

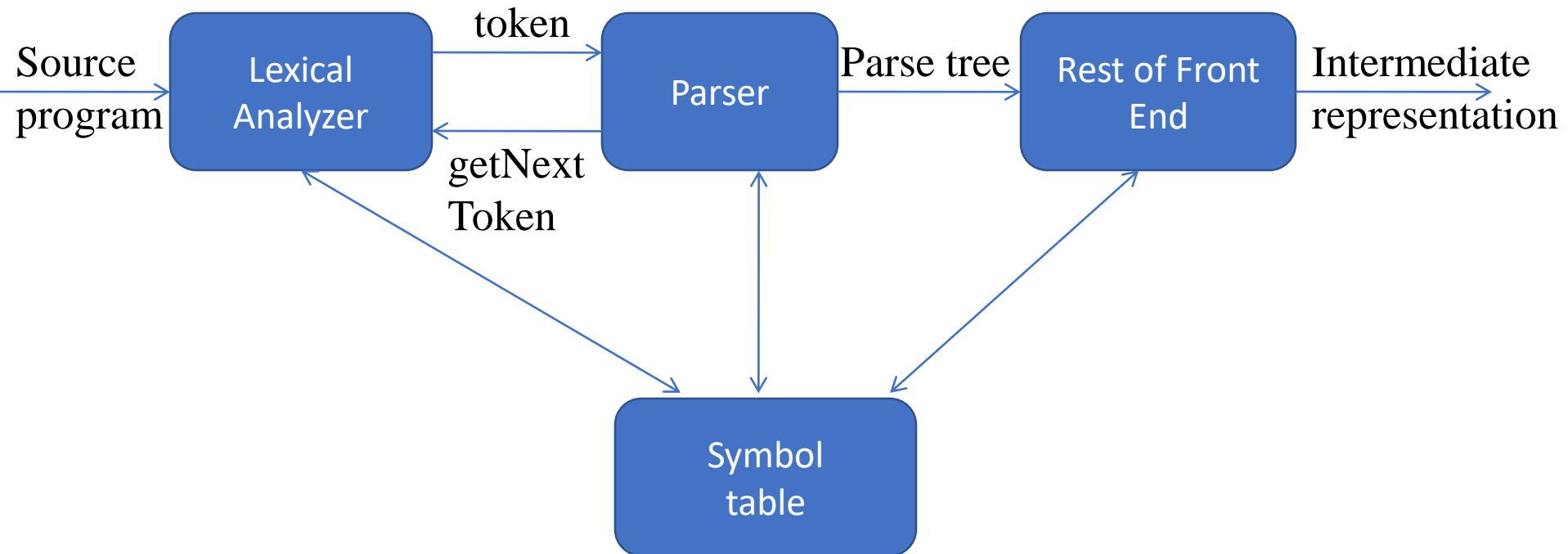
Compiler course

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 programs

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 \rightarrow expression + term

expression \rightarrow expression – term

expression \rightarrow term

term \rightarrow term * factor

term \rightarrow term / factor

term \rightarrow factor

factor \rightarrow (expression)

factor \rightarrow **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 Grammars

- Context-free grammars consist of:
 - Set of symbols:
 - **terminals** that denotes token types
 - **non-terminals** that denotes a set of strings
 - Start symbol
 - Rules:
 - $\text{symbol} ::= \text{symbol symbol} \dots \text{symbol}$
 - left-hand side: non-terminal
 - right-hand side: terminals and/or non-terminals
 - rules explain how to rewrite non-terminals (beginning with start symbol) into terminals

CONTEXT FREE GRAMMAR EXAMPLES

- ARITHMETIC EXPRESSIONS

$$\begin{aligned} E &::= T \mid E + T \mid E - T \\ T &::= F \mid T * F \mid T / F \mid F \\ &::= \text{id} \mid (E) \end{aligned}$$

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

- STATEMENTS

$$X \rightarrow Y_1 \dots Y_n$$

terminals

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

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

Context-Free Grammars

A string is in the language of the CFG if and only if it is possible to **derive** that string using the following non-deterministic procedure:

1. begin with the start symbol
2. while any non-terminals exist, pick a non-terminal and rewrite it using a rule **<== could be many choices here**
3. stop when all you have left are terminals (and check you arrived at the string you were hoping to)

Parsing is the process of checking that a string is in the CFG for your programming language. It is usually coupled with creating an abstract syntax tree.

Uses of grammars

$$E \rightarrow E + T \mid T$$
$$T \rightarrow T * F \mid F$$
$$F \rightarrow (E) \mid \mathbf{id}$$
$$E \rightarrow TE'$$
$$E' \rightarrow +TE' \mid \varepsilon$$
$$T \rightarrow FT'$$
$$T' \rightarrow *FT' \mid \varepsilon$$
$$F \rightarrow (E) \mid \mathbf{id}$$

Derivations

- Productions are treated as rewriting rules to generate a string
- Rightmost and leftmost derivations
 - $E \rightarrow E + E \mid E * E \mid -E \mid (E) \mid \mathbf{id}$
 - Derivations for $\mathbf{-(id+id)}$
 - $E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(E+E) \Rightarrow \mathbf{-(id+E)} \Rightarrow \mathbf{-(id+id)}$
- 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.

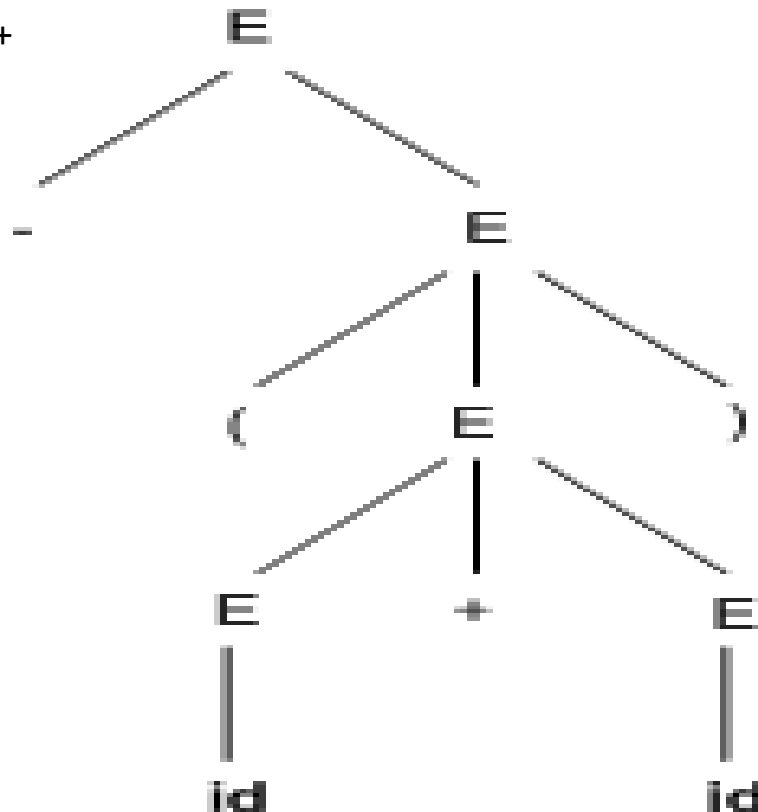
All leaf nodes are terminals.

All interior nodes are non-terminals.

In-order traversal gives original input string.

- **-(id+id)**

- $E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(E+E) \Rightarrow -(id+$

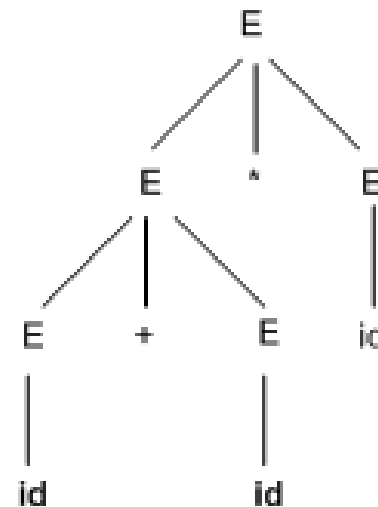
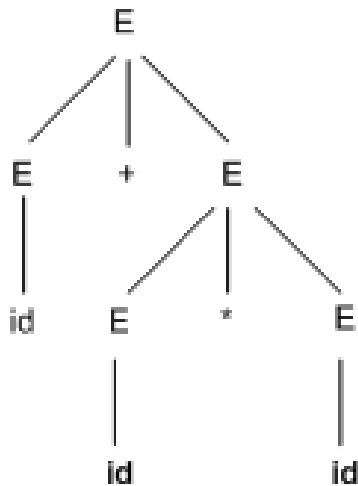


Parse Trees

- Representing derivations as trees
 - useful in compilers: Parse trees correspond quite closely (but not exactly) with abstract syntax trees we're trying to generate
 - difference: **abstract** syntax vs **concrete** (parse) syntax
- each internal node is labeled with a non-terminal
- each leaf node is labeled with a terminal
- each use of a rule in a derivation explains how to generate children in the parse tree from the parents

Ambiguity

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



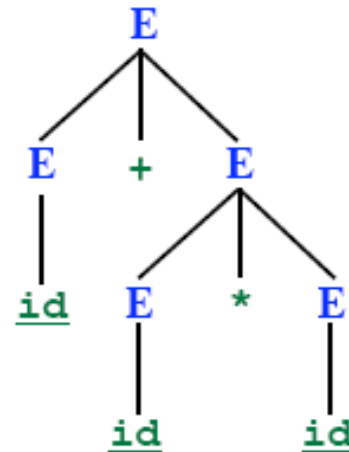
AMBIGUOUS GRAMMAR

1. $E \rightarrow E + E$
2. $\rightarrow E * E$
3. $\rightarrow (E)$
4. $\rightarrow - E$
5. $\rightarrow ID$

Input: $id+id*id$

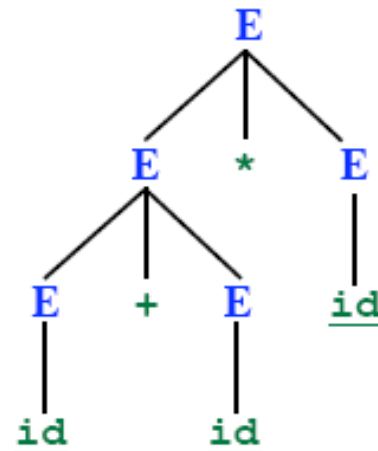
Leftmost Derivation #1

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



Leftmost Derivation #2

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



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 over others
 - Add disambiguating rule into the grammar

RESOLVING PROBLEMS: AMBIGUOUS GRAMMARS

Consider the following grammar segment:

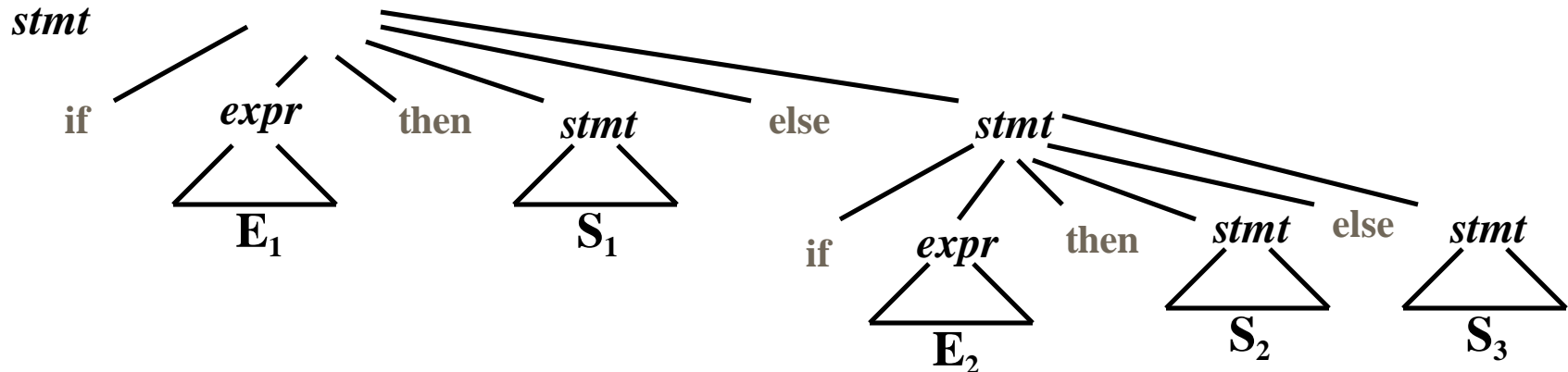
stmt \rightarrow **if** *expr* **then** *stmt*

| **if** *expr* **then** *stmt* **else** *stmt*

| **other** (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

Ambiguous Grammars

characters: 4 + 5 * 6

tokens: NUM(4) PLUS NUM(5) MULT NUM(6)

non-terminals:

E

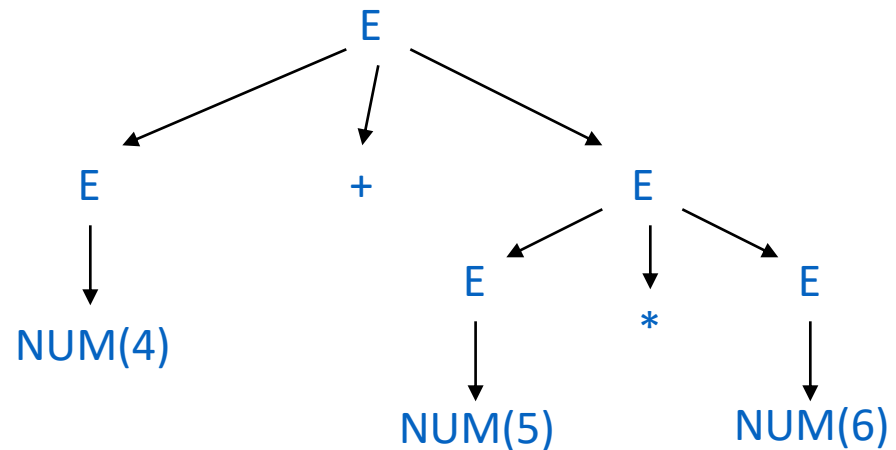
terminals:

ID

NUM

PLUS

MULT



E ::= ID

| NUM

| E + E

| E * E

I like using this notation where
I avoid repeating E ::=

Ambiguous Grammars

characters: 4 + 5 * 6

tokens: NUM(4) PLUS NUM(5) MULT NUM(6)

non-terminals:

E

terminals:

ID

NUM

PLUS

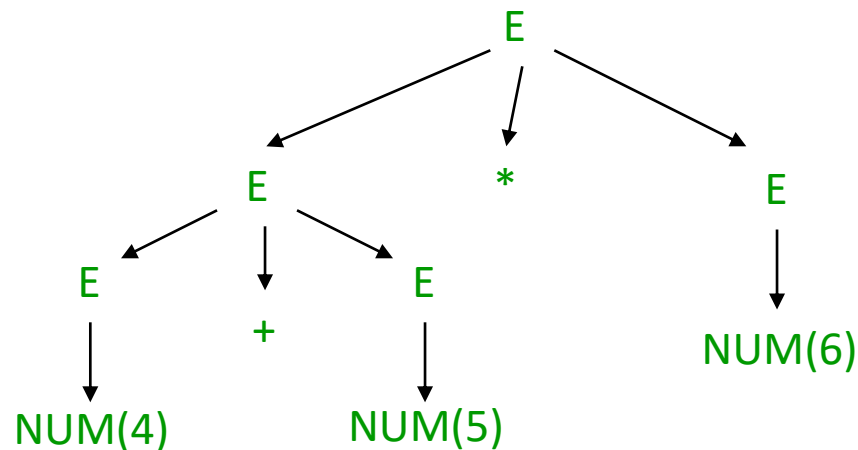
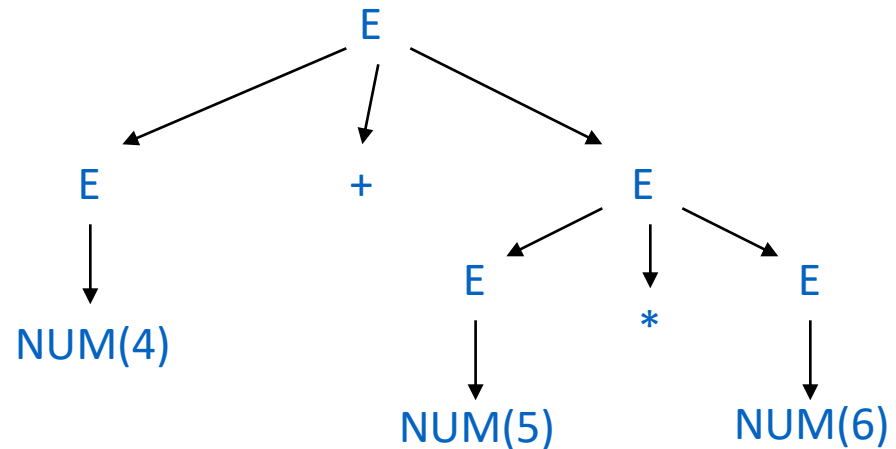
MULT

$E ::= ID$

$| NUM$

$| E + E$

$| E * E$



Ambiguous Grammars

- problem: compilers use parse trees to interpret the meaning of parsed expressions
 - different parse trees have different meanings
 - eg: $(4 + 5) * 6$ is not $4 + (5 * 6)$
 - languages with ambiguous grammars are **DISASTROUS**; The meaning of programs isn't well-defined! You can't tell what your program might do!
- solution: rewrite grammar to eliminate ambiguity
 - fold **precedence** rules into grammar to disambiguate
 - fold **associativity** rules into grammar to disambiguate
 - other tricks as well

MD

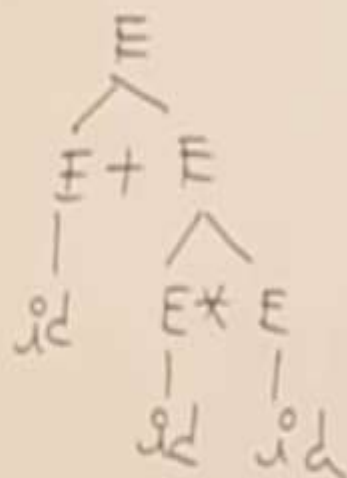
$$E \Rightarrow E * E$$

$$\Rightarrow E + E * E$$

$$\Rightarrow id + E * E$$

$$\Rightarrow id + id * E$$

$$\Rightarrow id + id * id$$



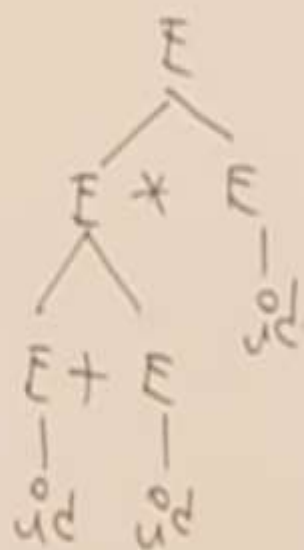
RMD: $E \Rightarrow E * E$

$$\Rightarrow E * id$$

$$\Rightarrow E + E * id$$

$$\Rightarrow E + id * id$$

$$\Rightarrow id + id * id$$



MD

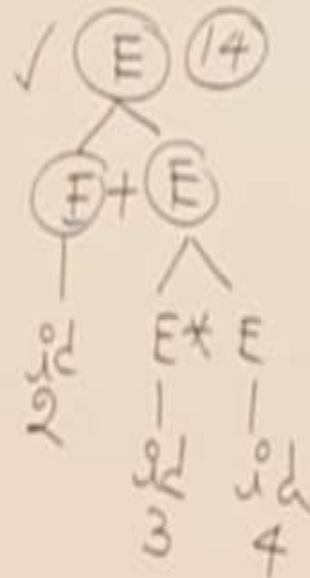
$$E \Rightarrow E * E$$

$$\Rightarrow E + E * E$$

$$\Rightarrow id + E * E$$

$$\Rightarrow id + id * E$$

$$\Rightarrow id + id * id$$



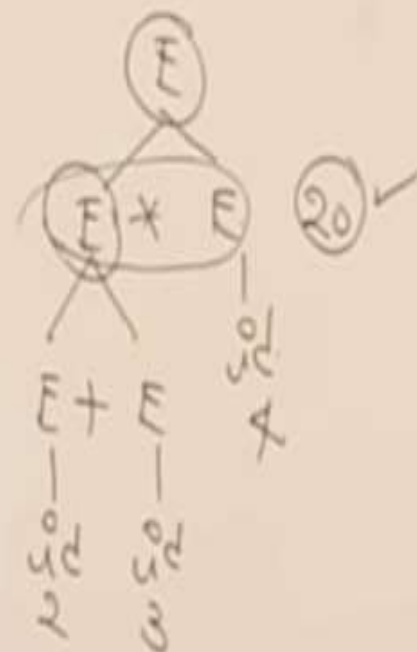
RMD: $E \Rightarrow E * E$

$$\Rightarrow E * id$$

$$\Rightarrow E + E * id$$

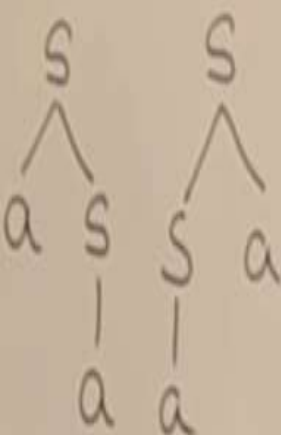
$$\Rightarrow E + id * id$$

$$\Rightarrow id + id * id$$



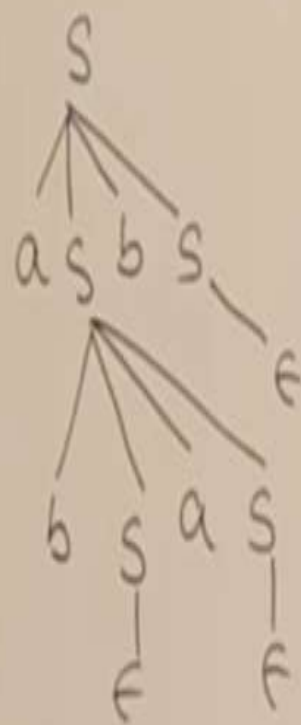
$$S \rightarrow aS / Sa / a$$

$$w = aa$$



$$S \rightarrow aSbS / bSaS / \epsilon$$

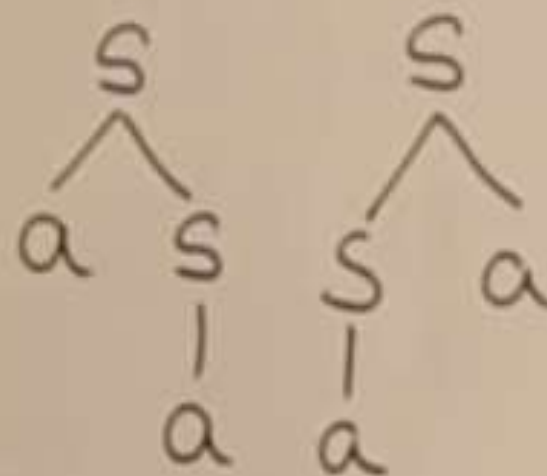
$$w = abab$$



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

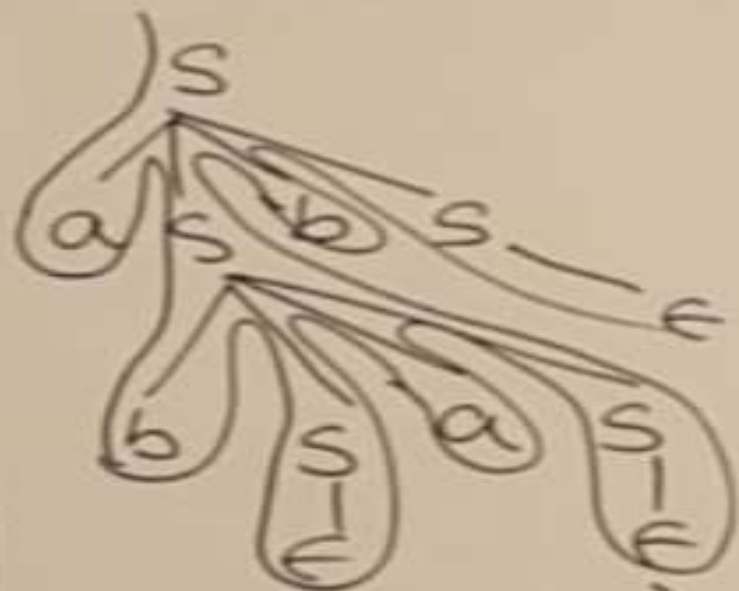
$$\checkmark S \rightarrow aS / Sa / a$$

$$w = aa$$



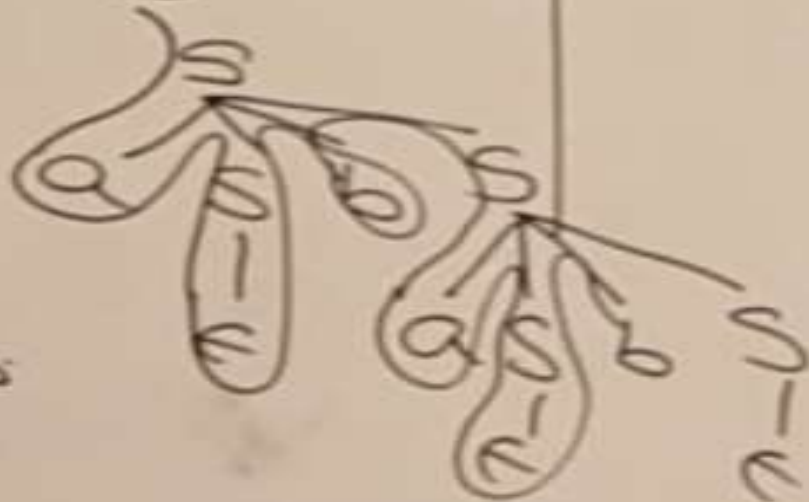
$S \rightarrow aSbS / bSaS / \epsilon$ | $R \rightarrow R+R / RR / R^* / \epsilon$

$w = abab$



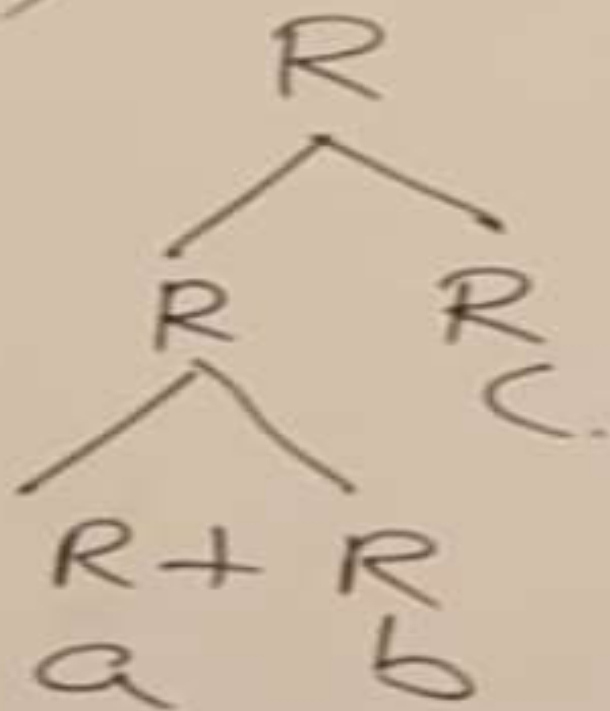
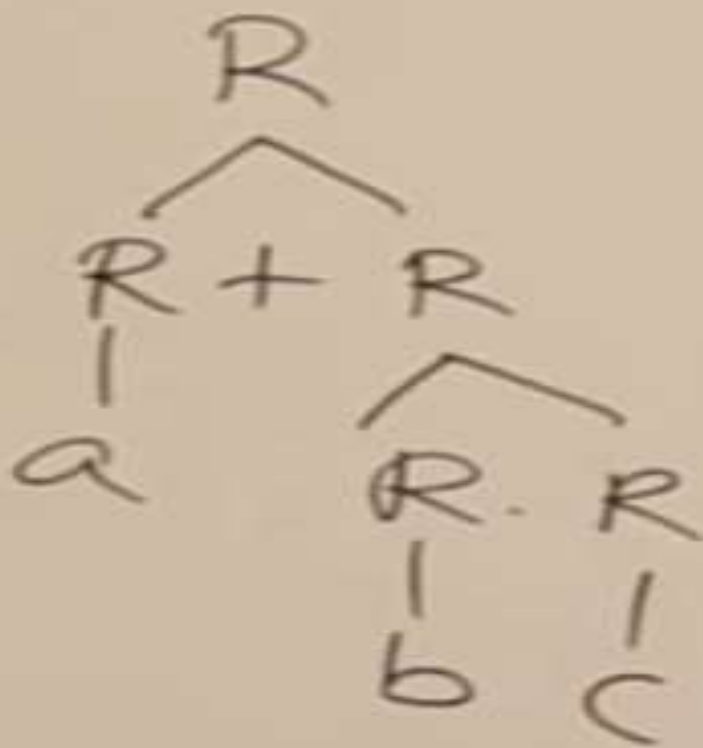
abab

abab



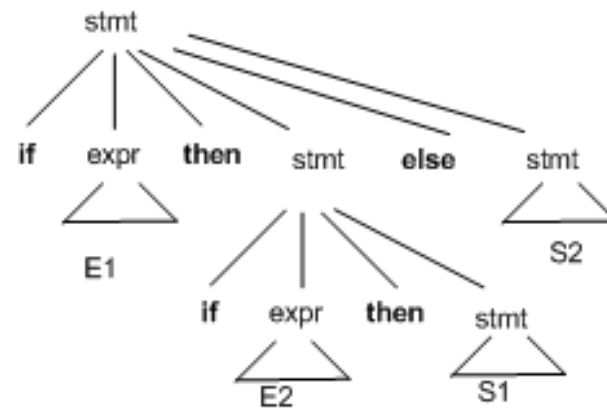
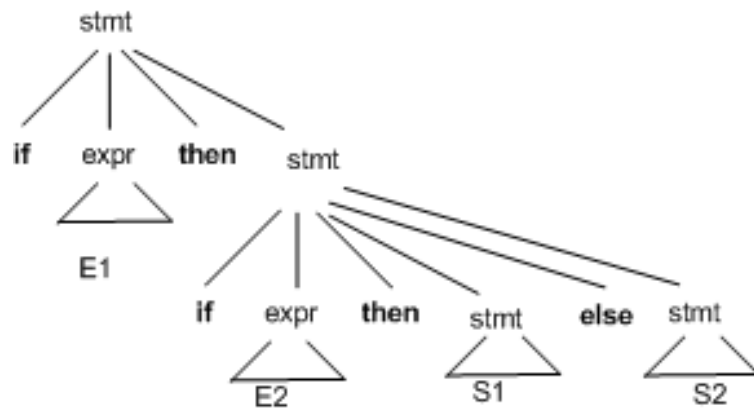
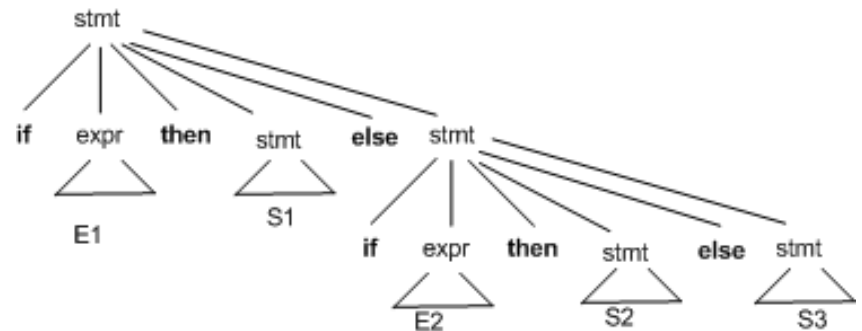
$R \rightarrow R+R / RR / R^* / a / b / c$

$a+(bc)$



Elimination of ambiguity

stmt \rightarrow **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:

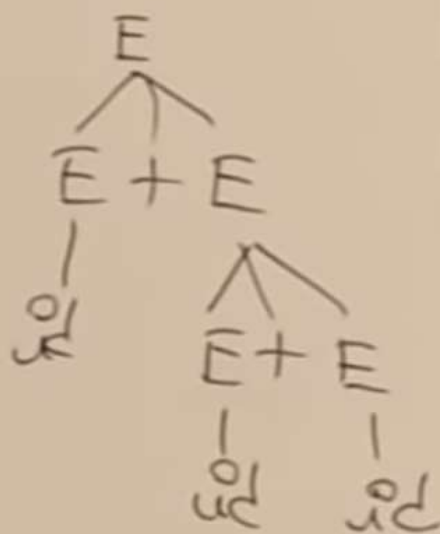
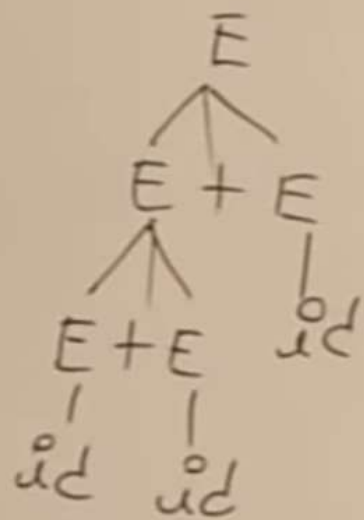
```
stmt → matched_stmt | unmatched_stmt  
matched_stmt → if expr then matched_stmt else matched_stmt /  
other  
unmatched_stmt → if expr then stmt  
                  | if expr then matched_stmt else unmatched_stmt
```


$E \rightarrow E + E$

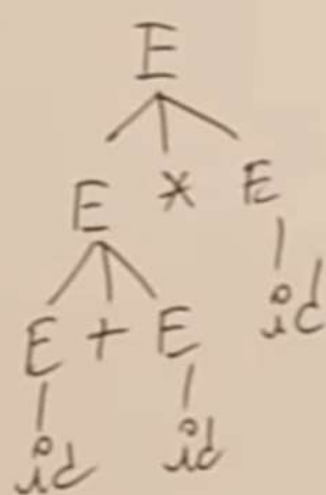
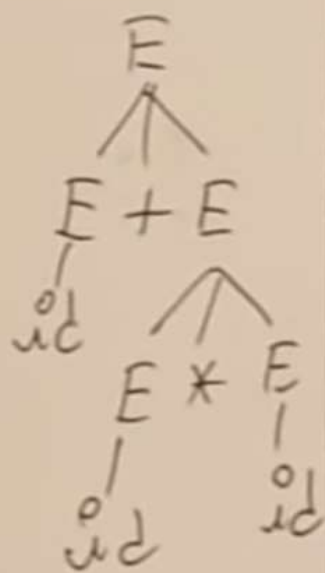
$/ E * E$

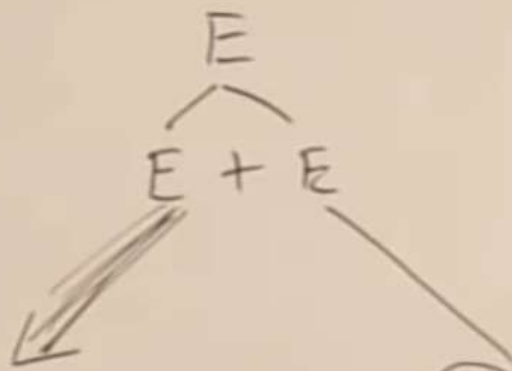
$/ id$

$id + id + id$



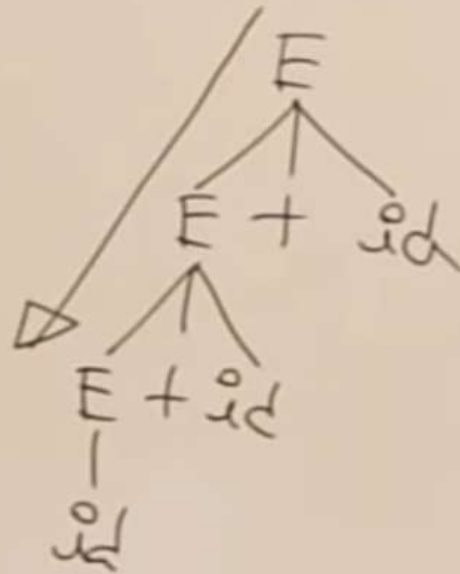
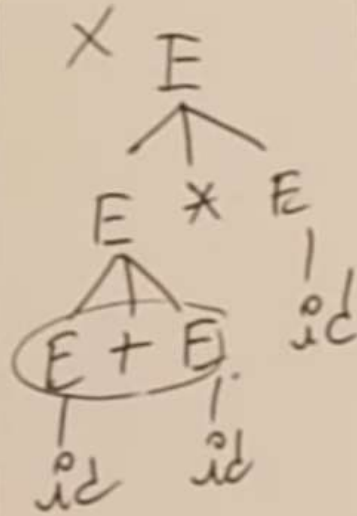
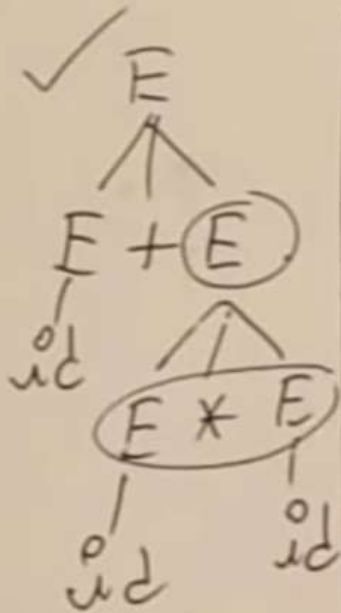
$id + id * id$





$E \rightarrow E + id / id$

$id + id * id$

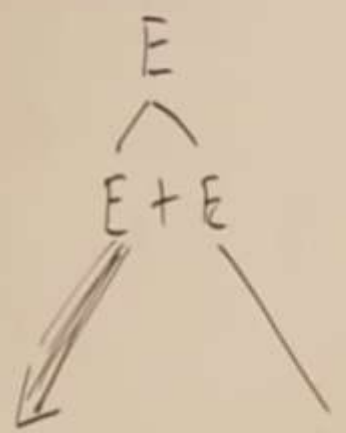
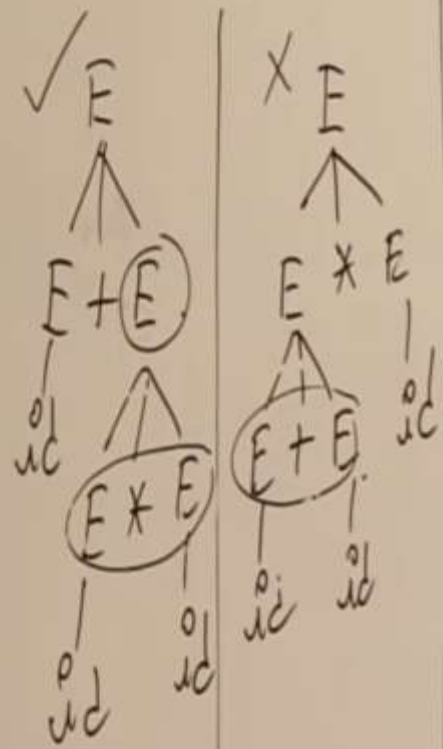


$id + id + id$



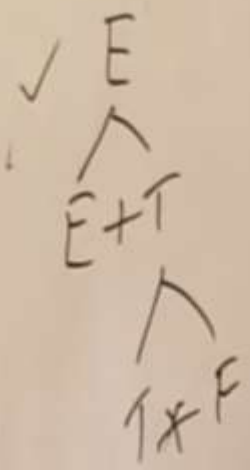
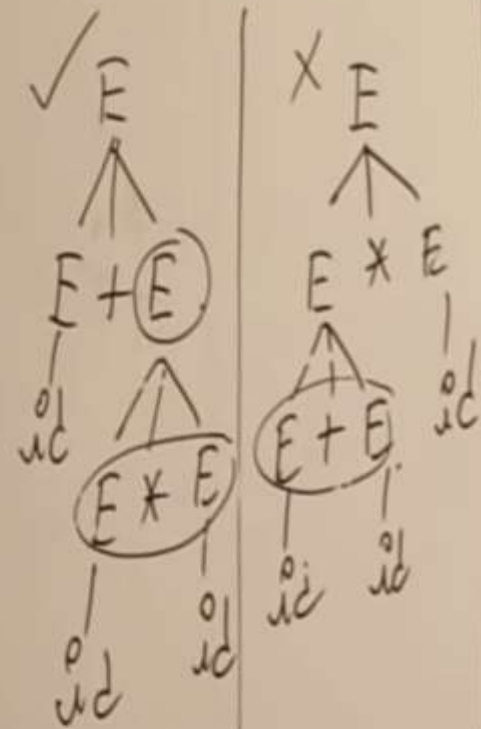
$id + id * id$

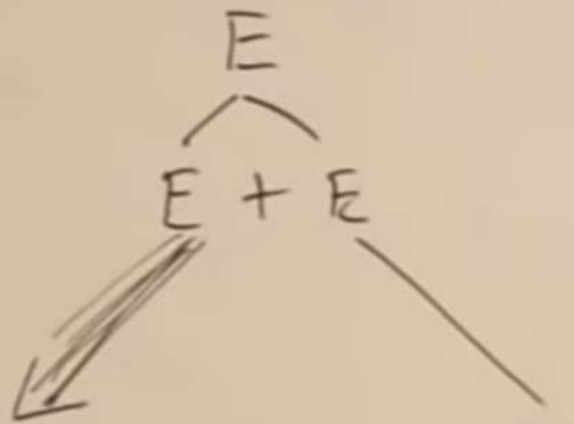
$E \rightarrow E + T$
 $T \rightarrow T * F$



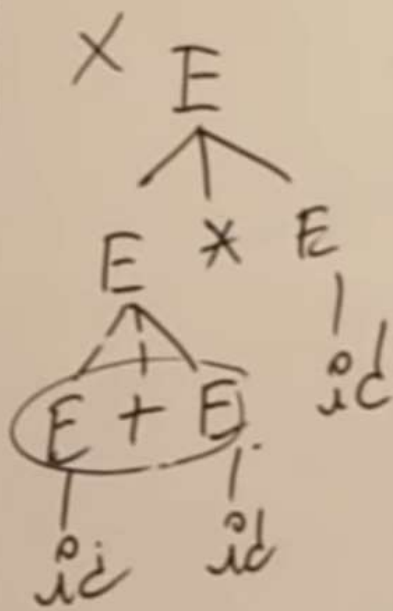
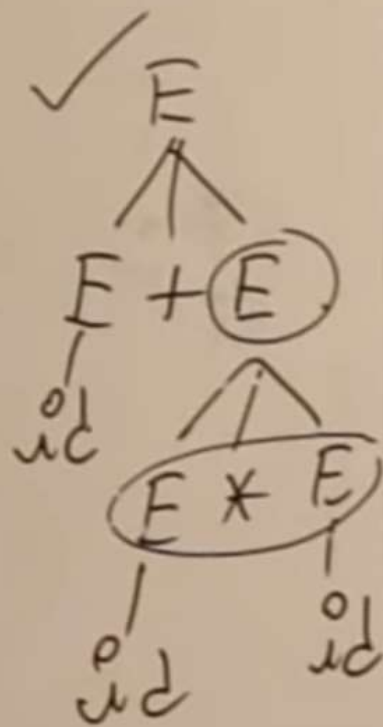
$id + id * id$

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



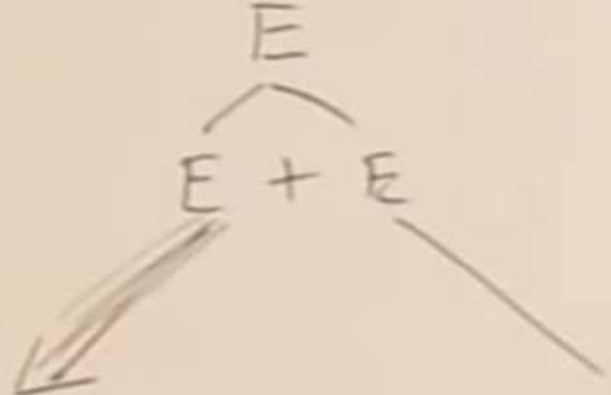


$id + id * id$

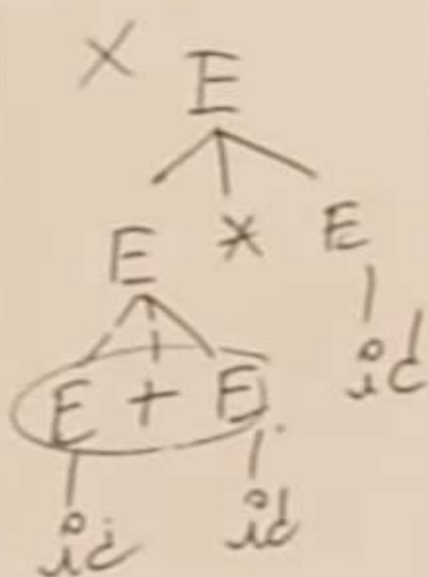
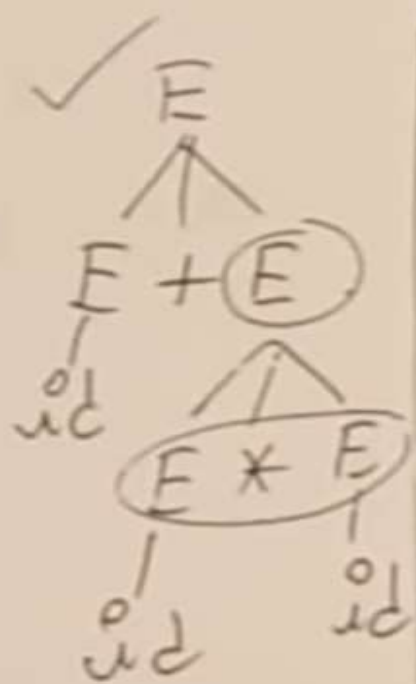


$E \rightarrow E + T / T$
 $T \rightarrow T * F / F$
 $F \rightarrow id$

$2 \uparrow 3 \uparrow 2$
 $2^3 2$



$id + id * id$



$E \rightarrow E + T / T$

$T \rightarrow T * F / F$

$F \rightarrow G \uparrow F / G$

$G \rightarrow id$

$(+), (*), (\uparrow)$

$$R \rightarrow R + R$$

$$/ R R$$

$$/ R *$$

$$/ a$$

$$/ b$$

$$/ c$$

$$E \rightarrow E \underline{+} T / T$$

$$T \rightarrow T F / F$$

$$F \rightarrow F * / a / b / c.$$

$bExp \rightarrow bExp \ \& \ bExp$

/ $bExp$ and $bExp$

/ not $bExp$

/ True

/ False.

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

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

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

$$\begin{aligned}
 A &\rightarrow A \$ B / B \\
 B &\rightarrow B \# C / C \\
 C &\rightarrow C @ D / D \\
 D &\rightarrow d
 \end{aligned}$$

$$\begin{aligned}
 \$ &\succ \$ \\
 \# &\succ \# \\
 @ &\succ @ \\
 \$ &\prec \# \prec @
 \end{aligned}$$

$$\begin{aligned}
 E &\rightarrow E * F \\
 &\quad / F + E \\
 &\quad / F \\
 &\quad / F - F \\
 &\quad / d
 \end{aligned}$$

~~$F \rightarrow F - F$~~

$$* \doteq +$$

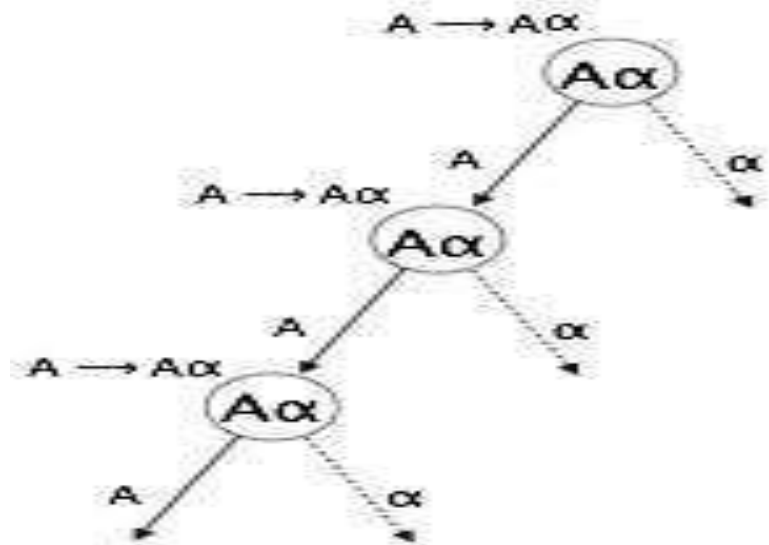
$$\begin{aligned}
 \$ &\succ \$ \\
 \# &\succ \# \\
 @ &\succ @ \\
 \$ &\prec \# \prec @
 \end{aligned}$$

$$\begin{aligned}
 * &\succ * \\
 + &\prec +
 \end{aligned}$$

Ambiguous Grammar	Unambiguous Grammar
<p>A grammar is said to be ambiguous if for at least one string generated by it, it produces more than one-</p> <ul style="list-style-type: none"> • parse tree • or derivation tree • or syntax tree • or leftmost derivation • or rightmost derivation 	<p>A grammar is said to be unambiguous if for all the strings generated by it, it produces exactly one-</p> <ul style="list-style-type: none"> • parse tree • or derivation tree • or syntax tree • or leftmost derivation • or rightmost derivation
<p>For ambiguous grammar, leftmost derivation and rightmost derivation represents different parse trees.</p>	<p>For unambiguous grammar, leftmost derivation and rightmost derivation represents the same parse tree.</p>
<p>Ambiguous grammar contains less number of non-terminals.</p>	<p>Unambiguous grammar contains more number of non-terminals.</p>
<p>For ambiguous grammar, length of parse tree is less.</p>	<p>For unambiguous grammar, length of parse tree is large.</p>
<p>Ambiguous grammar is faster than unambiguous grammar in the derivation of a tree.</p> <p>(Reason is above 2 points)</p>	<p>Unambiguous grammar is slower than ambiguous grammar in the derivation of a tree.</p>
	<p><u>Example-</u></p> $E \rightarrow E + T / T$ $T \rightarrow T * E / E$

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 $A \Rightarrow A\alpha$
- Top down parsing methods cant handle left-recursive grammars
- A simple rule for direct left recursion elimination:
 - For a rule like:
 - $A \rightarrow A\alpha|\beta$
 - We may replace it with
 - $A \rightarrow \beta A'$
 - $A' \rightarrow \alpha A' | \epsilon$



Left recursion elimination (cont.)

- There are cases like following
 - $S \rightarrow Aa \mid b$
 - $A \rightarrow Ac \mid Sd \mid \varepsilon$
- Left recursion elimination algorithm:
 - Arrange the non terminals in some order A_1, A_2, \dots, A_n .
 - For (each i from 1 to n) {
 - For (each j from 1 to $i-1$) {
 - Replace each production of the form $A_i \rightarrow A_j \gamma$ by the production $A_i \rightarrow \delta_1 \gamma \mid \delta_2 \gamma \mid \dots \mid \delta_k \gamma$ where $A_j \rightarrow \delta_1 \mid \delta_2 \mid \dots \mid \delta_k$ are all current A_j productions
 - }
 - Eliminate left recursion among the A_i -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.

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

To the following:

$$A \rightarrow \beta A'$$

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

Take left recursive grammar:

$$A \Rightarrow A\alpha \Rightarrow A\alpha\alpha \Rightarrow A\alpha\alpha\alpha \dots \text{etc.}$$

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

A

WHY IS LEFT RECURSION A PROBLEM ?

Consider:

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

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?

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

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

...

How does this build strings ?

What does each string have to start with ?

RESOLVING DIFFICULTIES : LEFT RECURSION (2)

Informal Discussion:

Take all productions for A and order as:

$$A \rightarrow A\alpha_1 \mid A\alpha_2 \mid \dots \mid A\alpha_m \mid \beta_1 \mid \beta_2 \mid \dots \mid \beta_n$$

Where no β_i begins with A.

Now apply concepts of previous slide: A

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

$$A' \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \dots \mid \alpha_m A' \mid \in$$

For our example:

$$\begin{array}{lcl}
 E \rightarrow E + T \mid T & \longrightarrow & \left\{ \begin{array}{l} E \rightarrow TE' \\ E' \rightarrow + TE' \mid \in \end{array} \right. \\
 T \rightarrow T * F & \longrightarrow & \left\{ \begin{array}{l} T \rightarrow FT' \\ T' \rightarrow * FT' \mid \in \end{array} \right. \\
 & & \text{where } E' \rightarrow (E) \mid id
 \end{array}$$

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

To the following:

$$A \rightarrow \beta A'$$

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

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

$$A' \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \dots \mid \alpha_m A' \mid \in$$

RESOLVING DIFFICULTIES : LEFT RECURSION (3)

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

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

Algorithm:

Input: Grammar G with ordered Non-Terminals A_1, \dots, A_n

Output: An equivalent grammar with no left recursion

1. Arrange the non-terminals in some order $A_1 = \text{start NT}, A_2, \dots, A_n$
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: $A_1 \rightarrow A_2 a \mid b \mid \epsilon$

$A_2 \rightarrow A_2 c \mid A_1 d$

$i = 1$

For A_1 there is no left recursion

$i = 2$

for $j=1$ to 1 do

Take productions: $A_2 \rightarrow A_1 \gamma$ and replace
with

$A_2 \rightarrow \delta_1 \gamma \mid \delta_2 \gamma \mid \dots \mid \delta_k \gamma$ where $A_1 \rightarrow \delta_1 \mid \delta_2$

$\mid \dots \mid \delta_k$ are A_1

productions

in our case, $A_2 \rightarrow \epsilon \mid A_1 d$ becomes $A_2 \rightarrow A_2 ad \mid bd \mid d$
What's left: $A_1 \rightarrow A_2 a \mid b \mid$

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

Are we done ?

USING THE ALGORITHM (2)

No ! We must still remove A_2 left recursion !

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

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

Recall:

$$A \rightarrow A\alpha_1 \mid A\alpha_2 \mid \dots \mid A\alpha_m \mid \beta_1 \mid \beta_2 \mid \dots \mid \beta_n$$

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

$$A' \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \dots \mid \alpha_m A' \mid \epsilon$$

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

$$A_2 \rightarrow bdA_2' \mid dA_2'$$

$$A_2' \rightarrow c A_2' \mid adA_2' \mid \epsilon$$

Apply to above case. What do you get ?

REMOVING DIFFICULTIES : ϵ -MOVES

Transformation: In order to remove $A \rightarrow \epsilon$ 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:

$$E \rightarrow TE'$$

$$T_{\epsilon} \rightarrow E' \quad F \rightarrow T, + TE' \mid$$

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

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

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

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

A is Grammar ϵ -free if:

1. It has no ϵ -production **or**
 2. There is exactly one ϵ -production
- $S \rightarrow \epsilon$ and then the start symbol S does not appear on the right side of any production.

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

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

REMOVING DIFFICULTIES : CYCLES

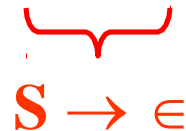
How would cycles be removed ?

Make sure every production is adding some **terminal(s)** (except a single ϵ -production in the start NT)...

e.g.

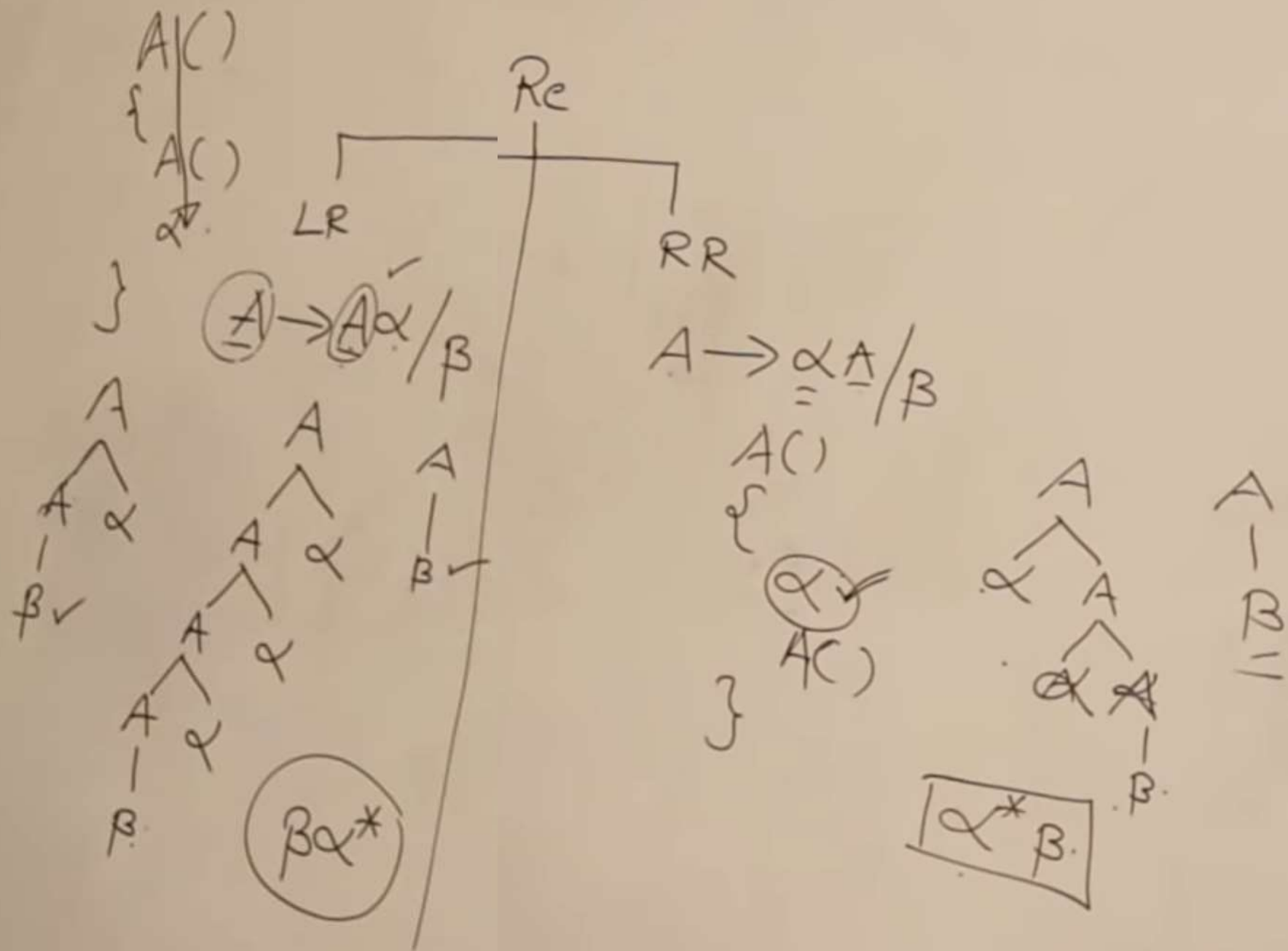
$S \rightarrow SS \mid (S) \mid \epsilon$

Has a cycle: $S \Rightarrow SS \Rightarrow S$


 $S \rightarrow \epsilon$

Transform to:

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



$A(C)$
 $\{ A(C) \}$
 α

LR

Re

RR

$\{ A \rightarrow A\alpha / \beta \}$

$A \rightarrow \alpha A / \beta$

$A(C)$

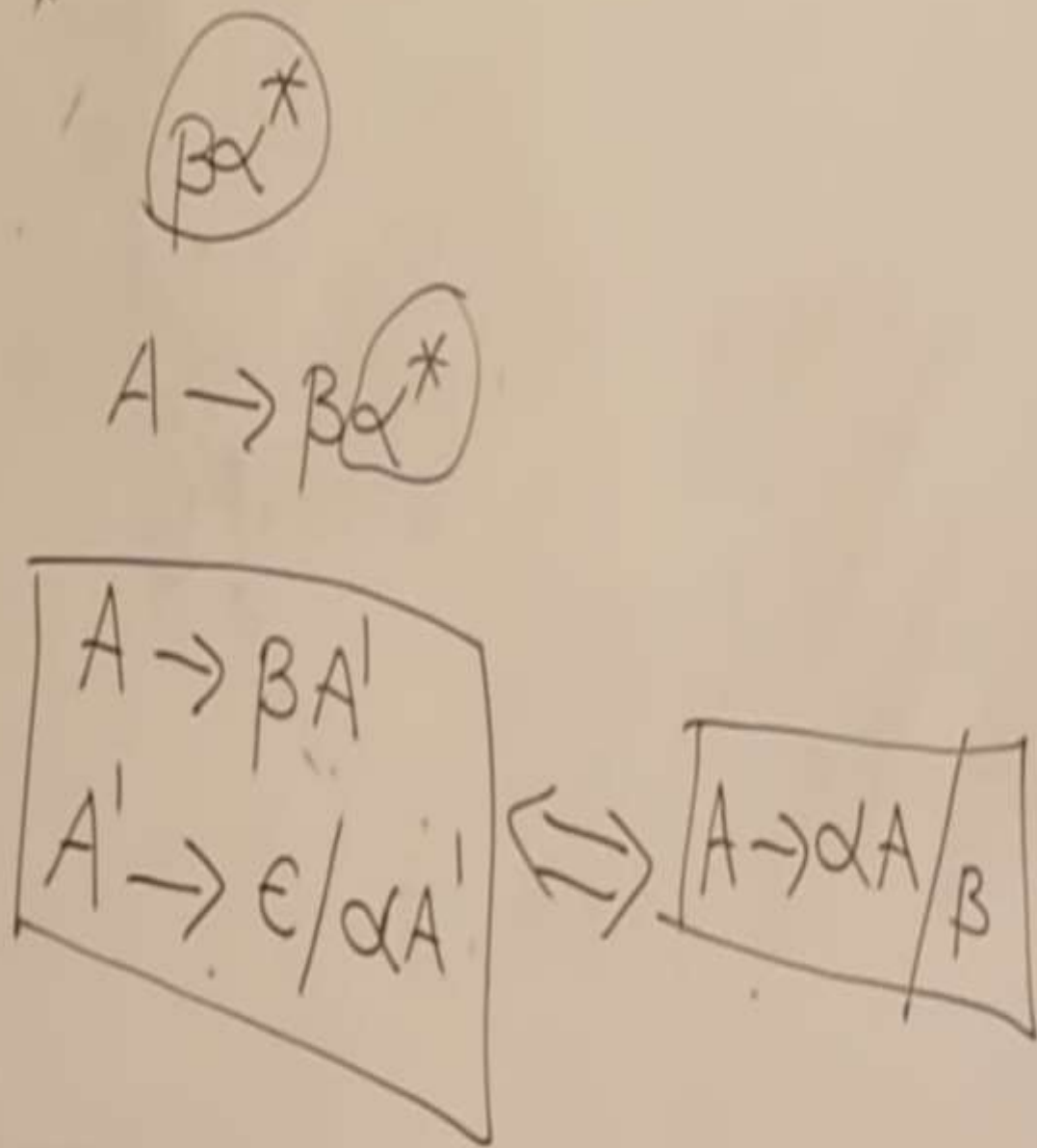
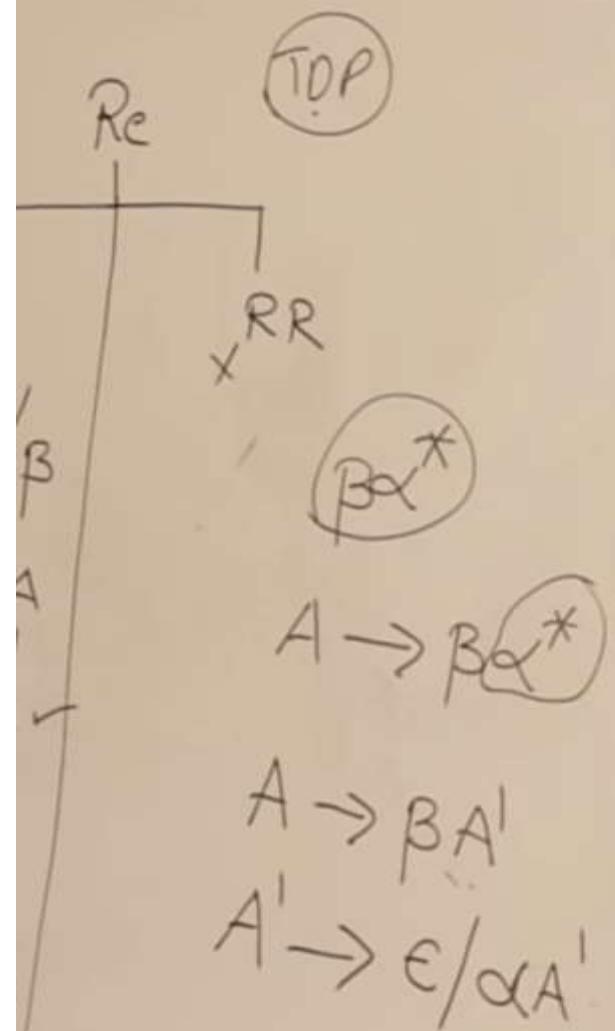
$\{ \alpha A(C) \}$

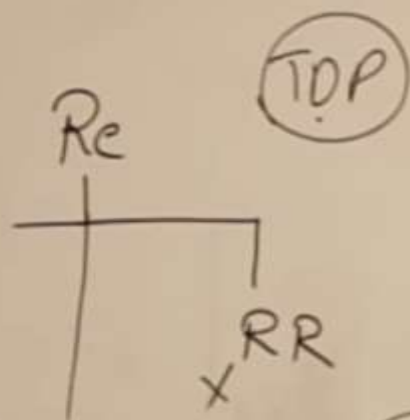
A
 α
 A
 α
 β

A
 β

$\{ A \}$
 A
 α
 β
 A
 α
 A
 α
 β
 A
 α
 β
 $\beta \alpha^*$

$\alpha^* \beta$





$$\boxed{\begin{array}{l} \bar{E} \rightarrow \bar{E} + \bar{T} / \bar{T} \\ \bar{A} \quad \bar{A} \quad \alpha \quad \beta \end{array}}$$

$$\beta \alpha^*$$

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

$$\boxed{\begin{array}{l} E \rightarrow TE' \\ E' \rightarrow \epsilon / +TE' \end{array}}$$

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

$$\Leftrightarrow \boxed{A \rightarrow A\alpha / \beta}$$

TOP

xRR

$\beta\alpha^*$

$A \rightarrow \beta\alpha^*$

$\frac{S}{A} \rightarrow \frac{S\alpha}{A} \frac{S/\alpha}{\beta}$

$S \rightarrow \alpha S'$

$S' \rightarrow \epsilon / \alpha S S'$

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

\Leftrightarrow

$A \rightarrow A\alpha / \beta$

$$S \rightarrow (L) / x$$

$$\underbrace{L \rightarrow \underbrace{L}_A, \underbrace{S}_\alpha / \underbrace{S}_\beta}_{\text{circled}}$$

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

\Downarrow

$$A \rightarrow \beta A'$$

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

$L \rightarrow SL'$
$L' \rightarrow , SL' / \epsilon$

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.

$$A \Rightarrow \beta d A'$$

$$A' \Rightarrow \alpha d A' \mid \varepsilon$$

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 \rightarrow **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 $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2$ then we replace it with
 - $A \rightarrow \alpha A'$
 - $A' \rightarrow \beta_1 \mid \beta_2$

Left factoring (cont.)

- Algorithm
 - For each non-terminal A, find the longest prefix α common to two or more of its alternatives. If $\alpha \neq \epsilon$, 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:
 - $S \rightarrow I E t S \mid i E t S e S \mid a$
 - $E \rightarrow b$
- 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.

REMOVING DIFFICULTIES : LEFT FACTORING

Problem : Uncertain which of 2 rules to choose:

stmt \rightarrow **if** *expr* **then** *stmt* **else** *stmt*

/ *if* *expr* **then** *stmt*

When do you know which one is valid ? What's the general form of *stmt* ?

$A \rightarrow \alpha\beta_1 \mid \alpha\beta_2$ $\alpha :$ **if** *expr* **then** *stmt*

$\beta_1 :$ **else** *stmt* $\beta_2 :$ \in

Transform to:

$A \rightarrow \alpha A'$

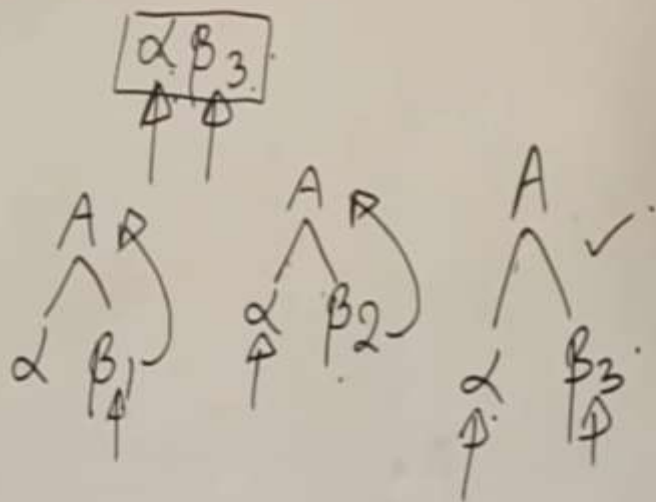
$A' \rightarrow \beta_1 \mid \beta_2$

EXAMPLE:

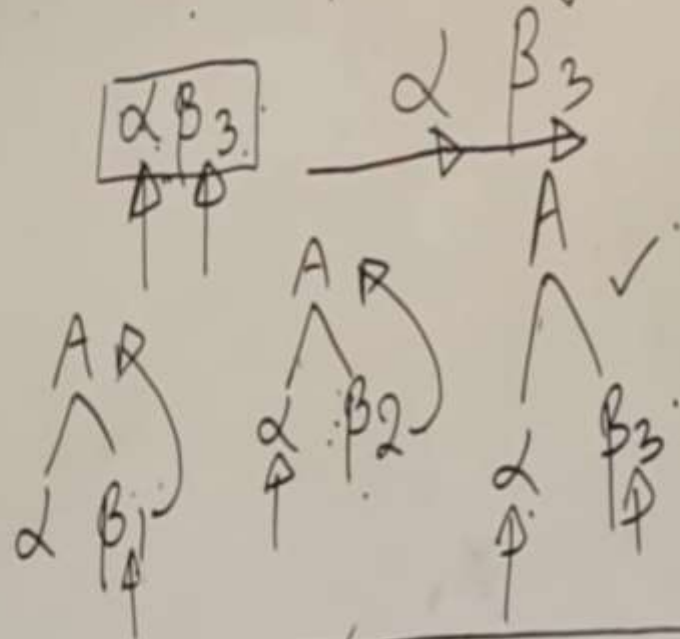
stmt \rightarrow **if** *expr* **then** *stmt*

rest rest \rightarrow **else** *stmt* **/** \in

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

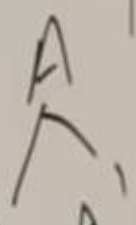
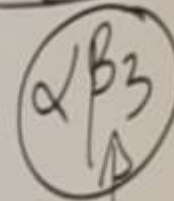


$$A \rightarrow \alpha \beta_1 \mid \alpha \beta_2 \mid \alpha \beta_3$$



$$A \rightarrow 2A$$

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



$$S \rightarrow \underline{iEtS}.$$

$$/\underline{iEtSeS}$$

$$/a.$$

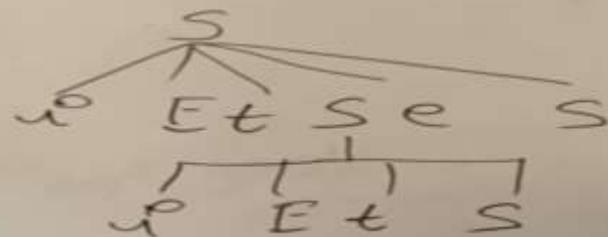
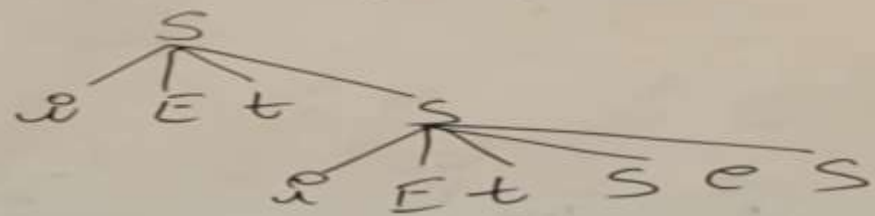
$$E \rightarrow b$$

$$S \rightarrow \underline{iEtSS'} / a.$$

$$S' \rightarrow \epsilon / eS$$

$$E \rightarrow b.$$

$\underline{a} E t \underline{a} E t S e S$.



$S \rightarrow \underline{aEtS}$
 $\quad \quad \underline{aEtSeS}$

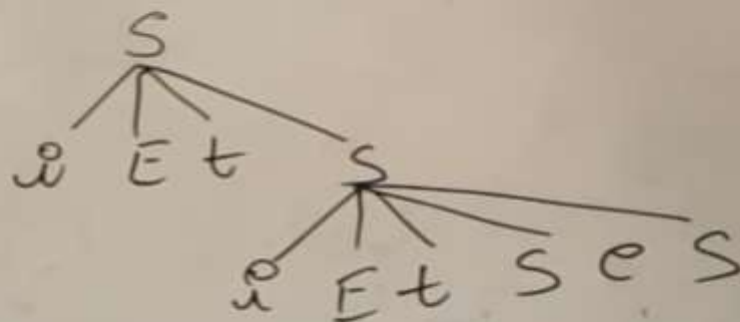
$\quad \quad \quad /a$
 $E \rightarrow b$

$S \rightarrow \underline{aEtSS'}/a$.

$S' \rightarrow \epsilon / eS$

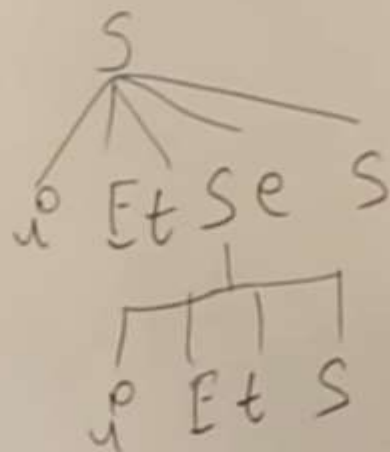
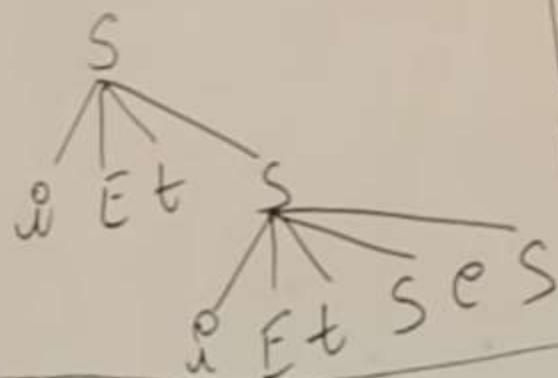
$E \rightarrow b$.

$\underline{a} E t \underline{a} E t S e S$.

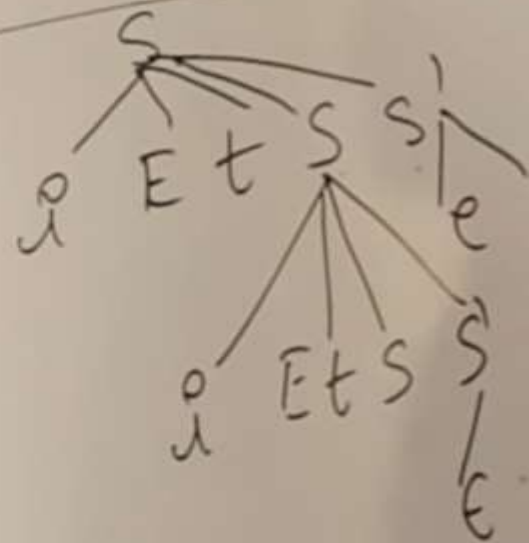
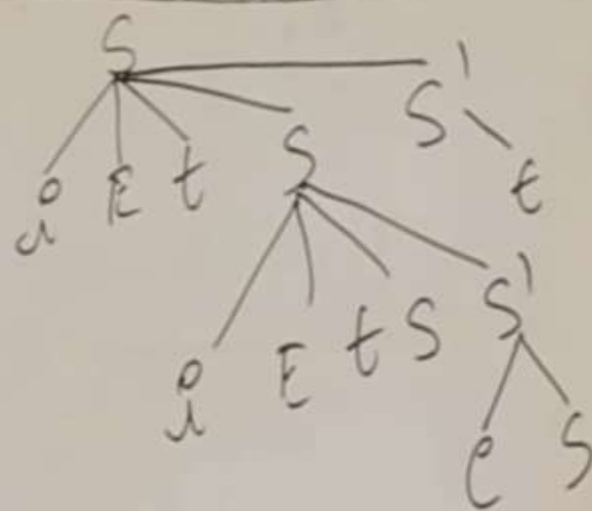


$S \rightarrow \underline{aEtS}$
 $\quad \underline{aEtSeS}$
 $\quad \underline{a}$
 $E \rightarrow b$

$aEt aEtSeS$ ✓



$S \rightarrow \underline{aEtSS'S'}/a$
 $S' \rightarrow \epsilon / eS$
 $E \rightarrow b$



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

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

EXAMPLES OF LEFT FACTORING

1. $S \rightarrow iEtS|iEtSES|a$ $E \rightarrow b$

2. $S \rightarrow aSSbS|aSaSb|abb|b$

3. $S \rightarrow bSSaaS|bSSaSb|bSb|a$

4. $A \rightarrow aAB / aBc / aAc$

5. $S \rightarrow a / ab / abc / abcd$

6. $S \rightarrow aAd / aB$

$A \rightarrow a / ab$

$B \rightarrow ccd / ddc$

CHECK GRAMMAR IS Ambiguous or not

1. $S \rightarrow aS \mid Sa \mid a$

2. $S \rightarrow aB \mid bA$

$$S \rightarrow aS \mid bAA \mid a$$

$$B \rightarrow bS \mid aBB \mid b$$

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

•LEFT RECURSION

1. $S \rightarrow Sab / Scd / Sef / g / h$

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. $A \rightarrow AA\alpha / \beta$

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. $S \rightarrow Aa / b$
 $A \rightarrow Ac / Sd / \epsilon$

Relationship Between Left Recursion, Left Factoring & Ambiguity-

1. $S \rightarrow aS / a / \epsilon$

2. $S \rightarrow aA / aB$

$$A \rightarrow a$$

$$B \rightarrow b$$

3. $S \rightarrow SS / \epsilon$

4. $S \rightarrow Sa / \epsilon$

5. $S \rightarrow aA / Ba$

$$A \rightarrow a$$

$$B \rightarrow a$$

6. $S \rightarrow Sa / \epsilon / bB / bD$

$$B \rightarrow b$$

$$D \rightarrow d$$