Unit – 2
Parsing Theory (I)

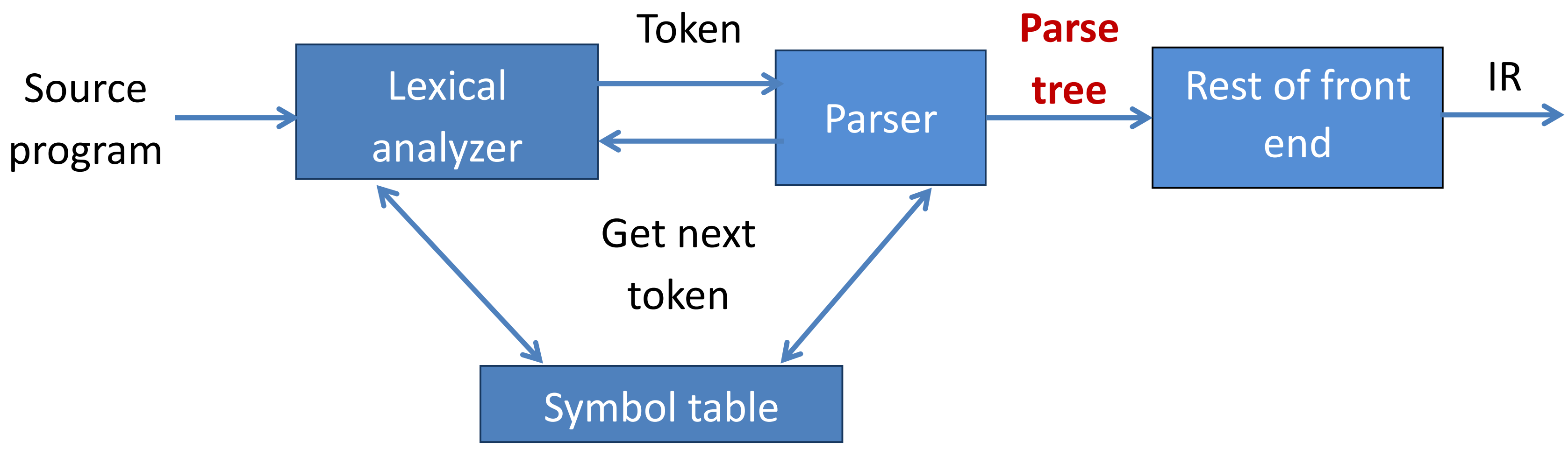
Mrs. Ruchi Sharma

ruchi.sharma@bkbiet.ac.in

Topics to be covered

- Role of parser
- Context free grammar
- Derivation & Ambiguity
- Left recursion & Left factoring
- Classification of parsing
- Backtracking
- LL(1) parsing
- Recursive descent paring
- Shift reduce parsing
- Operator precedence parsing
- LR parsing

Role of parser



- Parser obtains a string of token from the lexical analyzer and reports syntax error if any otherwise generates syntax tree.
- There are two types of parser:
    1. Top-down parser
    2. Bottom-up parser

Context free grammar

Context free grammars

- A context free grammar (CFG) is a 4-tuple $G = (V, \Sigma, S, P)$ where,

  V is finite set of non terminals,

  Σ is disjoint finite set of terminals,

  S is an element of V and it's a start symbol,

  P is a finite set formulas of the form $A \rightarrow \alpha$ where $A \in V$ and $\alpha \in (V \cup \Sigma)^*$

- Example:

  expr → expr op expr | (expr) | -expr | id

  *op → + | - | * | / | ↑*

Terminals: id + - * / ↑ ( ) **Non terminals: expr, op** **Start symbol: expr**

GRAMMARS

A grammar consists of 4 components

1. set of Tokens / Terminals.
2. set of Non Terminals
3. set of productions

$S \to PQ$

$\to P \to p$

$\to Q \to q$

⇓

each production has a NT, followed by arrow, followed by seq. of T and/or NT (RHS)

(4) Start symbol.

A grammar is specified by listing its productions, with the production for the start symbol appearing first

eg : expression with single digits having either + or - b/w them.

$L \to \underbrace{D+D} \mid \underbrace{D-D} \mid D$

$D \to 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$

$\underbrace{9+5}$

$L \to D + D$
$\quad\;\; 9 + 5$

How To Derive Strings from The Grammar

↳ Start w/ the production having the start symbol on the LHS.

- Repeatedly replace all the NT (on RHS) by their productions.

* All the strings that can be derived from a grammar belong to the language specified by that grammar.

Language → Grammar (Rules) → Productions

↙ ↘

NT (S) T

A GRAMMAR FOR ARITHMETIC EXPRESSIONS

$\underbrace{cstmt}_{NT} \rightarrow \underbrace{if}_{T}\ \underbrace{expr}_{NT}\ \underbrace{then}_{T}\ \underbrace{stmt}_{NT}\ \underbrace{else}_{T}\ stmt$

$op \rightarrow + \mid - \mid * \mid / \mid \uparrow$

$d \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$

expr : d op d , -d , (d)

expr op expr , -expr , (expr)

{expr → e}

$$\boxed{\begin{array}{l} e \rightarrow e\ op\ e \mid (e) \mid -e \mid d \\ d \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \\ op \rightarrow + \mid - \mid * \mid / \mid \uparrow \end{array}}$$

grammar

eg: -5

$e \rightarrow -e$
$\rightarrow -d$
$\rightarrow -5.$

7+5

$e \rightarrow e\ op\ e$
$\rightarrow d\ op\ d$
$\rightarrow 7+5.$

-(7+5)

-(7)

$e \rightarrow -e$
$\rightarrow -(e)$
$\rightarrow -(e\ op\ e)$
$\rightarrow -(d\ op\ d)$
$\rightarrow -(7+5)$

$\underbrace{7+5}\ \underbrace{-}\ \underbrace{9*2}$

$e \rightarrow e\ op\ e$
$\rightarrow e\ op\ e\ op\ e$
$\rightarrow e\ op\ e\ op\ e\ op\ e$
$\rightarrow d\ op\ d\ op\ d\ op\ d$
$\rightarrow 7+5-9*2$

$S \xRightarrow{*} abc$ $\quad A \Rightarrow a$

COMMON NOTATIONS

$\rightarrow$ or $\Rightarrow$: derives in 1 step

$\xRightarrow{*}$: derives in 0 or more steps

$\xRightarrow{+}$: derives in 1 or more steps

If we have a grammar G then the language generated by this grammar is denoted by L(G)

* A string w will be present in L(G) iff $S \xRightarrow{+} w$

Context Free Language and Grammar

Context Free Language: The languages derived from CFG are called CFL.

Grammar: Context Free Grammar.

Equivalent Grammar

If 2 grammars generate sam[e] ... then they are equivalent.

Sentential Form of Grammar : If $S \xRightarrow{*} \alpha$ where α may contain any Non Terminal (S is the start symbol) then α is called sentential form of G.

$E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(d) \Rightarrow -(7)$

$\Rightarrow -(d+d) \Rightarrow \ldots\ldots \Rightarrow (7+2)$

$E \Rightarrow d + d$

$A \rightarrow BDC$		$A \rightarrow XY$		
$B \rightarrow b$	$\Leftrightarrow$	$X \rightarrow bd$		
$D \rightarrow d$				
$C \rightarrow c$	bdc	$Y \rightarrow c$	bdc.	

Leftmost Derivation

The derivations in which only the leftmost Non Terminal, is replaced at each step.
in any sentential form.

eg: $A \rightarrow XYZ$, $X \rightarrow a$, $Y \rightarrow b$, $Z \rightarrow c$

$$A \xrightarrow[lm]{} XYZ \xrightarrow[lm]{} aYZ \xrightarrow[lm]{} abZ \xrightarrow{lm} \underbrace{abc}_{\text{Sentence}}$$

Canonical Derivation.

Rightmost Derivation

The derivations in which only the rightmost Non Terminal in any sentential form, is replaced at each step.

eg: $A \Rightarrow XYZ \Rightarrow XYc \Rightarrow Xbc \Rightarrow abc$

AMBIGUOUS GR

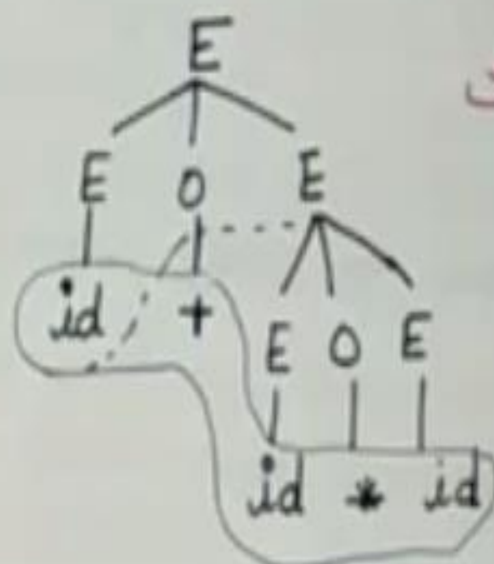
A grammar that produces more ... one Parse Tree for some sentence ... to be ambiguous.

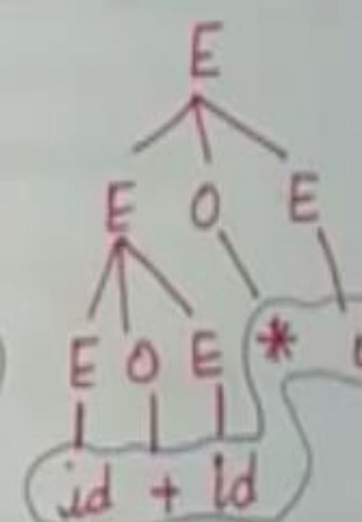
$E \rightarrow E\ O\ E \mid -E \mid (E) \mid id$ → Terminal

$O \rightarrow + \mid - \mid * \mid / \mid \uparrow$.

$\underbrace{id} + \underbrace{id\ *\ ...}$

$\underbrace{id + id} * \underbrace{id}$





1 string $\in L(G)$

More than 1 LMD or More tha... 1 RMD

ASSOCIATIVITY OF OPERATORS

When an operand has operators on both its sides (left and right) then we need rules to decide with which operator we will associate this operand.

Left Associative & **Right Associative**

$(1+2)+3$

$+$: left associative

$-, *, /$: L.A

$\underbrace{4*5}*6$

$\Downarrow$

Parse Trees

for left associative operators are more towards the left side in length.

$=, \uparrow$

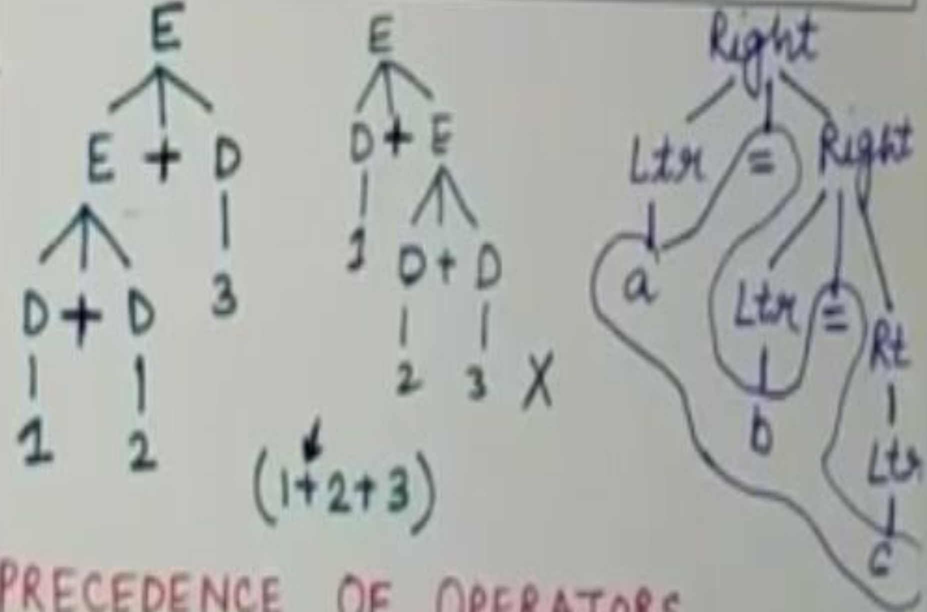
$a = \underbrace{b = 5}$

$a = b$; $2 \uparrow 3 \uparrow 5$

$(2)^{3^5}$

$1 \uparrow 2 \uparrow 3$

$(1)^8$



PRECEDENCE OF OPERATORS

Whenever an operator has a higher precedence than the other operator, it means that the first operator will get its operands before the operator with lower precedence.

$1+(2*3)$ $\quad \underbrace{+, -} < \underbrace{*, /}$

$1+2-3$

PARSE TREES

A parse tree is a pictoral depiction of how a start symbol symbol of a grammar derives a string in the language. eg: $A \to PQR$
$P \to a$
$Q \to b$
$R \to c|d$

PT

PROPERTIES

- Root is always labelled with the start symbol.
- Each leaf is labelled with a Terminal (Tokens)
- Each interior node is labelled by a NT.

Yield of Tree: The leaves of a Parse Tree when read from left to right form the yield.

Language defined by a gram
↳ set of all strings that are generated by some P.T formed by that gmr.

General Types of Parsers

1. Universal Parsers
   - ↳ can parse any kind of grammar
   - ↳ Not very efficient
   - ↳ CYK Algo, Earley's algo.
2. Top Down Parsers
   - ↳ builds the Parse Tree from the root (top) to leaves (bottom).
3. Bottom-Up Parsers
   - ↳ builds the Parse tree by starting at the leaves and ending at root.

$$\underline{E} \rightarrow \underline{E} + T \mid \underline{E} - T \mid T$$

$$\underline{T} \rightarrow \underline{T} * F \mid \underline{T} / F \mid F$$

$$F \rightarrow id$$

CONVERTING AN AMBIGUOUS GRAMMAR INTO UNAMBIGUOUS GRAMMAR

* / → Left Higher

\+ - → Left Lower

$E \to E + T \mid E - T \mid T$

$T \to T * F \mid T / F \mid F$

$F \to id$

$\underbrace{(1+2)}+3$

id + id + id.

- Left Recursion : If $A \stackrel{+}{\Rightarrow} A\alpha$

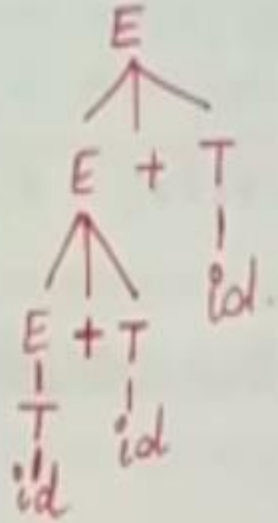
  Right Recursion : $A \stackrel{+}{\Rightarrow} \alpha A$

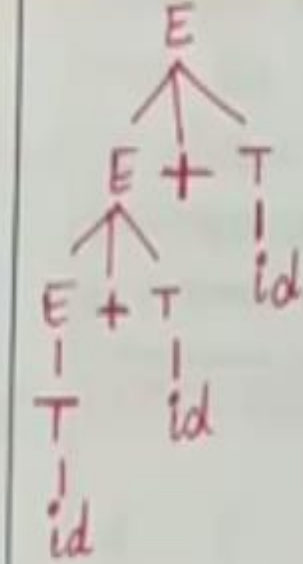
- $E \to E + T$

  $\to E + T + T$

  $\to T + T + T$

  $\to id + id + id.$





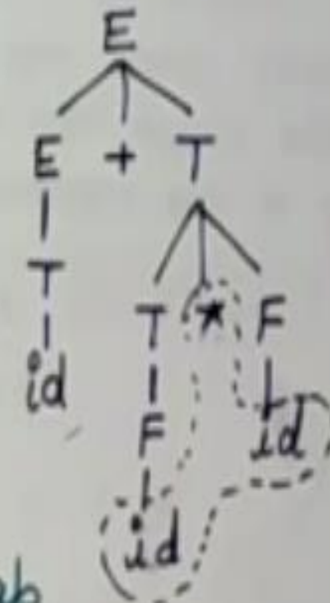
$\left.\begin{array}{l} A \to XY \\ X \to a \\ Y \to b \end{array}\right\}$

$L = \{\underline{ab}\}$

$A \to ab.$

$L = \{\underline{ab}\}$

1 + 2 * 3



Derivation & Ambiguity

Derivation

- Derivation is used to find whether the string belongs to a given grammar or not.
- Types of derivations are:
  1. Leftmost derivation
  2. Rightmost derivation

Leftmost derivation

- A derivation of a string W in a grammar G is a left most derivation if at every step the left most non terminal is replaced.
- Grammar: S→S+S | S-S | S*S | S/S | a   Output string: a*a-a

S

→**S-S**

→**S*S**-S

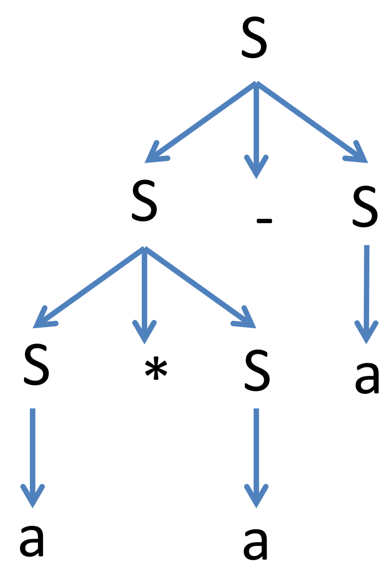
→**a***S-S

→a***a**-S

→a*a-**a**

Leftmost Derivation

Parse tree represents the structure of derivation



Parse tree

Rightmost derivation

- A derivation of a string W in a grammar G is a right most derivation if at every step the right most non terminal is replaced.
- It is all called canonical derivation.
- Grammar: S→S+S | S-S | S*S | S/S | a   Output string: a*a-a

S

→**S*S**

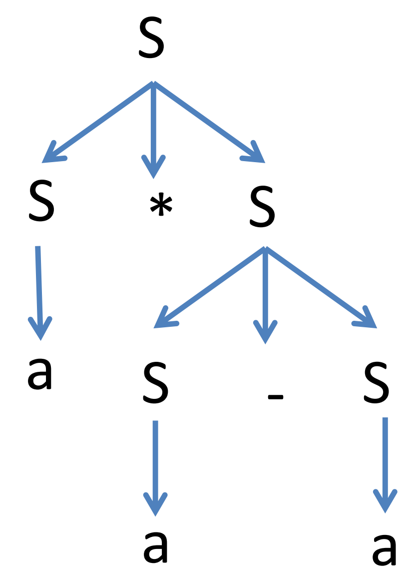
→S***S-S**

→S*S-**a**

→S***a**-a

→**a***a-a

Rightmost Derivation



Parse Tree

Exercise

1. $S \rightarrow A1B$

   $A \rightarrow 0A \mid \epsilon$

   $B \rightarrow 0B \mid 1B \mid \epsilon$

   String: 1001. Perform leftmost derivation.

2. $E \rightarrow E+E \mid E*E \mid id \mid (E) \mid -E$

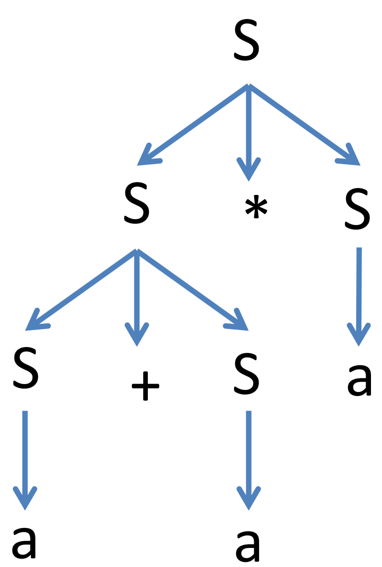
   String : id + id * id. Perform rightmost derivation

Ambiguous grammar

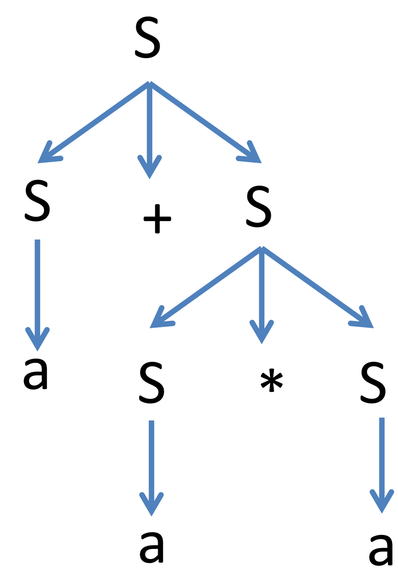
Ambiguous grammar

- Ambiguous grammar is one that produces more than one leftmost or more then one rightmost derivation for the same sentence.
- Grammar: S→S+S | S*S | (S) | a   Output string: a+a*a

S
→**S*S**
→**S+S***S
→**a**+S*S
→a+**a***S
→a+a***a**



S
→**S+S**
→**a**+S
→a+**S*S**
→a+**a***S
→a+a***a**



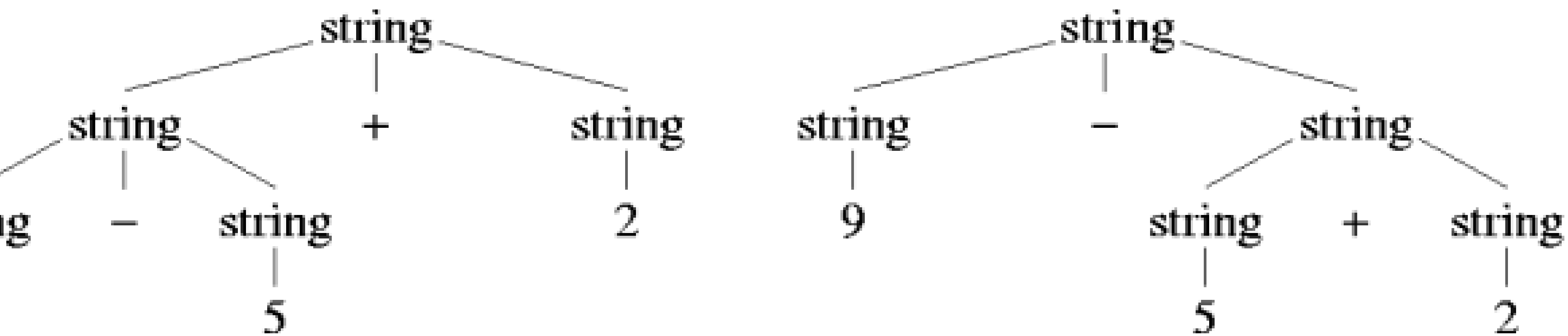
Here, ***Two leftmost derivation*** for string a+a*a is possible hence, above grammar is ambiguous.

Example of an Ambiguous Grammar

$string \rightarrow string + string$

$string \rightarrow string - string$

$string \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$



$string \rightarrow string + string \rightarrow string - string + string$

$\rightarrow 9 - string + string \rightarrow 9 - 5 + string \rightarrow 9 - 5 + 2$

$string \rightarrow string - string \rightarrow 9 - string$

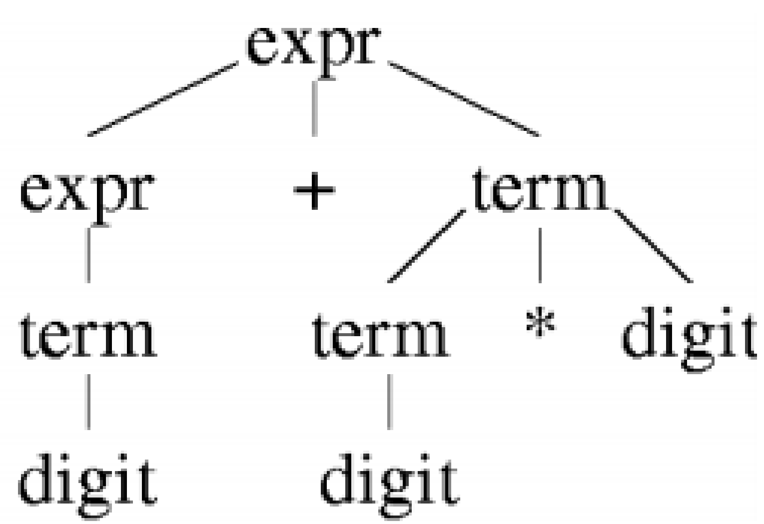
$\rightarrow 9 - string + string \rightarrow 9 - 5 + string \rightarrow 9 - 5 + 2$

Precedence

By convention

9 + 5 * 2   * has higher precedence than + because it takes its operands before +

expr –> expr + term | term
term –> term * digit | digit



Precedence (cont.)

Different operators have the same precedence when they are defined as alternative productions of the same nonterminal.

$$\text{expr} \rightarrow \text{expr} + \text{term} \mid \text{expr} - \text{term} \mid \text{term}$$
$$\text{term} \rightarrow \text{term} * \text{factor} \mid \text{term} / \text{factor} \mid \text{factor}$$
$$\text{factor} \rightarrow \text{digit} \mid (\text{expr})$$

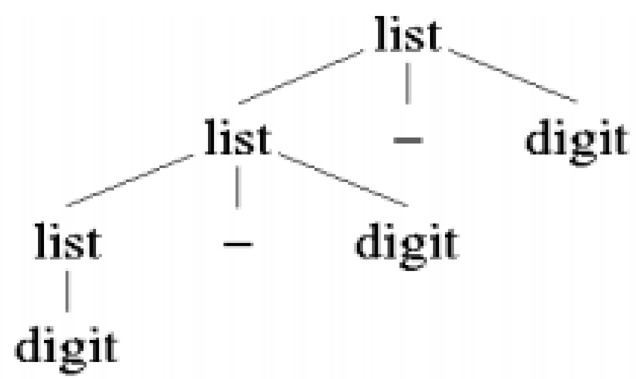
Associativity

By convention

9 – 5 – 2 left (operand with – on both sides, the operation on the left is performed first)

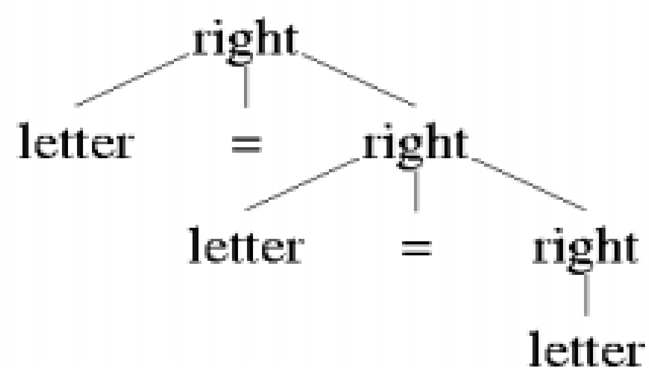
a = b = c right (operand with = on both sides, the operation on the right is performed first)

list –> list – digit
list –> digit



grows to the left

right –> letter = right
right –> letter



grows to the right

Eliminating Ambiguity

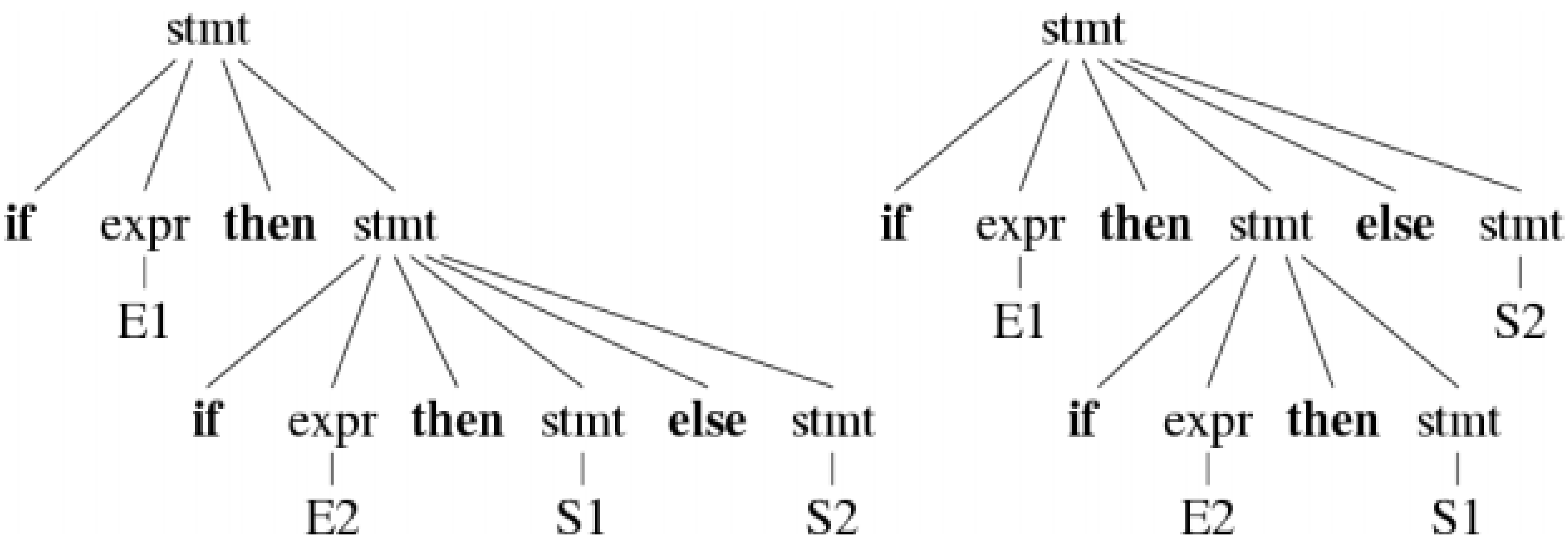
- Sometimes ambiguity can be eliminated by rewriting a grammar.

$$\begin{aligned} stmt \rightarrow\ & \textbf{if } expr \textbf{ then } stmt \\ |\ & \textbf{if } expr \textbf{ then } stmt \textbf{ else } stmt \\ |\ & other \end{aligned}$$

- How do we parse:

if E1 **then** if E2 **then** S1 **else** S2

Two Parse Trees for "if E1 then if E2 then S1 else S2"



Eliminating Ambiguity (cont.)

$$
\begin{array}{rcl}
\mathit{stmt} & \rightarrow & \mathit{matched\_stmt} \\
& | & \mathit{unmatched\_stmt} \\
\mathit{matched\_stmt} & \rightarrow & \textbf{if } \mathit{expr} \textbf{ then } \mathit{matched\_stmt} \textbf{ else } \mathit{matched\_stmt} \\
& | & \mathit{other} \\
\mathit{unmatched\_stmt} & \rightarrow & \textbf{if } \mathit{expr} \textbf{ then } \mathit{stmt} \\
& | & \textbf{if } \mathit{expr} \textbf{ then } \mathit{matched\_stmt} \textbf{ else } \mathit{unmatched\_stmt}
\end{array}
$$

Exercise

Check whether following grammars are ambiguous or not:

1. S→ aS | Sa | ϵ (string: aaaa)
2. S→ aSbS | bSaS | ϵ (string: abab)
3. S→SS+ | SS* | a (string: aa+a*)
4. Show that the CFG with productions: S → a | Sa | bSS | SSb | SbS is ambiguous.

Left recursion

A grammar is said to be left recursive if it has a non terminal A such that there is a derivation $A \rightarrow A\alpha$ for some string α.

Algorithm to eliminate left recursion

1. Arrange the non terminals in some order $A_1, \ldots, An$
2. for $i := 1\ to\ n$ **do begin**

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

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

        by the productions $A_i \rightarrow \delta_1\gamma | \delta_2\gamma | \ldots .. | \delta_k\gamma,$

        where $A_j \rightarrow \delta_1 | \delta_2 | \ldots . | \delta_k$ are all the current A_j productions;

    end

    eliminate the immediate left recursion among the A_i - productions

end

LEFT RECURSION

A grammar is left recursive if it has a Non Terminal A such that there is a derivation $A \xRightarrow{+} A\alpha$ for some string α.

Direct Left Recursion : $A \rightarrow Aa$

Indirect Left Recursion : $\left.\begin{array}{l} S \rightarrow Aa \\ A \rightarrow Sb \end{array}\right\}$

$S \xRightarrow{+} Sb$

REMOVING LEFT RECURSION

Why? Top Down Parsers cannot handle left recursion / grammars with LR

```
A()
{ A()
  α
}
```

How?

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

```
A()
{ α
  A()
}
```

* If there are multiple A productions

$A \rightarrow A\alpha_1 \mid A\alpha_2 \mid A\alpha_3 \mid \cdots \mid A\alpha_m \mid \underbrace{\beta_1 \mid \beta_2 \mid \cdots \mid \beta_n}_{\text{No } \beta_i \text{ begins w/ A}}$

$\boxed{\begin{array}{l} A \rightarrow \beta_1 A' \mid \beta_2 A' \mid \cdots \mid \beta_n A' \\ A' \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \cdots \mid \alpha_m A' \mid \epsilon \end{array}}$

Advantage :

↳ We are able to generate the same language even after removing LR.

Disadvantage :

↳ The above procedure only eliminates Direct LR but not indirect LR

Left recursion elimination

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

$A \rightarrow \quad A'$

$A' \rightarrow \quad A' \mid \epsilon$

EXAMPLES OF LEFT RECURSION

1) $E \to E+T \mid T$
$T \to T*F \mid F$
$F \to (E) \mid id$ ✓

$$\Rightarrow E \to E\underbrace{+T}_{\alpha} \mid \underbrace{T}_{\beta}$$

$A \to A\alpha \mid \beta$
$\Downarrow$
$A \to \beta A'$
$A' \to \alpha A' \mid \epsilon$

$\left\{ \begin{array}{l} E \to TE' \\ E' \to +TE' \mid \epsilon \end{array} \right\}$ ✓

$$\Rightarrow T \to T\underbrace{*F}_{\alpha} \mid \underbrace{F}_{\beta}$$

$\left\{ \begin{array}{l} T \to FT' \\ T' \to *FT' \mid \epsilon \end{array} \right\}$ ✓

2) $S \to S\underbrace{0S1S}_{\alpha} \mid \underbrace{01}_{\beta}$

$S \to 01S'$
$S' \to 0S1SS' \mid \epsilon$

3) $L \to L\underbrace{,S}_{\alpha} \mid \underbrace{S}_{\beta}$

$L \to SL'$
$L' \to ,SL' \mid \epsilon$

4) $S \to S\underbrace{X}_{\alpha_1} \mid S\underbrace{Sb}_{\alpha_2} \mid \underbrace{XS}_{\beta_1} \mid \underbrace{a}_{\beta_2} \Rightarrow$
$S \to SX$
$S \to SSb$
$S \to XS$
$S \to a$

$S \to XSS' \mid aS'$
$S' \to XS' \mid SbS' \mid \epsilon$

5) $A \to A\underbrace{A}_{\alpha_1} \mid A\underbrace{b}_{\alpha_2}$

$A' \to AA' \mid bA' \mid \epsilon$.

Examples: Left recursion elimination

E→E+T | T

E→TE'

E'→+TE' | ε

T→T*F | F

T→FT'

T'→*FT' | ε

X→X%Y | Z

X→ZX'

X'→%YX' | ε

Examples: Left recursion elimination

S→ Aa | b

A→ Ac | Sd | ε

Here, Non terminal S is left recursive because:

S→ Aa → Sda

To remove indirect left recursion replace S with productions of S

S→ Aa | b

A→ Ac

A→ Sd | Aad | bd

A→ ε

A→Ac | Aad | bd | ε

Now, remove left recursion

S→ Aa | b

A→Ac | Aad | bd | ε

S→ Aa | b

A→ bdA' | A'

A'→ cA' | adA' | ε

Exercise

1. A→Abd | Aa | a

   B→Be | b

2. A→AB | AC | a | b
3. S→A | B

   A→ABC | Acd | a | aa

   B→Bee | b

4. Exp→Exp+term | Exp-term | term

Left factoring

Left factoring is a grammar transformation that is useful for producing a grammar suitable for predictive parsing.

Algorithm to left factor a grammar

Input: Grammar G

Output: An equivalent left factored grammar.

Method:

For each non terminal A find the longest prefix α common to two or more of its alternatives. If $\alpha \neq \in$, i.e., there is a non trivial common prefix, replace all the A productions $A \rightarrow \alpha\beta_1 | \alpha\beta_2 | \ldots\ldots\ldots\ldots.. | \alpha\beta_n | \gamma$ where γ represents all alternatives that do not begin with α by

$$A \rightarrow \alpha A' | \gamma$$
$$A' \rightarrow \beta_1 | \beta_2 | \ldots\ldots\ldots\ldots. | \beta_n$$

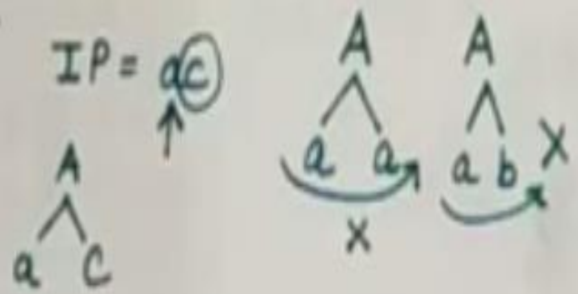
Here A' is new non terminal. Repeatedly apply this transformation until no two alternatives for a non-terminal have a common prefix.

LEFT FACTORING

At times, it is not clear which out of 2 (or more) productions to use to expand a Non-Terminal because multiple productions begin with same lookahead.

$A \to aa \mid$
$\to ab \mid$
$\to ac \mid$

IP = ac



A grammar with left factoring present is a NON DETERMINISTIC Grammar.

Removing Left Factoring

Why?

↳ Top Down Parsers cannot work with gmr. having L.F.

How? $A \to \alpha\beta_1 \mid \alpha\beta_2$

$$\left\{ \begin{array}{l} A \to \alpha A' \\ A' \to \beta_1 \mid \beta_2 \end{array} \right\}$$

$A \to aA'$
$A' \to a \mid b \mid c$

$$A \to \alpha\beta_1 \mid \alpha\beta_2 \mid \ldots\ldots \mid \alpha\beta_m \mid \gamma$$

$$\Downarrow$$

$$A \to \alpha A' \mid \gamma$$
$$A' \to \beta_1 \mid \beta_2 \mid \ldots\ldots \mid \beta_m .$$

eg: stmt → if expr then stmt else stmt
| if expr then stmt

stmt → if expr then stmt A
A → else stmt | ε

Left factoring elimination

$$A \rightarrow \alpha\,\beta \mid \alpha\,\delta \quad \Longrightarrow \quad \begin{array}{l} A \rightarrow \quad A' \\ A' \rightarrow \quad \mid \end{array}$$

LEFT FACTORING EXAMPLES

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

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

$A \to \alpha\beta_1 \mid \alpha\beta_2$
$\Downarrow$
$A \to \alpha A'$
$A' \to \beta_1 \mid \beta_2$

$X \to X+X \mid X*X \mid D$
$D \to 1 \mid 2 \mid 3$

$X \to XY \mid D$
$Y \to +X \mid *X$
$D \to 1 \mid 2 \mid 3$

$\alpha = X$
$\beta_1 = +X$
$\beta_2 = *X$
$\gamma = D$

$E \to T+E \mid T \longrightarrow \alpha = T, \ \beta_1 = +E$
$\beta_2 = \epsilon$
$T \to int \mid int*T \mid (E)$

$E \to TE'$
$E' \to +E \mid \epsilon$
$T \to int\ T' \mid (E)$
$T' \to *T \mid \epsilon.$

$S \to aSSbS \mid aSaSb \mid abb \mid b$
$S \to aS' \mid b$ ✓
$S' \to SSbS \mid SaSb \mid bb \Rightarrow \begin{cases} S' \to SS'' \mid bb \\ S'' \to SbS \mid aSb \end{cases}$

$\alpha = aS \quad \beta_1 = SbS \quad \beta_2 = aSb$

$\times S \to aSS' \mid abb \mid b \Longrightarrow S \to aS'' \mid b$ ✓
$S' \to SbS \mid aSb$ ✓ $\quad S'' \to SS' \mid bb.$ ✓

$A \to aA$
$B \to aB$
} No common Non Terminal on LHS.

Example: Left factoring elimination

$S \rightarrow aAB \mid aCD$

$S \rightarrow aS'$

$S' \rightarrow AB \mid CD$

$A \rightarrow xByA \mid xByAzA \mid a$

$A \rightarrow xByAA' \mid a$

$A' \rightarrow \epsilon \mid zA$

$A \rightarrow aAB \mid aA \mid a$

$A \rightarrow aA'$

$A' \rightarrow AB \mid A \mid \epsilon$

$A' \rightarrow AA'' \mid \epsilon$

$A'' \rightarrow B \mid \epsilon$

Exercise

1. S→iEtS | iEtSeS | a
2. A→ ad | a | ab | abc | x

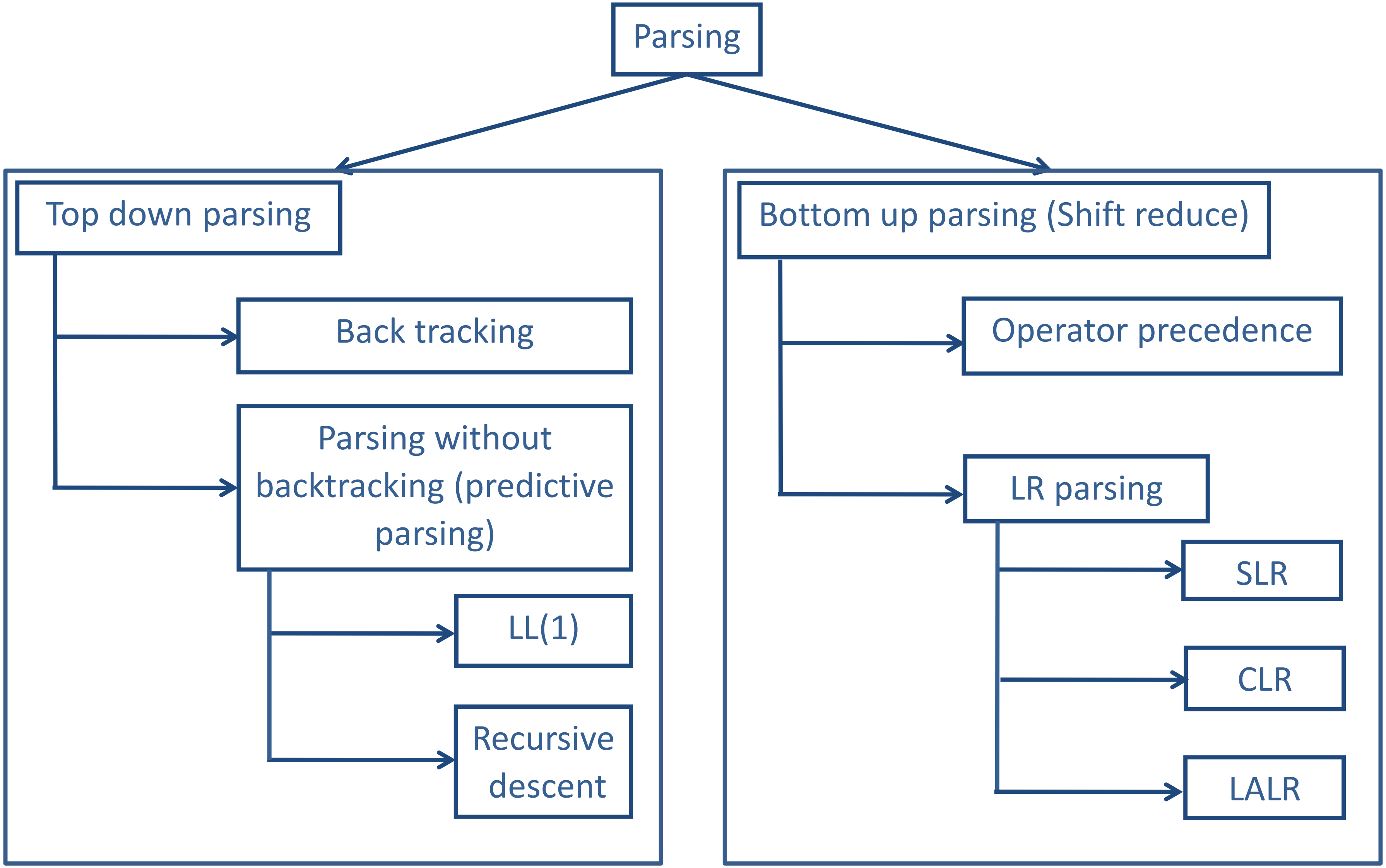
Parsing

Parsing

- Parsing is a technique that takes input string and produces output either a parse tree if string is valid sentence of grammar, or an error message indicating that string is not a valid.
- Types of parsing are:

1. **Top down parsing**: In top down parsing parser build parse tree from top to bottom.
2. **Bottom up parsing**: Bottom up parser starts from leaves and work up to the root.

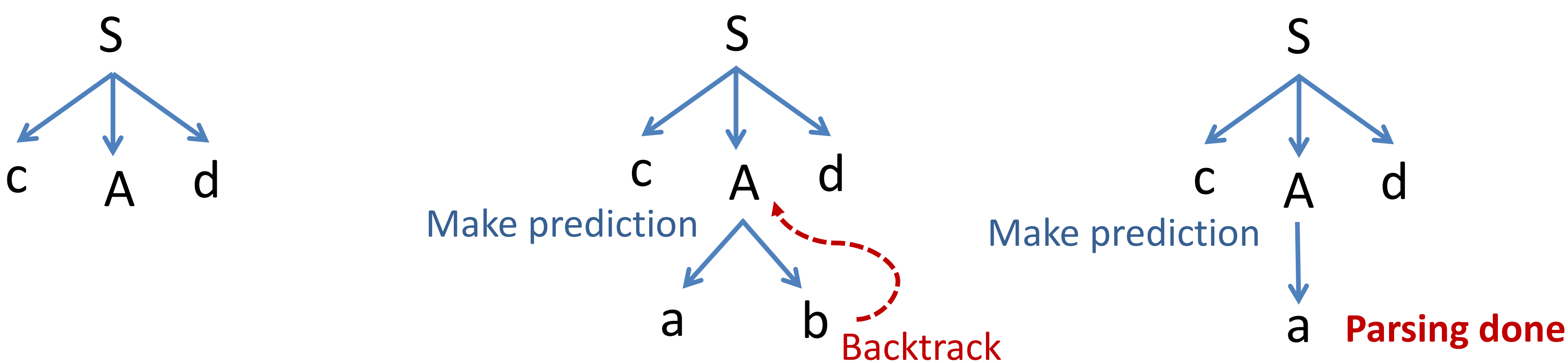
Classification of parsing methods



Backtracking

- In backtracking, expansion of nonterminal symbol we **choose one alternative** and **if any mismatch occurs** then we **try another alternative**.
- Grammar: S→ cAd   Input string: cad

  A→ ab | a



Exercise

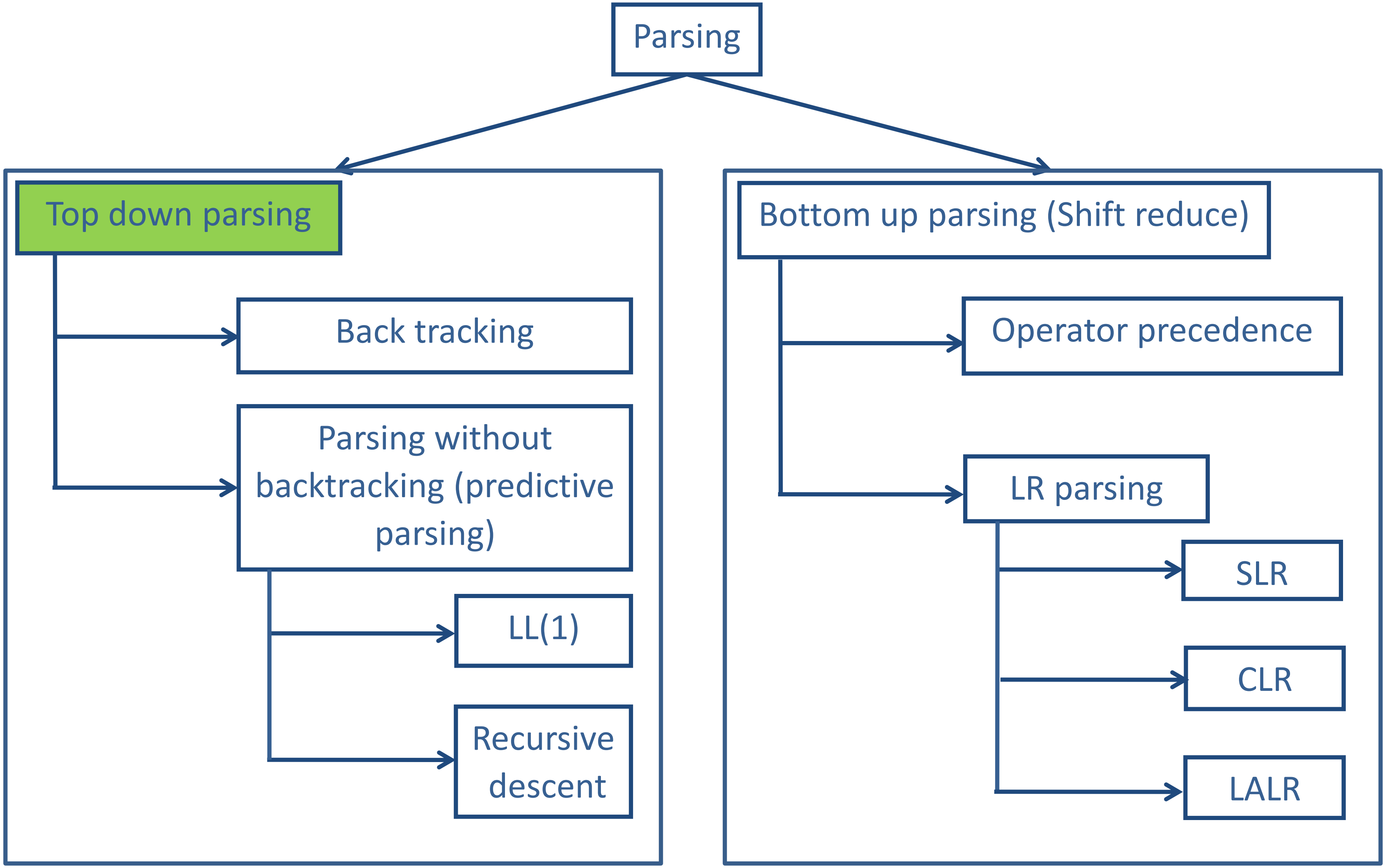
1. E→ 5+T | 3-T

   T→ V | V*V | V+V

   V→ a | b

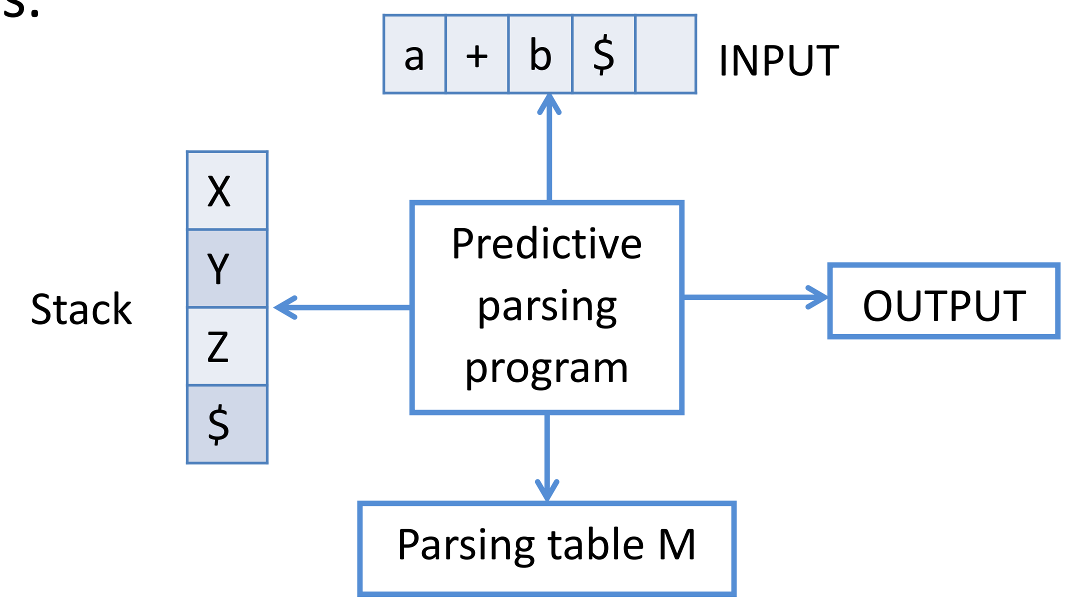
   String: 3-a+b

Parsing methods



LL(1) parser (predictive parser)

- LL(1) is non recursive top down parser.
  1. First **L** indicates input is scanned from left to right.
  2. The second **L** means it uses leftmost derivation for input string
  3. **1** means it uses only input symbol to predict the parsing process.



LL(1) parsing (predictive parsing)

Steps to construct LL(1) parser

1. Remove left recursion / Perform left factoring (if any).
2. Compute FIRST and FOLLOW of non terminals.
3. Construct predictive parsing table.
4. Parse the input string using parsing table.

Rules to compute first of non terminal

1. If $A \rightarrow \alpha$ and α is terminal, add α to $FIRST(A)$.
2. If $A \rightarrow \in$, add $\in$ to $FIRST(A)$.
3. If X is nonterminal and $X \rightarrow Y_1 Y_2 \ldots . Y_k$ is a production, then place a in $FIRST(X)$ if for some i, a is in $FIRST(Yi)$, and ϵ is in all of $FIRST(Y_1), \ldots \ldots \ldots, FIRST(Y_{i-1})$; that is $Y_1 \ldots Y_{i-1} \stackrel{*}{\Rightarrow} \epsilon$. If ϵ is in $FIRST(Y_j)$ for all $j = 1,2, \ldots \ldots, k$ then add ϵ to $FIRST(X)$.

   Everything in $FIRST(Y_1)$ is surely in $FIRST(X)$ If Y_1 does not derive ϵ, then we do nothing more to $FIRST(X)$, but if $Y_1 \stackrel{*}{\Rightarrow} \epsilon$, then we add $FIRST(Y_2)$ and so on.

Rules to compute first of non terminal

Simplification of Rule 3

If $A \rightarrow Y_1 Y_2 \ldots \ldots . . Y_K$,

- If Y_1 does not derives ∈ $then, FIRST(A) = FIRST(Y_1)$
- If Y_1 derives ∈ $then,$

  $FIRST(A) = FIRST(Y_1) - \epsilon$ U $FIRST(Y_2)$
- If Y_1 & Y_2 derives ∈ $then,$

  $FIRST(A) = FIRST(Y_1) - \epsilon$ U $FIRST(Y_2) - \epsilon\ U\ FIRST(Y_3)$
- If Y_1 , Y_2 & Y_3 derives ∈ $then,$

  $FIRST(A) = FIRST(Y_1) - \epsilon\ U\ FIRST(Y_2) - \epsilon\ U\ FIRST(Y_3) - \ \epsilon\ U\ FIRST(Y_4)$
- If Y_1 , Y_2 , Y_3 .....Y_K all derives ∈ $then,$

  $FIRST(A) = FIRST(Y_1) - \epsilon\ U\ FIRST(Y_2) - \epsilon\ U\ FIRST(Y_3) - \epsilon\ U\ FIRST(Y_4) - \epsilon\ U \ldots \ldots \ldots \ldots\ FIRST(Y_k)$ (note: if all non terminals derives ∈ then add ∈ to FIRST(A))

Finding FIRST()

If α is any string of grammar symbols then FIRST(α) is the set of terminals that begin the string derived from α.
If $\alpha \stackrel{*}{\Rightarrow} \epsilon$ then ϵ is also in FIRST(α)

Steps To Find FIRST()

1. If X is a terminal then FIRST(X) is {X}.
2. If X is a Non Terminal and $X \to Y_1 Y_2 \ldots Y_k$ is a production then
   (a) Add 'a' in FIRST(X) if for some i 'a' is in FIRST(Y_i) and ϵ is in all of FIRST(Y_1), FIRST(Y_2)….. FIRST(Y_{i-1}) i.e $Y_1 Y_2 \ldots, Y_{i-1} \stackrel{*}{\Rightarrow} \epsilon$
   (b) If ϵ is in FIRST(Y_j) for all $j = 1, 2, \ldots, k$ then add ϵ to FIRST(X)
3. If $X \to \epsilon$ is a production then add ϵ to FIRST(X)

$X \stackrel{*}{\Rightarrow} abc$
$\stackrel{*}{\Rightarrow} def$ $= \{a, d\}$

'a' $\to$ a
$\epsilon \to \epsilon$

(aABc)

(AB)

$A \to \underline{c}$

$X \to AB$
$A \to a \mid \epsilon$
$B \to b \mid \epsilon$

$\{a, b\}$

$X \to \underline{a}A.$
$\{a\}$

$X \to AB \to aB \to \underline{ab}.$
$X \to \epsilon B \to (B) \to b$
$X \to \cancel{\epsilon}B \to B \to \epsilon$

$E \to (TE') = \{id, (\}$
$E' \to +TE' \mid \epsilon$ $\{+, \epsilon\}$
$T \to (FT')$ First(T) = {id, (}
$T' \to (*FT') \mid \epsilon$ First(T') = {*, ϵ}
$F \to \underset{T}{\underline{id}} \mid \underset{T}{\underline{(}}E)$ First(F) = First(id) = {id, (}

SOLVED EXAMPLES

$S \rightarrow aABb$: $First(S) = First(aABb) = \{a\}$

$A \rightarrow c \mid \epsilon$ $First(A) = First(c) = \{c, \epsilon\}$

$B \rightarrow d \mid \epsilon$ $First(B) = First(d) = \{d, \epsilon\}$

$S \rightarrow aBDh$ $First(S) = First(aBDh) = \{a\}$

$B \rightarrow cC$ $\{c\}$

$C \rightarrow bC \mid \epsilon$ $\{b, \epsilon\}$

$First(f) / \epsilon$

$D \rightarrow EF$ $First(D) = First(EF) = \{g, f, \epsilon\}$

$E \rightarrow g \mid \epsilon$ $First(E) = \{g, \epsilon\}$

$F \rightarrow f \mid \epsilon$ $First(F) = \{f, \epsilon\}$

$D \rightarrow EF \rightarrow \epsilon F \rightarrow \epsilon$

$S \rightarrow Bb \mid Cd$ $First(S) = \{a, b, c, d\}$

$B \rightarrow aB \mid \epsilon$ $\{a, \epsilon\}$

$C \rightarrow cC \mid \epsilon$ $\{c, \epsilon\}$

$A \rightarrow da \mid BC$ $= \{d, g, h, \epsilon\}$

$S \rightarrow ACB \mid CbB \mid Ba$ $= \{d, g, h, \epsilon, b, a\}$

$B \rightarrow g \mid \epsilon$ $= \{g, \epsilon\}$

$C \rightarrow h \mid \epsilon$ $= \{h, \epsilon\}$

$S \rightarrow AB$ $\{b, a, c\}$

$A \rightarrow Ca \mid \epsilon$ $\{b, a, \epsilon\}$

$B \rightarrow BaAC \mid c$ $\{c\}$

$C \rightarrow b \mid \epsilon$ $\{b, \epsilon\}$

$S \rightarrow \cancel{A}\cancel{B}CDE = \{a, b, c\}$.

$A \rightarrow a \mid \epsilon$ $\{a, \epsilon\}$

$B \rightarrow b \mid \epsilon$ $\{b, \epsilon\}$

$C \rightarrow c$ $\{c\}$

$D \rightarrow d \mid \epsilon$ $\{d, \epsilon\}$

$E \rightarrow e \mid \epsilon$ $\{e, \epsilon\}$

Rules to compute FOLLOW of non terminal

1. Place $\$ \; in \; follow(S)$. (S is start symbol)
2. If $\mathrm{A} \rightarrow \alpha B \beta$, then everything in $FIRST(\beta)$ except for ϵ is placed in $FOLLOW(B)$
3. If there is a production $\mathrm{A} \rightarrow \alpha B$ or a production $\mathrm{A} \rightarrow \alpha B \beta$ where $FIRST(\beta)$ contains ϵ then everything in $FOLLOW(A) = FOLLOW(B)$

Finding FOLLOW()

For a Non Terminal A, Follow(A) is the set of terminals 'a' that can appear immediately to the right of A in some sentential form i.e the set of terminals 'a' such that there exists a derivation of the form $S \xRightarrow{*} \alpha A a \beta$ for some α and β

There may be symbols between A and 'a' which derived ϵ and disappeared

Rules for finding FOLLOW()

1) Put \$ in FOLLOW(S) where S is start symbol and \$ is input end marker
2) If $A \rightarrow \alpha B \beta$ then everything in First(β) is placed in Follow(B) except ϵ
3) If $A \rightarrow \alpha B$ or $A \rightarrow \alpha B \beta$ where First(β) contains ϵ, then everything in Follow(A) is in Follow(B).

ϵ never appears in FOLLOW()

⇒ To find Follow(A), look at the productions that have A present at the right hand side.

First(E′) = {+, ϵ}

First(T′) = {*, ϵ}

start symbol

E → TE′   {\$,)}

E′ → +TE′ | ϵ Follow(E′) = Follow(E) = {\$,)}

T → FT′   Follow(T) = First(E′) = {+, \$,)}

T′ → *FT′ | ϵ Follow(T′) = {+, \$,)}

F → (E) | id   Follow(F) = {*, +, \$,)}

SOLVED EXAMPLES

$S \to aABb = \{\$\}$
$A \to c \mid \epsilon = \{d, b\}$
$B \to d \mid \epsilon = \{b\}$

First(B) = d, ε

$S \to aBDh = \{\$\}$
$B \to cC = \{g, f, h\}$
$C \to bC \mid \epsilon = \{g, f, h\}$
$D \to EF = \{h\}$
$E \to g \mid \epsilon = \{f, h\}$
$F \to f \mid \epsilon = \{h\}$

Follow(B) =
First(D) =
First(EF)
First(E)
↳ g, ε
First(F) =
f, ε

$S \to Bb \mid Cd = \{\$\}$
$B \to aB \mid \epsilon = \{b\}$
$C \to cC \mid \epsilon = \{d\}$

$A \to da \mid BC = \{h, g, \$\}$
$S \to ACB \mid CbB \mid Ba \quad \{\$\}$
$B \to g \mid \epsilon = \{\$, a, h, g\}$
$C \to h \mid \epsilon = \{g, \$, h, b\}$

First(C) = {h, ε}
First(B) = {g, ε}

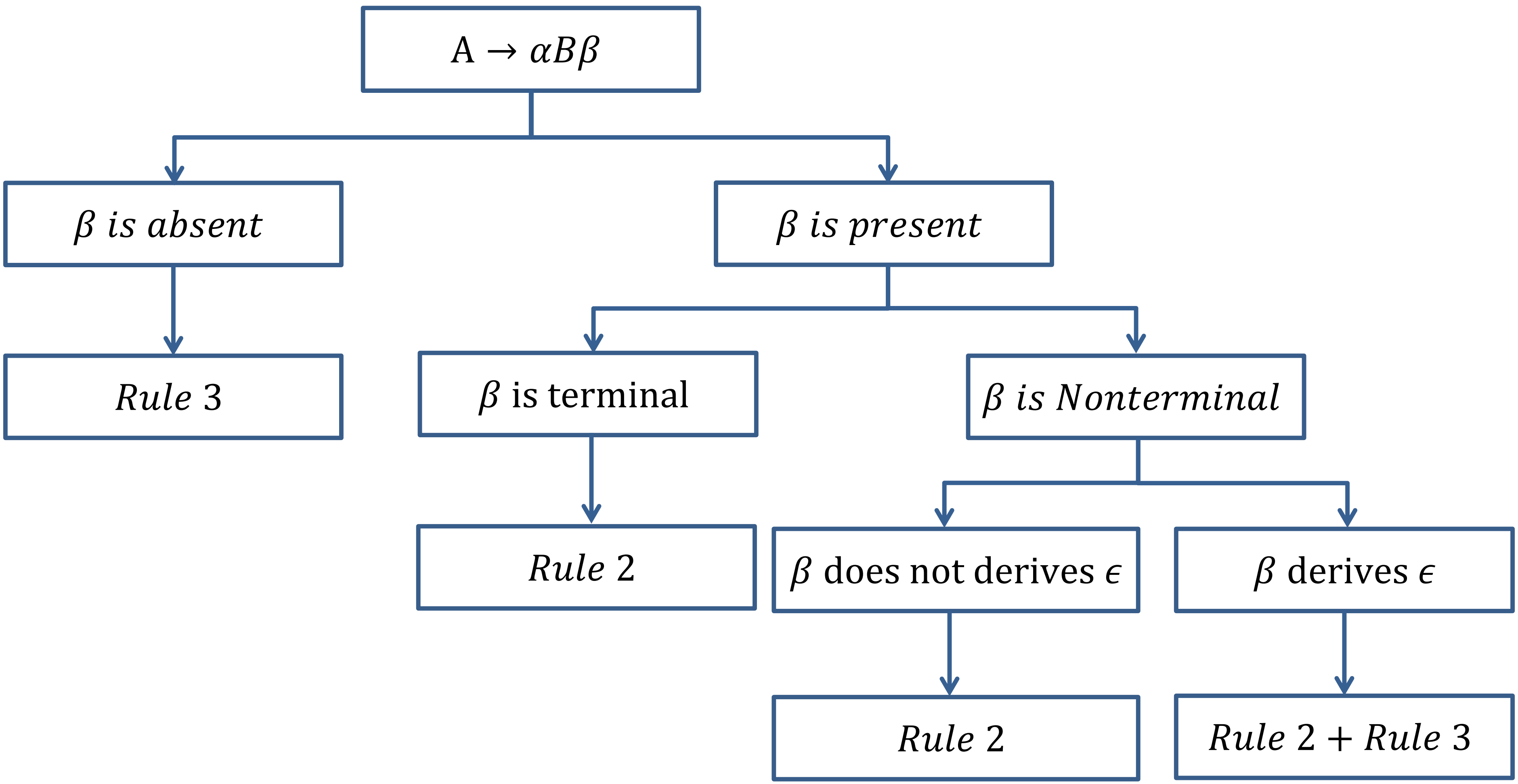
$S \to xyz \mid aBC \quad \{\$\}$
$B \to c \mid cd \quad \{e, d\}$
$C \to eg \mid df \quad \{\$\}$

First(C) = e, d

$S \to ABCDE = \{\$\}$
$A \to a \mid \epsilon \longrightarrow \{b, c\}$
$B \to b \mid \epsilon \longrightarrow \{c\}$
$C \to c \longrightarrow \{d, e, \$\}$
$D \to d \mid \epsilon \longrightarrow \{e, \$\}$
$E \to e \mid \epsilon \longrightarrow \{\$\}$.

First(B) = b, ε

How to apply rules to find FOLLOW of non terminal?



Rules to construct predictive parsing table

1. For each production $A \rightarrow \alpha$ of the grammar, do steps 2 and 3.
2. For each terminal a in $first(\alpha)$, Add $A \rightarrow \alpha$ to $M[A, a]$.
3. If ϵ is in $first(\alpha)$, Add $A \rightarrow \alpha$ to $M[A, b]$ for each terminal b in $FOLLOW(B)$. If ϵ is in $first(\alpha)$, and \$ is in $FOLLOW(A)$, add $A \rightarrow \alpha$ to $M[A, \$]$.
4. Make each undefined entry of M be error.

Example: LL(1) parsing

E→E+T | T

T→T*F | F

F→(E) | id

Step 1: Remove left recursion

E→TE'

E'→+TE' | ϵ

T→FT'

T'→*FT' | ϵ

F→(E) | id

Example: LL(1) parsing

Step 2: Compute FIRST

First(E)

E→TE'

E	**→**	**T**	**E'**
A	**→**	**Y_1**	**Y_2**

Rule 3
First(A)=First(Y1)

FIRST(E)=FIRST(T) = {(, id }

First(T)

T→FT'

T	**→**	**F**	**T'**
A	**→**	**Y_1**	**Y_2**

Rule 3
First(A)=First(Y1)

FIRST(T)=FIRST(F) = {(, id }

First(F)

F→(E)

F	**→**	**(**	**E**	**)**
A	**→**	α		

Rule 1
add α to $FIRST(A)$

FIRST(F)={ (, id }

F→id

F	**→**	**id**
A	**→**	α

Rule 1
add α to $FIRST(A)$

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

NT	First
E	
E'	
T	
T'	
F	

Example: LL(1) parsing

Step 2: Compute FIRST

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

First(E')

E'→+TE'

E'	→	+	T	E'
A	→	α		

Rule 1
add α to $FIRST(A)$

E'→ϵ

E'	→	ϵ
A	→	ϵ

Rule 2
add ϵ to $FIRST(A)$

NT	First
E	{ (,id }
E'	
T	{ (,id }
T'	
F	{ (,id }

FIRST(E')={ + , ϵ }

Example: LL(1) parsing

Step 2: Compute FIRST

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

First(T')

T'→*FT'

T'	**→**	*	**F**	**T'**
A	**→**	α		

Rule 1
add α to $FIRST(A)$

T'→ϵ

T'	**→**	ϵ
A	**→**	ϵ

Rule 2
add ϵ to $FIRST(A)$

NT	First
E	{ (,id }
E'	{ +, ϵ }
T	{ (,id }
T'	
F	{ (,id }

FIRST(T')={ * , ϵ }

Example: LL(1) parsing

Step 2: Compute FOLLOW

FOLLOW(E)

Rule 1: Place $ in FOLLOW(E)

F→(E)

F	**→**	**(**	**E**	**)**
A	**→**	**α**	**B**	**β**

Rule 2

FOLLOW(E)={ $,) }

E→TE’
E’→+TE’ | ϵ
T→FT’
T’→*FT’ | ϵ
F→(E) | id

NT	First	Follow
E	{ (,id }	
E’	{ +, ϵ }	
T	{ (,id }	
T’	{ *, ϵ }	
F	{ (,id }	

Example: LL(1) parsing

Step 2: Compute FOLLOW

FOLLOW(E’)

E→TE’

E	→	T	E’
A	→	α	B

Rule 3

E’→+TE’

E’	→	+T	E’
A	→	α	B

Rule 3

FOLLOW(E’)={ $,) }

E→TE’
E’→+TE’ | ϵ
T→FT’
T’→*FT’ | ϵ
F→(E) | id

NT	First	Follow
E	{ (,id }	{ $,) }
E’	{ +, ϵ }	
T	{ (,id }	
T’	{ *, ϵ }	
F	{ (,id }	

Example: LL(1) parsing

Step 2: Compute FOLLOW

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

FOLLOW(T)

E→TE'

E	**→**		**T**	**E'**
A	**→**	α	**B**	**β**

Rule 2

E	**→**		**T**	**E'**
A	**→**	α	**B**	**β**

Rule 3

NT	**First**	**Follow**
E	{ (,id }	{ $,) }
E'	{ +, ϵ }	{ $,) }
T	{ (,id }	
T'	{ *, ϵ }	
F	{ (,id }	

FOLLOW(T)={ +, $,)

Example: LL(1) parsing

Step 2: Compute FOLLOW

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

FOLLOW(T)

E'→+TE'

E'	→	+	T	E'
A	→	α	B	β

Rule 2

E'	→	+	T	E'
A	→	α	B	β

Rule 3

NT	First	Follow
E	{ (,id }	{ $,) }
E'	{ +, ϵ }	{ $,) }
T	{ (,id }	
T'	{ *, ϵ }	
F	{ (,id }	

FOLLOW(T)={ +, $,) }

Example: LL(1) parsing

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

Step 2: Compute FOLLOW

FOLLOW(T')

T→FT'

T	→	F	T'
A	→	α	B

Rule 3

T'→*FT'

T'	→	*F	T'
A	→	α	B

Rule 3

NT	First	Follow
E	{ (,id }	{ $,) }
E'	{ +, ϵ }	{ $,) }
T	{ (,id }	{ +,$,) }
T'	{ *, ϵ }	
F	{ (,id }	

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

Example: LL(1) parsing

Step 2: Compute FOLLOW

FOLLOW(F)

T→FT'

T	→		F	T'
A	→	α	B	β

Rule 2

T	→		F	T'
A	→	α	B	β

Rule 3

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

NT	First	Follow
E	{ (,id }	{ $,) }
E'	{ +, ϵ }	{ $,) }
T	{ (,id }	{ +,$,) }
T'	{ *, ϵ }	{ +,$,) }
F	{ (,id }	

FOLLOW(F)={ *, + ,$,) }

Example: LL(1) parsing

Step 2: Compute FOLLOW

FOLLOW(F)

T’→*FT’

T’	→	*	F	T’
A	**→**	**α**	**B**	**β**

Rule 2

T’	→	*	F	T’
A	**→**	**α**	**B**	**β**

Rule 3

FOLLOW(F)={ *,+,$,) }

E→TE’
E’→+TE’ | ϵ
T→FT’
T’→*FT’ | ϵ
F→(E) | id

NT	First	Follow
E	{ (,id }	{ $,) }
E’	{ +, ϵ }	{ $,) }
T	{ (,id }	{ +,$,) }
T’	{ *, ϵ }	{ +,$,) }
F	{ (,id }	

Example: LL(1) parsing

Step 3: Construct predictive parsing table

NT	Input Symbol					
	id	+	*	(	)	$
E						
E’						
T						
T’						
F						

NT	First	Follow
E	{ (,id }	{ $,) }
E’	{ +, ϵ }	{ $,) }
T	{ (,id }	{ +,$,) }
T’	{ *, ϵ }	{ +,$,) }
F	{ (,id }	{*,+,$,)}

E→TE’

a=FIRST(TE’)={ (,id }

M[E,(]=E→TE’

M[E,id]=E→TE’

Rule: 2

A→ α

a = first(α)

M[A,a] = A→ α

E→TE’

E’→+TE’ | ϵ

T→FT’

T’→*FT’ | ϵ

F→(E) | id

Example: LL(1) parsing

Step 3: Construct predictive parsing table

NT	Input Symbol					
	id	+	*	(	)	$
E	E→TE’			E→TE’		
E’						
T						
T’						
F						

NT	First	Follow
E	{ (,id }	{ $,) }
E’	{ +, ϵ }	{ $,) }
T	{ (,id }	{ +,$,) }
T’	{ *, ϵ }	{ +,$,) }
F	{ (,id }	{*,+,$,)}

E’→+TE’

a=FIRST(+TE’)={ + }

M[E’,+]=E’→+TE’

Rule: 2

A→ α

a = first(α)

M[A,a] = A→ α

E→TE’

E’→+TE’ | ϵ

T→FT’

T’→*FT’ | ϵ

F→(E) | id

Example: LL(1) parsing

Step 3: Construct predictive parsing table

NT	Input Symbol					
	id	+	*	(	)	$
E	E→TE’			E→TE’		
E’		E’→+TE’				
T						
T’						
F						

NT	First	Follow
E	{ (,id }	{ $,) }
E’	{ +, ϵ }	{ $,) }
T	{ (,id }	{ +,$,) }
T’	{ *, ϵ }	{ +,$,) }
F	{ (,id }	{*,+,$,)}

E’→ϵ

b=FOLLOW(E’)={ $,) }

M[E’,$]=E’→$\epsilon$

M[E’,)]=E’→ϵ

Rule: 3

A→ α

b = follow(A)

M[A,b] = A→ α

E→TE’

E’→+TE’ | ϵ

T→FT’

T’→*FT’ | ϵ

F→(E) | id

Example: LL(1) parsing

Step 3: Construct predictive parsing table

NT	Input Symbol					
	id	+	*	(	)	$
E	E→TE'			E→TE'		
E'		E'→+TE'			E'→ϵ	E'→ϵ
T						
T'						
F						

NT	First	Follow
E	{ (,id }	{ $,) }
E'	{ +, ϵ }	{ $,) }
T	{ (,id }	{ +,$,) }
T'	{ *, ϵ }	{ +,$,) }
F	{ (,id }	{*,+,$,)}

T→FT'

a=FIRST(FT')={ (,id }

M[T,(]=T→FT'

M[T,id]=T→FT'

Rule: 2

A→ α

a = first(α)

M[A,a] = A→ α

E→TE'

E'→+TE' | ϵ

T→FT'

T'→*FT' | ϵ

F→(E) | id

Example: LL(1) parsing

Step 3: Construct predictive parsing table

NT	Input Symbol					
	id	+	*	(	)	$
E	E→TE’			E→TE’		
E’		E’→+TE’			E’→ϵ	E’→ϵ
T	T→FT’			T→FT’		
T’						
F						

NT	First	Follow
E	{ (,id }	{ $,) }
E’	{ +, ϵ }	{ $,) }
T	{ (,id }	{ +,$,) }
T’	{ *, ϵ }	{ +,$,) }
F	{ (,id }	{*,+,$,)}

T’→*FT’

a=FIRST(*FT’)={ * }

M[T’,*]=T’→*FT’

Rule: 2

A→ α

a = first(α)

M[A,a] = A→ α

E→TE’
E’→+TE’ | ϵ
T→FT’
T’→*FT’ | ϵ
F→(E) | id

Example: LL(1) parsing

Step 3: Construct predictive parsing table

NT	Input Symbol					
	id	**+**	*****	**(**	**)**	**$**
E	E→TE’			E→TE’		
E’		E’→+TE’			E’→ϵ	E’→ϵ
T	T→FT’			T→FT’		
T’			T’→*FT’			
F						

NT	First	Follow
E	{ (,id }	{ $,) }
E’	{ +, ϵ }	{ $,) }
T	{ (,id }	{ +,$,) }
T’	{ *, ϵ }	{ +,$,) }
F	{ (,id }	{*,+,$,)}

T’→ϵ

b=FOLLOW(T’)={ +,$,) }

M[T’,+]=T’→ϵ

M[T’,$]=T’→ϵ

M[T’,)]=T’→ϵ

Rule: 3
A→ α
b = follow(A)
M[A,b] = A→ α

E→TE’
E’→+TE’ | ϵ
T→FT’
T’→*FT’ | ϵ
F→(E) | id

Example: LL(1) parsing

Step 3: Construct predictive parsing table

NT	Input Symbol					
	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						

NT	First	Follow
E	{ (,id }	{ $,) }
E'	{ +, ϵ }	{ $,) }
T	{ (,id }	{ +,$,) }
T'	{ *, ϵ }	{ +,$,) }
F	{ (,id }	{*,+,$,)}

F→(E)

a=FIRST((E))={ (}

M[F,(]=F→(E)

Rule: 2

A→ α

a = first(α)

M[A,a] = A→ α

E→TE'

E'→+TE' | ϵ

T→FT'

T'→*FT' | ϵ

F→(E) | id

Example: LL(1) parsing

Step 3: Construct predictive parsing table

NT	Input Symbol					
	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→(E)		

NT	First	Follow
E	{ (,id }	{ $,) }
E’	{ +, ϵ }	{ $,) }
T	{ (,id }	{ +,$,) }
T’	{ *, ϵ }	{ +,$,) }
F	{ (,id }	{*,+,$,)}

F→id

a=FIRST(id)={ id }

M[F,id]=F→id

Rule: 2

A→ α

a = first(α)

M[A,a] = A→ α

E→TE’

E’→+TE’ | ϵ

T→FT’

T’→*FT’ | ϵ

F→(E) | id

Example: LL(1) parsing

Step 4: Make each undefined entry of table be Error

NT	Input Symbol					
	id	+	*	(	)	$
E	E→TE’	Error	Error	E→TE’	Error	Error
E’	Error	E’→+TE’	Error	Error	E’→ϵ	E’→ϵ
T	T→FT’	Error	Error	T→FT’	Error	Error
T’	Error	T’→ϵ	T’→*FT’	Error	T’→ϵ	T’→ϵ
F	F→id	Error	Error	F→(E)	Error	Error

NT	First	Follow
E	{ (,id }	{ $,) }
E’	{ +, ϵ }	{ $,) }
T	{ (,id }	{ +,$,) }
T’	{ *, ϵ }	{ +,$,) }
F	{ (,id }	{*,+,$,)}

E→TE’
E’→+TE’ | ϵ
T→FT’
T’→*FT’ | ϵ
F→(E) | id

Example: LL(1) parsing

Step 4: Parse the string : id + id * id $

STACK	INPUT	OUTPUT
E$	id+id*id$	

NT	Input Symbol					
	id	+	*	(	)	$
E	E→TE'	Error	Error	E→TE'	Error	Error
E'	Error	E'→+TE'	Error	Error	E'→ϵ	E'→ϵ
T	T→FT'	Error	Error	T→FT'	Error	Error
T'	Error	T'→ϵ	T'→*FT'	Error	T'→ϵ	T'→ϵ
F	F→id	Error	Error	F→(E)	Error	Error

When Is a Grammar LL(1)?

When Is a Grammar LL(1)?

A grammar is LL(1) iff for each set of productions where $A \rightarrow \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_n$, the following conditions hold.

1. $FIRST(\alpha_i) \cap FIRST(\alpha_j) = \varnothing$ where $1 \leq i \leq n$ and $1 \leq j \leq n$ and $i \neq j$
2. If $\alpha_i \stackrel{*}{\Rightarrow} \varepsilon$ then
   a. $\alpha_1,\ldots,\alpha_{i-1},\alpha_{i+1},\ldots,\alpha_n$ does not $\stackrel{*}{\Rightarrow} \varepsilon$
   b. $FIRST(\alpha_j) \cap FOLLOW(A) = \varnothing$ where $j \neq i$ and $1 \leq j \leq n$

Example : Checking If Grammar is LL(1) or not

Checking If a Grammar is LL(1)

Production	FIRST	FOLLOW
$S \rightarrow iEtSS' \mid a$	{ i, a }	{ e, $ }
$S' \rightarrow eS \mid \varepsilon$	{ e, ε }	{ e, $ }
$E \rightarrow b$	{ b }	{ t }

Nonterminal	a	b	e	i	t	$
S	$S \rightarrow a$			$S \rightarrow iEtSS'$		
S'			$S' \rightarrow eS$ $S' \rightarrow \varepsilon$			$S' \rightarrow \varepsilon$
E		$E \rightarrow b$				

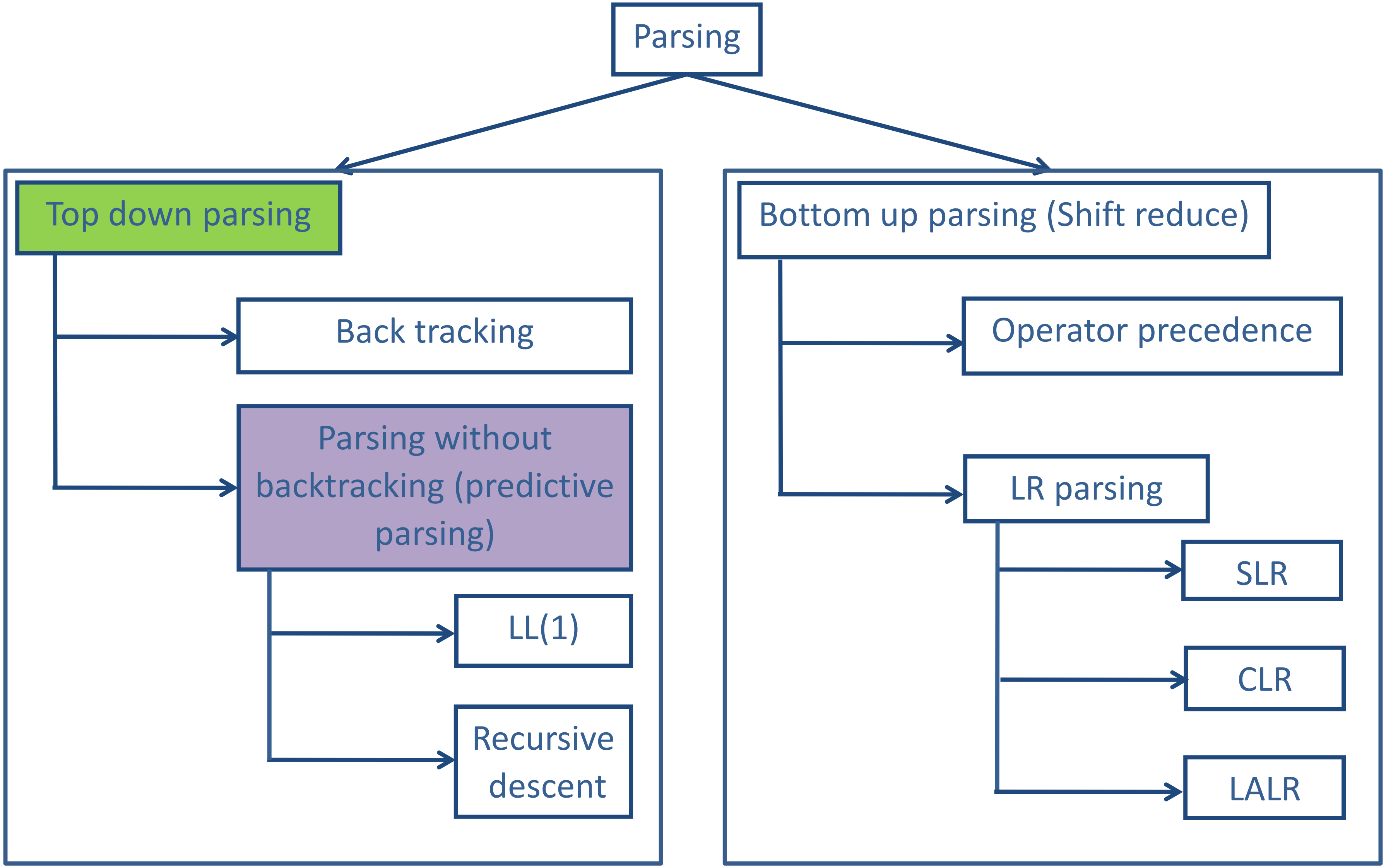
So this grammar is not LL(1).

Given Grammar is not LL(1) as S' has more than two row under terminal e

Exercise

S → AaAb | BbBa A→ ϵ B→ ϵ	S→ aAB | bA | ϵ A→ aAb | ϵ B→ bB | ϵ	S→iCtSA | a A→ eS | ϵ C→ b
S→ (L) | a L→ L,S | S	E→ TA A→ +TA | ϵ T→ VB B→ *VB |ϵ V→ id | (E)	S→ a | ^ | (R) T→ S, T | S R→ T

Parsing methods



Recursive descent parsing

- A top down parsing that executes a set of recursive procedure to process the input without backtracking is called recursive descent parser.
- There is a procedure for each non terminal in the grammar.
- Consider RHS of any production rule as definition of the procedure.
- As it reads expected input symbol, it advances input pointer to next position.

Example: Recursive descent parsing

```
{
        If lookahead=num
        {
                Match(num);
                T();
        }
        Else
                Error();
        If lookahead=$
        {
                Declare success;
        }
        Else
                Error();
}
```

```
{
        If lookahead='*'
        {
                Match('*');
                If lookahead=num
                {
                        Match(num);
                        T();
                }
                Else
                        Error();

        }
        Else
                NULL
}
```

```
Proceduce Match(token t)
{
    If lookahead=t
    lookahead=next_token;
    Else
        Error();
}
```

```
Procedure Error
{
        Print("Error");
}
```

$E \rightarrow num\ T$

$T \rightarrow *\ num\ T \mid \epsilon$

3	*	4	$

Success

Example: Recursive descent parsing

```
Procedure E
{
        If lookahead=num
        {
                Match(num);
                T();
        }
        Else
                Error();
        If lookahead=$
        {
                Declare success;
        }
        Else
                Error();
}
```

```
Procedure T
{
        If lookahead='*'
        {
                Match('*');
                If lookahead=num
                {
                        Match(num);
                        T();
                }
                Else
                        Error();

        }
        Else
                NULL
}
```

```
Proceduce Match(token t)
{
    If lookahead=t
    lookahead=next_token;
    Else
        Error();
}
```

```
Procedure Error
{
        Print("Error");
}
```

$E \rightarrow num\ T$

$T \rightarrow *\ num\ T \mid \epsilon$

3	*	4	$

Success

3	4	*	$

Error

END OF

TOP DOWN PARSING

PARSING