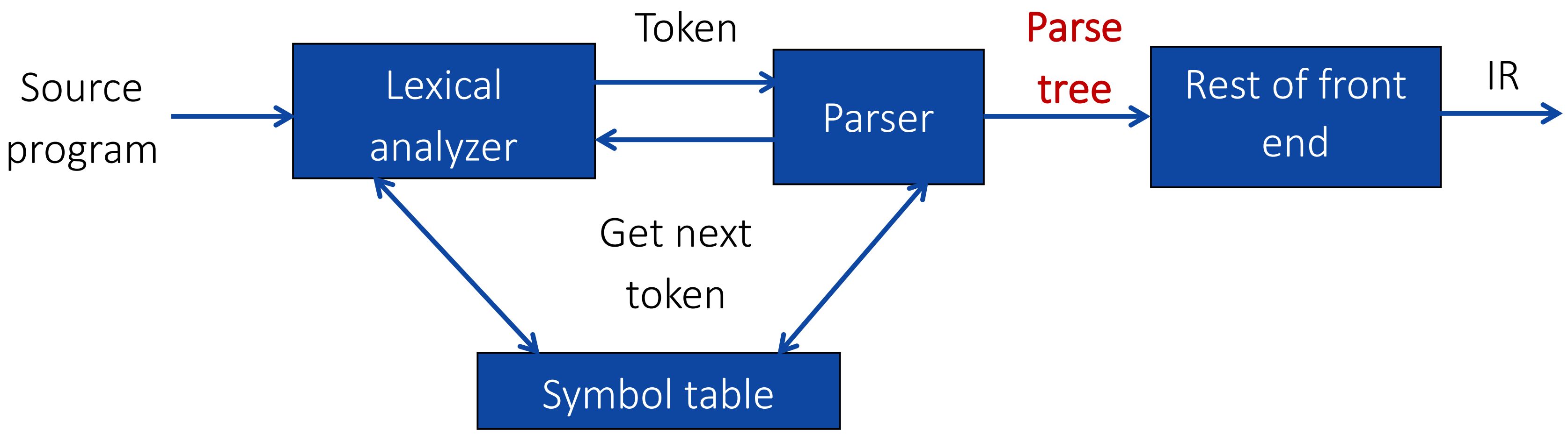
Module 2 – Syntax Analysis

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

- 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 $\alpha \in (V \cup \Sigma)^*$

- Nonterminal symbol:
  - The name of syntax category of a language, e.g., noun, verb, etc.
  - The It is written as a **single capital letter**, or as a **name enclosed between < … >,** e.g., A or <Noun>

<Noun Phrase> → <Article><Noun>

<Article> → a | an | the

<Noun> → boy | apple

Context free grammar

- 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 $\alpha \in (V \cup \Sigma)^*$

- Terminal symbol:
  - A symbol in the alphabet.
  - It is denoted by lower case letter and punctuation marks used in language.

<Noun Phrase> → <Article><Noun>
<Article> → a | an | the
<Noun> → boy | apple

Context free grammar

- 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 $\alpha \in (V \cup \Sigma)^*$

- Start symbol:
  - First nonterminal symbol of the grammar is called start symbol.

<Noun Phrase> → <Article><Noun>
<Article> → a | an | the
<Noun> → boy | apple

Context free grammar

- 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 $\alpha \in (V \cup \Sigma)^*$

- Production:
  - A production, also called a rewriting rule, is a rule of grammar. It has the form of

    A nonterminal symbol → String of terminal and nonterminal symbols

    <Noun Phrase> → <Article><Noun>

    <Article> → a | an | the

    <Noun> → boy | apple

Example: Grammar

Write terminals, non terminals, start symbol, and productions for following grammar.

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

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

Terminals: **id + - * / ↑ ( )**

Non terminals: E, O

Start symbol: E

Productions: $E \rightarrow E\ O\ E \mid (E) \mid -E \mid id$

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

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**

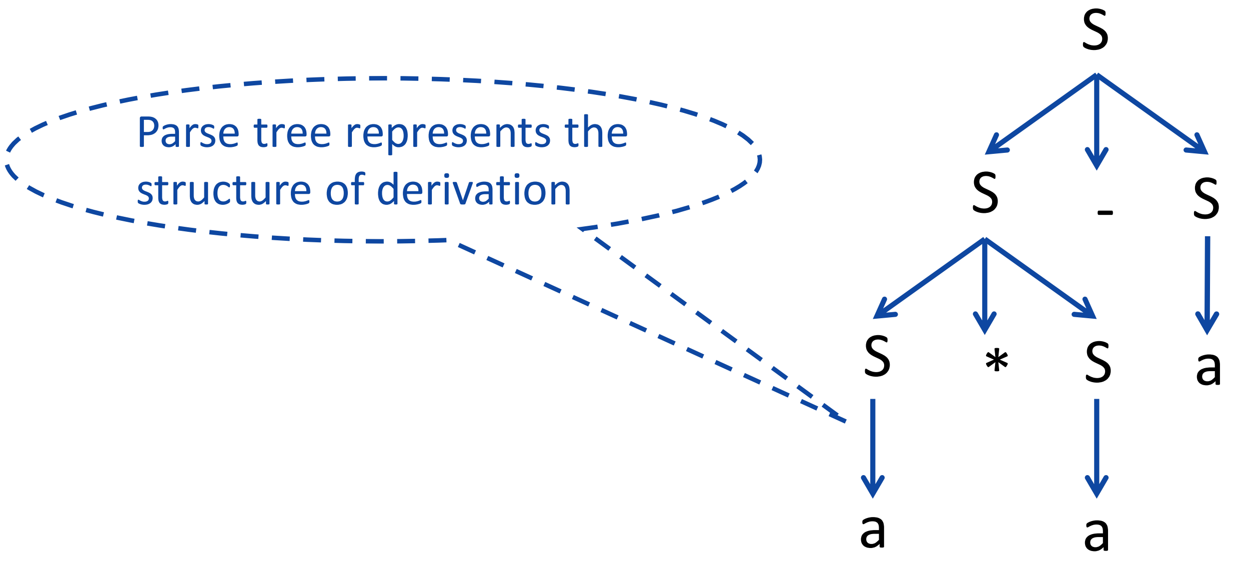
→**S*S**-S

→**a***S-S

→a***a**-S

→a*a-**a**

Leftmost Derivation



Parse tree

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**

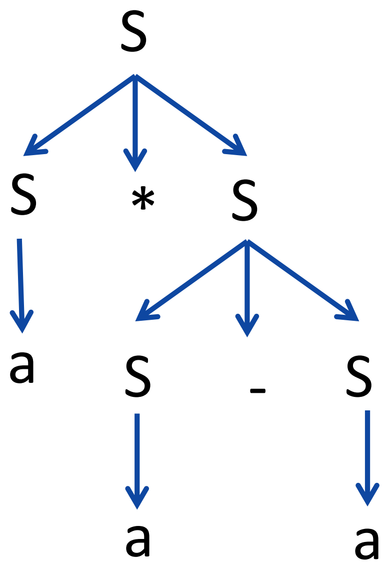
→S***S-S**

→S*S-**a**

→S***a**-a

→**a***a-a

Rightmost Derivation



Parse Tree

Exercise: Derivation

1. Perform leftmost derivation and draw parse tree.

   S→A1B

   A→0A | ϵ

   B→0B | 1B | ϵ

   Output string: 1001

2. Perform leftmost derivation and draw parse tree.

   S→0S1 | 01   Output string: 000111

3. Perform rightmost derivation and draw parse tree.

   E→E+E | E*E | id | (E) | -E

   Output string: id + id * id

Ambiguity

- Ambiguity, is a word, phrase, or statement which contains more than one meaning.

Chip

A long thin piece of potato

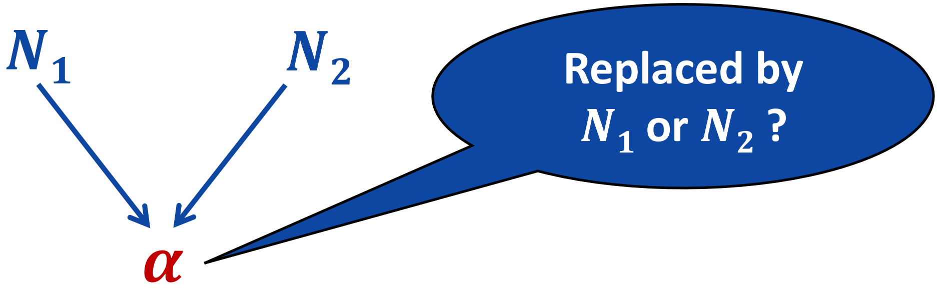
A small piece of silicon

Ambiguity

- In formal language grammar, ambiguity would arise if identical string can occur on the RHS of two or more productions.
- Grammar:

  $N1 \rightarrow \alpha$

  $N2 \rightarrow \alpha$

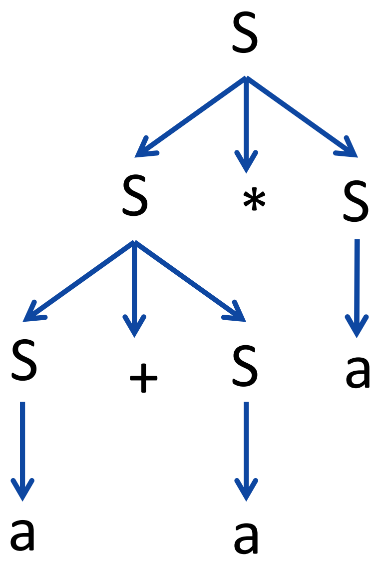


- α can be derived from either N1 or N2

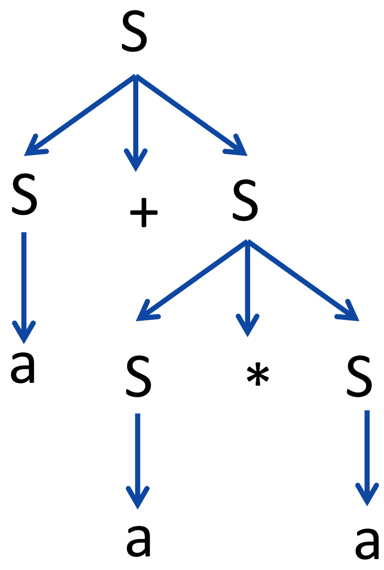
Ambiguous grammar

- 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.

Exercise: Ambiguous Grammar

Check Ambiguity in following grammars:

1. S→ aS | Sa | ϵ (output string: aaaa)
2. S→ aSbS | bSaS | ϵ (output string: abab)
3. S→ SS+ | SS* | a (output string: aa+a*)
4. <exp> → <exp> + <term> | <term>

   <term> → <term> * <letter> | <letter>

   <letter> → a|b|c|…|z (output string: a+b*c)
5. Prove that the CFG with productions: S → a | Sa | bSS | SSb | SbS is ambiguous (Hint: consider output string yourself)

Left recursion & Left factoring

Left recursion

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

$A \to A\alpha \mid$ $\Longrightarrow$ $A \to A'$

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

Examples: Left recursion elimination

$E \rightarrow E+T \mid T$

$E \rightarrow TE'$

$E' \rightarrow +TE' \mid \varepsilon$

$T \rightarrow T*F \mid F$

$T \rightarrow FT'$

$T' \rightarrow *FT' \mid \varepsilon$

$X \rightarrow X\%Y \mid Z$

$X \rightarrow ZX'$

$X' \rightarrow \%YX' \mid \varepsilon$

Exercise: Left recursion

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.

$S \rightarrow aAB \mid aCD$

$S \rightarrow aS'$
$S' \rightarrow AB \mid CD$

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

$A \rightarrow xByAA' \mid a$
$A' \rightarrow \in \mid zA$

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

$A \rightarrow aA'$
$A' \rightarrow AB \mid A \mid \epsilon$
$A' \rightarrow AA'' \mid \epsilon$
$A'' \rightarrow B \mid \epsilon$

Exercise

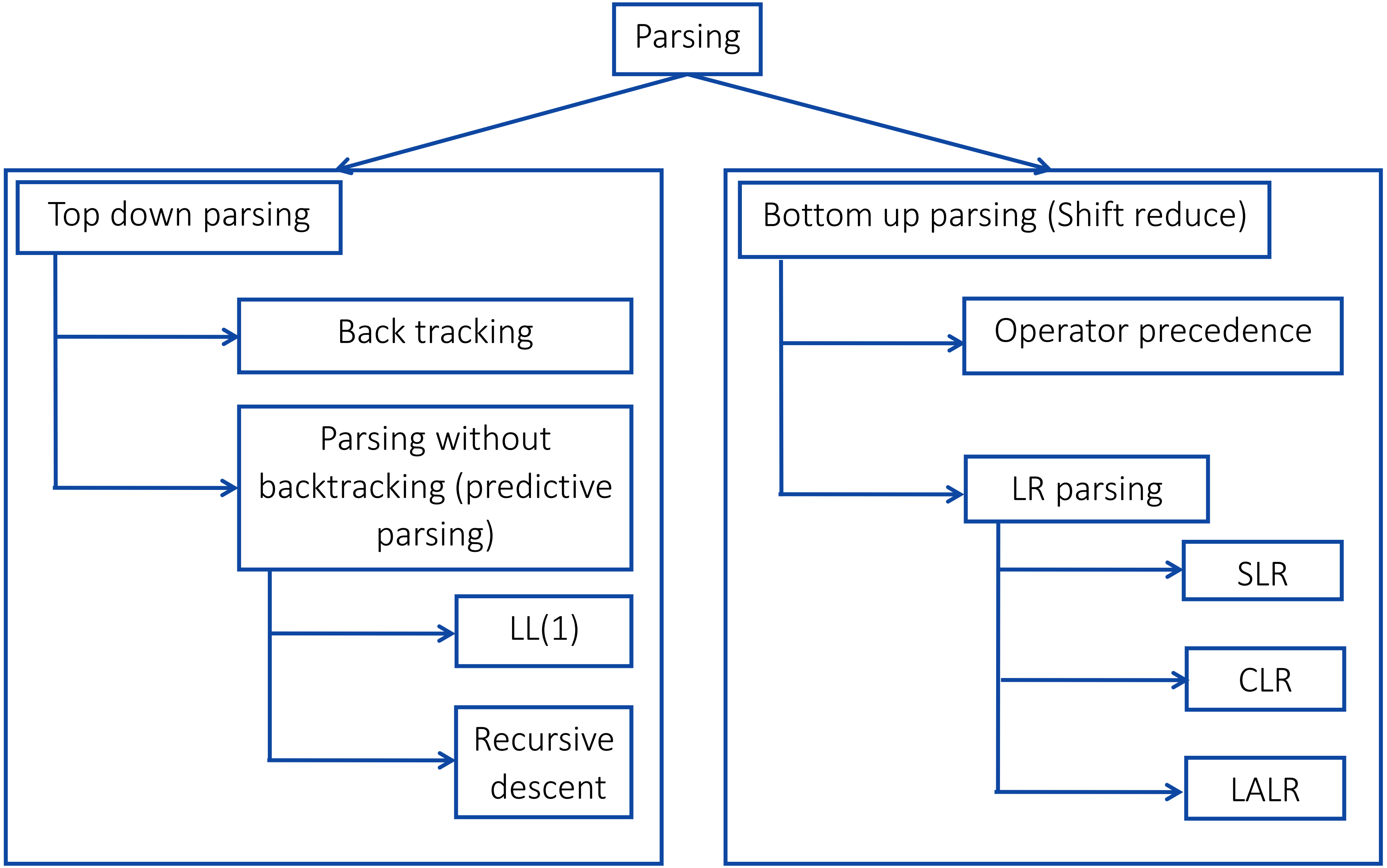
1. S→iEtS | iEtSeS | a
2. A→ ad | a | ab | abc | x

Parsing

- Parsing 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.

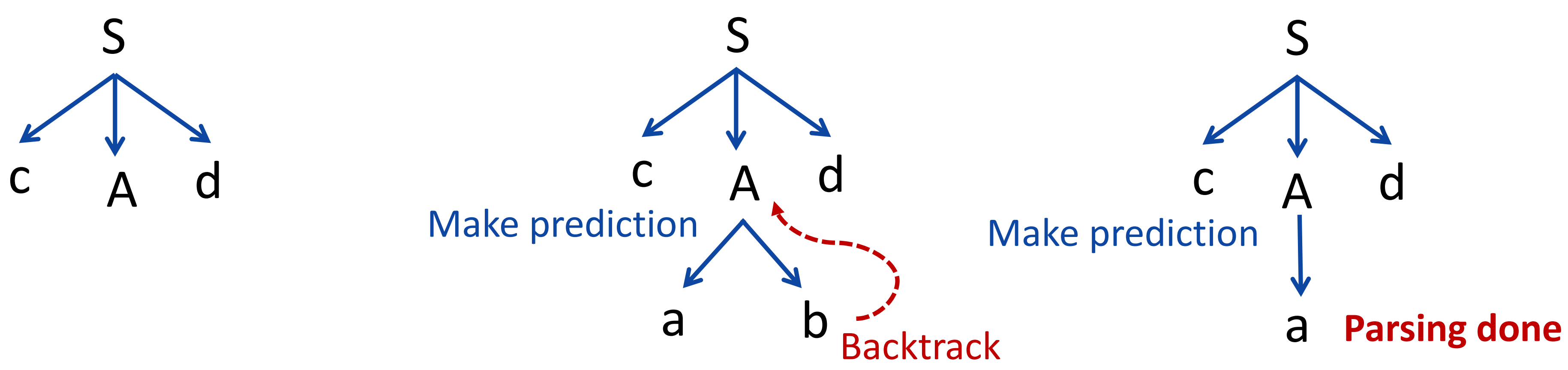
Classification of parsing methods



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

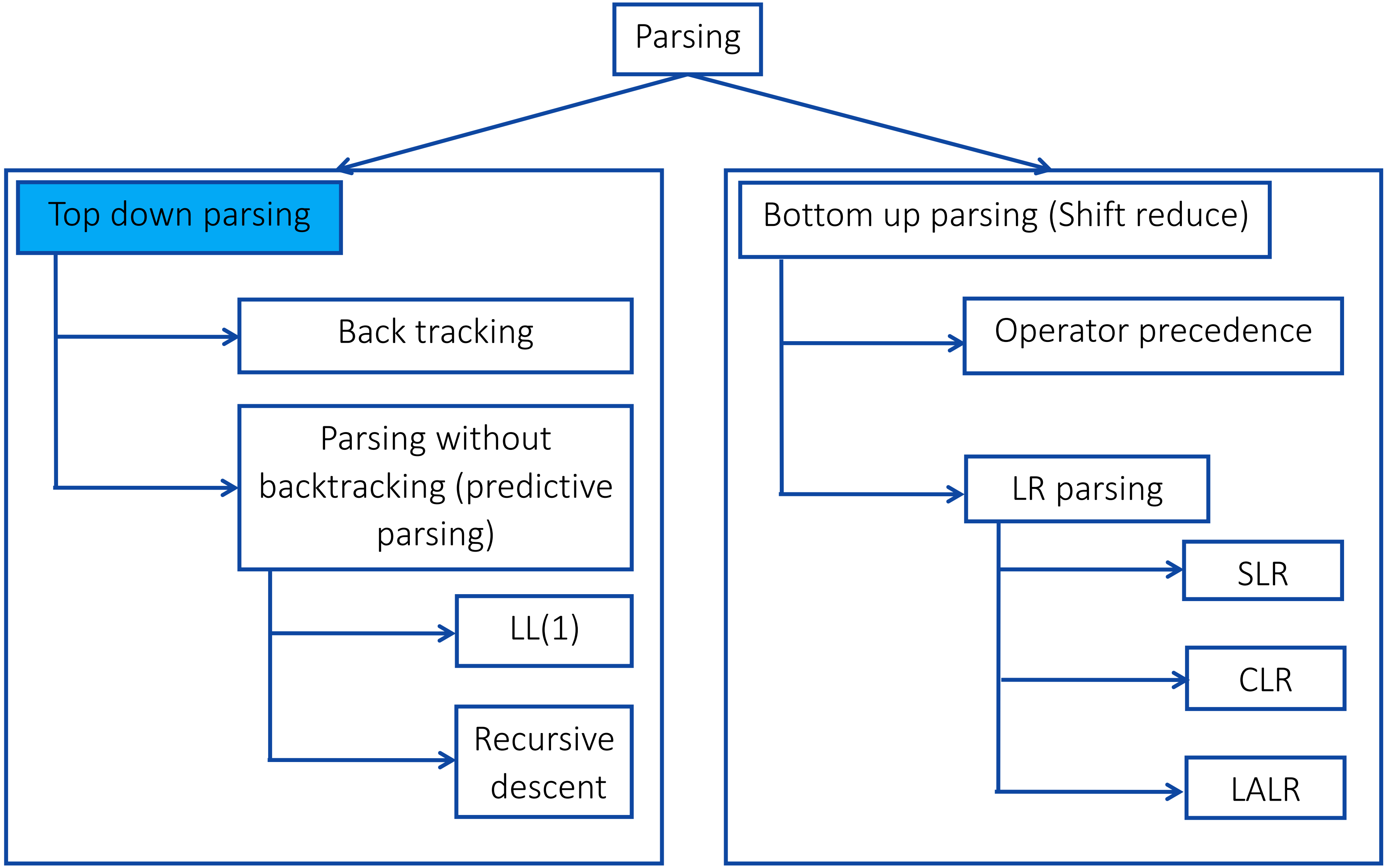
1. E→ 5+T | 3-T

   T→ V | V*V | V+V

   V→ a | b

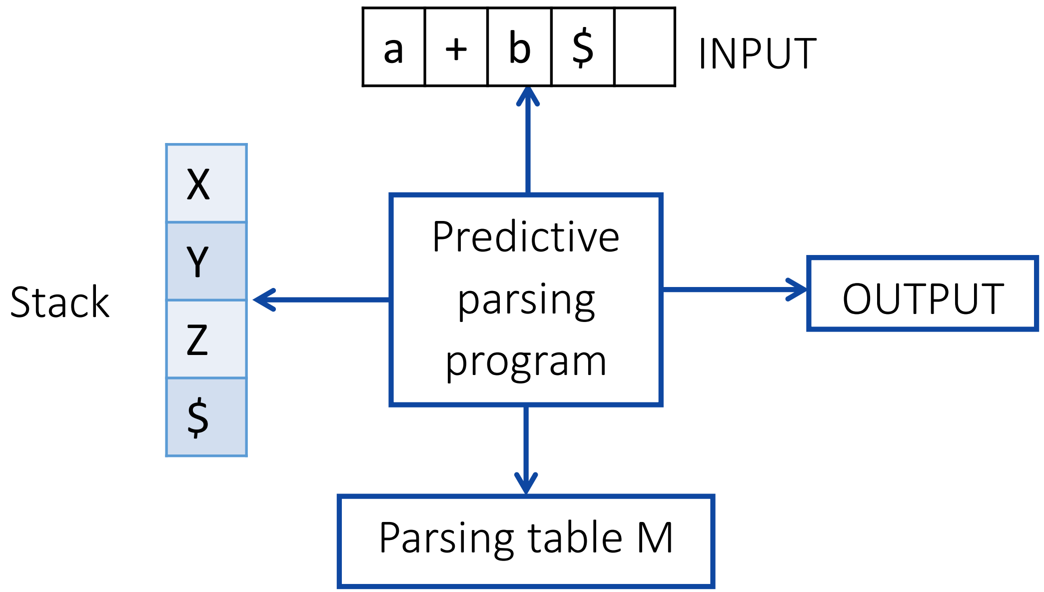
   String: 3-a+b

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 $Y_1 \ldots Y_{i-1} \Rightarrow \epsilon$. 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 $Y_1 \Rightarrow \epsilon$, then we add $FIRST(Y_2)$ and so on.

Rules to compute first of non terminal

Simplification of Rule 3

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

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

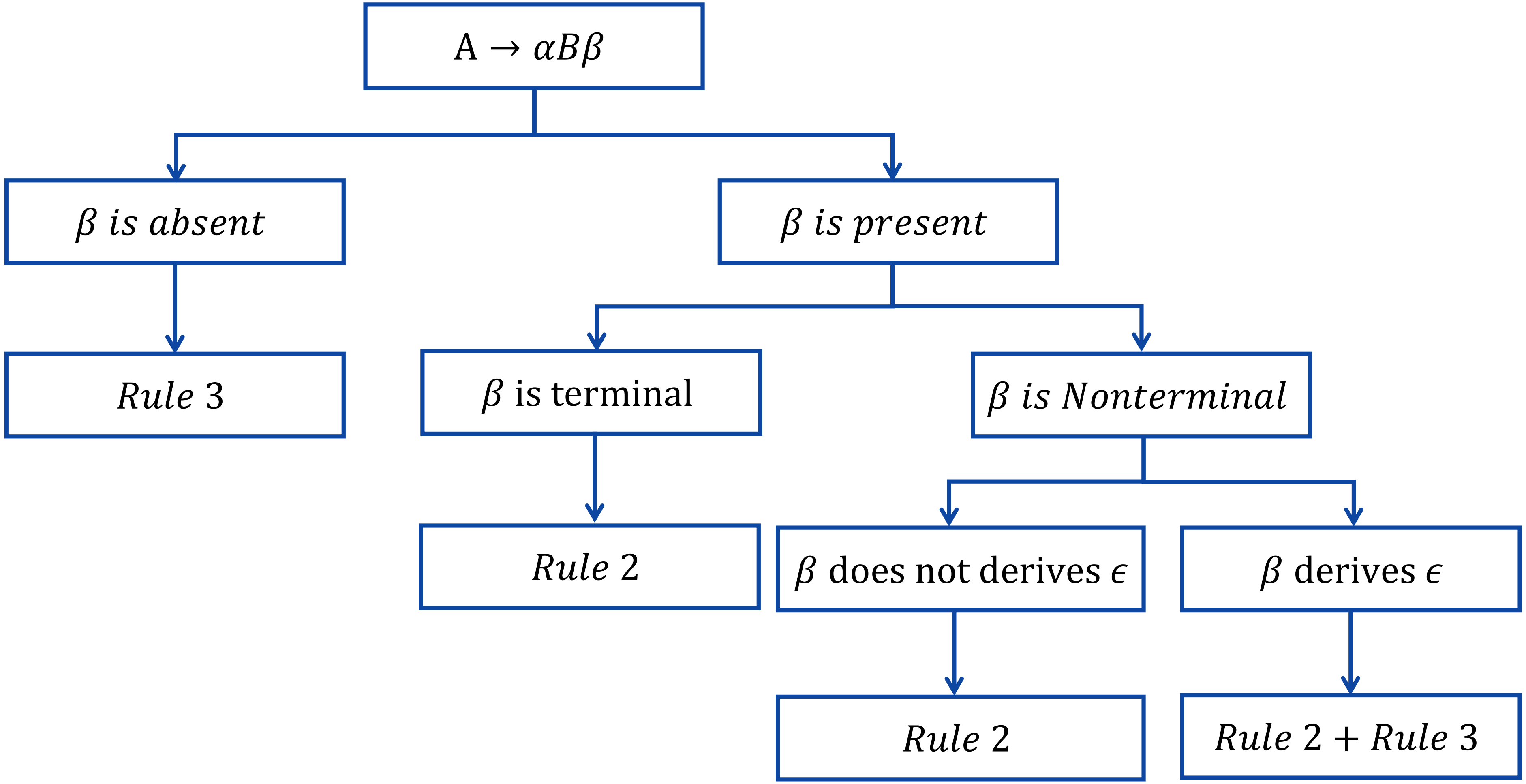
  $FIRST(A) = FIRST(Y_1) - \epsilon\ U\ FIRST(Y_2) - \epsilon\ U\ FIRST(Y_3) - \ \epsilon\ U\ FIRST(Y_4)$
- If Y_1 , Y_2, Y_3 …..Y_K all derives ∈ $then,$

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

Rules to compute FOLLOW of non terminal

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

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-1: LL(1) parsing

S→aBa
B→bB | ϵ

NT	First
S	
B	

Step 1: Not required

Step 2: Compute FIRST

First(S)

S→aBa

S	**→**	**a**	**B**	**a**
A	**→**	α		

Rule 1
add α to $FIRST(A)$

FIRST(S)={ a }

First(B)

B→bB

B	**→**	**b**	**B**
A	**→**	α	

Rule 1
add α to $FIRST(A)$

B→ϵ

B	**→**	ϵ
A	**→**	ϵ

Rule 2
add ϵ to $FIRST(A)$

FIRST(B)={ b , ϵ }

Example-1: LL(1) parsing

S→aBa
B→bB | ϵ

NT	First	Follow
S	{a}	
B	{b,ϵ}	

Step 2: Compute FOLLOW

Follow(S)

Rule 1: Place \$ in FOLLOW(S)

Follow(S)={ \$ }

Follow(B)

S→aBa

S	→	a	B	a
A	→	α	B	β

Rule 2
First(β) $-\ \epsilon$

B→bB

B	→	b	B
A	→	α	B

Rule 3
Follow(A)=follow(B)

Follow(B)={ a }

Example-1: LL(1) parsing

$S \rightarrow aBa$
$B \rightarrow bB \mid \epsilon$

NT	First	Follow
S	{a}	{$}
B	{b,ϵ}	{a}

Step 3: Prepare predictive parsing table

NT	Input Symbol		
	a	**b**	**$**
S			
B			

$S \rightarrow aBa$

a=FIRST(aBa)={ a }

M[S,a]=S→aBa

Rule: 2
$A \rightarrow \alpha$
a = first(α)
M[A,a] = $A \rightarrow \alpha$

Example-1: LL(1) parsing

$S \rightarrow aBa$
$B \rightarrow bB \mid \epsilon$

NT	First	Follow
S	{a}	{$}
B	{b,ϵ}	{a}

Step 3: Prepare predictive parsing table

NT	Input Symbol		
	a	b	$
S	$S \rightarrow aBa$		
B			

$B \rightarrow bB$

a=FIRST(bB)={ b }

M[B,b]=$B \rightarrow bB$

Rule: 2
$A \rightarrow \alpha$
a = first(α)
M[A,a] = $A \rightarrow \alpha$

Example-1: LL(1) parsing

$S \rightarrow aBa$
$B \rightarrow bB \mid \epsilon$

Step 3: Prepare predictive parsing table

NT	First	Follow
S	{a}	{$}
B	{b,ϵ}	{a}

NT	Input Symbol		
	a	**b**	**$**
S	S→aBa		
B		B→bB	

$B \rightarrow \epsilon$

b=FOLLOW(B)={ a }

M[B,a]=B→ϵ

Rule: 3
$A \rightarrow \alpha$
b = follow(A)
M[A,b] = $A \rightarrow \alpha$

Exercise: LL(1) parsing

S→aB | ϵ
B→bC | ϵ
C→cS | ϵ

E→E+T | T
T→T*F | F
F→(E) | id

, ϵ }

Q1: Find out whether the following grammar is LL(1):

$$S \rightarrow aSbS \mid bSaS \mid \varepsilon$$

Sol. FIRST(S): $\{a, b, \varepsilon\}$

FOLLOW(S): $\{\$, b, a\}$

Not LL(1) Grammar

	a	b	\$
S	$S \rightarrow aSbS$ $S \rightarrow \varepsilon$	$S \rightarrow bSaS$ $S \rightarrow \varepsilon$	$S \rightarrow \varepsilon$

Check whether the following Grammar is LL(1).

1.
$S \rightarrow aBa$
$B \rightarrow bB \mid \epsilon$

2.
$S \rightarrow aSbS \mid bSaS \mid \epsilon$

3.
$S \rightarrow (S) \mid \epsilon$

4.
$S \rightarrow iEtS \mid iEtSeS \mid a$
$E \rightarrow b$

5.
$S \rightarrow A$
$A \rightarrow Bb \mid Cd$
$B \rightarrow aB \mid \epsilon$
$C \rightarrow cC \mid \epsilon$

6.
$S \rightarrow (L) \mid a$
$L \rightarrow L,S \mid S$

Note: Ensure that each cell in the parsing table contains at most one production rule. If any cell contains more than one production rule, the grammar is not LL(1).

Q2: Find out whether the following grammar is LL(1):

$$S \rightarrow (S) \mid \varepsilon$$

Sol. FIRST(S): $\{(, \varepsilon\}$

FOLLOW(S): $\{\$,)\}$

LL(1) Grammar

	(	)	$
S	$S \rightarrow (S)$	$S \rightarrow \varepsilon$	$S \rightarrow \varepsilon$

Q3: Find out whether the following grammar is LL(1):

$$S \rightarrow AaAb \mid BbBa$$
$$A \rightarrow \varepsilon$$
$$B \rightarrow \varepsilon$$

LL(1) Grammar

Sol.

	FIRST	FOLLOW
$S \rightarrow AaAb \mid BbBa$	{a, b}	{\$}
$A \rightarrow \varepsilon$	{ε}	{a, b}
$B \rightarrow \varepsilon$	{ε}	{b, a}

	a	b	\$
S	$S \rightarrow AaAb$	$S \rightarrow BbBa$	
A	$A \rightarrow \varepsilon$	$A \rightarrow \varepsilon$	
B	$B \rightarrow \varepsilon$	$B \rightarrow \varepsilon$	

Q4: Find out whether the following grammar is LL(1):

$$S \rightarrow A \mid a$$
$$A \rightarrow a$$

Not LL(1) Grammar

Sol.

	FIRST	FOLLOW
$S \rightarrow A \mid a$	{a}	{$}
$A \rightarrow a$	{a}	{$}

Q5: Find out whether the following grammar is LL(1):

$$S \rightarrow aB \mid \varepsilon$$
$$B \rightarrow bC \mid \varepsilon$$
$$C \rightarrow cS \mid \varepsilon$$

LL(1) Grammar

Sol.

	FIRST	FOLLOW
$S \rightarrow aB \mid \varepsilon$	$\{a, \varepsilon\}$	$\{\$\}$
$B \rightarrow bC \mid \varepsilon$	$\{b, \varepsilon\}$	$\{\$\}$
$C \rightarrow cS \mid \varepsilon$	$\{c, \varepsilon\}$	$\{\$\}$

	a	b	c	\$
S	$S \rightarrow aB$			$S \rightarrow \varepsilon$
B		$B \rightarrow bC$		$B \rightarrow \varepsilon$
C			$C \rightarrow cS$	$C \rightarrow \varepsilon$

Q1: Find out whether the following grammar is LL(1):

$$S \rightarrow AB$$
$$A \rightarrow a \mid \varepsilon$$
$$B \rightarrow b \mid \varepsilon$$

LL(1) Grammar

Sol.

	FIRST	FOLLOW
$S \rightarrow AB$	$\{a, b, \varepsilon\}$	$\{\$\}$
$A \rightarrow a \mid \varepsilon$	$\{a, \varepsilon\}$	$\{b, \$\}$
$B \rightarrow b \mid \varepsilon$	$\{b, \varepsilon\}$	$\{\$\}$

	a	b	\$
S	$S \rightarrow AB$	$S \rightarrow AB$	$S \rightarrow AB$
A	$A \rightarrow a$	$A \rightarrow \varepsilon$	$A \rightarrow \varepsilon$
B		$B \rightarrow b$	$B \rightarrow \varepsilon$

Q2: Find out whether the following grammar is LL(1):

$$S \rightarrow aSA \mid \varepsilon$$
$$A \rightarrow c \mid \varepsilon$$

Not LL(1) Grammar

Sol.	FIRST	FOLLOW
$S \rightarrow aSA \mid \varepsilon$	$\{a, \varepsilon\}$	$\{\$, c\}$
$A \rightarrow c \mid \varepsilon$	$\{c, \varepsilon\}$	$\{\$, c\}$

	a	c	\$
S	$S \rightarrow aSA$	$S \rightarrow \varepsilon$	$S \rightarrow \varepsilon$
A		$A \rightarrow c$ $A \rightarrow \varepsilon$	$A \rightarrow \varepsilon$

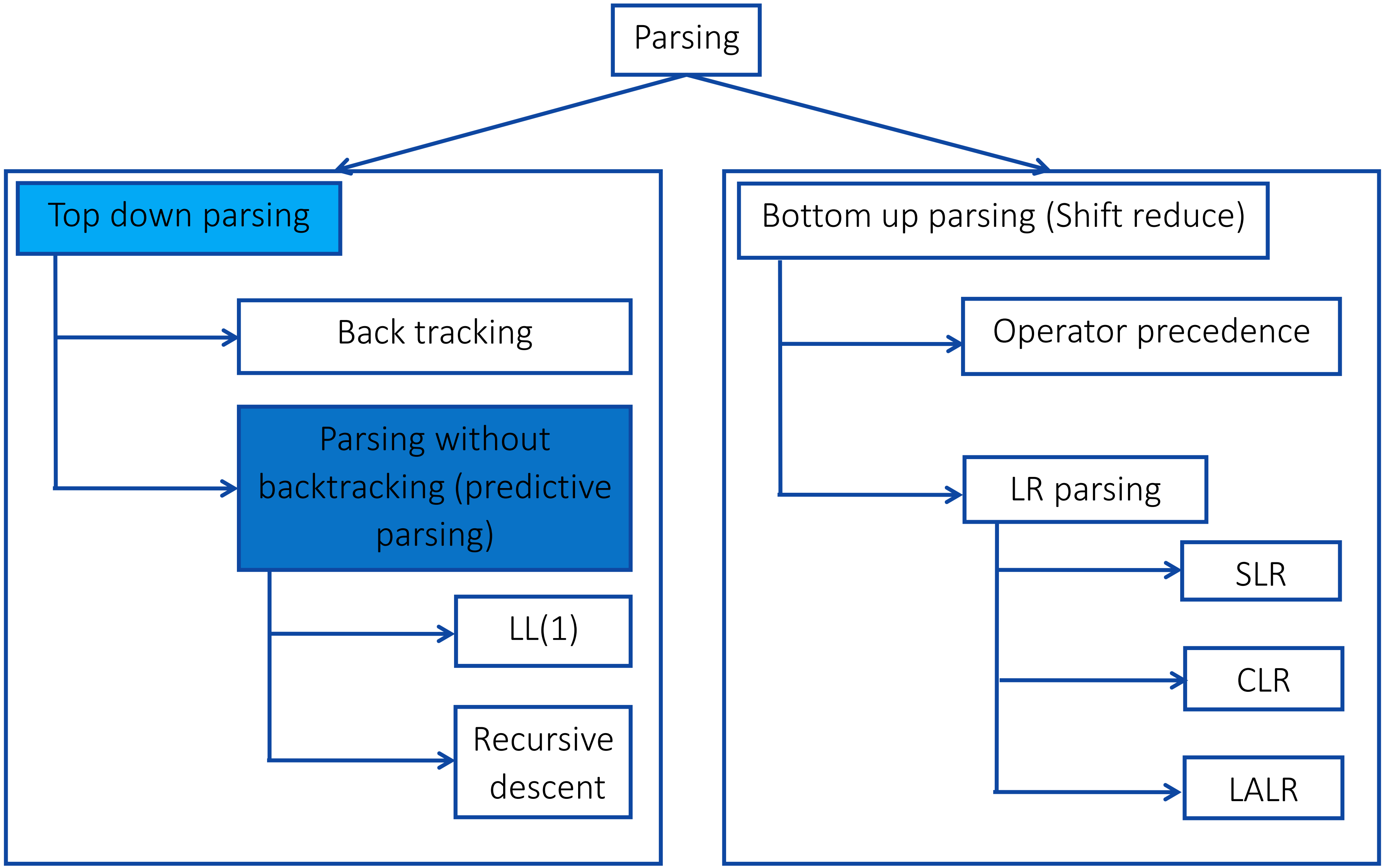
Q3: Find out whether the following grammar is LL(1):

Sol.

	FIRST	FOLLOW
$S \rightarrow A$	{a, b, c, d}	{\$}
$A \rightarrow Bb \mid Cd$	{a, b, c, d}	{\$}
$B \rightarrow aB \mid \varepsilon$	{a, ε}	{b}
$C \rightarrow cC \mid \varepsilon$	{c, ε}	{d}

	a	b	c	d	\$
S	$S \rightarrow A$	$S \rightarrow A$	$S \rightarrow A$	$S \rightarrow A$	
A	$A \rightarrow Bb$	$A \rightarrow Bb$	$A \rightarrow Cd$	$A \rightarrow Cd$	
B	$B \rightarrow aB$	$B \rightarrow \varepsilon$			
C			$C \rightarrow cC$	$C \rightarrow \varepsilon$	

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.

Consider the following grammar having rules,

$$E \rightarrow iE'$$
$$E' \rightarrow +iE' \mid \varepsilon$$

$$E \rightarrow iE'$$
$$E' \rightarrow +iE' \mid \varepsilon$$

Recursive Descent Parser:

```
E()
{
        if(look_ahead=='i')
        {
                match('i');
                E'();
        }
}
```

$$E \rightarrow iE'$$
$$E' \rightarrow +iE' \mid \varepsilon$$

```
E'()
{
        if(look_ahead=='+')
        {
                match('+');
                match('i');
                E'();
        }
        else
                return;
}
```

Recursive Descent Parser:

$$E \rightarrow iE'$$
$$E' \rightarrow +iE' \mid \varepsilon$$

```
E()
{
    if(look_ahead=='i')
    {
        match(‘i’);
        E’();
    }
}
```

```
E’()
{
    if(look_ahead=='+')
    {
        match(‘+’);
        match(‘i’);
        E’();
    }
    else
        return;
}
```

```
match(char c)
{
    if(look_ahead==c)
        look_ahead = getchar();
    else
        printf(“ERROR!”);
}
```

```
main()
{
    E();
    if(look_ahead=='$')
        printf(“Parsing Successful!”);
}
```

Input: i + i $