Chapter 5: Syntax Analysis — Bottom-Up Parsing

Zhen Gao
gaozhen@tongji.edu.cn

LR Parsing Method

- **LR parsing method: proposed in 1965 by Donald Knuth**

Grammar → Parsing Table Generator → Parsing Table

Input Tokens → LR Parsing Control Program (Parsing Table) → Output Syntax tree

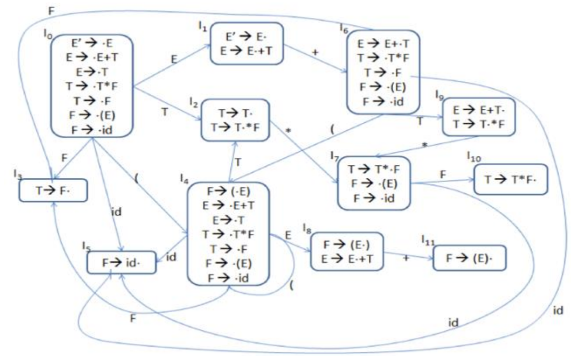
*

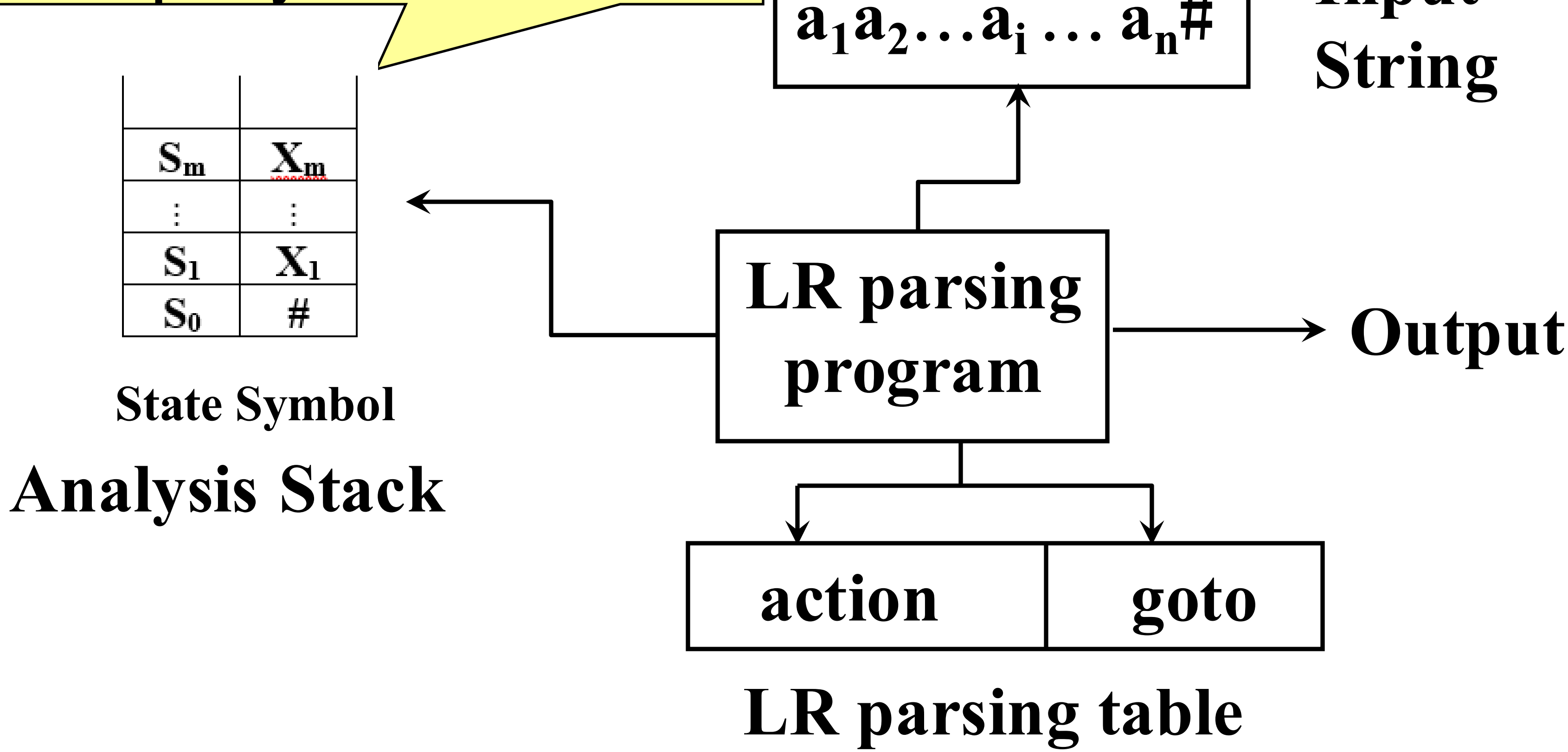
Principle of LR Parsing

- During the shift-reduce process, the parser looks for **the handle**
  - **History**: the sequence of symbols already shifted and reduced in the parsing stack
  - **Lookahead**: predicting possible upcoming input symbols based on the current production being used
  - **Current**: the current input symbol

LR Parser Model

Combine **history** and **lookahead** into **state**
Each step is uniquely determined by the **top state of the stack** and the **current input symbol**





*

Outline

- **Basic Issues of Bottom-Up Parsing**
- **Canonical Reduction**
- **Operator-Precedence Parsing Method**
- **LR Parsing Method**

(1) E→E＋T (2) E→T
(3) T→T*F (4) T→F
(5) F→(E) (6) F→i

状态	ACTION						GOTO		
	i	+	*	(	)	#	E	T	F
0	s5			s4			1	2	3
1		s6				acc			
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4			8	2	3
5		r6	r6		r6	r6			
6	s5			s4				9	3
7	s5			s4					10
8		s6			s11				
9		r1	s7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

Action[s, a]

	ACTION						GOTO		
状态	i	+	*	(	)	#	E	T	F
0	s5			s4			1	2	3
1		s6				acc			
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4			8	2	3
5		r6	r6		r6	r6			
6	s5			s4				9	3
7	s5			s4					10
8		s6			s11				
9		r1	s7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

- Four Actions of ACTION[s, a]:
  - **Shift** – Push next state s' and symbol a onto the stack; advance input.
  - **Reduce** – Apply $A \rightarrow \beta$: pop $|\beta|$ items, push (GOTO[$s_{m-|\beta|}$, A], A).
  - **Accept** – Parsing succeeds; stop.
  - **Error** – Report an error.

LR Parsing Process

Changes of the Triple (Stack State Sequence, Shift-Reduce String, Input String):

- **Start:** $(S_0, \#, a_1 a_2 \ldots a_n \#)$
- **Current Step:** $(S_0 S_1 \ldots S_m, \#X_1 X_2 \ldots X_m, a_i\, a_{i+1} \ldots a_n \#)$
- **Next Step:** ACTION $[S_m, a_i]$

  If ACTION $[S_m, a_i]$ is Shift and GOTO $[S_m, a_i] = S$

  Triple becomes

  $(S_0 S_1 \ldots S_m\, S, \ \#X_1 X_2 \ldots X_m a_i, \ a_{i+1} \ldots a_n \#)$

  If ACTION $[S_m, a_i]$ is Reduce $\{A \rightarrow \beta\}$,

  and $|\beta| = r$, $\beta = X_{m-r+1} \ldots X_m$, GOTO $[S_{m-r}, A] = S$,

  Triple becomes:

  $(S_0 S_1 \ldots S_{m-r}\, S, \#X_1 X_2 \ldots X_{m-r} A, a_i\, a_{i+1} \ldots a_n \#)$

  If ACTION $[S_m, a_i]$ is Accept, Stop

  If ACTION $[S_m, a_i]$ is Error, Handle error

*

LR Parser Control Program

```
let a be the first symbol of w$;
while(1) { /* repeat forever */
        let s be the state on top of the stack;
        if ( ACTION[s, a] = shift t ) {
                push t onto the stack;
                let a be the next input symbol;
        } else if ( ACTION[s, a] = reduce A → β ) {
                pop |β| symbols off the stack;
                let state t now be on top of the stack;
                push GOTO[t, A] onto the stack;
                output the production A → β;
        } else if ( ACTION[s, a] = accept ) break; /* parsing is done */
        else call error-recovery routine;
}
```

Figure 4.36: LR-parsing program

Book: Compilers Principles Techniques and Tools (2nd Edition) P251

LR Parsing Table

- **Control Program: Same for all LR parsers**
- **Parsing Table: Key to automatically generating a syntax parser**
  - **LR(0) Table: Basic, limited**
  - **SLR Table: Simple LR, practical**
  - **Canonical LR Table: Powerful, costly**
  - **LALR Table: Lookahead LR, between SLR and Canonical LR**

Next

How to Construct States for LR(0)

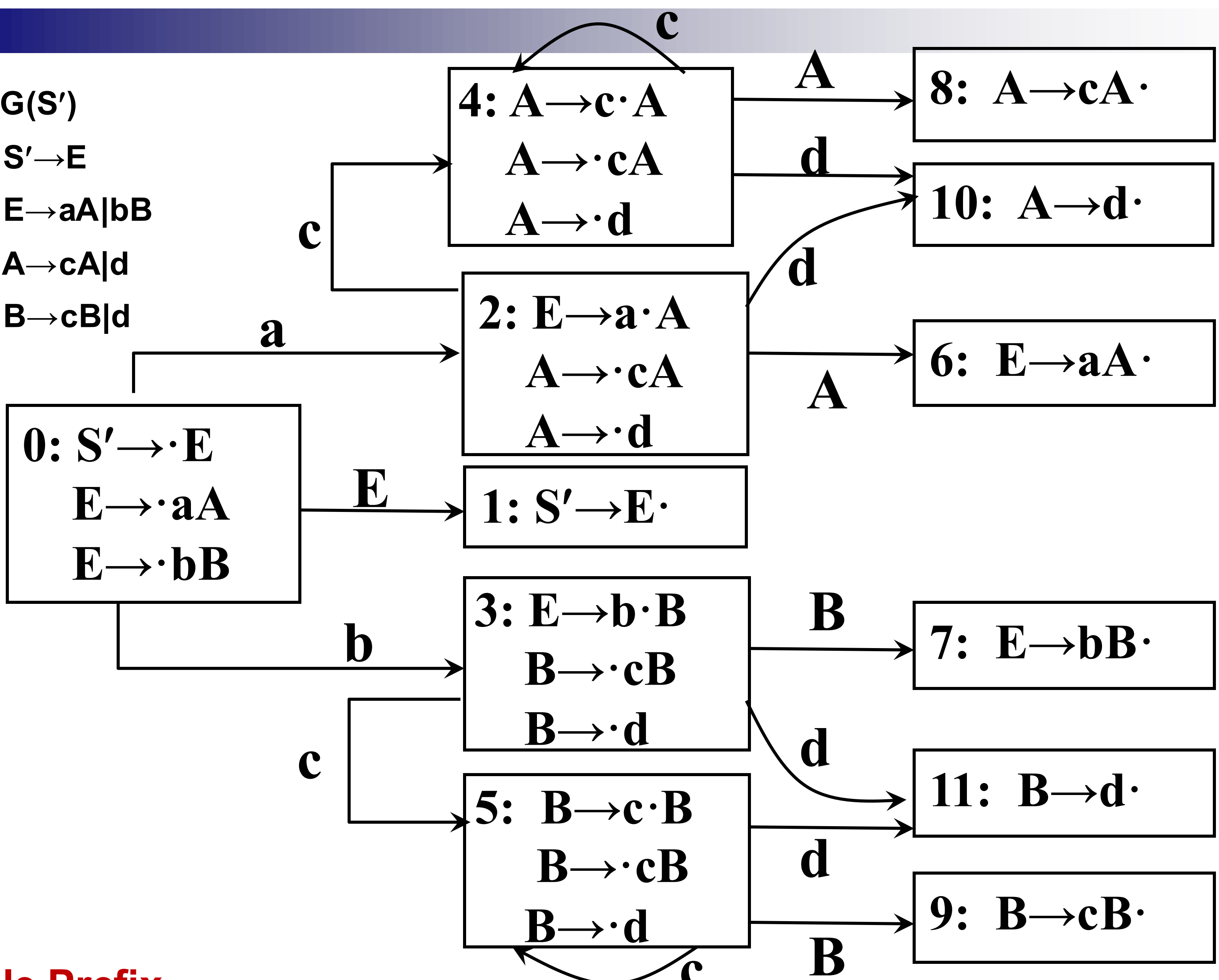
G(S′)

S′→E

E→aA|bB

A→cA|d

B→cB|d



Viable Prefix

The reduction process of a statement ‘ad’

G(E)

$E \rightarrow aAb$

$A \rightarrow Aa|c$

- $A \rightarrow \alpha\cdot$ is called a “reduction item”
- $S' \rightarrow \alpha\cdot$ is called an “accepting item”
- $A \rightarrow \alpha\cdot a\beta$ ($a \in VT$) is called a “shift item”
- $A \rightarrow \alpha\cdot B\beta$ ($B \in VN$) is called a ”pending item”/“item waiting for reduction”

LR(0) Grammar

- An automaton never contains the following situations:

  E->E · *E
  E->E+E ·

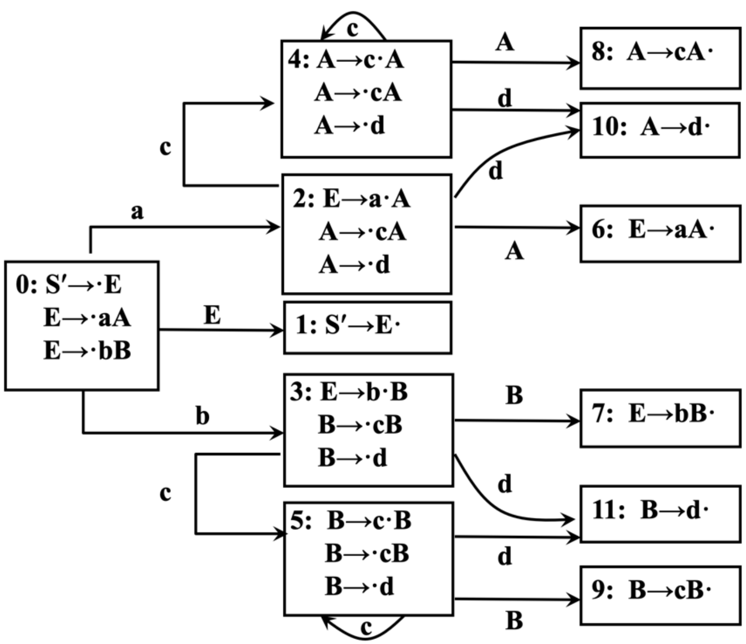
  - Both shift items and reduction items at the same time
  - Multiple reduction items

    P->A·
    Q->A·
- Then G is called an LR(0) grammar.
  - That is: each LR(0) items set in the canonical collection contains no conflicting items

ACTION and GOTO table construction

- $A \rightarrow \alpha \cdot a\beta \in I_k$, $GO(I_k, a) = I_j$, a terminal $\rightarrow$ $ACTION[k, a] = s_j$
- $A \rightarrow \alpha \cdot \in I_k \rightarrow ACTION[k, a] = r_j$ for all terminals a (including #)
- $S' \rightarrow S\cdot \in I_k \rightarrow ACTION[k, \#] = acc$
- $GO(I_k, A) = I_j$, A nonterminal $\rightarrow$ $GOTO[k, A] = j$
- Other entries $\rightarrow$ error

G(S′)
(0) S′→E
(1) E→aA
(2) E→bB
(3) A→cA
(4) A→d
(5) B→cB
(6) B→d



	ACTION					GOTO		
S	a	b	c	d	#	E	A	B
0	s2	s3				1		
1					acc			
2			s4	s10			6	
3			s5	s11				7
4			s4	s10			8	
5			s5	s11				9
6	r1	r1	r1	r1	r1			
7	r2	r2	r2	r2	r2			
8	r3	r3	r3	r3	r3			
9	r5	r5	r5	r5	r5			
10	r4	r4	r4	r4	r4			
11	r6	r6	r6	r6	r6			

SLR

SLR Parsing Table

- **LR(0) grammars are simple and rarely practical.**
- **Conflict not always present** in shift/reduce or reduce/reduce items.
  - Example: $\mathbf{I = \{ X \to \alpha \cdot b\beta, A \to \alpha \cdot, B \to \alpha \cdot \}}$, with **FOLLOW(A)** ∩ **FOLLOW(B) = ∅**, **b** ∉ **FOLLOW(A)** ∪ **FOLLOW(B)**
- For input **a**:
  - **a = b** → shift
  - **a** ∈ **FOLLOW(A)** → reduce $A \to \alpha$
  - **a** ∈ **FOLLOW(B)** → reduce $B \to \alpha$
  - Else → error
- **Key: FOLLOW** sets determine whether a conflict occurs.

SLR Parsing Table

- Suppose an LR(0) item set:
  $I = \{ A_1 \to \alpha \cdot a_1 \beta_1, \ldots, A_m \to \alpha \cdot a_m \beta_m, B_1 \to \alpha \cdot, \ldots, B_n \to \alpha \cdot \}$
- If $\{a_1, \ldots, a_m\}$, $\text{FOLLOW}(B_1), \ldots, \text{FOLLOW}(B_n)$ are pairwise disjoint, then:
  - $a = a_i \to$ shift
  - $a \in \text{FOLLOW}(B_i) \to$ reduce $B_i \to \alpha$
  - Otherwise $\to$ error
- This conflict-resolution method is called SLR(1).

Example: The canonical collection of sets of LR(0) items for the following grammar is

(0) $S' \to E$
(1) $E \to E+T$
(2) $E \to T$
(3) $T \to T*F$
(4) $T \to F$
(5) $F \to (E)$
(6) $F \to i$

I_0: $S' \to \cdot E$
$E \to \cdot E+T$
$E \to \cdot T$
$T \to \cdot T*F$
$T \to \cdot T*F$
$T \to \cdot F$
$F \to \cdot\ (E)$
$F \to \cdot i$

I_1: $S' \to E\cdot$
$E \to E\cdot+T$

I_2: $E \to T\cdot$
$T \to T\cdot*F$

I_3: $T \to F\cdot$

I_4: $F \to (\cdot E)$
$E \to \cdot E+T$
$E \to \cdot T$
$T \to \cdot T*F$
$T \to \cdot F$
$F \to \cdot\ (E)$
$F \to \cdot i$

I_5 : $F \to i\cdot$

I_6: $E \to E+\cdot T$
$T \to \cdot T*F$
$T \to \cdot F$
$F \to \cdot(E)$
$F \to \cdot i$

I_7: $T \to T*\cdot F$
$F \to \cdot(E)$
$F \to \cdot i$

I_8: $F \to (E\cdot)$
$E \to E\cdot+T$

I_9: $E \to E+T\cdot$
$T \to T\cdot*F$

I_{10}: $T \to T*F\cdot$

I_{11}: $F \to (E)\cdot$

- I_1, I_2, and I_9 all contain "shift–reduce" conflicts.

I_1: $S' \to E\cdot$ $E \to E\cdot+T$	I_2: $E \to T\cdot$ $T \to T\cdot*F$	I_9: $E \to E+T\cdot$ $T \to T\cdot*F$

FOLLOW(E)＝{#,), +}

$$E \to T\cdot$$
$$T \to T\cdot * F$$

Since FOLLOW(E) = {#,) ,+},
action[2, #]=*action*[2, +]=*action*[2,)]=*r2*
action[2, ∗] = *s7*

状态	ACTION					
	i	+	*	(	)	#
2		r2	s7		r2	r2

(0) $S' \to E$
(1) $E \to E+T$
(2) $E \to T$
(3) $T \to T*F$
(4) $T \to F$
(5) $F \to (E)$
(6) $F \to i$

状态	ACTION						GOTO		
	i	+	*	(	)	#	E	T	F
0	s5			s4			1	2	3
1		s6				acc			
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4			8	2	3
5		r6	r6		r6	r6			
6	s5			s4				9	3
7	s5			s4					10
8		s6			s11				
9		r1	s7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

Construction for SLR(1) parsing table

- If $\mathbf{A \rightarrow \alpha \cdot a\beta \in I_k}$ and $\mathbf{GO(I_k, a) = I_j}$, a terminal $\rightarrow$ $\mathbf{ACTION[k, a] = s_j}$ (shift)
- If $\mathbf{A \rightarrow \alpha \cdot \in I_k}$, then for each $\mathbf{a \in FOLLOW(A)}$ $\rightarrow$ $\mathbf{ACTION[k, a] = r_j}$ (j = production number in G′)
- If $\mathbf{S' \rightarrow S \cdot \in I_k}$ $\rightarrow$ **ACTION[k, #] = acc**
- If $\mathbf{GO(I_k, A) = I_j}$, A nonterminal $\rightarrow$ **GOTO[k, A] = j**
- Any table entry not filled by rules 1–4 $\rightarrow$ **error**

*

Exercise

- **A->aAb|ε**
  - Construct LR parsing table
  - analyze whether aabb is a valid sentence

LR(1)

Problem with SLR: The FOLLOW sets used may include more lookahead symbols than actually possible in practice.

- **Example of a non-SLR grammar:**
  - **(0) S′→S**
  - **(1) S→L=R**
  - **(2) S→R**
  - **(3) L→*R**
  - **(4) L→i**
  - **(5) R→L**

Canonical LR(0) collection

(0) $S' \to S$
(1) $S \to L=R$
(2) $S \to R$
(3) $L \to *R$
(4) $L \to i$
(5) $R \to L$

I_0: $S' \to \cdot S$
$S \to \cdot L=R$
$S \to \cdot R$
$L \to \cdot *R$
$L \to \cdot i$
$R \to \cdot L$

I_1: $S' \to S\cdot$

I_2: $S \to L\cdot=R$
$R \to L\cdot$

I_3: $S \to R\cdot$

I_4: $L \to *\cdot R$
$R \to \cdot L$
$L \to \cdot *R$
$L \to \cdot i$

I_5: $L \to i\cdot$

I_6: $S \to L=\cdot R$
$R \to \cdot L$
$L \to \cdot *R$
$L \to \cdot i$

I_7: $L \to *R\cdot$

I_8: $R \to L\cdot$

I_9: $S \to L=R\cdot$

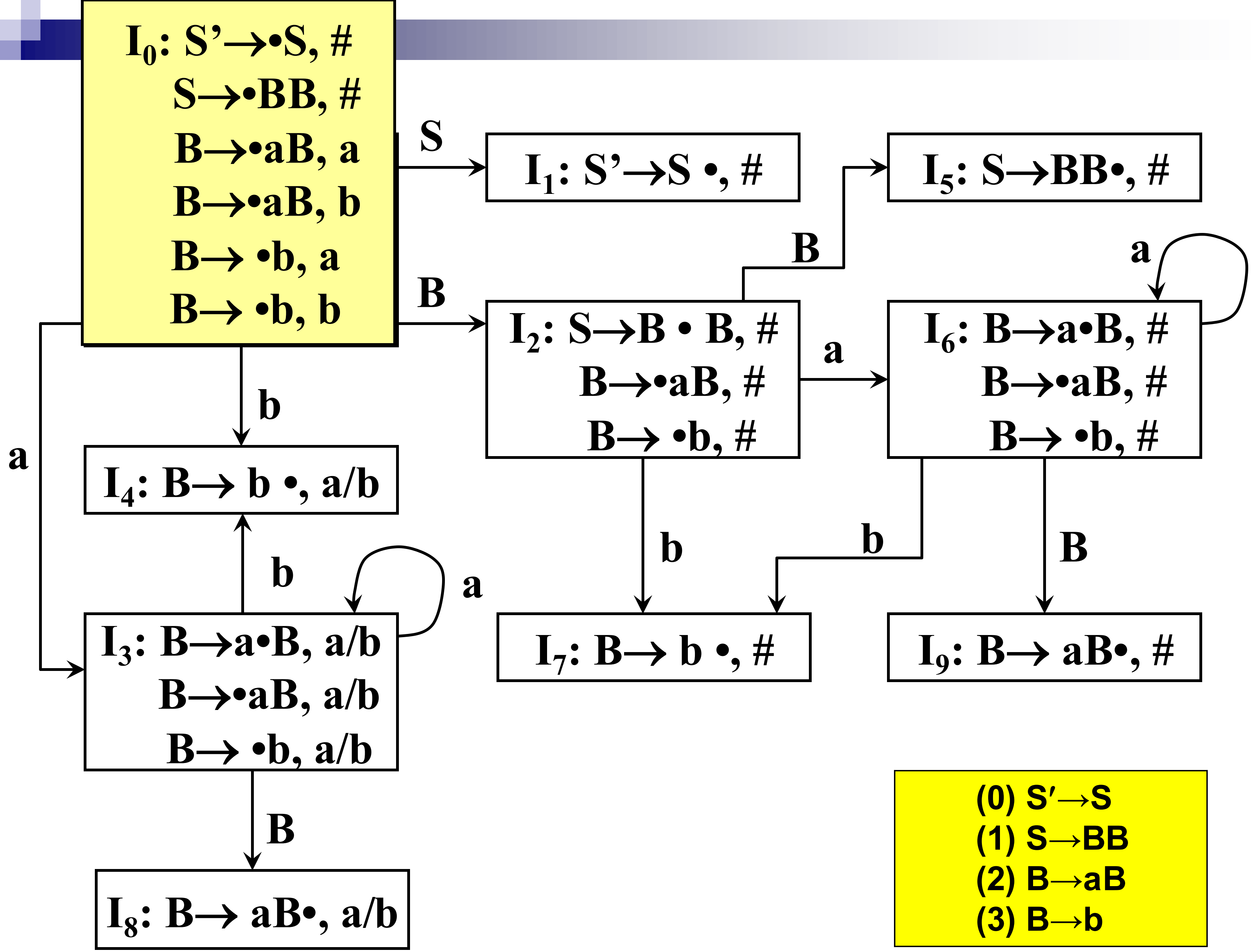
I_2: shift–reduce conflict
FOLLOW(R)={#, =}

The FOLLOW sets provide **overly** broad lookahead information !

LR(k) Analysis

- We need to **redefine items** so that each item carries **k terminal symbols**.
- **LR(k) item: [$A \rightarrow \alpha \cdot \beta$, $a_1 a_2 \ldots a_k$]**
- **lookahead string** : **$a_1 a_2 \ldots a_k$**
- **Notes:**
  - The lookahead string is meaningful **only for reduction items [$A \rightarrow \alpha \cdot$, $a_1 a_2 \ldots a_k$]**.
  - For any **shift or pending items [$A \rightarrow \alpha \cdot \beta$, $a_1 a_2 \ldots a_k$]** with **$\beta \neq \varepsilon$**, the lookahead string has **no effect**.

*



(0) $S' \to S$
(1) $S \to BB$
(2) $B \to aB$
(3) $B \to b$

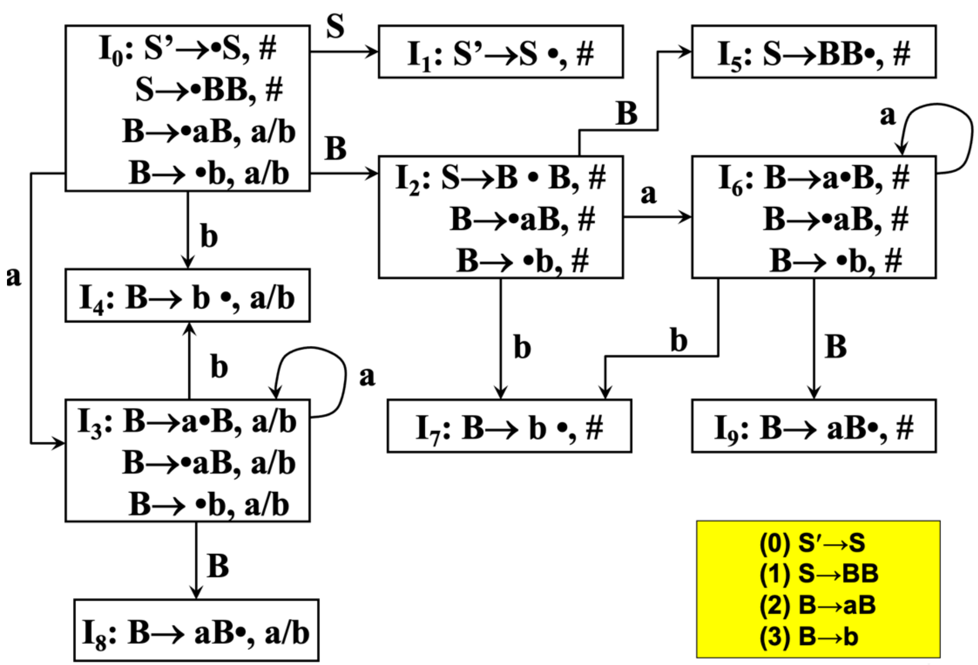
*

Construction for LR(1) parsing table

- If $[A \rightarrow \alpha \cdot a\beta, b] \in I_k$ and $GO(I_k, a) = I_j$, a terminal $\rightarrow$ $ACTION[k, a] = s_j$ (shift)
- If $[A \rightarrow \alpha \cdot, a] \in I_k \rightarrow ACTION[k, a] = r_j$ (j = production number in G')
- If $[S' \rightarrow S \cdot, \#] \in I_k \rightarrow ACTION[k, \#] = acc$ (accept)
- If $GO(I_k, A) = I_j$, A nonterminal $\rightarrow GOTO[k, A] = j$
- Any table entry not filled by rules 1–4 $\rightarrow$ error

*

The LR(1) parsing table



(0) S′→S
(1) S→BB
(2) B→aB
(3) B→b

	ACTION			GOTO	
状态	a	b	#	S	B
0	s3	s4		1	2
1			acc		
2	s6	s7			5
3	s3	s4			8
4	r3	r3			
5			r1		
6	s6	s7			9
7			r3		
8	r2	r2			
9			r2		

The LR(1) parsing table

(0) S′→S

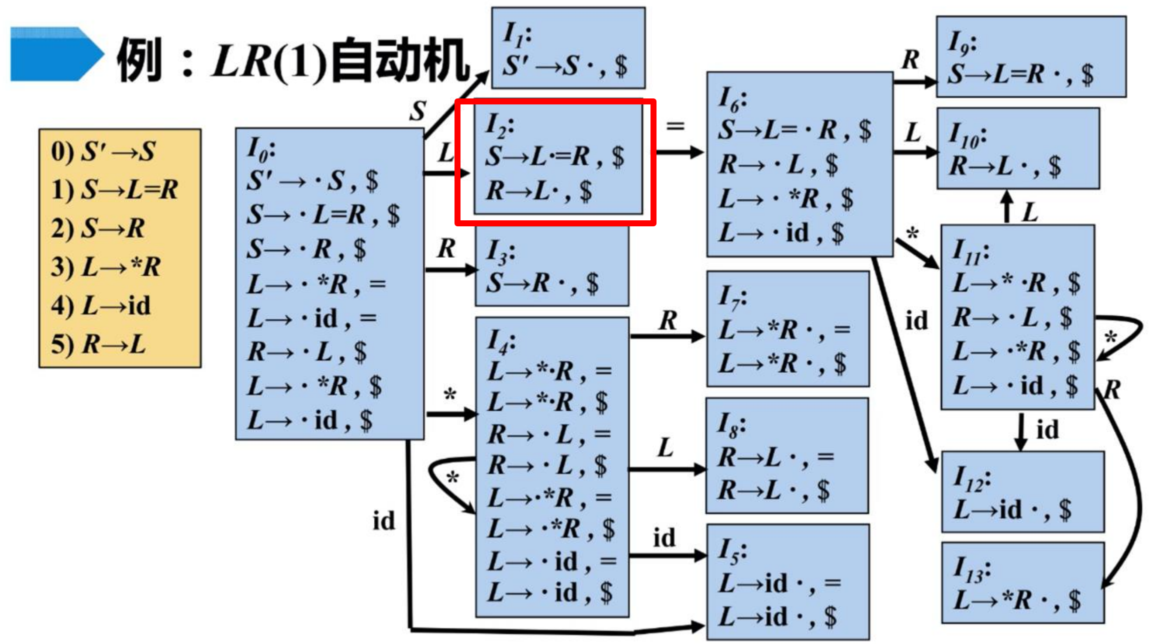
(1) S→L=R

(2) S→R

(3) L→*R

(4) L→id

(5) R→L



*

Conclusion

- If the parsing table has **no conflicts**, it is a **canonical LR(1) table**.
- A parser using it is a **canonical LR parser**.
- Such a grammar is an **LR(1) grammar**.
- LR(1) usually has more states than SLR.

LALR

Constructing an LALR(1) parsing table

- Construction of LALR parsing table / Method:
  - Combine LR(1) states with the same LR(0) core
- Reason for studying LALR:
  - Canonical LR tables have too many states
- Features of LALR:
  - Same number of states as SLR, much smaller than canonical LR tables
  - Power is between SLR and canonical LR
  - In many cases, LALR is sufficient

Key features of LALR

1, LALR parsing tables have the same number of states as SLR tables

2, Merging compatible states does not introduce new shift–reduce conflicts, but may introduce new reduce–reduce conflicts

3, On errors, the parser may perform some extra reductions, but no extra shifts

Key features of LALR-2

Merging core-identical item sets may cause conflicts.

- Such merging **does not introduce new shift–reduce conflicts**.

After Merging

$[A \rightarrow \alpha \cdot, a]$
$[B \rightarrow \beta \cdot a\gamma, ?]$
…

Before Merging

$[A \rightarrow \alpha \cdot, ?]$
$[B \rightarrow \beta \cdot a\gamma, ?]$
……

$[A \rightarrow \alpha \cdot, ?]$
$[B \rightarrow \beta \cdot a\gamma, ?]$
……

…

$[A \rightarrow \alpha \cdot, ?]$
$[B \rightarrow \beta \cdot a\gamma, ?]$
……

K item sets

Key features of LALR-2

Merging may introduce new **reduce–reduce** conflicts.

$$S' \rightarrow S$$
$$S \rightarrow aAd \mid bBd \mid aBe \mid bAe$$
$$A \rightarrow c$$
$$B \rightarrow c$$

Before Merging

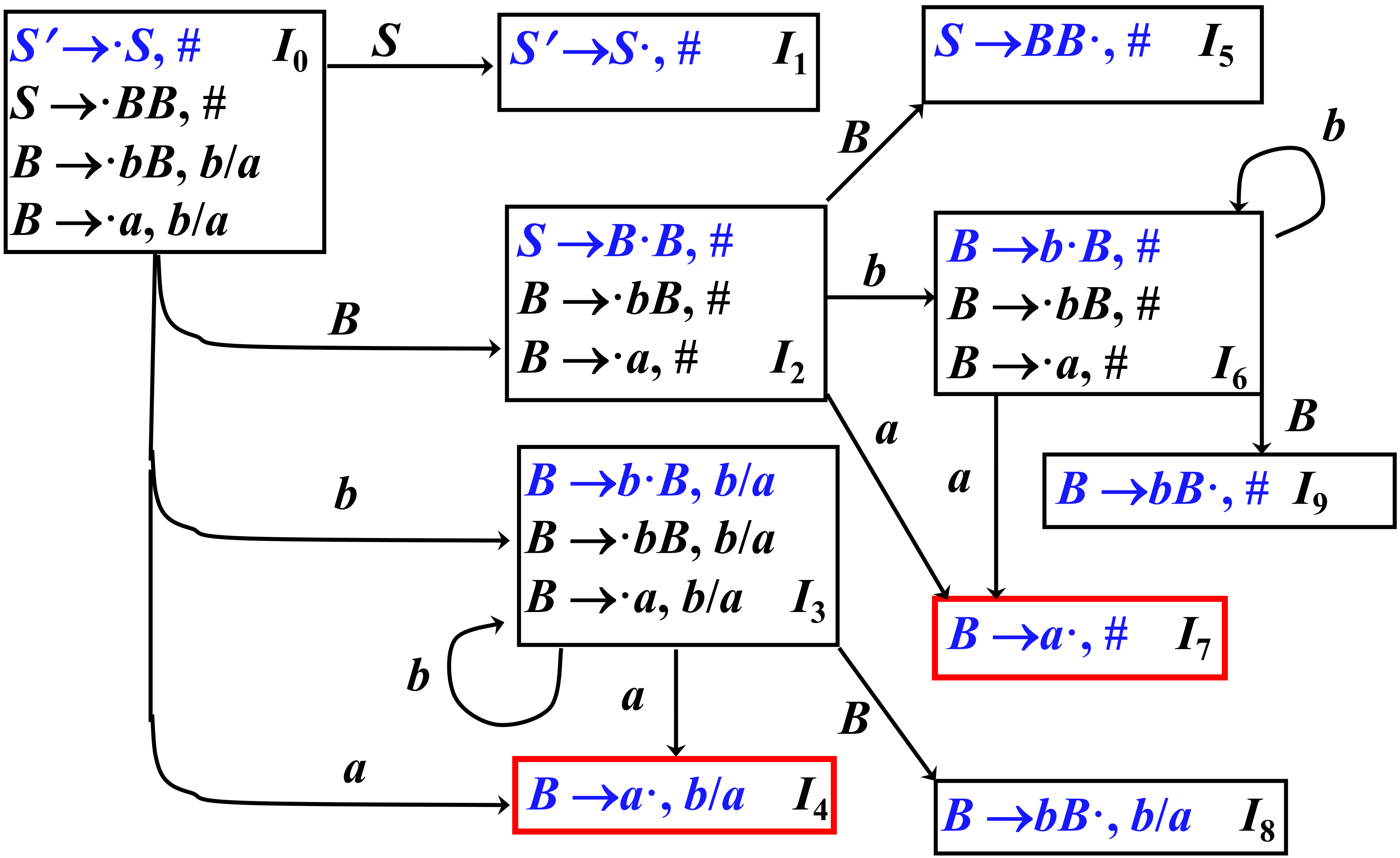
$$\boxed{\begin{array}{l} A \rightarrow c\cdot, d \\ B \rightarrow c\cdot, e \end{array}} \qquad \boxed{\begin{array}{l} A \rightarrow c\cdot, e \\ B \rightarrow c\cdot, d \end{array}}$$

After Merging

$$\boxed{\begin{array}{l} A \rightarrow c\cdot, d/e \\ B \rightarrow c\cdot, d/e \end{array}}$$

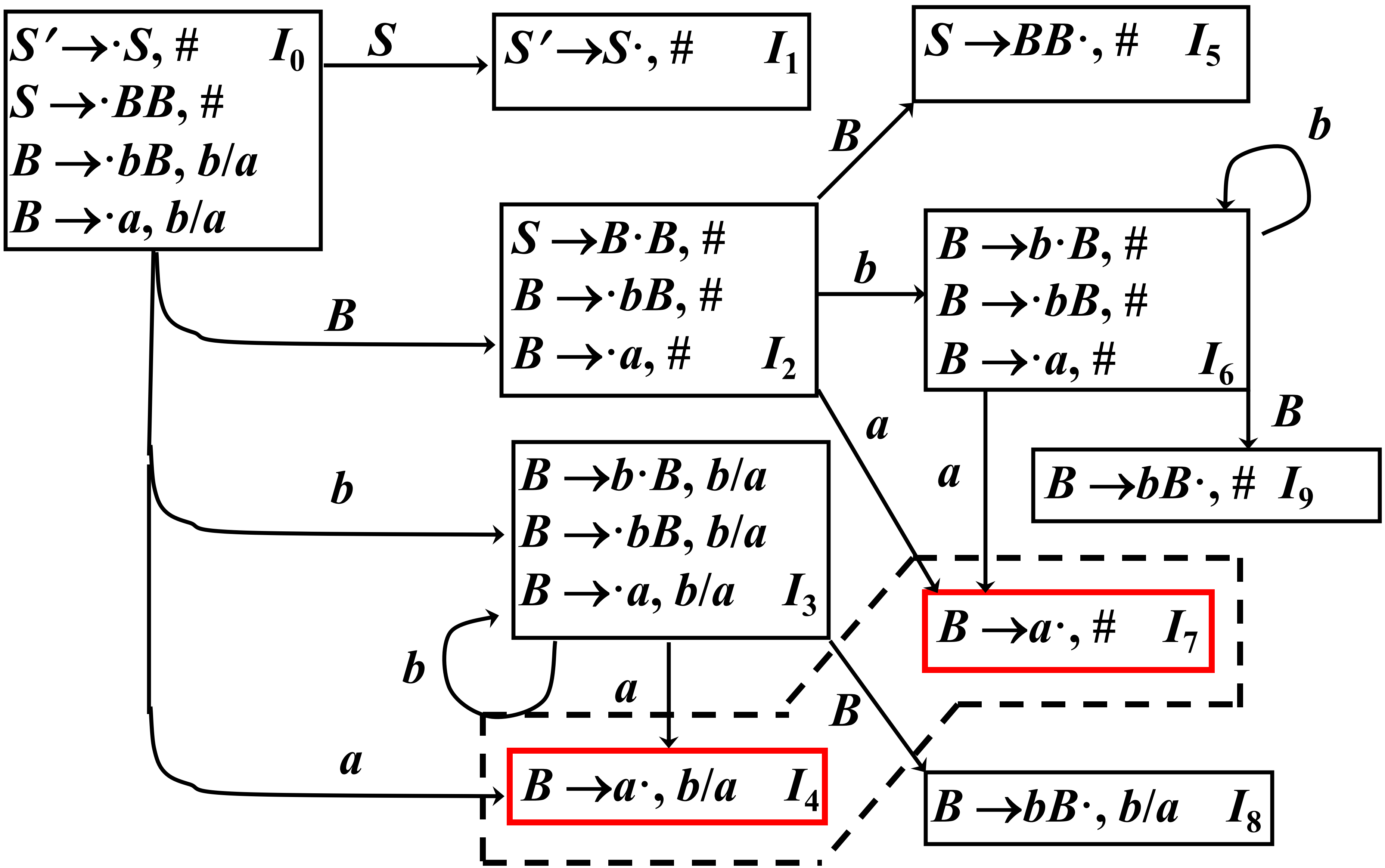
LR(1): YES
LALR(1): NO

Key features of LALR-3

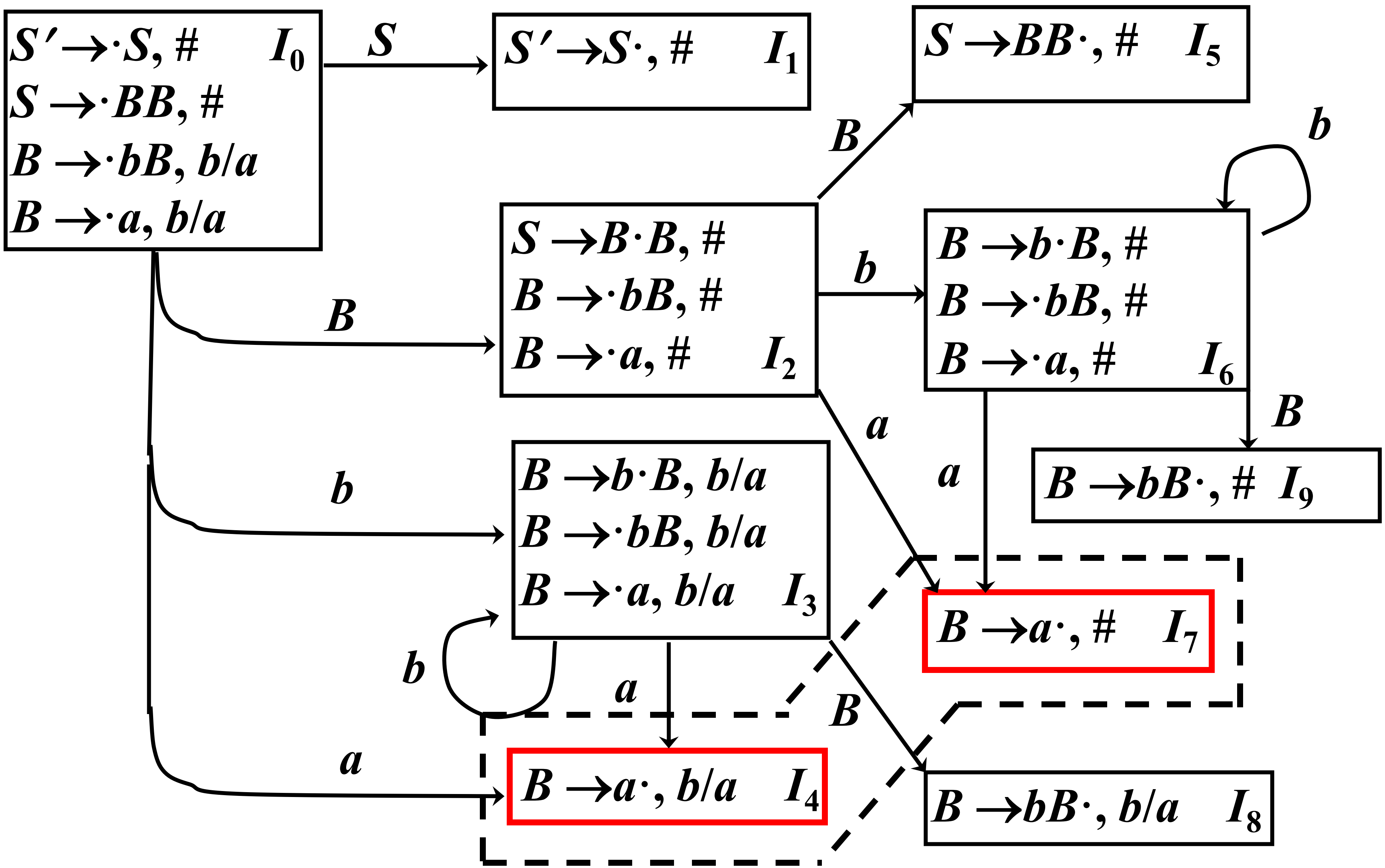


Key features of LALR-3

Merge I4,I7

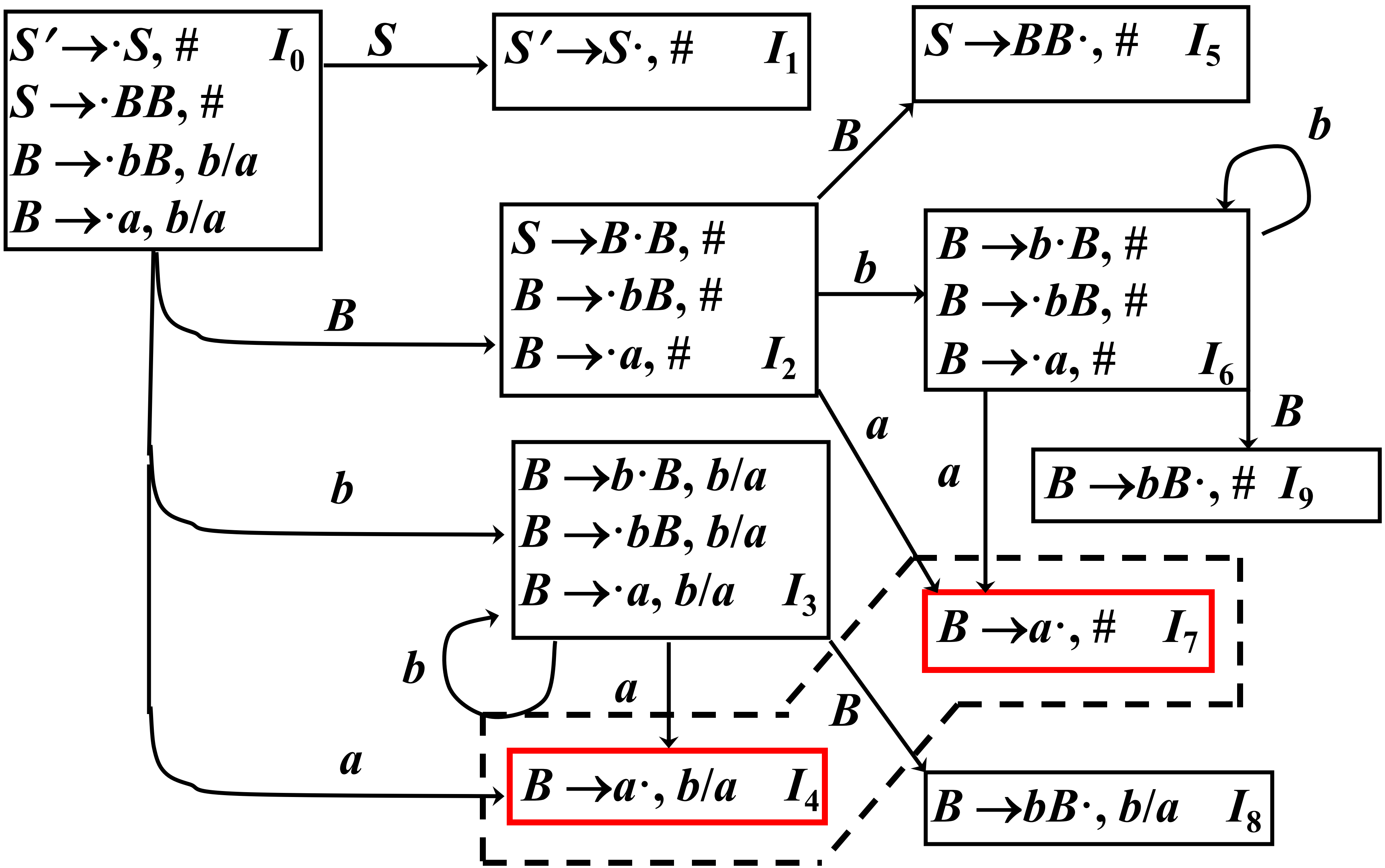


Key features of LALR-3 *bbabba#* **accepted**



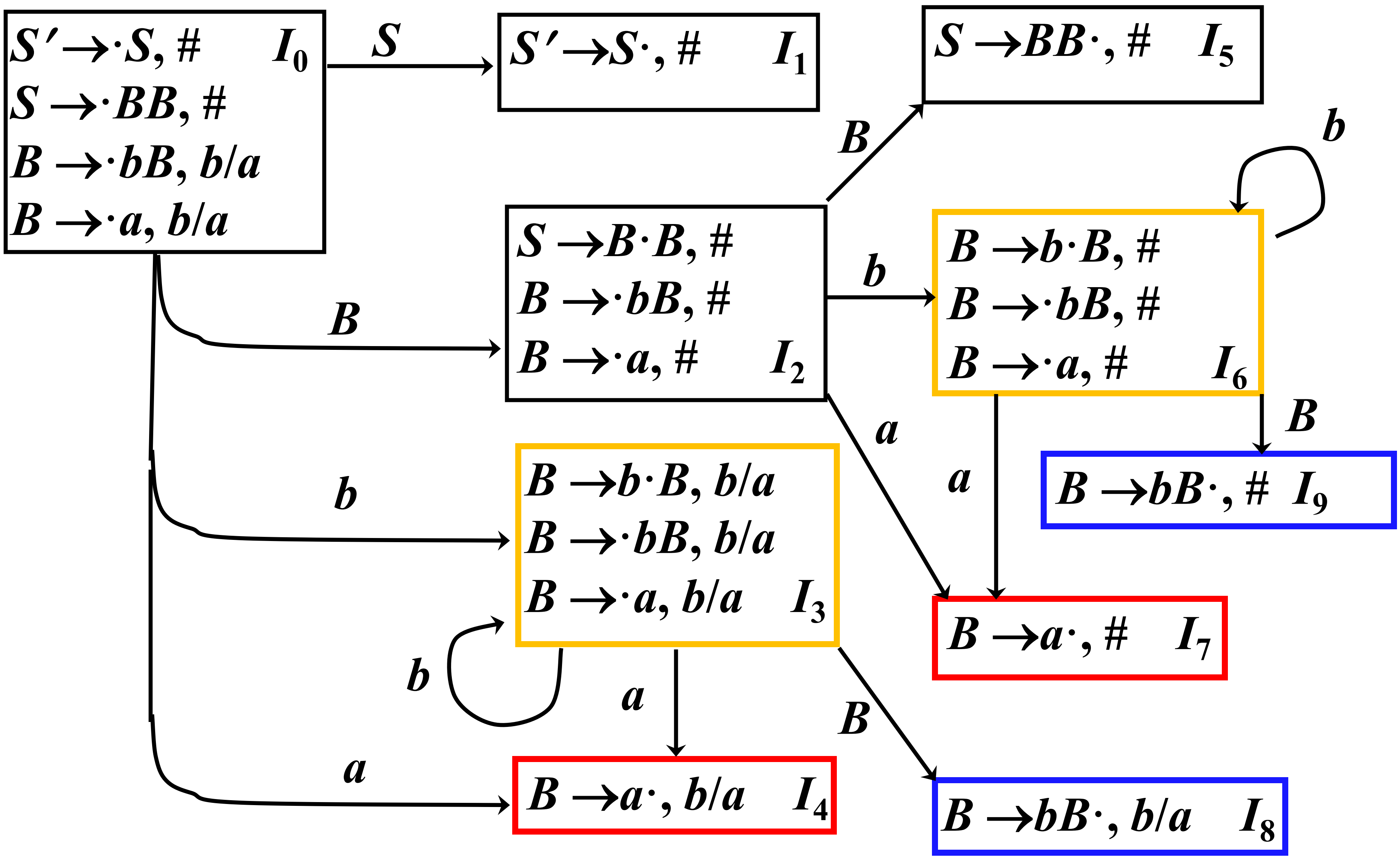
Key features of LALR-3

bba# **Error**



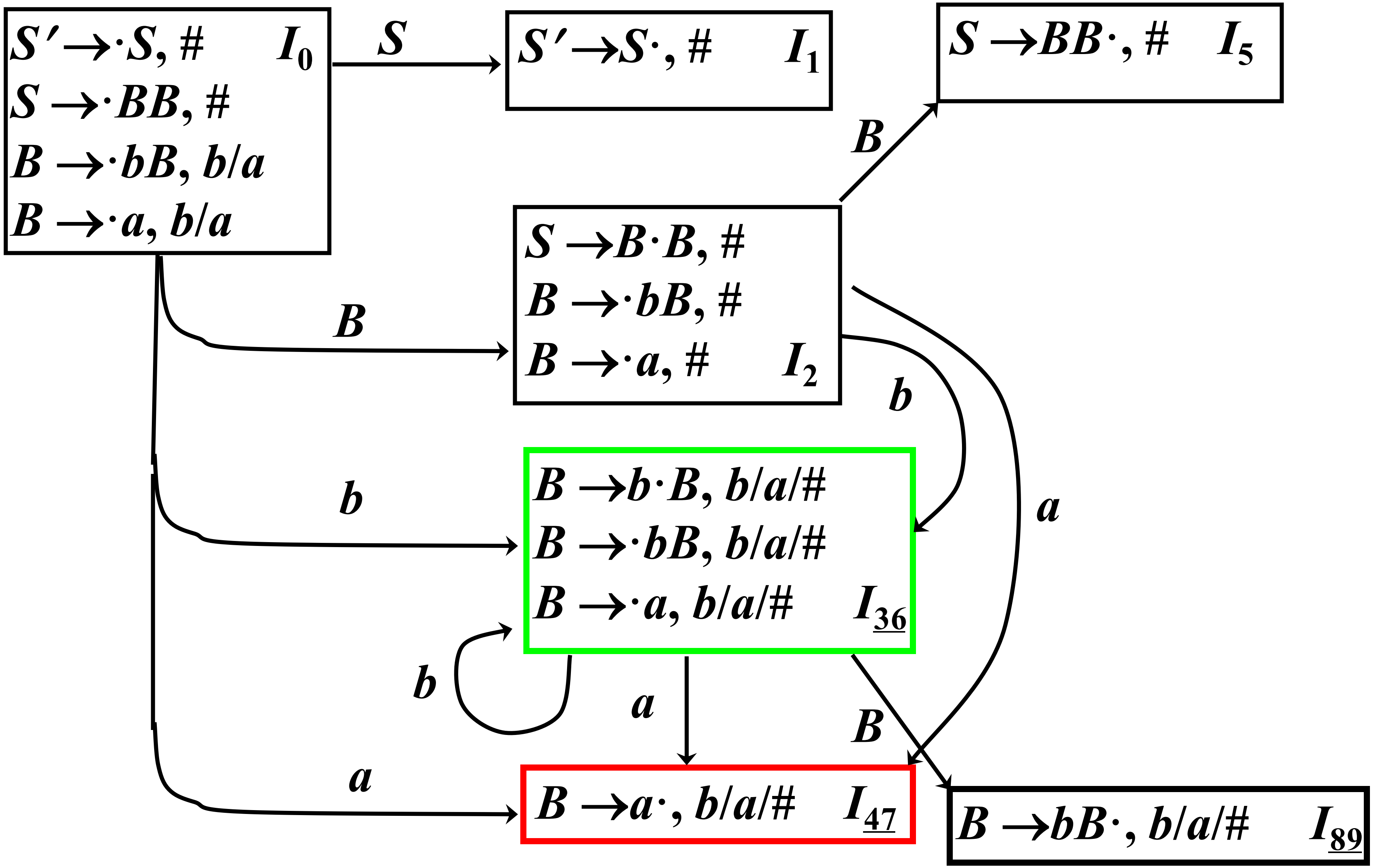
Key features of LALR-3

Merging



Key features of LALR-3

bba# ?



Conclusion

- **Bottom-up parsing methods:**
  - **LR(0) method**
  - **SLR(1) method**
  - **Canonical LR(1) method**
  - **LALR(1) method**
- **LR parsing program**

Construction for LR(0) parsing table

- If $\mathbf{A} \rightarrow \boldsymbol{\alpha \cdot a \beta} \in \mathbf{I_k}$ and $\mathbf{GO(I_k, a) = I_j}$, a terminal $\rightarrow$ $\mathbf{ACTION[k, a] = s_j}$ (shift)
- $\mathbf{A} \rightarrow \boldsymbol{\alpha \cdot} \in \mathbf{I_k} \rightarrow \mathbf{ACTION[k, a] = r_j}$ **for all terminals a (including #)**
- If $\mathbf{S'} \rightarrow \mathbf{S \cdot} \in \mathbf{I_k} \rightarrow \mathbf{ACTION[k, \#] = acc}$
- If $\mathbf{GO(I_k, A) = I_j}$, A nonterminal $\rightarrow$ $\mathbf{GOTO[k, A] = j}$
- Any table entry not filled by rules 1–4 $\rightarrow$ **error**

*

Construction for SLR(1) parsing table

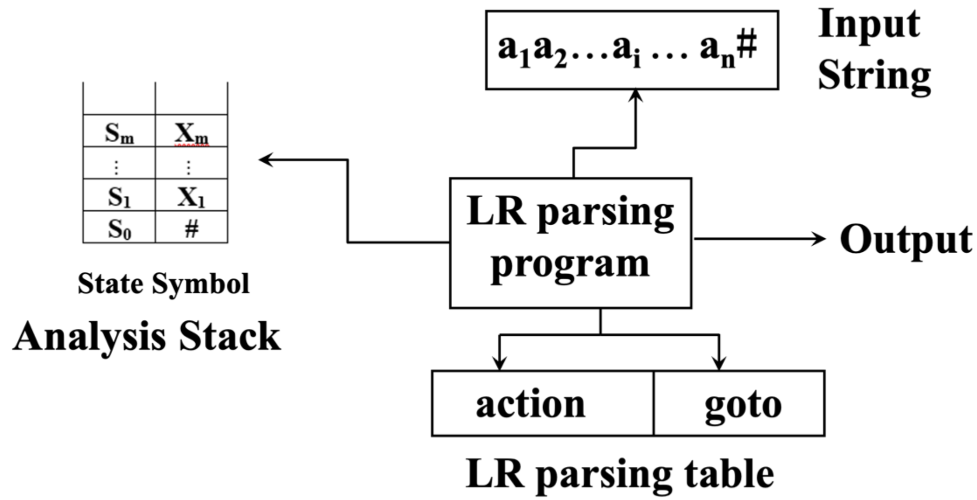
- If $\mathbf{A} \rightarrow \boldsymbol{\alpha}\cdot\mathbf{a}\boldsymbol{\beta} \in \mathbf{I_k}$ and $\mathbf{GO(I_k, a) = I_j}$, a terminal $\rightarrow$ $\mathbf{ACTION[k, a] = s_j}$ (shift)
- If $\mathbf{A} \rightarrow \boldsymbol{\alpha}\cdot \in \mathbf{I_k}$, then for each $\mathbf{a} \in$ **FOLLOW(A)** $\rightarrow$ $\mathbf{ACTION[k, a] = r_j}$ (j = production number in G′)
- If $\mathbf{S'} \rightarrow \mathbf{S}\cdot \in \mathbf{I_k} \rightarrow$ **ACTION[k, #] = acc**
- If $\mathbf{GO(I_k, A) = I_j}$, A nonterminal $\rightarrow$ **GOTO[k, A] = j**
- Any table entry not filled by rules 1–4 $\rightarrow$ **error**

*

Construction for LR(1) parsing table

- If $[A \to \alpha \cdot a\beta, b] \in I_k$ and $GO(I_k, a) = I_j$, a terminal $\to$ ACTION[k, a] = s_j (shift)
- If $[A \to \alpha \cdot, a] \in I_k \to$ ACTION[k, a] = r_j (j = production number in G′)
- If $[S' \to S \cdot, \#] \in I_k \to$ ACTION[k, #] = acc (accept)
- If $GO(I_k, A) = I_j$, A nonterminal $\to$ GOTO[k, A] = j
- Any table entry not filled by rules 1–4 $\to$ error

LR Parsing Program




```
let a be the first symbol of w$;
while(1) { /* repeat forever */
        let s be the state on top of the stack;
        if ( ACTION[s, a] = shift t ) {
                push t onto the stack;
                let a be the next input symbol;
        } else if ( ACTION[s, a] = reduce A → β ) {
                pop |β| symbols off the stack;
                let state t now be on top of the stack;
                push GOTO[t, A] onto the stack;
                output the production A → β;
        } else if ( ACTION[s, a] = accept ) break; /* parsing is done */
        else call error-recovery routine;
}
```

Figure 4.36: LR-parsing program

Book: Compilers Principles Techniques and Tools (2nd Edition) P251

*

Conclusion

- **Differences:**
  - **LR(0):** Reduces **without looking** at stack contents or input (no lookahead).
  - **SLR:** Reduces by checking **only the next input symbol** (via FOLLOW set).
  - **LR(1):** Reduces by considering **both stack contents and 1 lookahead symbol**.
  - **LR(k):** Reduces by considering **both stack contents and k lookahead symbols**.

CONSTRUCTION OF AUTOMATA

Method 1: NFA

- Method 1: NFA for viable prefixes
  - Start state: item 1; all other states are accepting.
  - Rule 1: If "·" moves over a symbol **Xi**, add an edge labeled **Xi**. ($\mathbf{X \to X_1 \cdots X_{i-1} \cdot X_i \cdots X_n}$)
  - Rule 2: If after "·" is a nonterminal **A**, add ε-edges to items of **A**. ($\mathbf{X \to \alpha \cdot A\beta, A \to \cdot \gamma}$)
  - Then, convert the NFA to DFA.

*

1. S′→ ·E	2. S′→E ·
3. E→ ·aA	4. E→a ·A
5. E→aA ·	6. A→ ·cA
7. A→c ·A	8. A→cA ·
9. A→ ·d	10. A→d ·
11. E→ ·bB	12. E→b ·B
13. E→bB ·	14. B→ ·cB
15. B→c ·B	16. B→cB ·
17. B→ ·d	18. B→d ·

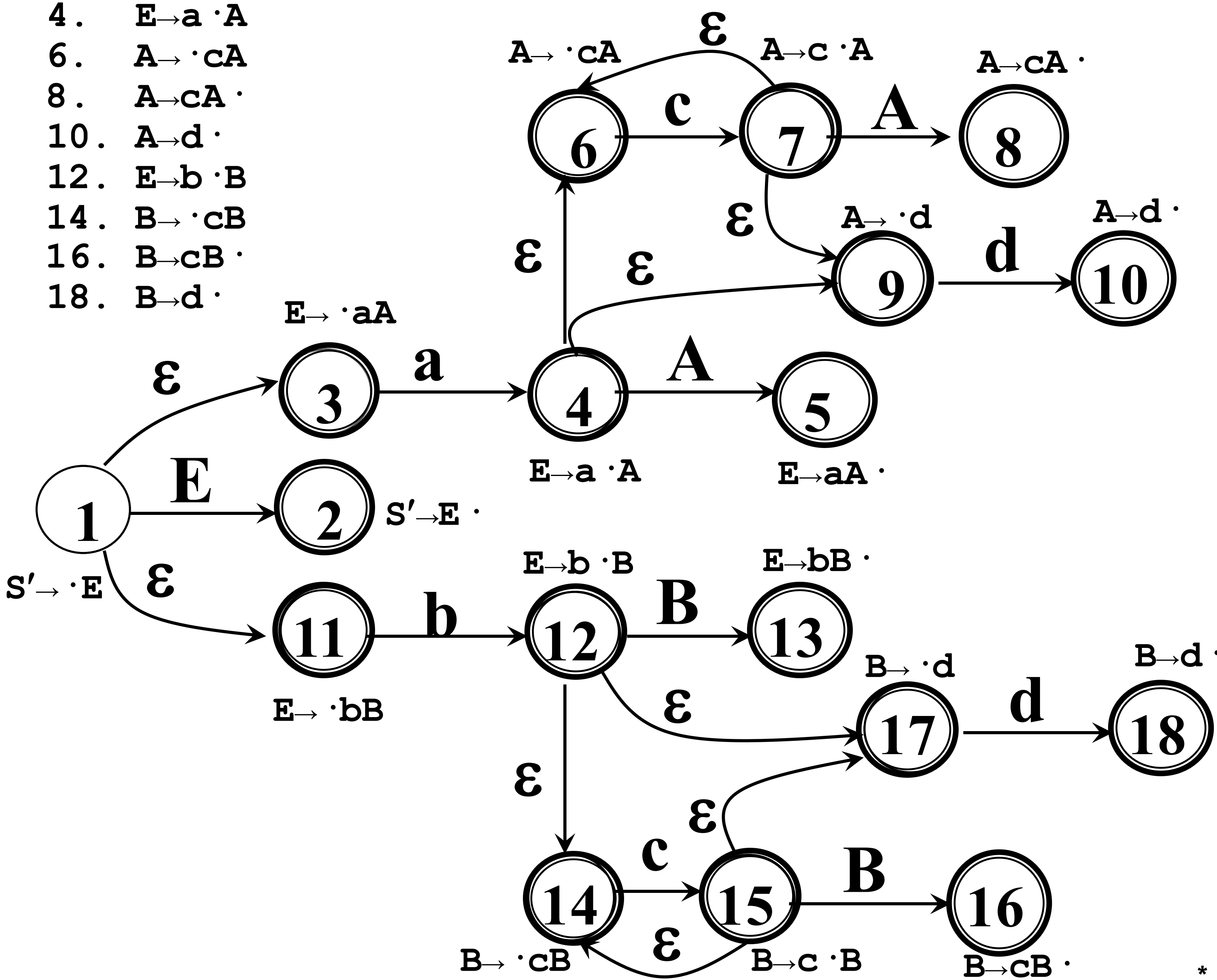
G(S′)

S′→E

E→aA|bB

A→cA|d

B→cB|d



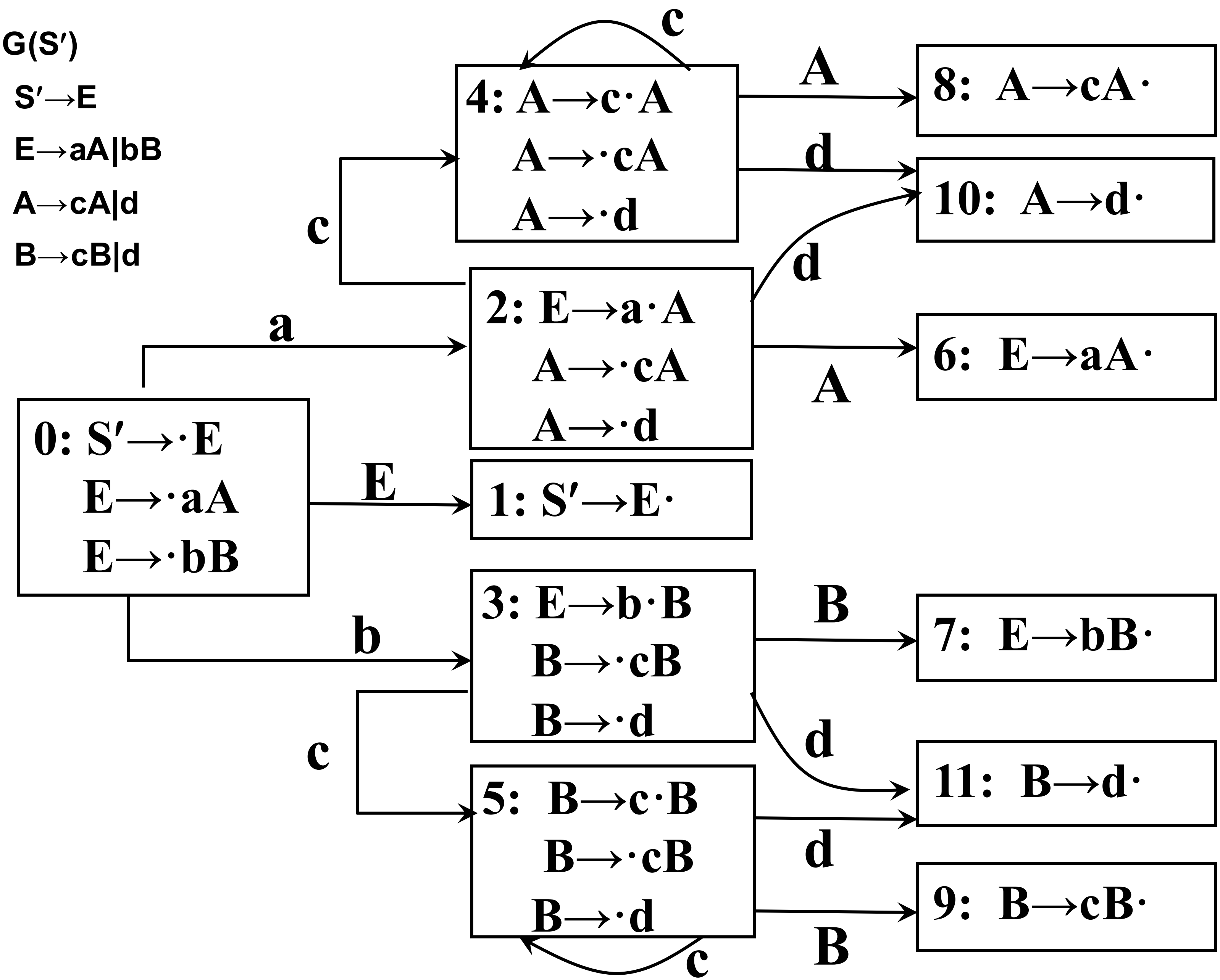
G(S′)

S′→E

E→aA|bB

A→cA|d

B→cB|d



*

Method 2: Canonical LR（0）Collection

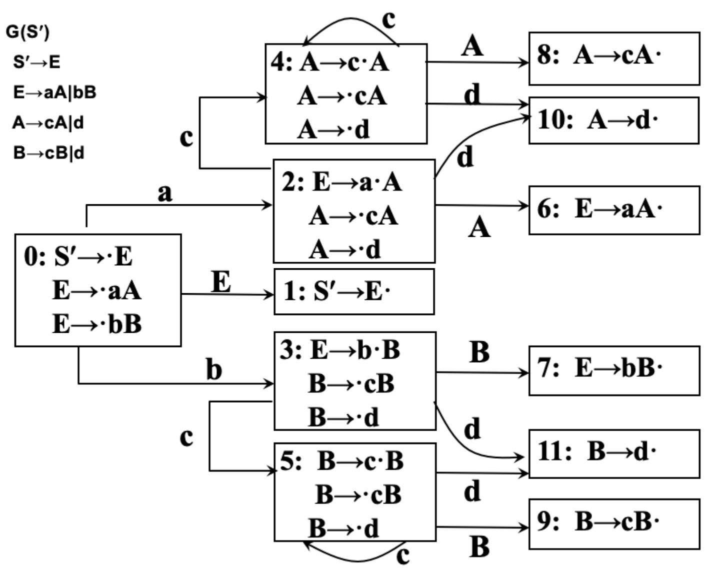
- The complete set of item sets (states) forming a DFA that recognizes all viable prefixes of a grammar is called the **canonical LR(0) collection** of the grammar.
  - **A → α·** is called a **reduction item**
  - **S′ → α·** is called an **accepting item**
  - **A → α· aβ** (a ∈ VT) is called a **shift item**
  - **A → α· Bβ** (B ∈ VN) is called a **pending (goto) item**

Augment Grammar

- Assume G is a grammar with start symbol S. We construct G′ as follows:
  - G′ includes all of G.
  - Add a new nonterminal S′ (not in G), with S′ as the start symbol.
  - Add the production S′ → S.
- G′ is the augmented grammar of G, and it has the accepting state S′ → S·.

The **Closure** of an item set **I**

- **CLOSURE(I)**, is defined and constructed as follows:
  - **All items in I are included in CLOSURE(I).**
  - **If A → α·Bβ is in CLOSURE(I), then for every production B → γ, the item B → ·γ is added to CLOSURE(I).**
  - **Repeat steps 1 and 2 until CLOSURE(I) no longer increases.**

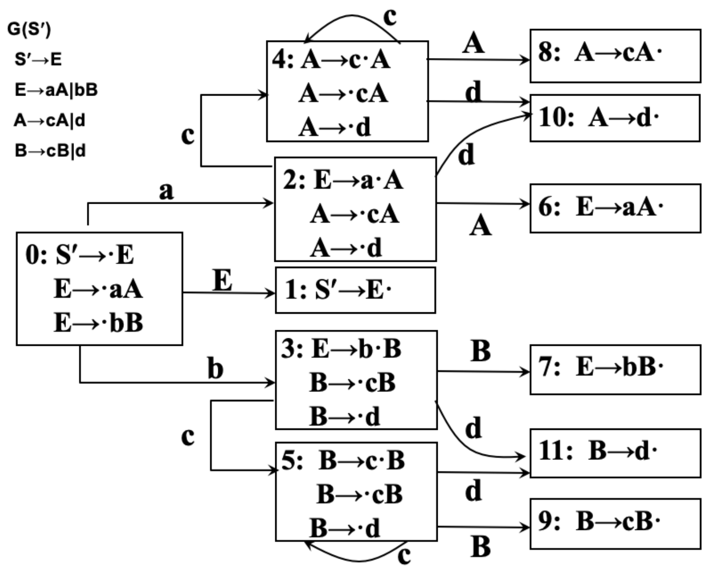


*

State transition function GO

- **GO** is a state transition function. Let **I** be an item set and **X** be a grammar symbol. The value of the function **GO(I, X)** is defined as:
  - $GO(I, X) = CLOSURE(J)$

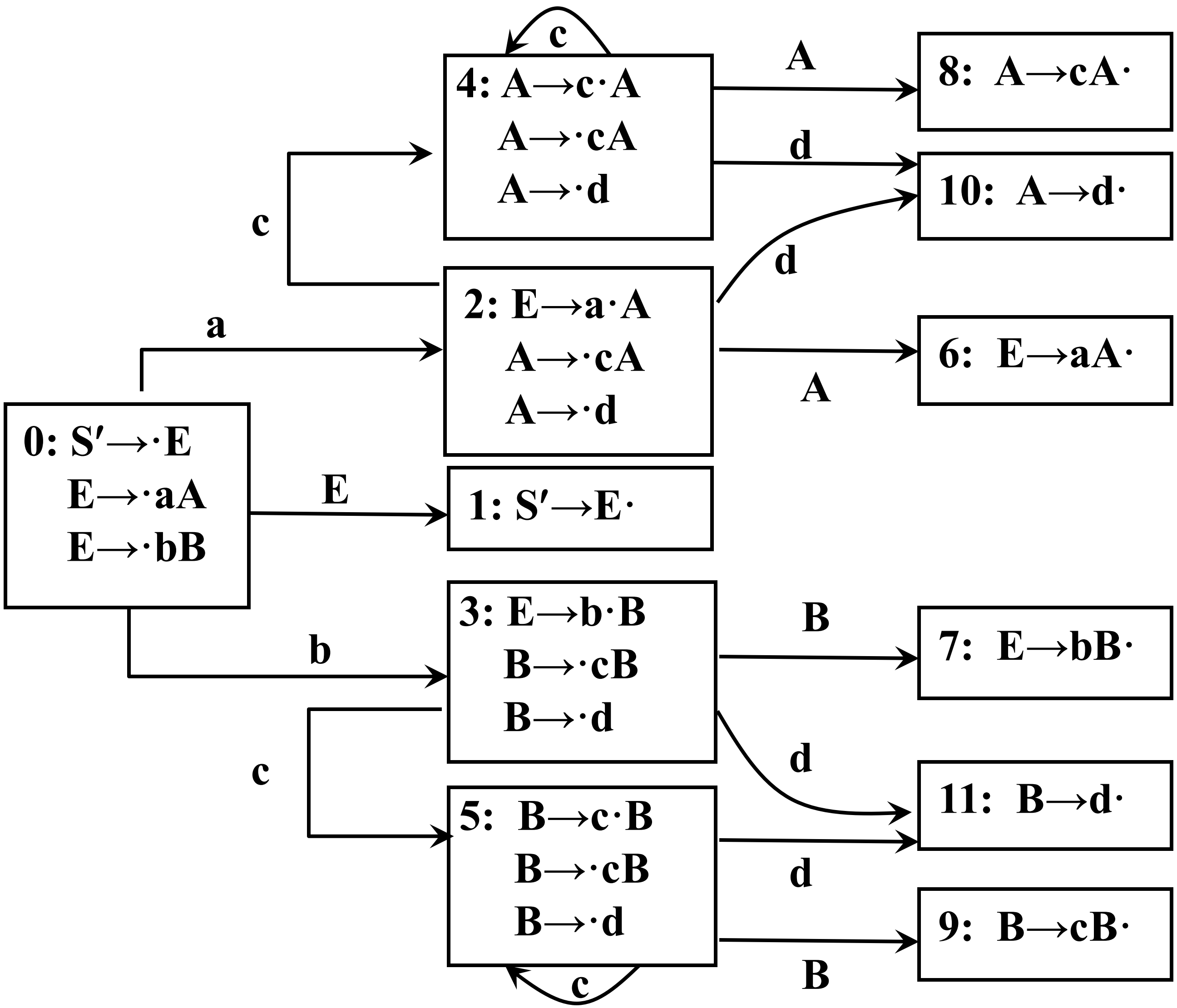
where J
$= \{any\ item\ of\ the\ form\ \cdot A \cdot\!\to \alpha X\beta \mid A \to \alpha X\beta \in I\}$



Construction of Canonical LR(0) Collection

```
BEGIN
    C := {CLOSURE({S′ → ·S})};
    REPEAT
        FOR each item set I in C and each symbol X in G′ DO
            IF GO(I, X) is non-empty and not in C THEN
                Add GO(I, X) to the collection C;
    UNTIL C no longer grows
END
```

The transition function **GO** connects the item sets into a DFA transition graph.


4: A→c·A
A→·cA
A→·d
8: A→cA·
10: A→d·
2: E→a·A
A→·cA
A→·d
6: E→aA·
0: S′→·E
E→·aA
E→·bB
1: S′→E·
3: E→b·B
B→·cB
B→·d
7: E→bB·
11: B→d·
5: B→c·B
B→·cB
B→·d
9: B→cB·
a
b
c
d
A
B
E

*

Ambiguous Grammar

Ambiguous Grammar

- **Ambiguous Grammar Characteristics:**
  - **Not LR grammars**
  - **Concise** and **natural**
  - Ambiguity can be eliminated with **additional information**
  - **Higher parsing efficiency** after disambiguation

Ambiguous $E \rightarrow E + E \mid E * E \mid (E) \mid \mathbf{id}$

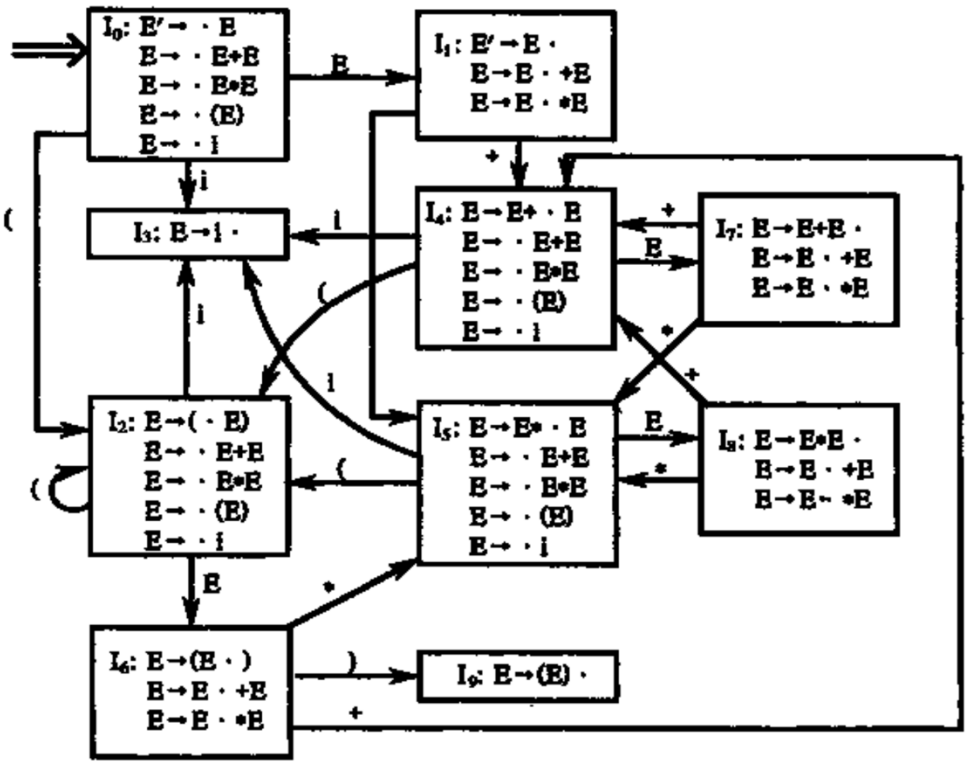
Non-ambiguous

$$E \rightarrow E + T \mid T$$
$$T \rightarrow T * F \mid F$$
$$F \rightarrow (E) \mid \mathbf{id}$$

Ambiguous Grammar

Using **information** beyond the grammar to resolve parsing action conflicts

$$E \rightarrow E + E \mid E * E \mid (E) \mid \mathbf{id}$$



Using **information** beyond the grammar

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

Rule: * has higher precedence than +, both are left-associative.

I_7 （P124 figure 5.11）

$E \rightarrow E + E\cdot$

$E \rightarrow E\cdot + E$ **id + id** **+ id** **Facing +，？**

$E \rightarrow E\cdot * E$ **id + id** **∗ id** **Facing *，？**

Exercise

How about the following item set?

LR(0) item set I_8

$$
\boxed{\begin{array}{l} E \rightarrow E * E\cdot \\ E \rightarrow E\cdot + E \\ E \rightarrow E\cdot * E \end{array}}
$$

Outline

- Bottom-up Parsing Methods
  - Basic issues in bottom-up parsing
  - Canonical reduction
  - Operator-precedence parsing
  - LR parsing methods:
    - LR(0) method
    - SLR(1) method
    - Canonical LR(1) method
    - LALR(1) method
  - Applying LR methods to ambiguous grammars

Quiz-Canvas

- **ch5 Syntax Analysis - LR Parsing Table**

Dank u
Dutch

Merci
French

Спасибо
Russian

Gracias
Spanish

شكراً
Arabic

धन्यवाद
Hindi

감사합니다
Korean

Tack så mycket
Swedish

תודה רבה
Hebrew

Obrigado
Brazilian Portuguese

Dankon
Esperanto

Thank You !

谢谢
Chinese

ありがとうございます
Japanese

Trugarez
Breton

Danke
German

Tak
Danish

Grazie
Italian

நன்றி
Tamil

děkuji
Czech

ขอบคุณ
Thai

go raibh maith agat
Gaelic

*