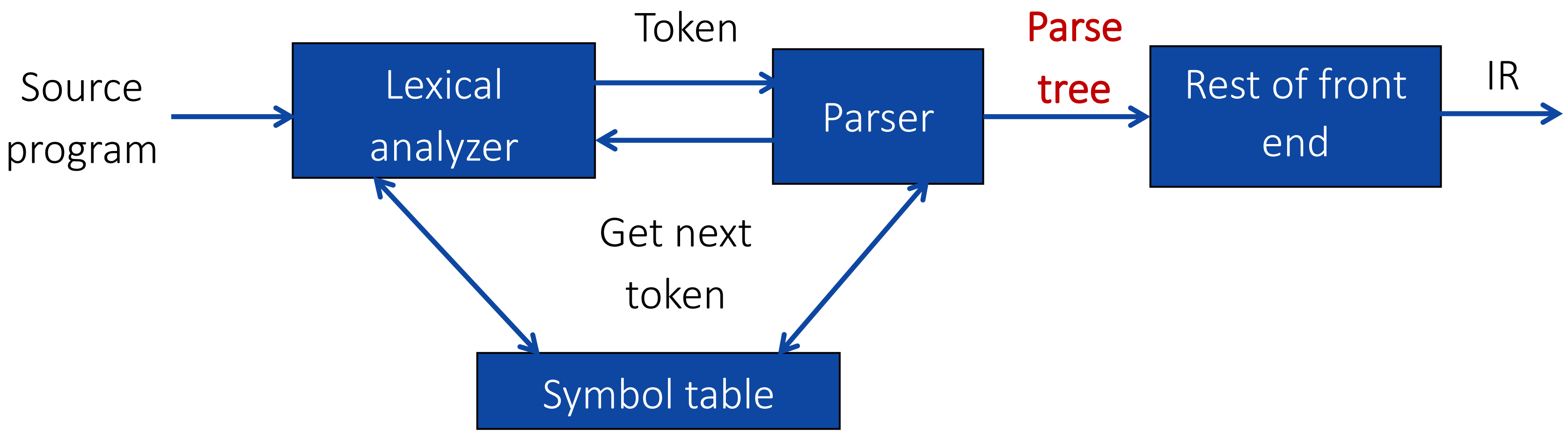
Module 2 – Syntax Analysis

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 $A \to \alpha$ 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 → E O E| (E) | -E | id

O → + | - | * | / | ↑

Terminals: id + - * / ↑ ( )

Non terminals: E, O

Start symbol: E

Productions: E → E O E| (E) | -E | id
O → + | - | * | / | ↑

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

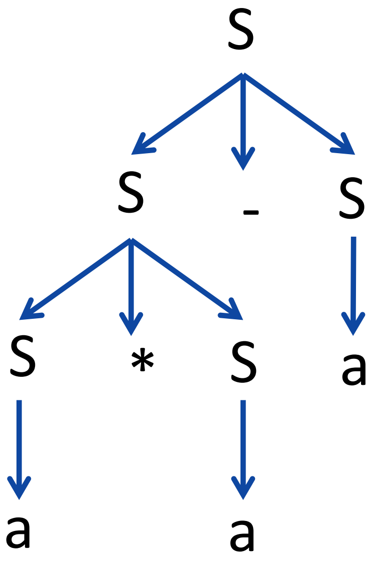
→**a***S-S

→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**

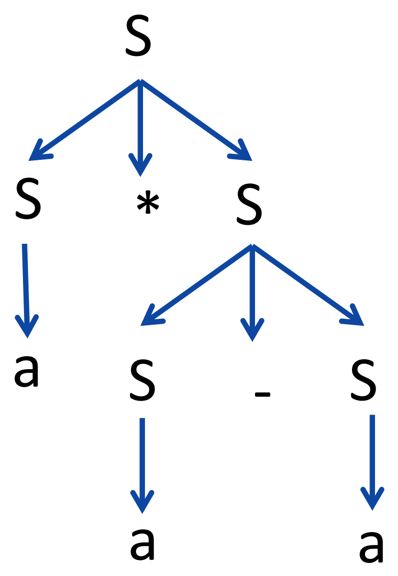
→S***S-S**

→S*S-**a**

→S***a**-a

→**a***a-a

Rightmost Derivation



Parse Tree

Exercise: 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

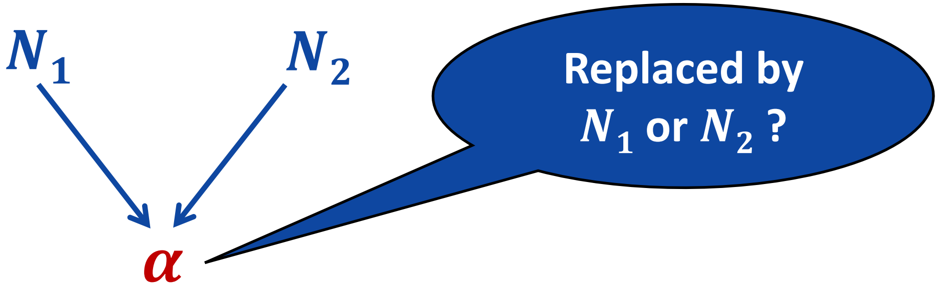
A small piece of silicon

Ambiguity

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

  $N1 \rightarrow \alpha$

  $N2 \rightarrow \alpha$

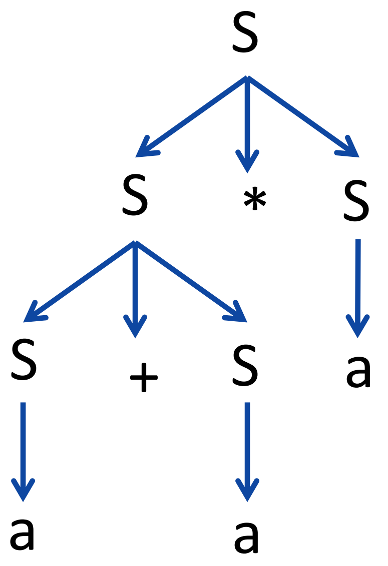


- α can be derived from either N1 or N2

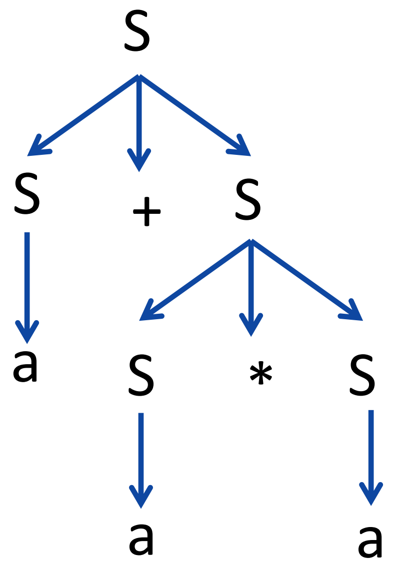
Ambiguous grammar

- 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 $\boldsymbol{A \rightarrow A\alpha}$ for some string α.

$$A \rightarrow A\alpha \mid \quad \longrightarrow \quad A \rightarrow \quad A'$$

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

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→aAB | aCD

S→aS'
S'→AB | CD

A→ xByA | xByAzA | a

A→ xByAA' | a
A'→ Є | zA

A→ aAB | aA |a

A→aA'
A'→AB | A | ϵ
A'→AA'' | ϵ
A''→B | ϵ

Exercise

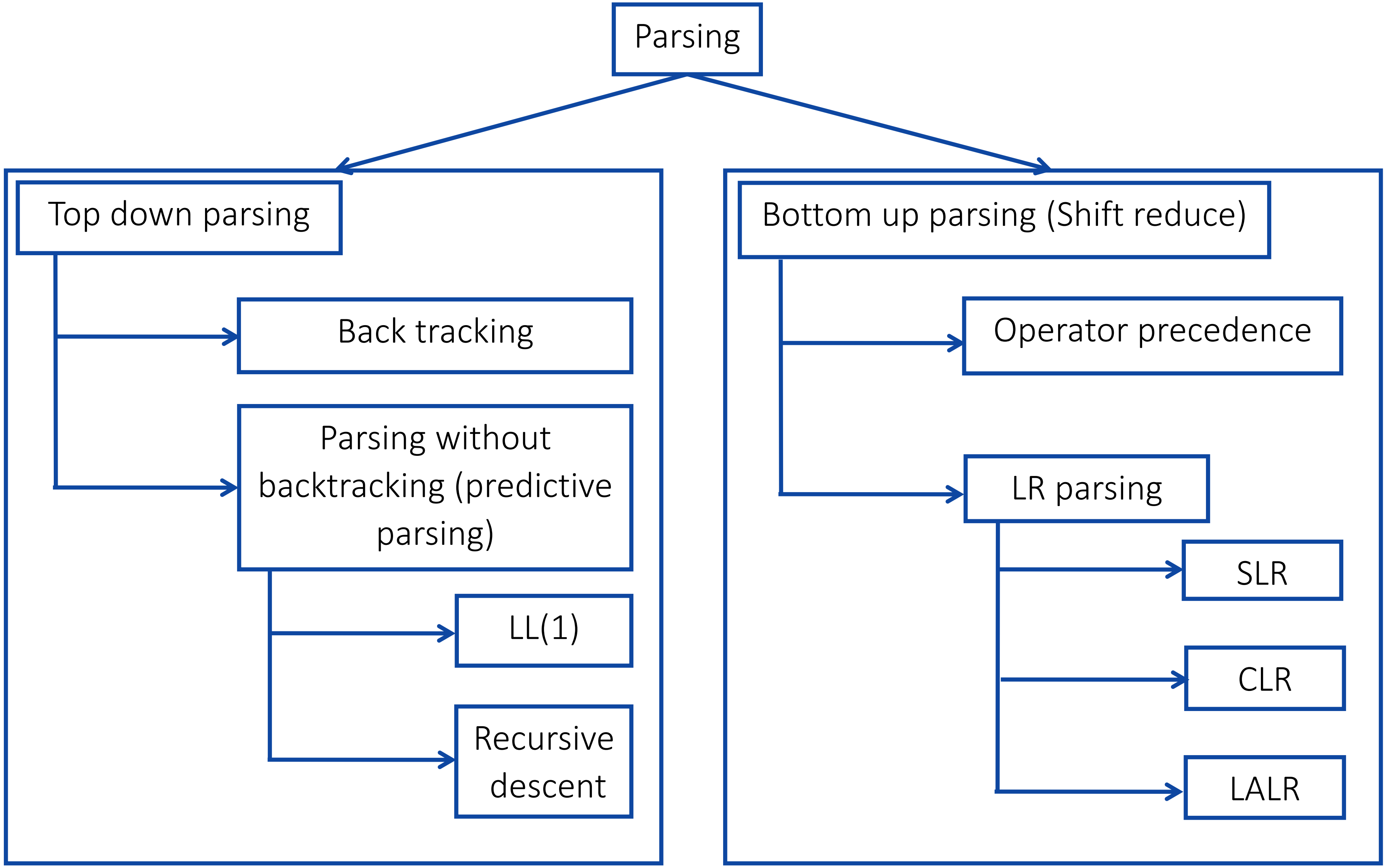
1. $S \rightarrow iEtS \mid iEtSeS \mid a$
2. $A \rightarrow ad \mid a \mid ab \mid abc \mid x$

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.

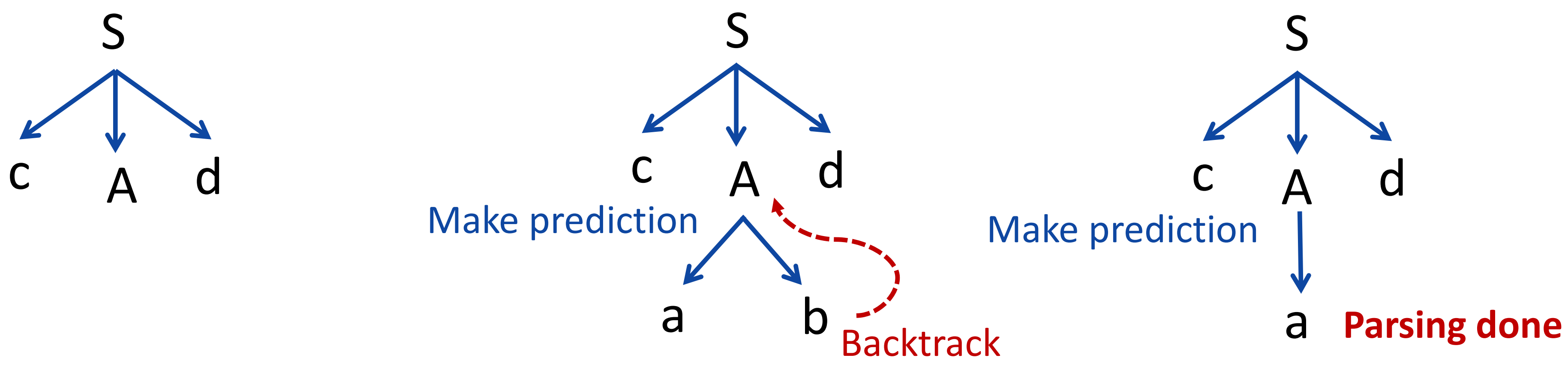
Classification of parsing methods



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

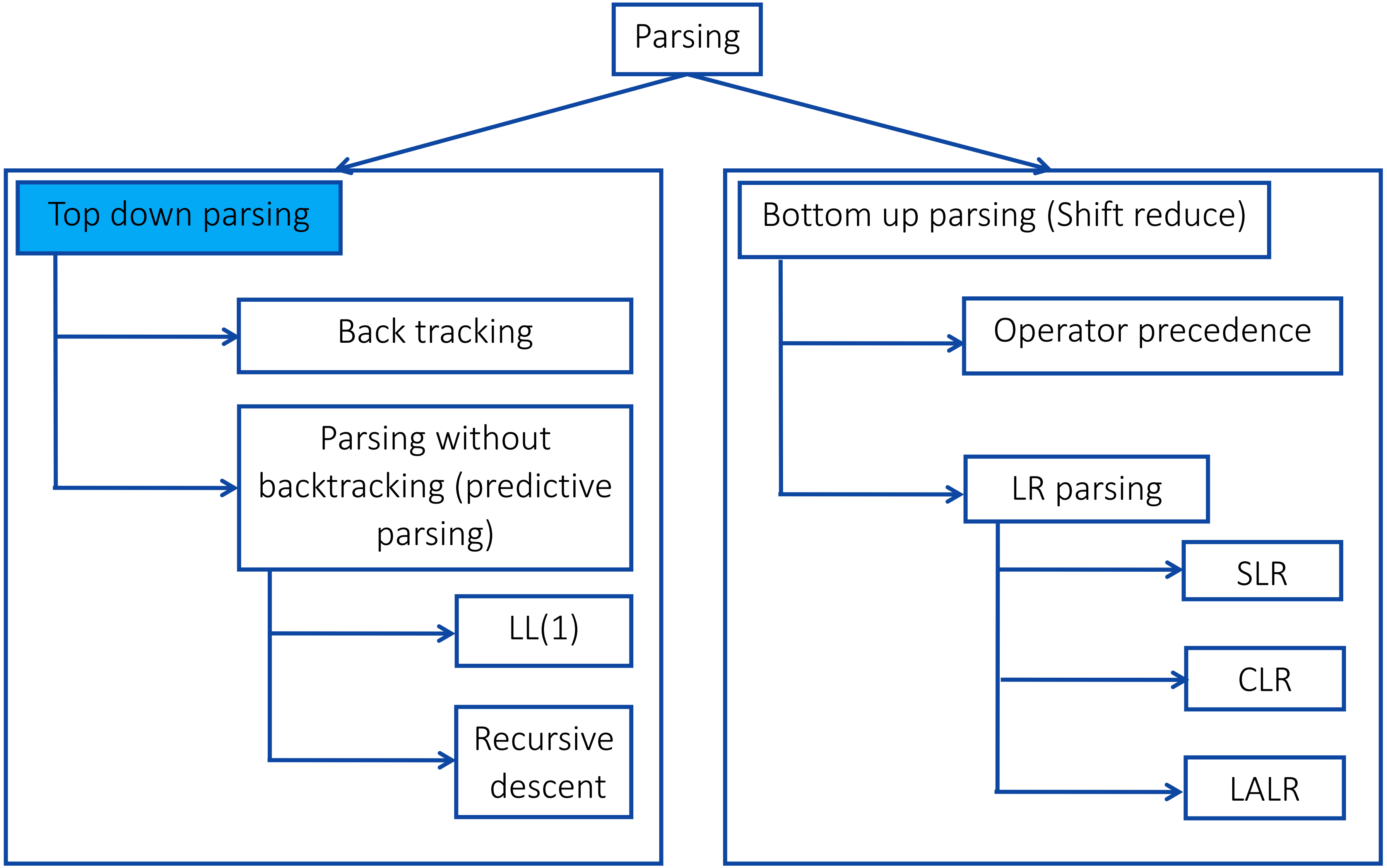
1. E→ 5+T | 3-T

   T→ V | V*V | V+V

   V→ a | b

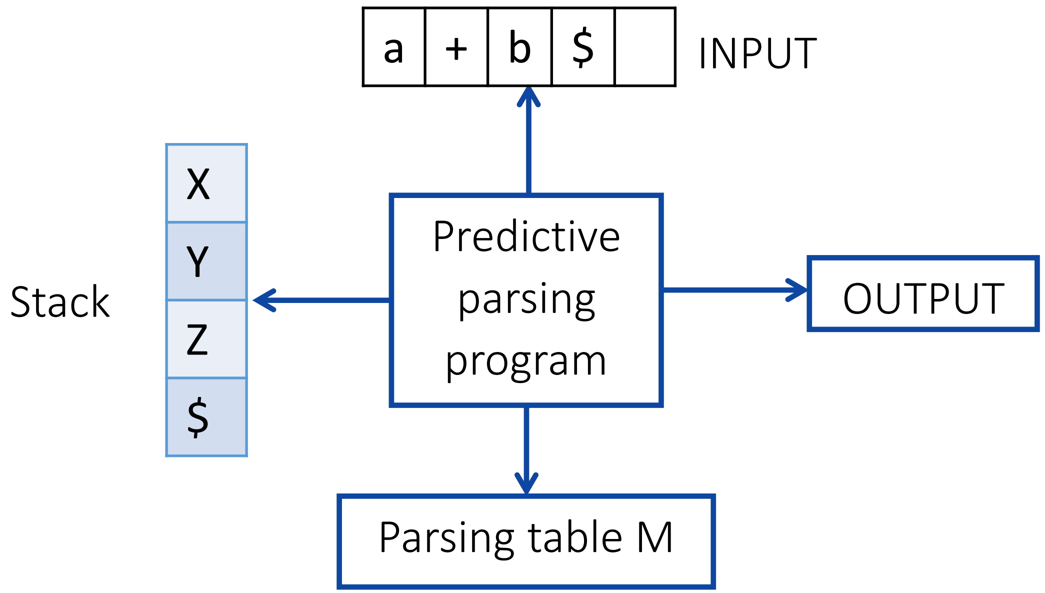
   String: 3-a+b

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 $A \rightarrow Y_1Y_2 \ldots\ldots\ldots Y_K$,

- 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 ∈ $then,$

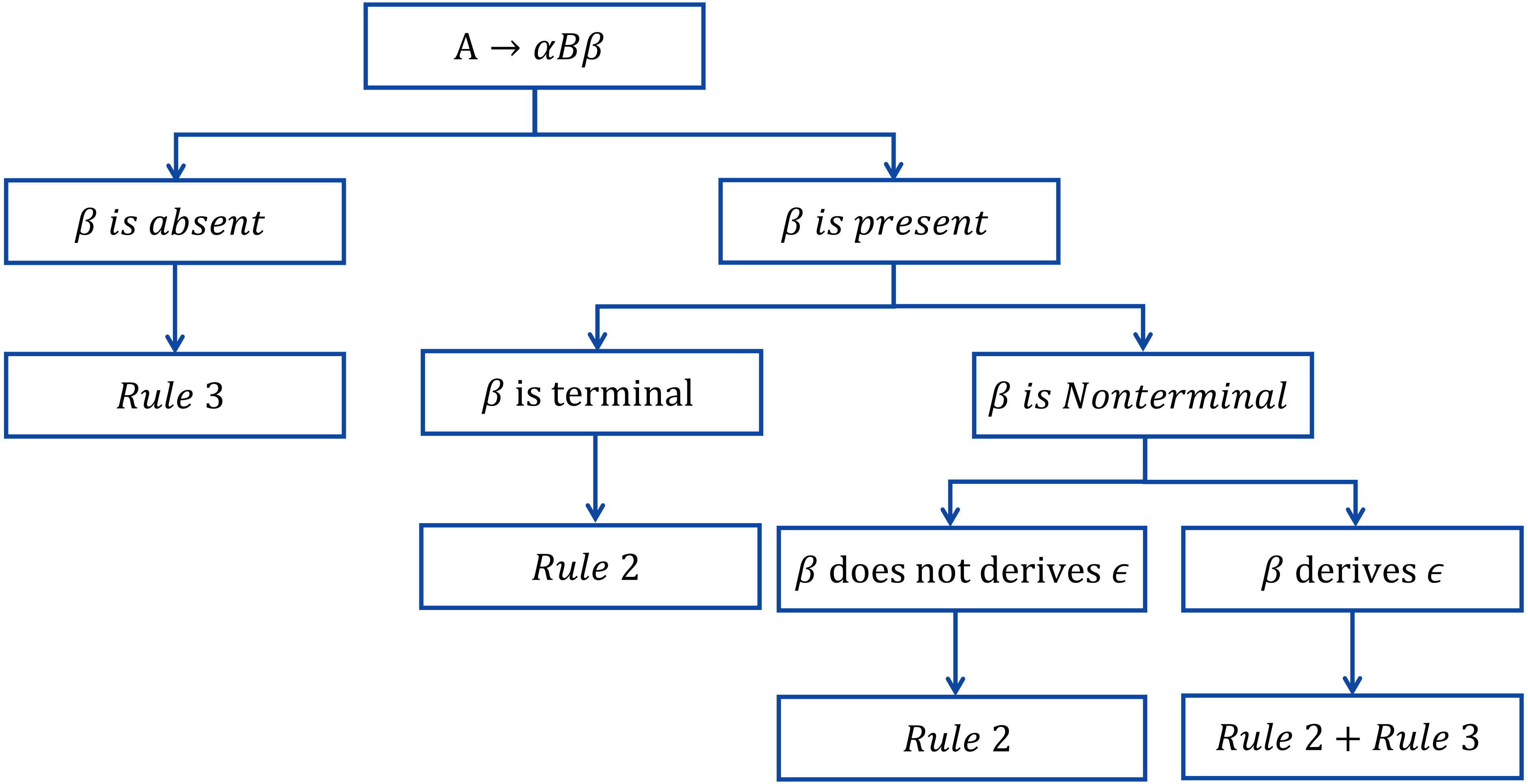
  $FIRST(A) = FIRST(Y_1) - \epsilon\ U\ FIRST(Y_2) - \epsilon\ U\ FIRST(Y_3) - \epsilon\ U\ FIRST(Y_4)$
- If Y_1 , Y_2, Y_3 …..Y_K all derives ∈ $then,$

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

Rules to compute FOLLOW of non terminal

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

$S \rightarrow aBa$

$B \rightarrow bB \mid \epsilon$

NT	First	Follow
S	{a}	
B	{b,ϵ}	

Step 2: Compute FOLLOW

Follow(S)

Rule 1: Place $ in FOLLOW(S)

Follow(S)={ $ }

Follow(B)

$S \rightarrow aBa$

S	**→**	**a**	**B**	**a**
A	**→**	**α**	**B**	**β**

Rule 2

First(β) $-$ ϵ

$B \rightarrow bB$

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]=$S \rightarrow aBa$

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

Example-1: LL(1) parsing

S→aBa
B→bB | ϵ

Step 3: Prepare predictive parsing table

NT	Input Symbol		
	a	b	$
S	S→aBa		
B			

NT	First	Follow
S	{a}	{$}
B	{b,ϵ}	{a}

B→bB

a=FIRST(bB)={ b }

M[B,b]=B→bB

Rule: 2
A→ α
a = first(α)
M[A,a] = A→ α

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	$S \rightarrow aBa$		
B		$B \rightarrow bB$	

$B \rightarrow \epsilon$

b=FOLLOW(B)={ a }

M[B,a]=$B \rightarrow \epsilon$

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

Example-2: LL(1) parsing

S→aB | ϵ
B→bC | ϵ
C→cS | ϵ

Step 1: Not required

Step 2: Compute FIRST

First(S)

NT	First
S	
B	
C	

S→aB

S	→	a	B
A	→	α	

Rule 1
add α to $FIRST(A)$

S→ϵ

S	→	ϵ
A	→	ϵ

Rule 2
add ϵ to $FIRST(A)$

FIRST(S)={ a , ϵ }

Example-2: LL(1) parsing

S→aB | ϵ
B→bC | ϵ
C→cS | ϵ

Step 1: Not required

Step 2: Compute FIRST

First(B)

NT	First
S	{ a, ϵ }
B	
C	

B→bC

B	→	b	C
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-2: LL(1) parsing

S→aB | ϵ
B→bC | ϵ
C→cS | ϵ

Step 1: Not required

Step 2: Compute FIRST

First(C)

NT	First
S	{ a, ϵ }
B	{b,ϵ}
C	

C→cS

C	**→**	**c**	**S**
A	**→**	α	

Rule 1
add α to $FIRST(A)$

FIRST(B)={ c , ϵ }

C→ϵ

C	**→**	ϵ
A	**→**	ϵ

Rule 2
add ϵ to $FIRST(A)$

Example-2: LL(1) parsing

Step 2: Compute FOLLOW

Follow(S)

Rule 1: Place $ in FOLLOW(S)

Follow(S)={ $ }

S→aB | ϵ
B→bC | ϵ
C→cS | ϵ

C→cS

C	→	c	S
A	→	α	B

Rule 3
Follow(A)=follow(B)
Follow(S)=Follow(C)={$}

NT	First	Follow
S	{a,ϵ}	
B	{b,ϵ}	
C	{c,ϵ}	

B→bC

B	→	b	C
A	→	α	B

Rule 3
Follow(A)=follow(B)
Follow(C)=Follow(B) ={$}

S→aB

S	→	a	B
A	→	α	B

Rule 3
Follow(A)=follow(B)
Follow(B)=Follow(S)={$}

Example-2: LL(1) parsing

S→aB | ϵ
B→bC | ϵ
C→cS | ϵ

NT	First	Follow
S	{a,ϵ}	{\$}
B	{b,ϵ}	{\$}
C	{c,ϵ}	{\$}

Step 3: Prepare predictive parsing table

NT	Input Symbol			
	a	b	c	\$
S				
B				
C				

Rule: 2
A→ α
a = first(α)
M[A,a] = A→ α

S→aB
a=FIRST(aB)={ a }
M[S,a]=S→aB

Example-2: LL(1) parsing

S→aB | ϵ
B→bC | ϵ
C→cS | ϵ

NT	First	Follow
S	{a}	{$}
B	{b,ϵ}	{$}
C	{c,ϵ}	{$}

Step 3: Prepare predictive parsing table

NT	Input Symbol			
	a	b	c	$
S	S→aB			
B				
C				

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

S→ϵ

b=FOLLOW(S)={ $ }

M[S,$]=S→$\epsilon$

Example-2: LL(1) parsing

S→aB | ϵ
B→bC | ϵ
C→cS | ϵ

NT	First	Follow
S	{a}	{$}
B	{b,ϵ}	{$}
C	{c,ϵ}	{$}

Step 3: Prepare predictive parsing table

NT	Input Symbol			
	a	b	c	$
S	S→aB			S→ϵ
B				
C				

Rule: 2
A→ α
a = first(α)
M[A,a] = A→ α

B→bC

a=FIRST(bC)={ b }

M[B,b]=B→bC

Example-2: LL(1) parsing

S→aB | ϵ
B→bC | ϵ
C→cS | ϵ

NT	**First**	**Follow**
S	{a}	{$}
B	{b,ϵ}	{$}
C	{c,ϵ}	{$}

Step 3: Prepare predictive parsing table

NT	**Input Symbol**			
	a	**b**	**c**	**$**
S	S→aB			S→ϵ
B		B→bC		
C				

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

B→ϵ

b=FOLLOW(B)={ $ }

M[B,$]=B→$\epsilon$

Example-2: LL(1) parsing

S→aB | ϵ
B→bC | ϵ
C→cS | ϵ

NT	First	Follow
S	{a}	{$}
B	{b,ϵ}	{$}
C	{c,ϵ}	{$}

Step 3: Prepare predictive parsing table

NT	Input Symbol			
	a	**b**	**c**	**$**
S	S→aB			S→ϵ
B		B→bC		B→ϵ
C				

Rule: 2
A→ α
a = first(α)
M[A,a] = A→ α

C→cS
a=FIRST(cS)={ c }
M[C,c]=C→cS

Example-2: LL(1) parsing

S→aB | ϵ
B→bC | ϵ
C→cS | ϵ

NT	First	Follow
S	{a}	{$}
B	{b,ϵ}	{$}
C	{c,ϵ}	{$}

Step 3: Prepare predictive parsing table

NT	Input Symbol			
	a	b	c	$
S	S→aB			S→ϵ
B		B→bB		B→ϵ
C			C→cS	

C→ϵ

b=FOLLOW(C)={ $ }

M[C,$]=C→$\epsilon$

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

Example-3: 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-3: 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
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-3: LL(1) parsing

Step 2: Compute FIRST

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 $FIRST(A)$

FIRST(E’)={ + , ϵ }

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

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

Example-3: LL(1) parsing

Step 2: Compute FIRST

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 $FIRST(A)$

FIRST(T')={ * , ϵ }

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

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

Example-3: LL(1) parsing

Step 2: Compute FOLLOW

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

FOLLOW(E)

Rule 1: Place $ in FOLLOW(E)

F→(E)

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

Rule 2

FOLLOW(E)={ $,) }

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

Example-3: 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-3: LL(1) parsing

Step 2: Compute FOLLOW

FOLLOW(T)

E→TE’

E	→		T	E’
A	→	α	B	β

Rule 2

E	→		T	E’
A	→	α	B	β

Rule 3

FOLLOW(T)={ +, $,)

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-3: 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-3: LL(1) parsing

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

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

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-3: 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-3: LL(1) parsing

Step 2: Compute FOLLOW

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

FOLLOW(F)

T’→*FT’

T’	→	*	F	T’
A	→	α	B	β

Rule 2

T’	→	*	F	T’
A	→	α	B	β

Rule 3

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

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

Example-3: LL(1) parsing

Step 3: Construct predictive parsing table

$E \rightarrow TE'$
$E' \rightarrow +TE' \mid \epsilon$
$T \rightarrow FT'$
$T' \rightarrow *FT' \mid \epsilon$
$F \rightarrow (E) \mid id$

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

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

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

$E \rightarrow TE'$

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

M[E,(]=E→TE'

M[E,id]=E→TE'

Example-3: LL(1) parsing

Step 3: Construct predictive parsing table

NT	Input Symbol					
	id	+	*	(	)	$
E	E→TE'			E→TE'		
E'						
T						
T'						
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 }	{ +,$,) }
F	{ (,id }	{*,+,$,)}

E'→+TE'

a=FIRST(+TE')={ + }

M[E',+]=E'→+TE'

Rule: 2
A→ α
a = first(α)
M[A,a] = A→ α

Example-3: LL(1) parsing

Step 3: Construct predictive parsing table

NT	Input Symbol					
	id	+	*	(	)	$
E	$E \rightarrow TE'$			$E \rightarrow TE'$		
E'		$E' \rightarrow +TE'$				
T						
T'						
F						

$E \rightarrow TE'$
$E' \rightarrow +TE' \mid \epsilon$
$T \rightarrow FT'$
$T' \rightarrow *FT' \mid \epsilon$
$F \rightarrow (E) \mid id$

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

$E' \rightarrow \epsilon$

b=FOLLOW(E')={ $,) }

$M[E',\$]=E' \rightarrow \epsilon$

$M[E',)]=E' \rightarrow \epsilon$

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

Example-3: 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						

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

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

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

Example-3: 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						

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

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

T'→*FT'

a=FIRST(*FT')={ * }

M[T',*]=T'→*FT'

Rule: 2
A→ α
a = first(α)
M[A,a] = A→ α

Example-3: 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						

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

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

T'→ϵ

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

M[T',+]=T'→ϵ

M[T',$]=T'→$\epsilon$

M[T',)]=T'→ϵ

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

Example-3: 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						

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

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

Rule: 2
A→ α
a = first(α)
M[A,a] = A→ α

F→(E)

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

M[F,(]=F→(E)

Example-3: 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)		

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

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

Rule: 2
A→ α
a = first(α)
M[A,a] = A→ α

F→id

a=FIRST(id)={ id }

M[F,id]=F→id

Example-3: 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

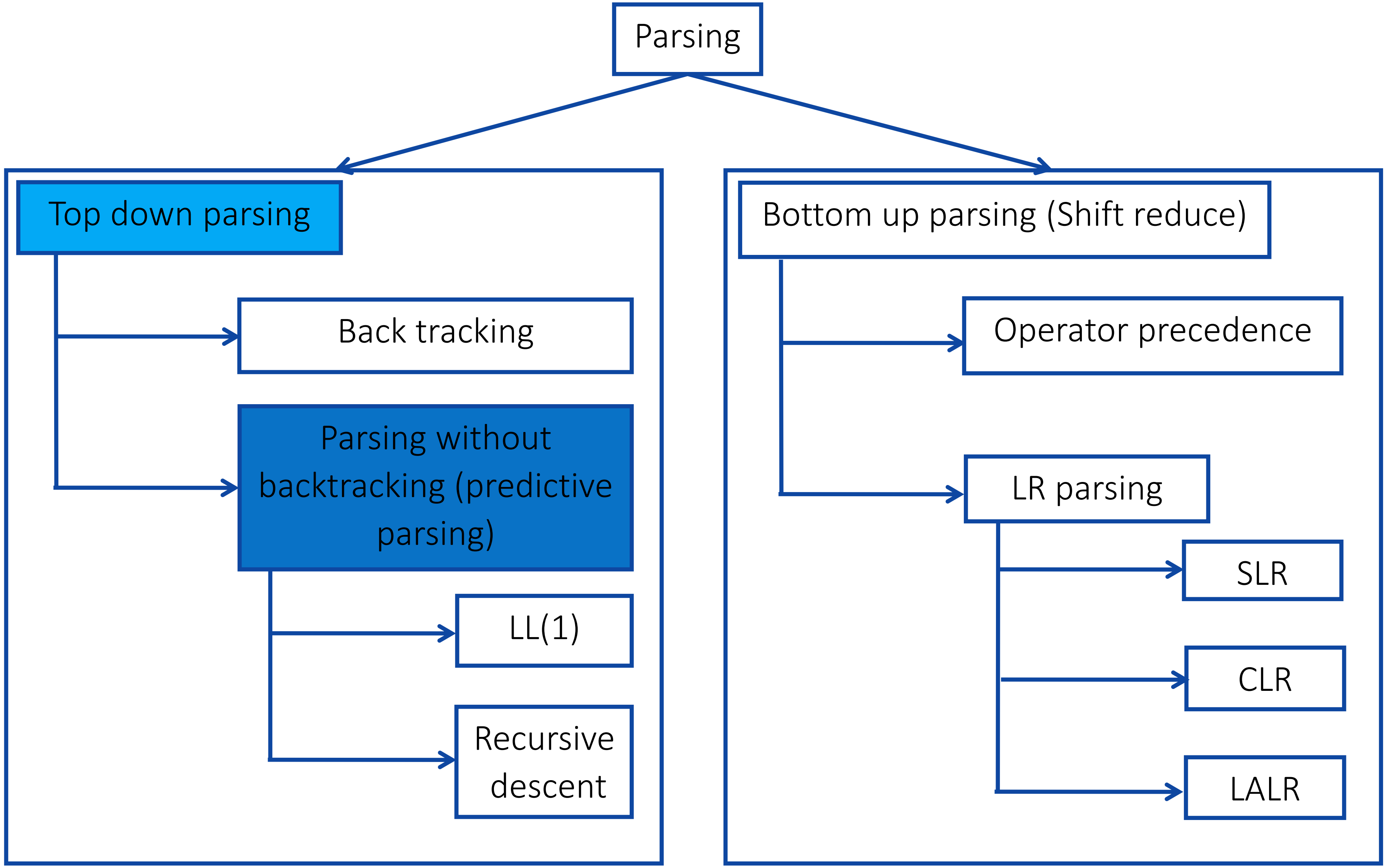
Example-3: LL(1) parsing

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

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.

Cont.,

```
void A() {
        Choose an A-production, A → X1X2 ··· Xk;
        for ( i = 1 to k ) {
                if ( Xi is a nonterminal )
                        call procedure Xi();
                else if ( Xi equals the current input symbol a )
                        advance the input to the next symbol;
                else /* an error has occurred */;
        }
}
```

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 | ϵ

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