Unit – 2
Parsing Theory (II)
BOTTOM –UP-PARSING

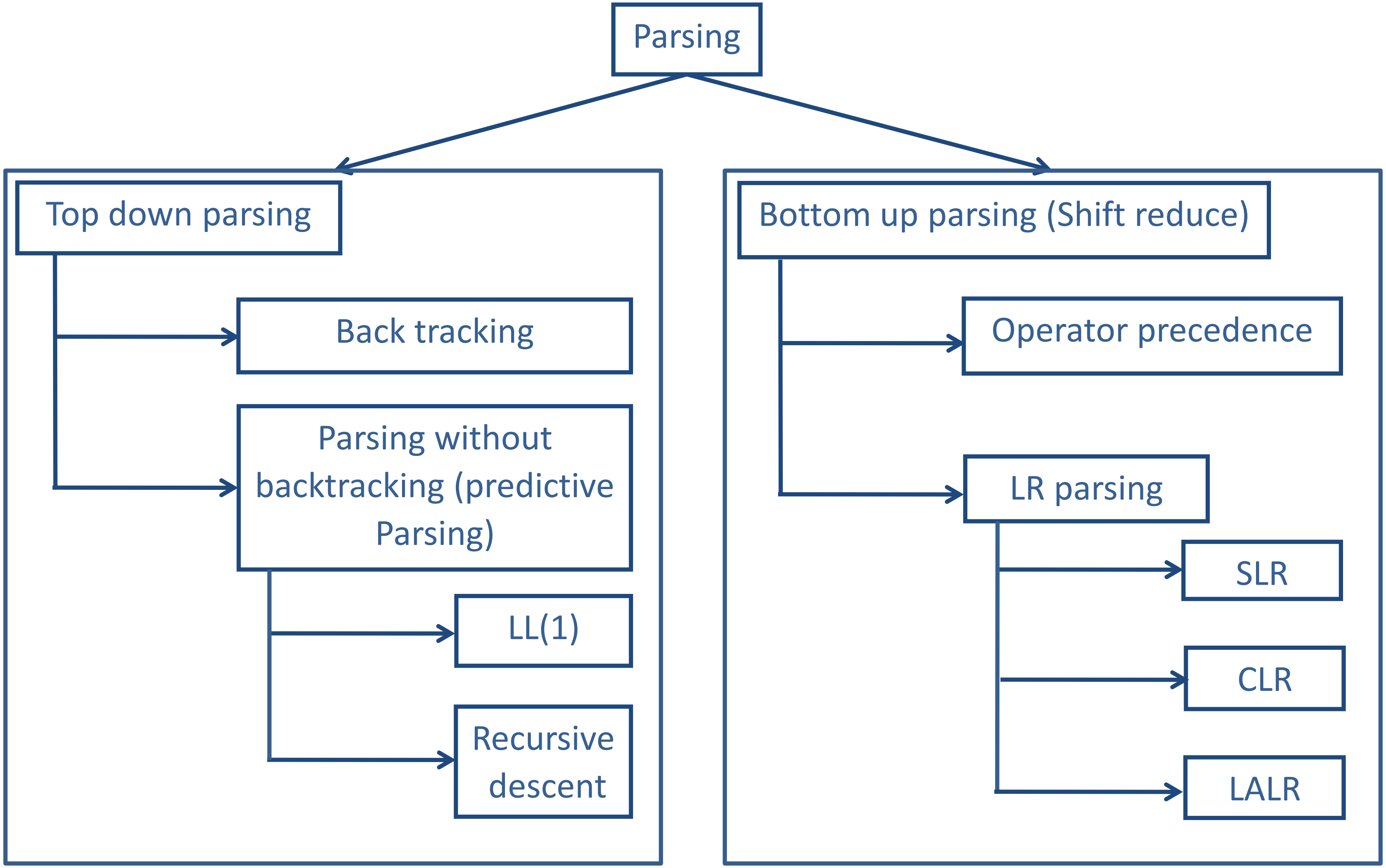
Mrs. Ruchi Sharma

ruchi.sharma@bkbiet.ac.in

Topics to be covered

- LR parsing
- LR(0)
- SLR(1)
- CLR
- LALR

Classification of parsing methods



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
E→id

String: id1+id2*id3

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		

HANDLES

A handle of a string is a substring that matches the right side of a production and whose reduction to the non terminal on the left side of the production represents one step along the reverse of a rightmost derivation.

Is leftmost substring always handle?

↳ No, choosing the leftmost substr. as the handle ALWAYS, may not give correct SR Parsing.

A handle of a right sentential form γ is a production $A \to \beta$ and a position of γ where the string β may be found and replaced by A to produce the previous right sentential form in a rightmost derivati

$S \xrightarrow[rm]{} aABe \xrightarrow[rm]{} aAde \xrightarrow[rm]{} aAbcde \xrightarrow[rm]{}$

abbcde : γ = abbcde, $A \to b$, Handle = b.

aAbcde γ = RHS = aAbcde, $A \to Abc$
Handle : Abc.

aAde : γ = aAde, $B \to d$, Handle = d

aABe : γ = aABe, Handle = aABe

S

$S \to aABe$

$A \to Abc \mid b$ ✓

$B \to d$. ✓

PRUNING THE HANDLE

Removing the children of Left Hand Side non terminal from the Parse Tree is called Handle Pruning.

A rightmost derivation in Reverse can be obtained by Handle Pruning.

Steps To Follow :

- Start with a string of terminals 'w' that is to be parsed.
- Let $w = \gamma_n$, where γ_n is the n^{th} right sentential form of an unknown RMD.
- To reconstruct the RMD in reverse, locate handle β_n in γ_n; Replace β_n by LHS of some $A_n \rightarrow \beta_n$ to get $(n-1)^{th}$ RSF γ_{n-1}. Repeat.

Right Sentential Form	Handle	Red… Pr…
$id_1 + id_2 * id_3$	id_1	$E \rightarrow id$
$E + id_2 * id_3$	id_2	$E \rightarrow id$
$E + E * id_3$	id_3	$E \rightarrow id$
$E + E * E$	$E + E$	$E \rightarrow E + E$
$E * E$	$E * E$	$E \rightarrow E * E$
E		

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

$$S \xRightarrow[rm]{} \gamma_0 \xRightarrow[rm]{} \gamma_1 \xRightarrow[rm]{} \gamma_2 \cdots\cdots \xRightarrow[rm]{} \gamma_{n-1} \xRightarrow[rm]{} \underset{\Uparrow}{\gamma_n}$$

w

Exercise

S→aABe

A→Abc | b

B→d

String: abbcde

BOTTOM - UP PARSING

It is the process of reducing the Input string to start symbol i.e the Parse Tree is constructed in from leaves to the root (bottom to top)

It is also known as Shift Reduce Parsing. Also called $\boxed{\text{LR Parser}}$

{Left To Right Scanning of the input}

{Rightmost Derivation in reverse order}

E → E + E |
E * E |
id

E + E

E * E + E

E * E + id

E * id + id

id * id + id

E → id

E

E + E

At each reduction step a parti substring matching the right production is replaced by the symbol on the left of that production and if the substring is chosen correctly at each step a rightmost derivation is traced in reverse

S → aABe
A → Abc | b
B → d

abbcde
aAbcde
aAde
aABe
S

b, d

Abc, b, d

d

aABe

abbcde
abbcBe
aAcBe
X

abbcde
aAAcde
aAAcBe

Shift reduce parser- as Bottom up Parser

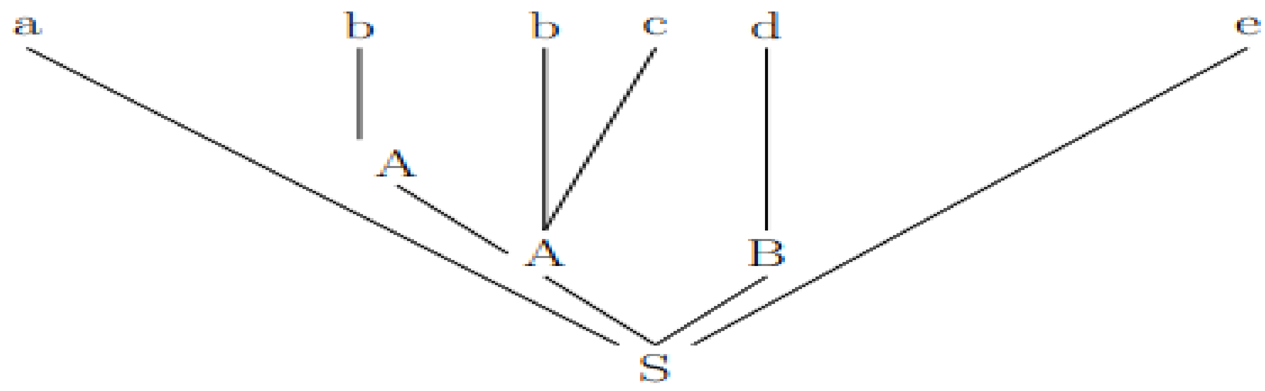
Shift-reduce parsing

Shift-reduce parsing attempts to construct a parse tree for an input string beginning at the leaves (the bottom) and working up towards the root (the top)

Example: Consider the following grammar:

$$S \longrightarrow aABe$$
$$A \longrightarrow Abc \mid b$$
$$B \longrightarrow d$$

the parse tree for the input string abbcde can be formed (bottom-up) as follows:



this is similar to rightmost derivation but in reverse order:

$$abbcde \Leftarrow aAbcde \Leftarrow aAde \Leftarrow aABe \Leftarrow S$$

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.

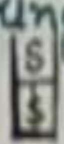
PERFORMING SHIFT - REDUCE PARSING USING A STACK

Major Data structures used by Shift Reduce Parsing are:

1. STACK : It is used to hold grammar symbols

2. INPUT BUFFER : Holds the input string that needs to be parsed.

$ →



IP Buffer : 'w'$

↳ abc

Major Actions Performed are:

1. SHIFT : PUSHING
   ↳ Pushing the next input symbol on the top of the stack.

2. REDUCE : POPPING
   ↳ Popping the handle whose right end is at T.O.S And Replacing it with left side Nonterminal

3. ACCEPT :
   ↳ Denotes successful completion of Parsing process

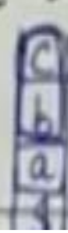
4. ERROR :
   ↳ Syntax error generation.

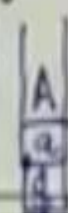
Stack Implementation of SR Parser

Shift input symbols onto stack untill a handle β is on Top of stack.

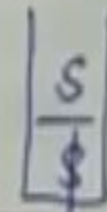
Reduce β to left side Non Terminal of appropriate production.

Repeat untill error OR stack has the start symbol left and input is empty.





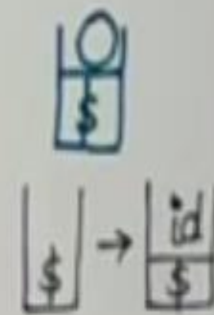
S → aA
A → bc



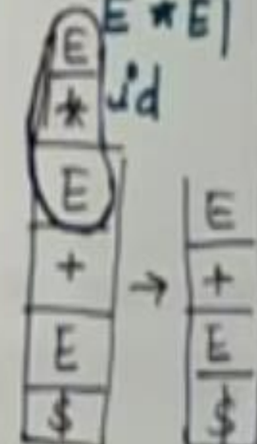
PERFORM SHIFT-REDUCE PARSING USING A STACK (Part 2)

STACK CONTENT	INPUT	ACTION
$	$id_1 + id_2 * id_3$ $	Shift
$ id_1	$+ id_2 * id_3$ $	Reduce by $E \rightarrow id$
$ E	$+ id_2 * id_3$ $	Shift
$ E +	$id_2 * id_3$ $	Shift
$ $E + id_2$	$* id_3$ $	Reduce by $E \rightarrow id$
$ E + E	$* id_3$ $	Shift
$ E + E *	id_3 $	Shift
$ $E + E * id_3$	$	Reduce by $E \rightarrow id$
$ $E + \underbrace{E * E}$	$	Reduce by $E \rightarrow E * E$
$ E + E	$	Reduce by $E \rightarrow E + E$
$ E	$	Accept

Start Symbol



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



Example: Shift reduce parser

Grammar:

E→E+T | T

T→T*F | F

F→id

String: id+id*id

Stack	Input Buffer	Action

STACK	INPUT	ACTION
\$	x*x \$	Shift
\$x	*x \$	Reduce
\$F	*x \$	Reduce
\$T	*x \$	Shift
\$T*	x \$	Shift
\$T*x	\$	Reduce
\$ $\underbrace{T*F}$	\$	Reduce
\$T	\$	Reduce
\$E	\$	Accept.

STACK	INPUT	ACTION
\$	x*x \$	
\$x	*x \$	Reduce
\$F	*x \$	Reduce
\$T	*x \$	Reduce
\$E	*x \$	Shift
\$E*	x \$	Shift
\$E*x	\$	Reduce
\$E*F	\$	Reduce
\$E*T	\$	Reduce
\$E*E	\$	ERROR

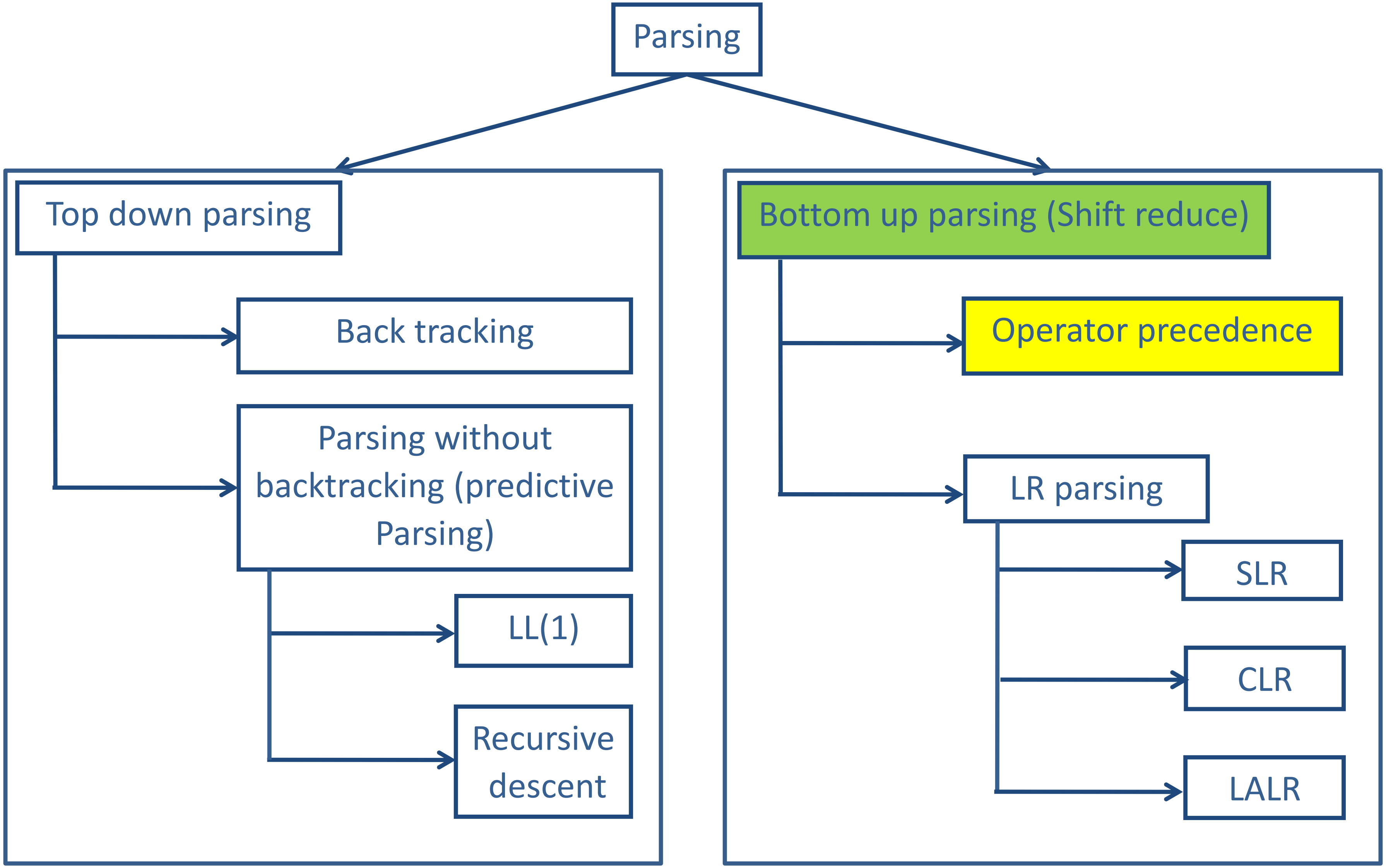
$$\left.\begin{array}{l} E \rightarrow E+T \mid T \\ T \rightarrow T*F \mid F \\ F \rightarrow (E) \mid x \end{array}\right\}$$

\$ T*F	R
\$ $\underbrace{T*T}$	Error

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 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

OPERATOR GRAMMAR

Shift Reduce Parsers can be built successfully using/for 2 Main classes of grammar

- LR Grammar
- Operator Grammar

Properties of Operator Grammar

1. No production in the grammar has ϵ on the right hand side.

2. No 2 Non Terminals appear together on RHS of any production.

eg $E \rightarrow EAE \mid (E) \mid -E \mid$ ...

$(A \rightarrow + \mid * \mid - \mid / \mid \uparrow)$

Violates Rule 2 → Not O.P.G.

⇓

$E \rightarrow E+E \mid E*E \mid E-E \mid E/E \mid E\uparrow E \mid$
$(E) \mid -E \mid id$ ⟶ OPG.

$A \rightarrow \underbrace{\ldots\ldots}$ ✗

eg $S \rightarrow SAS \mid a$
$A \rightarrow bSb \mid b$ } Not in form of OPG.

⇓

$S \rightarrow S\underline{bSb}S \mid a \mid S\underline{b}S$ } Operator Precedence Grammar

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		

OPERATOR PRECEDENCE PARSING

We define 3 precedence relation operations between pairs of terminals.

↳ $\lessdot$, $\gtrdot$, $\doteq$

USE ? They help us in selection of handles.

MEANING ? $a \lessdot b$: 'a' has lower precedence than 'b'

$a \gtrdot b$: a has higher precedence than 'b'

$a \doteq b$: a has equal precedence as 'b'

HOW TO ASSIGN RELATIONS ?

↳ Using Associativity and Precedence

↳ For all terminals (including \$) we design an operator prec. table

OPERATOR PRECEDENCE TABLE

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

$(+, *, id), \$$

$id \gtrdot +$, $\$ \lessdot id$

$id \gtrdot *$, $\$ \lessdot +$

$id \gtrdot \$$, $\$ \lessdot *$

	id	+	*	\$
id		$\gtrdot$	$\gtrdot$	$\gtrdot$
+	$\boxed{\lessdot}$	$\gtrdot$	$\lessdot$	$\gtrdot$
*	$\lessdot$	$\gtrdot$	$\gtrdot$	$\gtrdot$
\$	$\lessdot$	$\lessdot$	$\lessdot$	ACC

$(id, +)$ $(+, id)$ $(* \gtrdot +)$

$(+, +)$ $(1+2)+3$

Symbol on T.O.S

Symbol I/P str

* If (I/P str sym) $\gtrdot$ T.OS : PUSH I/P symb.

else (I/P str sym) $\lessdot$ T.O.S : POP and reduce T.OS

else ERROR

Rules to establish a relation

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

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

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

$$NT \cdot Op \quad | \quad (Trailing(NT)) \cdot > Op$$

$$E + \quad | \quad \{+,*,id\} \cdot > \ +$$

$$T * \quad | \quad \{*,id\} \cdot > *$$

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→ E+ T| T
T→ T* F| F
F→ id

Step 2: Establish Relation

$<· Leading (start symbol)

$ <· {+,∗, id}

Trailing (start symbol) ·> $

{+,∗, id} ·> $

Step 3: Creation of Table

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

SOLVED EXAMPLE

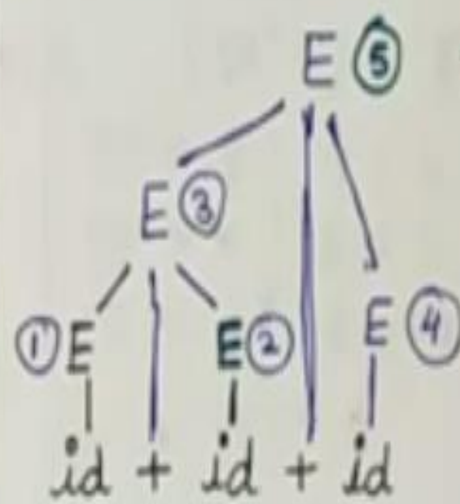
Input String	Stack	
id + id + id $	$	
+ id + id $	$ id	1
+ id + id $	$	
id + id $	$ +	
+ id $	$ + id	2
+ id $	$ +	3
+ id $	$	
id $	$ +	
$	$ + id	4
$	$ +	
$	$	5

E → E + E | E * E | id

I.S > T.O.S : PUSH

I.S < T.O.S : POP & REDUCE

	id	+	*	$
id		>	>	>
+	<	>	<	>
*	<	>	>	>
$	<	<	<	ACC



(T.O.S , I.S)

Successful

Operator Precedence Parser is the only parser that is capable of handling Ambiguous Gramr.

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

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

Operator precedence function

Algorithm for constructing precedence functions

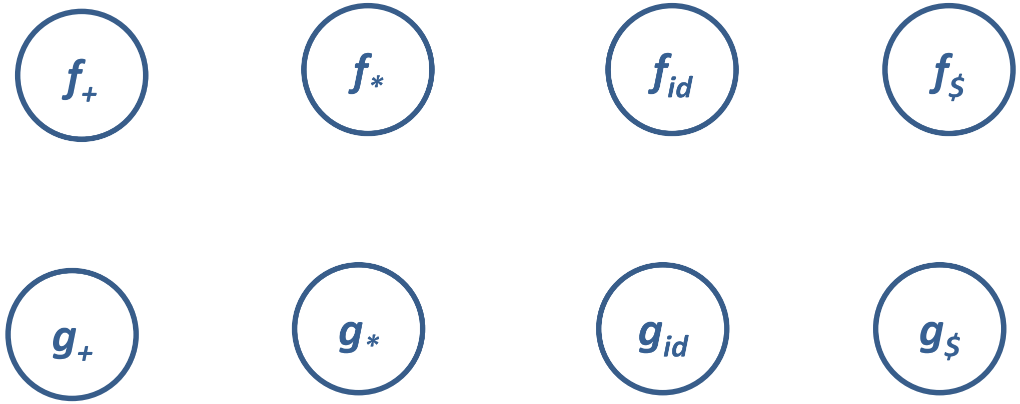
1. Create functions f_a and g_a for each a that is terminal or \$.
2. Partition the symbols in as many as groups possible, in such a way that f_a and g_b are in the same group if $a \doteq b$.
3. Create a directed graph whose nodes are in the groups, next for each symbols $a\ and\ b$ do:
   a) if $a <\cdot b$, 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 $.*

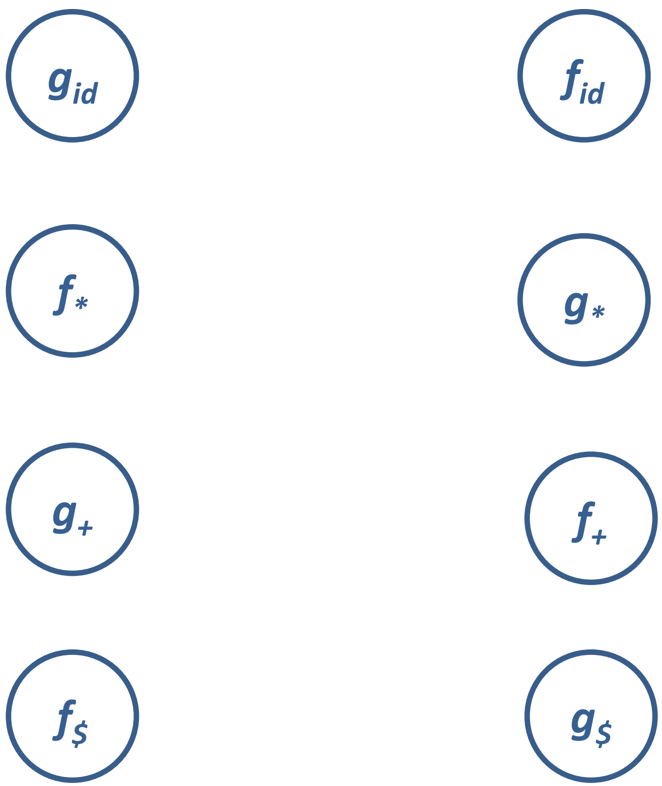
$$a = \{+, *, id\} \ or \ \$$$

E→ E+T | T
T→ T*F | F
F→ id



Operator precedence function

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 $a \doteq b$.*

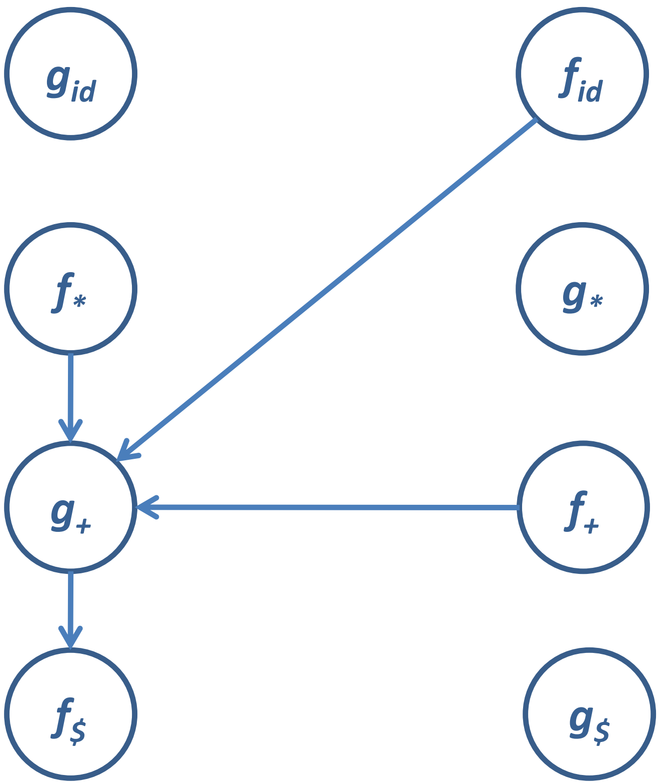


	+	*	id	$
+	·>	<·	<·	·>
*	·>	·>	<·	·>
id	·>	·>		·>
$	<·	<·	<·	

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



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

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

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

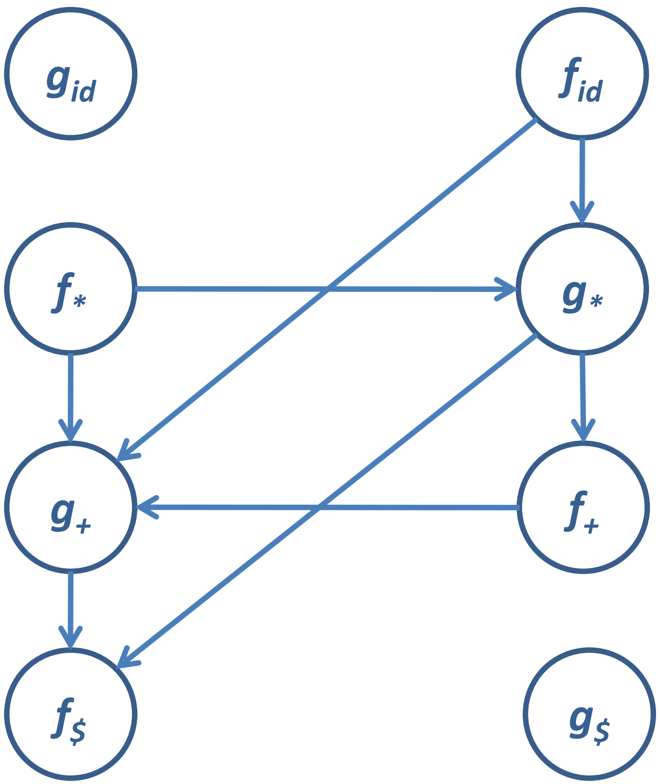
$f_{id} \cdot> g_+$ $\quad$ $f_{id} \rightarrow g_+$

$f_\$ <\cdot\, g_+$ $\quad$ $f_\$ \leftarrow g_+$

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	·>	·>		·>
	$	<·	<·	<·	

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

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

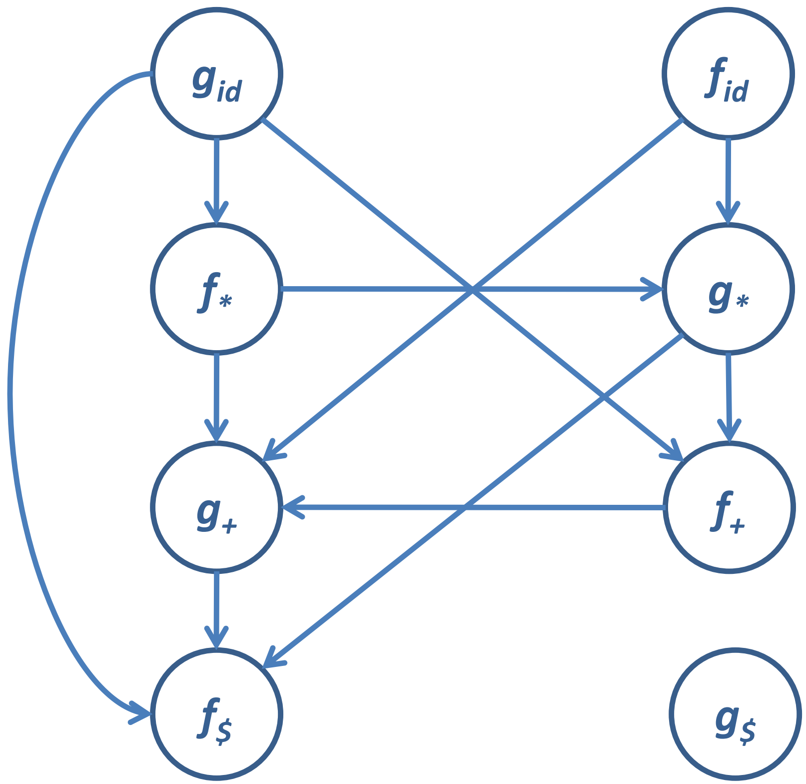
$f_{id} \cdot> g_*$ $\qquad f_{id} \rightarrow g_*$

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

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



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

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

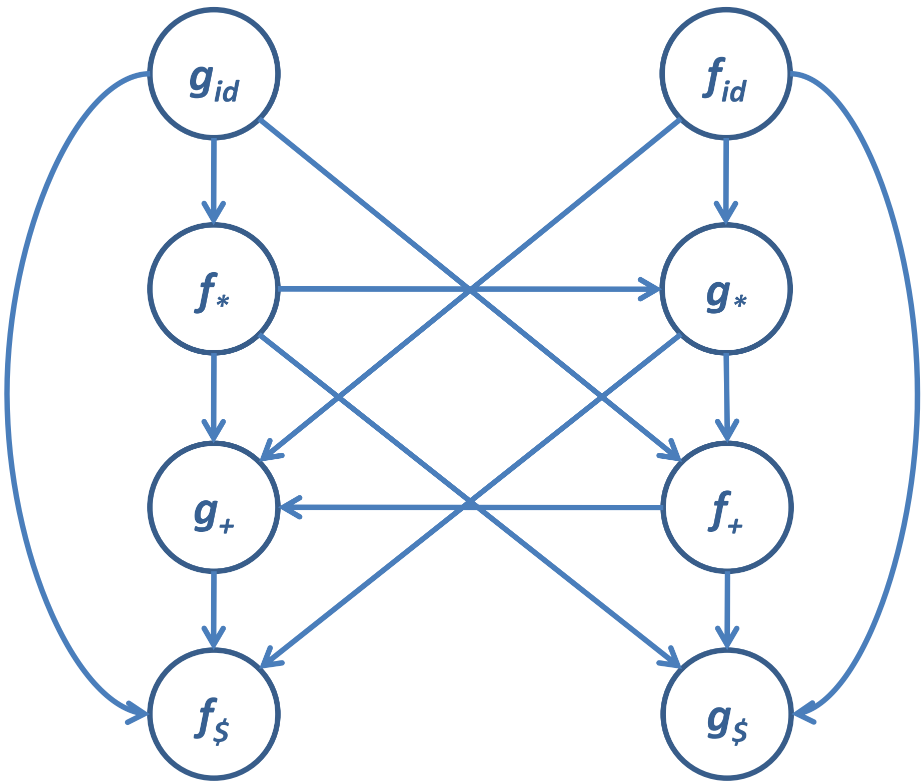
$f_* <\cdot g_{id}$ $\qquad f_* \leftarrow g_{id}$

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

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



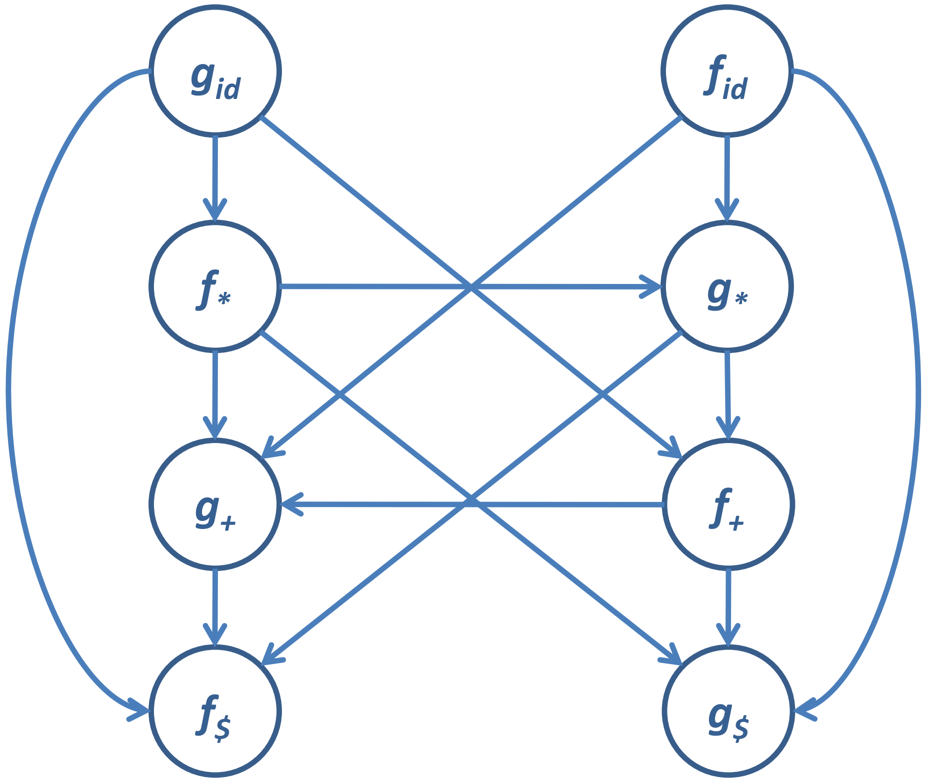
f \ g	+	*	id	$
+	·>	<·	<·	·>
*	·>	·>	<·	·>
id	·>	·>		·>
$	<·	<·	<·	

$f_+ <\cdot\ g_\$$ $\quad f_+ \rightarrow g_\$$

$f_* <\cdot\ g_\$$ $\quad f_* \rightarrow g_\$$

$f_{id} <\cdot\ g_\$$ $\quad f_{id} \rightarrow g_\$$

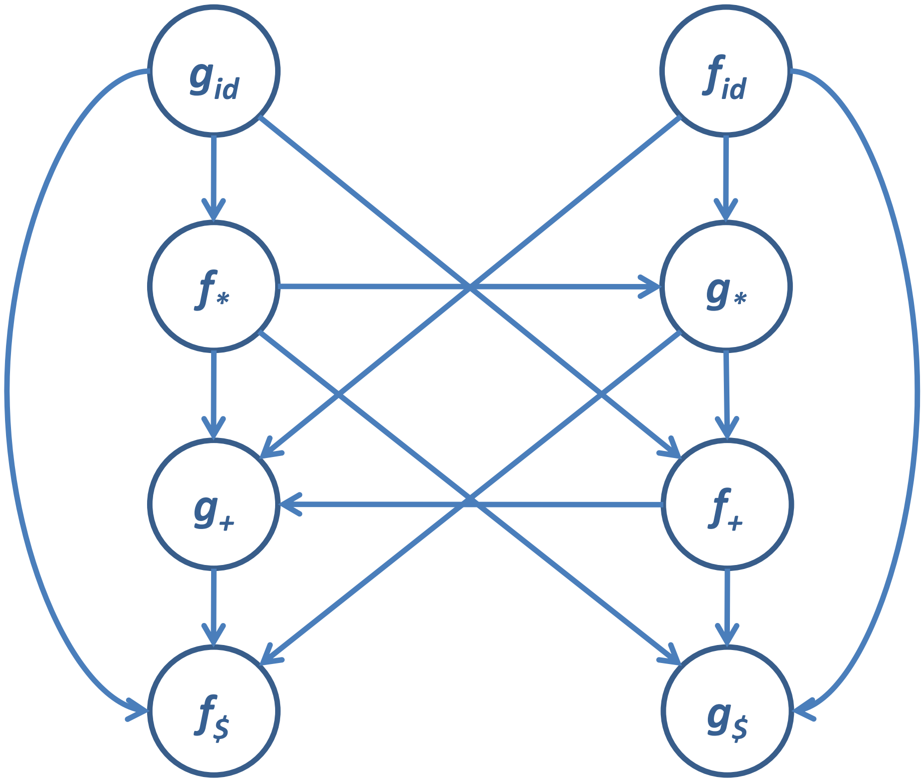
Operator precedence function



	+	*	id	$
f				
g				

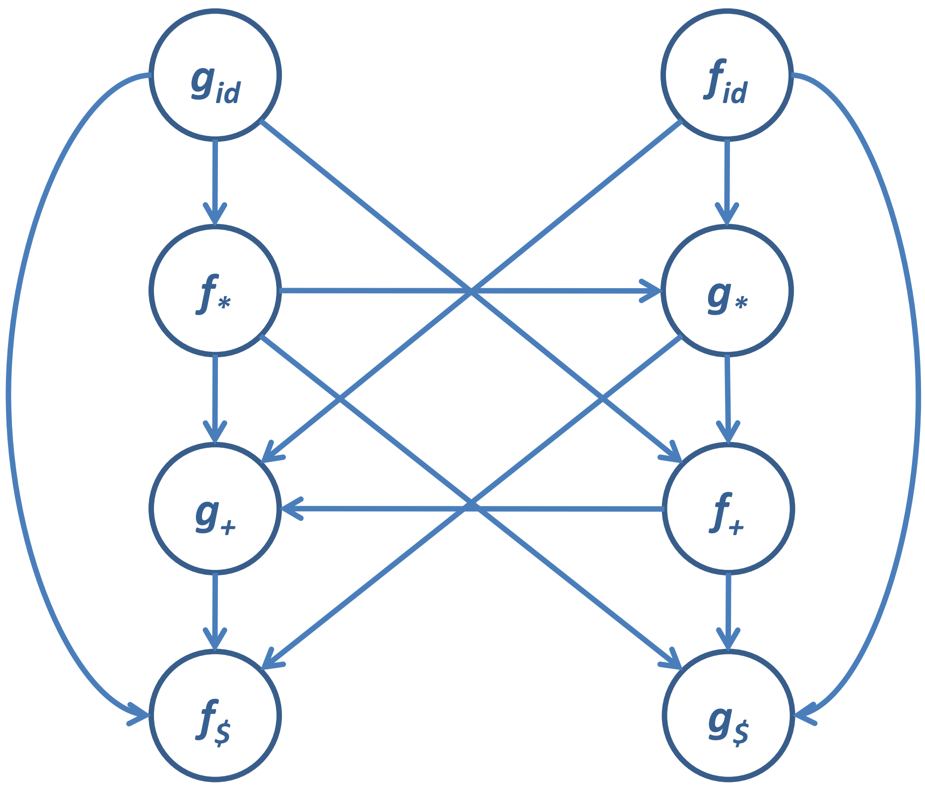
4. *If the constructed graph has a cycle then no precedence functions exist. When there are no cycles collect the length of the longest paths from the groups of f_a and g_b respectively.*

Operator precedence function



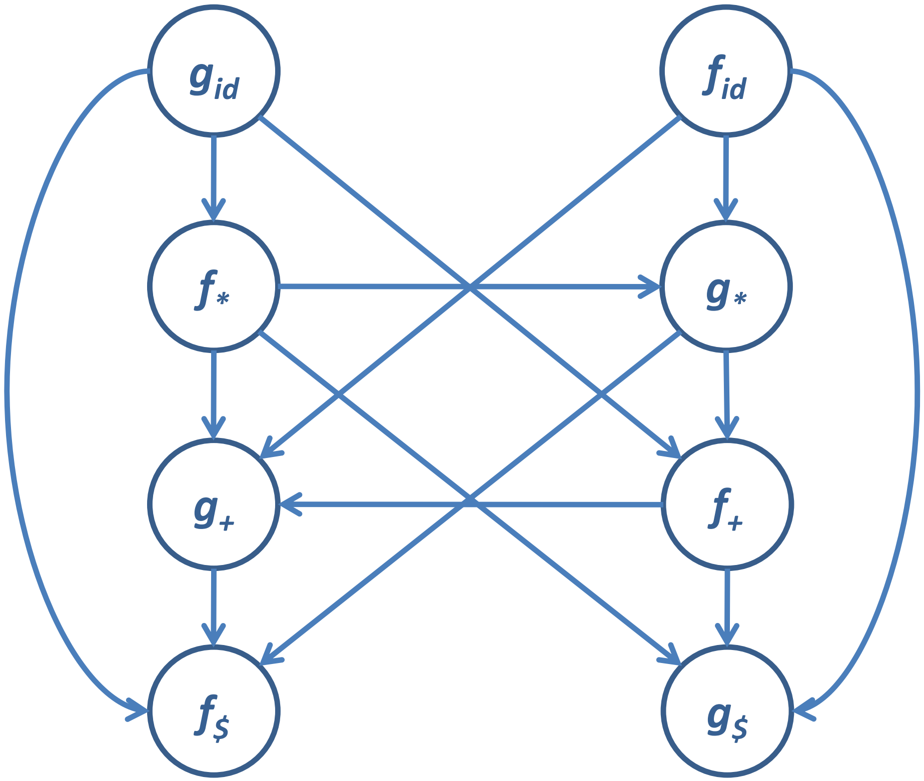
	+	*	id	$
f	2			
g				

Operator precedence function



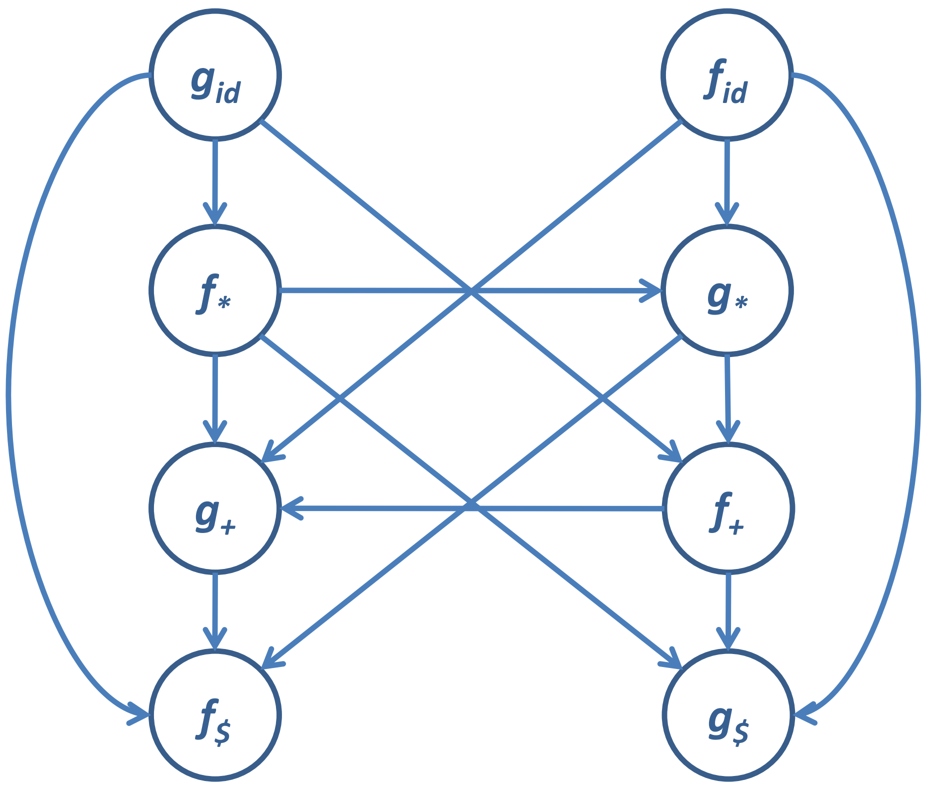
	+	*	id	$
f	2			
g	1			

Operator precedence function



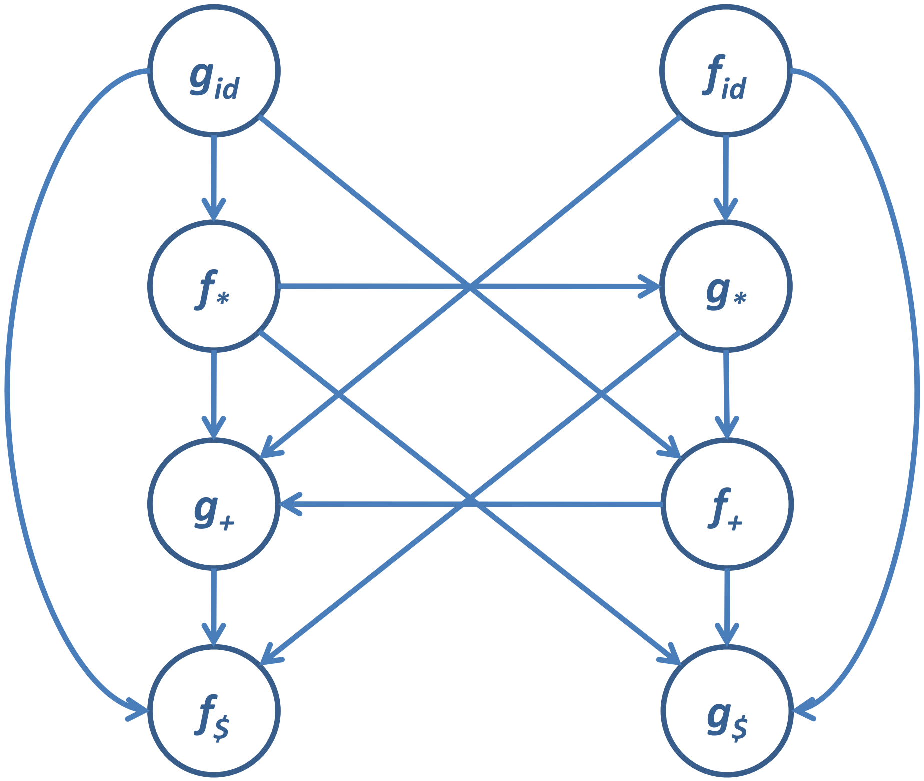
	+	*	id	$
f	2	4		
g	1			

Operator precedence function



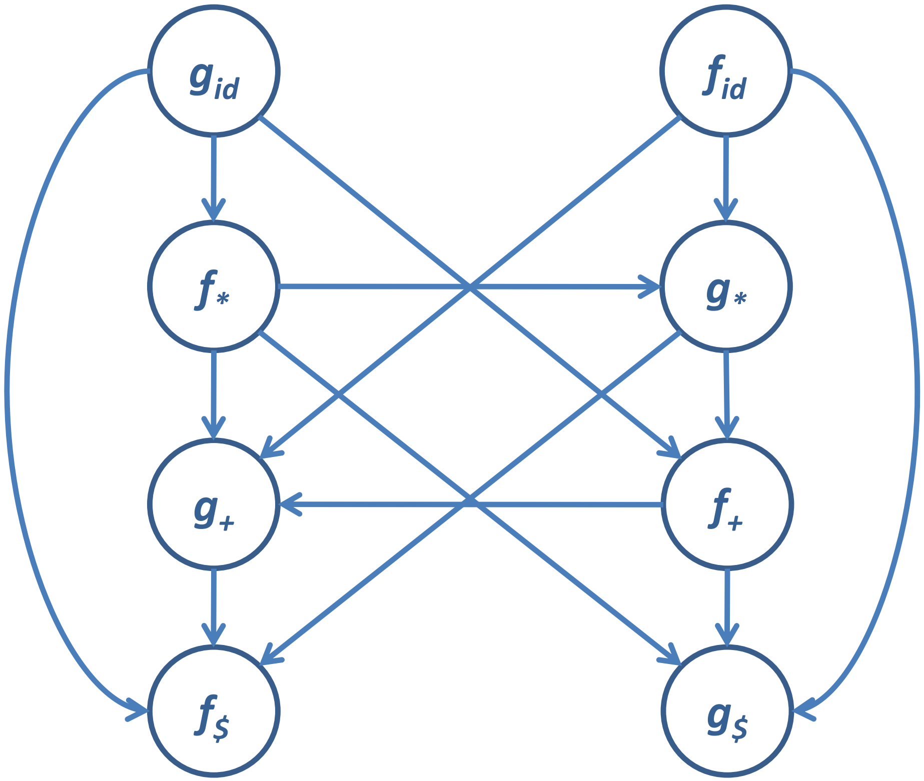
	+	*	id	$
f	2	4		
g	1	3		

Operator precedence function



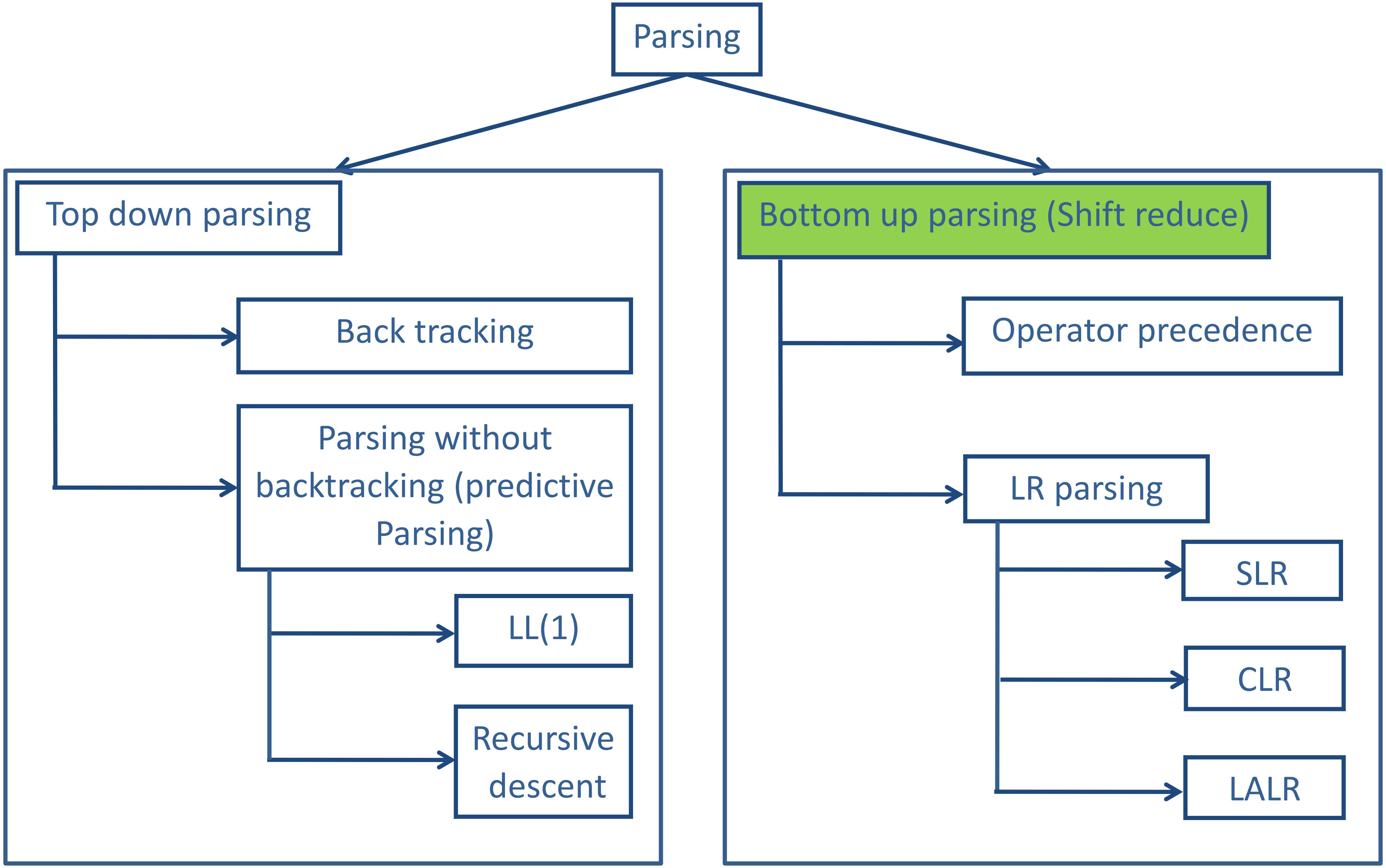
	+	*	id	$
f	2	4	4	
g	1	3		

Operator precedence function



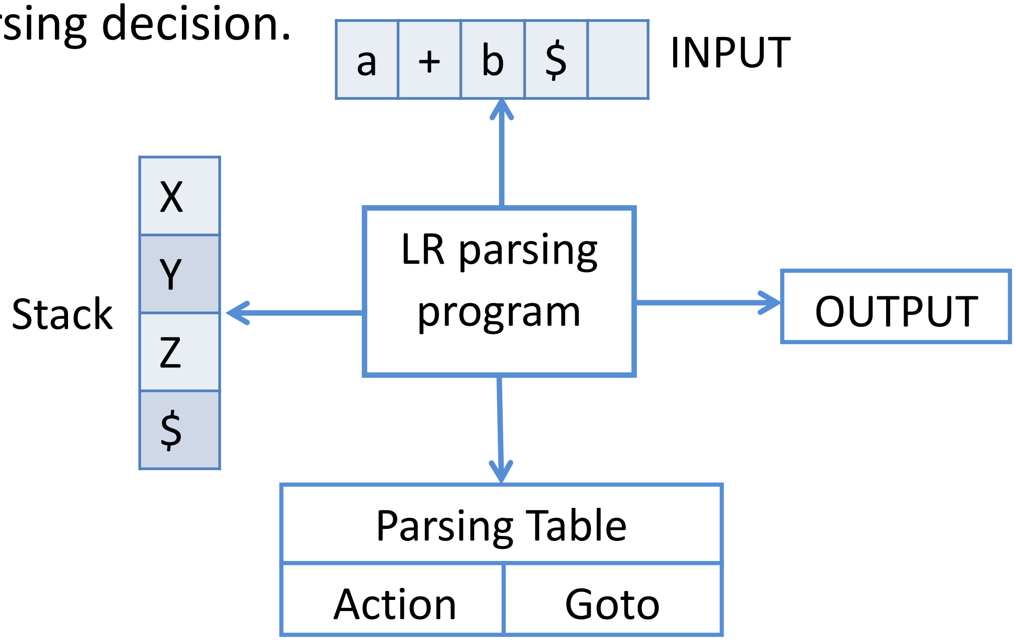
	+	*	id	$
f	2	4	4	
g	1	3	5	

Parsing methods



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.



LR parser

LR(K) PARSING

LR(0)
LR(1)
LR

What? Bottom-Up Technique to perform syntax analysis.

L : left to right scanning of input

R : Reverse of Rightmost Derivation

K : No. of lookahead symbols that are used to make parsing decisions

★ When K's value is skipped, it is assumed to be 1.

Benefits of using LR(K) Parsing

1. Most generic Non-Backtracking Shift Reduce Parsing Technique.
2. These parsers can recognise all programming languages for which context-free grammars can be written
3. They are capable of detecting syntactic error as soon as possible in scanning of input.

Types Of LR Parsers

(I) SIMPLE LR Parser (SLR)

- easiest to implement
- least powerful
- May fail to work on some grammars accepted by CLR, LALR

(II) CANONICAL LR Parser (CLR)

- Most powerful LR Parser
- Most expensive

(III) LOOKAHEAD LR Parser (LALR).

- Intermediate to SLR and CLR in terms of power and cost.

LR parser

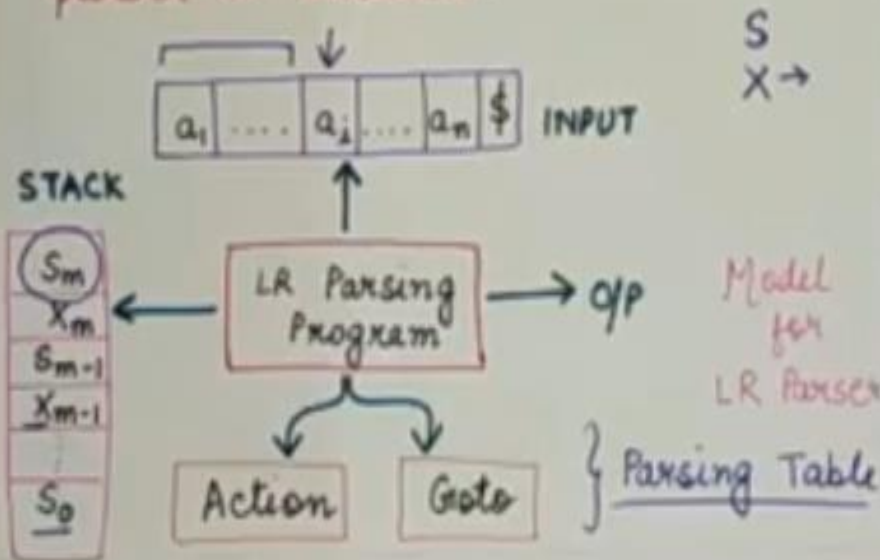
COMPONENTS OF LR PARSER

Major Components of LR Parser are:

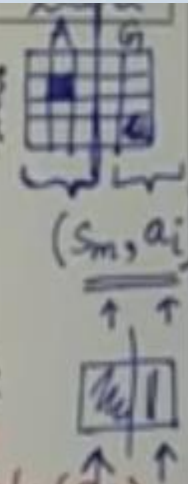
1. Input Buffer
2. Stack
3. Parsing Algo
4. Parsing Table

Parsing Algo: also called the driver program, remains same for all LR Parsers

Parsing Table: parsing table changes from 1 LR parser to another.



Behaviour of LR Parser



* Parsing algorithm reads the next unread I/P char from the Input Buffer.
* Parsing algo also reads the character on the Top of Stack

  A stack can have grammar symbols (X_i) or state symbols (S_i)
* Combination of I/P char and T.O.S char is used to index Parsing Table

Parsing Actions can Be

(1) Shift (2) Reduce (3) Error (4) Accept

Goto function takes a state and a grammar symbol and produces a state.

LR parser

Parsing Action Performed By LR-Parser

CONFIGURATION OF LR PARSER : (Contents of Stack , The current input string)

$(s_0 X_1 s_1 X_2 s_2 \ldots X_m s_m)$, $(a_i a_{i+1} \ldots a_n \$)$

Term | N.T

To determine next configuration of LR Parser, consult $[s_m, a_i]$ entry of Parse Table

Case 1 If action $[s_m, a_i]$ = shift s, a shift move is executed and the new configuration becomes

$(s_0 X_1 s_1 X_2 s_2 \ldots X_m s_m a_i s \,,\, a_{i+1} \ldots a_n \$)$

Case 2 : If action $[s_m, a_i]$ = reduce $A \rightarrow \beta$, then

- Parser pops $2r$ symbols from the stack (r is length of β)
- State s_{m-r} appears on T.O.S
- Parser pushes A & goto $[s_{m-r}, A] = s$

$(s_0 X_1 s_1 X_2 s_2 \ldots X_{m-r} \boxed{s_{m-r} A} s \,,\, a_i a_{i+1} \ldots a_n \$)$

Case 3 : If action $[s_m, a_i]$ = accept then Parsing is successfully complete.

Case 4 : If action $[s_m, a_i]$ = error it denotes an error has been found by Parser. Error handling function is called.

LR parser

LR(0) ITEMS

LR(0) item of a grammar G is a production of G with a dot at some position of the right side.

$A \to XYZ$

$A \to .XYZ$ → leftmost position in RHS

$A \to X.YZ$
$A \to XY.Z$
$A \to XYZ.$

} LR(0) items.

$A \to .w$, $A \to w.$

$A \to .\epsilon.$ ⇒ $A \to .$

What does an item indicate ?

It indicates, how much part of a production we have seen at a given point in parsing process.

AUGMENTED GRAMMAR

$S' \to S$ — named anything

If G is a grammar with start symbol S, then G′ (the augmented grammar) contains the production from G along with a new prod^n $S' \to S$ where S′ is new start symbol

1 $A \to .XY$ $\xrightarrow{X}$ 2 $A \to X.Y$ $\xrightarrow{Y}$ 3 $A \to XY.$

Why required ?

It indicates that parser should stop parsing and announce acceptance when it is about to reduce $S' \to S$.

KERNEL ITEMS

$S' \to .S$ & {all items with dots NOT at the left end}

NON-KERNEL ITEMS

↳ All those LR(0) items which have dot at the leftmost position (except augmented prod^n : $S' \to .S$)

LR parser

Closure Function

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

$\Rightarrow$

$[E' \to E]$
$E \to E + T \mid T$
$T \to T * F \mid F$
$F \to (E) \mid id$.

$\Downarrow$

Construct a set I of all LR(0) items for the given grammar

$\Leftarrow$

CLOSURE (I)

$E' \to .E$
$E \to .E + T$
$E \to .T$
$T \to .T * F$
$T \to .F$
$F \to .(E)$
$F \to .id$

closure

closure

closure

Items added by calculating Closure, will never have Kernel Items

We compute CLOSURE whenever there is a dot to the immediate left of a Non-Terminal and the NT has not yet been expanded. Expansion of such NT into items with dot at extreme left is called Closure.

STEPS

- Construct the Augmented Grammar
- Construct set I of LR(0) items of Augmented Grammar.
- For each item that has dot to the immediate left of a non-terminal expand the set I by including items formed from this NT; including only those items at with dot at extreme left.
- Repeat untill new items are added

LR parser

Goto Operation

I_0:

$E' \to \cdot E$ ①
$E \to .E+T$ ②
$E \to .T$ ③
$T \to .T*F$ ④
$T \to .F$ ⑤
$F \to .(E)$ ⑥
$F \to .id$ ⑦

① $\xrightarrow{c}$ ②, ③ $\Downarrow c$ ④, ⑤ $\Downarrow c$ ⑥, ⑦

Goto → I_i, X
X → Terminal, Non Term

Goto(I_0, E) : I_1:
$E' \to E.$ ✓
$E \to E.+T$
No further closure.

Goto(I_0, T) : I_2:
$E \to T.$
$T \to T.*F$
No further closure

Goto(I_0, F) I_3: $T \to F.$
No more closure

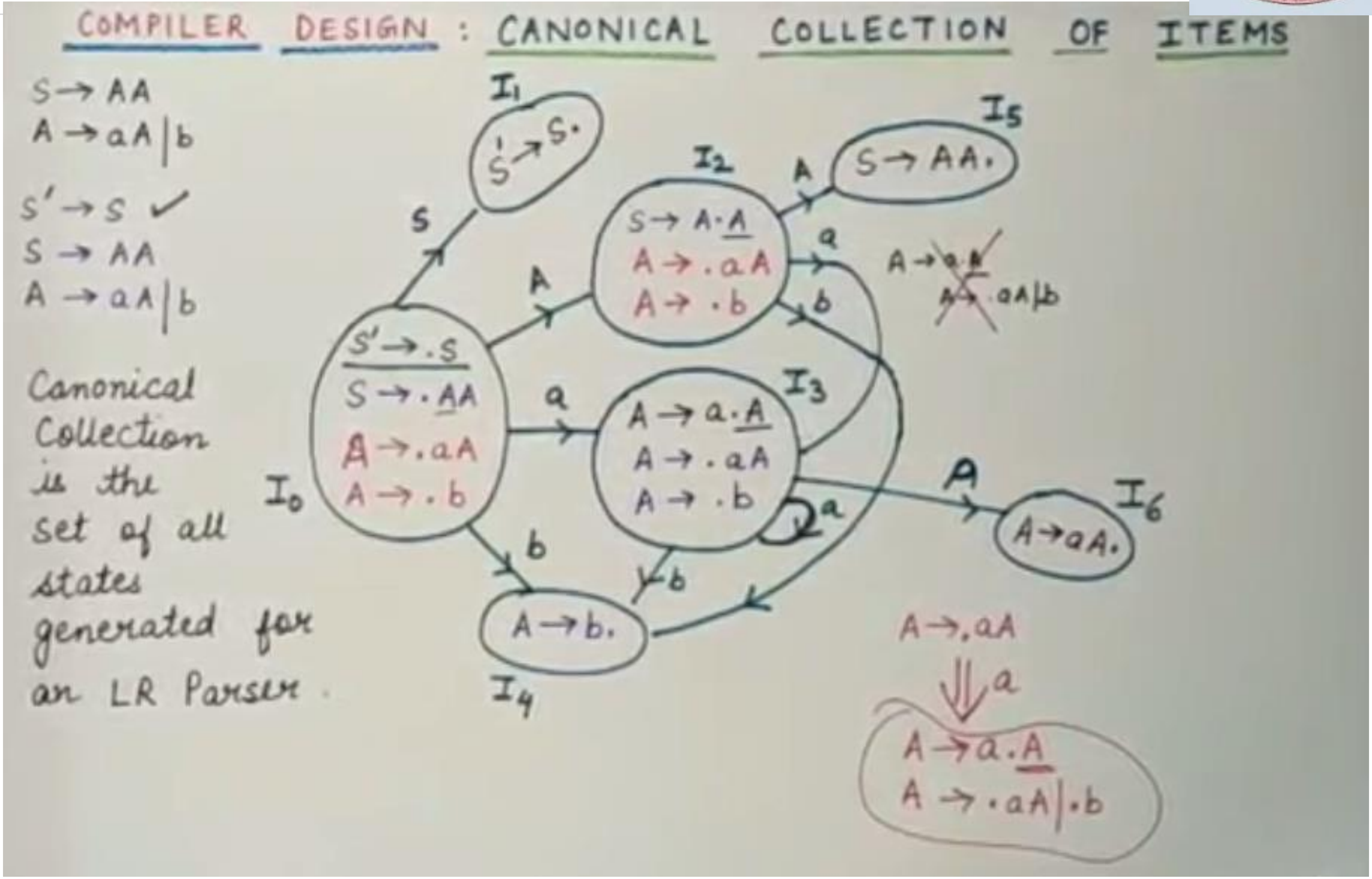
Goto$(I_0, ($ $)$: I_4:
$F \to (.E)$
$E \to .E+T$
$E \to .T$
$T \to .F$
$T \to .T*F$
$F \to .(E)$
$F \to .id$

In Goto Fn, we shift the dot one step to the right. After shifting we again check for closure.

Goto(I_0, id) : I_5: $F \to id.$

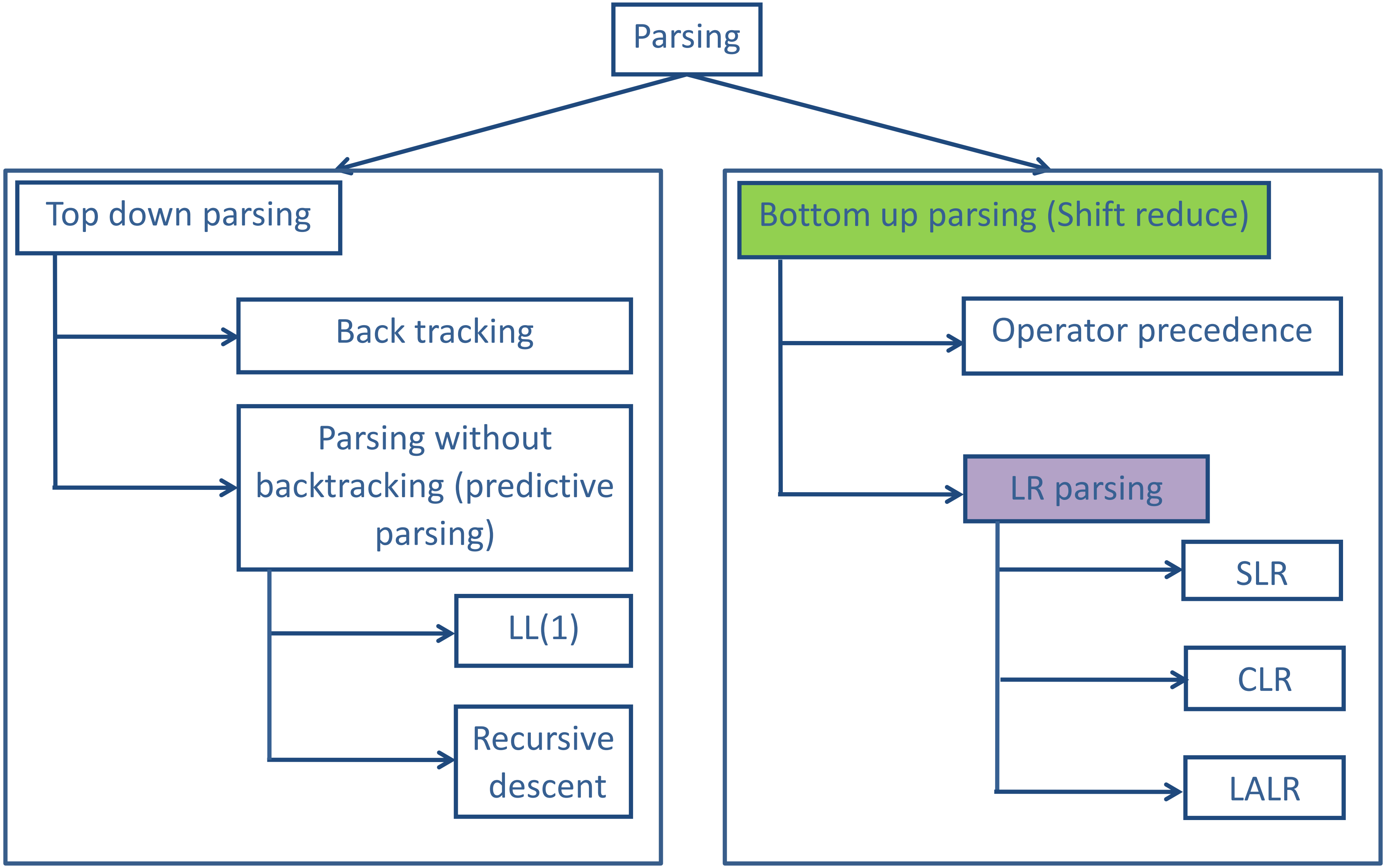
Goto$(I_1, +)$ I_6:
$E \to E+.T$
$T \to .T*F$
$T \to .F$
$F \to .id$
$F \to .(E)$

LR Parser



COMPILER DESIGN : CANONICAL COLLECTION OF ITEMS
S → AA
A → aA | b
S' → S
S → AA
A → aA | b
Canonical Collection is the set of all states generated for an LR Parser.
I0
S' → .S
S → .AA
A → .aA
A → .b
I1
S' → S.
I2
S → A.A
A → .aA
A → .b
I3
A → a.A
A → .aA
A → .b
I4
A → b.
I5
S → AA.
I6
A → aA.
A → .aA
a
A → a.A
A → .aA | .b

Parsing methods



Computation of closure & go to function

$X \rightarrow Xb$

Closure(I):

$X \rightarrow .\,X\,b$

Goto(I,X)

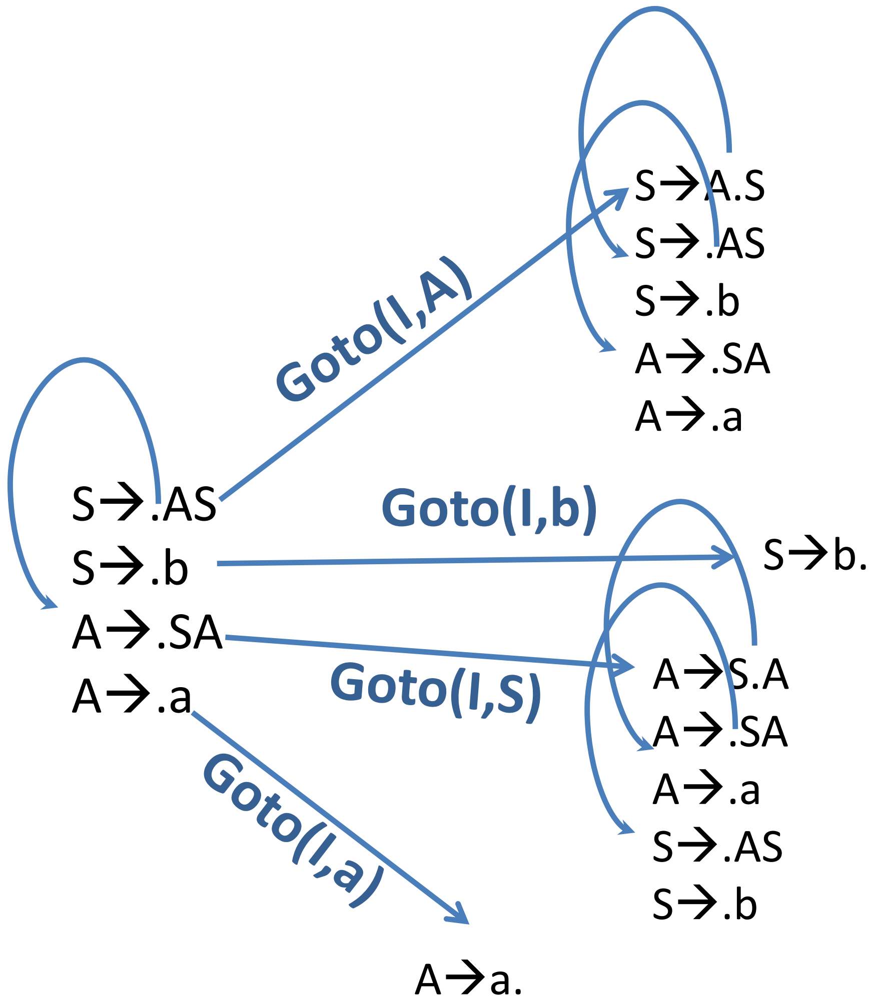
$X \rightarrow .\,X\,b$

Computation of closure & goto function

S→AS | b

A→SA | a

Closure(I):



Exercise

$S \rightarrow Aa \mid bAc \mid Bc \mid bBc$

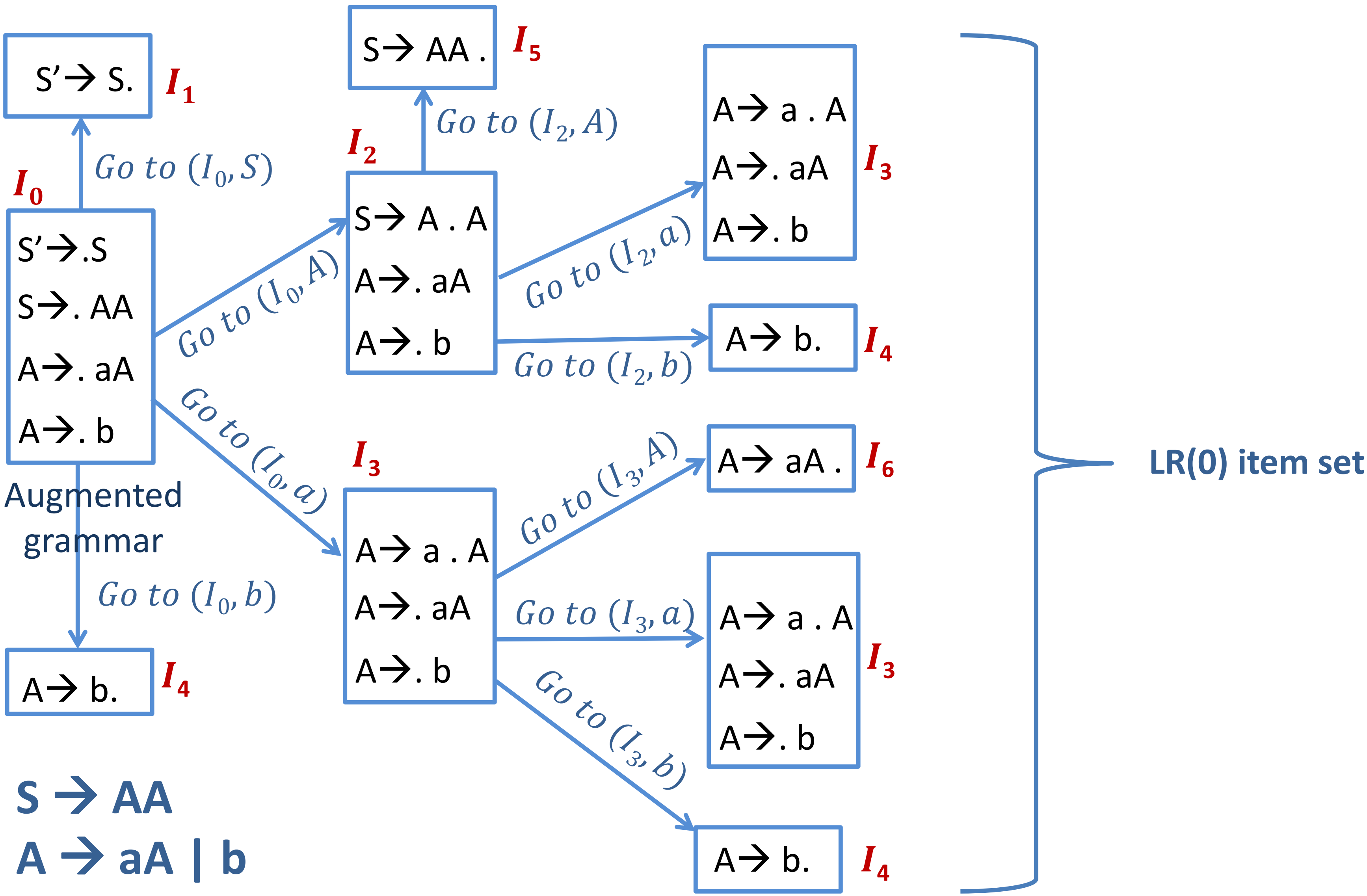
$A \rightarrow d$

$B \rightarrow d$

Steps to construct SLR parser

1. Construct Canonical set of LR(0) items
2. Construct SLR parsing table
3. Parse the input string

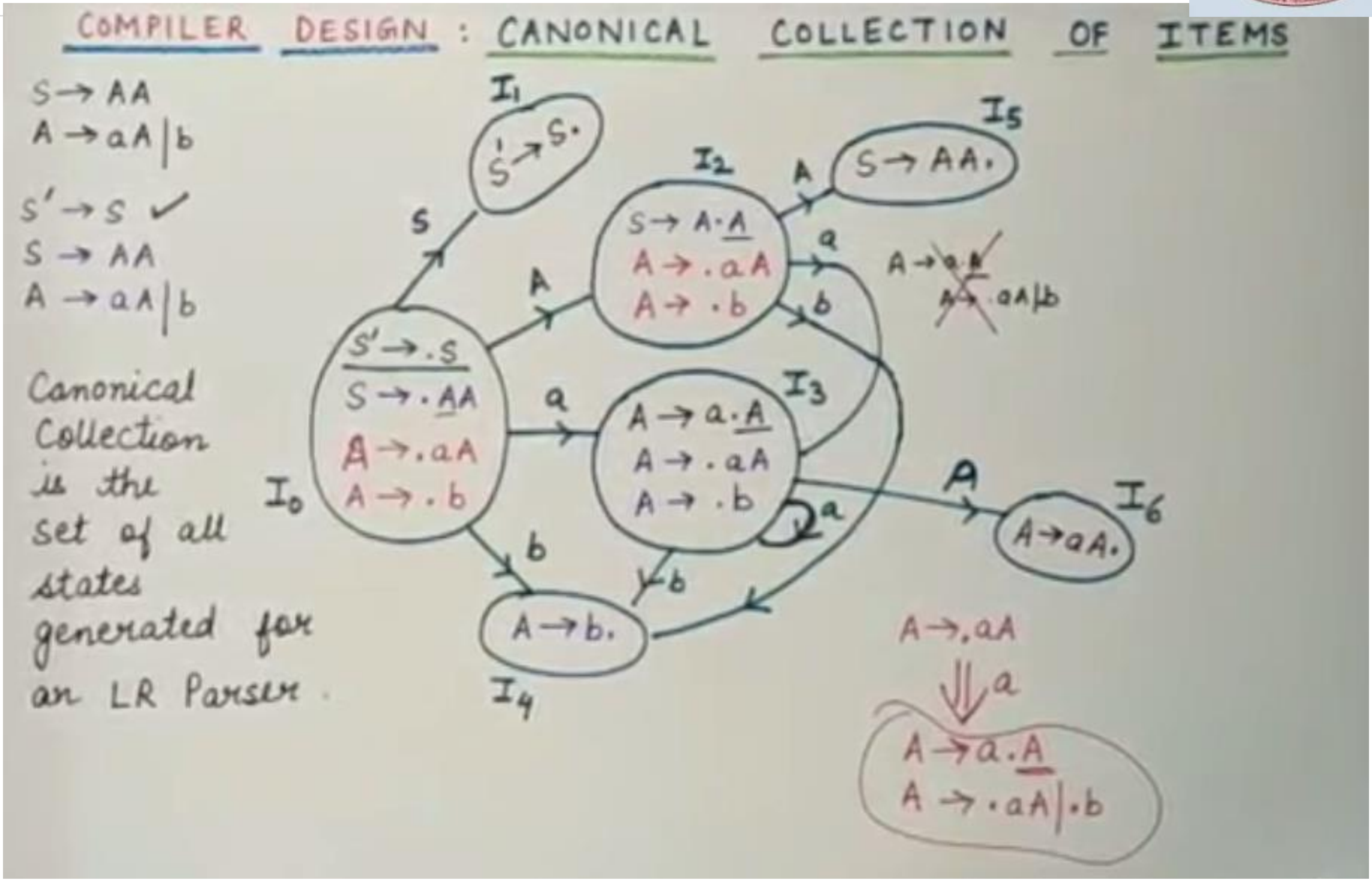
Example: SLR(1)- simple LR



Rules to construct SLR parsing table

1. Construct $C = \{ I_0, I_1, \ldots \ldots . In\}$, 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 $(I_i\ , a) \ = \ I_j$, 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.

LR Parser


COMPILER DESIGN : CANONICAL COLLECTION OF ITEMS
S → AA
A → aA | b
S' → S ✓
S → AA
A → aA | b
Canonical Collection is the set of all states generated for an LR Parser.
I0
S' → .S
S → .AA
A → .aA
A → .b
I1
S' → S.
I2
S → A.A
A → .aA
A → .b
I3
A → a.A
A → .aA
A → .b
I4
A → b.
I5
S → AA.
I6
A → aA.
S
A
a
b
A → .aA
A → a.A
A → .aA | .b

Rules to construct SLR parsing table

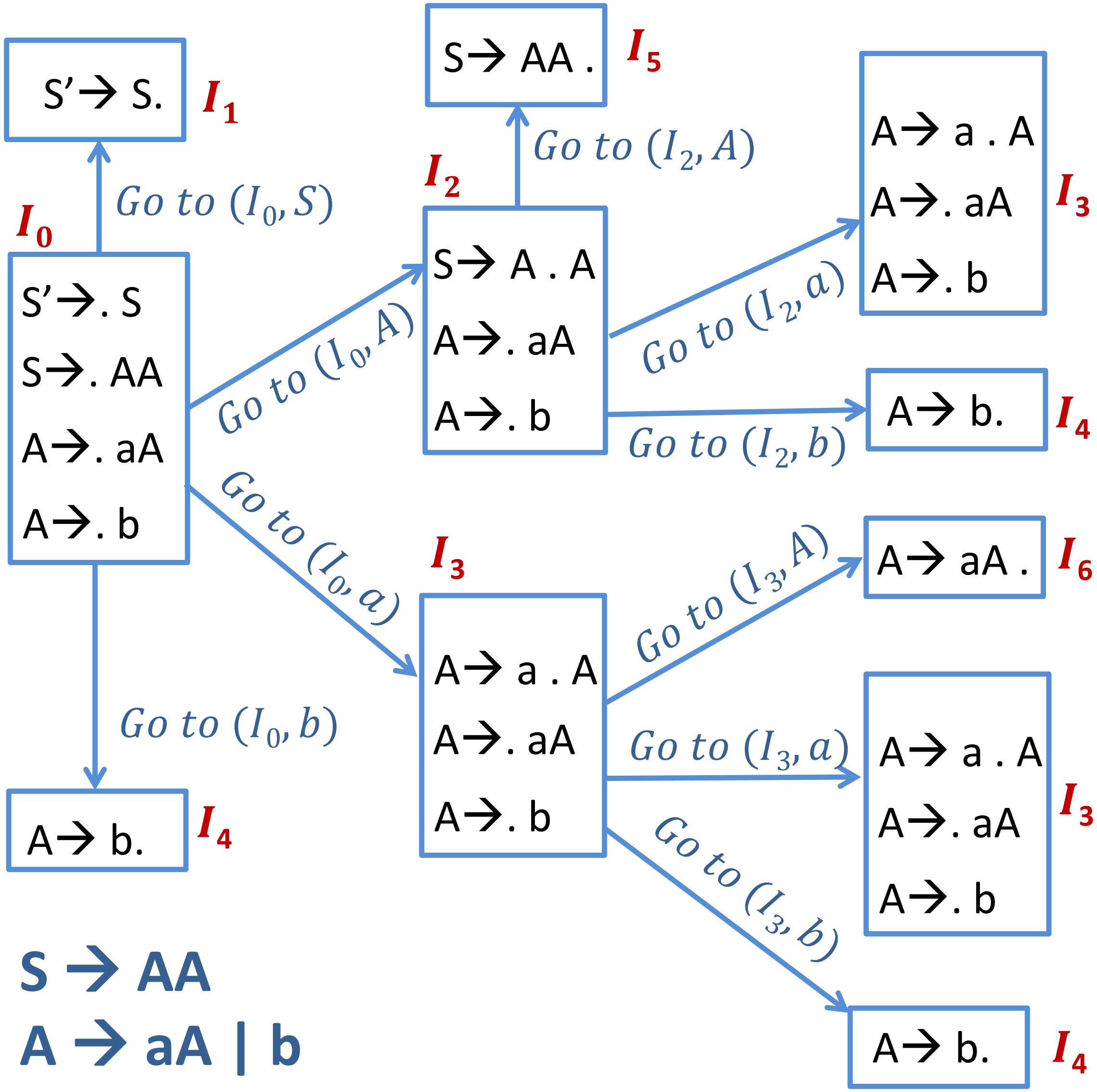
LR(0) PARSING TABLE

	ACTION			GOTO	
	a	b	$	S	A
0	S_3	S_4		1	2
1			accept ACC		
2	S_3	S_4			5
3	S_3	S_4			6
4	r_3	r_3	r_3		
5	r_1	r_1	r_1		
6	r_2	r_2	r_2		

Steps To Construct Parsing Table

1. Write all state numbers in leftmost column.
2. Divide the Parsing Table (PT) into 2 parts: ACTION and GOTO
3. For every state I_i if there is a transition on Non Terminal X, to state I_j, fill cell $(i, X) = j$ in PT.
4. For every state I_i, if there is a transition to state I_j on terminal 'y' then fill cell $(i, y) =$ shift j (S_j)
5. For the state I_z having final item for the Augmented Production fill cell $(z, \$) =$ accept
6. For all remaining states I_y having final items fill all the cells in the action part of row y, by reduce (n) → The production no. corresponding to final item.

Example: SLR(1)- simple LR



S → AA
A → aA | b

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

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

LR parsing program

Input: An input string w and an LR parsing table with functions actions and goto for a grammar G

Output: if w is in $L(G)$, a bottom up parse for w; otherwise, an error indication.

set ip to point to the first symbol of $w\$$;

repeat forever begin

$s = top\ of\ stack$

$a\ is\ input\ symbol\ pointed\ by\ ip$;

$if\ action[s,a] = \text{“}shift\ s\text{”}$ then

PUSH a

PUSH s'

advance ip to next input symbol

End

else $if\ action[s,a] = \text{“}reduce\ A \rightarrow \beta\text{“}$ then

$POP\ 2 * |\beta|$ symbols off the stack;

let s' be the new state on top of stack;

$PUSH\ A$ then $goto[s', A]$ on top of stack

output the production $A \rightarrow \beta$

end

else $if\ action[s,a] = \text{“}accept\text{”}$ then

return

else error()

end

String parsing using SLR parsing table

Stack	i/p buffer	Action table	Go to table	Parsing action
$ 0	abb$			
A6				
A2				
A5				
S1				

I	Action			Go to	
	a	b	$	S	A
0	S3	S4		1	2
1			Accept		
2	S3	S4			5
3	S3	S4			6
4	R3	R3	R3		
5			R1		
6	R2	R2	R2		

1. **S → AA**
2. **A → aA**
3. **A → b**

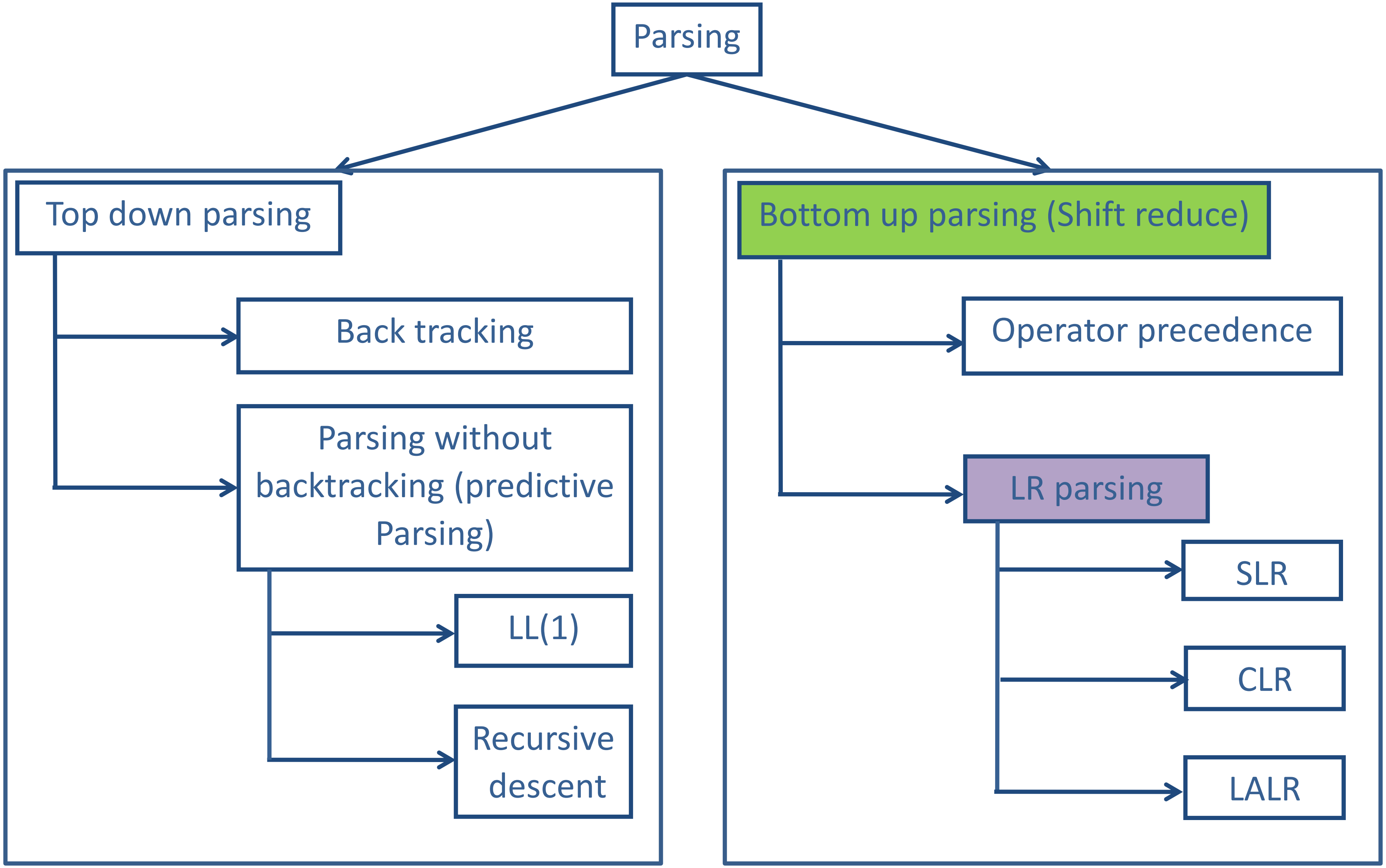
Exercise

E→ E+T | T

T→ TF | F

F→ F* | a | b

Parsing methods



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

How to calculate look ahead?

S→CC

C→ cC | d

Closure(I)

S'→.S,$

S→.CC,

C→.cC,

C→.d,

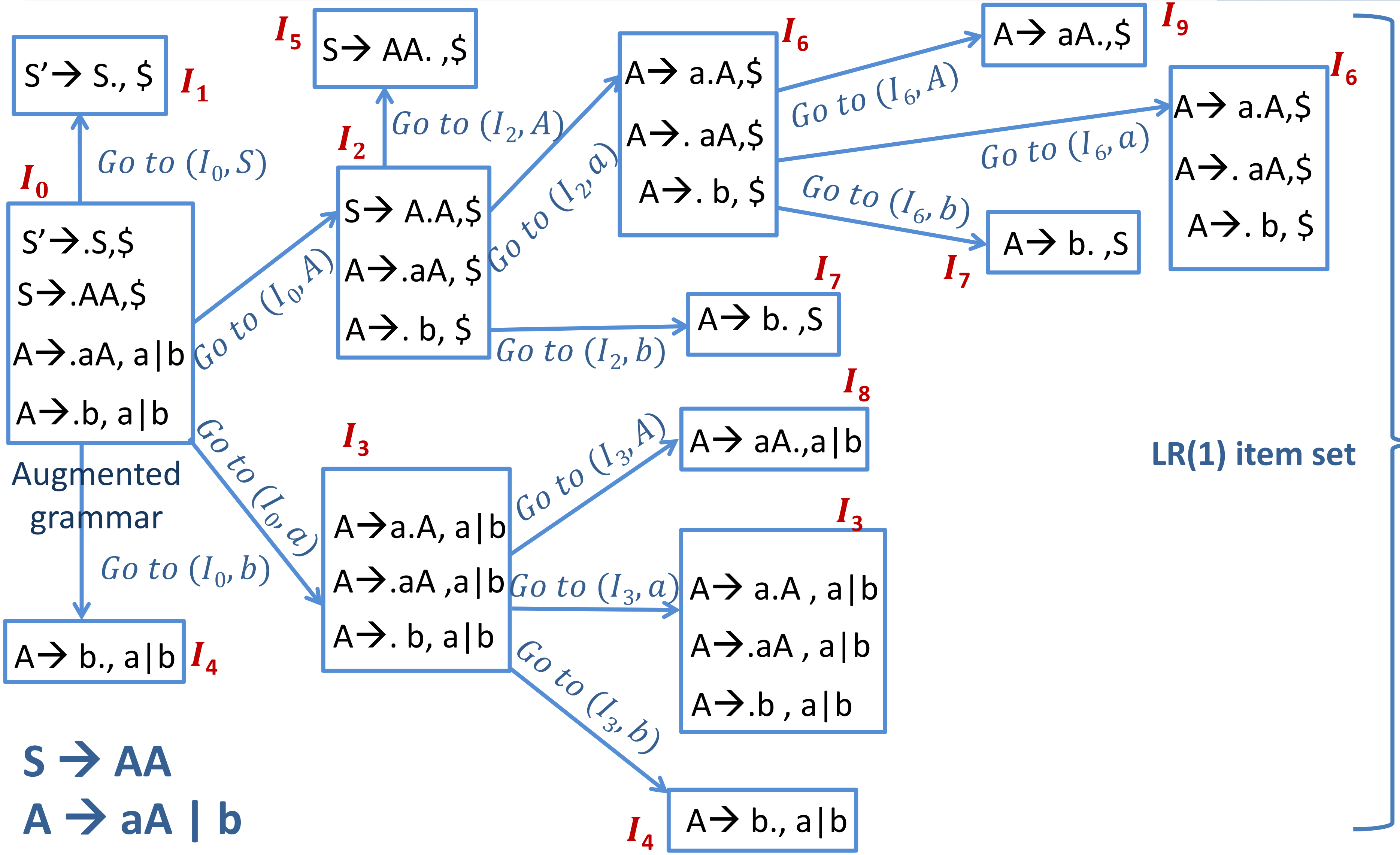
S'	→		.	S		,	$
A	→	α	.	X	β	,	a

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

S	→		.	C	C	,	$
A	→	α	.	X	β	,	a

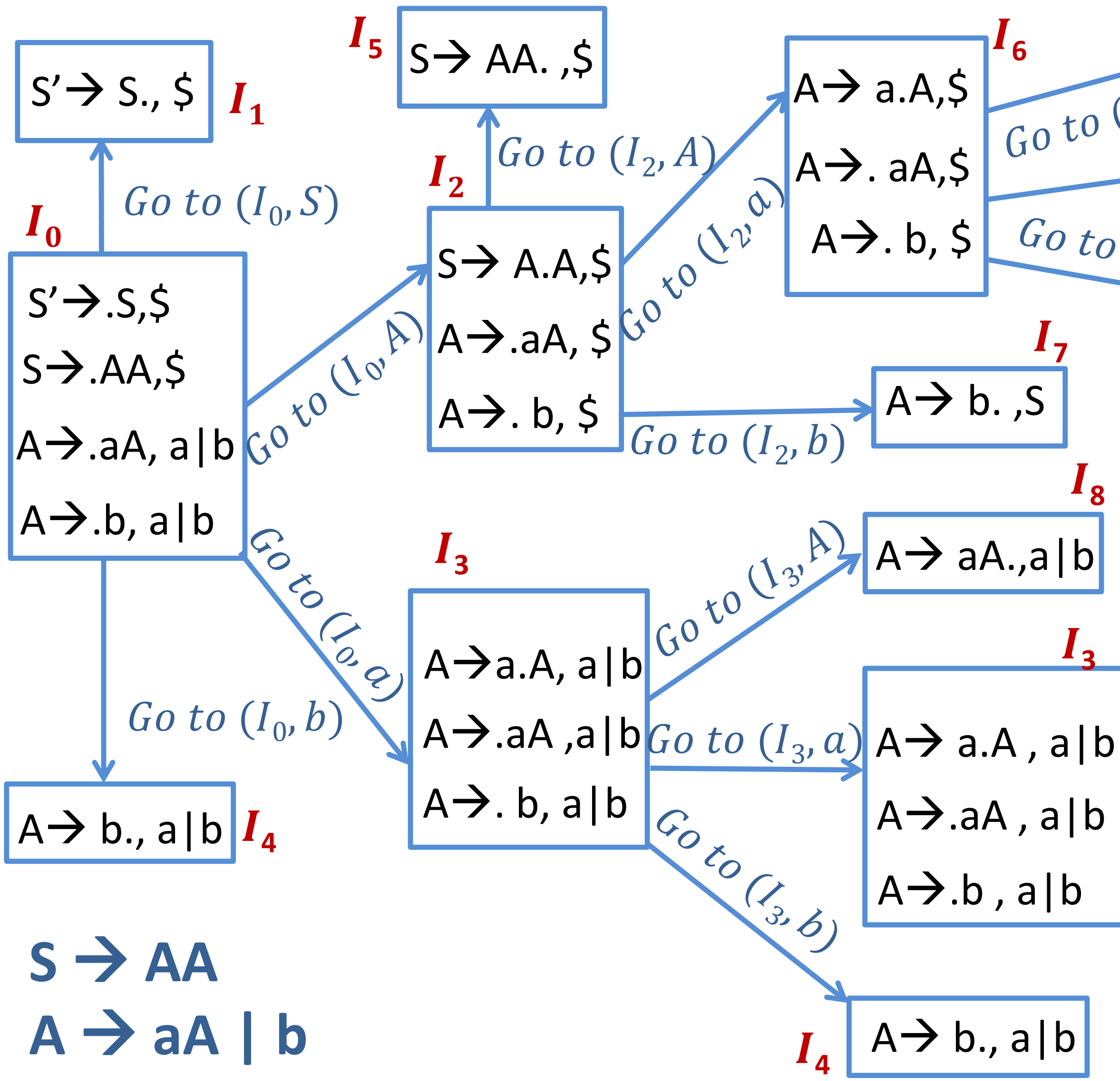
$$\text{Lookahead} = \text{First}(\boldsymbol{\beta a})$$
$$\text{First}(C\$)$$
$$= \boldsymbol{c, d}$$

Example: CLR(1)- canonical LR



S → AA
A → aA | b

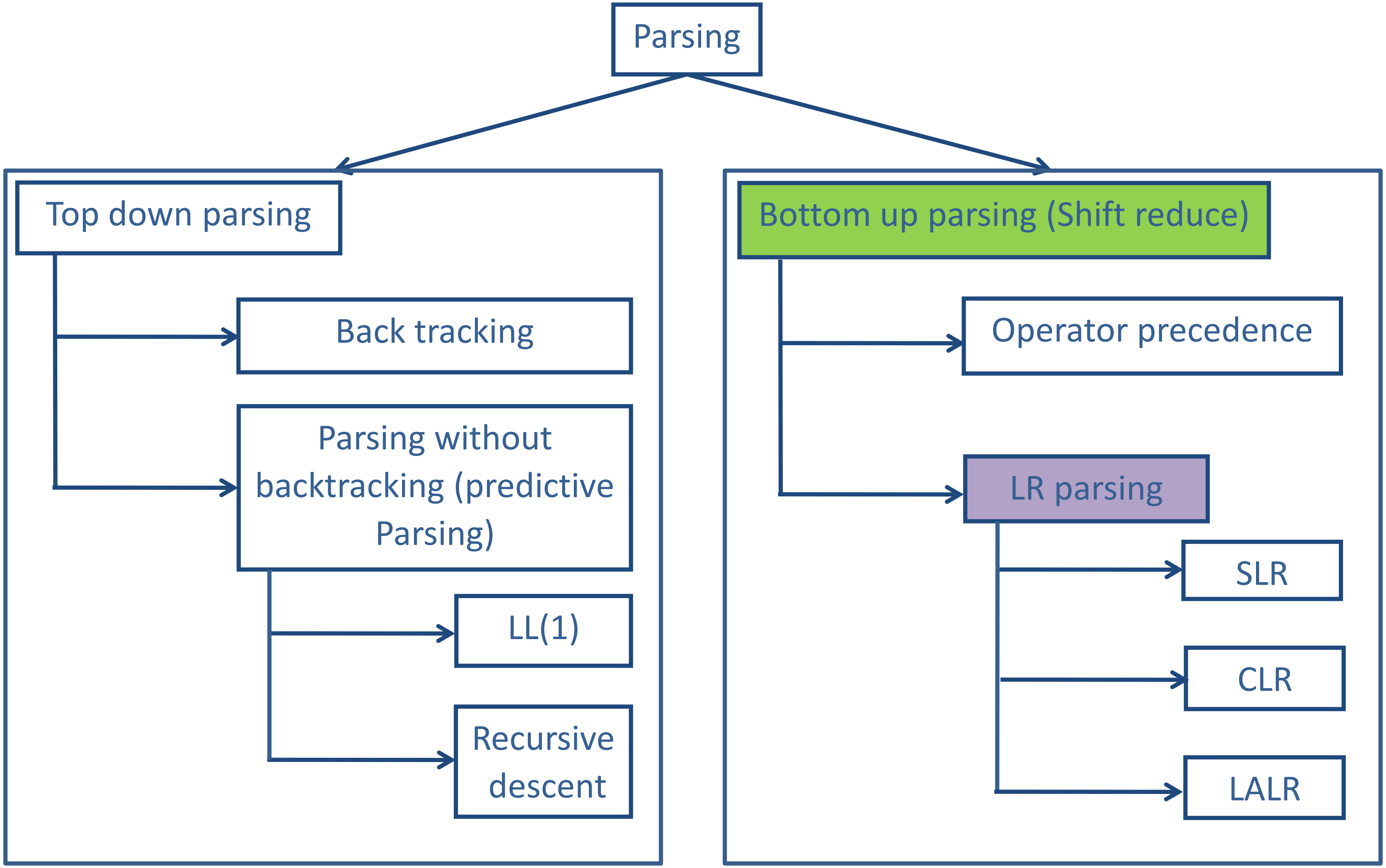
Example: CLR(1)- canonical LR



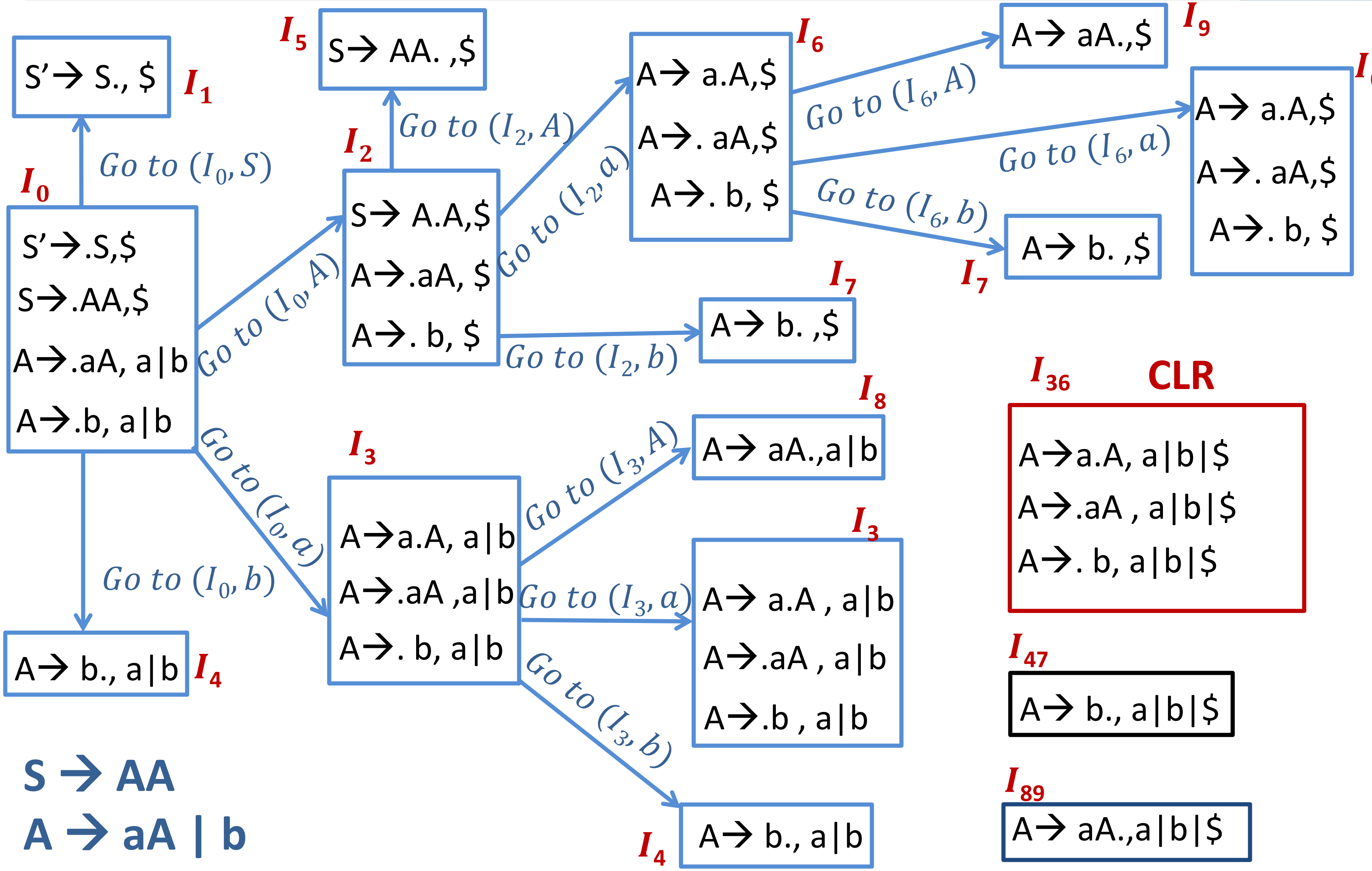
S → AA
A → aA | b

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

Parsing methods



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

END OF PARSING