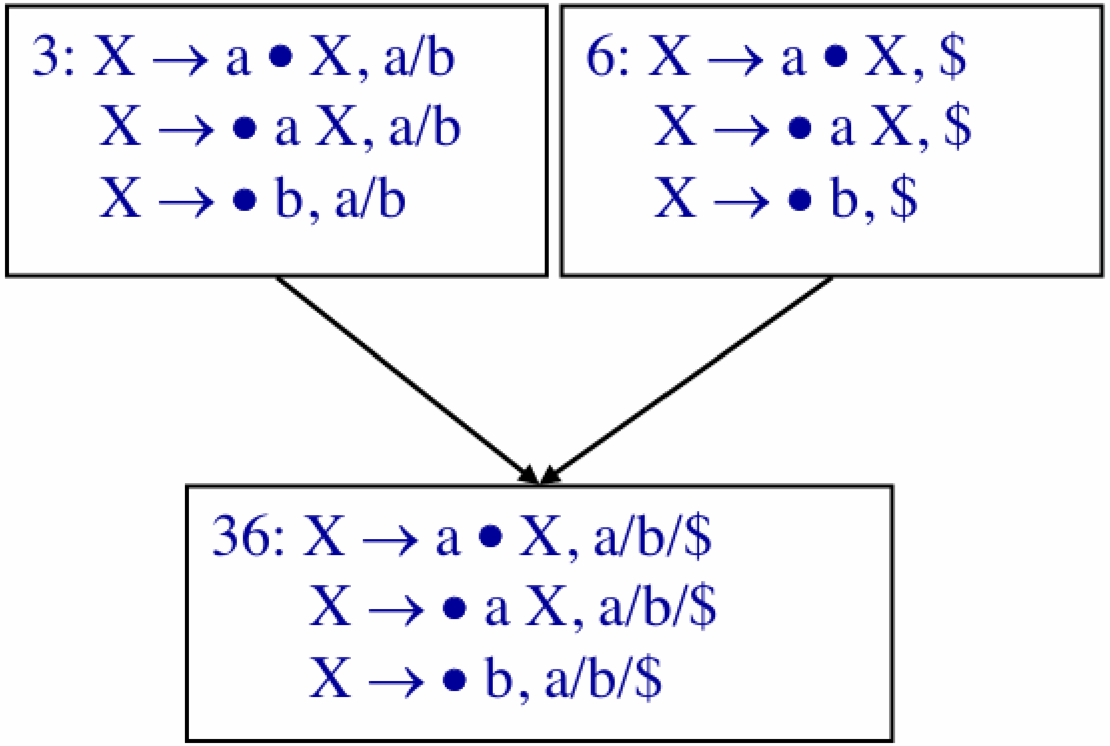
Canonical LR(1) Recap

- LR(1) uses left context, current handle, and lookahead to decide when to reduce or shift
- Most powerful parser so far (can handle more context-free grammars)
- LALR(1) is a practical simplification with fewer states used by yacc/bison to avoid the very large tables generated by LR(1)

Merging States in LALR(1)

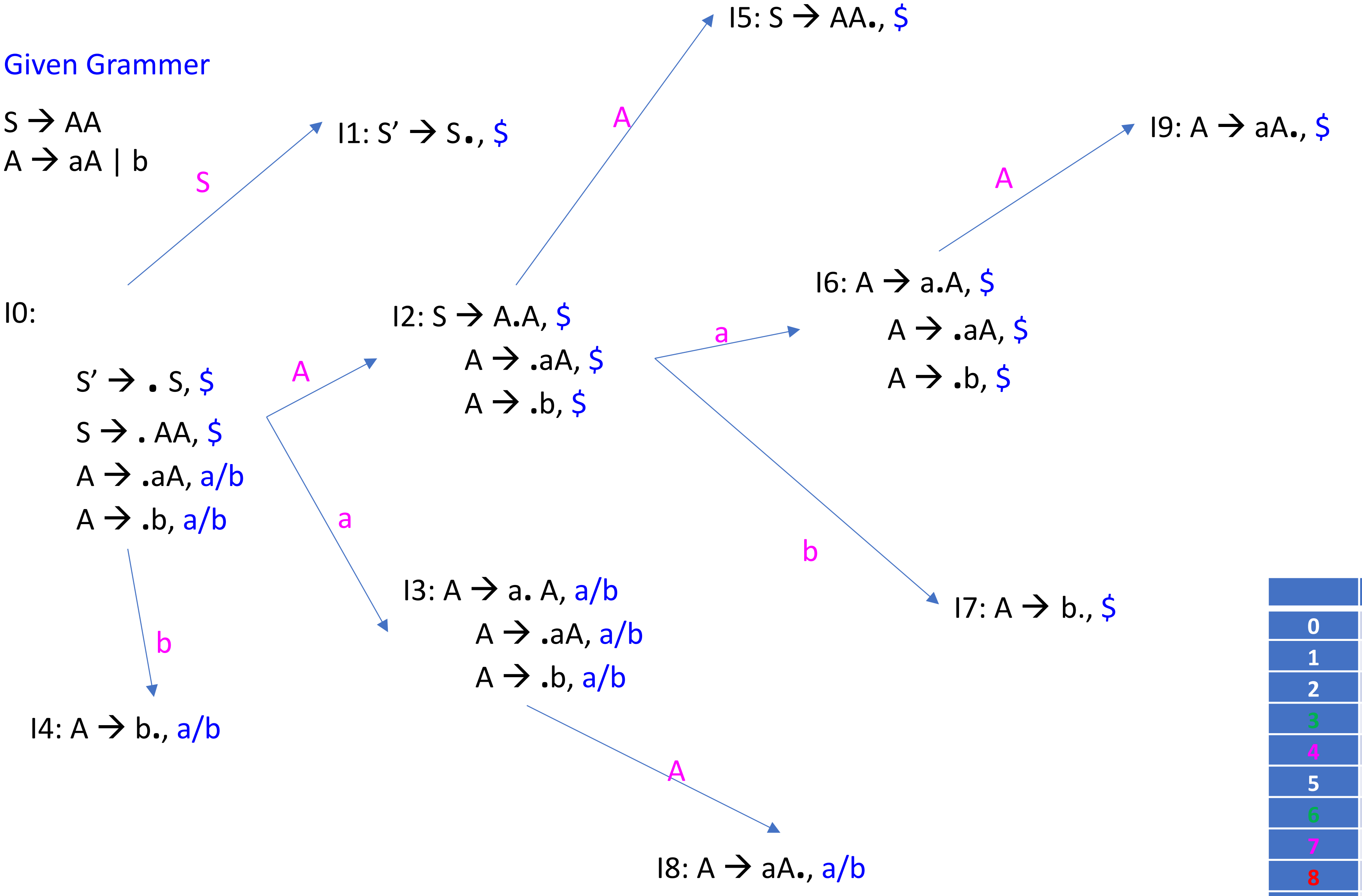
- $S' \to S$
  $S \to XX$
  $X \to aX$
  $X \to b$
- Same **Core Set**
- Different lookaheads



LALR Parsing

Given Grammer

S → AA
A → aA | b



	Action			GoTo	
	a	b	$	S	A
0	S_3	S_4		1	2
1			Accept		
2	S_6	S_7			5
3	S_3	S_4			8
4	R_3	R_3			
5			R_1		
6	S_6	S_7			9
7			R_3		
8	R_2	R_2			
9			R_2		

Contd.,

Before Merging States in LALR(1) parsing Table

	a	b	$	S	A
0	S_3	S_4		1	2
1			Accept		
2	S_6	S_7			5
3	S_3	S_4			8
4	R_3	R_3			
5		R_1			
6	S_6	S_7			9
7		R_3			
8	R_2	R_2			
9		R_2			

After Merging States in LALR(1) parsing Table

	a	b	$	S	A
0	S_3	S_4		1	2
1			Accept		
2	S_6	S_7			5
36	S_{36}	S_{47}			89
47	R_3	R_3	R_3		
5		R_1			
36	S_{36}	S_{47}	R_3		89
47	R_3	R_3	R_3		
89	R_2	R_2	R_2		
89	R_2	R_2	R_2		

Merging States in LALR(1) from the previous slide LR(1) Items

	a	b	$	S	A
36	S_{36}	S_{47}			89
47	R_3	R_3	R_3		
89	R_2	R_2	R_2		

Contd.,

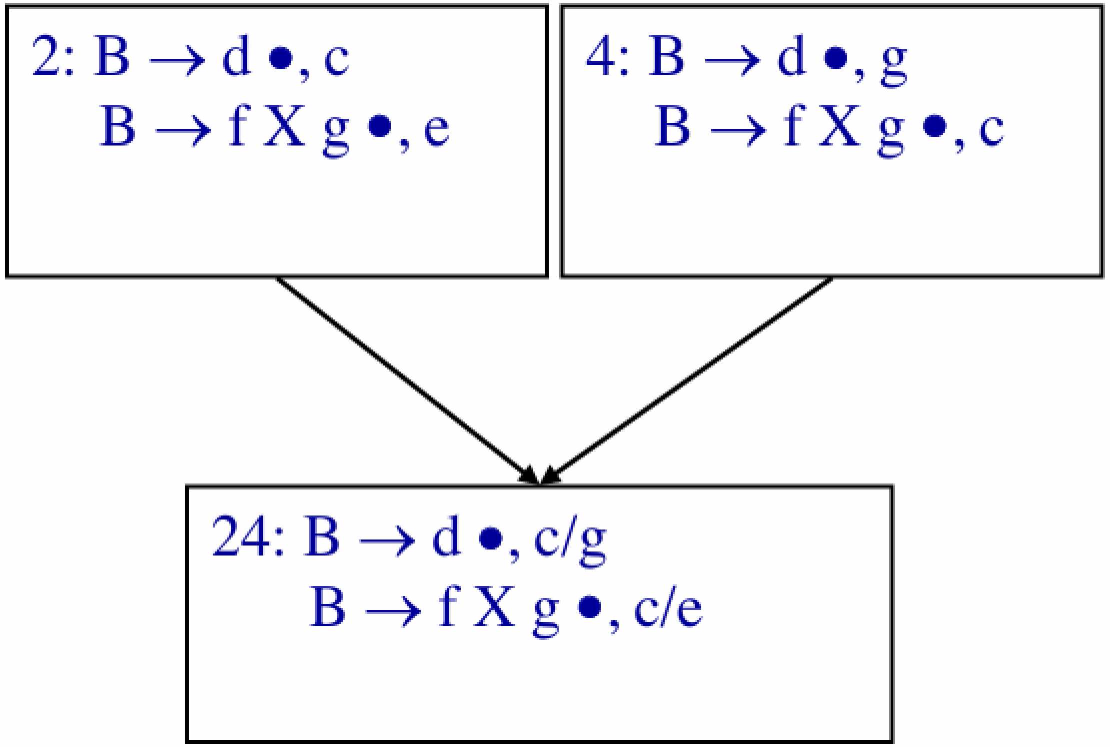
Final LALR (1) parsing Table

	A	B	$	S	A
0	S_3	S_4		1	2
1			Accept		
2	S_6	S_7			5
~~36~~	~~S_{36}~~	~~S_{47}~~	-	-	~~89~~
47	R_3	R_3	R_3		
5		R_1			
36	S_{36}	S_{47}	R_3		89
~~47~~	~~R_3~~	~~R_3~~	~~R_3~~-	-	-
89	R_2	R_2	R_2		
~~89~~	~~R_2~~	~~R_2~~	~~R_2~~-	-	-

R/R conflicts when merging

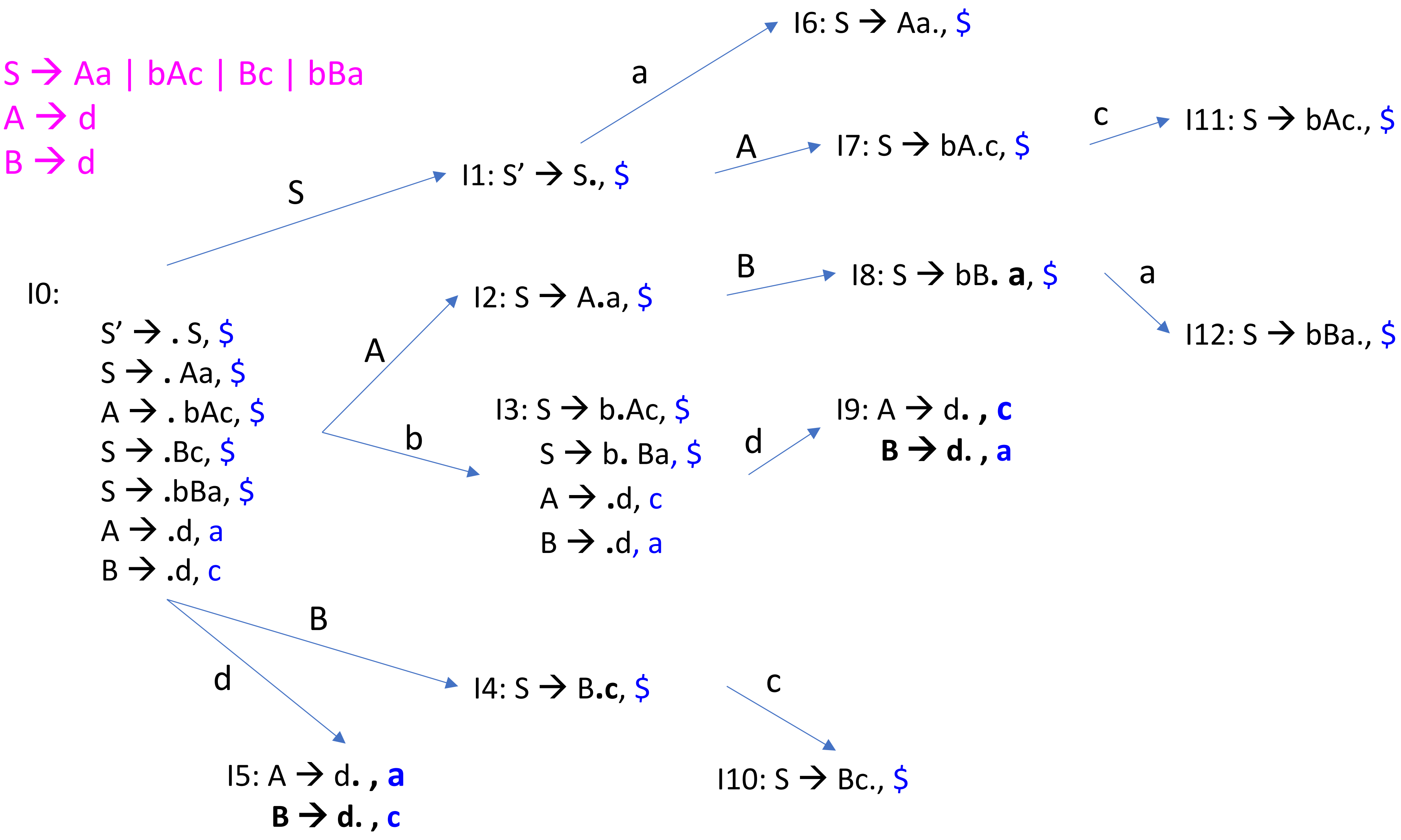
- $B \rightarrow d$
  $B \rightarrow f\ X\ g$
  $X \rightarrow \ldots$

- If R/R conflicts are introduced, grammar is not LALR(1)!



Contd.,

S → Aa | bAc | Bc | bBa
A → d
B → d



Parsing Table

Given Grammar is Not LALR (1)

	a	b	c	d	$	S	A	B
0		S3		S5		1	2	4
1					Accept			
2	S6							
3				S9			7	8
4			S10					
~~5~~	~~R5~~		~~R6~~					
6					R1			
7				S11				
8	S12							
~~9~~	~~R6~~		~~R5~~					
10					R3			
11					R2			
12					R4			

LALR(1)

- LALR(1) Condition:
  - Assumption: merging does not introduce reduce/reduce conflicts
  - Shift/reduce cannot be introduced
- Merging brute force or step-by-step
- More compact than canonical LR, like SLR(1)
- More powerful than SLR(1)
  - Not always merge to full Follow Set

Operator Precedence Parsing

- Operator precedence grammar is a kind of shift-reduce parsing method that can be applied to a small class of operator grammars.
- An operator grammar has two important characteristics:

  1. There are no € productions.

  2. No production would have two adjacent non-terminals.

- The operator grammar to accept expressions is given below.
  - E → E+E / E
  - E → E-E / E
  - E → E*E / E
  - E → E/E / E
  - E → E^E / E
  - E → -E / E
  - E → (E) / E
  - E → id

Contd.,

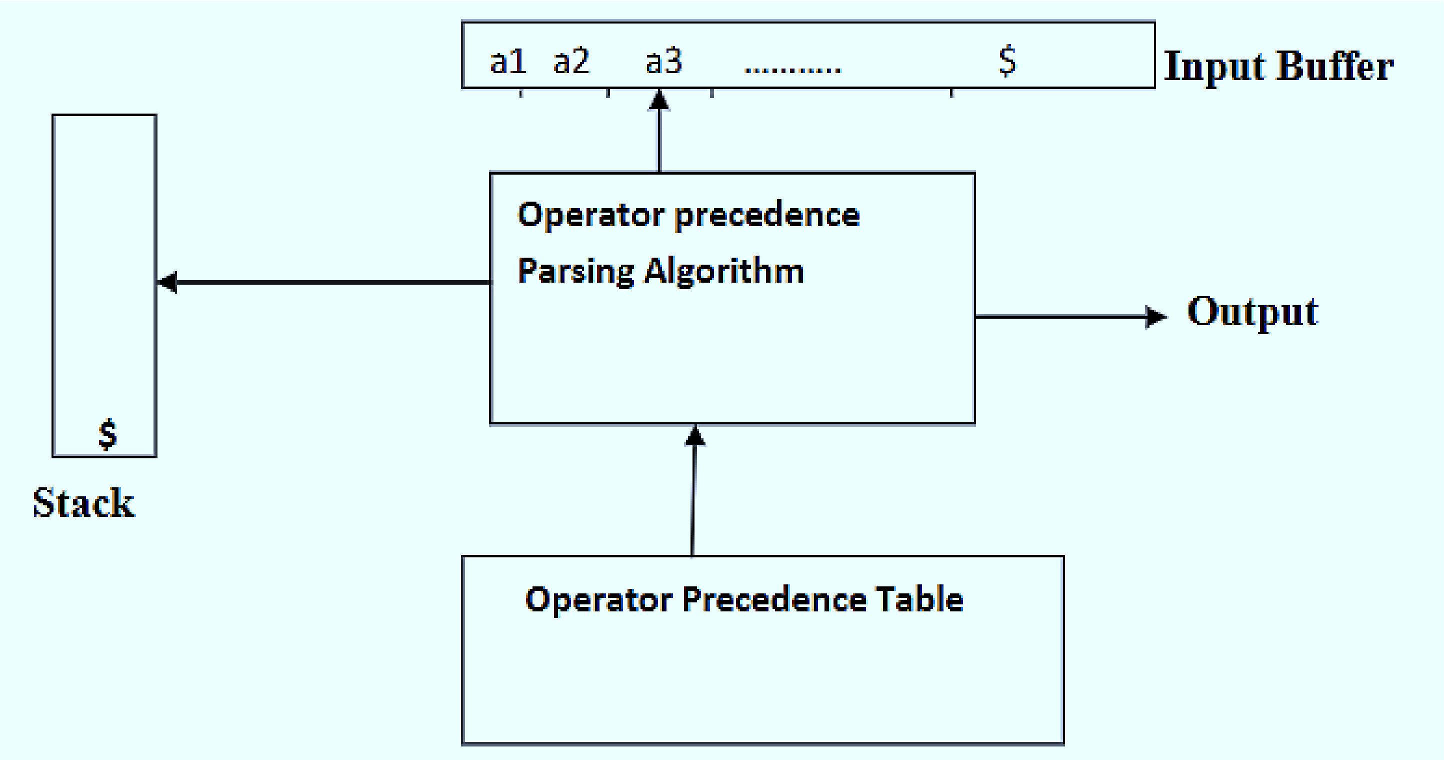
- Two main Challenges in the operator precedence parsing are:
  - 1. Identification of Correct handles in the reduction step, such that the given input should be reduced to the starting symbol of the grammar.
  - 2. Identification of which production to use for reducing in the reduction steps, such that we should correctly reduce the given input to the starting symbol of the grammar.

- There are three kinds of precedence relations that will exist between the pair of terminals "a" and "b" as follows:
  - If a has higher precedence over b; a .> b
  - If a has lower precedence over b; a <. b
  - If a and b have equal precedence, a =. b

Note:

- **id** has higher precedence than any other symbol
- **$** has the lowest precedence.
- **If two operators have equal precedence**, then we check the **Associativity** of that particular operator.

Components of an operator precedence parser



Example, If the grammar is E → E+E

E → E * E

E → id

Construct an operator precedence table and accept an input string " id +id * id"

Operator Relation Table

	id	**+**	*****	**$**
id		.>	.>	.>
+	<.	.>	<.	.>
*	<.	.>	.>	.>
$	<.	<.	<.	.>

The input string: id1 + id2 * id3

After inserting precedence relations becomes: $ <· id1 ·> + <· id2 ·> * <· id3 ·> $

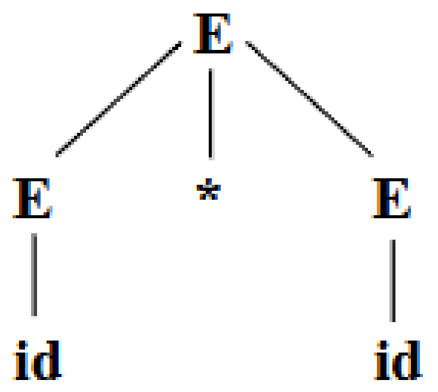
Basic Principle: Having precedence relations allows identifying handles as follows.

1. Scan the string from left until seeing **·>** and put a pointer.
2. Scan backwards the string from right to left until seeing **<·**
3. Everything between the two relations **<· and ·>** forms the handle
4. Replace the **handle with the head of the production.**

id * id and corresponding Parse Tree are as under.

Stack	Input	Operations
$	id * id $	$ <• id, shift_id‘ in to stack
$ id	*id $	id •> *, reduce_id‘ using E-> id
$E	*id $	$ <• *, shift_*‘ in to stack
$E*	id$	* <• id , shift_id‘ in to Stack
$E*id	$	id •> $, reduce_id‘ using E->id
$E*E	$	*•> $, reduce_*‘ using E->E*E
$E	$	$=$=$, so parsing is successful

	id	+	*	$
id		.>	.>	.>
+	<.	.>	<.	.>
*	<.	.>	.>	.>
$	<.	<.	<.	.>



id + id* id and corresponding Parse Tree are as under.

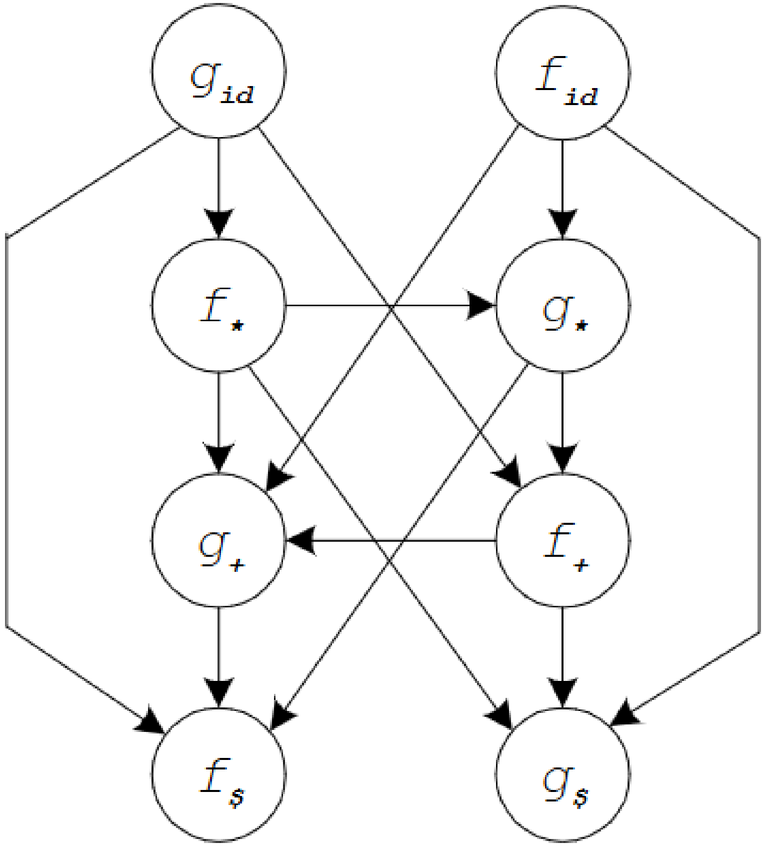
Stack	Input	Operations
$	id + id * id $	($ <. Id) Shift
$ id	+ id* id $	(id >. +) Reduce E-> id
$ E	+ id * id $	($ <. +) Shift
$ E+	id * id $	(+ <. Id) Shift
$ E+ id	* id $	(id .> *) Reduce E->id
$ E+ E	* id $	(+ <. *) Shift
$ E + E *	id $	(* <. Id) Shift
$ E + E * id	$	(id .> $) Reduce E → id
$ E + E * E	$	(* .> $) Reduce E → E*E
$ E + E	$	(+ .> $) Reduce E → E + E
$ E	$	Accept

	id	+	*	$
id		.>	.>	.>
+	<.	.>	<.	.>
*	<.	.>	.>	.>
$	<.	<.	<.	.>

Consider the following table

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

Resulting graph



From the previous graph, we have to extract the longest path, then the following precedence functions:

	id	+	*	$
f	4	2	4	0
g	5	1	3	0

Operator Precedence Parsing Algorithm

- Initialize: Set *ip* to point to the first symbol of the input string *w$*
- Repeat: Let *b* be the top stack symbol, *a* the input symbol pointed to by *ip*

```
if (a is $ and b is $)
            return
else
    if a ·> b or a =· b then
          push a onto the stack
          advance ip to the next input symbol
else
    if a <· b then
          repeat
             c ← pop the stack
           until (c .> stack-top)
    else error
end
```

Consider the following grammar

- $E \rightarrow EOE \mid -E \mid (E) \mid id$
- $O \rightarrow - \mid + \mid * \mid / \mid \uparrow$

Using Operator precedence for parse the expression

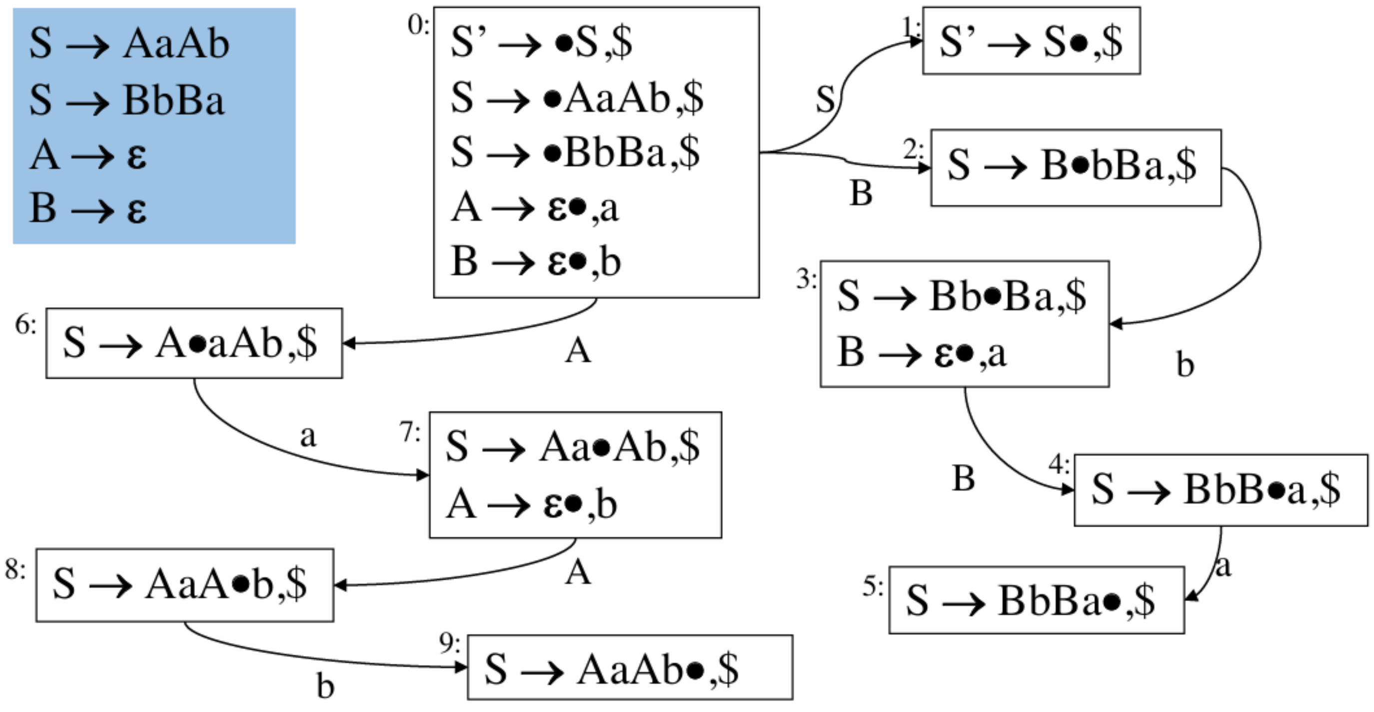
$id1*(id2+id3) \uparrow id$

Solution:

$E \rightarrow E+E \mid E-E \mid E*E \mid E/E \mid E \hat{} E \mid (E) \mid -E \mid id$

Stack		input	Action
\$	$<$	$id1*(id2+id3) \uparrow id$ \$	shift
\$ id1	$>$	$*(id2+id3) \uparrow id$ \$	Reduce $E \rightarrow id$
\$E	$>$	$*(id2+id3) \uparrow id$ \$	shift
\$E*	$>$	$(id2+id3) \uparrow id$ \$	shift
\$ E*(	$<$	$id2+id3) \uparrow id$ \$	shift
\$ E*(id2	$>$	$+id3) \uparrow id$ \$	Reduce $E \rightarrow id$
\$ E*(E	$<$	$+id3) \uparrow id$ \$	shift
\$ E*(E+	$<$	$id3) \uparrow id$ \$	shift
\$ E*(E+ id3	$>$	$) \uparrow id$ \$	Reduce $E \rightarrow id$
\$ E*(E+ E	$>$	$) \uparrow id$ \$	Reduce $E \rightarrow E+E$
\$ E*(E	$=$	$) \uparrow id$ \$	Shift
\$E*(E)	$>$	$\uparrow id$\$	Reduce $E \rightarrow (E)$
\$E * E	$<$	$\uparrow id$\$	shift
\$E*E$\uparrow$	$<$	id\$	shift
\$E*E$\uparrow$ id	$>$	\$	Reduce $E \rightarrow id$
\$E*E$\uparrow$ E	$>$	\$	Reduce $E \rightarrow E \uparrow E$
\$ E * E	$>$	\$	Reduce $E \rightarrow E*E$
\$ E		\$	Accept

Set of items with Epsilon rules



The diagram is the canonical LR(1) item collection for this grammar; every numbered box is an LR(1) state.