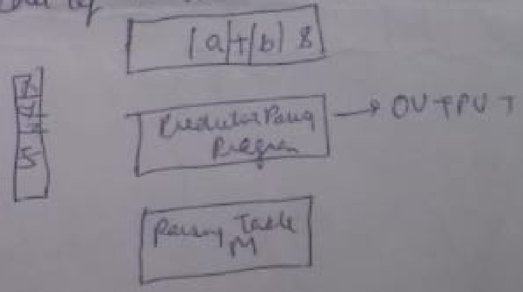
Parser

- Topdown parser
  - Recursive descent
    - Top with Backtrack
      - Brute force Method
  - Top without Backtrack (LR X, NDX)
    - Recursive descent
    - Non Recursive descent
- Bottomup parsers (SR) BUP
  - Operator precedence (ambiguous)
  - LR parser
    - LR(0) SLR(1)
    - LALR(1)
    - CLR

Model of Non recursive PP

LL(1) → look ahead — left to right, leftmost derivation

STACK



Top down Parser

what to use

which rule to choose

Top down

when to reduce

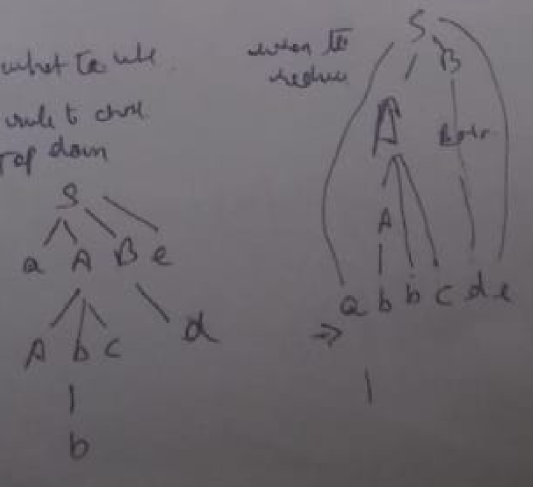
Bottom up

$S \to aABe$

$A \to Abc / b$

$B \to d$

$w = abbcde$

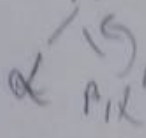


Top down Left most

DETERMINISTIC GRAMMAR

$A \to \alpha\beta_1 | \alpha\beta_2 | \alpha\beta_3$

$\alpha\beta_2$



Backtrack

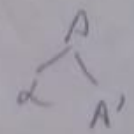
Prob: α is ok but don't know β_1 or β_2 or β_3 to need to backtrack

Backtrack is happening due to choosing α so if we could see β also we can see which production to choose, we are making decision to choose by using β_3

Left factoring

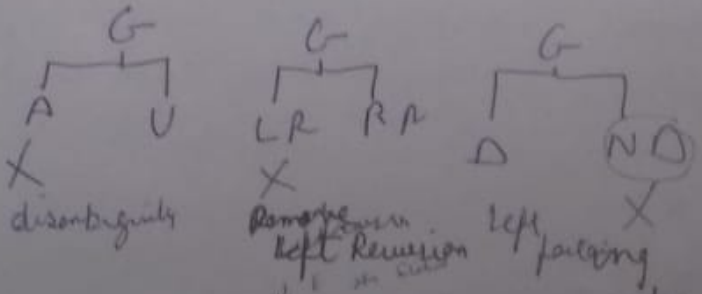
$A \to \alpha A'$

$A' \to \beta_1 | \beta_2 | \beta_3$



Postponing till next point this is deterministic.

Procedure to convert Non deterministic to Deterministic grammar is called left factoring



$S \to iEtS | iEtSeS | a$

$E \to b$

⇒ Is this ambiguous or not

$S \to iEtSS'$

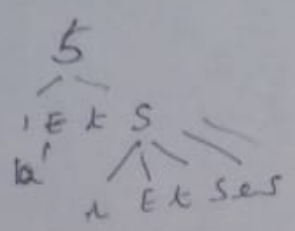
$S' \to \varepsilon | eS$

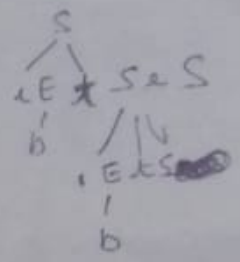
$E \to b$

(6)

i E t i E t S e S

i b t i a t a a

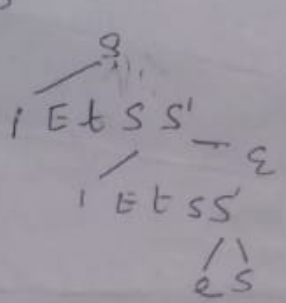


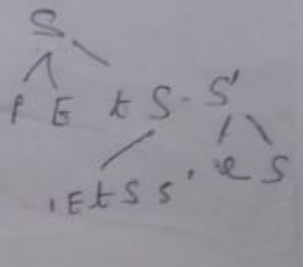


$S \rightarrow iEtSS'$

$S' \rightarrow \varepsilon / eS$

$E \rightarrow b$





Left parsing does not eliminate ambiguity

$A \rightarrow a\beta_1/\alpha\beta_2/\alpha\beta_3$

$A \rightarrow \alpha A'$

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

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

$S \rightarrow bSSaaS / bSSaSb / bSb / a$

$S \rightarrow aS' / b$

$S' \rightarrow SSbS / SaSb / bb$

$S' \rightarrow SS'' /$

$S'' \rightarrow SbS / aSb$

$S \rightarrow bSS' / a$ ①

$S' \rightarrow SaaS / SeSb / b$

$S' \rightarrow aS'$ ②

$S'' \rightarrow aS / Sb$ ③

(8)

	FIRST	FOLLOW
$S \to ACB / CbB / Ba$	$\{d, g, h, \varepsilon, b, a\}$	$\{\$\}$
$A \to da / BC$	$\{a\ g\ h\ \varepsilon\}$	$\{h, g, \$\}$
$B \to g / \varepsilon$	$\{g\ \varepsilon\}$	$\{\$, a, h, g\}$
$C \to h / \varepsilon$	$\{h, \varepsilon\}$	$\{g, \$, b, h\}$

	FIRST	FOLLOW
$E \to TE'$	$\{id, (\}$	$\{\$,)\}$
$E' \to +TE' / \varepsilon$	$\{+\ \varepsilon\}$	$\{\$\ )\}$
$T \to FT'$	$\{id, (\}$	$\{+, \$,)\}$
$T' \to *FT' / \varepsilon$	$\{*, \varepsilon\}$	$\{+\ \$\ )\}$
$F \to id / (E)$	$\{id, (\}$	$\{*, +, \$,)\}$

	FIRST	FOLLOW
$S \to aABb$	$\{a\}$	$\{\$\}$
$A \to c / \varepsilon$	$\{c, \varepsilon\}$	$\{a, b\}$
$B \to d / \varepsilon$	$\{d, \varepsilon\}$	b

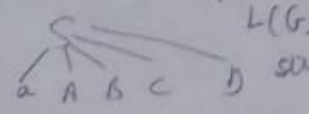
$S \to aBDh$	$\{a\}$	$\{\$\}$
$B \to cC$	$\{c\}$	$g + h$
$C \to bC / \varepsilon$	$\{b, \varepsilon\}$	$g + h$
$D \to EF$	$\{g, +, \varepsilon\}$	h
$E \to g / \varepsilon$	$\{g, \varepsilon\}$	$+ h$
$F \to + / \varepsilon$	$\{+, \varepsilon\}$	h

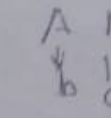
(7) $S \Rightarrow \alpha$ where α may contains NT then α is sentential form of G. $\overset{+}{\Rightarrow}$ relation to

given a grammar G with S (start) $\overset{+}{\Rightarrow}$ relation to define L(G), language generated by G. strings in L(G) contain only terminals symbol of G.

First and follow → say string of terminals is in L(G) if $S \overset{+}{\Rightarrow} w$ string w is sentence.

$S \rightarrow aABCD$
$A \rightarrow b$
$B \rightarrow c$
$C \rightarrow d$
$D \rightarrow e$





First ( ) - will start from S and derive all the string that will be derived

FOLLOW ( ) → terminal that will follow a NT in process of derivation

$S \Rightarrow \alpha A a \beta$

input followed by $

A B a D $
follow of B is a

	FIRST	FOLLOW
$S \rightarrow ABCDE$	{a, b, c}	{$}
$A \rightarrow a/\varepsilon$	{a, ε}	{b, c}
$B \rightarrow b/\varepsilon$	{b, ε}	{c}
$C \rightarrow c$	{c}	{d, e, $}
$D \rightarrow d/\varepsilon$	{d, ε}	{e, $}
$E \rightarrow e/\varepsilon$	{e, ε}	{$}

Substitute ε in A so S → BCDE

Follow of S is $

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

Remove left recursion	First	Follow
$E \rightarrow TE'$	id, (	$,)
$E' \rightarrow +TE'/\varepsilon$	+, ε	$,)
$T \rightarrow FT'$	id, (	+, $,)
$T' \rightarrow *FT'/\varepsilon$	*, ε	+, $,)
$F \rightarrow id/(E)$	id, (	*, +, $,)

follow of E is in prod 2

$S \rightarrow Bb/Cd$	a, b, c, a	$
$B \rightarrow aB/\varepsilon$	a, ε	b
$C \rightarrow cC/\varepsilon$	c, ε	d

1) Place $ in follow(S) where S is start symbol $ is input right end marker
2) If there is production $A \rightarrow \alpha B \beta$ everything in First(β) except ε is placed in follow(B)
3) If there is production $A \rightarrow \alpha B$ or $A \rightarrow \alpha B \beta$ where First(β) contain ε, then everything in follow(A) is in follow(B)

LL(1) PARSING

Here L represents that scanning of input will be done from left to right and second L stands that in this Parsing technique we are going to use Leftmost Derivation tree and 1 represents number of look ahead.

	id	+	*	(	)	$
E	E→TE'			E→TE'		
E'		E'→+TE'			E'→ε	E'→ε
T	T→FT'			T→FT'		
T'		T'→ε	T'→*FT'		T'→ε	T'→ε
F	F→id			F→(E)		

Here first L represents that scanning

First() if there is a variable/NT if we try to derive all strings then leading Terminal Symbol is first.

Follow() Terminal Symbol which follow a variable in the process of derivation.

continuation of LL(1)

1) First ( ) , FOLLOW( )
2) Construct parsing table
3) stack implementation
4) parse the input string

stack implementation — Moves by predictive parser on id + id * id

Stack	Input	Action
$ E	id + id * id $	E → TE'
$ E' T	id + id * id $	T → FT'
$ E' T' F	id + id * id $	F → id
$ E' T' id	id + id * id $	
$ E' T'	+ id * id $	T' → ε
$ E'	+ id * id $	E' → +TE'
$ E' T +	+ id * id $	
$ E' T	id * id $	T → FT'
$ E' T' F	id * id $	F → id
$ E' T' id	id * id $	
$ E' T'	* id $	T' → *FT'
$ E' T' F *	* id $	
$ E' T' F	id $	F → id
$ E' T' id	id $	
$ E' T'	$	T' → ε
$ E'	$	E' → ε
$	$	

16/9 Some grammar has multiple defined entries (11)

$S \to iEtSS' \mid a$
$S' \to eS \mid \varepsilon$
$E \to b$

FIRST(S) {i, a}
(S') {e, ε}
(E) {b}

FOLLOW(S) {e, $}
FOLLOW(S') {e, $}
FOLLOW(E) {t}

Parsing Table M (Not LL(1))

	Input Symbol					
	a	b	e	i	t	$
S	S → a			S → iEtSS'		
S'			S' → ε S' → eS			S' → ε
E		E → b				

The entry for M[S', e] contains both S' → eS and S' → ε, since FOLLOW(S') = {e, $}. The grammar is ambiguous and ambiguity is manifested by a choice in what production to use when an e (else) is seen.

A grammar whose parsing table has no multiple-defined entries is said to be LL(1).

I - L → scanning input from left to right
II - L → leftmost derivation
III - 1 → using one input symbol for look ahead at each step to make parsing decision

No ambiguous left recursive grammar can be LL(1).

A grammar G is LL(1) if and only if whenever $A \to \alpha \mid \beta$ are two distinct productions of G the following conditions hold:

1) For no terminal 'a' do both α and β derive strings beginning with 'a'
2) At most one of α and β can derive empty string
3) If $\beta \Rightarrow \varepsilon$, then α does not derive any string beginning with a terminal in FOLLOW(A).

Shift Reduce Parsing

Eg

E → E+E
E → E*E
E → (E)
E → id

Stack	Input	Action
$	id1+id2*id3$	shift
$id1	+id2*id3$	Reduce by E→id
$E	+id2*id3$	shift
$E+	id2*id3$	shift
$E+id2	*id3$	reduce by E→id
$E+E	*id3$	shift
$E+E*	id3$	shift
$E+E*id3	$	reduce by E→id
$E+E*E	$	reduce by E→E*E
$E+E	$	reduce E→E+E
$E	$	accept

1) shift next input symbol shifted onto top of stack
2) reduce parser knows right end of handle is at top of stack. It must locate left end of handle within stack and decide with what NT to replace handle
3) Accept action, parser announces successful completion of parsing
4) In an error case, parser discovers that a syntax error has occurred and calls an error recovery routine

Bottom up parsing

Consider the grammar

$S \to aABe$
$A \to Abc / b$
$B \to d$

The sentence abbcde can be reduced to S by following

abbcde
aAbcde
aAde
aABe
S

$S \to aABe \to aAde \to aAbcde \to abbcde$
rm rm rm rm

Rightmost derivation in reverse.

Handles

A 'handle' of a string is a substring that matches right side of a production and whose reduction to NT on left side of production represents one step along reverse of rightmost derivation

Handle of a right sentential form γ is a production $A \to \beta$ and a position of γ where string β may be found and replaced by A to produce previous right sentential form in rightmost derivation of γ.

i.e $S \Rightarrow \alpha Aw \Rightarrow \alpha\beta w$, then $A \to \beta$ in position following α is a handle of $\alpha\beta w$. String 'w' to right of handle contains only terminal symbols.

22-9

Operator Precedence parsing

The grammar have the property that no production on right side is ε or has two adjacent NON terminals.

$E \to EAE \mid (E) \mid -E \mid id$

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

is not operator grammar because right hand 3 consecutive NT. However we can substitute for A each of its alternate

$E \to E+E \mid E-E \mid E/E \mid E*E \mid (E) \mid -E \mid id$

operator precede can be defined using three relation $<\cdot$ $\doteq$ $\cdot>$

$S \to SAS/a$

$A \to bSb/b$

$S \to SbSbS \mid SbS \mid a$

$A \to bSb/b$

operator precedence grammar can parse ambiguous grammar

$E \to E+E$

$E \to E*E$

$E \to id$

Operator precedence relation

	id	+	*	$
id	-	$\cdot>$	$\cdot>$	$\cdot>$
+	$<\cdot$	$\cdot>$	$<\cdot$	$\cdot>$
*	$<\cdot$	$\cdot>$	$\cdot>$	$\cdot>$
$	$<\cdot$	$<\cdot$	$<\cdot$	-

Rules for Precedence $<\cdot$ $\cdot>$ $\doteq$

1) No ids come adjacent.
2) id has more preference than any other operator
3) $ has least preference than any other.
4) * $\cdot>$ + * operator with high precedence have high precedence
5) + left associative $\Rightarrow$ + $\cdot>$ +, exponential right $\uparrow <\cdot \uparrow$

23-9 Convert relation table to function table

→ length of longest path

$f\,id \to g* \to f+ \to g+ \to f\,\$$
(1) (2) (3) (4)

$g\,id \to f* \to g* \to f+ \to g+ \to f\,\$$

	id	+	*	$
f	4	2	4	0
g	5	1	3	0

$f_4^{*} > {}_1^{g+}$ compare by length

Function table represent same information but size of function table is very less than relation table.

	Function Table	Relation Table
size	$O(2n)$	$O(n^2)$

Suppose 100 operators are there then

function table 20 Relation table 10000

Disadvantage

f-id and g-id

$P \to SR/S$
$R \to bSR/bS$
$S \to wbS/w$
$w \to L*w/L$
$L \to id$

$P \to SbSR/SbS/S$
$P \to SbP/SbS/S$
$S \to wbS/w$
$w \to L*w/L$
$L \to id$

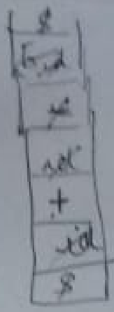
	id	*	b	$
id	-	>	>	>
*	<	<	>	>
b	<	<	<	>
$	<	<	<	-

bright answer

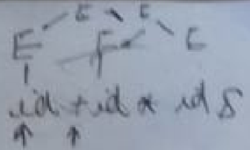
* is right associative

* and blank | * higher level >

23-9 Stack



(15)



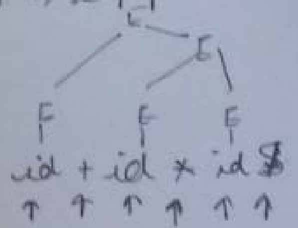
<id> + <id> * <id> $

Put $ in stack when stack has lower we push

$ id < so push, id + ·> so pop.

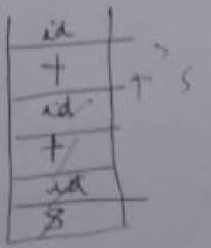
·> pop

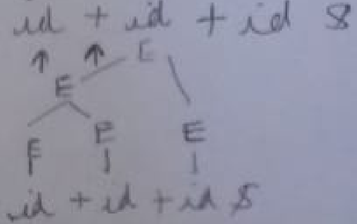
*·> $
+·> $



Rules

1) $ bottom at stack and $ on right side of string
2) Scan left to right when input and top of stack are compared when top of stack is <· than input we push otherwise pop
3) $ $ accept





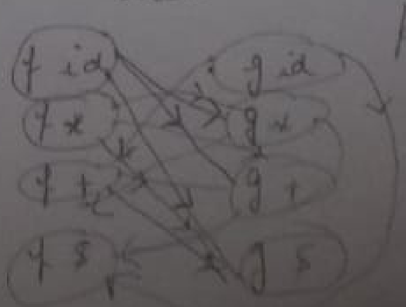
Inspite of ambiguous we get one parse tree due to precedence.

Relation table $O(n^2)$ size of table is very large.

f \ g	id	+	*	$
id	-	·>	·>	·>
+	<·	·>	<·	·>
*	<·	·>	·>	·>
$	<·	<·	<·	-

Rows function f
Column function g

We construct operator function table.



f → g
f ← g
No cycles