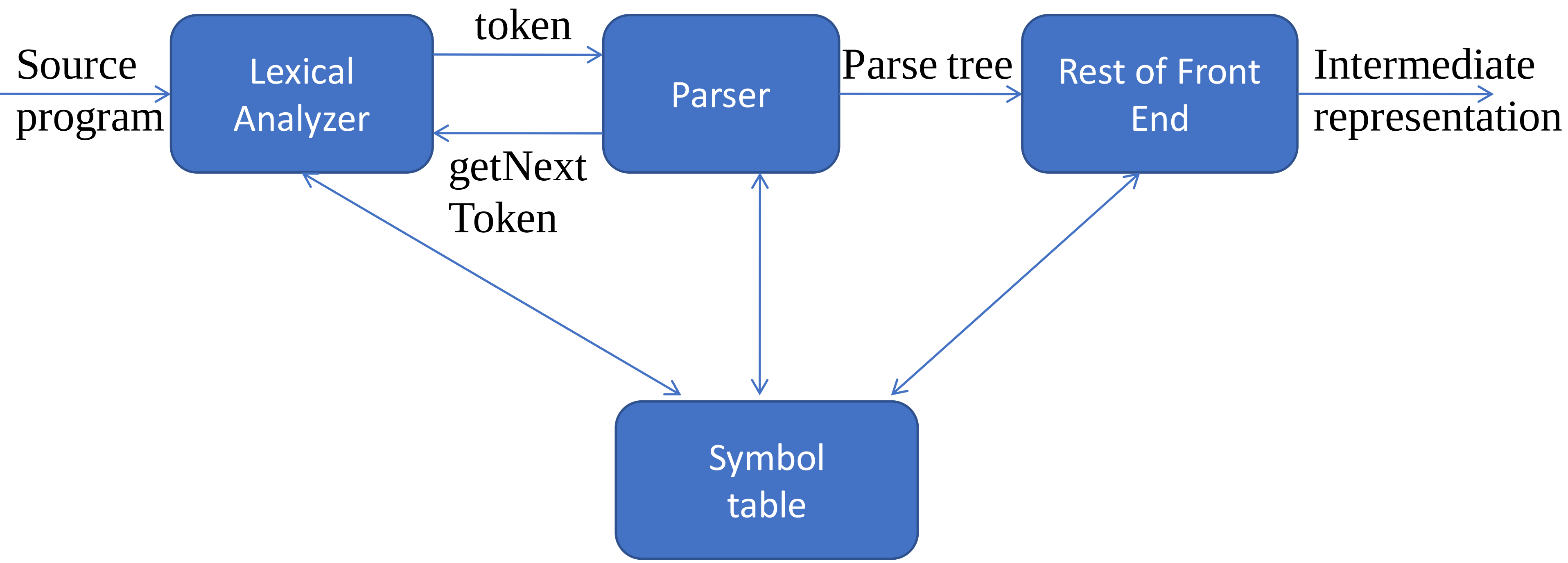
Compiler course

Syntax Analysis

Outline

- Role of parser
- Context free grammars
- Top down parsing
- Bottom up parsing
- Parser generators

The role of parser



SYNTAX ANALYSIS INTRODUCTION

- LEXICAL PHASE IS IMPLEMENTED ON FINITE AUTOMATA & FINITE AUTOMATA CAN REALLY ONLY EXPRESS THINGS WHERE YOU CAN COUNT MODULUS ON K.
- REGULAR LANGUAGES – THE WEAKEST FORMAL LANGUAGES WIDELY USED
- – MANY APPLICATIONS
- – CAN'T HANDLE ITERATION & NESTED LOOPS(NESTED IF ELSE).
- TO SUMMARIZE, THE LEXER TAKES A STRING OF CHARACTER AS INPUT AND PRODUCES A STRING
- OF TOKENS AS OUTPUT.
- THAT STRING OF TOKENS IS THE INPUT TO THE PARSER WHICH TAKES A STRING OF TOKENS AND PRODUCES A PARSE TREE OF THE PROGRAM.
- SOMETIMES THE PARSE TREE IS ONLY IMPLICIT. SO THE, A COMPILER MAY NEVER ACTUALLY BUILD THE FULL PARSE

Error handling

- Common programming errors
    - Lexical errors
    - Syntactic errors
    - Semantic errors
    - Lexical errors
- Error handler goals
    - Report the presence of errors clearly and accurately
    - Recover from each error quickly enough to detect subsequent errors
    - Add minimal overhead to the processing of correct progrms

Error-recover strategies

- Panic mode recovery
  - Discard input symbol one at a time until one of designated set of synchronization tokens is found
- Phrase level recovery
  - Replacing a prefix of remaining input by some string that allows the parser to continue
- Error productions
  - Augment the grammar with productions that generate the erroneous constructs
- Global correction
  - Choosing minimal sequence of changes to obtain a globally least-cost correction

Context free grammars

- Terminals
- Nonterminals
- Start symbol
- productions

expression -> expression + term
expression -> expression – term
expression -> term
term -> term * factor
term -> term / factor
term -> factor
factor -> (expression)
factor -> **id**

$G=(\Sigma, T, P, S)$

Σ – IS A FINITE SET OF TERMINALS
T– IS A FINITE SET OF NON-TERMINALS P – IS A FINITE SUBSET OF PRODUCTION RULES

S–ISTHESTARTSYMBOL

A context-free grammar has four components:

•A set of **non-terminals** (V). Non-terminals are syntactic variables that denote sets of strings. The non-terminals define sets of strings that help define the language generated by the grammar.

•A set of tokens, known as **terminal symbols** (Σ). Terminals are the basic symbols from which strings are formed.

•A set of **productions** (P). The productions of a grammar specify the manner in which the terminals and non-terminals can be combined to form strings. Each production consists of a **non-terminal** called the left side of the production, an arrow, and a sequence of tokens and/or **on- terminals**, called the right side of the production.

•One of the non-terminals is designated as the start symbol (S); from where the production begins.

The strings are derived from the start symbol by repeatedly replacing a non-terminal (initially the start symbol) by the right side of a production, for that non-terminal.

CONTEXT FREE GRAMMAR EXAMPLES

- ARITHMETIC EXPRESSIONS

E ::= T | E + T | E - T
T ::= F | T * F |T / F F
::= id | (E)

- STATEMENTS

Steps:

1.Begin with a string with only the start symbol S
2.Replace any non-terminal X in the string by the right-hand side of some production

X -> Y1…Yn

3. Repeat (2) until there are no non-terminals

If Statement ::= if *E* then *Statement* else *Statement*

Uses of grammars

E -> E + T | T
T -> T * F | F
F -> (E) | **id**

E -> TE’
E’ -> +TE’ | Ɛ
T -> FT’
T’ -> *FT’ | Ɛ
F -> (E) | **id**

Derivations

- Productions are treated as rewriting rules to generate a string
- Rightmost and leftmost derivations
  - E -> E + E | E * E | -E | (E) | **id**
  - Derivations for **–(id+id)**
    - E => -E => -(E) => -(E+E) => -(**id**+E)=>-(**id**+**id**)
- A derivation is basically a sequence of production rules, in order to get the input string. During parsing, we take two decisions for some sentential form of input:
- Deciding the non-terminal which is to be replaced.
- Deciding the production rule, by which, the non-terminal will be replaced.
- To decide which non-terminal to be replaced with production rule, we can have two options.
- Left-most Derivation
- If the sentential form of an input is scanned and replaced from left to right, it is called left-most derivation. The sentential form derived by the left-most derivation is called the left-sentential form.
- Right-most Derivation
- If we scan and replace the input with production rules, from right to left, it is known as right-most derivation. The sentential form derived from the right-most derivation is called the right-sentential form.

•Parse trees

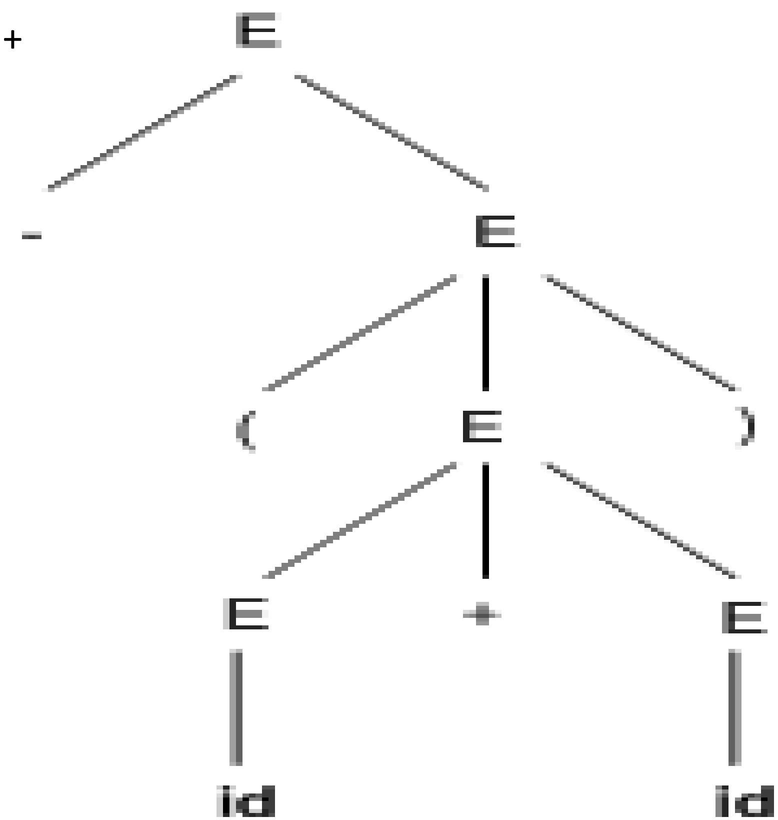
A parse tree is a graphical depiction of a derivation. It is convenient to see how strings are derived from the start symbol. The start symbol of the derivation becomes the root of the parse tree.

All leaf nodes are terminals.

All interior nodes are non-terminals.

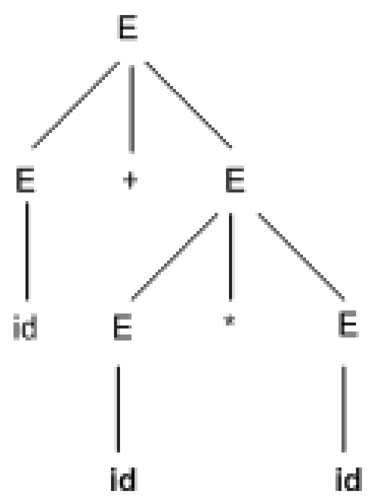
In-order traversal gives original input string.

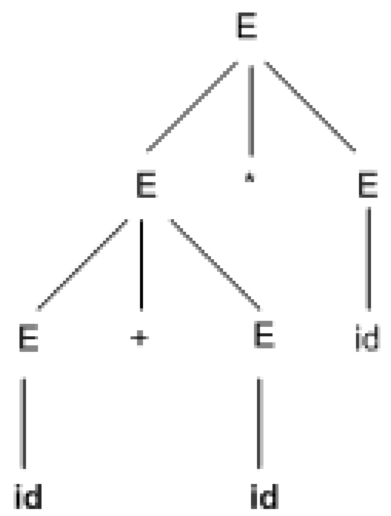
- -(**id**+**id**)
- E => -E => -(E) => -(E+E) => -(**id**+



Ambiguity

- For some strings there exist more than one parse tree
- Or more than one leftmost derivation
- Or more than one rightmost derivation
- Example: id+id*id





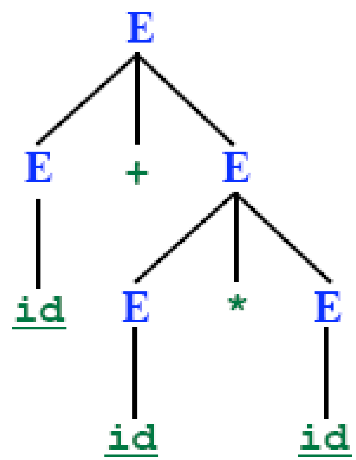
AMBIGUOUS GRAMMAR

1. E → E + E
2. → E * E
3. → (E)
4. → - E
5. → ID

Input: `id+id*id`

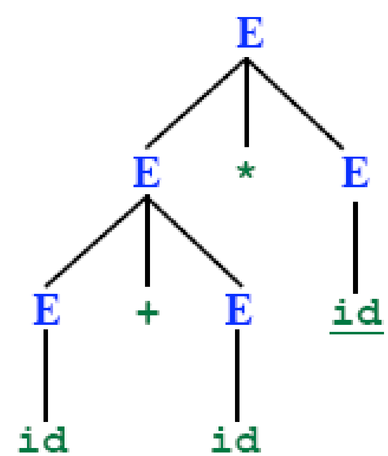
Leftmost Derivation #1

E
⇒ E+E
⇒ id+E
⇒ id+E*E
⇒ id+id*E
⇒ id+id*id



Leftmost Derivation #2

E
⇒ E*E
⇒ E+E*E
⇒ id+E*E
⇒ id+id*E
⇒ id+id*id



AMBIGUOUS GRAMMAR

- More than one Parse Tree for some sentence.
- The grammar for a programming language may be ambiguous
- Need to modify it for parsing.

- Also: Grammar may be left recursive.
- Need to modify it for parsing.

ELIMINATION OF AMBIGUITY

- Ambiguous

- A Grammar is ambiguous if there are multiple parse trees for the same sentence.

- Disambiguation

- Express Preference for one parse tree overothers
  - Add disambiguating rule into the grammar

RESOLVING PROBLEMS: AMBIGUOUS **GRAMMARS**

Consider the following grammar segment:

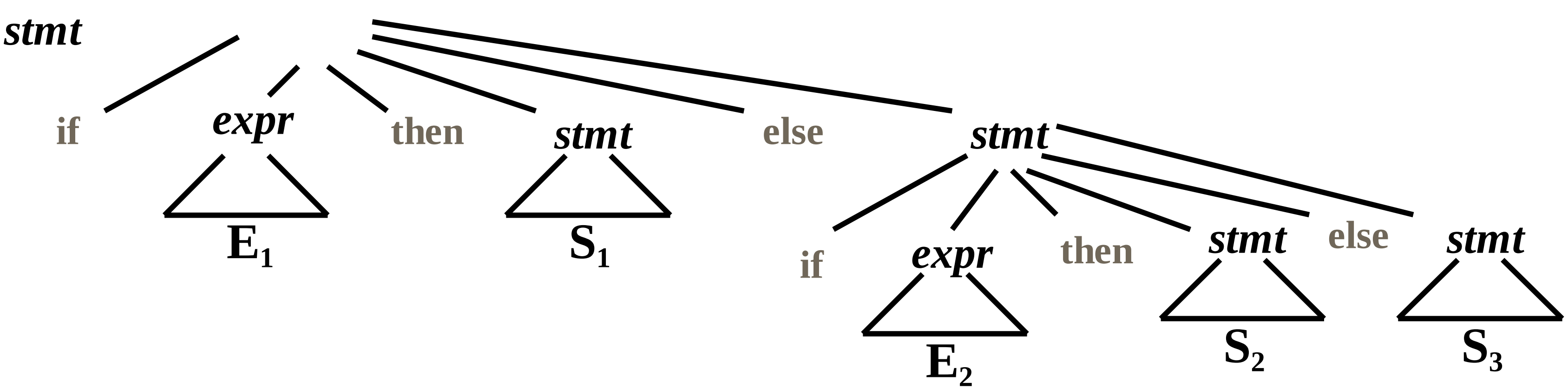
$$stmt \rightarrow \textbf{if } expr \textbf{ then } stmt$$

$$|\quad \textbf{if } expr \textbf{ then } stmt \textbf{ else } stmt$$

$$|\quad \textbf{other} \quad \textbf{(any other statement)}$$

If E1 then S1 else if E2 then S2 else S3

simple parse tree:



EXAMPLE : WHAT HAPPENS WITH THIS STRING?

If E_1 then if E_2 then S_1 else S_2

How is this parsed ?

if E_1 then
if E_2 then
S_1
else
S_2

vs.

if E_1 then
if E_2 then
S_1
else
S_2

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

$E \rightarrow E + E$
$\quad | \; E * E$
$\quad | \; id$

$V = \{E\}$

$T = \{+, *, id\}.$

$id + id * id$

Lmd

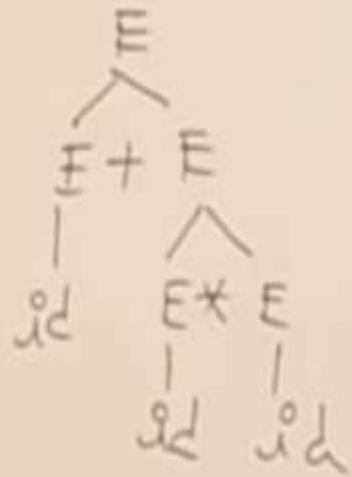
$E \Rightarrow E + E$
$\Rightarrow id + E$
$\Rightarrow id + E * E$
$\Rightarrow id + id * E$
$\Rightarrow id + id * id$

Rmd

$E \Rightarrow E + E$
$\Rightarrow E + E * E$
$\Rightarrow E + E *$
$\Rightarrow E + id *$
$\Rightarrow id + id *$

mD

$E \Rightarrow E*E$
$\Rightarrow \underline{E+E}*E$
$\Rightarrow id+E*E$
$\Rightarrow id+id*E$
$\Rightarrow id+id*id$

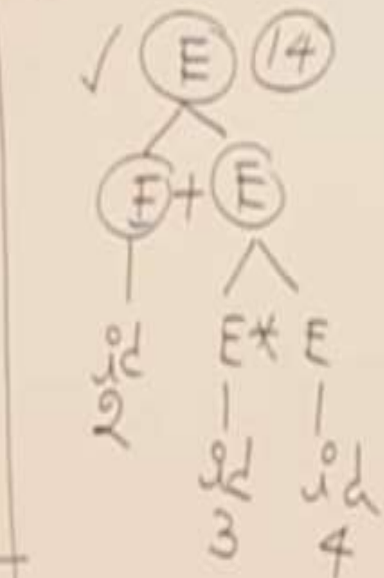


RmD: $E \Rightarrow E*E$
$\Rightarrow E*id$
$\Rightarrow E+E*id$
$\Rightarrow E+id*id$
$\Rightarrow id+id*id$

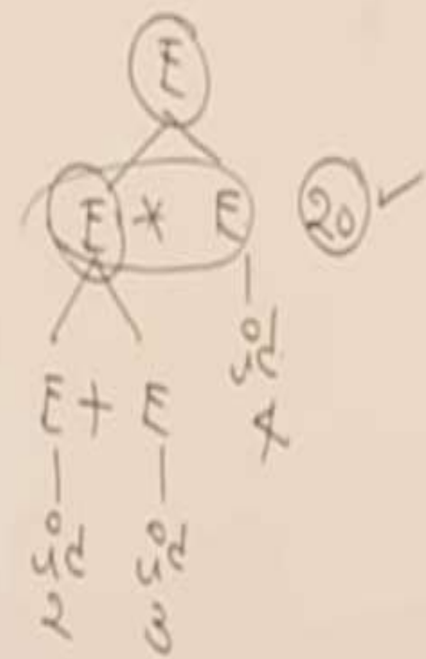
D

$E \Rightarrow E*E$
$\Rightarrow \underline{E+E}*E$
$\Rightarrow id+E*E$
$\Rightarrow id+id*E$
$\Rightarrow id+id*id$

2+3*4

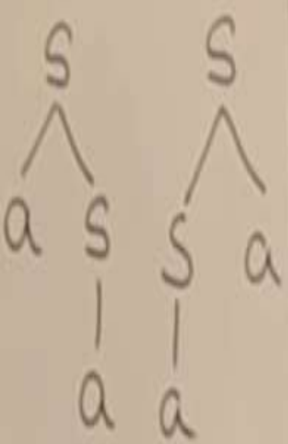


RmD: $E \Rightarrow E*E$
$\Rightarrow E*id$
$\Rightarrow E+E*id$
$\Rightarrow E+id*id$
$\Rightarrow id+id*id$



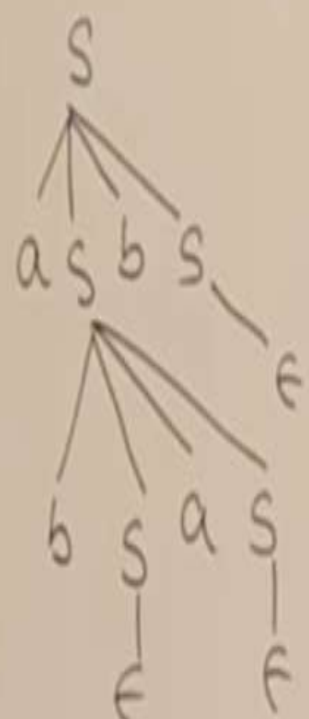
S → aS/Sa/a

w= aa



S → aSbS/bSaS/ϵ

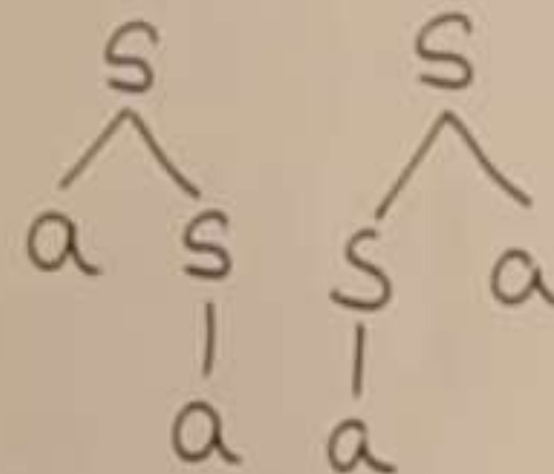
w= abab



R → R+R/RR/R*/a/b/c

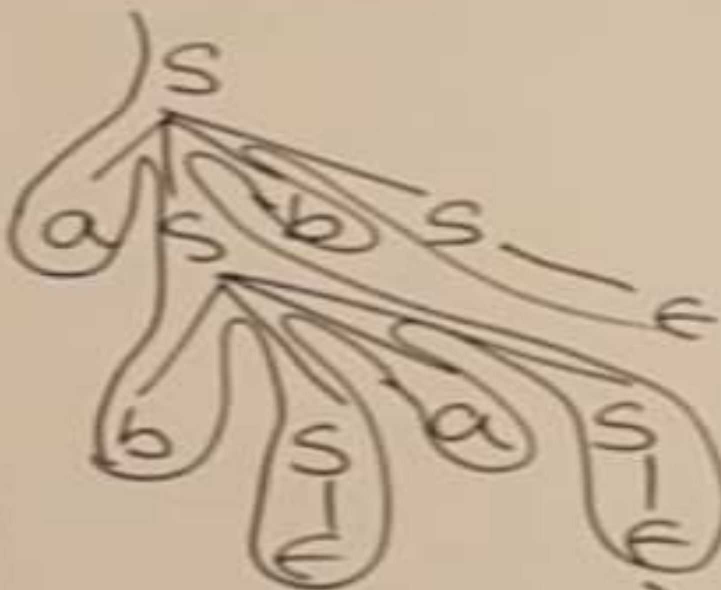
✓S → aS/Sa/a

w= aa

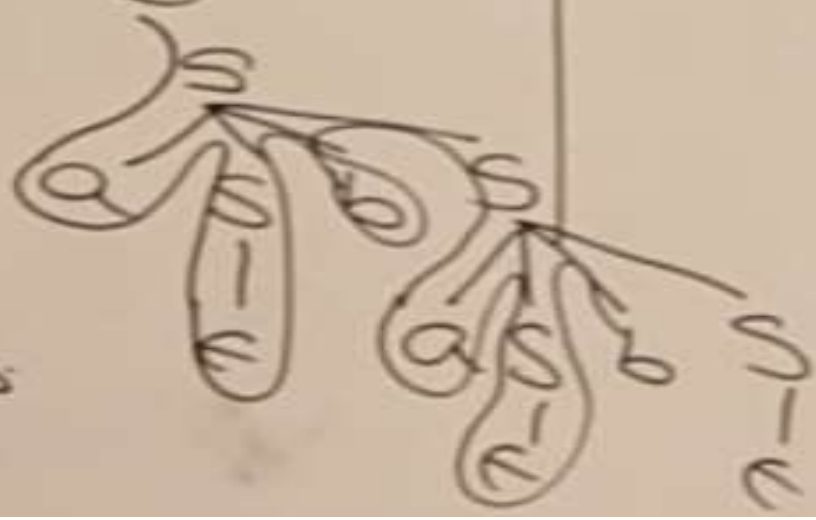


$S \to aSbS / bSaS / \epsilon$

$\omega = abab$



abab

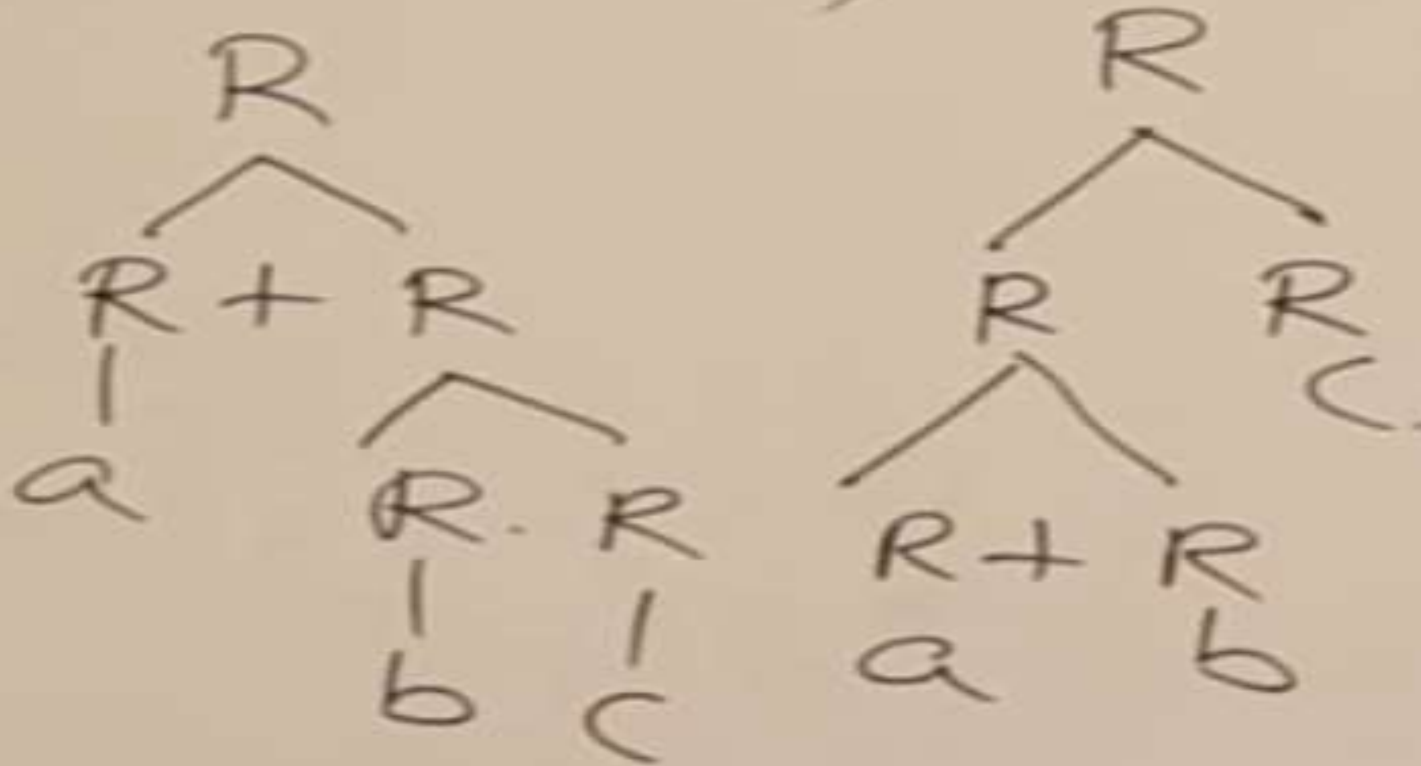


abab

$R \to R+R / RR / R^* / ($

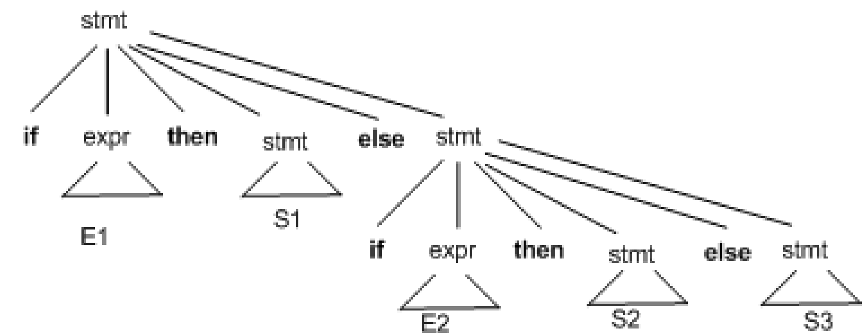
$R \rightarrow R+R \,/\, RR \,/\, R^* \,/\, a \,/\, b \,/\, c$

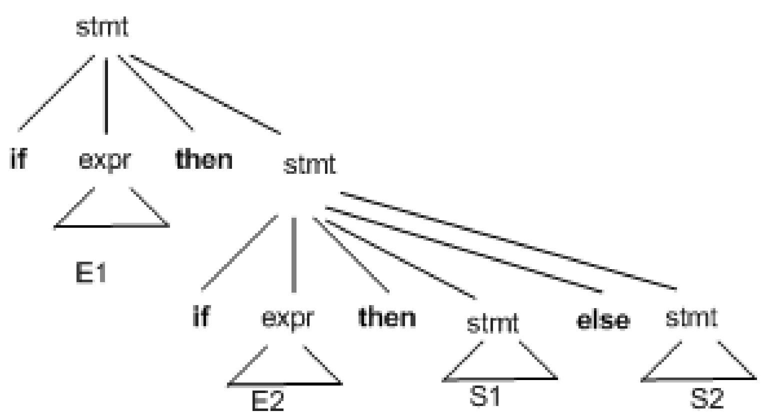
$a+(bc)$

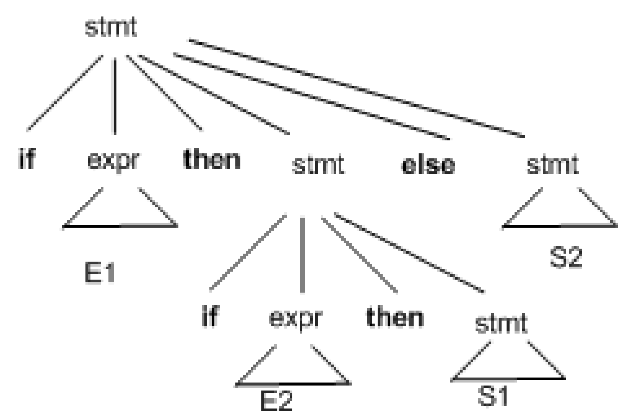


Elimination of ambiguity

stmt $\longrightarrow$ **If** expr **then** stmt
| **If** expr **then** stmt **else** stmt
| **other**







Elimination of ambiguity (cont.)

- Idea:
  - A statement appearing between a **then** and an **else** must be matched

stmt → matched_stmt
  | open_stmt

matched_stmt → **If** expr **then** matched_stmt **else** matched_stmt
  | **other**

open_stmt → **If** expr **then** stmt
  | **If** expr **then** matched_stmt **else** open_stmt

REMOVING AMBIGUITY

Take Original Grammar:

stmt → **if** ***expr*** **then** ***stmt***
  | **if** ***expr*** **then** ***stmt*** **else** ***stmt***
  | **other** **(any other statement)**

Rule: Match each **else** with the closest previous unmatched **then**.

Revise to remove ambiguity:

stmt → ***matched_stmt*** | ***unmatched_stmt***

matched_stmt → **if** ***expr*** **then** ***matched_stmt*** **else** ***matched_stmt*** | **other**

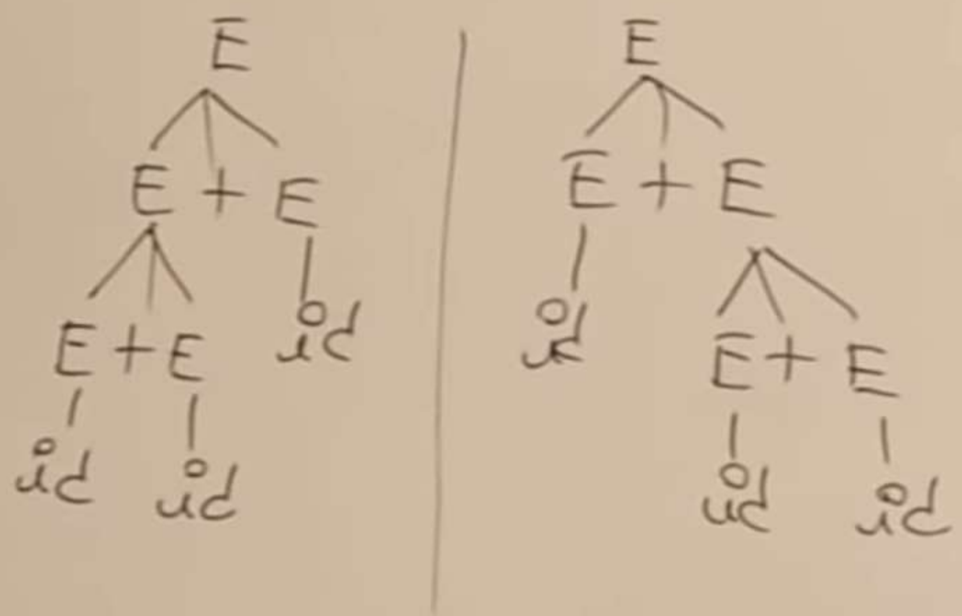
unmatched_stmt → **if** ***expr*** **then** ***stmt***
  | **if** ***expr*** **then** ***matched_stmt*** **else** ***unmatched_stmt***

$E \rightarrow E + E$

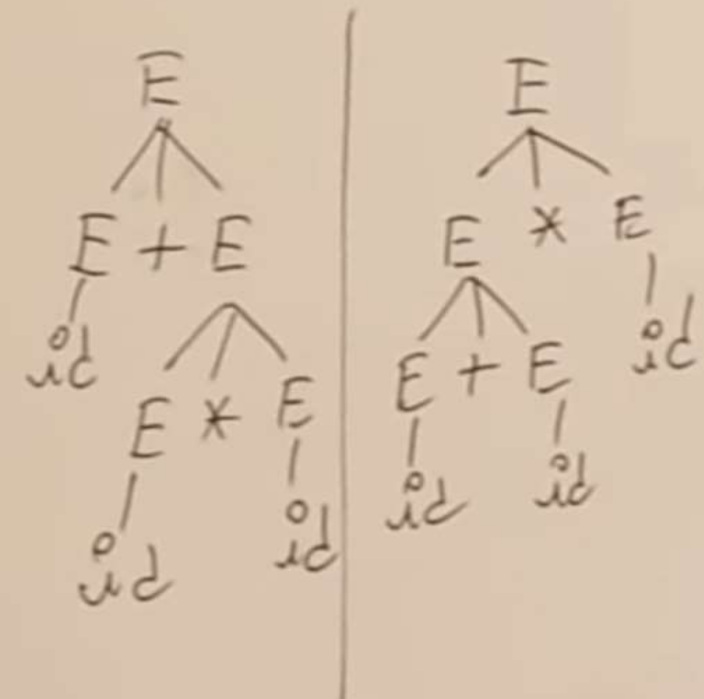
$\quad / E * E$

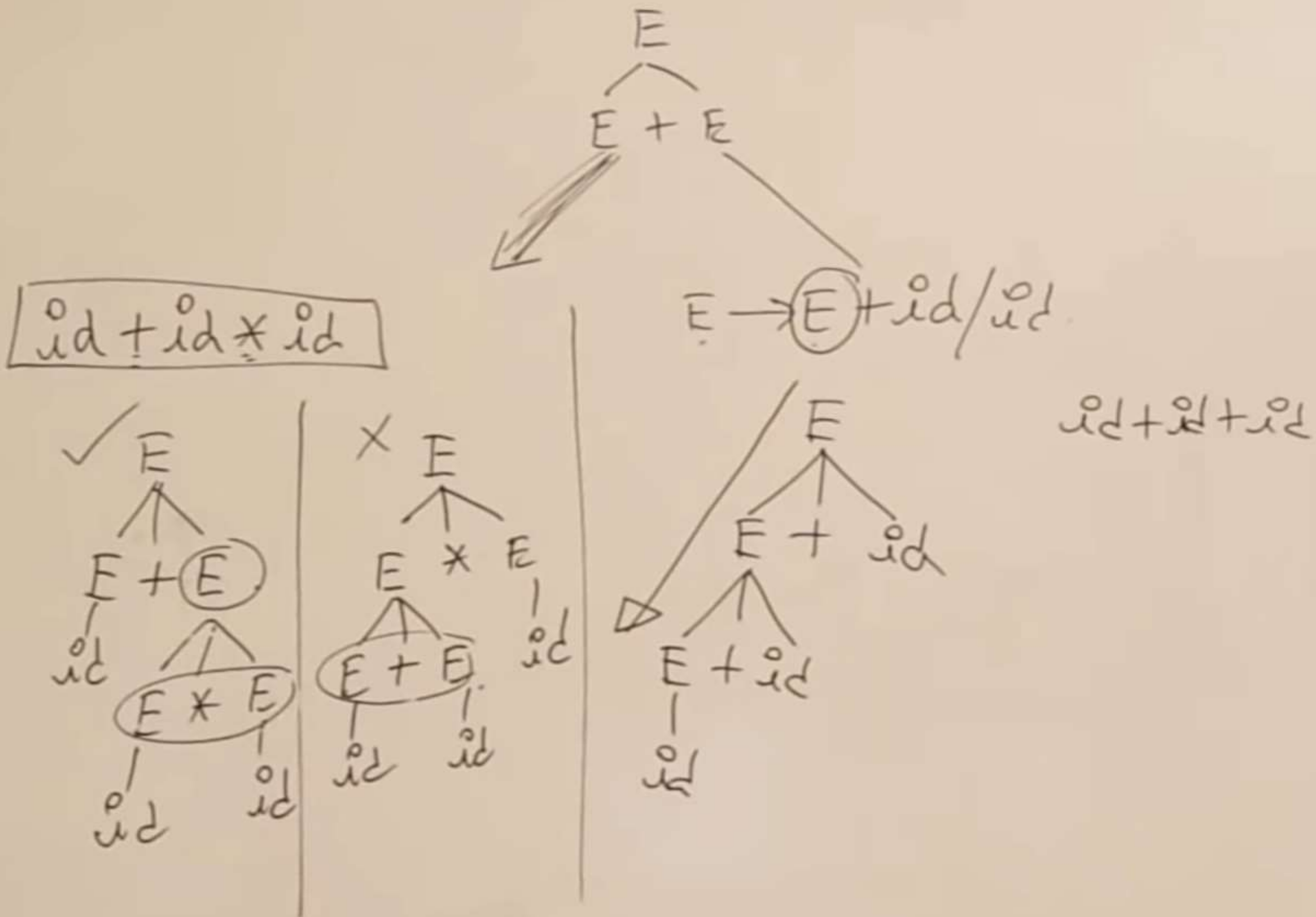
$\quad / id$

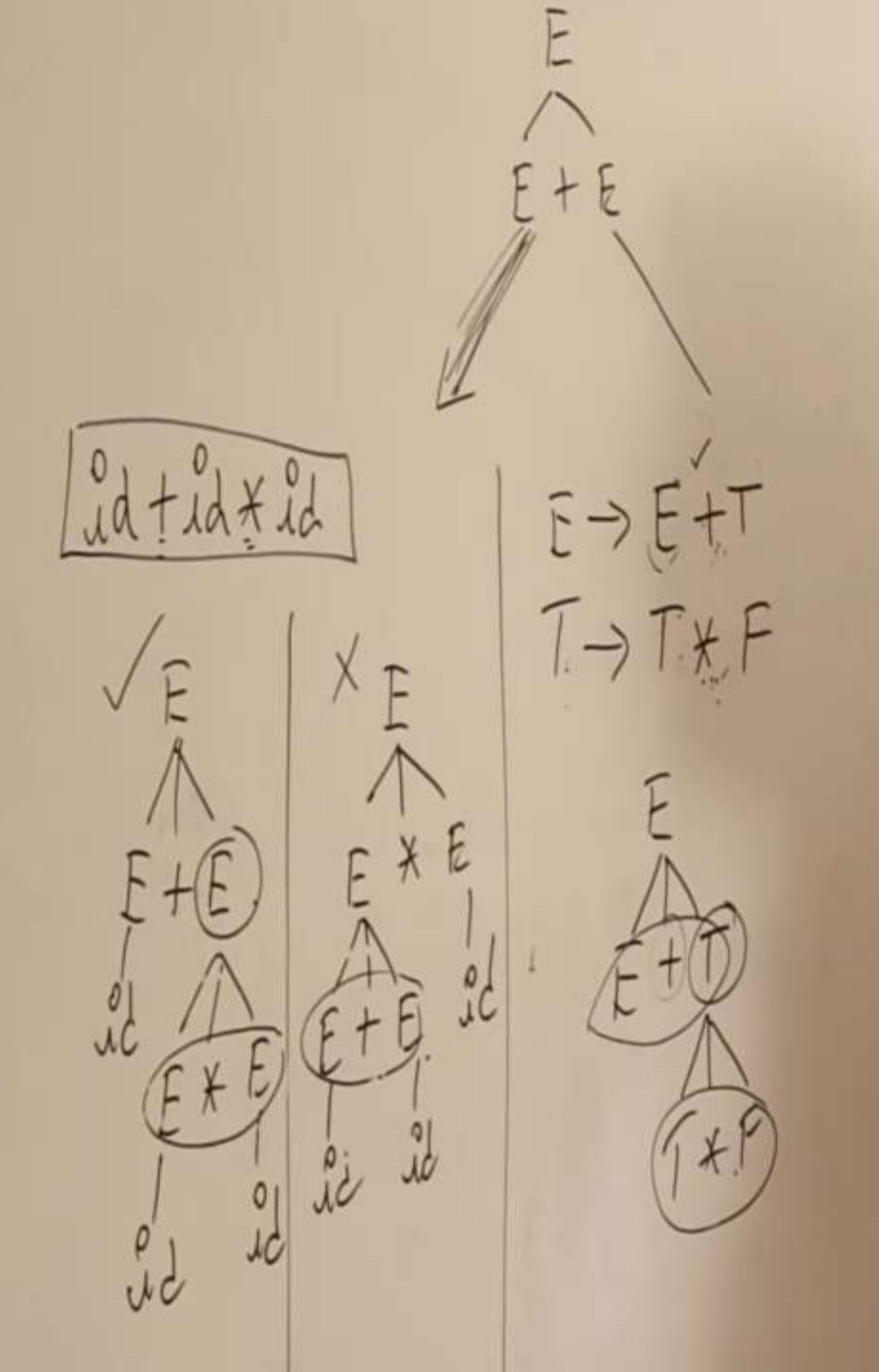
$\boxed{id + id + id}$

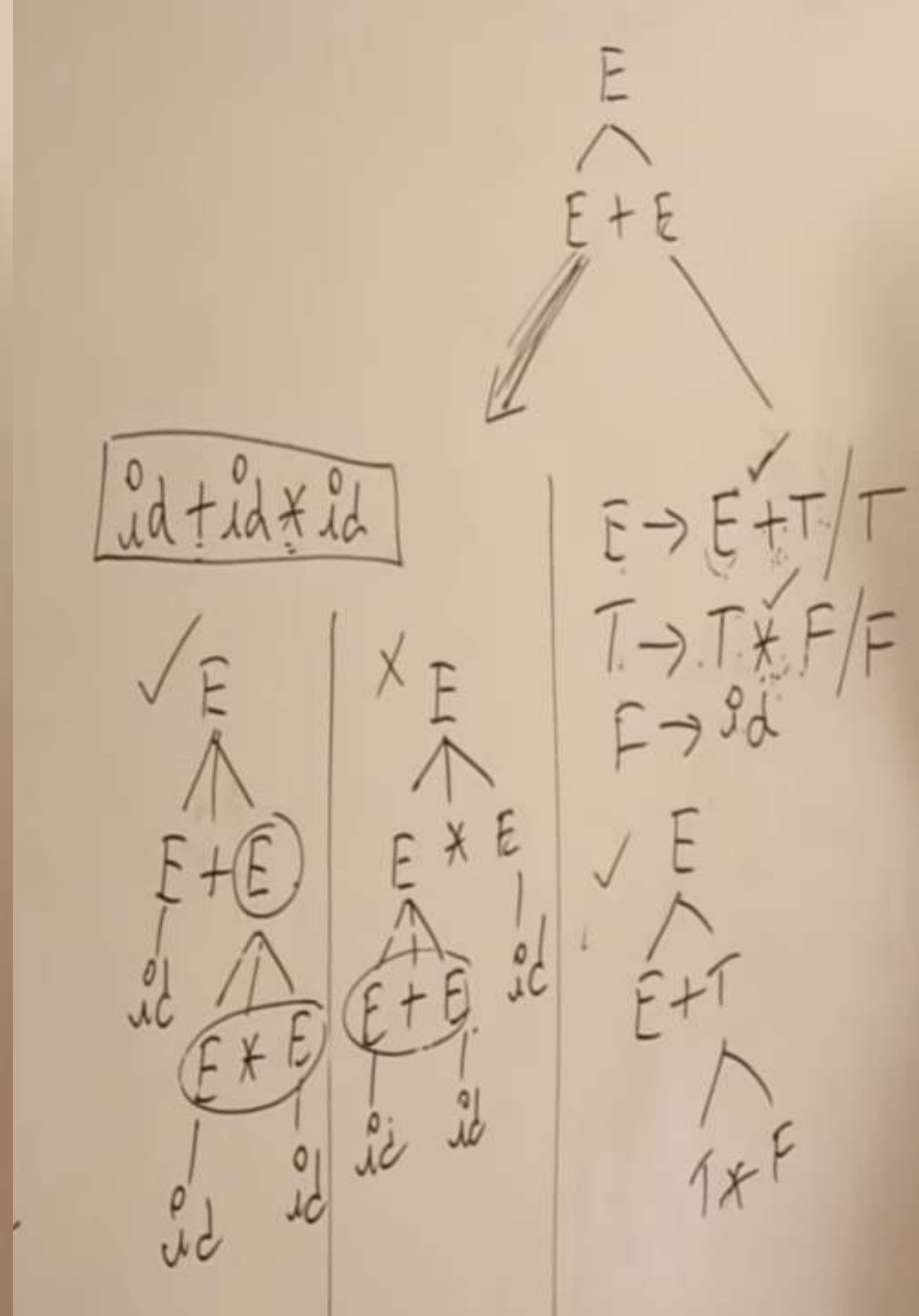


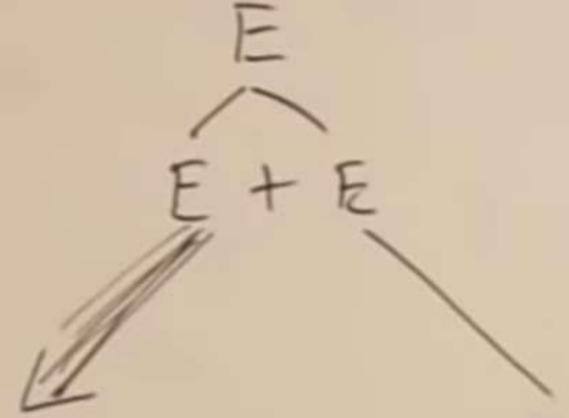
$\boxed{id + id * id}$



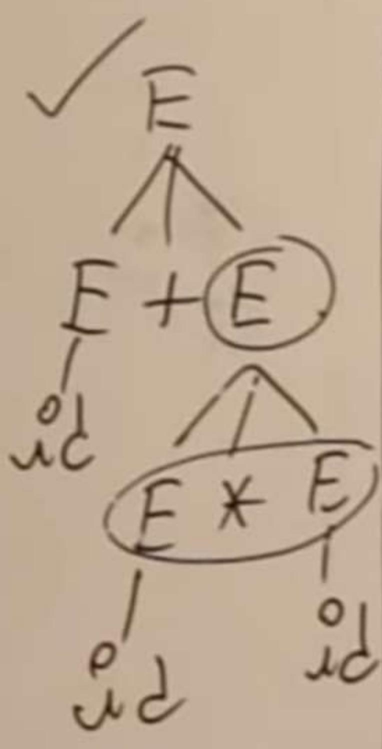

id + id * id
E → E + id / id
id + id + id

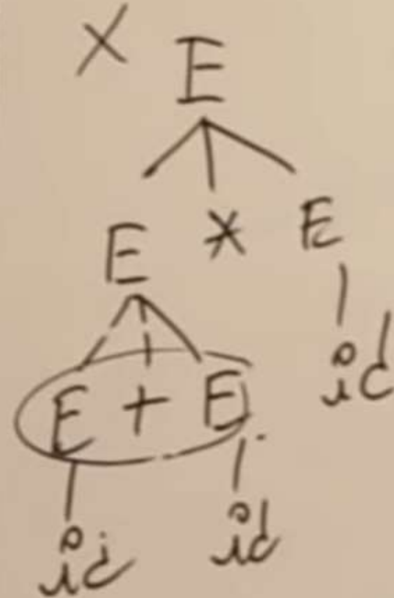

E
E + E
id+id*id
E -> E+T
T -> T*F


E
E + E
id+id*id
E -> E+T/T
T -> T*F/F
F -> id



$\boxed{id + id * id}$





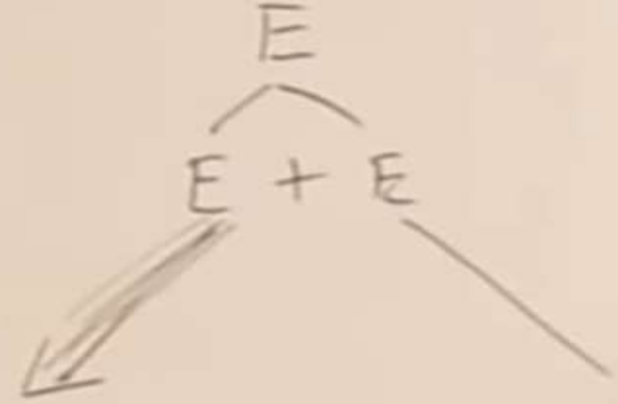
$E \rightarrow E + T \,/\, T$

$T \rightarrow T * F \,/\, F$

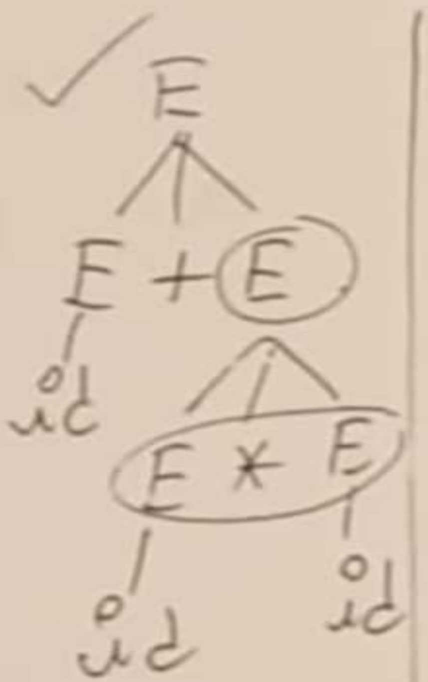
$F \rightarrow id$

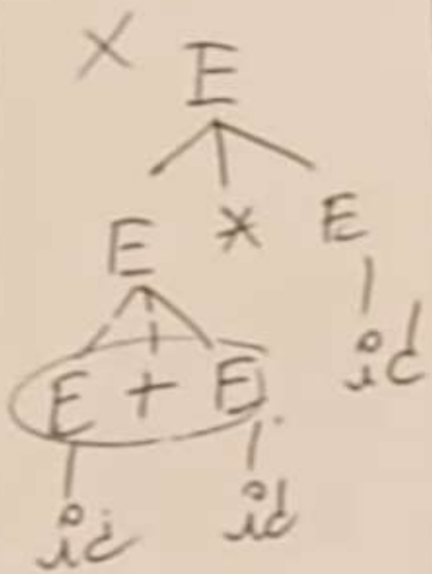
$2 \uparrow 3 \uparrow 2.$

$2^{3^{2}}$



$\boxed{id + id * id}$





$E \rightarrow E + T \,/\, T$

$T \rightarrow T * F \,/\, F$

$F \rightarrow G \uparrow F \,/\, G$

$G \rightarrow id$

$+, *, \uparrow$

$R \to R+R$

$/ RR$

$/ R^*$

$/ a$

$/ b$

$/ c$

$E \to E + T / T$

$T \to TF / F$

$F \to F^* / a / b / c.$

$bExp \rightarrow bExp \ \& \ bExp$
$\quad / \ bExp \text{ and } bExp$
$\quad / \text{ not } bExp$
$\quad / \text{ True}$
$\quad / \text{ False.}$

$E \rightarrow E \ \& \ F \ / \ F$

$F \rightarrow F \text{ and } G \ / \ G$

$G \rightarrow \text{Not } G \ / \text{ True } / \text{ False.}$

$A \rightarrow A \$ B / B$

$B \rightarrow B \# C / C$

$C \rightarrow C @ D / D$

$D \rightarrow d$

$\$ \gtrdot \$$

$\# \gtrdot \#$

$@ \gtrdot @$

$\$ \lessdot \# \lessdot @$

$E \rightarrow E * F$

$/ F + E$

$/ F$

$F \rightarrow F - F$

$/ id$

$* \doteq +$

$* \gtrdot *$

$+ \lessdot +$

$\$ \gtrdot \$$

$\# \gtrdot \#$

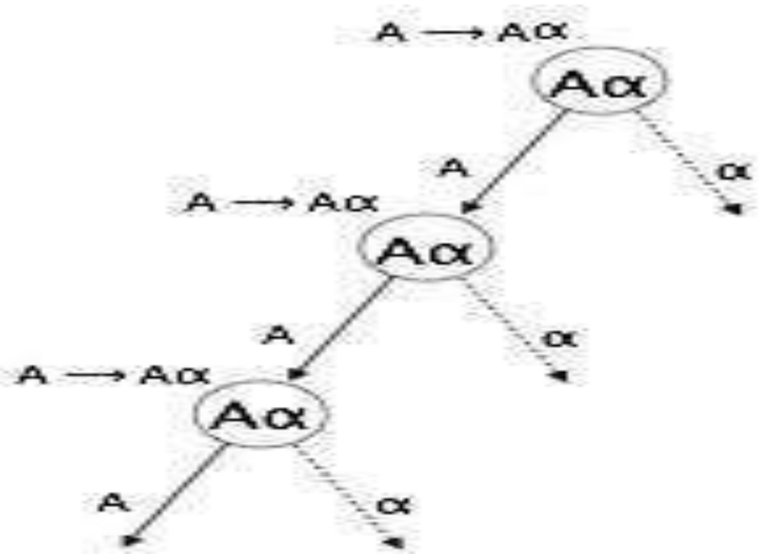
$@ \gtrdot @$

$\$ \lessdot \# \lessdot @$

Ambiguous Grammar	Unambiguous Grammar
A grammar is said to be ambiguous if for at least one string generated by it, it produces more than one- • parse tree • or derivation tree • or syntax tree • or leftmost derivation • or rightmost derivation	A grammar is said to be unambiguous if for all the strings generated by it, it produces exactly one- • parse tree • or derivation tree • or syntax tree • or leftmost derivation • or rightmost derivation
For ambiguous grammar, leftmost derivation and rightmost derivation represents different parse trees.	For unambiguous grammar, leftmost derivation and rightmost derivation represents the same parse tree.
Ambiguous grammar contains less number of non-terminals.	Unambiguous grammar contains more number of non-terminals.
For ambiguous grammar, length of parse tree is less.	For unambiguous grammar, length of parse tree is large.
Ambiguous grammar is faster than unambiguous grammar in the derivation of a tree. (Reason is above 2 points)	Unambiguous grammar is slower than ambiguous grammar in the derivation of a tree.
	Example- $E \rightarrow E + T / T$ $T \rightarrow T \times F / F$

Elimination of left recursion

- A grammar becomes left-recursive if it has any non-terminal ‘A’ whose derivation contains ‘A’ itself as the left-most symbol.
- Left-recursive grammar is considered to be a problematic situation for top-down parsers. Top-down parsers start parsing from the Start symbol, which in itself is non-terminal.
- So, when the parser encounters the same non-terminal in its derivation, it becomes hard for it to judge when to stop parsing the left non-terminal and it goes into an infinite loop.
- A grammar is left recursive if it has a non-terminal A such that there is a derivation A=> Aα
- Top down parsing methods cant handle left-recursive grammars
- A simple rule for direct left recursion elimination:
  - For a rule like:
    - A -> A α|β
  - We may replace it with
    - A -> β A’
    - A’ -> α A’ | ε



Left recursion elimination (cont.)

- There are cases like following
  - S -> Aa | b
  - A -> Ac | Sd | ε
- Left recursion elimination algorithm:
  - Arrange the non terminals in some order A1,A2,…,An.
  - For (each i from 1 to n) {
    - For (each j from 1 to i-1) {
      - Replace each production of the form Ai-> Aj γ by the production Ai -> $\delta 1 \gamma$ | $\delta 2 \gamma$ | … | $\delta k \gamma$ where Aj-> $\delta 1$ | $\delta 2$ | … | δk are all current Aj productions
      - }
      - Eliminate left recursion among the Ai-productions
    - }

RESOLVING DIFFICULTIES : LEFT RECURSION

A left recursive grammar has rules that support the derivation :
Top-Down parsing can't reconcile this type of grammar,
since it could consistently make choice which wouldn't allow termination.

This does not impact the strings derived from the grammar, but it removes immediate left recursion.

Take left recursive grammar:

$\mathbf{A} \Rightarrow \mathbf{A}\alpha \Rightarrow \mathbf{A}\alpha\alpha \Rightarrow \mathbf{A}\alpha\alpha\alpha$ **… etc.**

$\mathbf{A} \rightarrow \mathbf{A}\alpha \mid \beta$

$\mathbf{A} \rightarrow \mathbf{A}\alpha \mid \beta$

To the following:

$\mathbf{A} \rightarrow \beta \mathbf{A'}$

$\mathbf{A'} \rightarrow \alpha\, \mathbf{A'} \mid \in$

WHY IS LEFT RECURSION A PROBLEM ?

Consider:

$E \rightarrow E + T \quad | \quad T$

$T \rightarrow T * F \quad | \quad F \quad F$

$\rightarrow (E) \quad |$

id

Derive : id + id + id

$E \Rightarrow E + T \Rightarrow$

How can left recursion be removed ?

$E \rightarrow E + T \mid T$

What does this generate?

$E \Rightarrow E + T \Rightarrow T + T$

$E \Rightarrow E + T \Rightarrow E + T + T \Rightarrow T + T + T$

$\ldots$

How does this build strings ?

What does each string have to start with ?

RESOLVING DIFFICULTIES : LEFT RECURSION (2)

Informal Discussion:

Take all productions for <u>A</u> and order as:

$\mathbf{A \rightarrow A\alpha_1 \mid A\alpha_2 \mid \ldots \mid A\alpha_m \mid \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n}$

Where no β_i begins with A.

Now apply concepts of previous slide: A

$\mathbf{\rightarrow \beta_1 A' \mid \beta_2 A' \mid \ldots \mid \beta_n A'}$

$\mathbf{A' \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \ldots \mid \alpha_m A' \mid \in}$

$\mathbf{A \rightarrow A\alpha \mid \beta}$

To the following:

$\mathbf{A \rightarrow \beta A'}$

$\mathbf{A' \rightarrow \alpha A' \mid \in}$

$\mathbf{\rightarrow \beta_1 A' \mid \beta_2 A' \mid \ldots \mid \beta_n A'}$

$\mathbf{A' \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \ldots \mid \alpha_m A' \mid \in}$

For our example:

$\mathbf{E \rightarrow E + T \mid T} \longrightarrow \begin{cases} \mathbf{E \rightarrow TE'} \\ \mathbf{E' \rightarrow + TE' \mid \in} \end{cases}$

$\mathbf{T \rightarrow T * F \mid F} \longrightarrow \begin{cases} \mathbf{T \rightarrow FT'} \\ \mathbf{T' \rightarrow * FT' \mid \in} \end{cases}$

$\mathbf{F \rightarrow (E) \mid id} \longrightarrow \mathbf{F \rightarrow (E) \mid id}$

RESOLVING DIFFICULTIES : LEFT RECURSION (3)

Problem: If left recursion is two-or-more levels deep, this isn't enough

$$\left.\begin{array}{l} S \rightarrow Aa \mid b \\ A \rightarrow Ac \mid Sd \mid \in \end{array}\right\} \quad S \Rightarrow Aa \Rightarrow Sda$$

Algorithm:

Input: **Grammar G with ordered Non-Terminals $A_1, ..., A_n$**

Output: **An equivalent grammar with no left recursion**

1. **Arrange the non-terminals in some order A_1=start NT,A_2,…A_n**
2. **for *i* := 1 to *n* do begin**

   for *j* := 1 to *i* - 1 do begin

   replace each production of the form $A_i \rightarrow A_j\gamma$

   by the productions $A_i \rightarrow \delta_1\gamma \mid \delta_2\gamma \mid \dots \mid \delta_k\gamma$

   where $A_j \rightarrow \delta_1 \mid \delta_2 \mid \dots \mid \delta_k$ are all current A_j productions;

   end

   eliminate the immediate left recursion among A_i productions

USING THE ALGORITHM

Apply the algorithm to: $A_1 \to A_2 a \mid b \mid \in$

$A_2 \to A_2 c \mid A_1 d$

i = 1

For A_1 there is no left recursion

i = 2

for j=1 to 1 do

Take productions: $A_2 \to A_1 \gamma$ **and replace with**

$A_2 \to \delta_1 \gamma \mid \delta_2 \gamma \mid \ldots \mid \delta_k \gamma \mid$ **where** $A_1 \to \delta_1 \mid \delta_2 \mid \ldots \mid \delta_k$ **are A_1 productions**

in our case $A_2 \to A_1 d$ **becomes** $A_2 \to A_2 ad \mid bd \mid d$

What's left: $A_1 \to A_2 a \mid b \mid \in$

$A_2 \to A_2 c \mid A_2 ad \mid bd \mid d$

Are we done ?

USING THE ALGORITHM (2)

No ! We must still remove A_2 left recursion !

$$A_1 \to A_2a \mid b \mid \in$$

$$A_2 \to A_2c \mid A_2ad \mid bd \mid d$$

Recall:

$$A \to A\alpha_1 \mid A\alpha_2 \mid \ldots \mid A\alpha_m \mid \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n$$

$$A \to \beta_1A' \mid \beta_2A' \mid \ldots \mid \beta_nA'$$

$$A' \to \alpha_1A' \mid \alpha_2A' \mid \ldots \mid \alpha_m A' \mid \in$$

$$A_1 \to A_2a \mid b \mid \in$$

$$A_2 \to bdA_2' \mid dA_2'$$

$$A_2' \to c\,A_2' \mid adA_2' \mid \in$$

Apply to above case. What do you get ?

REMOVING DIFFICULTIES : $\in$-MOVES

Transformation: **In order to remove $A \rightarrow \in$ find all rules of the form $B \rightarrow uAv$ and *add* the rule $B \rightarrow uv$ to the grammar G.** **Why does this work ?**

Examples:

$E \rightarrow TE'$

$T_{\in} \rightarrow^{E'} F \rightarrow T'^{+} TE' \mid$

$F_{\in}{}^{T} \rightarrow' \rightarrow (E^{*})^{FT} \mid i' d \mid$

A is Grammar $\in$-free if:

1. It has no $\in$ -production or
2. There is exactly one $\in$ -production

$S \rightarrow \in$ and then the start symbol S does not appear on the right side of any production.

$A_1 \rightarrow A_2 a \mid b$

$A_2 \rightarrow bd A_2' \mid A_2'$

$A_2' \rightarrow c A_2' \mid bd A_2' \mid \in$

$A_1 \rightarrow A_2 a \mid b \mid \in$

$A_2 \rightarrow A_2 c \mid A_2 ad \mid bd \mid d$

REMOVING DIFFICULTIES : CYCLES

How would cycles be removed ?

Make sure every production is adding some terminal(s) (except a single $\in$ -production in the start NT)…

e.g.

$S \rightarrow SS \mid (S) \mid \in$

Has a cycle: $S \Rightarrow \underbrace{SS \Rightarrow S}_{S \rightarrow \in}$

Transform to:

$S \rightarrow S (S) \mid (S) \mid \in$

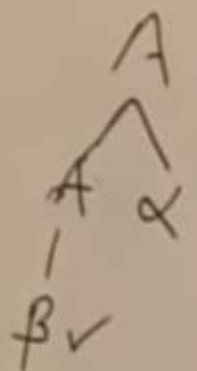
Re

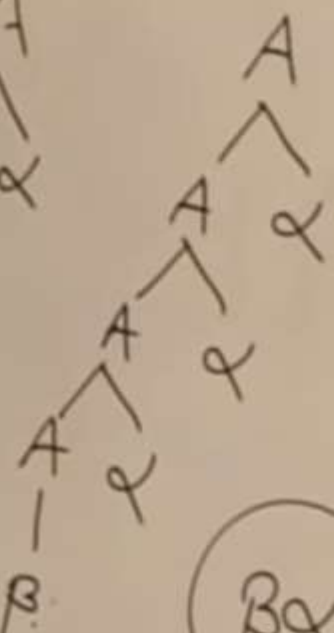
LR

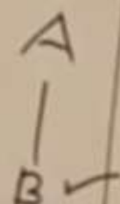
RR

```
A()
{
  A()
  α.
}
```

$A \to A\alpha / \beta$



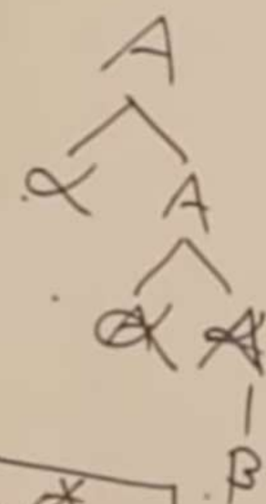


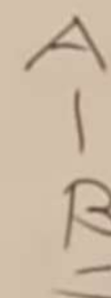


$\beta\alpha^*$

$A \to \alpha A / \beta$

```
A()
{
  α
  A()
}
```





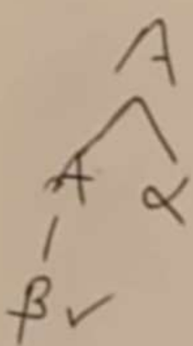
$\boxed{\alpha^*\beta.}$

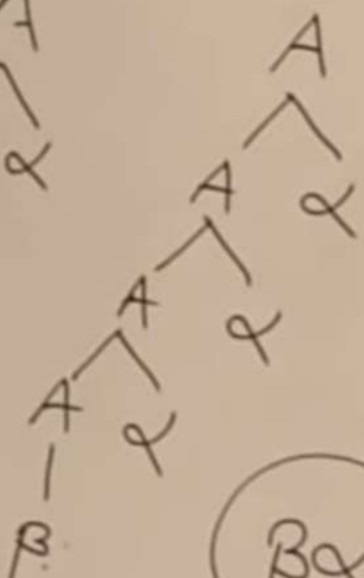
A()
{
A()
α
}

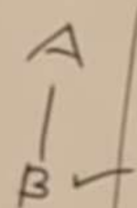
Re

LR

$A \rightarrow A\alpha / \beta$





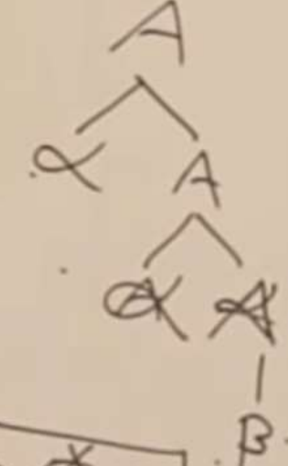


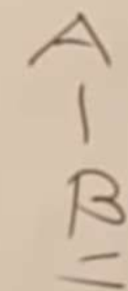
$\beta\alpha^*$

RR

$A \rightarrow \alpha A / \beta$

A()
{
α
A()
}





$\boxed{\alpha^*\beta}$

TDP

Re

RR

β

A

$\beta\alpha^{*}$

$A \rightarrow \beta\alpha^{*}$

$A \rightarrow \beta A'$

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

$\beta\alpha^{*}$

$A \rightarrow \beta\alpha^{*}$

$\boxed{\begin{array}{l} A \rightarrow \beta A' \\ A' \rightarrow \epsilon / \alpha A' \end{array}} \Longleftrightarrow \boxed{A \rightarrow \alpha A / \beta}$

TOP

Re

x^{RR}

$$\underbrace{E}_{A} \to \underbrace{E}_{A}\underbrace{+T}_{\alpha} / \underbrace{T}_{\beta}$$

$\beta\alpha^*$

$$A \to \beta\alpha^*$$

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

$$A \to \beta A'$$
$$A' \to \epsilon / \alpha A'$$
$$\Longleftrightarrow \quad A \to A\alpha / \beta$$

TOP

x^{RR}

$$\underbrace{S}_{A} \to \underbrace{S}_{A}\underbrace{0S1S}_{\alpha} / \underbrace{01}_{\beta}.$$

$$S \to 01S'$$

$$S' \to \epsilon / 0S1SS'$$

$$\beta\alpha^*$$

$$A \to \beta\alpha^*$$

$$\boxed{\begin{aligned} A &\to \beta A' \\ A' &\to \epsilon / \alpha A' \end{aligned}} \iff \boxed{A \to A\alpha / \beta}$$

$S \to (L) / x$

$\underset{A}{L} \to \underset{\alpha}{L, S} / \underset{\beta}{S}$

$A \to A\alpha / \beta$

$\Downarrow$

$A \to \beta A'$

$A' \to \alpha A' / \epsilon$

$\boxed{\begin{array}{l} L \to S L' \\ L' \to , S L' / \epsilon \end{array}}$

Example

The production set

$S \Rightarrow A\alpha \mid \beta$ $A \Rightarrow Sd$

after applying the above algorithm, should become

$S \Rightarrow A\alpha \mid \beta$ $A \Rightarrow A\alpha d \mid \beta d$

and then, remove immediate left recursion using the first technique.

$A \Rightarrow \beta dA'$ $A' \Rightarrow \alpha dA' \mid \varepsilon$

Now none of the production has either direct or indirect left recursion.

Left factoring

- If more than one grammar production rules has a common prefix string, then the top-down parser cannot make a choice as to which of the production it should take to parse the string in hand
- Left factoring is a grammar transformation that is useful for producing a grammar suitable for predictive or top-down parsing.
- Then it cannot determine which production to follow to parse the string as both productions are starting from the same terminal (or non-terminal). To remove this confusion,.
- Left factoring transforms the grammar to make it useful for top-down parsers. In this technique, we make one production for each common prefixes and the rest of the derivation is added by new productions.

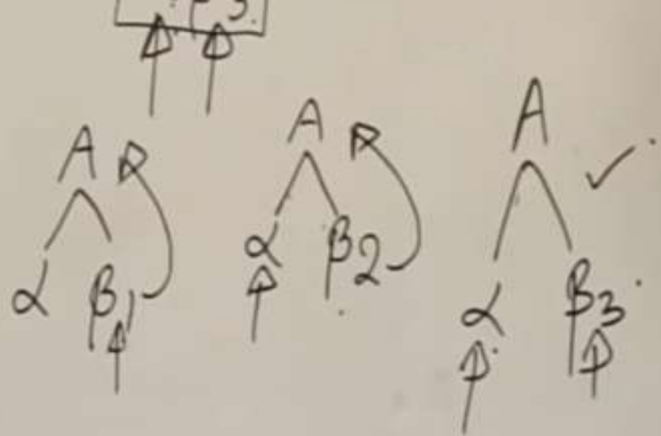
- Consider following grammar:
  - Stmt -> **if** expr **then** stmt **else** stmt
  - | **if** expr **then** stmt
- On seeing input **if** it is not clear for the parser which production to use
- We can easily perform left factoring:
  - If we have A->αβ1 | αβ2 then we replace it with
    - A -> αA'
    - A' -> β1 | β2

Left factoring (cont.)

- Algorithm
  - For each non-terminal A, find the longest prefix α common to two or more of its alternatives. If $\alpha <> \varepsilon$, then replace all of A-productions $A \rightarrow \alpha\beta 1 \mid \alpha\beta 2 \mid \dots \mid \alpha\beta n \mid \gamma$ by
    - $A \rightarrow \alpha A' \mid \gamma$
    - $A' \rightarrow \beta 1 \mid \beta 2 \mid \dots \mid \beta n$
- Example:
  - S -> I E t S | i E t S e S | a
  - E -> b
- We make one production for each common prefixes.
- The common prefix may be a terminal or a non-terminal or a combination of both.
- Rest of the derivation is added by new productions.

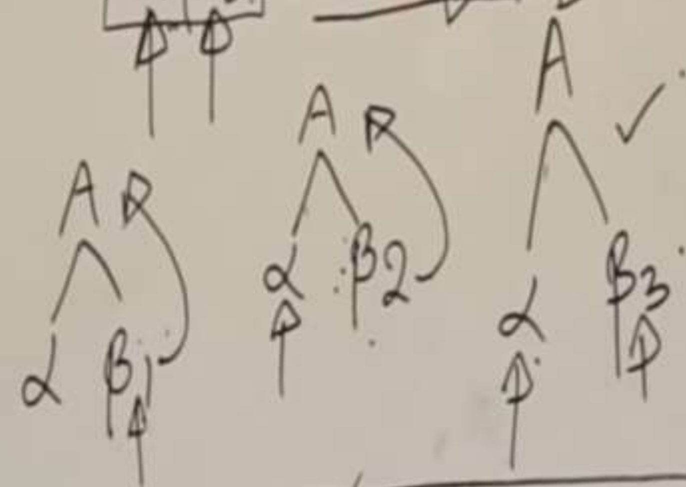
$A \to \alpha\beta_1 \,/\, \alpha\beta_2 \,/\, \alpha\beta_3$

$\boxed{\alpha\beta_3}$



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

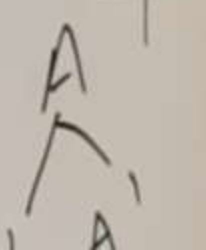
$\boxed{\alpha\beta_3}$ $\quad \alpha \ \beta_3$



$A \to \alpha A'$

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

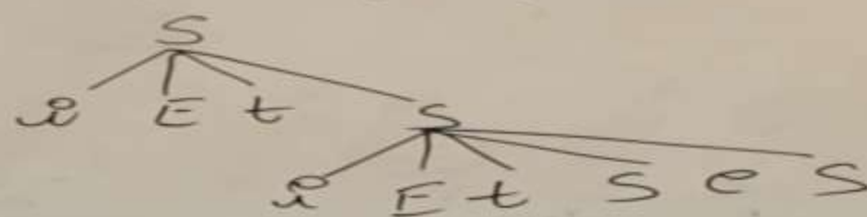
$\alpha\beta_3$

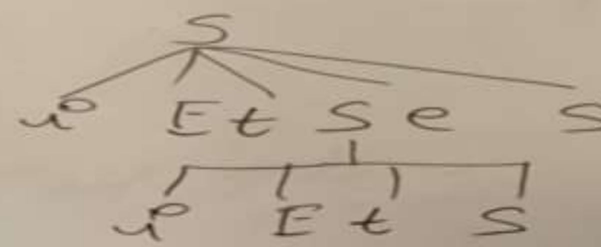


$S \rightarrow \underline{iEtS}$
$\quad | \underline{iEtS}eS$
$\quad | a$
$E \rightarrow b$

$S \rightarrow \underline{iEtS}S' | a$
$S' \rightarrow \epsilon | eS$
$E \rightarrow b$

iEtiEtSeS.

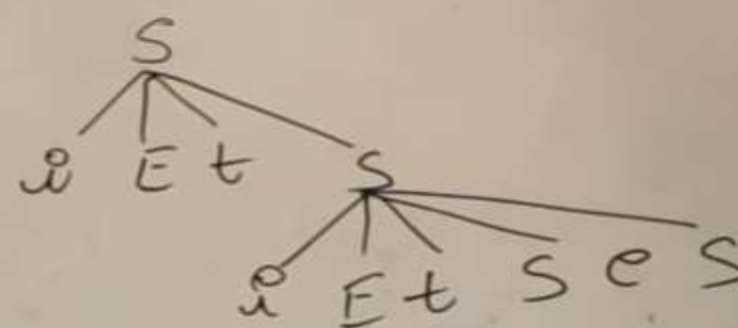




$S \to iEtS$
$/ iEtSeS$
$/ a$
$E \to b$

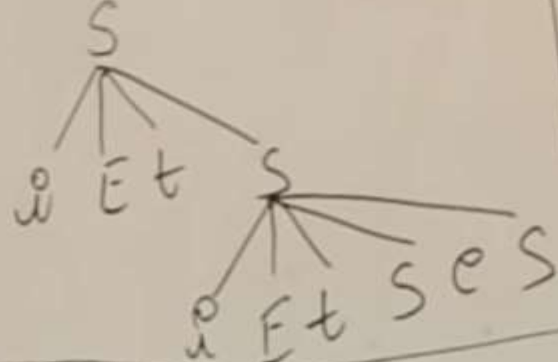
$S \to iEtSS' / a$
$S' \to \epsilon / eS$
$E \to b$

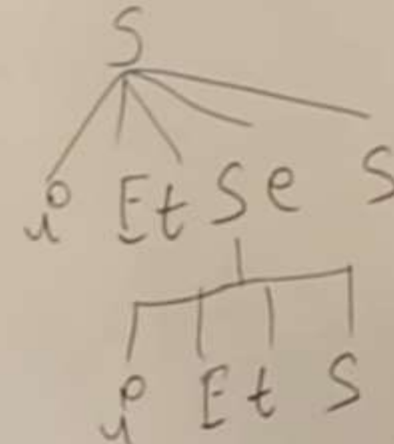
iEtiEtSeS.



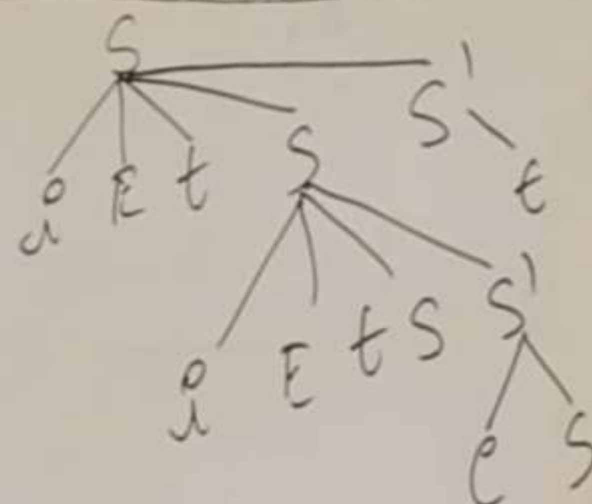
$S \to iEtS$
$\quad / iEtSeS$
$\quad / a$
$E \to b$

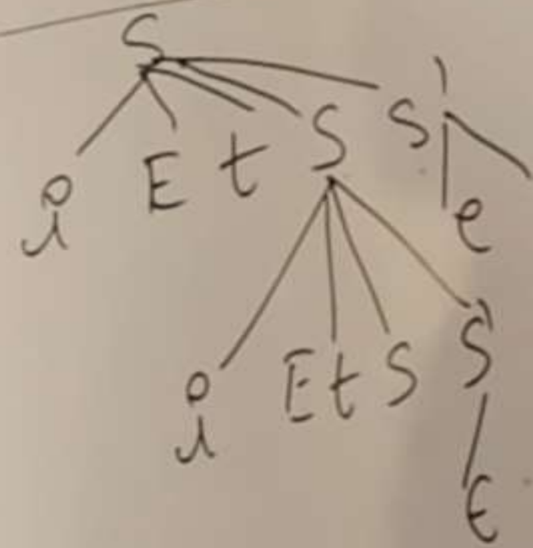
$iEtiEtSeS$





$S \to iEtSS' / a$
$S' \to \epsilon / eS$
$E \to b$





$$S \to \underline{a}SSbS$$
$$/\,\underline{a}SaSb$$
$$/\,\underline{a}bb$$
$$/\,b$$

$$S \to bSSaaS$$
$$/\,bSSaSb$$
$$/\,bSb$$
$$/\,a$$

EXAMPLES OF LEFT FACTORING

1. S -> iEtS|iEtSES|a E ->b
2. S-> aSSbS|aSaSb|abb|b
3. S-> bSSaaS|bSSaSb|bSb|a
4. A → aAB / aBc / aAc
5. S → a / ab / abc / abcd
6. S → aAd / aB
   A → a / ab
   B → ccd / ddc

CHECK GRAMMAR IS Ambiguous or not

1. S -> aS | Sa | a

2. $S \rightarrow aB / bA$

   $S \rightarrow aS / bAA / a$

   $B \rightarrow bS / aBB / b$

- Let us consider a string w = aaabbabbba
- Now, let us derive the string w using leftmost derivation.

• LEFT RECURSION

1. S->Sab/Scd/Sef/g/h
2. $A \rightarrow ABd$ / Aa / a

   $B \rightarrow Be$ / b

3. $E \rightarrow E + E$ / $E \times E$ / a
4. $E \rightarrow E + T$ / T

   $T \rightarrow T \times F$ / F

   $F \rightarrow id$

5. $S \rightarrow (L)$ / a

   $L \rightarrow L , S$ / S

6. $S \rightarrow S0S1S$ / 01
7. $S \rightarrow A$

   $A \rightarrow Ad$ / Ae / aB / ac

   $B \rightarrow bBc$ / f

8. $A \rightarrow AA\alpha$ / β

Left Recursion Example

1. $A \rightarrow Ba \,/\, Aa \,/\, c$

   $B \rightarrow Bb \,/\, Ab \,/\, d$

2. $X \rightarrow XSb \,/\, Sa \,/\, b$

   $S \rightarrow Sb \,/\, Xa \,/\, a$

3. $S \rightarrow Aa \,/\, b$

   $A \rightarrow Ac \,/\, Sd \,/\, \epsilon$