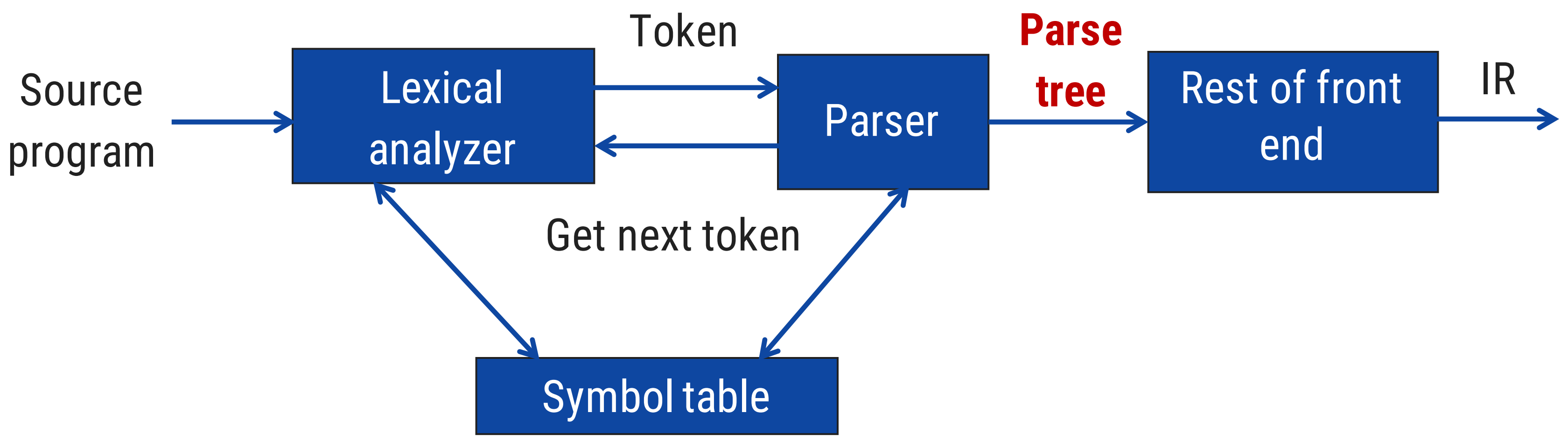
Unit – 3
Syntax Analysis (I)

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
- Parser generator

Role of Parser

Role of parser



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

Context free grammar

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: Context Free Grammar

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

E → E O E | (E) | id

O → + | - | * | / | ↑

Non terminals: **E, O**

Terminals: **id + - * / ↑ ( )**

Start symbol: **E**

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

Derivation

Derivation

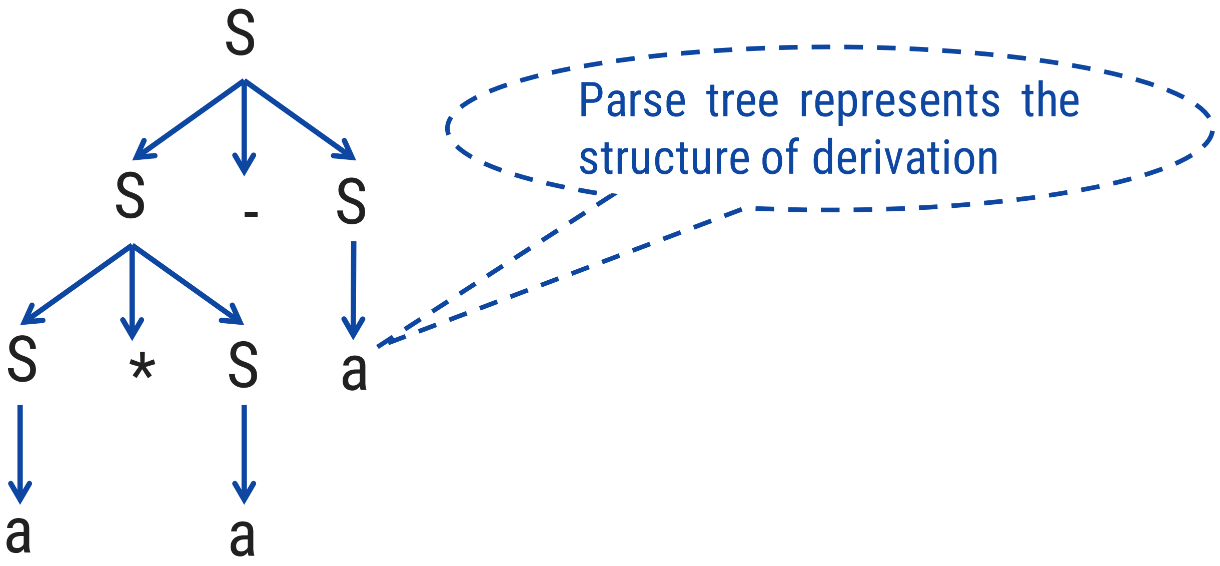
- A derivation is basically a sequence of production rules, in order to get the input string.
- To decide which non-terminal to be replaced with production rule, we can have two options:
  1. Leftmost derivation
  2. Rightmost derivation

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**

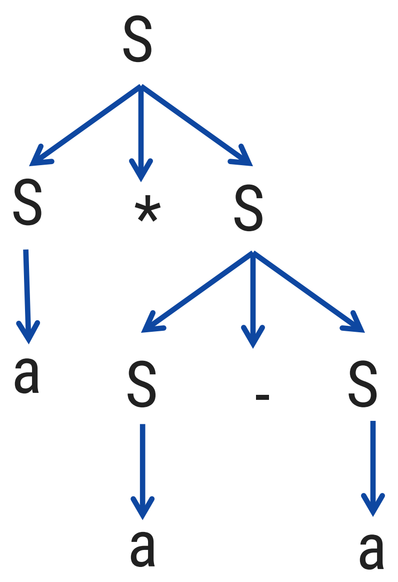
→S***S-S**

→S*S-**a**

→S***a**-a

→**a***a-a

Rightmost Derivation



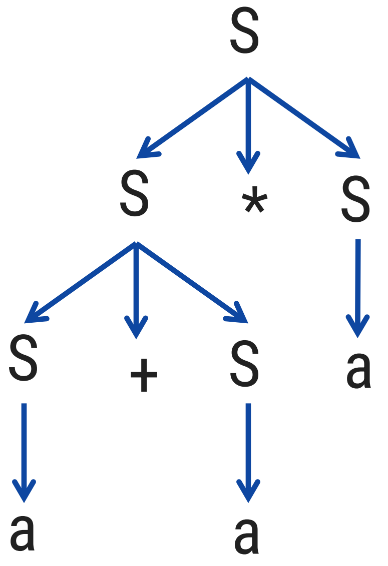
Parse Tree

Ambiguous grammar

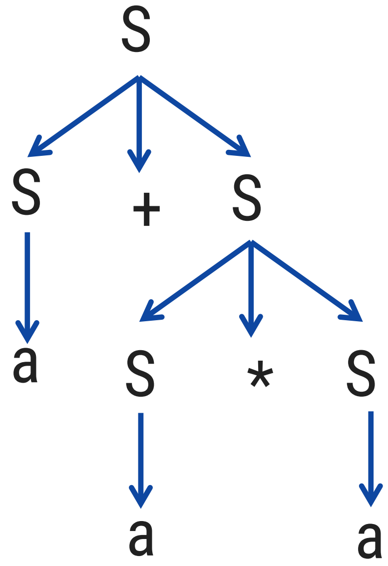
Ambiguous grammar

- 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.

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

Top down parsing: In top down parsing parser build parse tree from top to bottom.

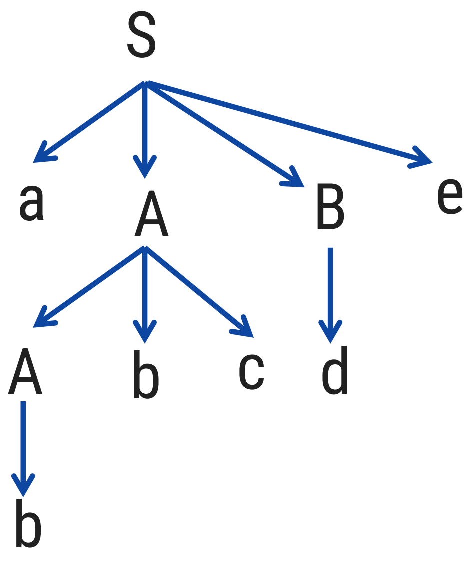
Grammar:

S→aABe

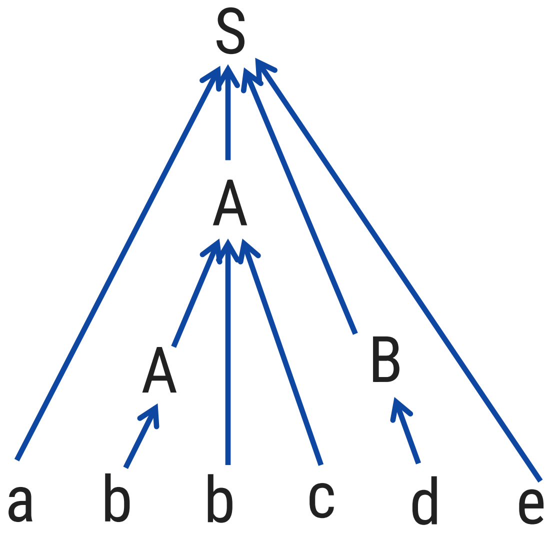
A→Abc | b

B→d

String: abbcde

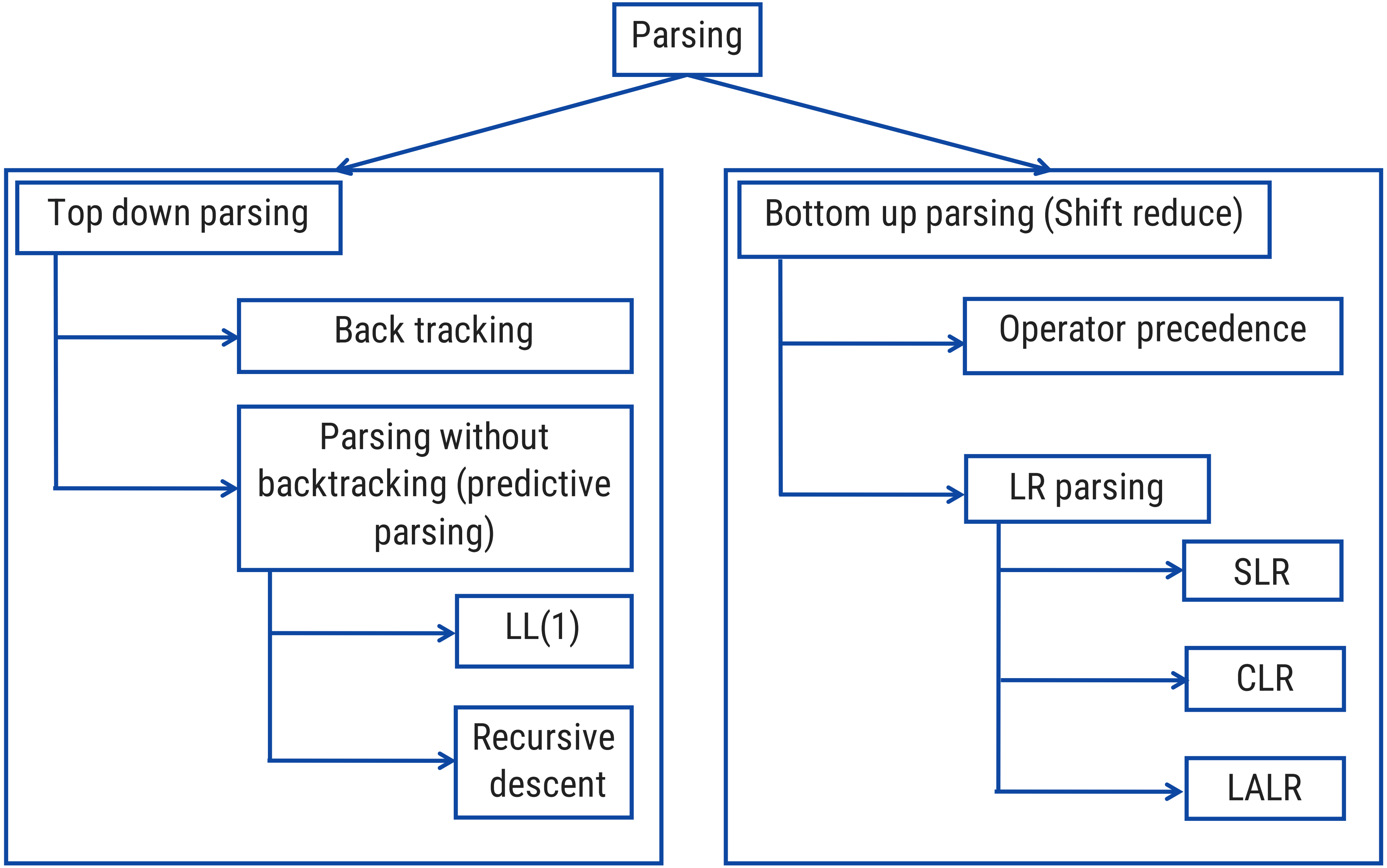


Bottom up parsing: Bottom up parser starts from leaves and work up to the root.

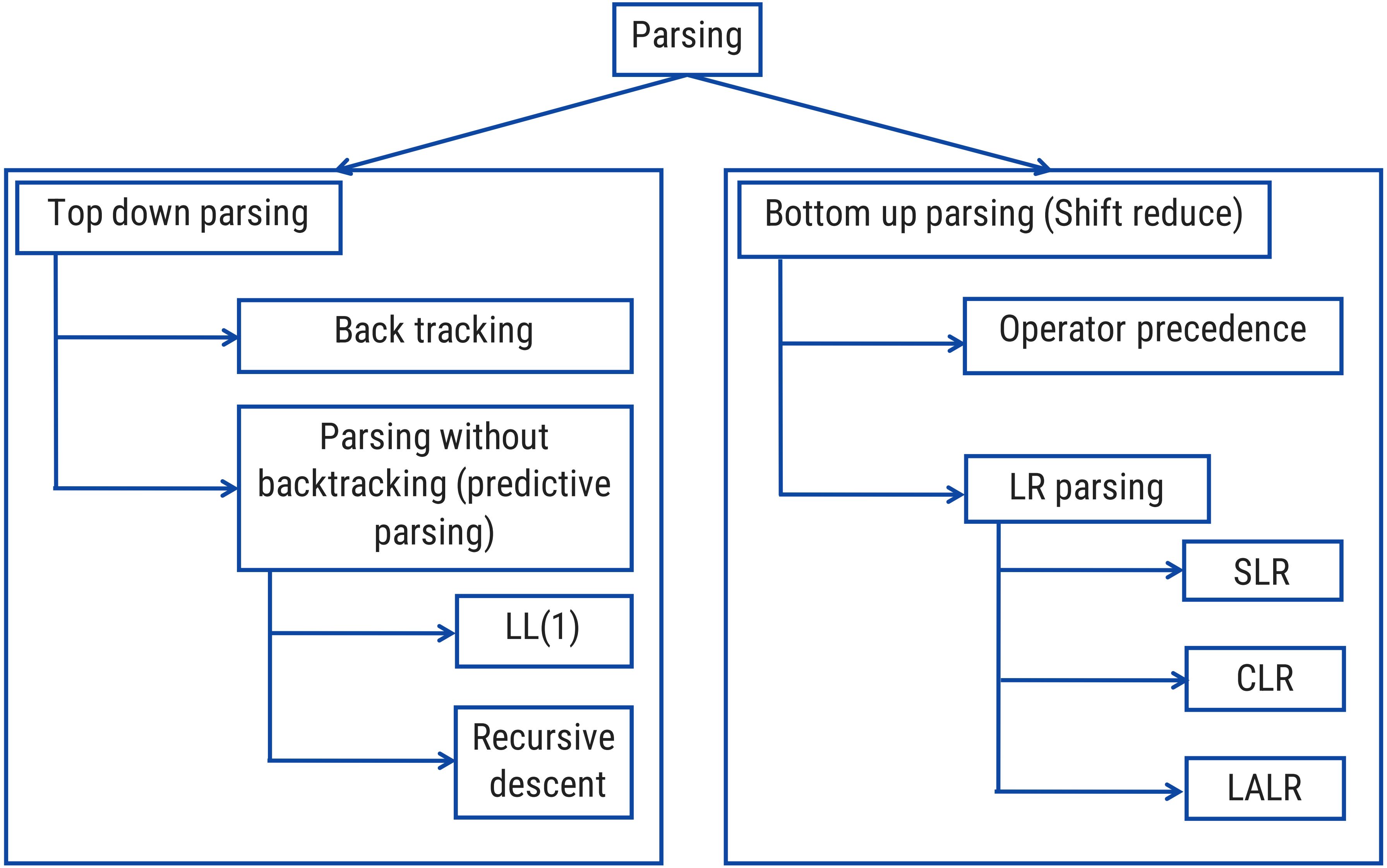


Classification of Parsing

Classification of parsing



Classification of parsing

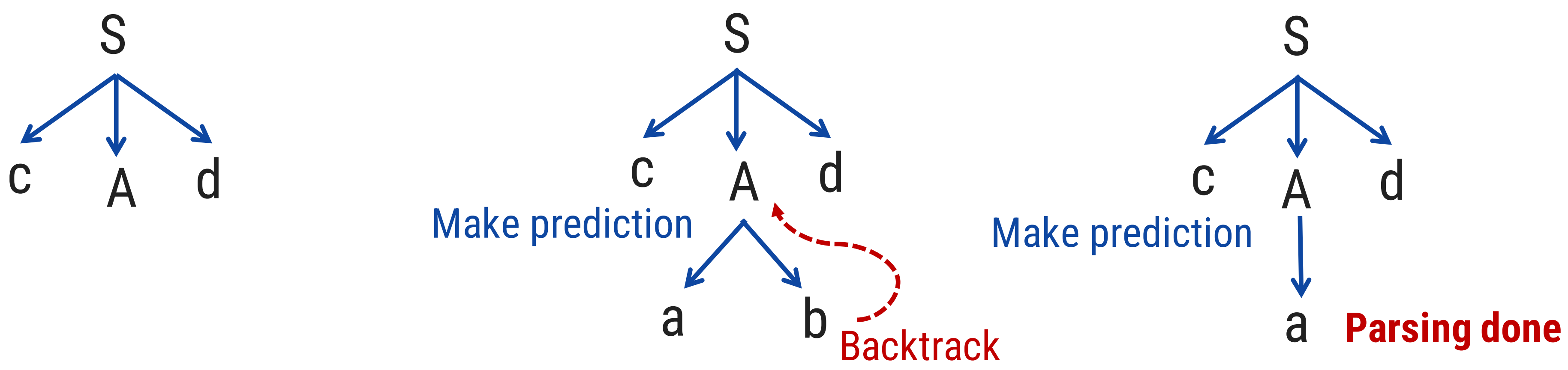


Backtracking

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

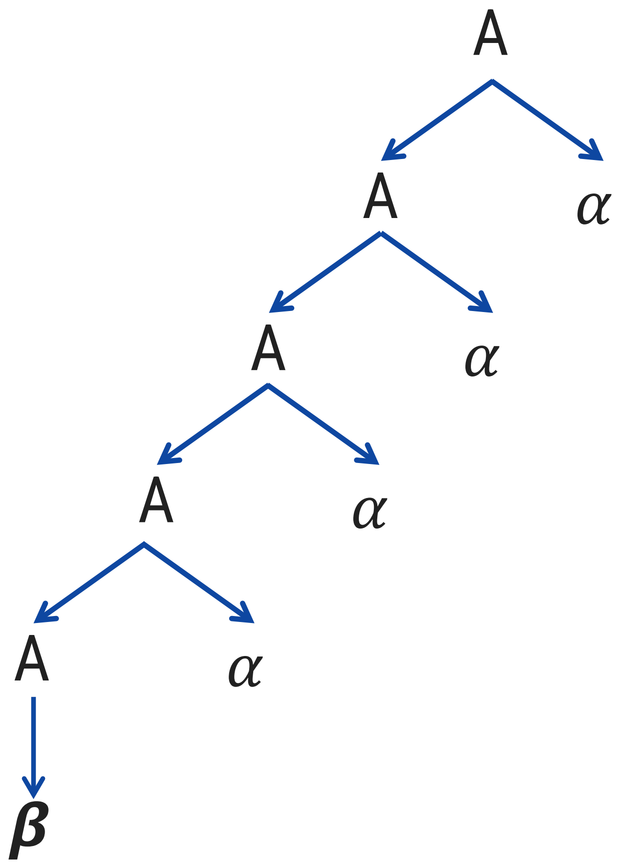


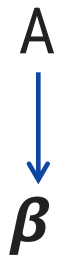
Left Recursion

Problems in Top-down Parsing

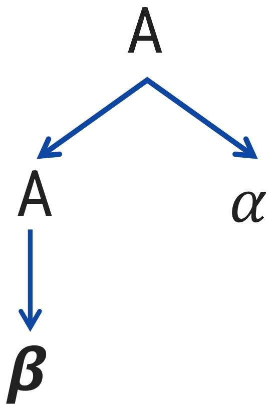
Left recursion

- 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.
- Grammar: $A \rightarrow A\boldsymbol{\alpha} \mid \boldsymbol{\beta}$





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



Left recursion elimination

$\beta\alpha^*$

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

$\Longrightarrow$

$A \rightarrow \quad A'$

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

Examples: Left recursion elimination

$E \rightarrow E+T \mid T$

$E \rightarrow TE'$

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

$T \rightarrow T*F \mid F$

$T \rightarrow FT'$

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

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

$X \rightarrow ZX'$

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

Left Factoring

Problems in Top-down Parsing

Left factoring

$$A \rightarrow \alpha\boldsymbol{\beta}1 \mid \alpha\boldsymbol{\beta}2 \mid \alpha\boldsymbol{\beta}3$$

- Left factoring is a grammar transformation that is useful for producing a grammar suitable for predictive parsing.
- It is used to remove nondeterminism from the grammar.

Left factoring

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

Example: Left factoring

S→aAB | aCD

S→aS'

S'→AB | CD

A→ xByA | xByAzA | a

A→ xByAA' | a

A'→ Є | zA

First & Follow

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

- 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 $\mathrm{A} \rightarrow \alpha B\beta$, 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 $\mathrm{F}OLLOW(B) = FOLLOW(A)$

Example-1: First & Follow

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-1: First & Follow

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-1: First & Follow

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-1: First & Follow

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-1: First & Follow

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-1: First & Follow

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-1: First & Follow

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-1: First & Follow

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-1: First & Follow

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-1: First & Follow

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-2: First & Follow

S→ABCDE

A→ a | ϵ

B→ b | ϵ

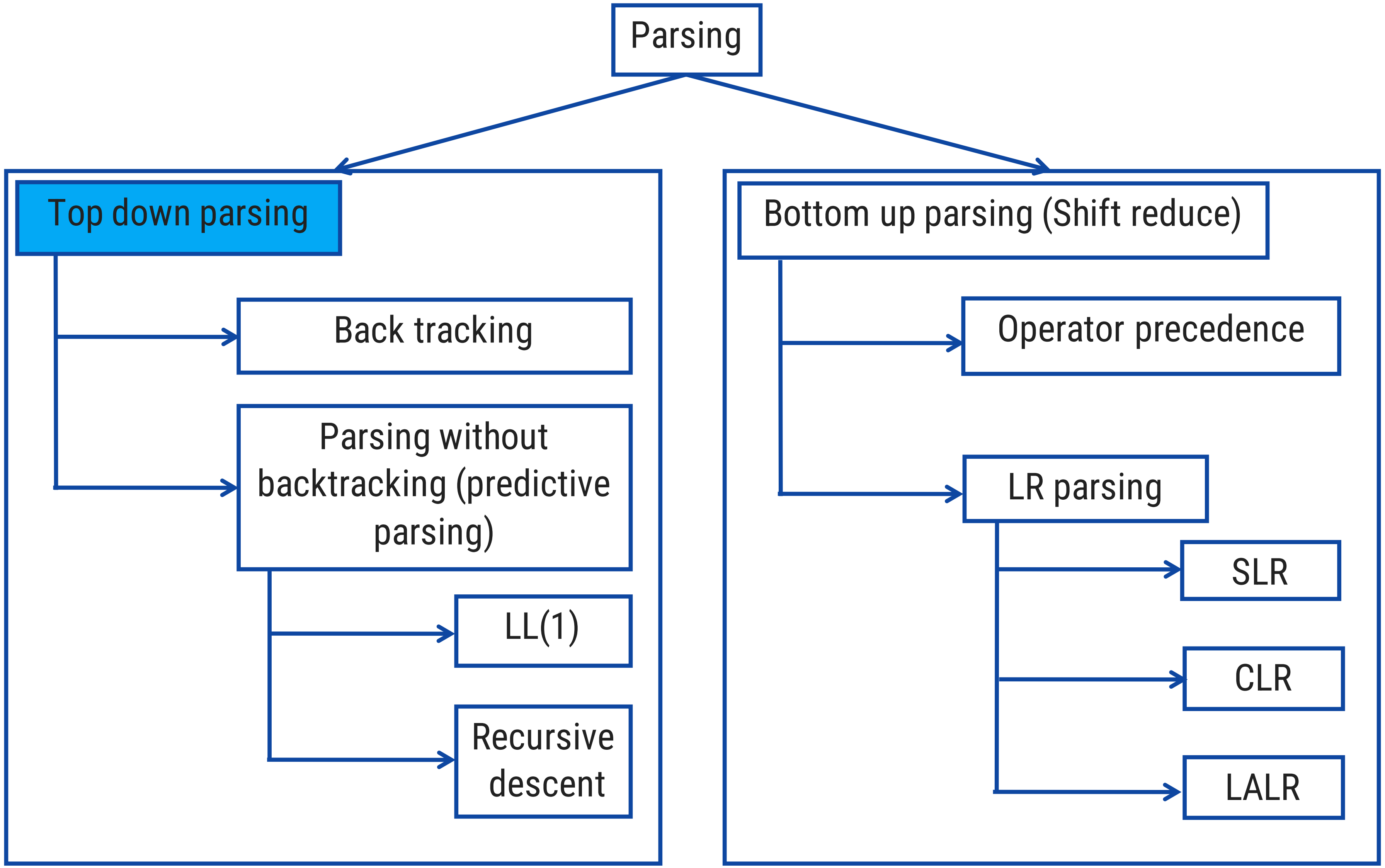
C→ c

D→ d | ϵ

E→ e | ϵ

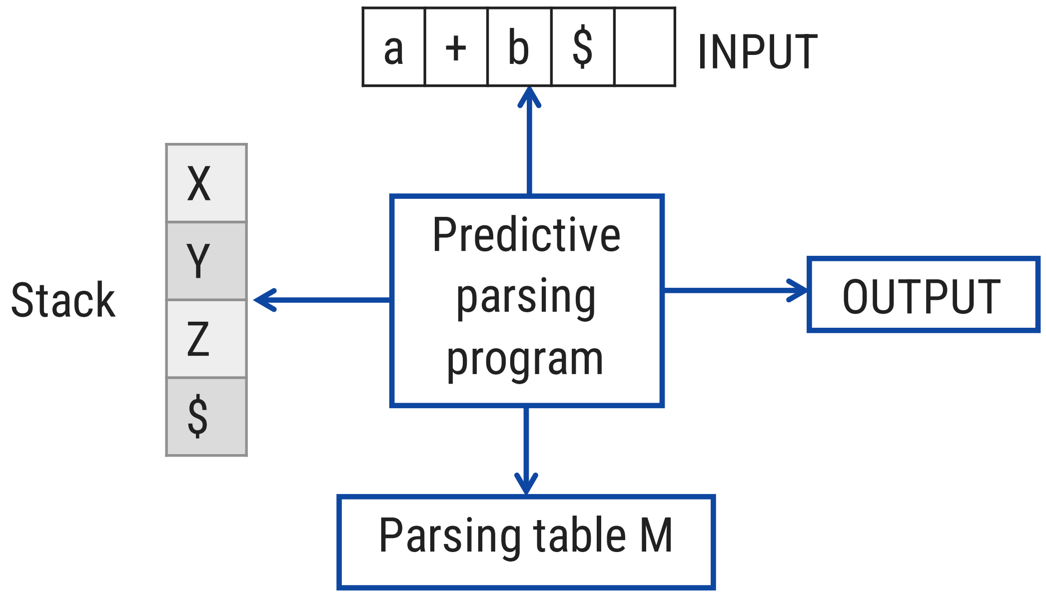
NT	First	Follow
S		
A		
B		
C		
D		
E		

Parsing Methods



LL(1) parser (Predictive parser or Non recursive descent 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.



Model of LL(1) Parser

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 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 $FOLLOW(A)$. 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

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

NT	First
S	
B	

Step 1: Not required

Step 2: Compute FIRST

First(S)

$S \rightarrow aBa$

S	$\rightarrow$	a	B	a
A	$\rightarrow$	α		

Rule 1
add α to $FIRST(A)$

FIRST(S)={ a }

First(B)

$B \rightarrow bB$

B	$\rightarrow$	b	B
A	$\rightarrow$	α	

Rule 1
add α to $FIRST(A)$

$B \rightarrow \epsilon$

B	$\rightarrow$	ϵ
A	$\rightarrow$	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) − ϵ

B→bB

B	→	b	B
A	→	α	B

Rule 3
Follow(A)=follow(B)

Follow(B)={ a }

Example-1: LL(1) parsing

S→aBa
B→bB | ϵ

Step 3: Prepare predictive parsing table

NT	First	Follow
S	{a}	{$}
B	{b,ϵ}	{a}

NT	Input Symbol		
	a	b	$
S			
B			

S→aBa

a=FIRST(aBa)={ a }

M[S,a]=S→aBa

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

Example-1: LL(1) parsing

S→aBa
B→bB | ϵ

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

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

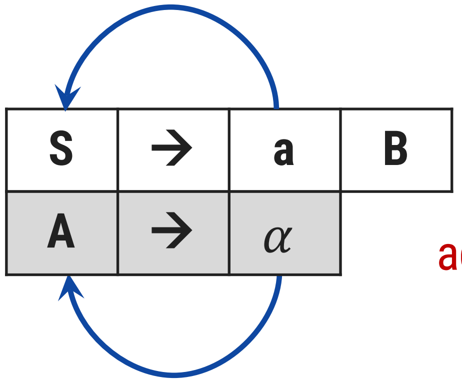
S→aB | ϵ
B→bC | ϵ
C→cS | ϵ

Step 1: Not required

Step 2: Compute FIRST

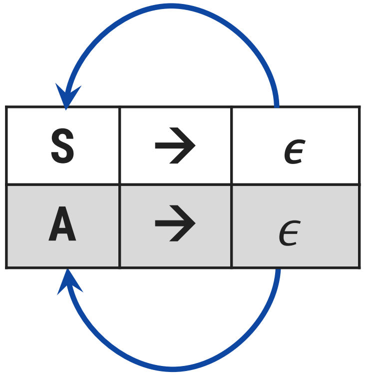
First(S)

S→aB



Rule 1
add α to $FIRST(A)$

S→ϵ



Rule 2
add ϵ to $FIRST(A)$

FIRST(S)={ a , ϵ }

NT	First
S	
B	
C	

Example-2: LL(1) parsing

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

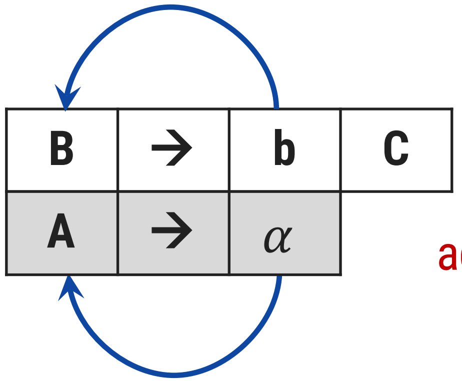
Step 1: Not required

Step 2: Compute FIRST

First(B)

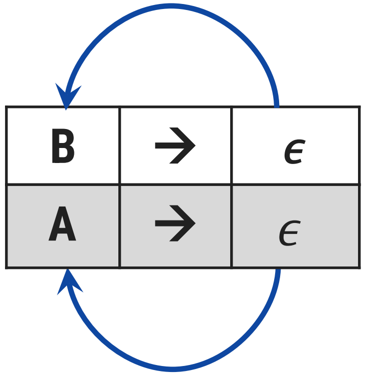
NT	First
S	$\{ a, \epsilon \}$
B	
C	

$B \rightarrow bC$



Rule 1
add α to $FIRST(A)$

$B \rightarrow \epsilon$



Rule 2
add ϵ to $FIRST(A)$

$FIRST(B)=\{ b , \epsilon \}$

Example-2: LL(1) parsing

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

Step 1: Not required

Step 2: Compute FIRST

First(C)

NT	First
S	$\{ a, \epsilon \}$
B	$\{b,\epsilon\}$
C	

$C \rightarrow cS$

C	→	c	S
A	→	α	

Rule 1
add α to $FIRST(A)$

$C \rightarrow \epsilon$

C	→	ϵ
A	→	ϵ

Rule 2
add ϵ to $FIRST(A)$

FIRST(B)=$\{ c , \epsilon \}$

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	→	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

Step 3: Prepare predictive parsing table

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

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

S→aB

a=FIRST(aB)={ a }

M[S,a]=S→aB

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

Example-2: LL(1) parsing

S→aB | ϵ
B→bC | ϵ
C→cS | ϵ

Step 3: Prepare predictive parsing table

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

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

S→ϵ

b=FOLLOW(S)={ $ }

M[S,$]=S→$\epsilon$

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

Example-2: LL(1) parsing

S→aB | ϵ
B→bC | ϵ
C→cS | ϵ

Step 3: Prepare predictive parsing table

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

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

B→bC

a=FIRST(bC)={ b }

M[B,b]=B→bC

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

Example-2: LL(1) parsing

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

Step 3: Prepare predictive parsing table

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

NT	Input Symbol			
	a	**b**	**c**	**\$**
S	$S \rightarrow aB$			$S \rightarrow \epsilon$
B		$B \rightarrow bC$		
C				

$B \rightarrow \epsilon$

b=FOLLOW(B)={ \$ }

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

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 3: Prepare predictive parsing table

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

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

C→cS

a=FIRST(cS)={ c }

M[C,c]=C→cS

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

Example-2: LL(1) parsing

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

Step 3: Prepare predictive parsing table

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

NT	Input Symbol			
	a	**b**	**c**	**\$**
S	$S \rightarrow aB$			$S \rightarrow \epsilon$
B		$B \rightarrow bB$		$B \rightarrow \epsilon$
C			$C \rightarrow cS$	

$C \rightarrow \epsilon$

b=FOLLOW(C)={ \$ }

M[C,\$]=$C \rightarrow \epsilon$

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

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

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

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-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

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

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

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 3: Construct predictive parsing table

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

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

Example-3: LL(1) parsing

Step 3: Construct predictive parsing table

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

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

Example-3: LL(1) parsing

Step 3: Construct predictive parsing table

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

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

Example-3: LL(1) parsing

Step 3: Construct predictive parsing table

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

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

Example-3: LL(1) parsing

Step 3: Construct predictive parsing table

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

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

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						

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

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

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 }	{ +,$,) }
F	{ (,id }	{*,+,$,)}

F→(E)

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

M[F,(]=F→(E)

Rule: 2

A→ α

a = first(α)

M[A,a] = A→ α

Example-3: LL(1) parsing

Step 3: Construct predictive parsing table

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

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

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

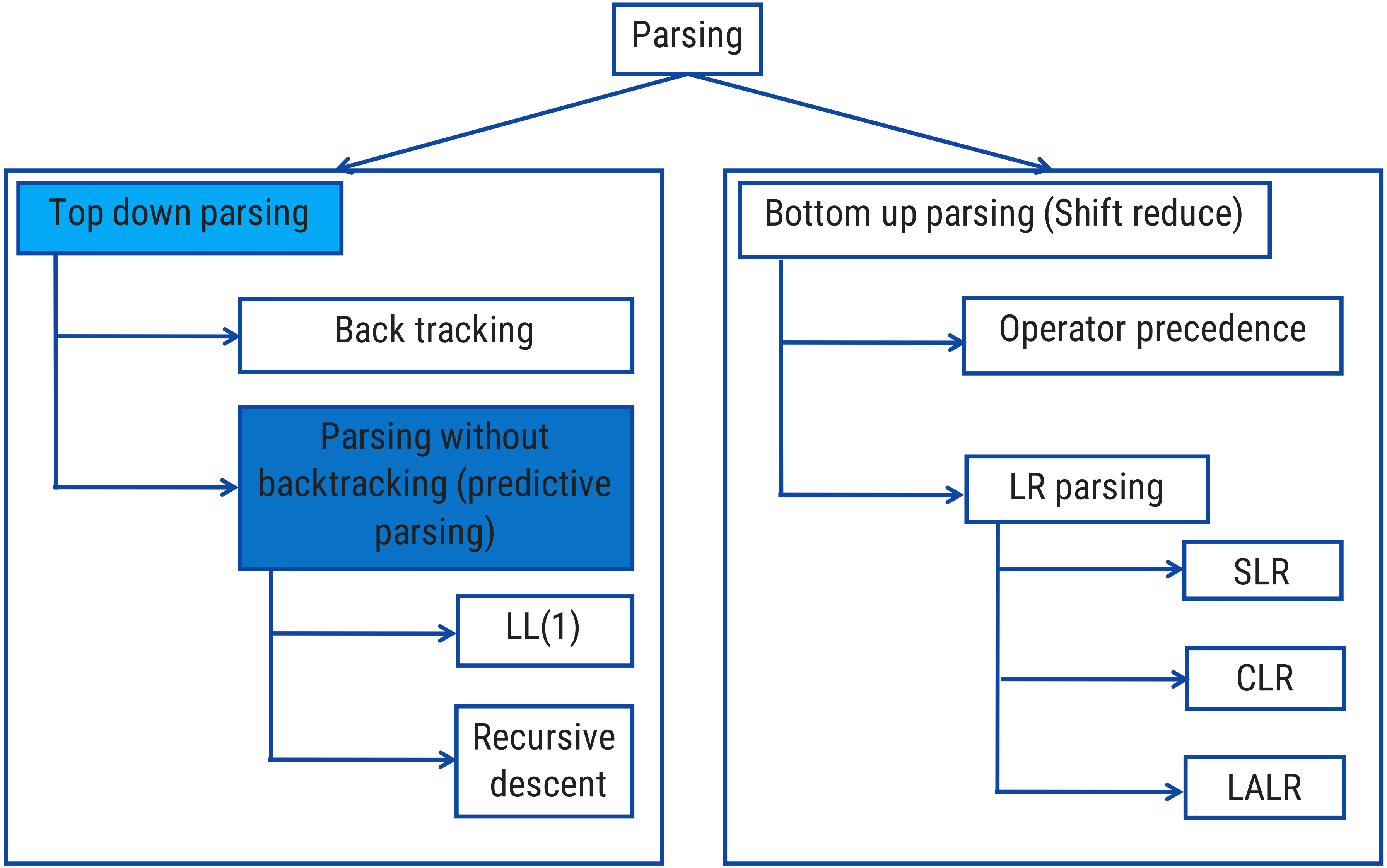
Example-3: LL(1) parsing

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.

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

Handle & Handle pruning

Handle & Handle pruning

- **Handle**: A “handle” of a string is a substring of the string that matches the right side of a production, and whose reduction to the non terminal of the production is one step along the reverse of rightmost derivation.
- **Handle pruning:** The process of discovering a handle and reducing it to appropriate left hand side non terminal is known as handle pruning.

E→E+E
E→E*E String: id1+id2*id3
E→id

Rightmost Derivation

E
E+E
E+E*E
E+E*id3
E+id2*id3
id1+id2*id3

Right sentential form	Handle	Production
id1+id2*id3		

Shift reduce parser

- The shift reduce parser performs following basic operations:

1. **Shift**: Moving of the symbols from input buffer onto the stack, this action is called shift.
2. **Reduce**: If handle appears on the top of the stack then reduction of it by appropriate rule is done. This action is called reduce action.
3. **Accept**: If stack contains start symbol only and input buffer is empty at the same time then that action is called accept.
4. **Error**: A situation in which parser cannot either shift or reduce the symbols, it cannot even perform accept action then it is called error action.

Example: Shift reduce parser

Grammar:

E→E+T | T

T→T*F | F

F→id

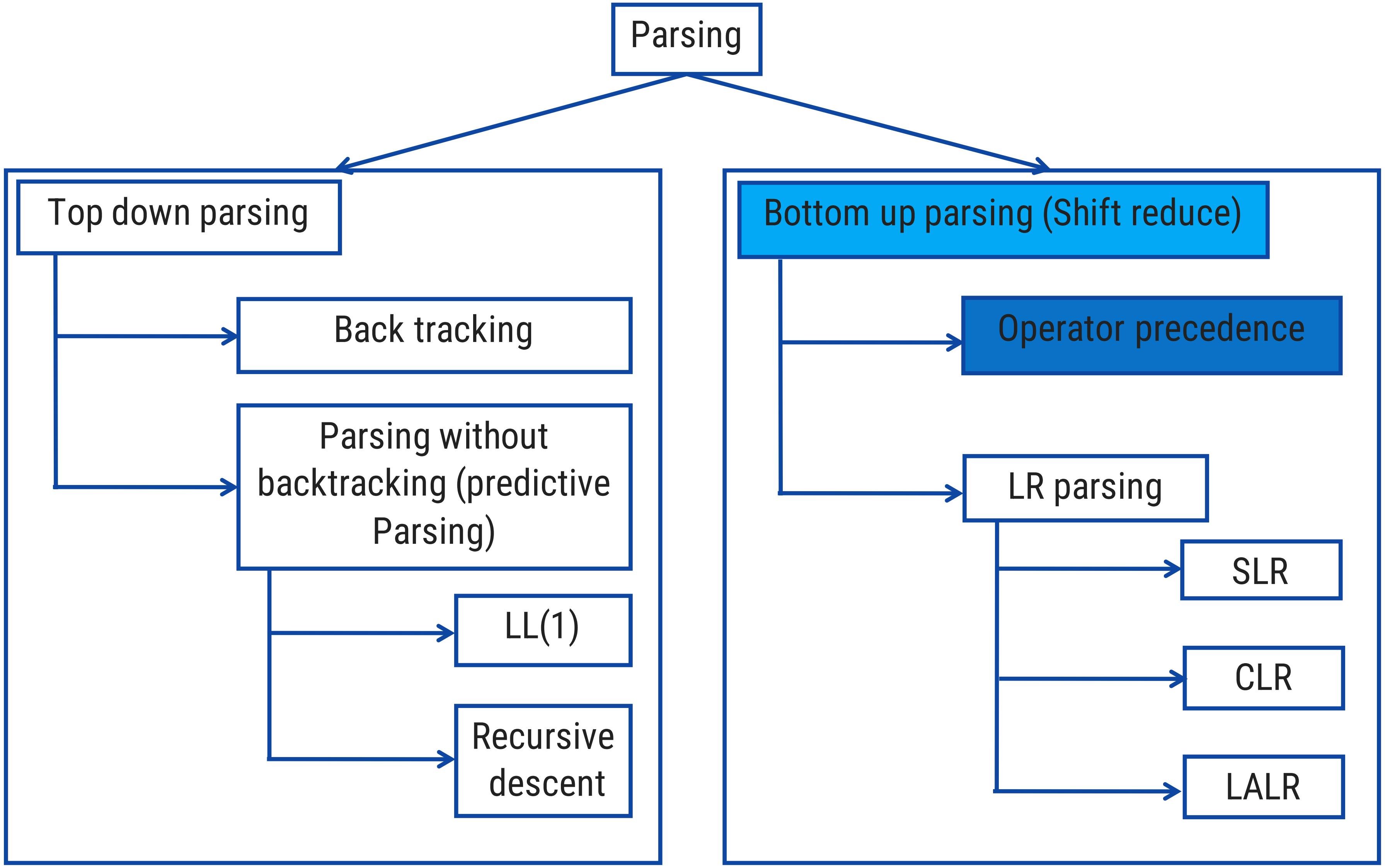
String: id+id*id

Stack	Input Buffer	Action

Viable Prefix

- The set of prefixes of right sentential forms that can appear on the stack of a shift-reduce parser are called viable prefixes.

Parsing Methods



Operator precedence parsing

Operator precedence parsing

- **Operator Grammar**: A Grammar in which there is no Є in RHS of any production or no adjacent non terminals is called operator grammar.
- Example: E→ EAE | (E) | id

  A→ + | * | -
- Above grammar is not operator grammar because right side ***EAE*** has consecutive non terminals.
- In operator precedence parsing we define following disjoint relations:

Relation	**Meaning**
a<·b	a “yields precedence to” b
a=b	a “has the same precedence as” b
a·>b	a “takes precedence over” b

Precedence & associativity of operators

Operator	Precedence	Associative
↑	1	right
*, /	2	left
+, -	3	left

Steps of operator precedence parsing

1. Find Leading and trailing of non terminal
2. Establish relation
3. Creation of table
4. Parse the string

Leading & Trailing

Leading:- Leading of a non terminal is the first terminal or operator in production of that non terminal.

Trailing:- Trailing of a non terminal is the last terminal or operator in production of that non terminal.

Example: $E \rightarrow E+T \mid T$

$T \rightarrow T*F \mid F$

$F \rightarrow id$

Non terminal	Leading	Trailing
E		
T		
F		

Rules to establish a relation

1. For a ≐ b, ⇒ aAb, where A is ϵ or a single non terminal [e.g : (E)]
2. a <· b ⇒ $Op\,.\,NT \;\; then \;\; Op \;\; <.\,Leading(NT)$ [e.g : +T]
3. a ·>b ⇒ $NT\,.\,Op \; then \; (Trailing(NT)) \cdot > Op$ [e.g : E+]
4. $ <· Leading (start symbol)
5. Trailing (start symbol) ·> $

Example: Operator precedence parsing

Step 1: Find Leading & Trailing of NT

Nonterminal	Leading	Trailing
E	{+,*,id}	{+,*,id}
T	{*,id}	{*,id}
F	{id}	{id}

E→ E+T| T
T→ T*F| F
F→ id

Step 2: Establish Relation

a <·b

$Op \cdot NT$	$Op \lessdot Leading(NT)$
$+T$	$+ \lessdot \{*, id\}$
$*F$	$* \lessdot \{id\}$

Step3: Creation of Table

	+	*	id	$
+				
*				
id				
$				

Example: Operator precedence parsing

Step 1: Find Leading & Trailing of NT

Nonterminal	Leading	Trailing
E	{+,*,id}	{+,*,id}
T	{*,id}	{*,id}
F	{id}	{id}

E→ E+ T| T
T→ T* F| F
F→ id

Step2: Establish Relation

a ·>b

$$NT \cdot Op \mid (Trailing(NT)) \cdot > Op$$

$$E + \qquad \{+,*,id\} \cdot > \ +$$

$$T * \qquad \{*,id\} \cdot > *$$

Step3: Creation of Table

	+	*	id	$
+		<·	<·	
*****			<·	
id				
$				

Example: Operator precedence parsing

Step 1: Find Leading & Trailing of NT

Nonterminal	Leading	Trailing
E	{+,*,id}	{+,*,id}
T	{*,id}	{*,id}
F	{id}	{id}

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

Step 2: Establish Relation

$<· Leading (start symbol)
$ <· $\{+, *, id\}$
Trailing (start symbol) ·> $
$\{+, *, id\}$ ·> $

Step 3: Creation of Table

	+	*	id	$
+	·>	<·	<·	
*****	·>	·>	<·	
id	·>	·>		
$				

Example: Operator precedence parsing

Step 4: Parse the string using precedence table

Assign precedence operator between terminals

String: id+id*id

$ id+id*id $

$ <· id+id*id$

$ <· id ·> +id*id$

$ <· id ·> + <· id*id$

$ <· id ·> + <· id ·> *id$

$ <· id ·> + <· id ·> *<· id$

$ <· id ·> + <· id ·> *<· id ·> $

	+	*****	**id**	**$**
+	·>	<·	<·	·>
*****	·>	·>	<·	·>
id	·>	·>		·>
$	<·	<·	<·	

Example: Operator precedence parsing

Step 4: Parse the string using precedence table

E→E+T | T
T→T*F | F
F→id

1. Scan the input string until first ·> is encountered.
2. Scan backward until <· is encountered.
3. The handle is string between <· and ·>

$ <· Id ·> + <· Id ·> * <· Id ·> $	
$ F + <· Id ·> * <· Id ·> $	
$ F + F * <· Id ·> $	
$ F + F * F $	
$ E + T * F $	
$ + * $	
$ <· + <· * >$	
$ <· + >$	
$ $	

	+	*	id	$
+	·>	<·	<·	·>
*****	·>	·>	<·	·>
id	·>	·>		·>
$	<·	<·	<·	

Operator precedence function

Operator precedence function

Algorithm for constructing precedence functions

1. Create functions f_a and g_a for each a that is terminal or \$.
2. Partition the symbols in as many as groups possible, in such a way that f_a and g_b are in the same group if $a \; = \; b$.
3. Create a directed graph whose nodes are in the groups, next for each symbols $a \; and \; b$ do:
    a) if $a \; <\cdot \; b$, place an edge from the group of g_b to the group of f_a
    b) if $a \; \cdot> \; b$, place an edge from the group of f_a to the group of g_b
4. If the constructed graph has a cycle then no precedence functions exist. When there are no cycles collect the length of the longest paths from the groups of f_a and g_b respectively.

Operator precedence function

1. Create functions f_a and g_a for each a that is terminal or $.

$$a = \{+, *, id\} \text{ or } \$$$

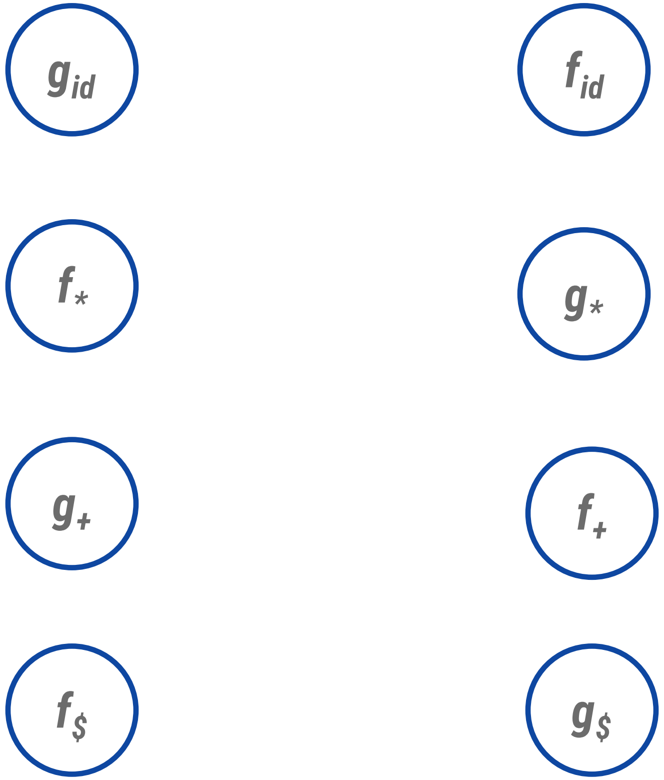
E→ E+T | T
T→ T*F | F
F→ id

f_+ f_* f_{id} $f_\$$

g_+ g_* g_{id} $g_\$$

Operator precedence function

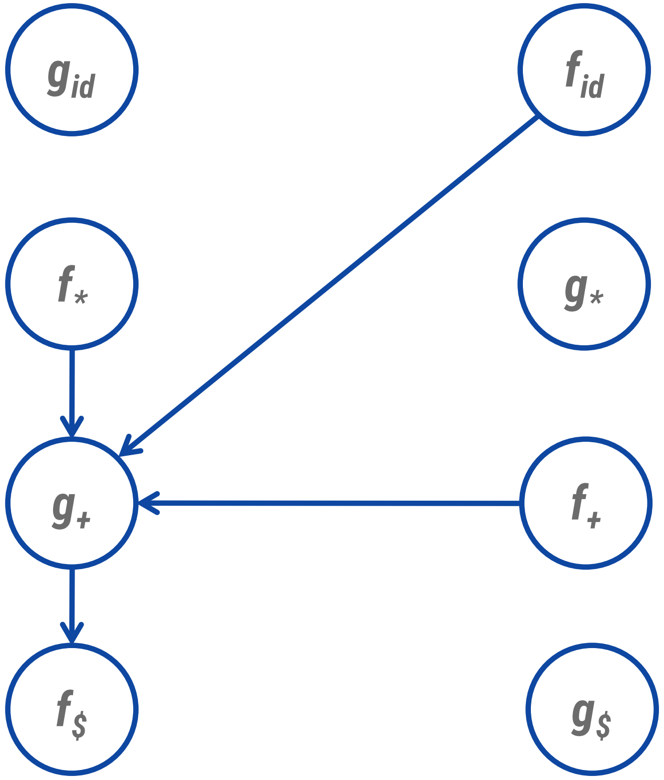
- *Partition the symbols in as many as groups possible, in such a way that f_a and g_b are in the same group if $a \doteq b$.*



	+	*	id	$
+	·>	<·	<·	·>
*	·>	·>	<·	·>
id	·>	·>		·>
$	<·	<·	<·	

Operator precedence function

3. *if a <· b, place an edge from the group of g_b to the group of f_a*
   if a ·> b, place an edge from the group of f_a to the group of g_b



f \ g	+	*	id	$
+	·>	<·	<·	·>
*	·>	·>	<·	·>
id	·>	·>		·>
$	<·	<·	<·	

$f_+ \cdot> g_+$ $\qquad$ $f_+ \rightarrow g_+$

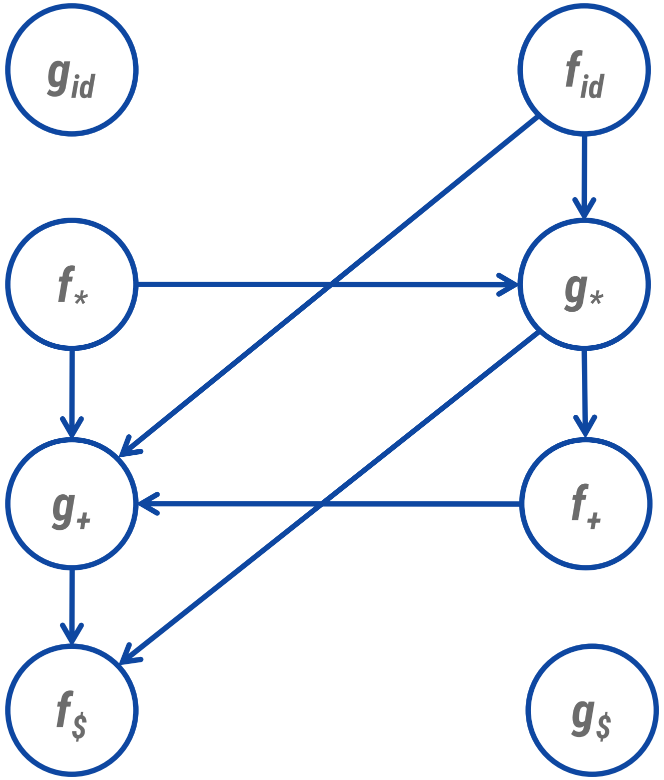
$f_* \cdot> g_+$ $\qquad$ $f_* \rightarrow g_+$

$f_{id} \cdot> g_+$ $\qquad$ $f_{id} \rightarrow g_+$

$f_\$ <\cdot\ g_+$ $\qquad$ $f_\$ \leftarrow g_+$

Operator precedence function

3. *if a <· b, place an edge from the group of g_b to the group of f_a*
   if a ·> b, place an edge from the group of f_a to the group of g_b



		g			
f		+	*	id	$
	+	·>	<·	<·	·>
	*	·>	·>	<·	·>
	id	·>	·>		·>
	$	<·	<·	<·	

$f_+ <\cdot\ g_*$ $\qquad$ $f_+ \leftarrow g_*$

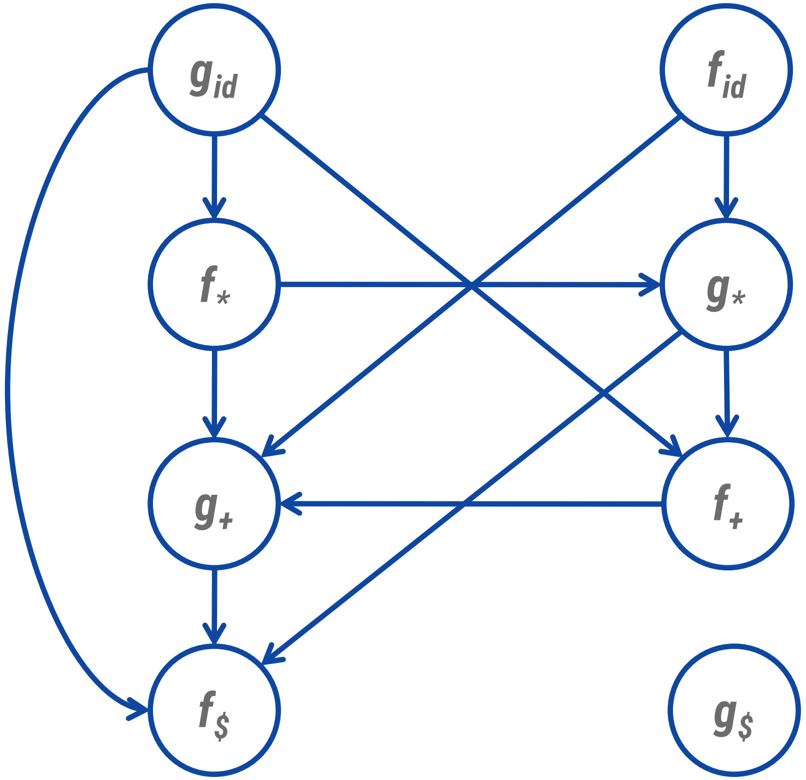
$f_* \cdot> g_*$ $\qquad$ $f_* \rightarrow g_*$

$f_{id} \cdot> g_*$ $\qquad$ $f_{id} \rightarrow g_*$

$f_\$ <\cdot\ g_*$ $\qquad$ $f_\$ \leftarrow g_*$

Operator precedence function

3. *if a <· b, place an edge from the group of g_b to the group of f_a*
   if a ·> b, place an edge from the group of f_a to the group of g_b



	g			
f	**+**	*****	**id**	**$**
+	·>	<·	<·	·>
*****	·>	·>	<·	·>
id	·>	·>		·>
$	<·	<·	<·	

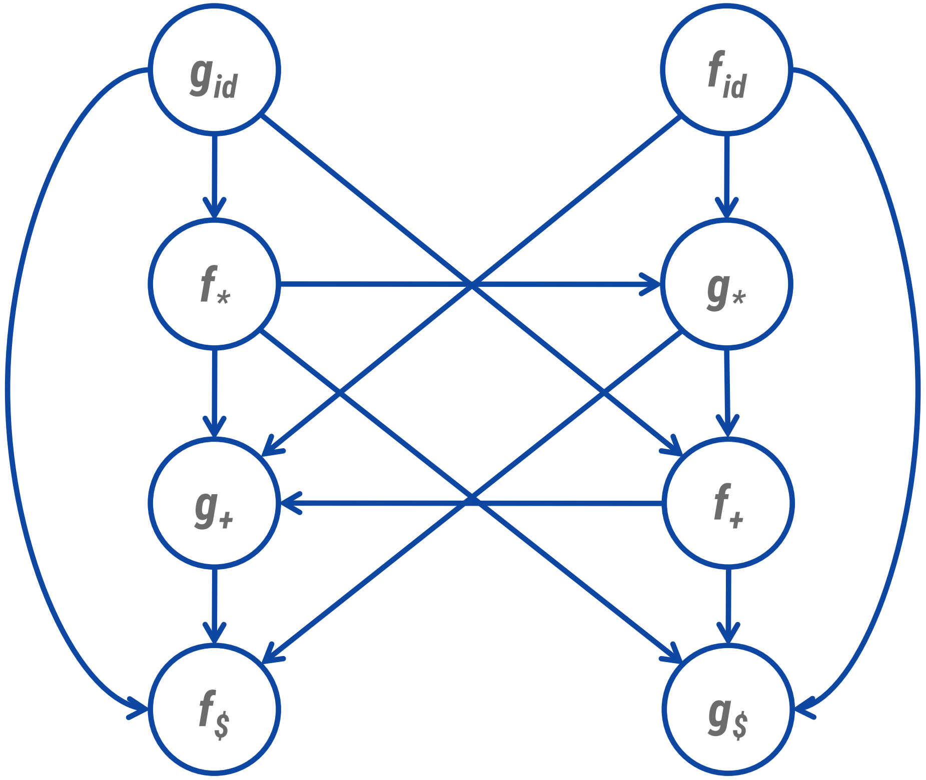
$f_+ <\cdot\ g_{id}$ $\qquad$ $f_+ \leftarrow g_{id}$

$f_* <\cdot\ g_{id}$ $\qquad$ $f_* \leftarrow g_{id}$

$f_\$ <\cdot\ g_{id}$ $\qquad$ $f_\$ \leftarrow g_{id}$

Operator precedence function

3. *if a <· b, place an edge from the group of g_b to the group of f_a*
   if a ·> b, place an edge from the group of f_a to the group of g_b



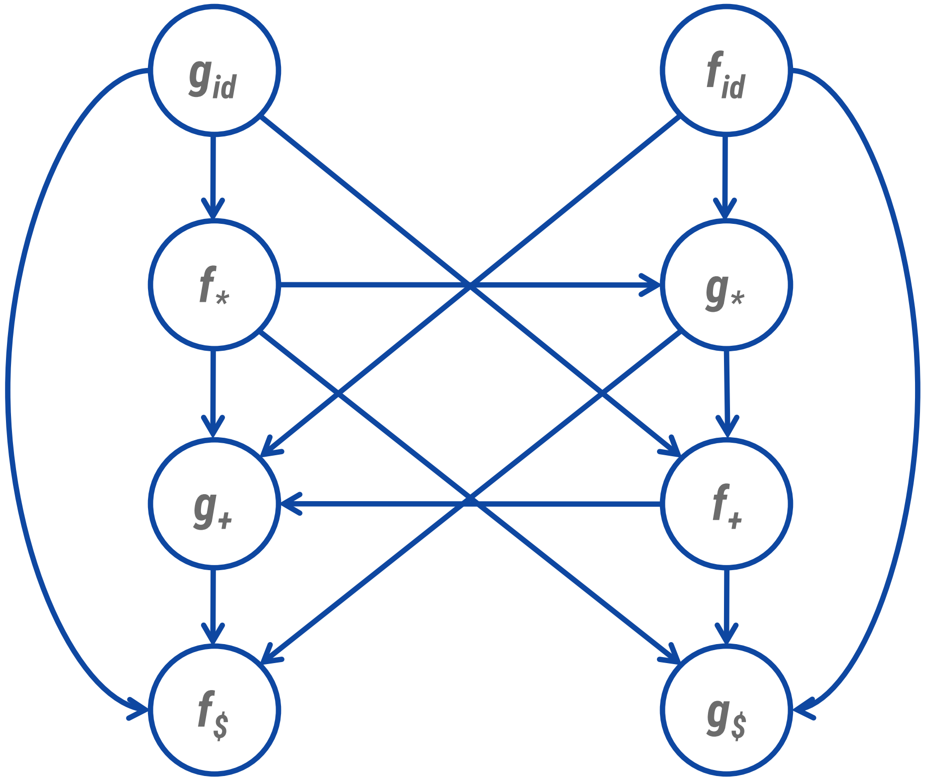
f \ g	+	*	id	$
+	·>	<·	<·	·>
*	·>	·>	<·	·>
id	·>	·>		·>
$	<·	<·	<·	

$f_+ \lessdot g_\$$ $\quad$ $f_+ \rightarrow g_\$$

$f_* \lessdot g_\$$ $\quad$ $f_* \rightarrow g_\$$

$f_{id} \lessdot g_\$$ $\quad$ $f_{id} \rightarrow g_\$$

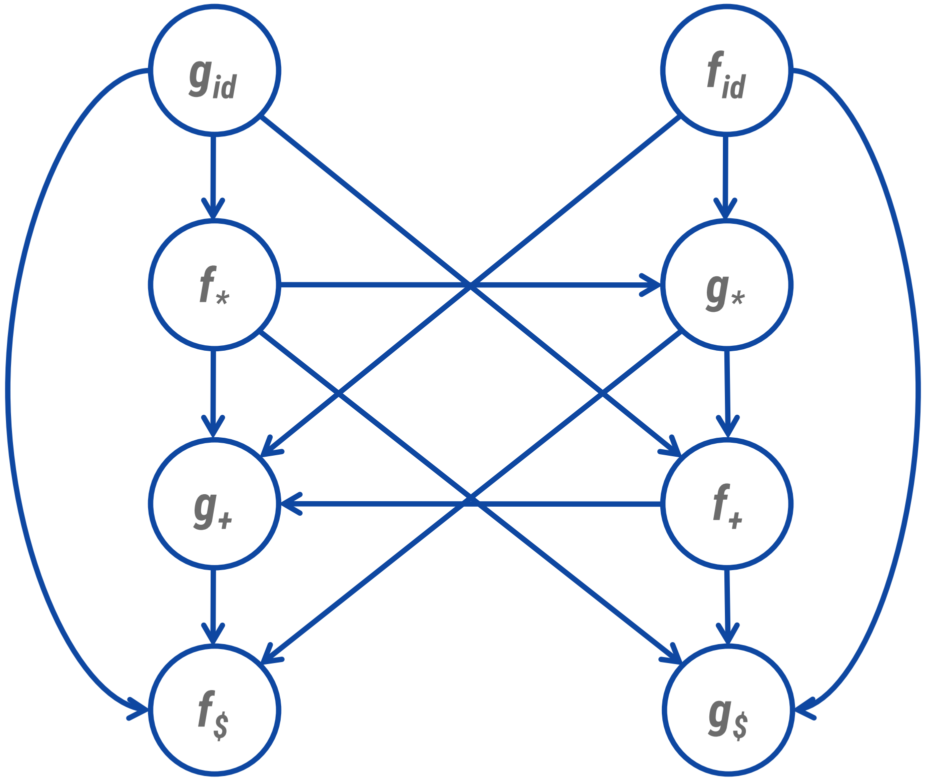
Operator precedence function



	+	*	id	$
f				
g				

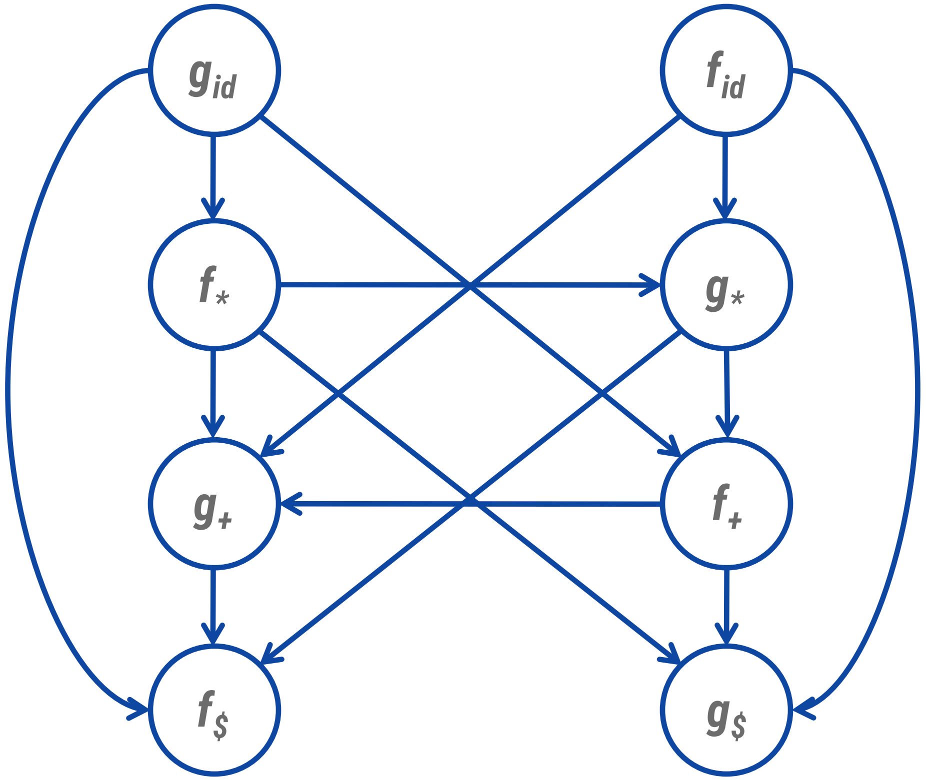
4. *If the constructed graph has a cycle then no precedence functions exist. When there are no cycles collect the length of the longest paths from the groups of f_a and g_b respectively.*

Operator precedence function



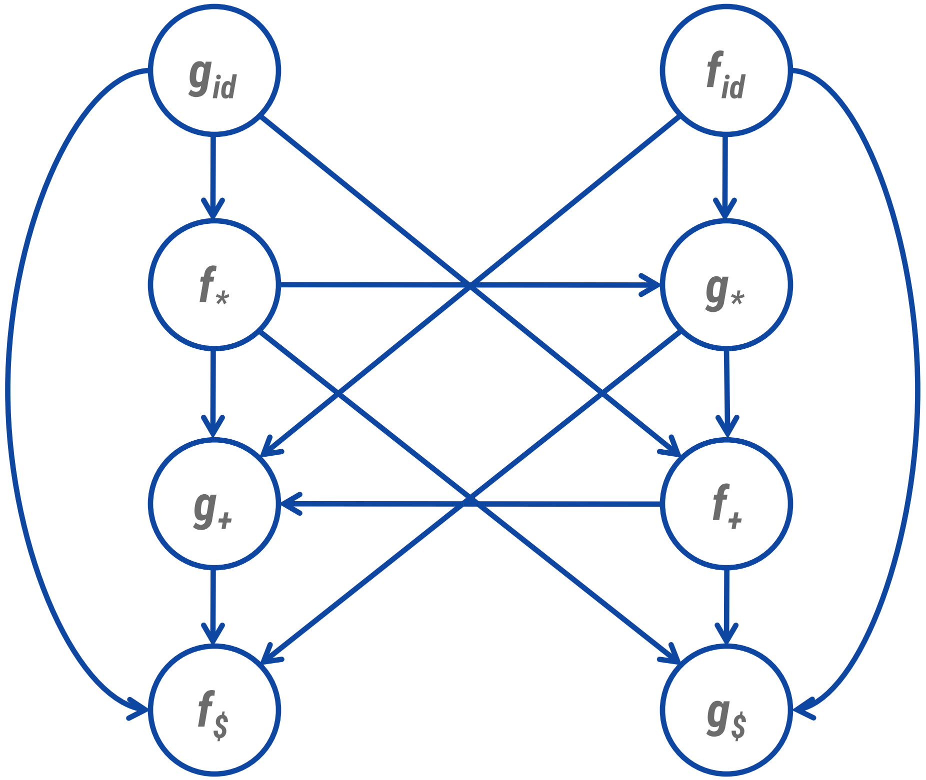
	+	*	id	$
f	2			
g				

Operator precedence function



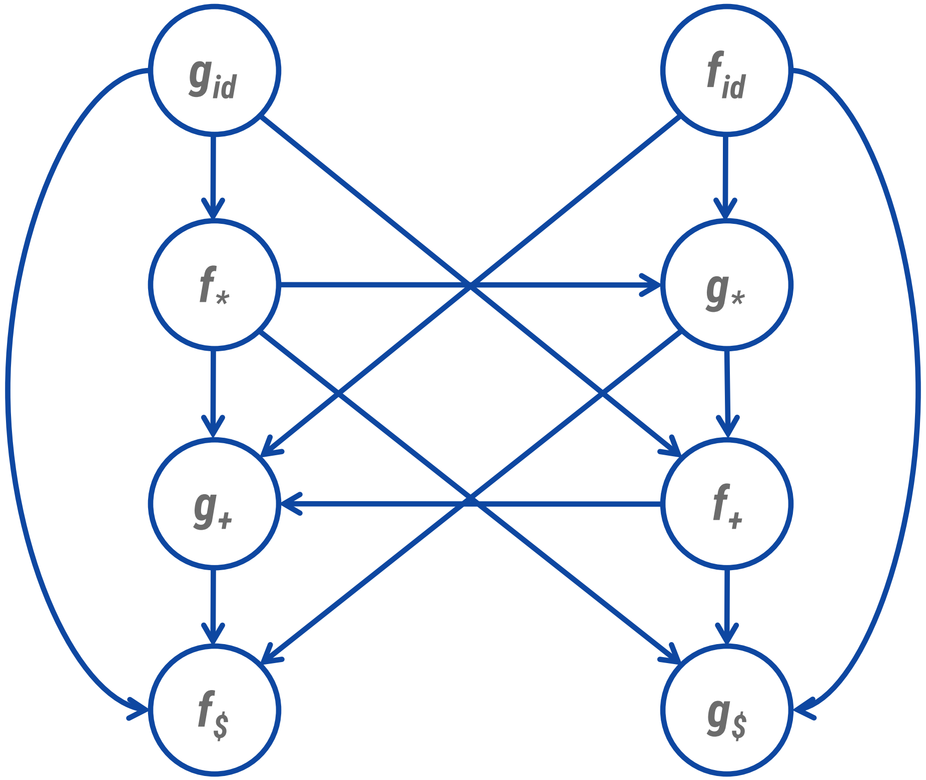
	+	*	id	$
f	2			
g	1			

Operator precedence function



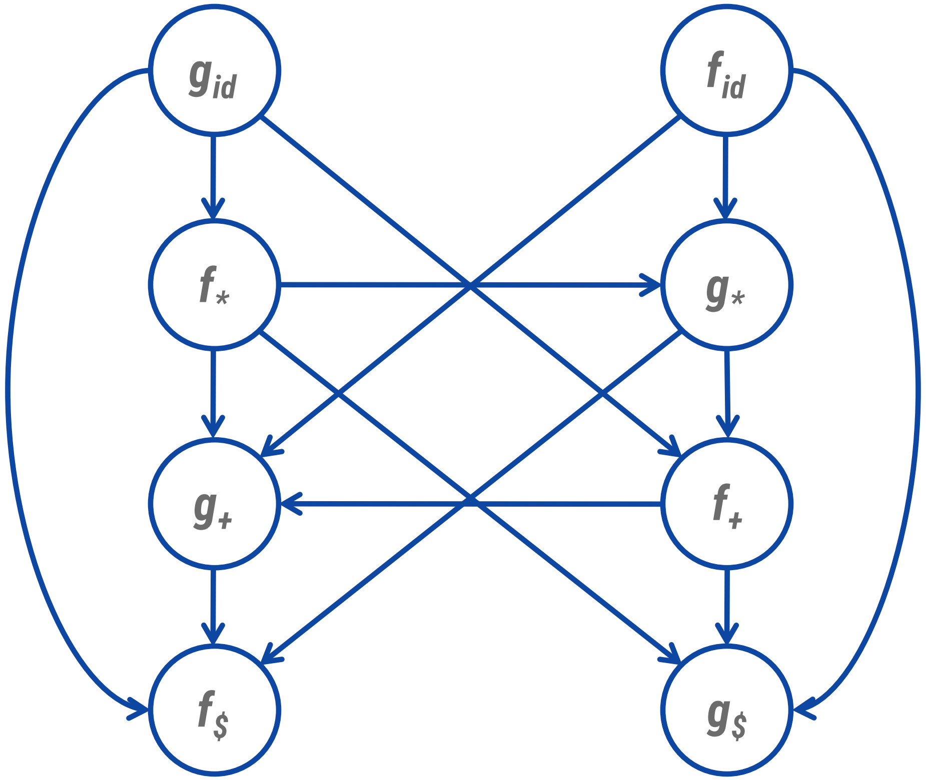
	+	*	id	$
f	2	4		
g	1			

Operator precedence function



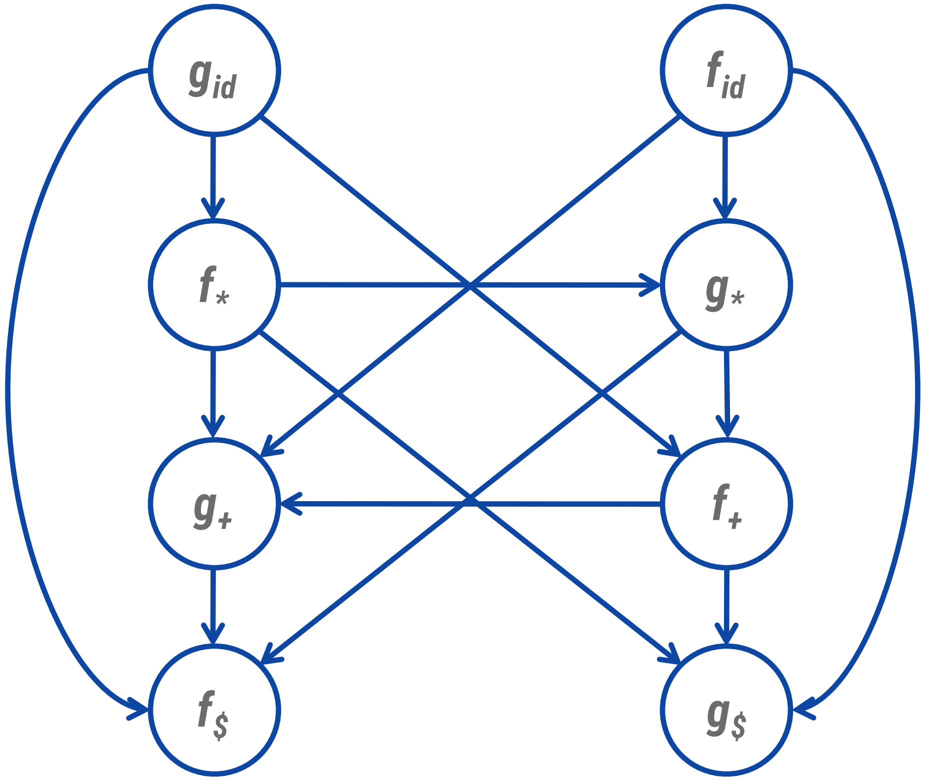
	+	*	id	$
f	2	4		
g	1	3		

Operator precedence function



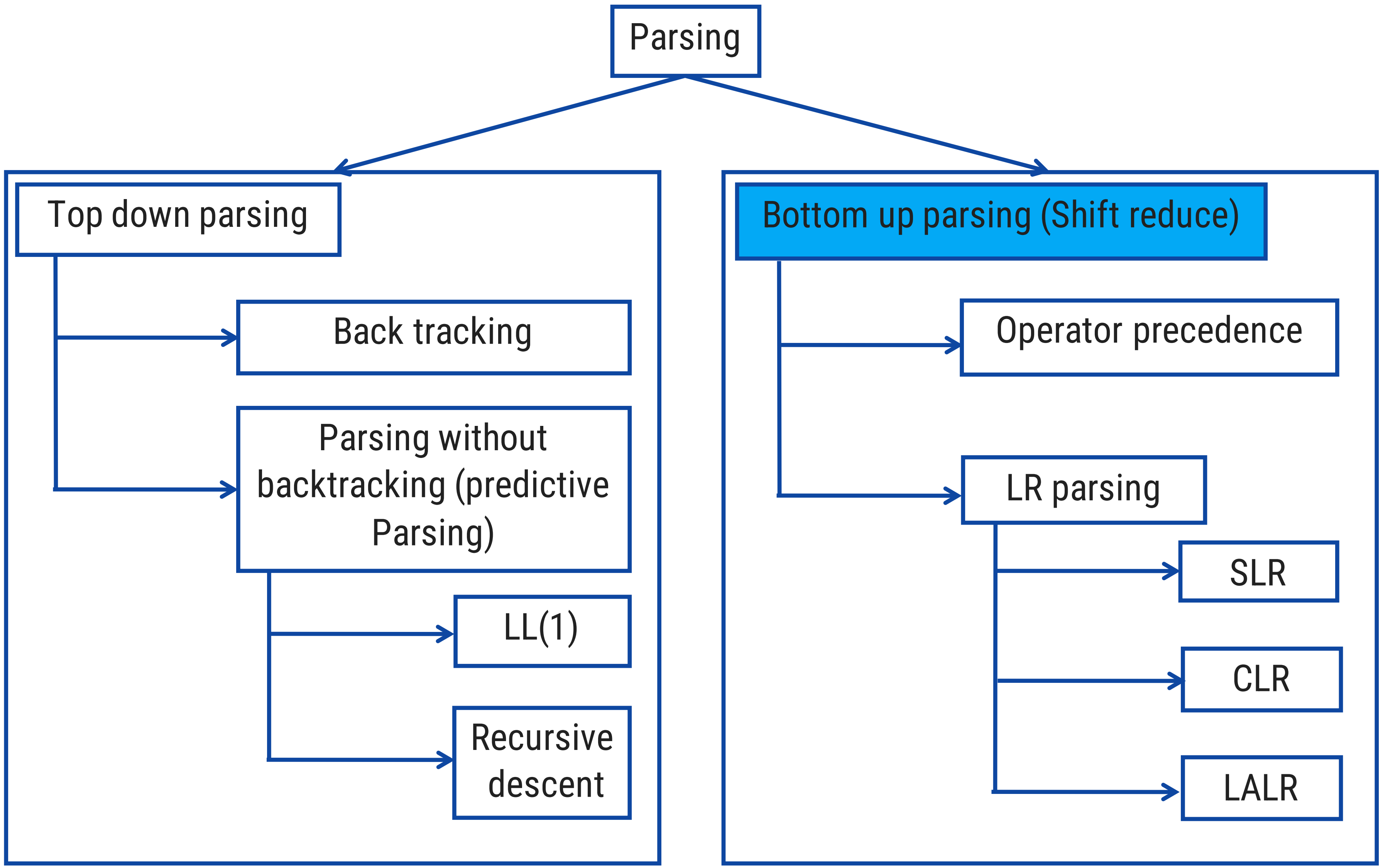
	+	*	id	$
f	2	4	4	
g	1	3		

Operator precedence function



	+	*	id	$
f	2	4	4	
g	1	3	5	

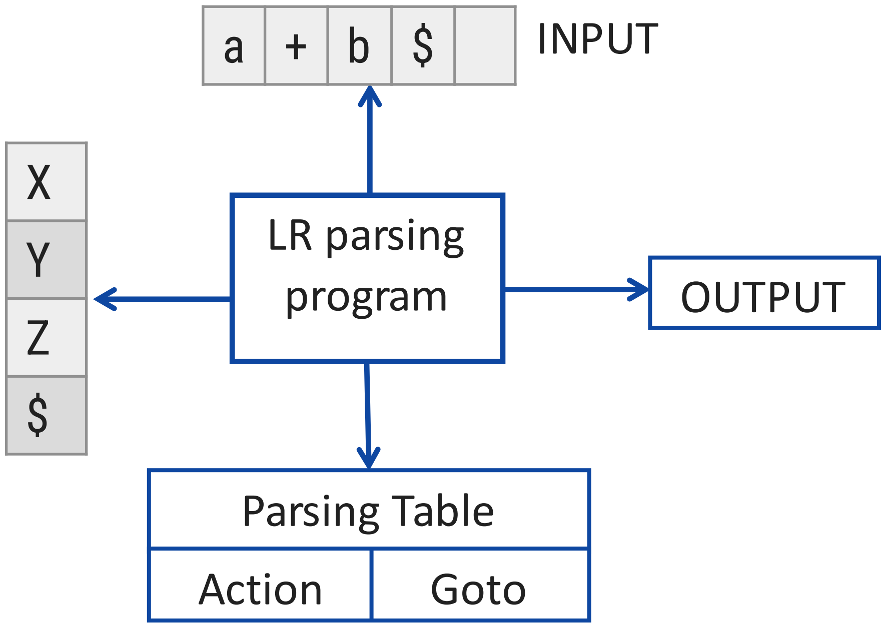
Parsing Methods



Introduction to LR Parser

LR parser

- LR parsing is most efficient method of bottom up parsing which can be used to parse large class of context free grammar.
- The technique is called LR(k) parsing:
  1. The "L" is for left to right scanning of input symbol,
  2. The "R" for constructing right most derivation in reverse,
  3. The "k" for the number of input symbols of look ahead that are used in making parsing decision.



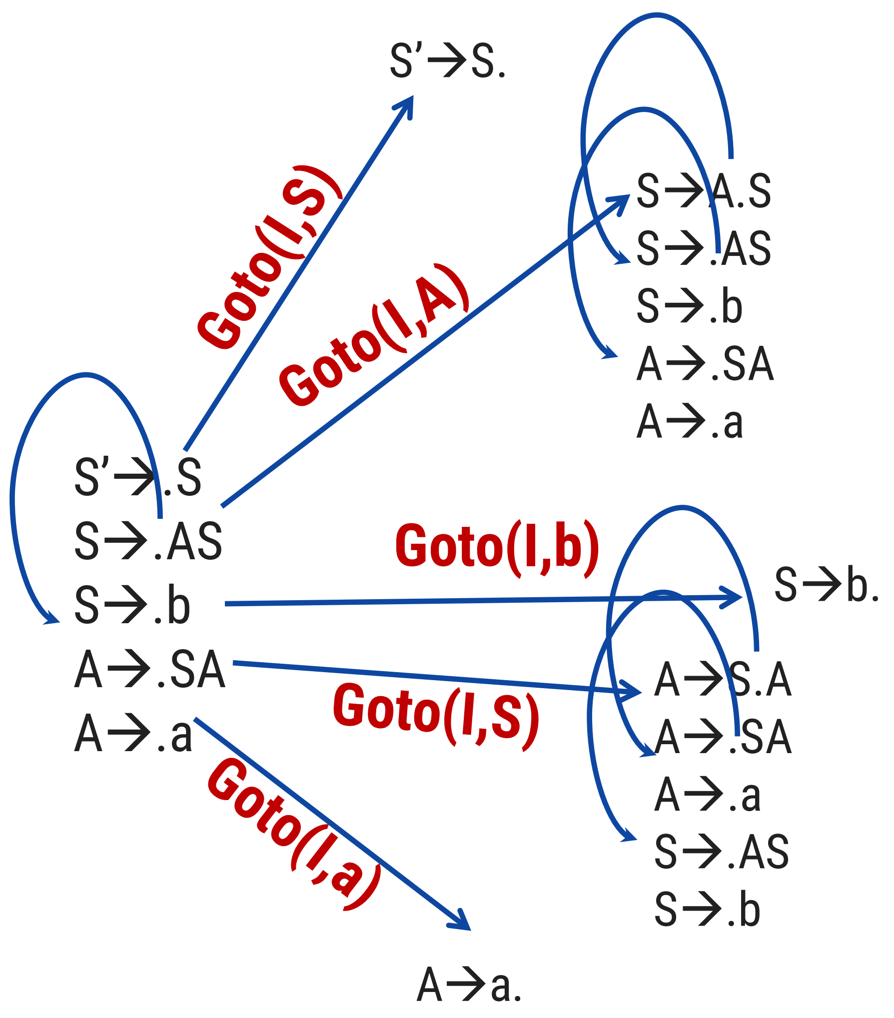
Closure & goto function

Computation of closure & goto function

S→AS | b

A→SA | a

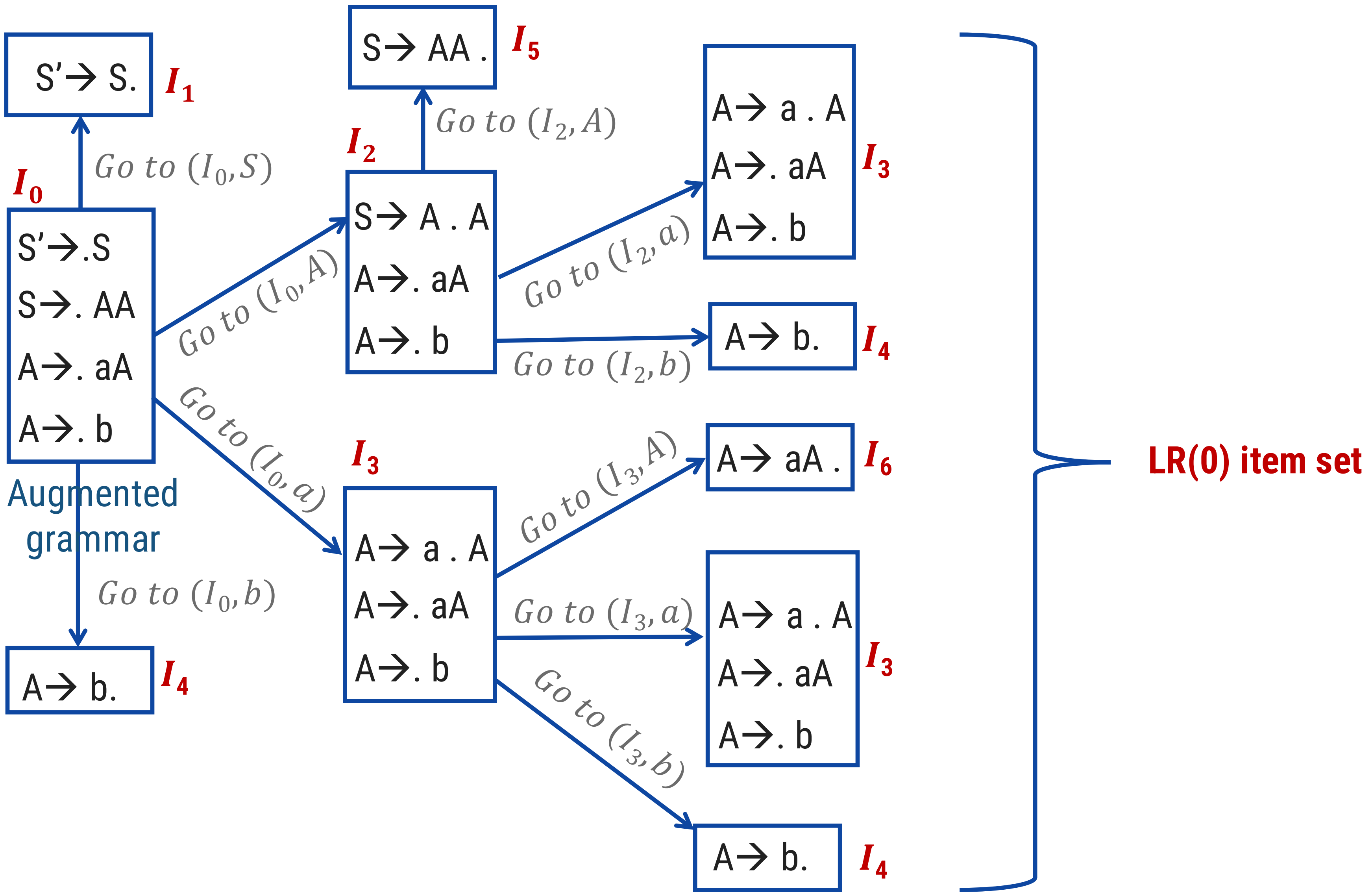
Closure(I):



SLR Parser

Example: SLR(1)- simple LR

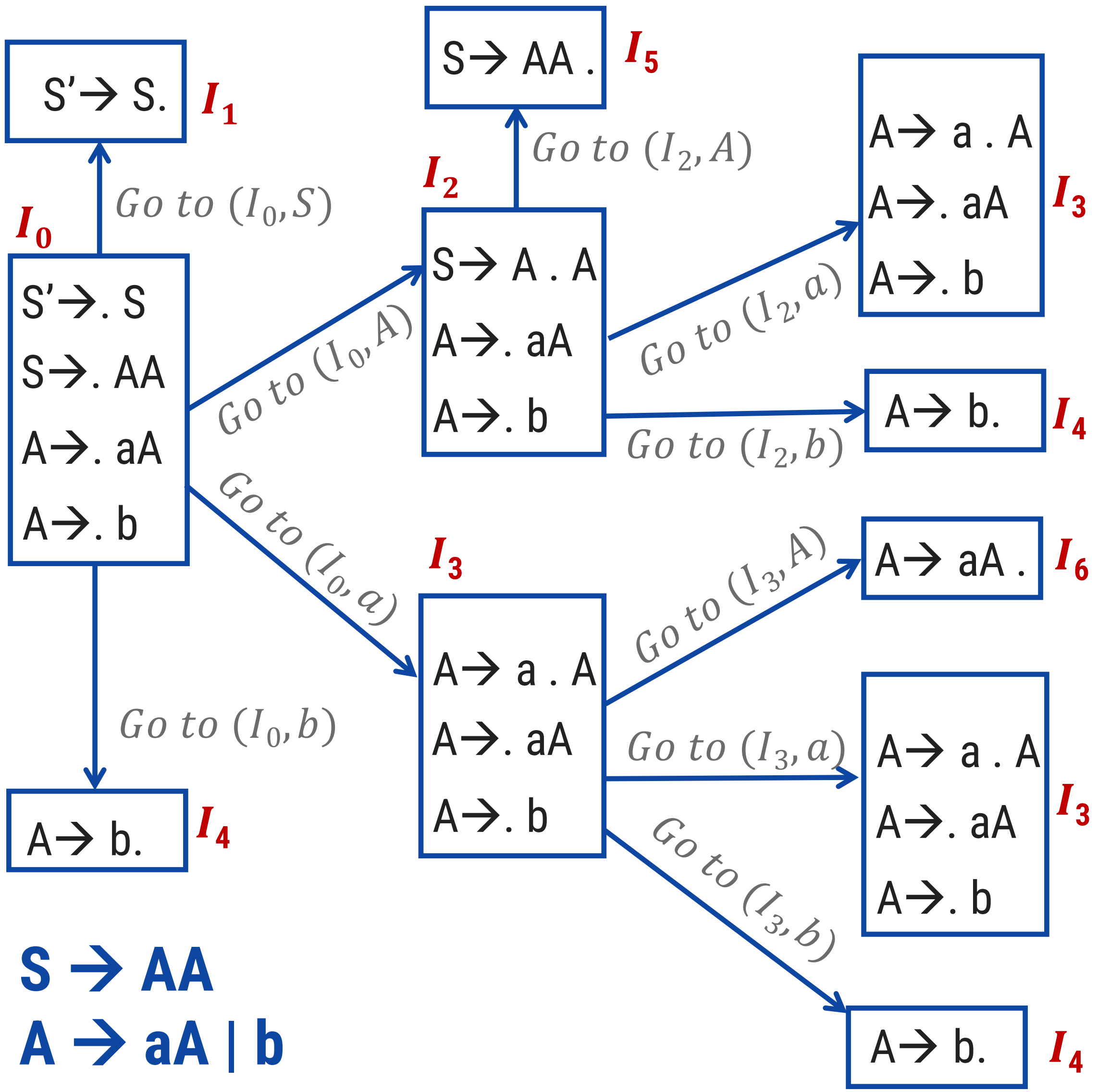
S → AA
A → aA | b



Rules to construct SLR parsing table

1. Construct $C = \{ I_0, I_1, \dots \dots . In\}$, the collection of sets of LR(0) items for G'.
2. State i is constructed from I_i. The parsing actions for state i are determined as follow :
   a) If $[A \rightarrow \alpha . a\beta]$ is in I_i and GOTO $(Ii , a) = I_j$, then set $ACTION[i, a]$ to "shift j". Here a must be terminal.
   b) If $[A \rightarrow \alpha .]$ is in I_i, then set $ACTION[i, a]$ to "reduce A→ α" for all a in $FOLLOW(A)$; here A may not be S'.
   c) If $[S \rightarrow S .]$ is in I_i, then set action $[i, \$]$ to "accept".
3. The goto transitions for state i are constructed for all non terminals A using the $if\ GOTO(Ii , A) = I_j\ then\ GOTO\ [i, A] = j$.
4. All entries not defined by rules 2 and 3 are made error.

Example: SLR(1)- simple LR



S → AA
A → aA | b

$Follow(S) = \{\$\}$
$Follow(A) = \{a, b, \$\}$

	Action			Go to	
Item set	a	b	$	S	A
0					
1					
2					
3					
4					
5					
6					

CLR Parser

How to calculate look ahead?

How to calculate look ahead?

S→CC

C→ cC | d

Closure(I)

S'→.S,$

S→.CC,$

C→.cC, c|d

C→.d,c|d

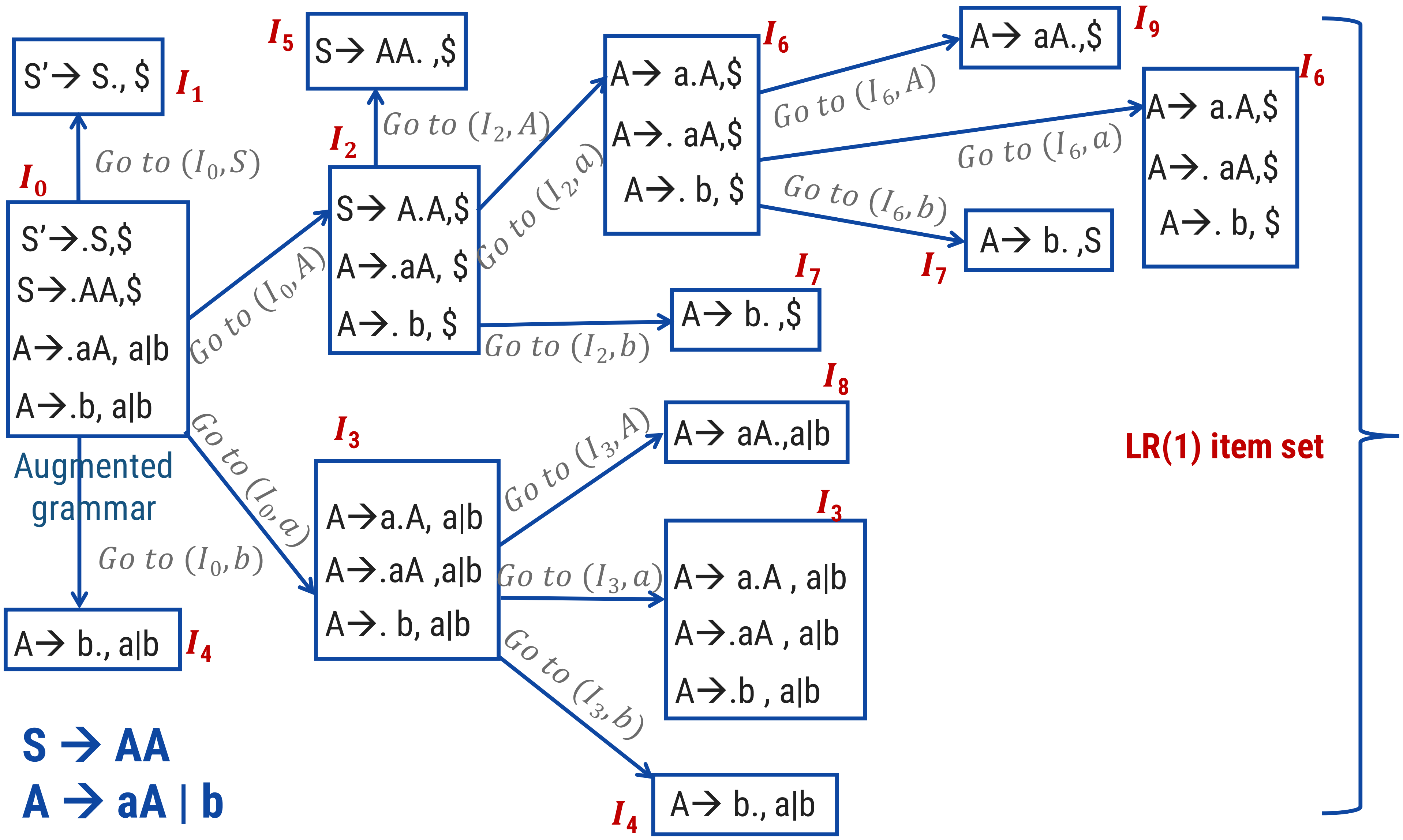
S'	→		.	S		,	$
A	→	α	.	X	β	,	a

$$\text{Lookahead} = \text{First}(\boldsymbol{\beta a})$$
$$\text{First}(\$)$$
$$= \$$$

S	→		.	C	C	,	$
A	→	α	.	X	β	,	a

$$\text{Lookahead} = \text{First}(\boldsymbol{\beta a})$$
$$\text{First}(C\$)$$
$$= \boldsymbol{c, d}$$

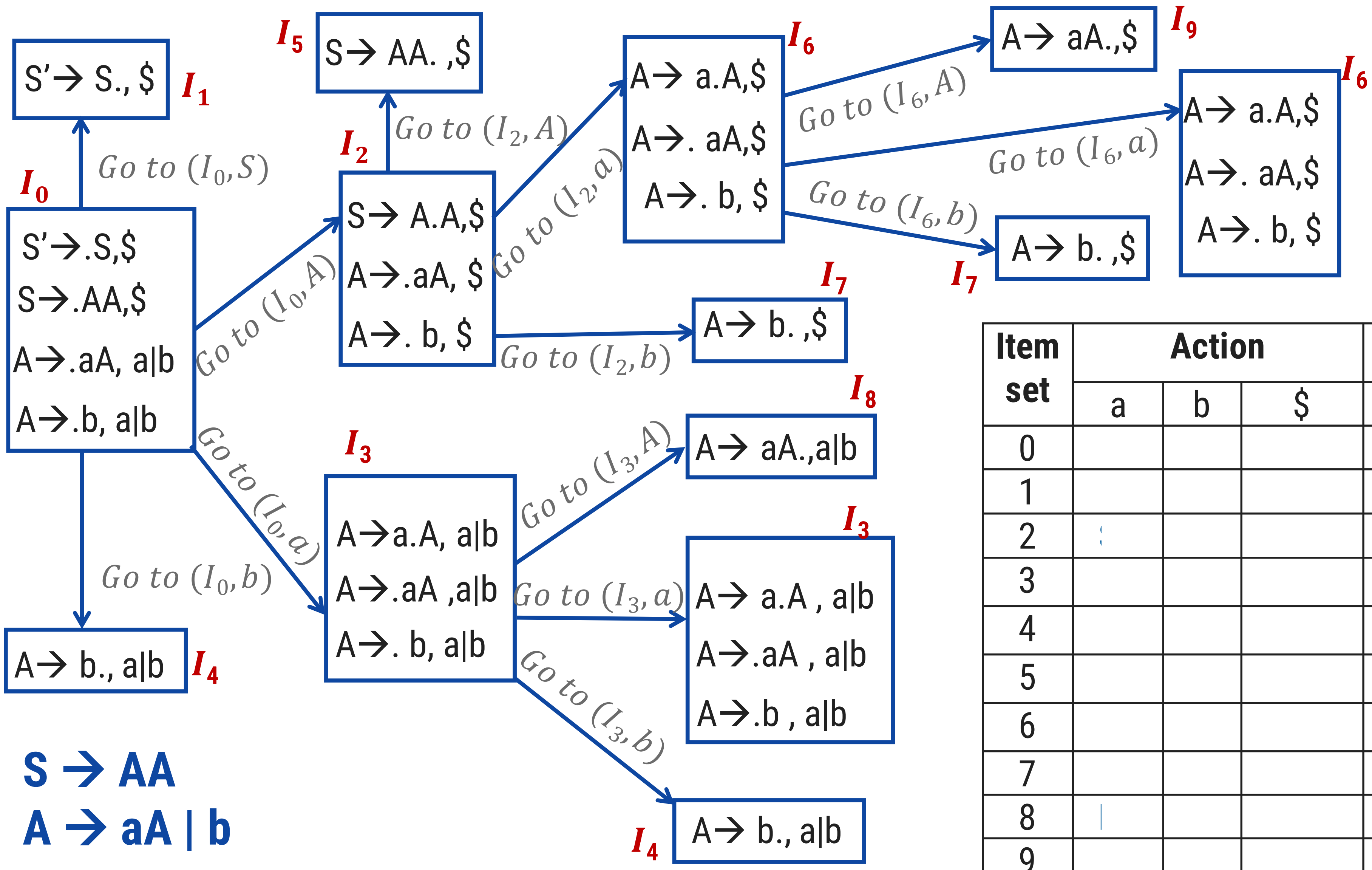
Example: CLR(1)- canonical LR



S → AA

A → aA | b

Example: CLR(1)- canonical LR

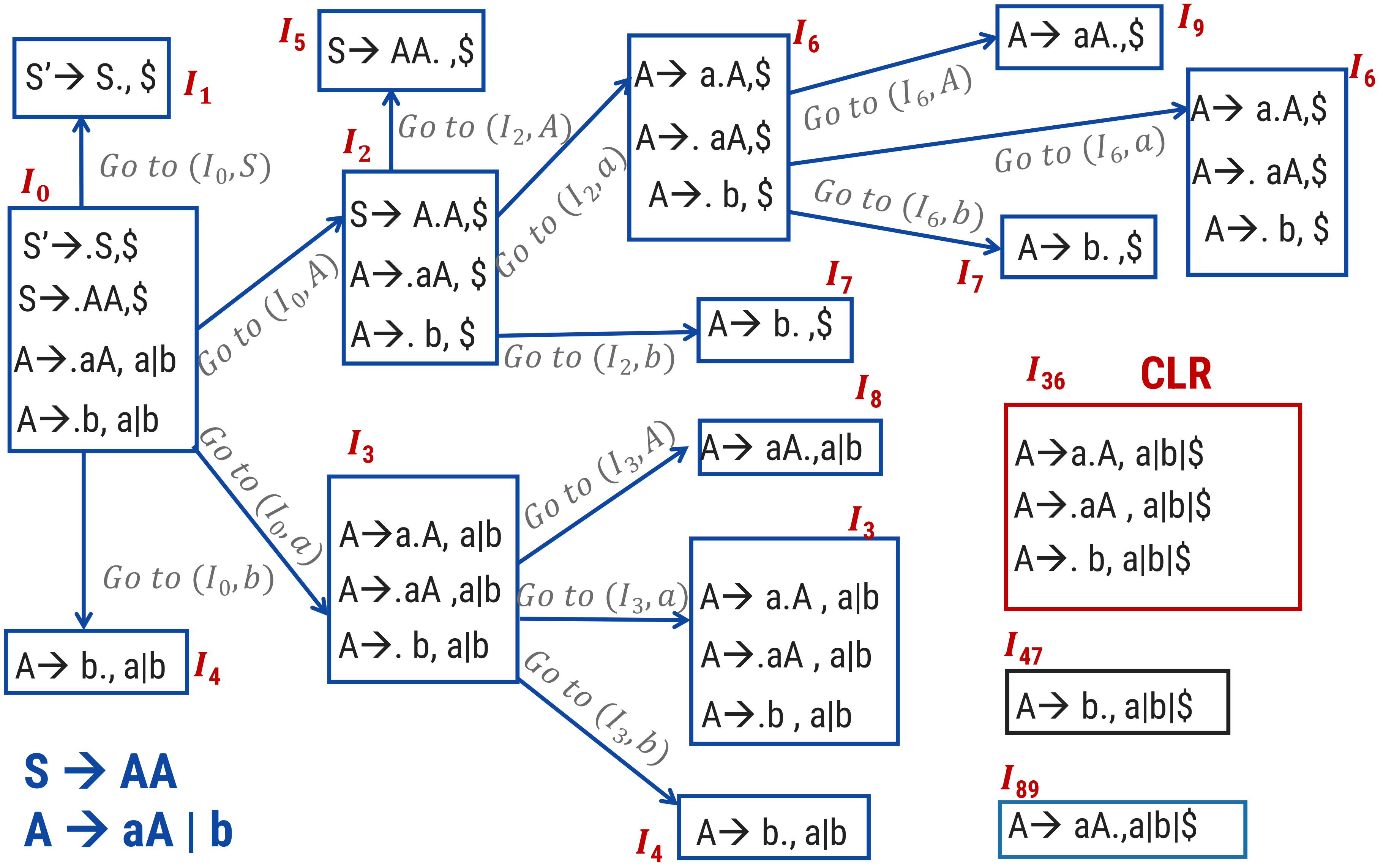


S → AA
A → aA | b

Item set	Action			Go to	
	a	b	$	S	A
0					
1					
2					
3					
4					
5					
6					
7					
8					
9					

LALR Parser

Example: LALR(1)- look ahead LR



S → AA

A → aA | b

Example: LALR(1)- look ahead LR

Item set	Action			Go to	
	a	b	$	S	A
0	S3	S4		1	2
1			Accept		
2	S6	S7			5
3	S3	S4			8
4	R3	R3			
5			R1		
~~6~~	~~S6~~	~~S7~~			~~9~~
~~7~~			~~R3~~		
8	R2	R2			
~~9~~			~~R2~~		

CLR Parsing Table

Item set	Action			Go to	
	a	b	$	S	A
0	S36	S47		1	2
1			Accept		
2	S36	S47			5
36	S36	S47			89
47	R3	R3	R3		
5			R1		
89	R2	R2	R2		

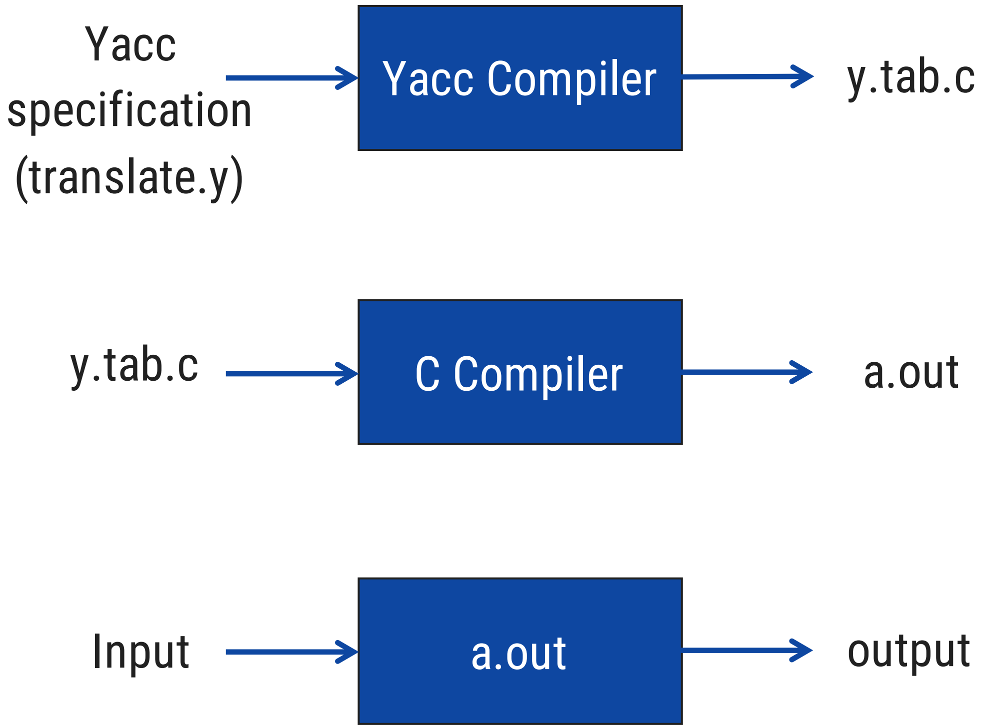
LALR Parsing Table

Parser Generator

(YACC)

YACC tool or YACC Parser Generator

- YACC is a tool which generates the parser.
- It takes input from the lexical analyzer (tokens) and produces parse tree as an output.



Structure of Yacc Program

- Any Yacc program contains mainly three sections
  1. Declaration
  2. Translation rules
  3. Supporting C-routines

Structure of Program

Declaration →

<left side>→<alt 1>|<alt 2>|……..|<alt n>

%%

%%

Translation rule →

<left side> : <alt 1> {semantic action 1}
| <alt 2> {semantic action 2}
| <alt n> {semantic action n}

%%

%%

Supporting C routines → All the function needed are specified over here.

Example: Yacc Program

- Program: Write Yacc program for simple desk calculator

```
/* Declaration */
%{
        #include <ctype.h>
%}
% token DIGIT
```

```
/* Translation rule */
%%
line      : expr '\n'           {print("%d\n",$1);}
expr      : expr '+' term       {$$=$1 + $3;}
          | term;
term      : term '*' factor     {$$=$1 * $3;}
          | factor;
factor    : '(' expr ')'        {$$=$2;}
          | DIGIT;
%%
```

```
/* Supporting C routines*/
yylex()
{
        int c;
        c=getchar();
        if(isdigit(c))
        {
                yylval= c-'0'
                return DIGIT
        }
        return c;
}
```

$E \rightarrow E+T \mid T$

$T \rightarrow T*F \mid F$

$F \rightarrow (E) \mid id$

References

Books:

1. **Compilers Principles, Techniques and Tools, PEARSON Education (Second Edition)**

   Authors: Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman

2. **Compiler Design, PEARSON (for Gujarat Technological University)**

   Authors: Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman

Thank You