LECTURE WEEK
08-10

Bottom-up Analysis

COMPILER
CONSTRUCTION

Basic Concepts

- Sentential form
  - For a grammar G with start symbol S
    A string α is a sentential form of G if $S \Rightarrow^* \alpha$
    - α may contain terminals and nonterminals
    - If α is in T^*, then α is a sentence of L(G)
  - Left sentential form: A sentential form that occurs in the leftmost derivation of some sentence
  - Right sentential form: A sentential form that occurs in the rightmost derivation of some sentence

Basic Concepts

- Example of the sentential form
  - $E \to E * E \mid E + E \mid (E) \mid id$
  - Leftmost derivation:

    $E \Rightarrow E + E \Rightarrow E * E + E \Rightarrow id * E + E \Rightarrow id * id + E \Rightarrow$

    $id * id + E * E \Rightarrow id * id + id * E \Rightarrow id * id + id * id$

    - All the derived strings are of the left sentential form
  - Rightmost derivation

    $E \Rightarrow E + E \Rightarrow E + E * E \Rightarrow E + E * id \Rightarrow E + id * id \Rightarrow$

    $E * E + id * id \Rightarrow E * id + id * id \Rightarrow id * id + id * id$

    - All the derived strings are of the right sentential form

A Small example

$E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$

$F \rightarrow (E) \mid id$

A Rightmost Derivation:

$E \rightarrow \underline{E + T}$

$\rightarrow E + \underline{T * F}$

$\rightarrow E + T * \underline{id}$

$\rightarrow E + \underline{F} * id$

$\rightarrow E + \underline{id} * id$

$\rightarrow \underline{T} + id * id$

$\rightarrow \underline{F} + id * id$

$\rightarrow \underline{id} + id * id$

The Parsing Problem

- Given a right sentential form, α, determine what substring of α is the **right-hand side (RHS)** of the rule in the grammar that must be reduced to produce the previous sentential form in the right derivation
- The correct RHS is called the <u>handle</u>

Basic Concepts

Informally, a **handle of a string is a substring that matches the right side of a production rule**.

But not every substring matches the right side of a production rule is handle

Reduction of a handle represents one step along the reverse of a rightmost derivation

The leftmost substring is the handle

If the grammar is unambiguous, then every right-sentential form of the grammar has exactly one handle.

Handles

- The **handle** of a parse tree *T* is the leftmost complete cluster of leaf nodes.
- A left-to-right, bottom-up parser works by iteratively searching for a handle, then reducing the handle.

Basic Concepts

RIGHT SENTENTIAL FORM	HANDLE	REDUCING PRODUCTION
$\mathbf{id}_1 * \mathbf{id}_2$	$\mathbf{id}_1$	$F \to \mathbf{id}$
$F * \mathbf{id}_2$	F	$T \to F$
$T * \mathbf{id}_2$	$\mathbf{id}_2$	$F \to \mathbf{id}$
$T * F$	$T * F$	$E \to T * F$

Figure 4.26: Handles during a parse of $\mathbf{id}_1 * \mathbf{id}_2$

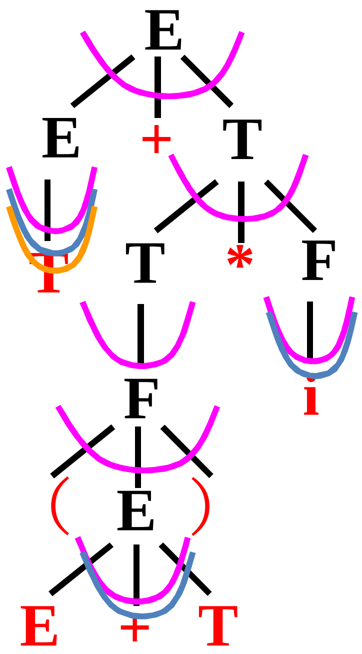
$E \to T * F$

$T \to F$

$F \to id$

Basic Concepts

- Illustration via Parse Tree



<u>Sentential form</u>: leave nodes (from left to right)
T+ (E+T)*i

<u>Phrases: leave nodes of each subtree</u>
T+ (E+T)*i 、T、(E+T)*i、(E+T)、E+T、i

<u>Simple phrase: leave nodes of all simple subtree</u>
(i.e. a subtree with only one level of leaves)
T、E+T、i

<u>Handle: leave nodes of the leftmost simple subtree</u>
T

LR Parsing

- Produce a parse tree starting at the leaves
- The order will be the reverse of a rightmost derivation
- The most common bottom-up parsing algorithms are in the LR family

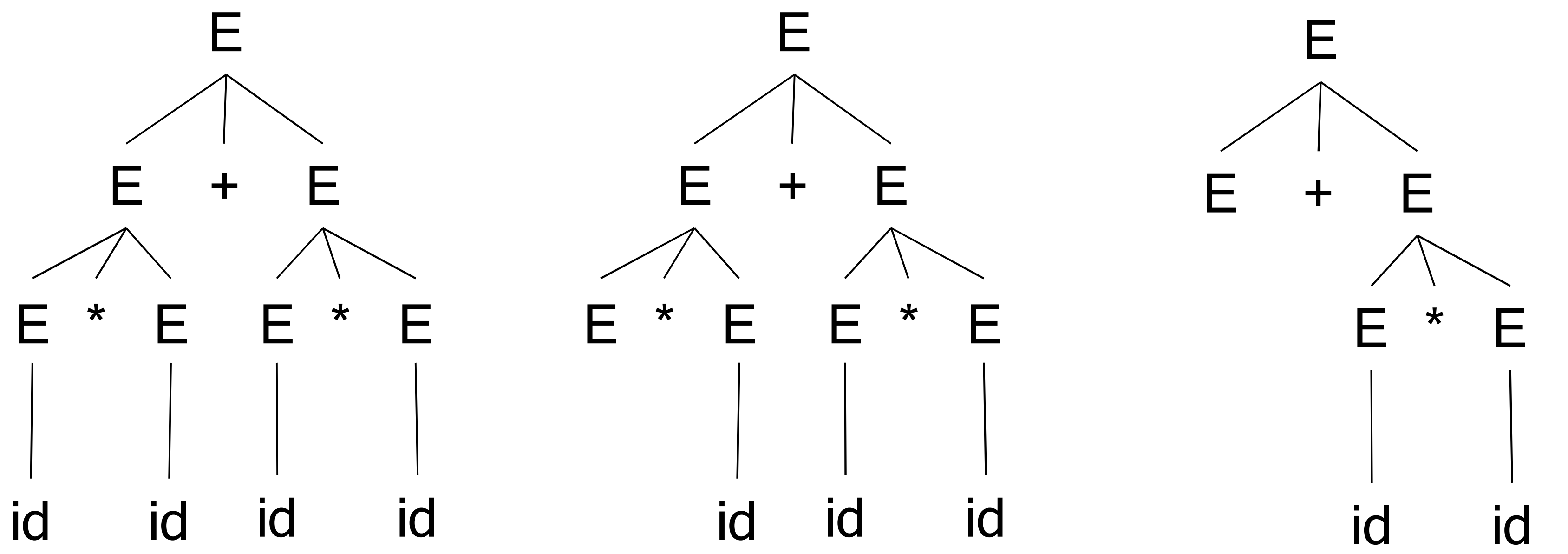
  L – Read the input left to right

  R – Trace out a rightmost parse tree

Meaning of LR

- L: Process input from left to right
- R: Use rightmost derivation, but in reversed order
- $E \Rightarrow E + E \Rightarrow E + E * E \Rightarrow E + E * id \Rightarrow E + id * id$

  $\Rightarrow E * E + id * id \Rightarrow E * id + id * id \Rightarrow id * id + id * id$



LR Parsers Use Shift-Reduce

- Shift-Reduce Algorithms
  - Reduce: replace the handle on the top of the parse stack with its corresponding LHS
  - Shift: move the next token to the top of the parse stack

STACK	INPUT	ACTION
$\$$	$\mathbf{id}_1 * \mathbf{id}_2 \$$	shift
$\$\,\mathbf{id}_1$	$* \mathbf{id}_2 \$$	reduce by $F \to \mathbf{id}$
$\$\,F$	$* \mathbf{id}_2 \$$	reduce by $T \to F$
$\$\,T$	$* \mathbf{id}_2 \$$	shift
$\$\,T *$	$\mathbf{id}_2 \$$	shift
$\$\,T * \mathbf{id}_2$	$\$$	reduce by $F \to \mathbf{id}$
$\$\,T * F$	$\$$	reduce by $T \to T * F$
$\$\,T$	$\$$	reduce by $E \to T$
$\$\,E$	$\$$	accept

Figure 4.28: Configurations of a shift-reduce parser on input $\mathbf{id}_1 * \mathbf{id}_2$

Shift/Reduce/Accept/Error

A Shift-Reduce Parser

- $E \rightarrow E+T \mid T$
- $T \rightarrow T*F \mid F$
- $F \rightarrow (E) \mid id$

Right-Most Derivation of id+id*id

$$E \Rightarrow E+T \Rightarrow E+T*F \Rightarrow E+T*id \Rightarrow E+F*id$$
$$\Rightarrow E+id*id \Rightarrow T+id*id \Rightarrow F+id*id \Rightarrow id+id*id$$

<u>Right-Most Sentential Form</u>	<u>Reducing Production</u>
<u>id</u>+id*id	reduce, $F \rightarrow id$
<u>F</u>+id*id	reduce, $T \rightarrow F$
<u>T</u>+id*id	reduce, $E \rightarrow T$
E+<u>id</u>*id	nothing to reduce, shift (twice), $F \rightarrow id$
E+<u>F</u>*id	reduce, $T \rightarrow F$
E+T*<u>id</u>	nothing to reduce, shift (twice), $F \rightarrow id$
E+<u>T*F</u>	reduce, $T \rightarrow T*F$
<u>E+T</u>	$E \rightarrow E+T$
E	

<u>Handles</u> are red and underlined in the right-sentential forms.

A Detail about Handles

$E \rightarrow F$
$E \rightarrow E + F$
$F \rightarrow F * T$
$F \rightarrow T$
$T \rightarrow \texttt{int}$
$T \rightarrow (E)$

int	+	int	*	int

A Detail about Handles

E → F

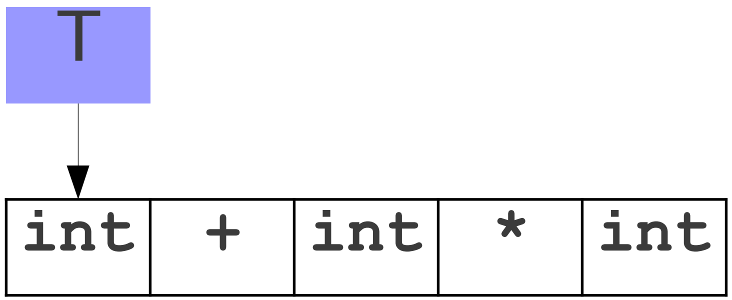
E → E + F

F → F * T

F → T

T → int

T → (E)



A Detail about Handles

E → F

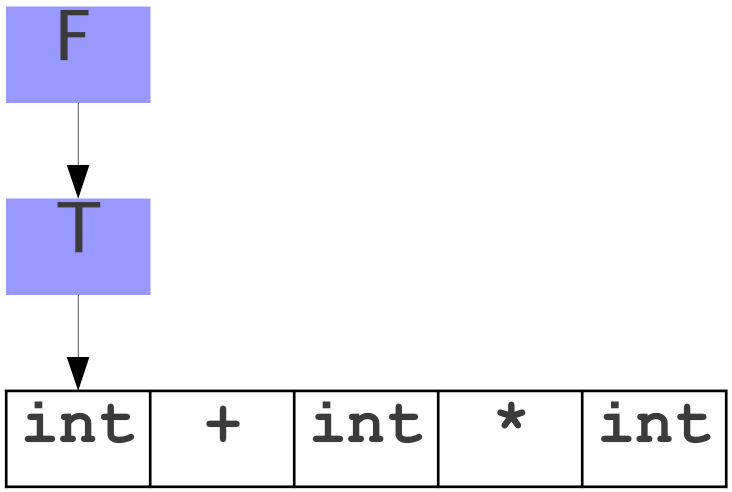
E → E + F

F → F * T

F → T

T → int

T → (E)



A Detail about Handles

E → F

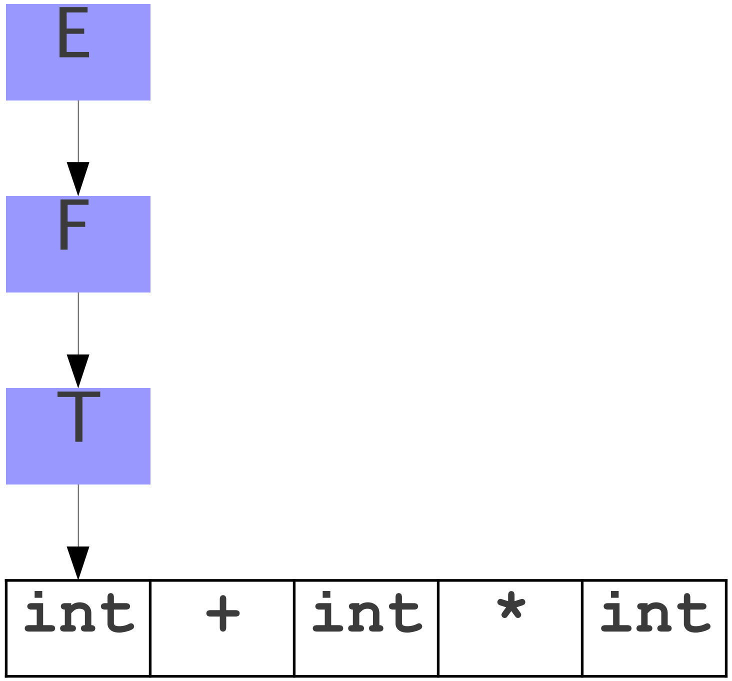
E → E + F

F → F * T

F → T

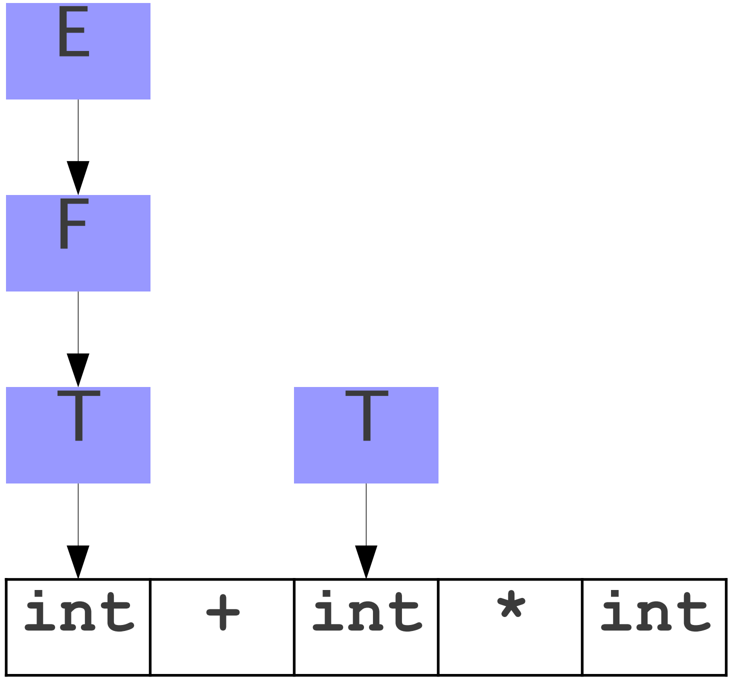
T → int

T → (E)



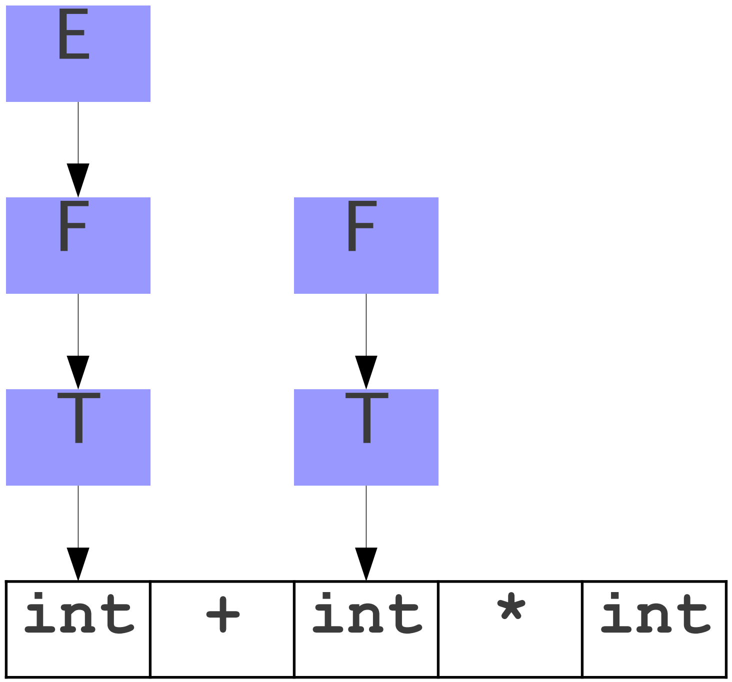
A Detail about Handles

E → F
E → E + F
F → F * T
F → T
T → int
T → (E)



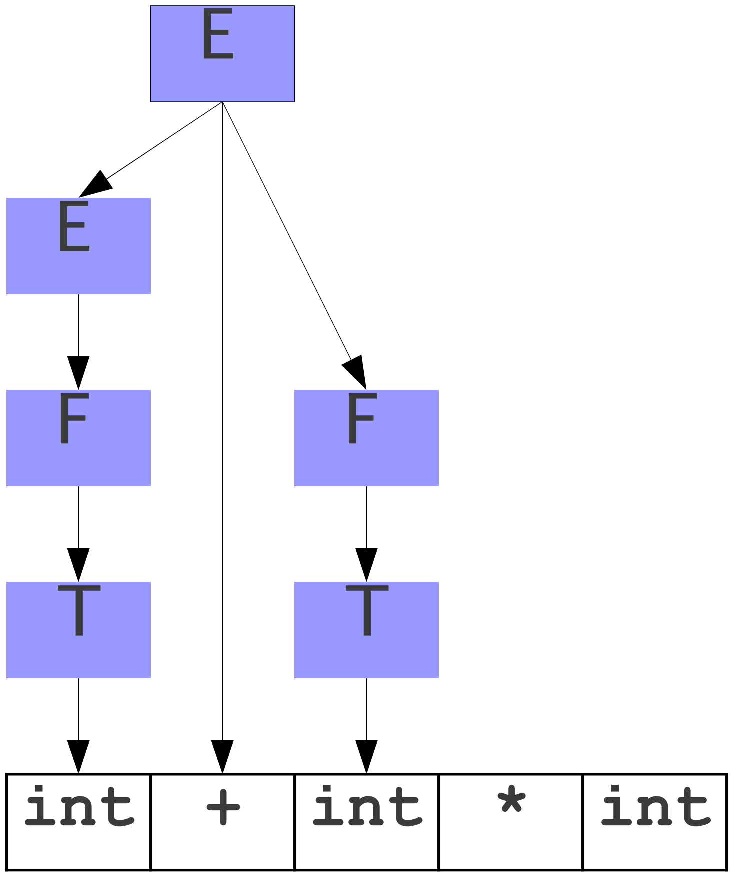
A Detail about Handles

E → F
E → E + F
F → F * T
F → T
T → int
T → (E)



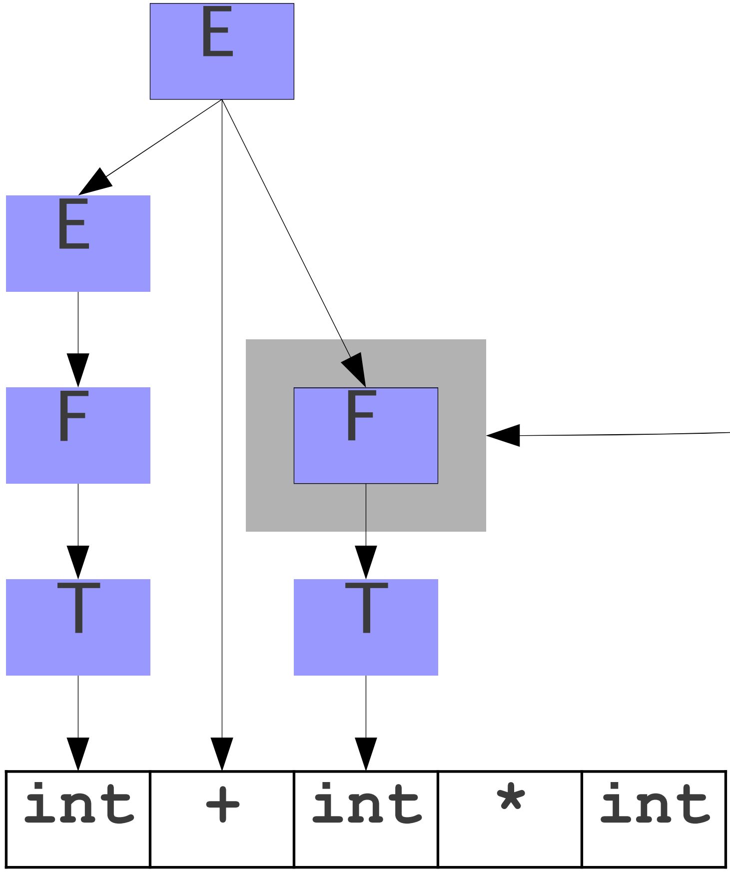
A Detail about Handles

$E \rightarrow F$
$E \rightarrow E + F$
$F \rightarrow F * T$
$F \rightarrow T$
$T \rightarrow$ `int`
$T \rightarrow (E)$



A Detail about Handles

E → F
E → E + F
F → F * T
F → T
T → int
T → (E)



This reduction wasn't a handle!

Key: Finding Handles

- Where do we look for handles?
  - Where in the string might the handle be?
- How do we search for possible handles?
  - Once we know where to search, how do we identify candidate handles?
- How do we recognize handles?
  - Once we've found a candidate handle, how do we check that it really is the handle?
    - Use a stack to keep track of the viable prefix
    - The prefix of the handle will always be at the top of the stack

Viable prefix

- Two types of viable prefix
  - Nonreducible (for shift operation): no simple phrase, need to shift more symbols to form the first leftmost simple phrase (i.e. handle)
  - Reducible (for reduction operation): contain one simple phrase, at the end of the

(1) $Z \rightarrow ABb$
(2) $A \rightarrow a$
(3) $A \rightarrow b$
(4) $B \rightarrow d$
(5) $B \rightarrow c$
(6) $B \rightarrow bB$

$Z \Rightarrow ABb$ Viable prefixes:
AB(no simple phrase) --- nonreducible
ABb (contain a simple phrase) --- reducible

Bottom-up Parsing

- Shift-reduce operations in bottom-up parsing
  - Shift the input into the stack
    - Wait for the current handle to complete or to appear
    - Or wait for a handle that may complete later
  - Reduce
    - Once the handle is completely in the stack, then reduce
  - The operations are determined by the parsing table

LR(0) algorithm

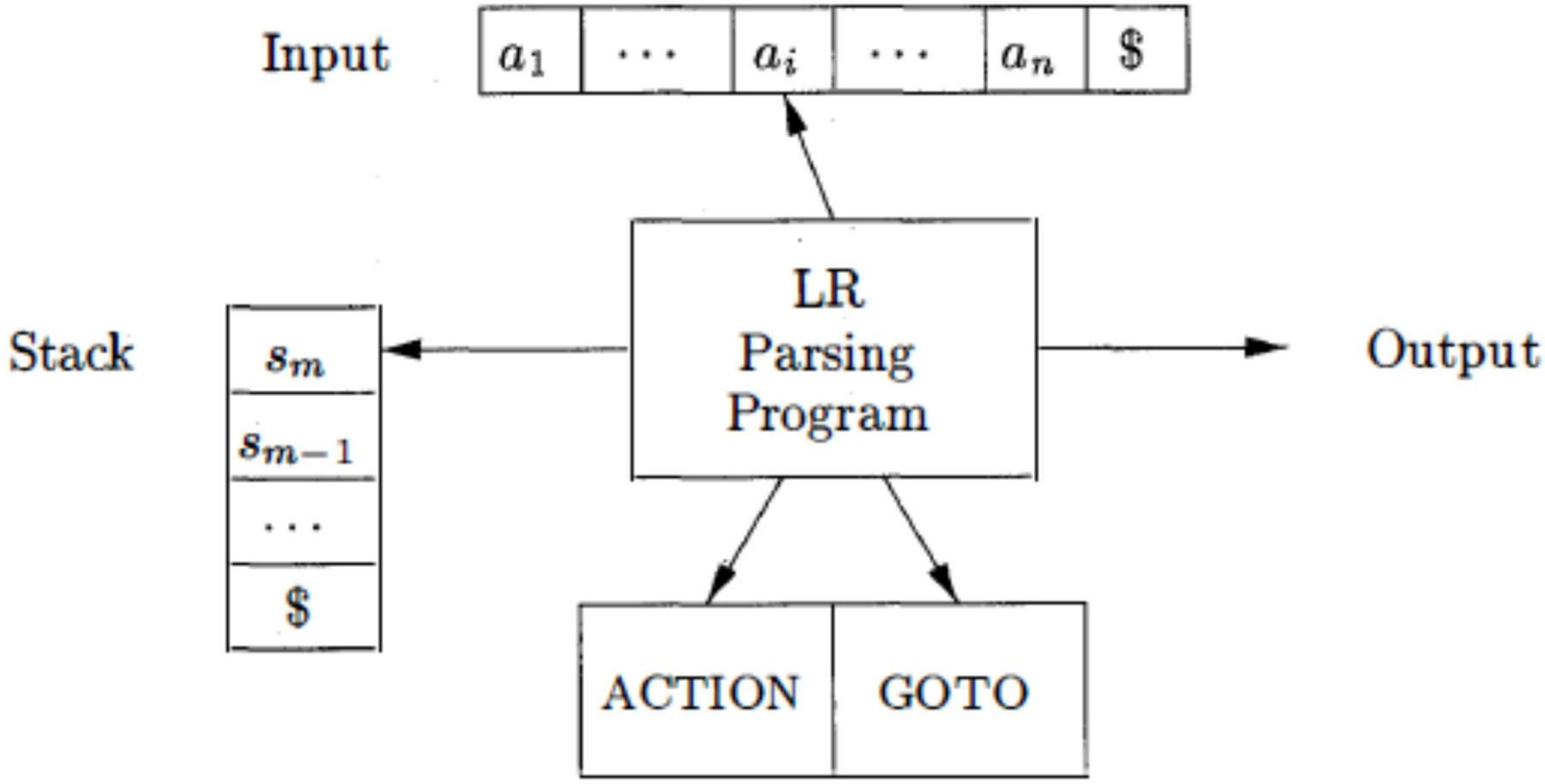


Figure 4.35: Model of an LR parser

Build the Automata

- LR(0) Item of a grammar G
  - Is a production of G with a distinguished position
  - Position is used to indicate how much of the handle has already been seen (in the stack)
    - For production $S \to a\ B\ S$, items for it include

      $S \to \bullet\ a\ B\ S$

      $S \to a \bullet B\ S$

      $S \to a\ B \bullet S$

      $S \to a\ B\ S \bullet$

      - Left of $\bullet$ are the parts of the handle that has already been seen
      - When $\bullet$ reaches the end of the handle $\Rightarrow$ reduction
    - For production $S \to \varepsilon$, the single item is

      $S \to \bullet$

Building the Automata

- Closure function Closure(I)
  - I is a set of items for a grammar G
  - Every item in I is in Closure(I), that is, I itself is in Closure(I)

    if $A \rightarrow \alpha \bullet B \beta$ is in Closure(I) and $B \rightarrow \gamma$ is a production in G, then add $B \rightarrow \bullet \gamma$ to Closure(I)
    - If it is not already there
    - Meaning
      - When α is in the stack and B is expected next
      - One of the B-production rules may be used to reduce the input to B
        - » May not be one-step reduction though
  - Apply the rule until no more new items can be added

Building the Automata

- CLOSURE(IS)Example

$V_T = \{a, b, c\}$
$V_N = \{S, A, B\}$
$S = S$
P:
$\{\ S \rightarrow aAc$
$A \rightarrow ABb$
$A \rightarrow Ba$
$B \rightarrow b$
$\}$

$IS = \{S \rightarrow \bullet aAc\}$
$CLOSURE(IS) = \{S \rightarrow \bullet aAc\}$

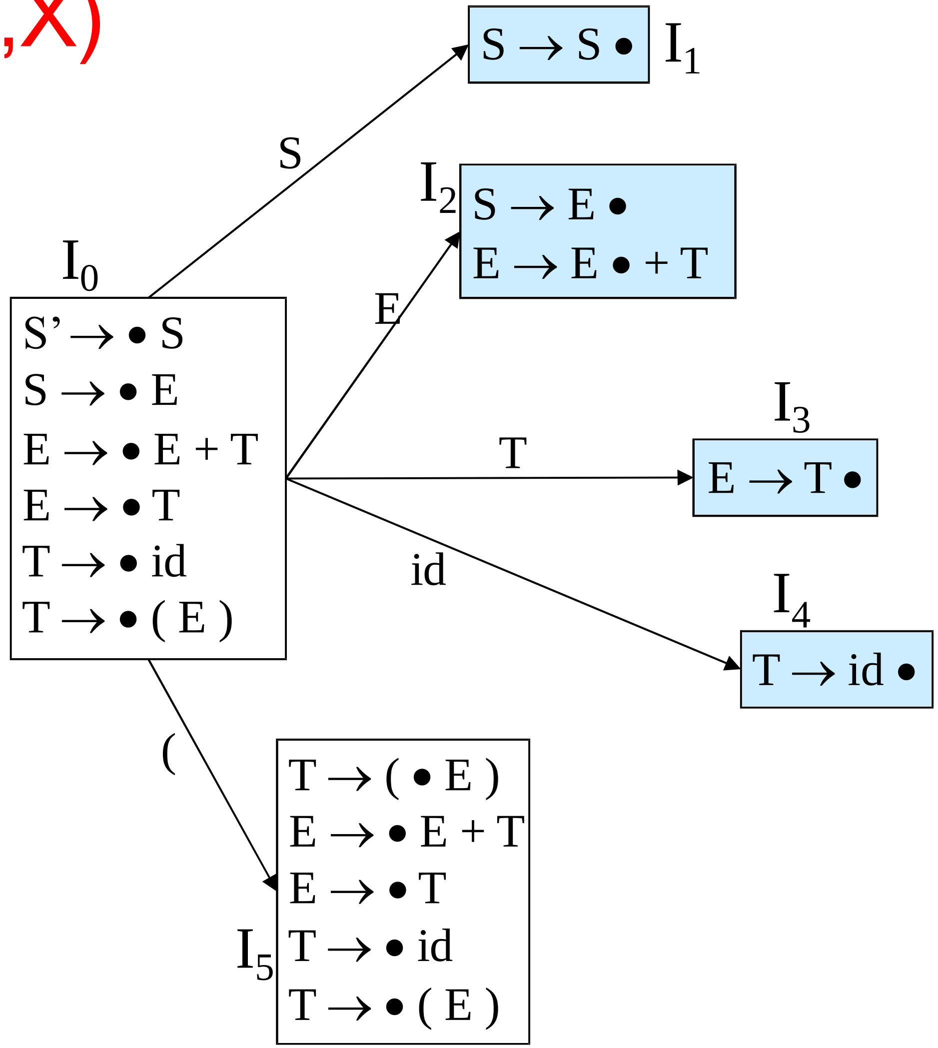
$IS = \{S \rightarrow a \bullet Ac\}$
$CLOSURE(IS)$
$= \{S \rightarrow a \bullet Ac,$
$A \rightarrow \bullet ABb, A \rightarrow \bullet Ba,$
$B \rightarrow \bullet b\}$

Building the Automata

- Goto function Goto(I,X)
  - X is a grammar symbol
  - If $A \rightarrow \alpha \bullet X \beta$ is in I then $A \rightarrow \alpha X \bullet \beta$ is in Goto(I, X)
    - Let J denote the set constructed by this step
  - All items in Closure(J) are in Goto(I, X)
  - Meaning
    - If I is the set of valid items for some viable prefix γ
    - Then goto(I, X) is the set of valid items for the viable prefix γX

Building the Automata

- Goto function Goto(I,X)



Building the Automata

- Augmented grammar
  - G is the grammar and S is the staring symbol
  - Construct G' by adding production $S' \rightarrow S$ into G
    - S' is the new starting symbol
    - E.g.: G: $S \rightarrow \alpha \mid \beta$ $\Rightarrow$ G': $S' \rightarrow S$, $S \rightarrow \alpha \mid \beta$
  - Meaning
    - The starting symbol may have several production rules and may be used in other non-terminal's production rules
    - Add $S' \rightarrow S$ to force the starting symbol to have a single production
    - When $S' \rightarrow S \bullet$ is seen, it is clear that parsing is done

Building the Automata

- Complete process: Given a grammar G
  - Step 1: augment G
  - Step 2: initial state
    - Construct the valid item set "I" of State 0 (the initial state)
    - Add $S' \rightarrow \bullet S$ into I
      - All expansions have to start from here
    - Compute Closure(I) as the complete valid item set of state 0
      - All possible expansions S can lead into
  - Step 3:
    - From state I, for all grammar symbol X

      Construct J = Goto(I, X)

      Compute Closure(J)
    - Create the new state with the corresponding Goto transition
      - Only if the valid item set is non-empty and does not exist yet
  - Repeat Step 3 till no new states can be derived

Building the Automata -- Example

- Grammar G:

  $S \rightarrow E$

  $E \rightarrow E + T \mid T$

  $T \rightarrow id \mid (E)$

  - Step 1: Augment G

    $S' \rightarrow S \qquad S \rightarrow E \qquad E \rightarrow E + T \mid T \qquad T \rightarrow id \mid (E)$

  - Step 2:
    - Construct Closure(I_0) for State 0
    - First add into I_0: $S' \rightarrow \bullet\, S$
    - Compute Closure(I_0) $S' \rightarrow \bullet\, S$

      $S \rightarrow \bullet\, E$

      $E \rightarrow \bullet\, E + T$ $E \rightarrow \bullet\, T$

      $T \rightarrow \bullet\, id$

      $T \rightarrow \bullet\, (E)$

Building the Automata -- Example

I_0

$S' \rightarrow \bullet S$
$S \rightarrow \bullet E$
$E \rightarrow \bullet E + T$
$E \rightarrow \bullet T$
$T \rightarrow \bullet id$
$T \rightarrow \bullet (E)$

Building the Automata -- Example

I_0:

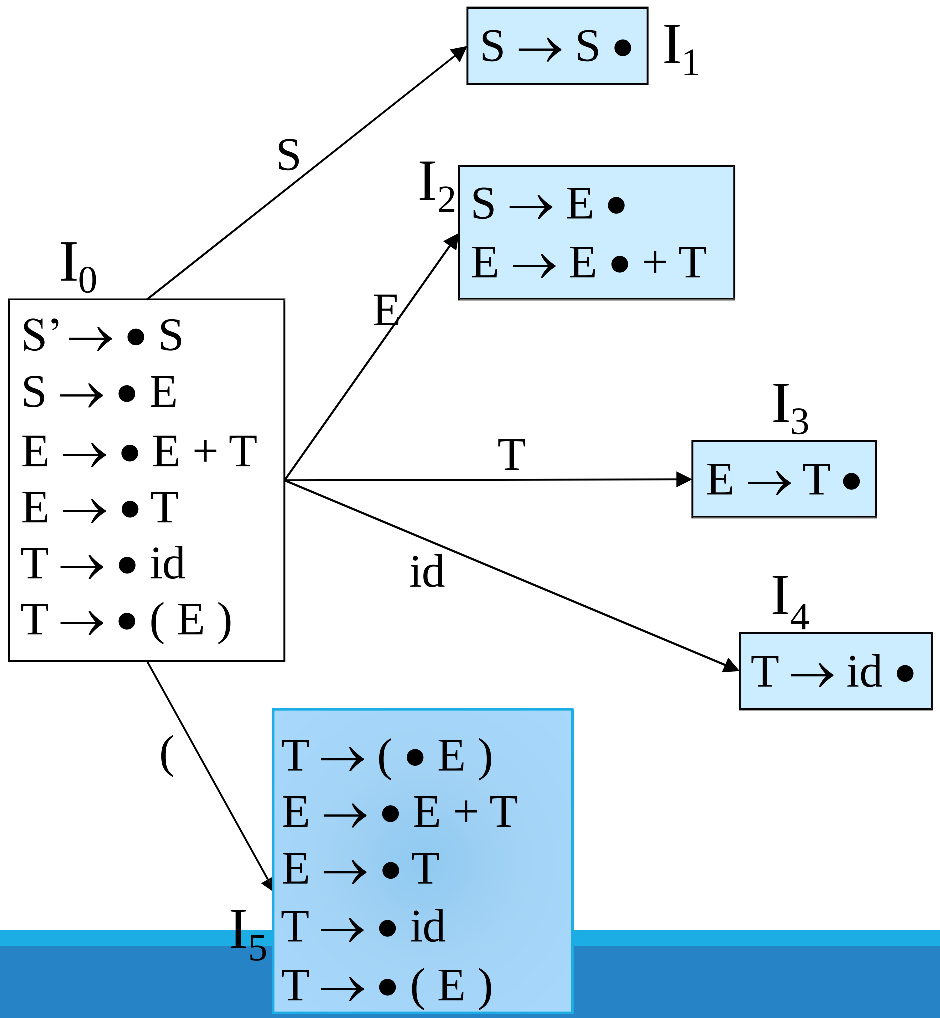
$S' \to \bullet S \qquad S \to \bullet E$

$E \to \bullet E + T \qquad E \to \bullet T$

$T \to \bullet id \qquad T \to \bullet (E)$

- Step 3
  - I_1
    - Add into I_1: Goto(I_0, S) = $S' \to S \bullet$
    - No new items to be added to Closure (I_1)
  - I_2
    - Add into I_2: Goto(I_0, E) = $S \to E \bullet \qquad E \to E \bullet + T$
    - No new items to be added to Closure (I_2)
  - I_3
    - Add into I_3: Goto(I_0, T) = $E \to T \bullet$
    - No new items to be added to Closure (I_3)
  - I_4
    - Add into I_4: Goto(I_0, id) = $T \to id \bullet$
    - No new items to be added to Closure (I_4)

Building the Automata -- Example



I_0:

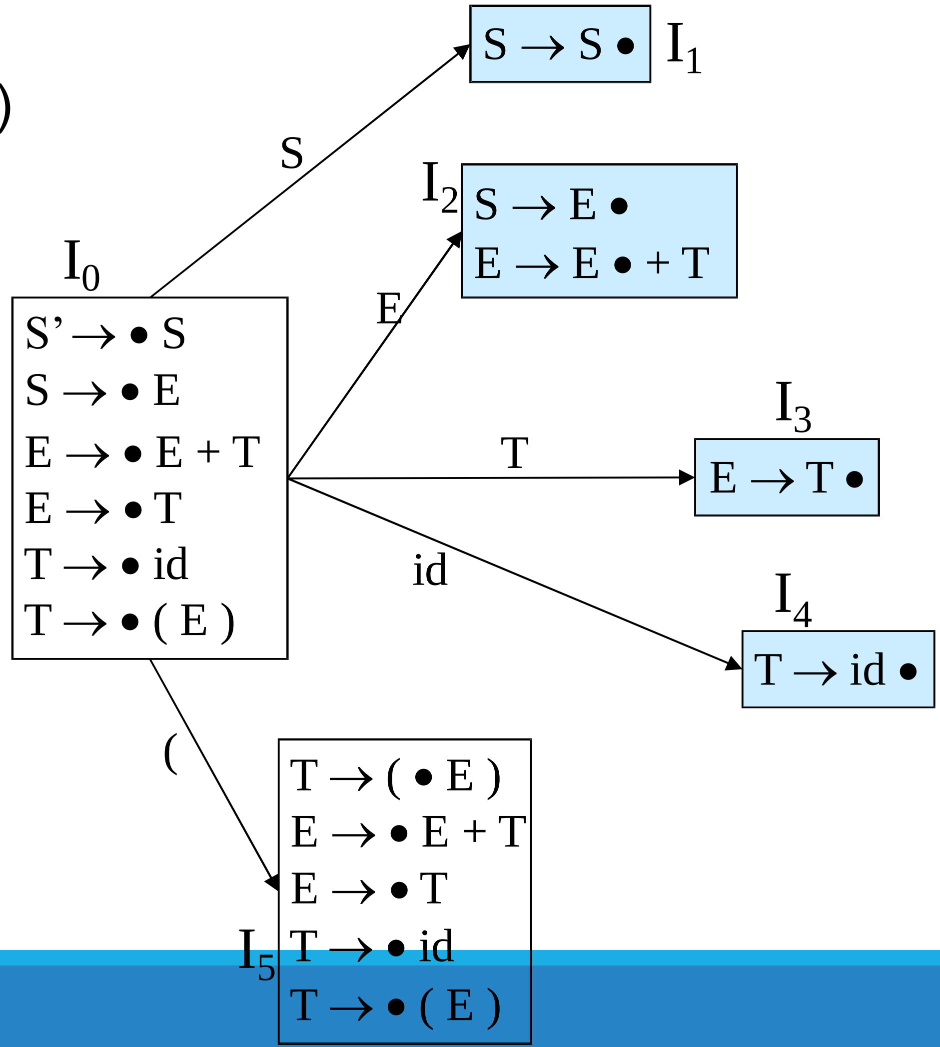
$S' \to \bullet S \quad S \to \bullet E \quad E \to \bullet E + T \quad E \to \bullet T$

$T \to \bullet id \quad T \to \bullet (E)$

- Step 3
  - I_5
    - Add into I_5: Goto(I_0, "(") = $T \to (\bullet E)$
    - Closure(I_5)

      $E \to \bullet E + T \quad E \to \bullet T$

      $T \to \bullet id \quad T \to \bullet (E)$



I_0:

$S' \rightarrow \bullet S \quad S \rightarrow \bullet E$

$E \rightarrow \bullet E + T \quad E \rightarrow \bullet T$

$T \rightarrow \bullet id \quad T \rightarrow \bullet (E)$

- Step 3
  - No more moves from I_0
  - No possible moves from I_1
  - I_6
    - Add into I_6: Goto(I_2, +) = $E \rightarrow E + \bullet T$
    - Closure(I_5)

      $T \rightarrow \bullet id \quad T \rightarrow \bullet (E)$
  - No possible moves from I_3 and I_4

Building the Automata -- Example

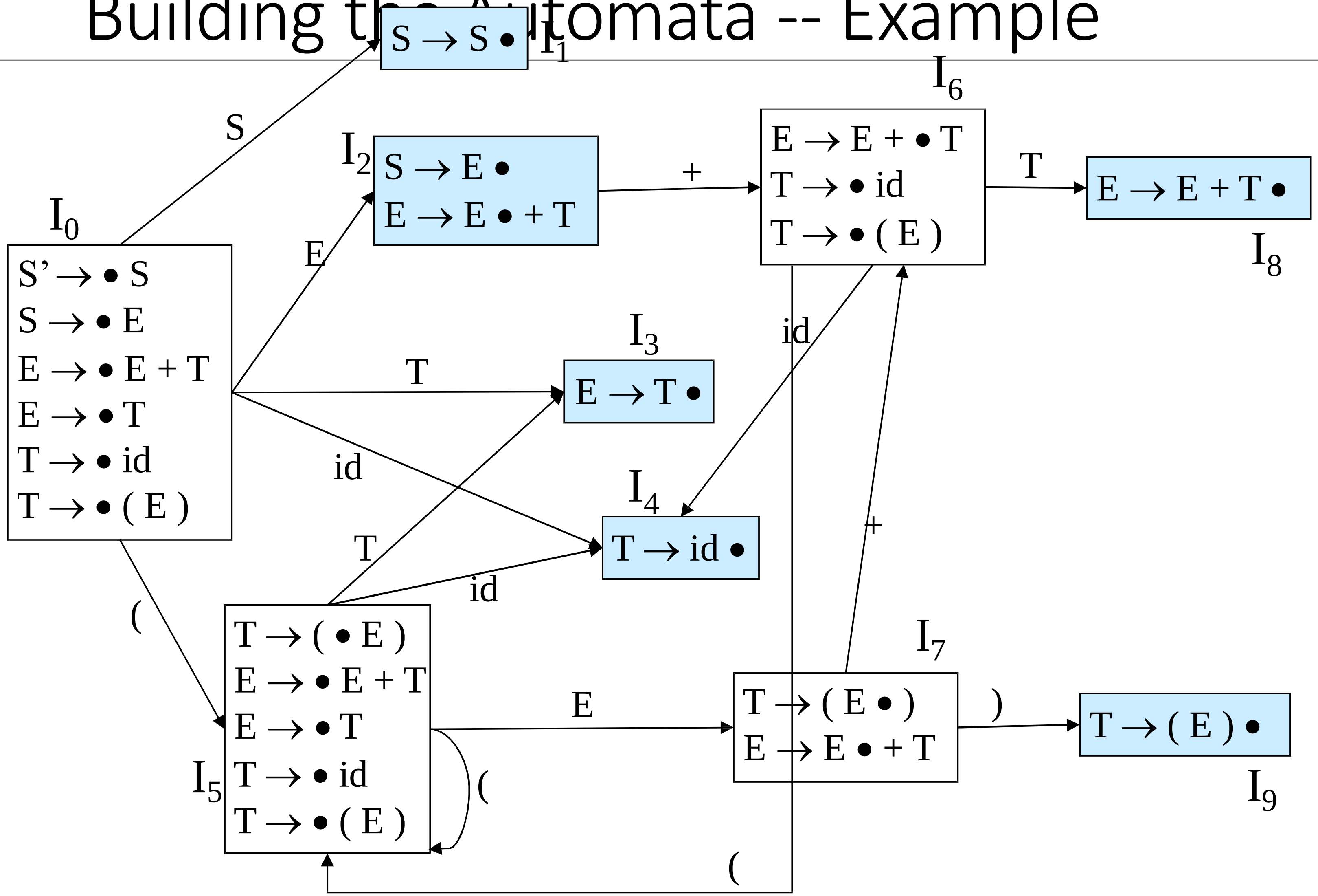
- Step 3
  - I_7
    - Add into I_7: Goto(I_5, E) =
      $T \rightarrow (E \bullet)$ $\quad$ $E \rightarrow E \bullet + T$
    - No new items to be added to Closure (I_7)
  - Goto(I_5, T) = I_3
  - Goto(I_5, id) = I_4
  - Goto(I_5, “(”) = I_5
  - No more moves from I_5
  - I_8
    - Add into I_8: Goto(I_6, T) = $E \rightarrow E + T \bullet$
    - No new items to be added to Closure (I_8)
  - Goto(I_6, id) = I_4
  - Goto(I_6, “(”) = I_5

Building the Automata -- Example

- Step 3
  - I_9
    - Add into I_9: Goto(I_7, ")") =

      $T \rightarrow (E) \bullet$
    - No new items to be added to Closure (I_9)
  - Goto(I_7, +) = I_6
  - No possible moves from I_8 and I_9

Building the Automata -- Example



Reducible or Nonreducible

- LR(0) parser
  - Shift item: $A \rightarrow \alpha \bullet a\beta$, $a \in V_T$
  - Reducible item: $A \rightarrow \alpha \bullet$,
  - Accepted item: $Z \rightarrow S \bullet$, ($Z \rightarrow S$ is from the augmented grammar)
  - Shift status: include shift item
  - Reducible state: include reducible item
  - Conflict state:
    - A state contains different reducible items: reduce-reduce conflict;
    - A state contains both shift states and reducible items: shift-reduce conflict

Building the Automata – Example 2

$V_T = \{a, b, c\}$

$V_N = \{S, A, B\}$

$S = S$

P:

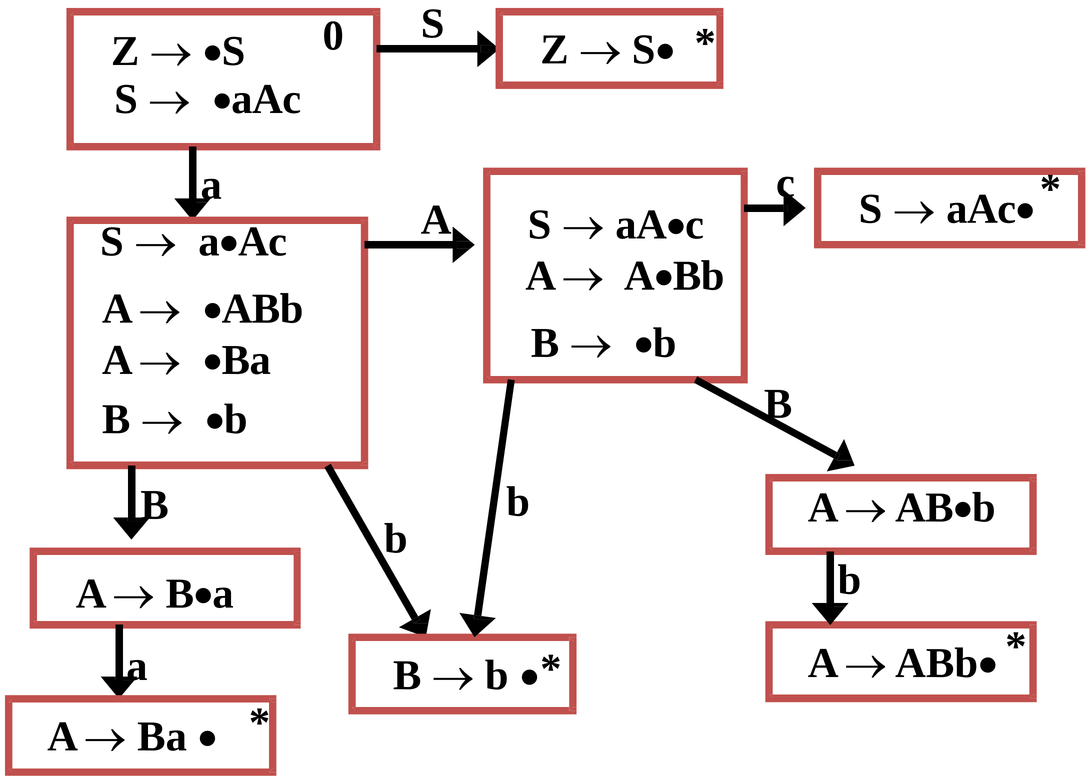
$\{\ S \rightarrow aAc$

$A \rightarrow ABb$

$A \rightarrow Ba$

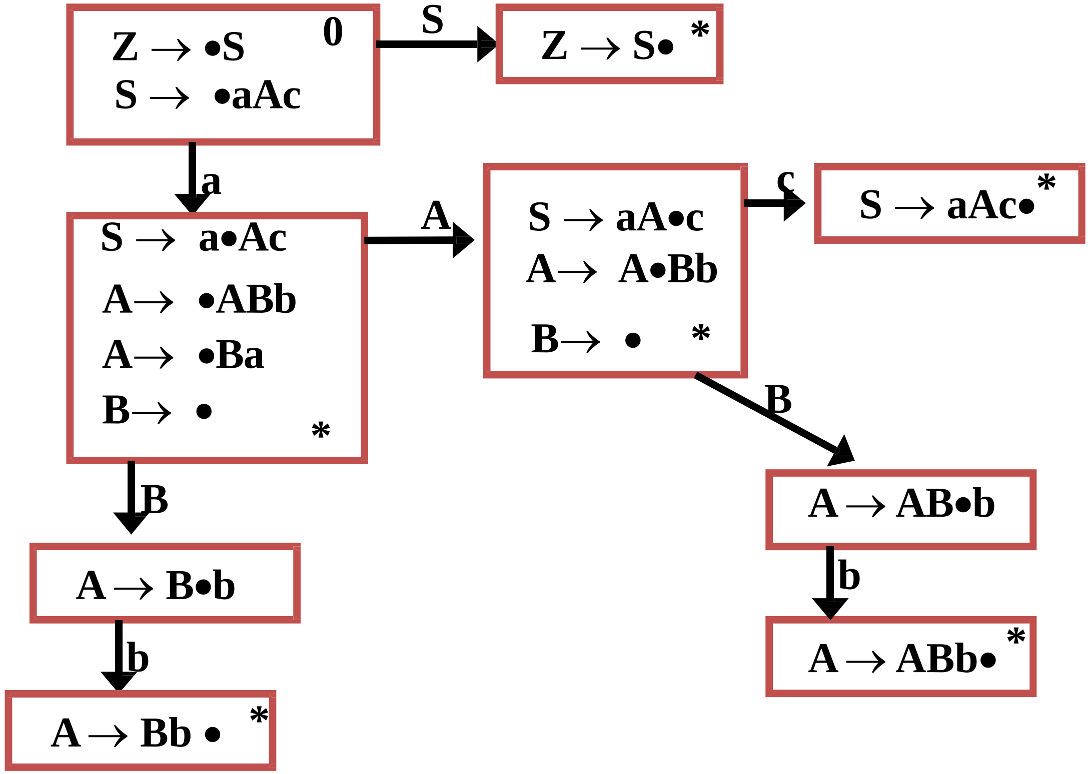
$B \rightarrow b$

$\}$



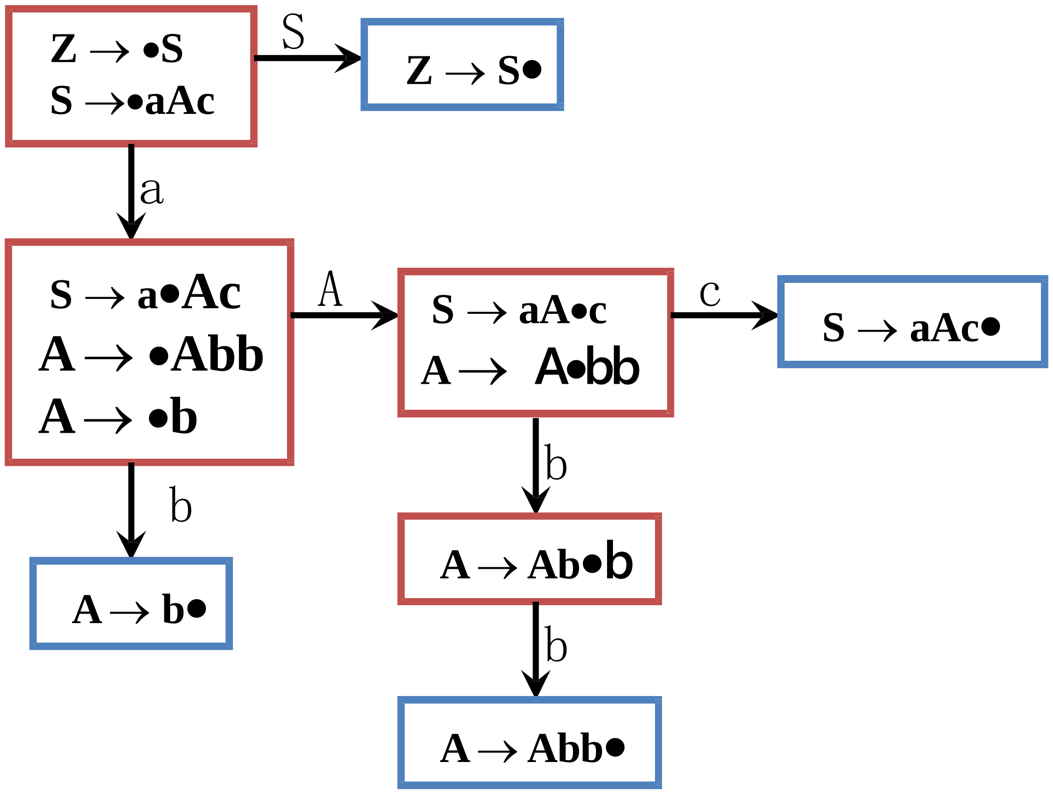
Building the Automata – Example 3

$V_T = \{a, b, c\}$
$V_N = \{S, A, B\}$
$S = S$
P:
{ $S \to aAc$
$A \to ABb$
$A \to Ba$
$B \to \varepsilon$
}



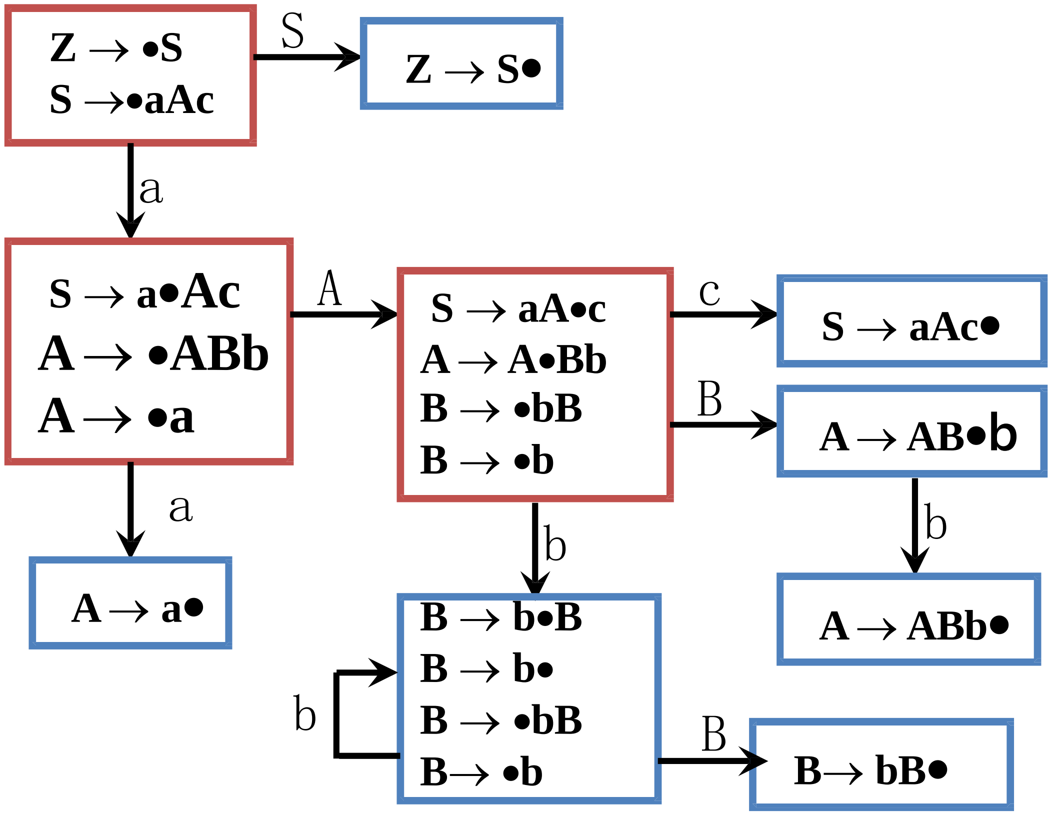
Building the Automata – Example 4

V_T = {a, b, c}
V_N = {S, A}
S = S
P:
{ S → aAc
A → Abb
A → b
}



Building the Automata – Example 5

V_T = {a, b, c}
V_N = {S, A, B}
S = S
P:
{ $S \to aAc$
$A \to ABb$
$A \to a$
$B \to bB$
$B \to b$
}



LR(0) algorithm

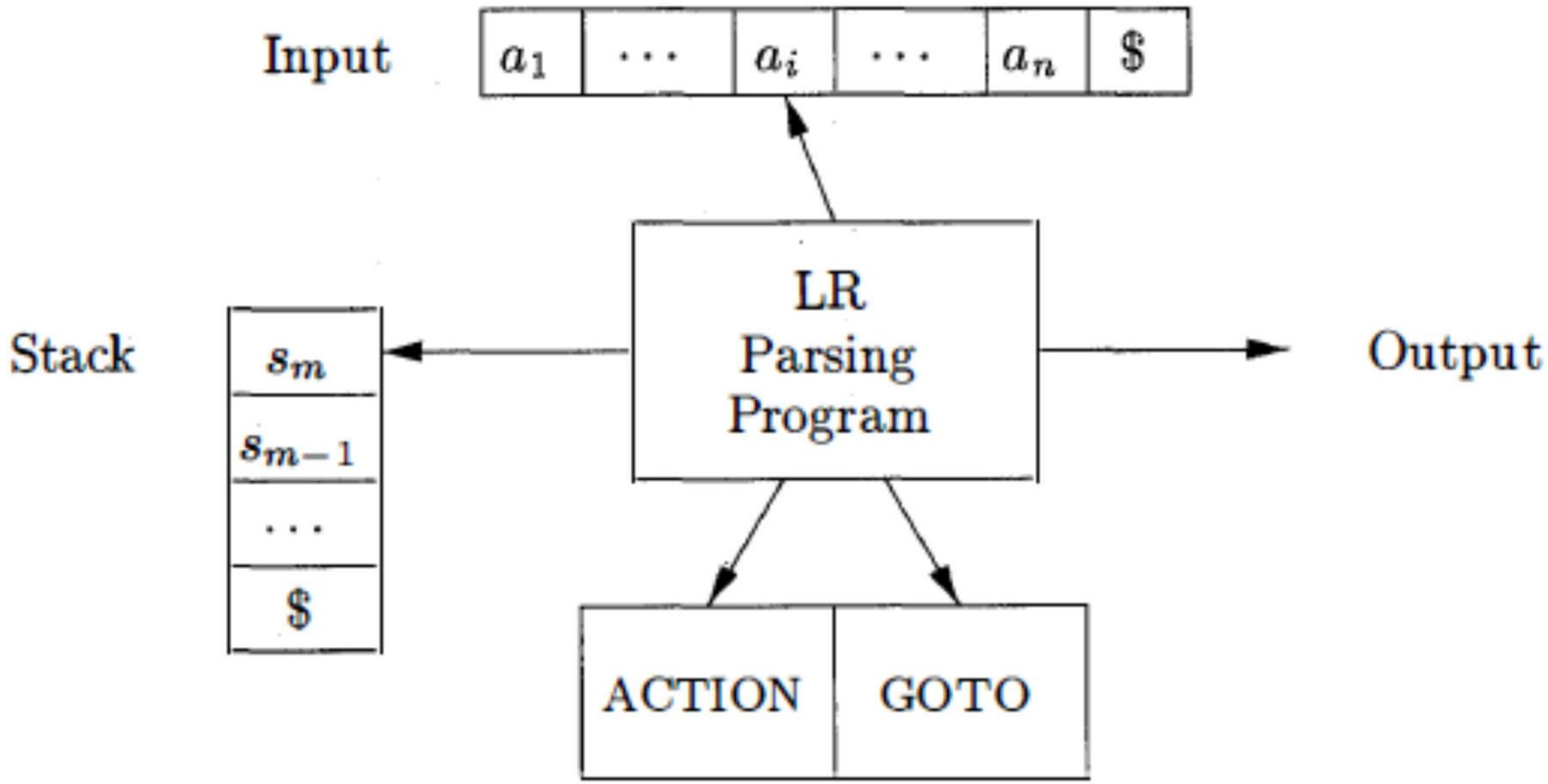


Figure 4.35: Model of an LR parser

Building the Action Table

- If state I_i has item $A \to \alpha \bullet a\ \beta$, and
  - Goto(I_i, a) = I_j
  - Next symbol in the input is a
- Then Action[I_i, a] = I_j
  - Meaning: Shift "a" to the stack and move to state I_j
    - Need to wait for the handle to appear or to complete
- If State I_i has item $A \to \alpha \bullet$
- Then Action[S, b] = reduce using $A \to \alpha$
  - For all b in Follow(A)
  - Meaning: The entire handle α is in the stack, need to reduce
  - Need to wait to see Follow(A) to know that the handle is ready
    - E.g. $S \to E \bullet$ $\quad E \to E \bullet + T$
    - Current input can be either Follow(S) or +

Building the Action Table

- If state has $S' \to S_0 \bullet$
- Then Action[S, $] = accept

- Current state
  - The action to be taken depends on the current state
    - In LL, it depends on the current non-terminal on the top of the stack
    - In LR, non-terminal is not known till reduction is done
  - Who is keeping track of current state?
  - The stack
    - Need to push the state also into the stack
    - The stack includes the viable prefix and the corresponding state for each symbol in the viable prefix

Building the Action Table

Action Table

action(S_i,a) = S_j, if there is an edge from S_i to S_j labeled as a

action(S_i,c) = R_p, if S_i is a p-reducible state, $c \in Vt \cup \{\#\}$

action(S_i,#) = accept, if S_i is acceptance state

action(S_i,a) = error, otherwise

Terminal symbols / States	a_1	…	#
S_1			
…			
S_n			

Building the Goto Table

- If $Goto(I_i, A) = I_j$
- Then $Goto[i, A] = j$
- Meaning
  - When a reduction $X \rightarrow \alpha$ taken place
  - The non-terminal X is added to the stack replacing α
  - What should the state be after adding X
  - This information is kept in Goto table

Building the Goto Table

GOTO Table

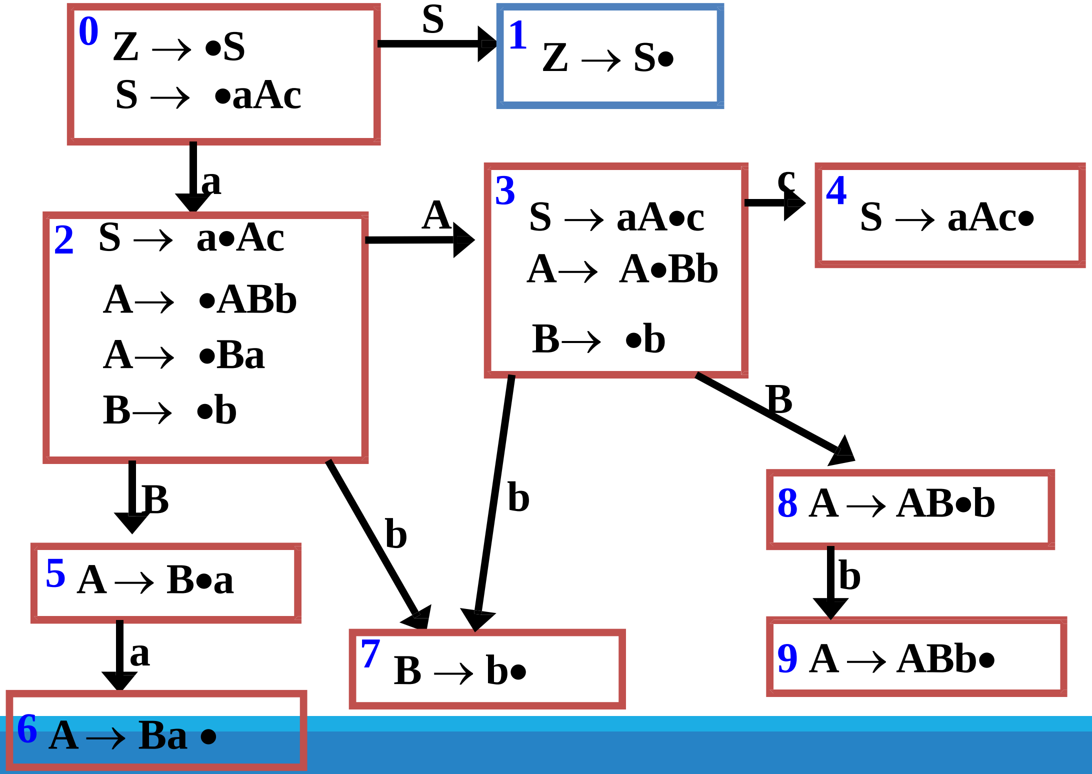
goto (S_i, A) = S_j, if there is an edge from S_i to S_j labeled as A
goto (S_i, A) = error, if there is no edge from S_i to S_j labeled as A

non-terminal / **State**	A_1	…	#
S_1			
…			
S_n			

LR(0) Parsing algorithm

$V_T = \{a, b, c\}$
$V_N = \{S, A, B\}$
$S = S$
P: { (1) $S \to aAc$ (2)$A \to ABb$ (3)$A \to Ba$ (4)$B \to b$}

action					goto		
	a	b	c	#	S	A	B
0	S2				1		
1				accept			
2		S7				3	5
3		S7	S4				8
4	R1	R1	R1	R1			
5	S6						
6	R3	R3	R3	R3			
7	R4	R4	R4	R4			
8		S9					
9	R2	R2	R2	R2			



LR(0) Parsing algorithm

Algorithm 4.44: LR-parsing algorithm.

INPUT: An input string w and an LR-parsing table with functions ACTION and GOTO for a grammar G.

OUTPUT: If w is in $L(G)$, the reduction steps of a bottom-up parse for w; otherwise, an error indication.

METHOD: Initially, the parser has s_0 on its stack, where s_0 is the initial state, and $w\$$ in the input buffer. The parser then executes the program in Fig. 4.36.

□

```
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;
}
```

LR(0) Parsing algorithm

a	b	a	c

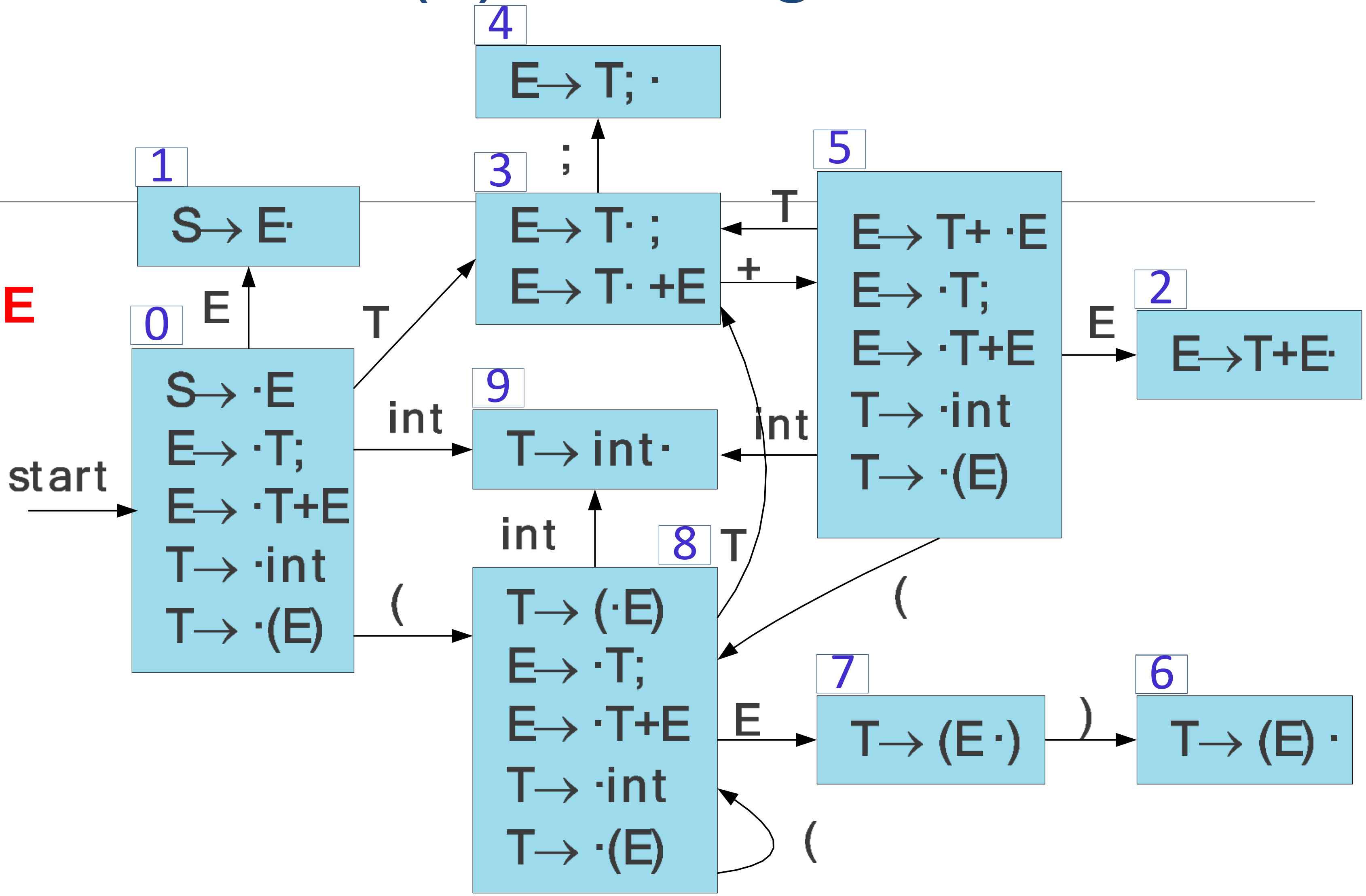
P: (0) $Z \to S$; (1) $S \to aAc$; (2) $A \to ABb$;
(3) $A \to Ba$; (4) $B \to b$

	action					goto		
	a	b	c	#		S	A	B
0	S2					1		
1				accept				
2		S7					3	5
3		S7	S4					8
4	R1	R1	R1	R1				
5	S6							
6	R3	R3	R3	R3				
7	R4	R4	R4	R4				
8		S9						
9	R2	R2	R2	R2				

Stack	Input	Actions
0	abac#	S2
02	bac#	S7
027	ac#	R4,Goto(2, B)=5
025	ac#	S6
0256	c#	R3,Goto(2, A)=3
023	c#	S4
0234	#	R1, Goto(0, S)=1
01	#	Accept

LR(0) Parsing

- $S \rightarrow E$
- $E \rightarrow T;$
- $E \rightarrow T + E$
- $T \rightarrow$ int
- $T \rightarrow (E)$



int	+	(	int	+	int	;	)	;

LR(0) Parsing

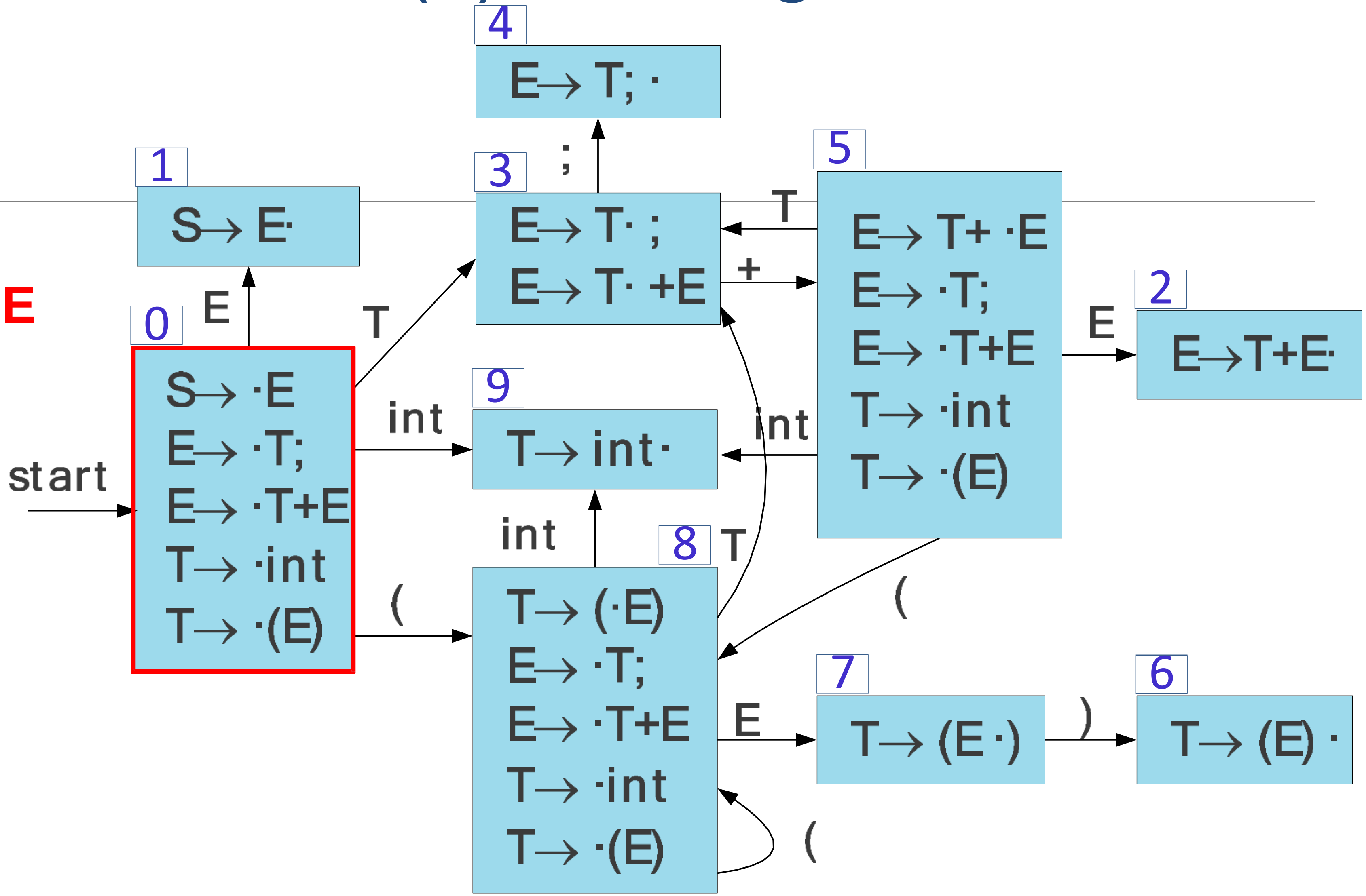
$S \rightarrow E$

$E \rightarrow T;$

$E \rightarrow T + E$

$T \rightarrow \text{int}$

$T \rightarrow (E)$



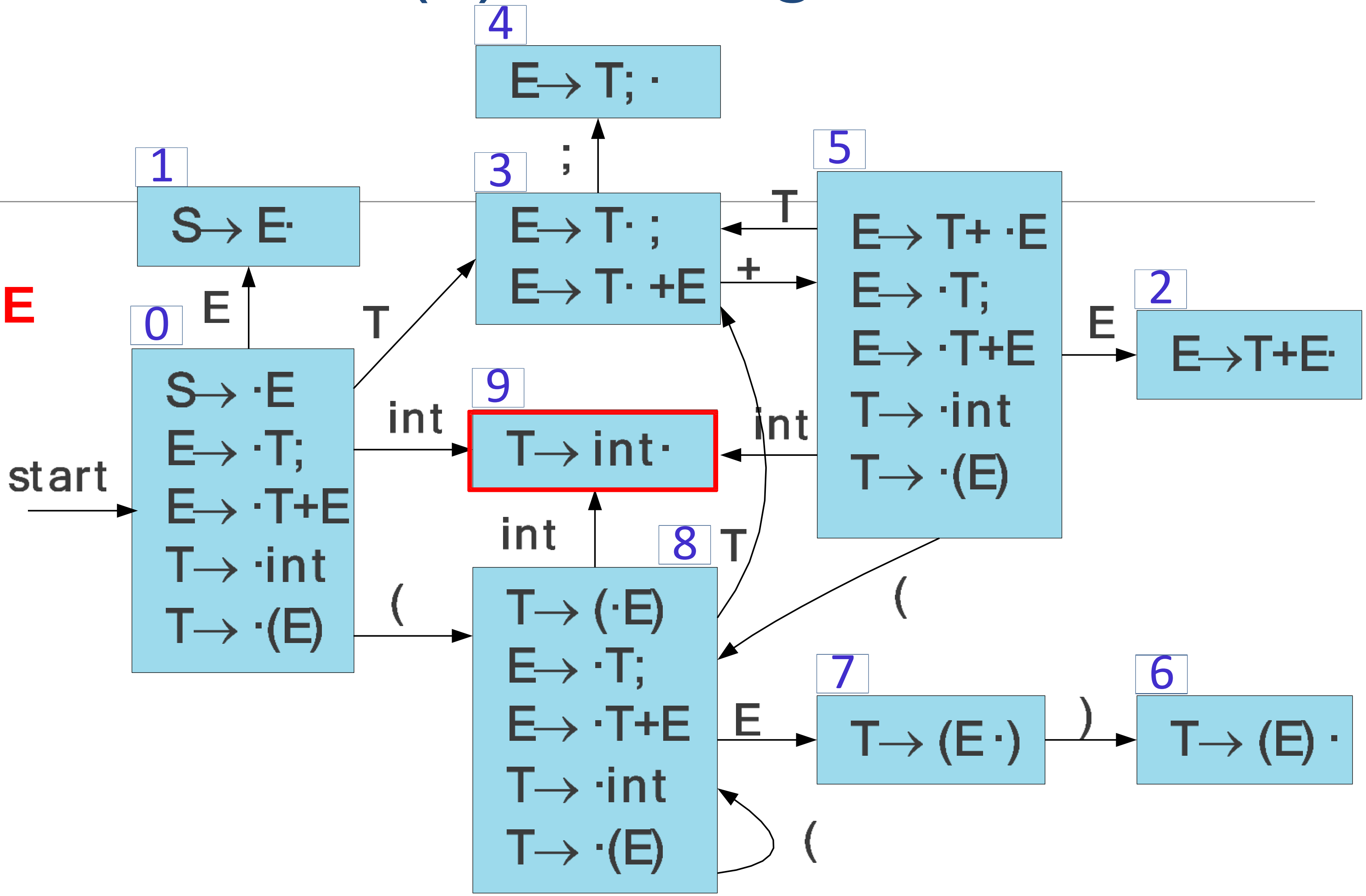
int	+	(	int	+	int	;	)	;

$ $

0

LR(0) Parsing

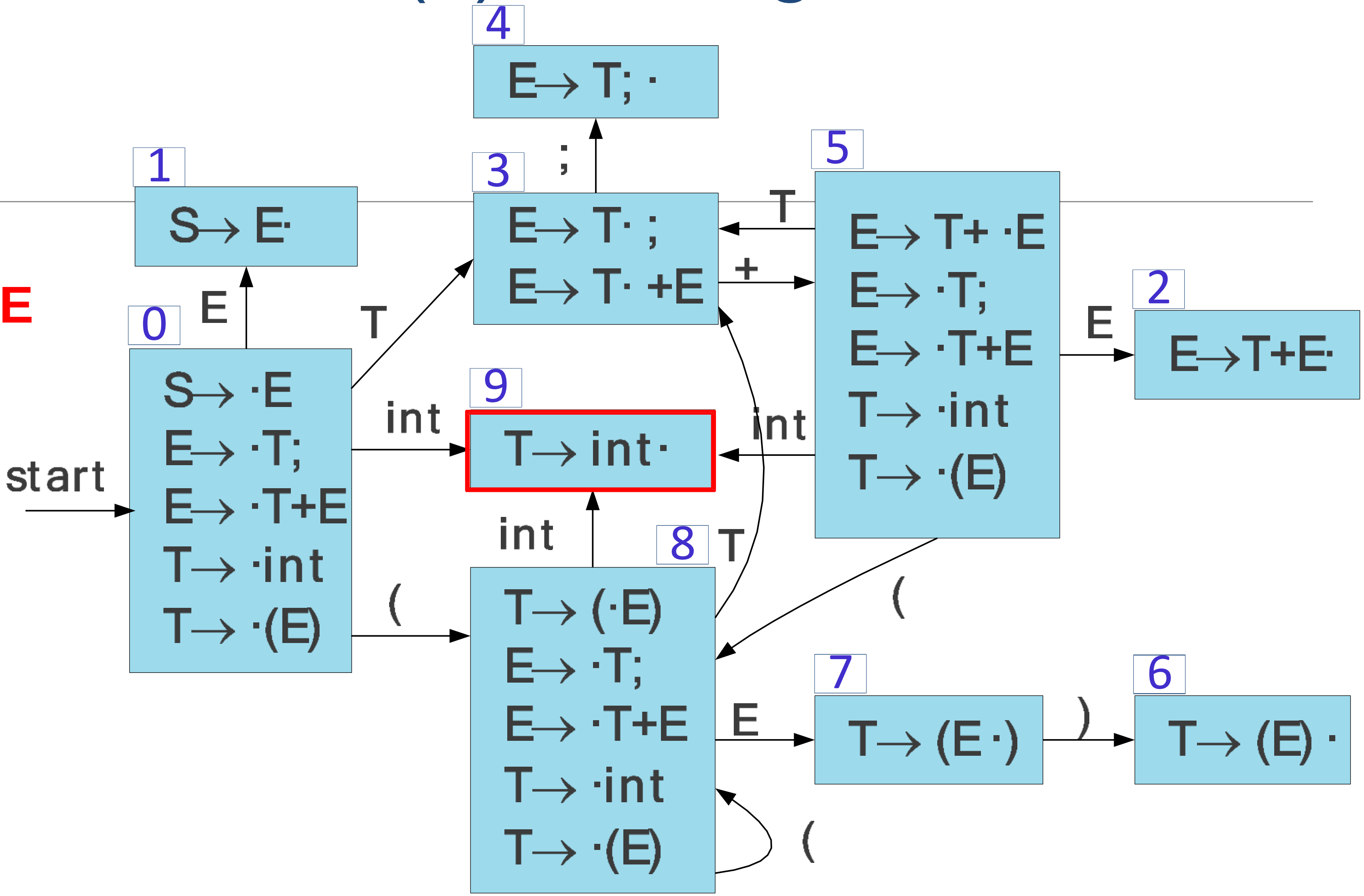
$S \rightarrow E$
$E \rightarrow T;$
$E \rightarrow T + E$
$T \rightarrow \text{int}$
$T \rightarrow (E)$



$	int	|	+	(	int	+	int	;	)	;	$

LR(0) Parsing

S → E
E → T;
E → T + E
T → int
T → (E)



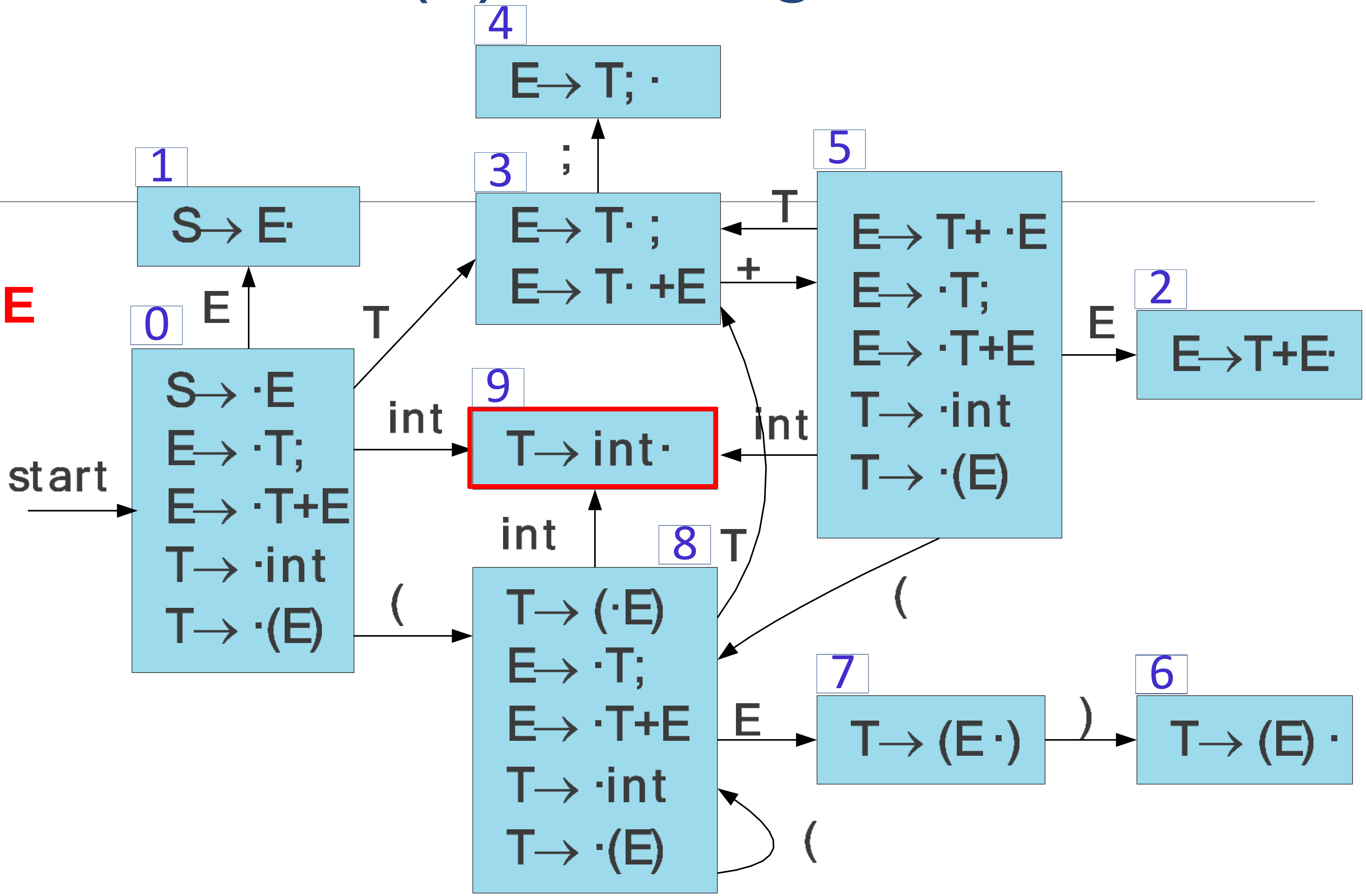
$

0

+	(	int	+	int	;	)	;	$

LR(0) Parsing

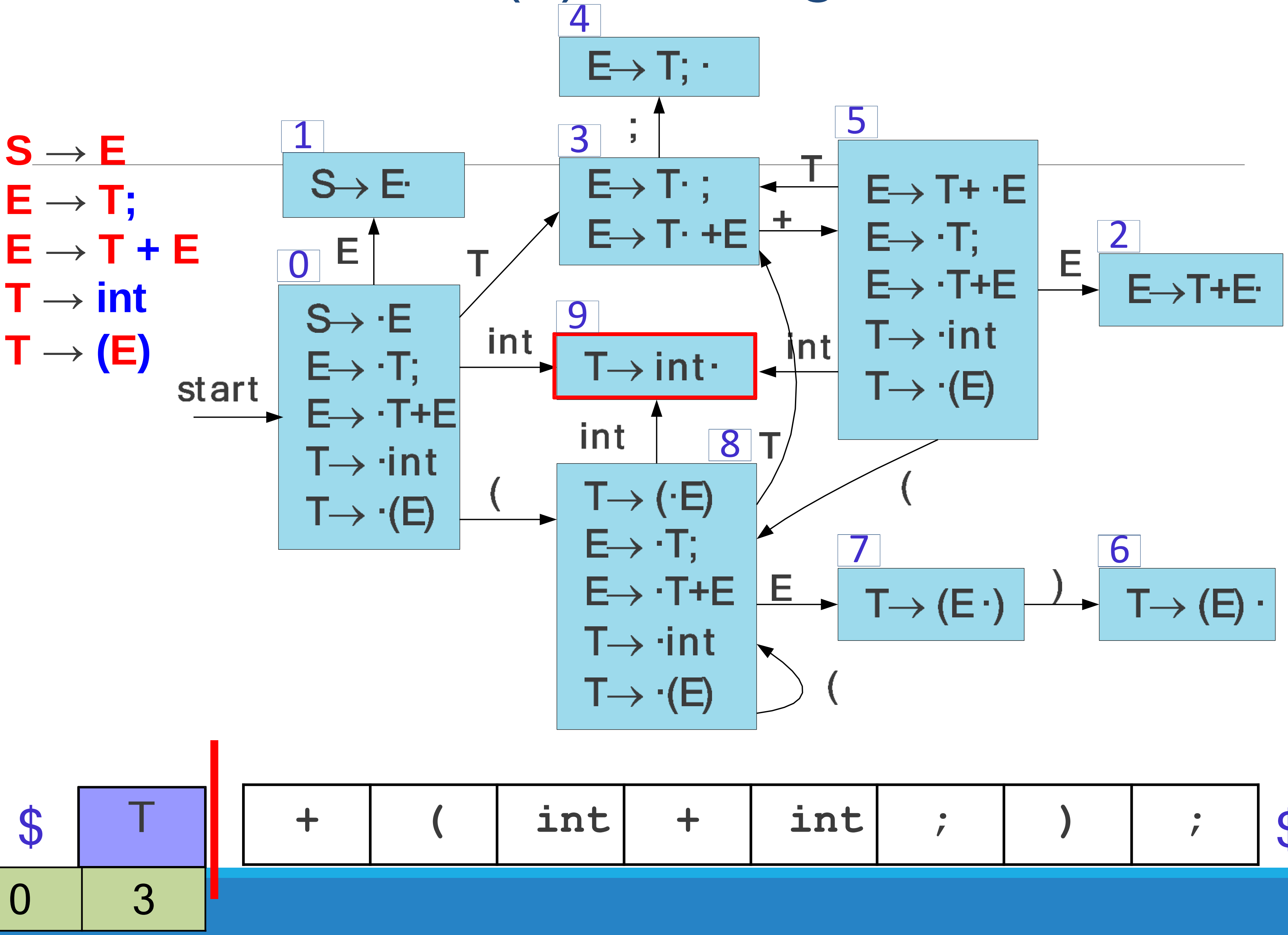
S → E
E → T;
E → T + E
T → int
T → (E)



$	T	|	+	(	int	+	int	;	)	;

0

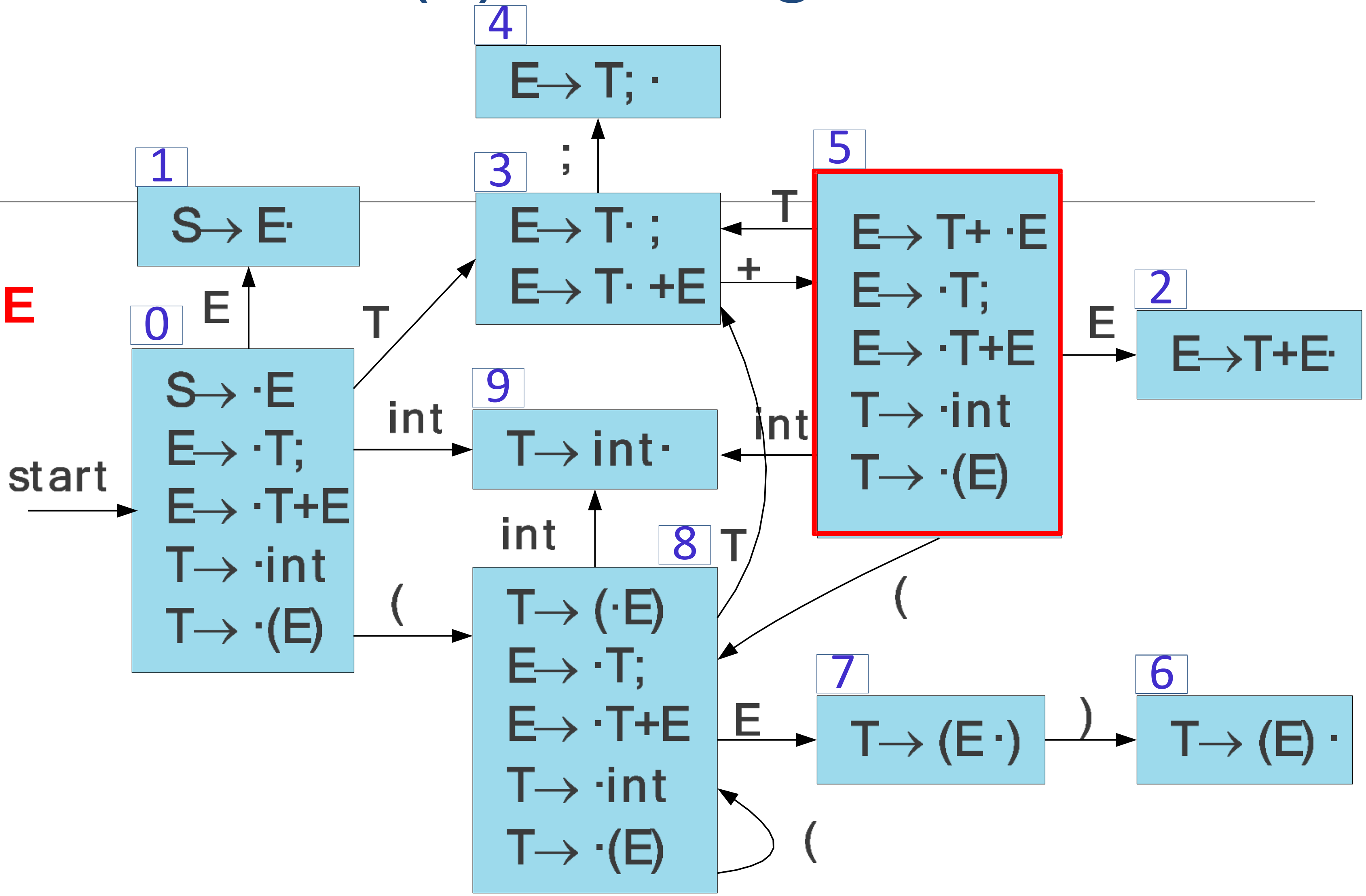
LR(0) Parsing



S → E
E → T;
E → T + E
T → int
T → (E)
start
0
S→ ·E
E→ ·T;
E→ ·T+E
T→ ·int
T→ ·(E)
1
S→ E·
2
E→T+E·
3
E→ T· ;
E→ T· +E
4
E→ T; ·
5
E→ T+ ·E
E→ ·T;
E→ ·T+E
T→ ·int
T→ ·(E)
6
T→ (E) ·
7
T→ (E ·)
8
T→ (·E)
E→ ·T;
E→ ·T+E
T→ ·int
T→ ·(E)
9
T→ int·
E
T
int
(
;
+
)
$
T
0
3
+
(
int
+
int
;
)
;
$

LR(0) Parsing

$S \rightarrow E$
$E \rightarrow T;$
$E \rightarrow T + E$
$T \rightarrow$ int
$T \rightarrow (E)$

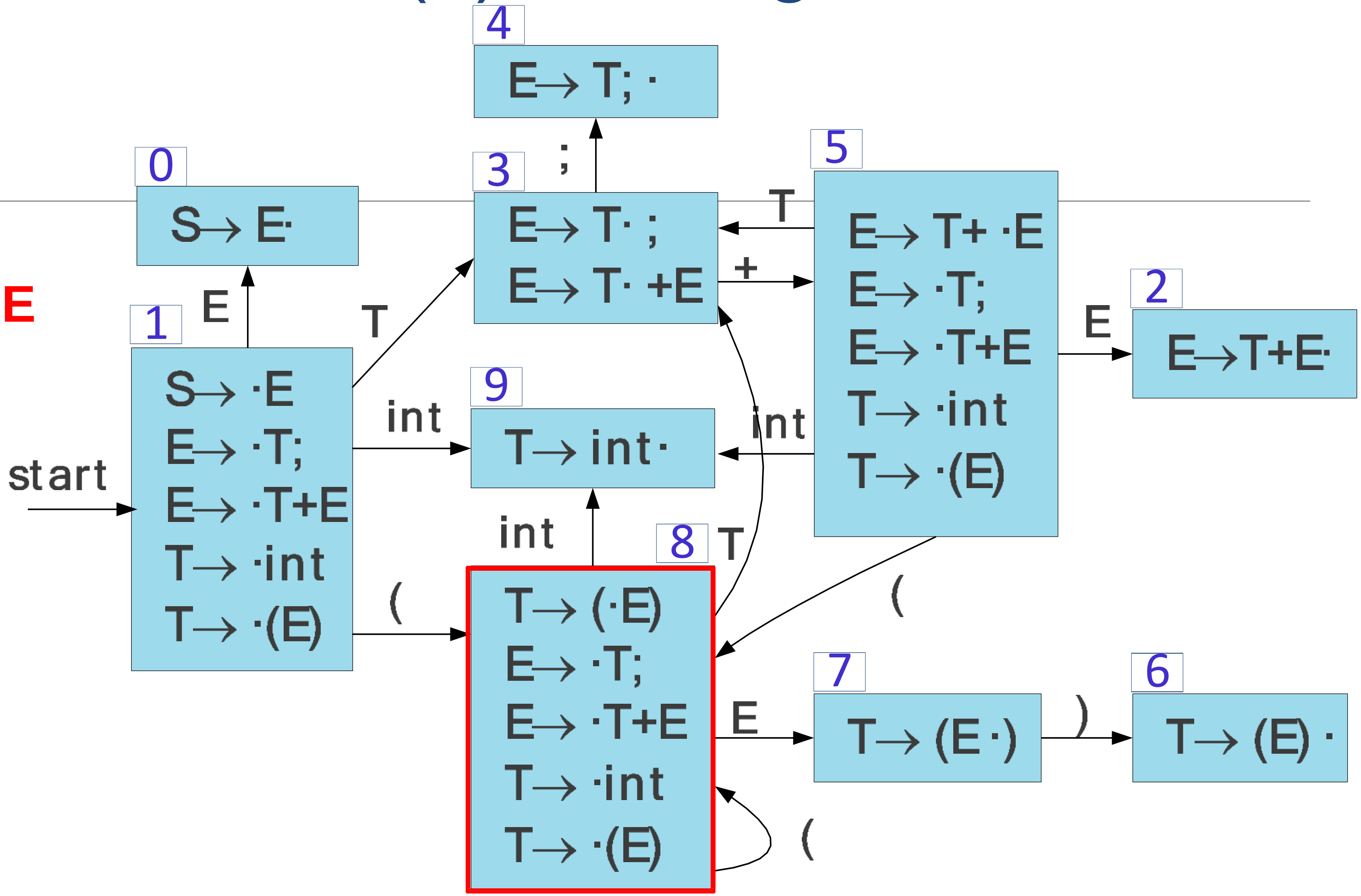


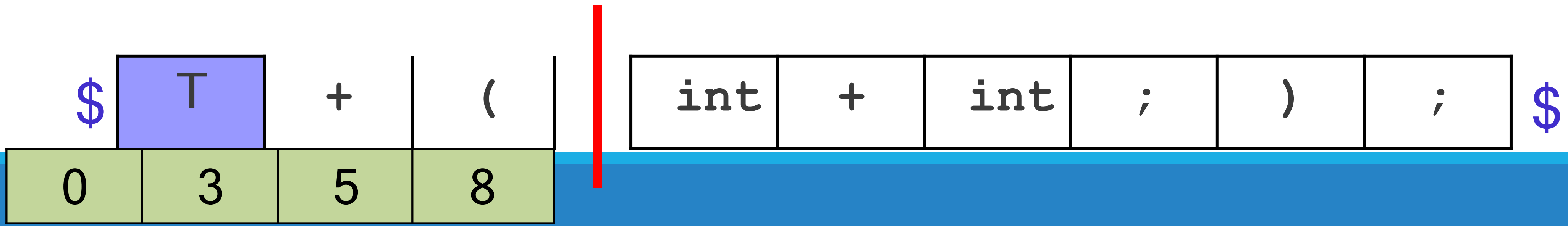
$	T	+
	0	3	5

(	int	+	int	;	)	;	$

LR(0) Parsing

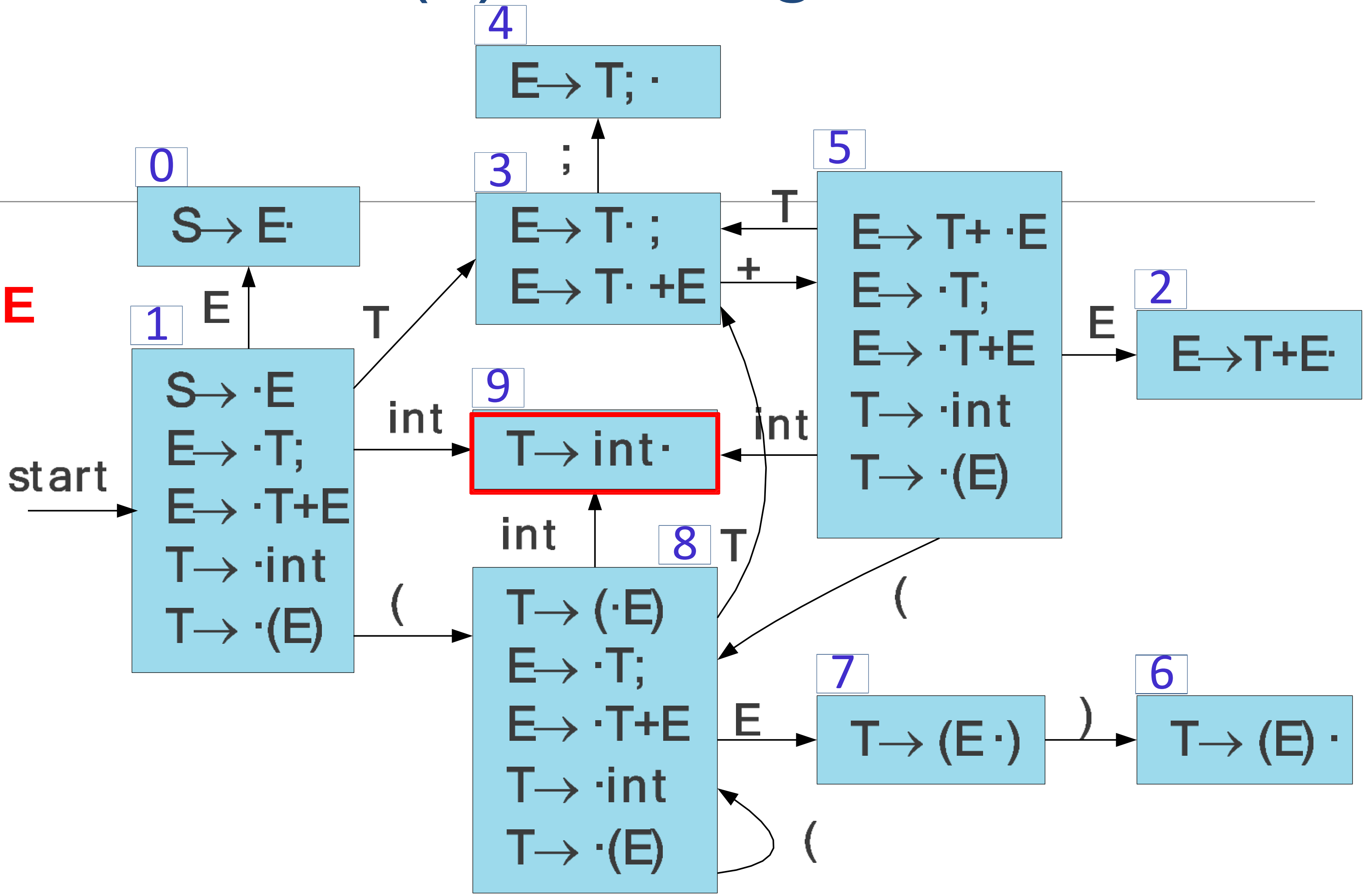
$S \rightarrow E$
$E \rightarrow T;$
$E \rightarrow T + E$
$T \rightarrow int$
$T \rightarrow (E)$

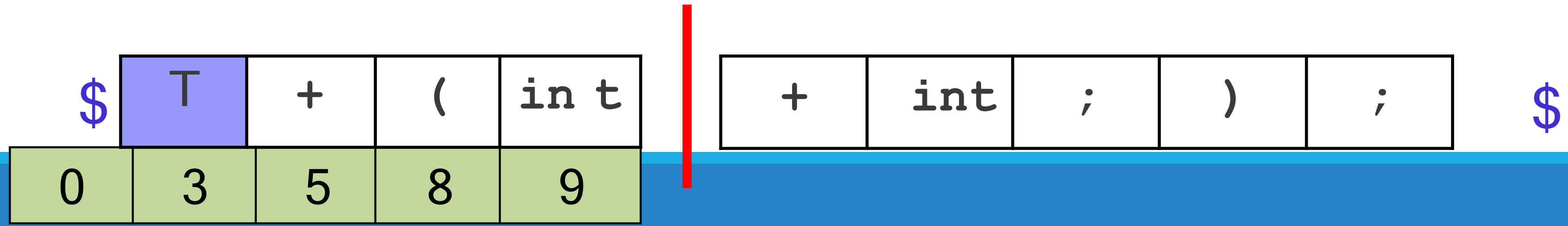




LR(0) Parsing

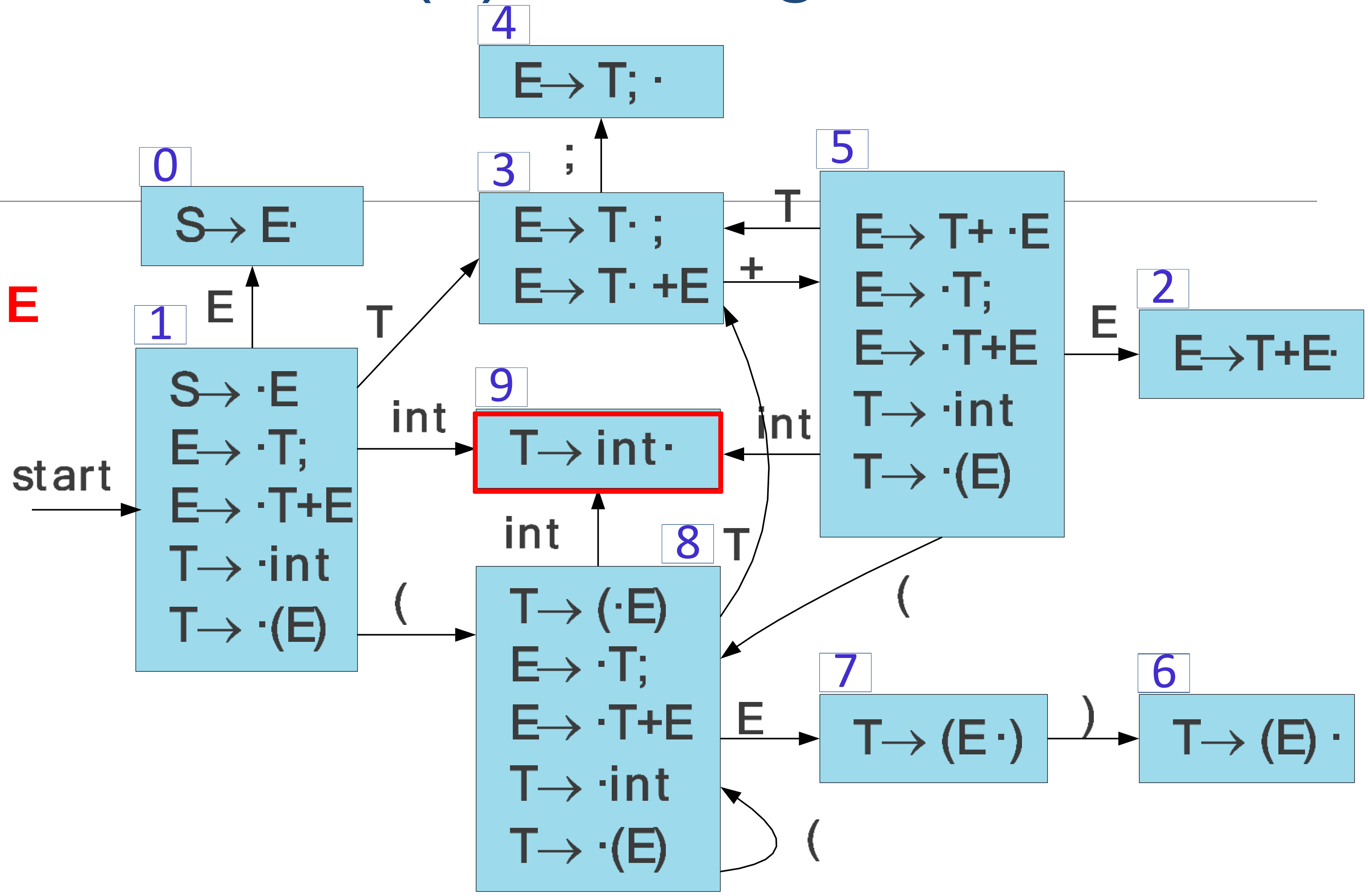
$S \rightarrow E$
$E \rightarrow T;$
$E \rightarrow T + E$
$T \rightarrow \text{int}$
$T \rightarrow (E)$





LR(0) Parsing

S → E
E → T;
E → T + E
T → int
T → (E)

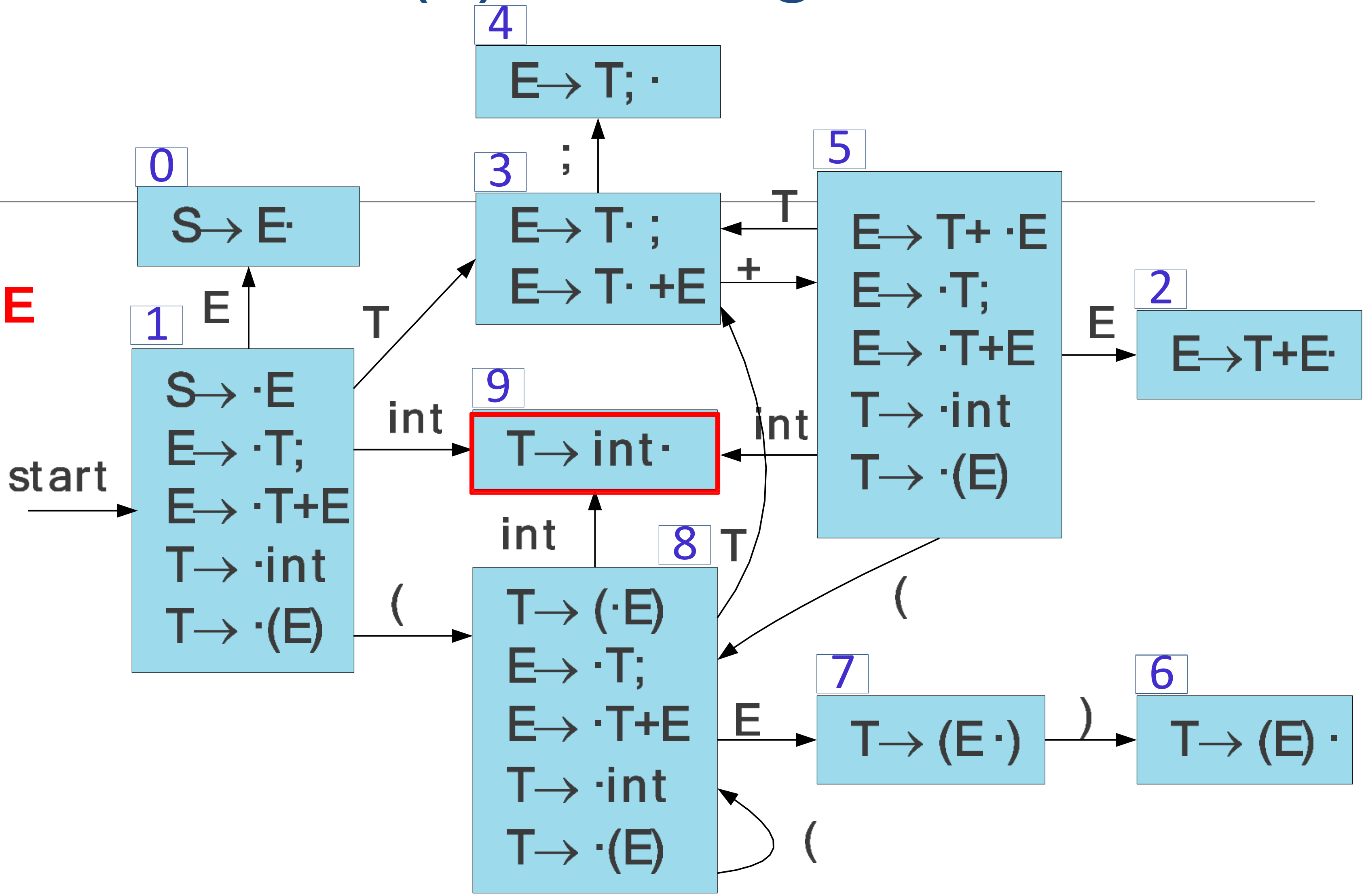


$	T	+	(
	0	3	5	8

+	int	;	)	;	$

LR(0) Parsing

S → E
E → T;
E → T + E
T → int
T → (E)

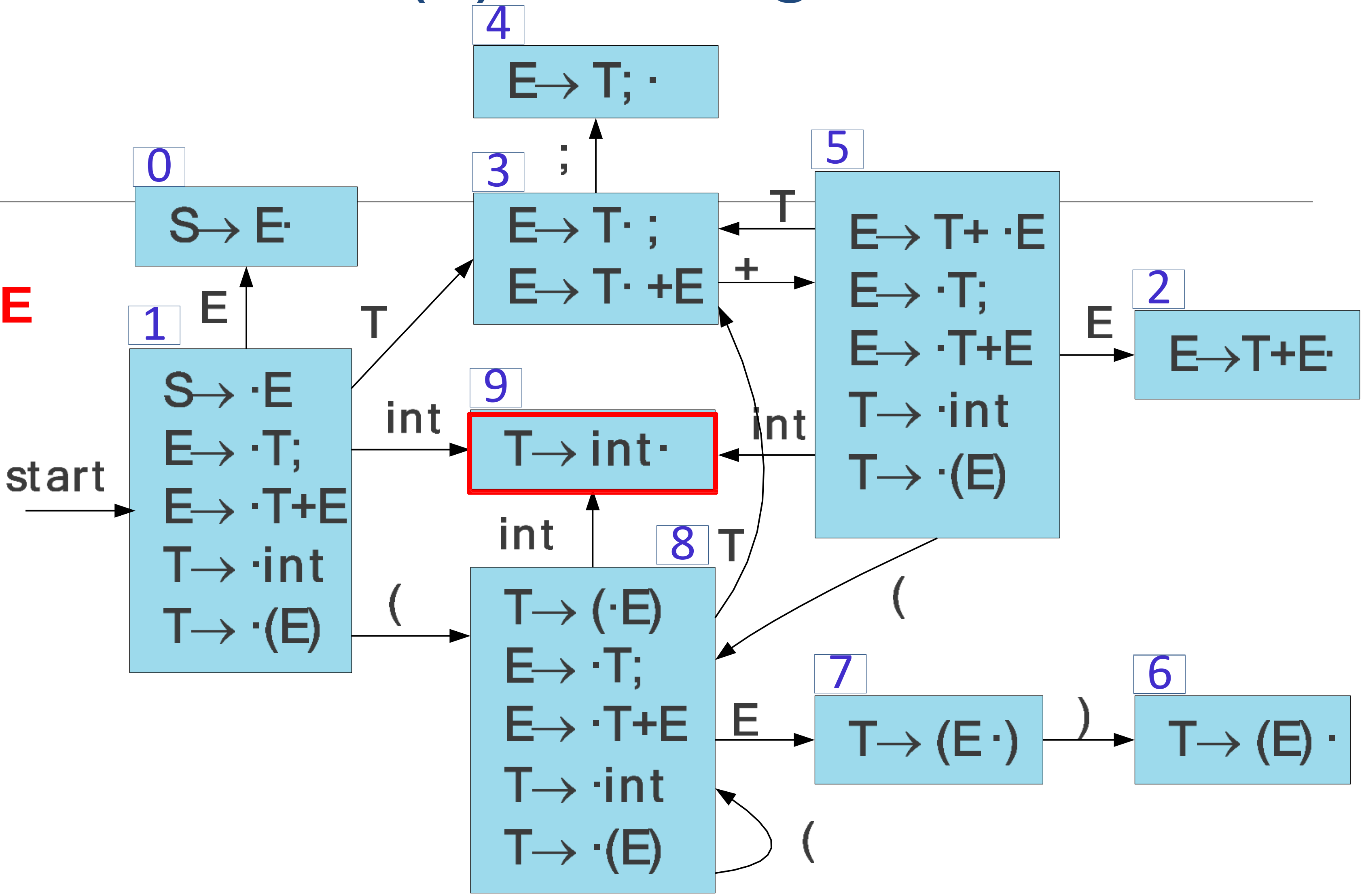


$	T	+	(	T
	0	3	5	8

+	int	;	)	;	$

LR(0) Parsing

S → E
E → T;
E → T + E
T → int
T → (E)

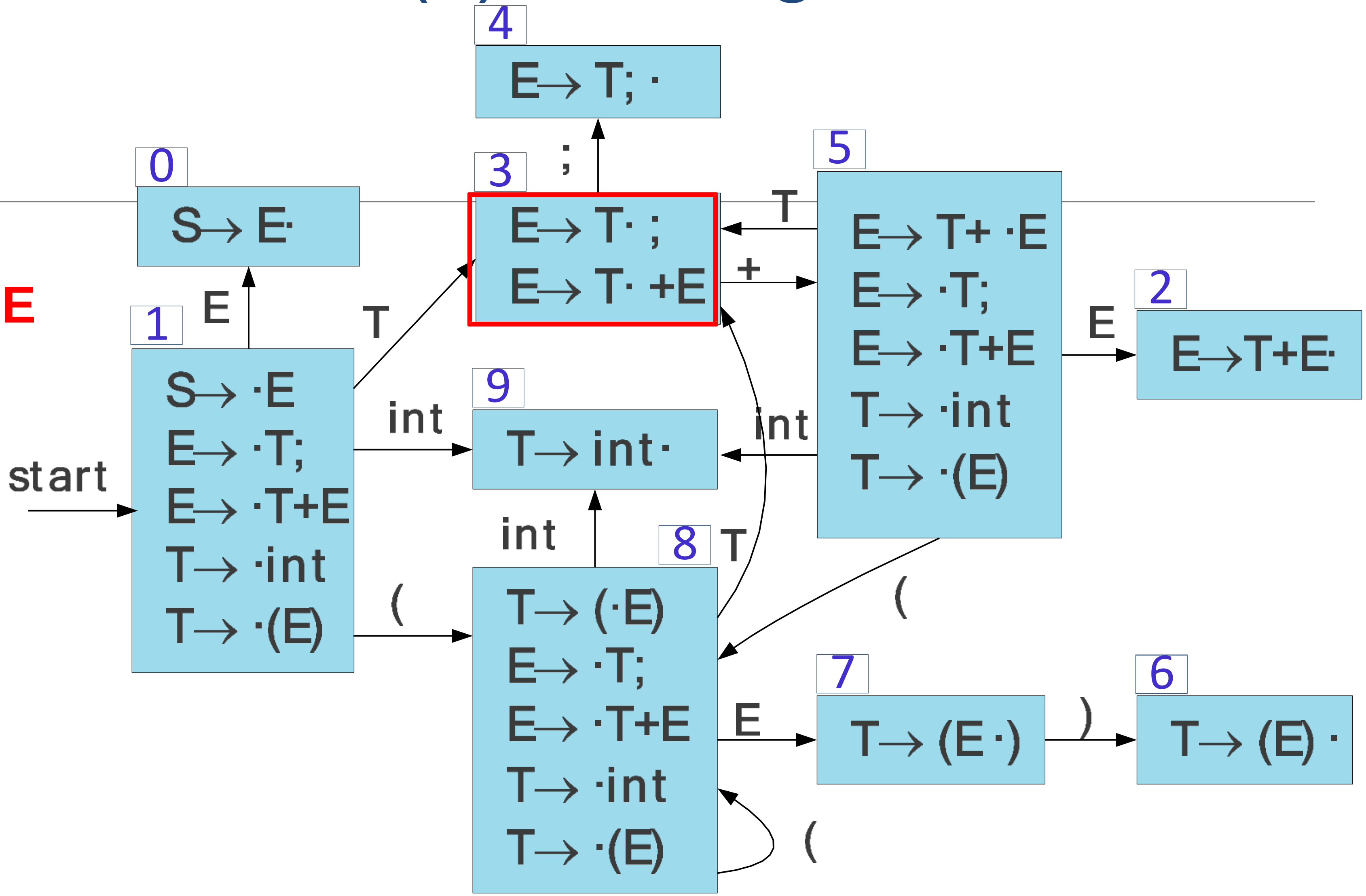


$	T	+	(	T
	0	3	5	8	3

+	int	;	)	;	$

LR(0) Parsing

S → E
E → T;
E → T + E
T → int
T → (E)

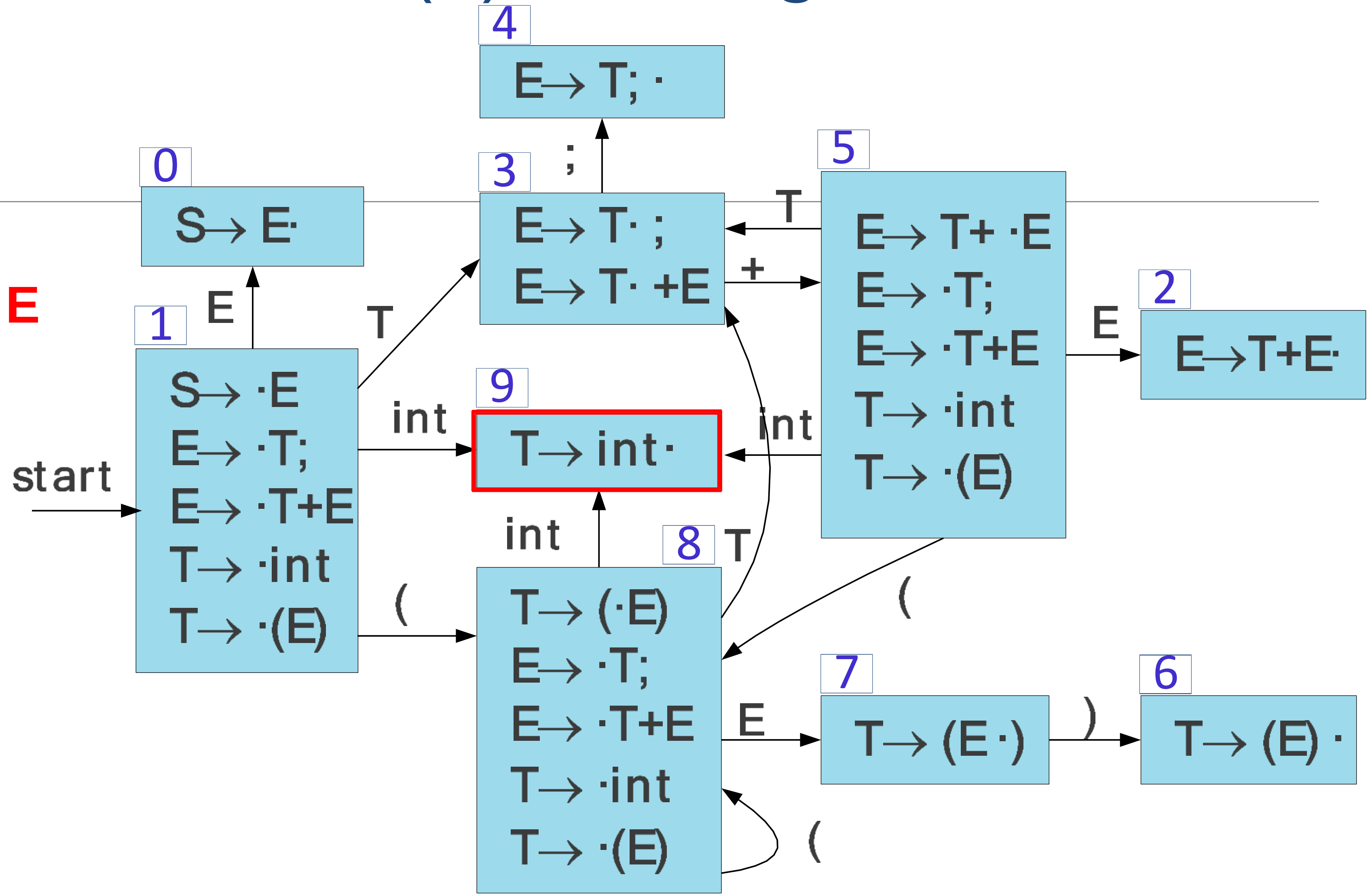


$	T	+	(	T	+
0	3	5	8	3	5

int	;	)	;	$

LR(0) Parsing

$S \rightarrow E$
$E \rightarrow T;$
$E \rightarrow T + E$
$T \rightarrow \mathbf{int}$
$T \rightarrow (E)$

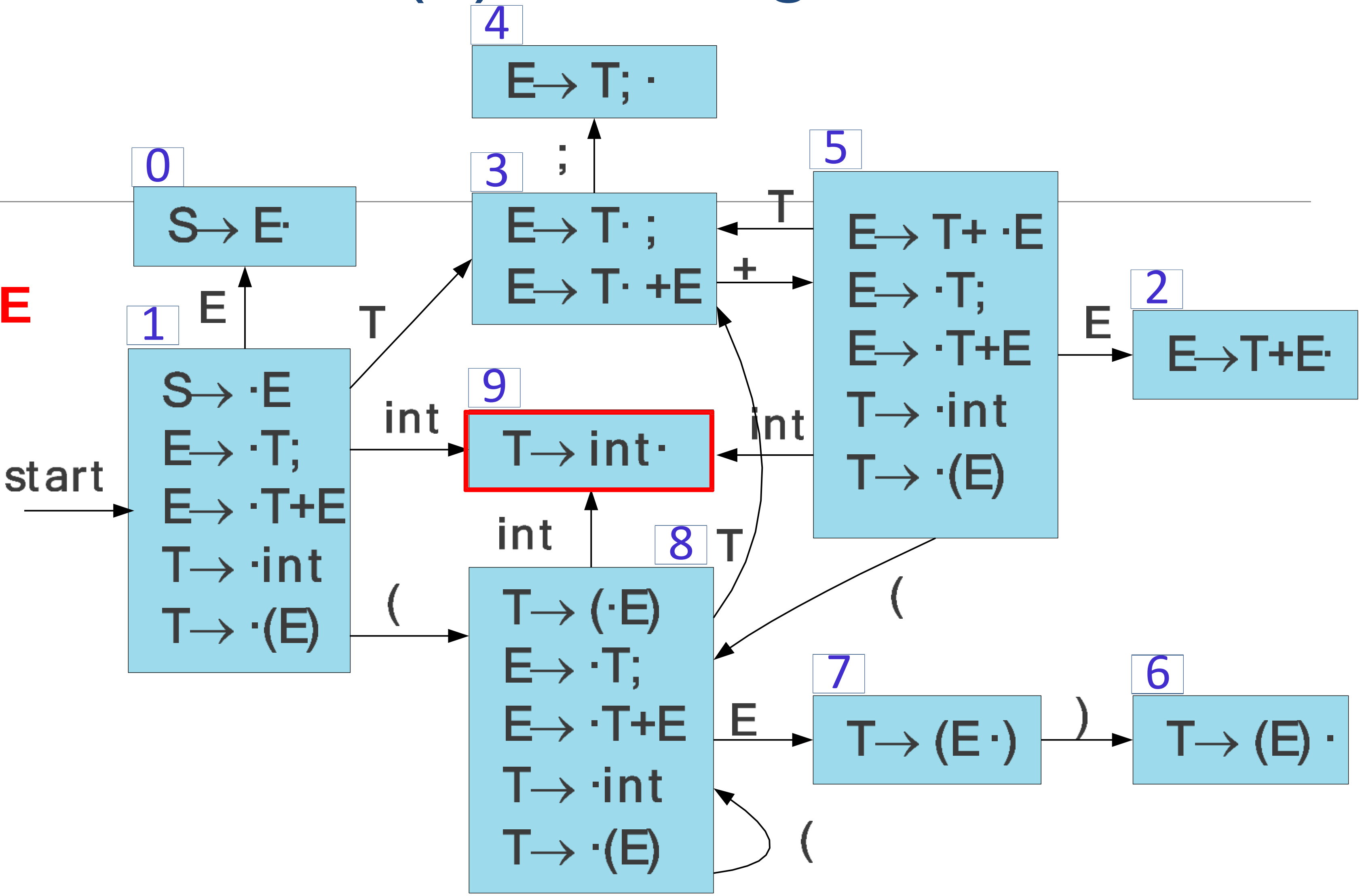


$	T	+	(	T	+	int
0	3	5	8	3	5	9

;	)	;	$

LR(0) Parsing

S → E
E → T;
E → T + E
T → int
T → (E)

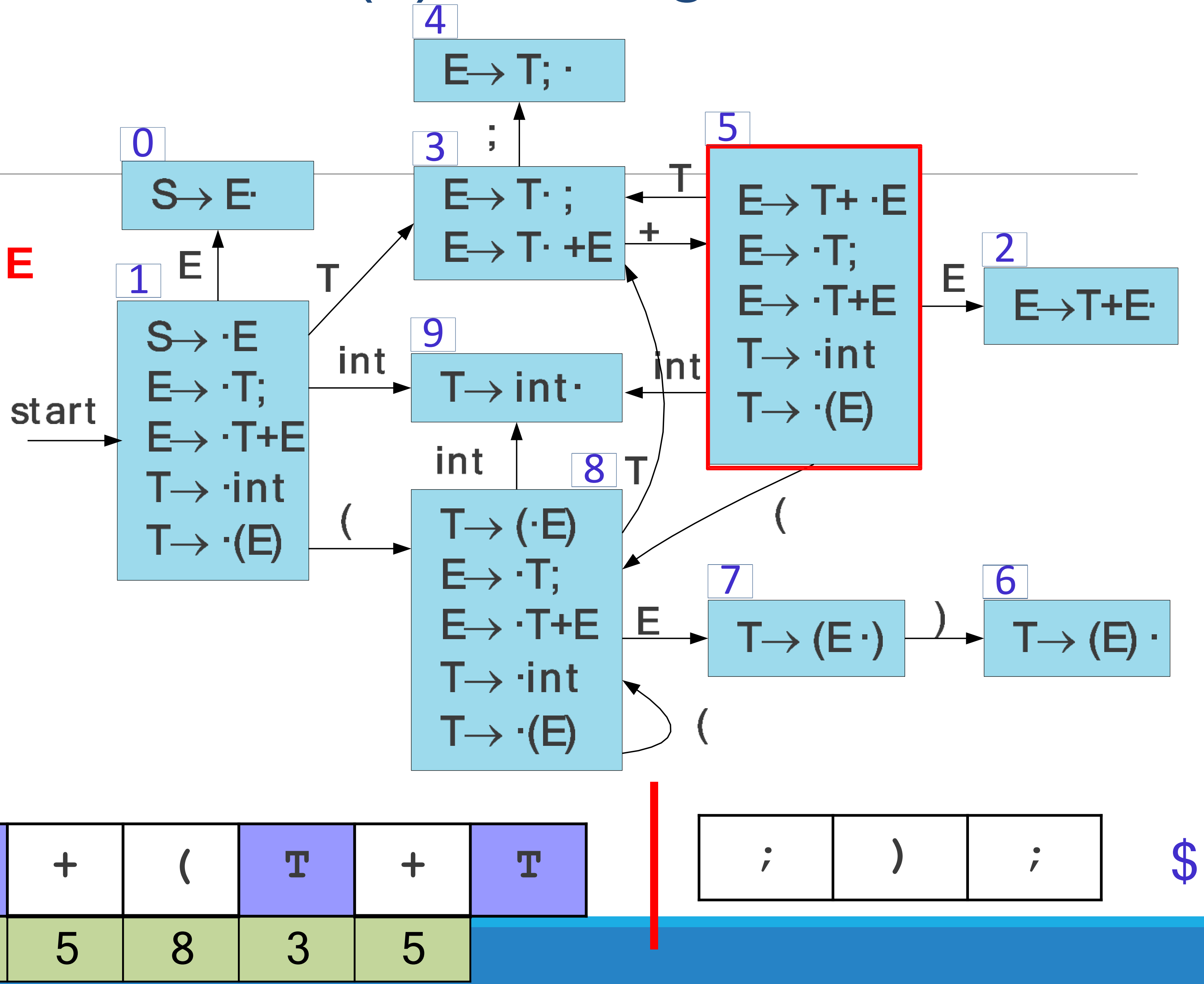


$	T	+	(	T	+
0	3	5	8	3	5

;	)	;	$

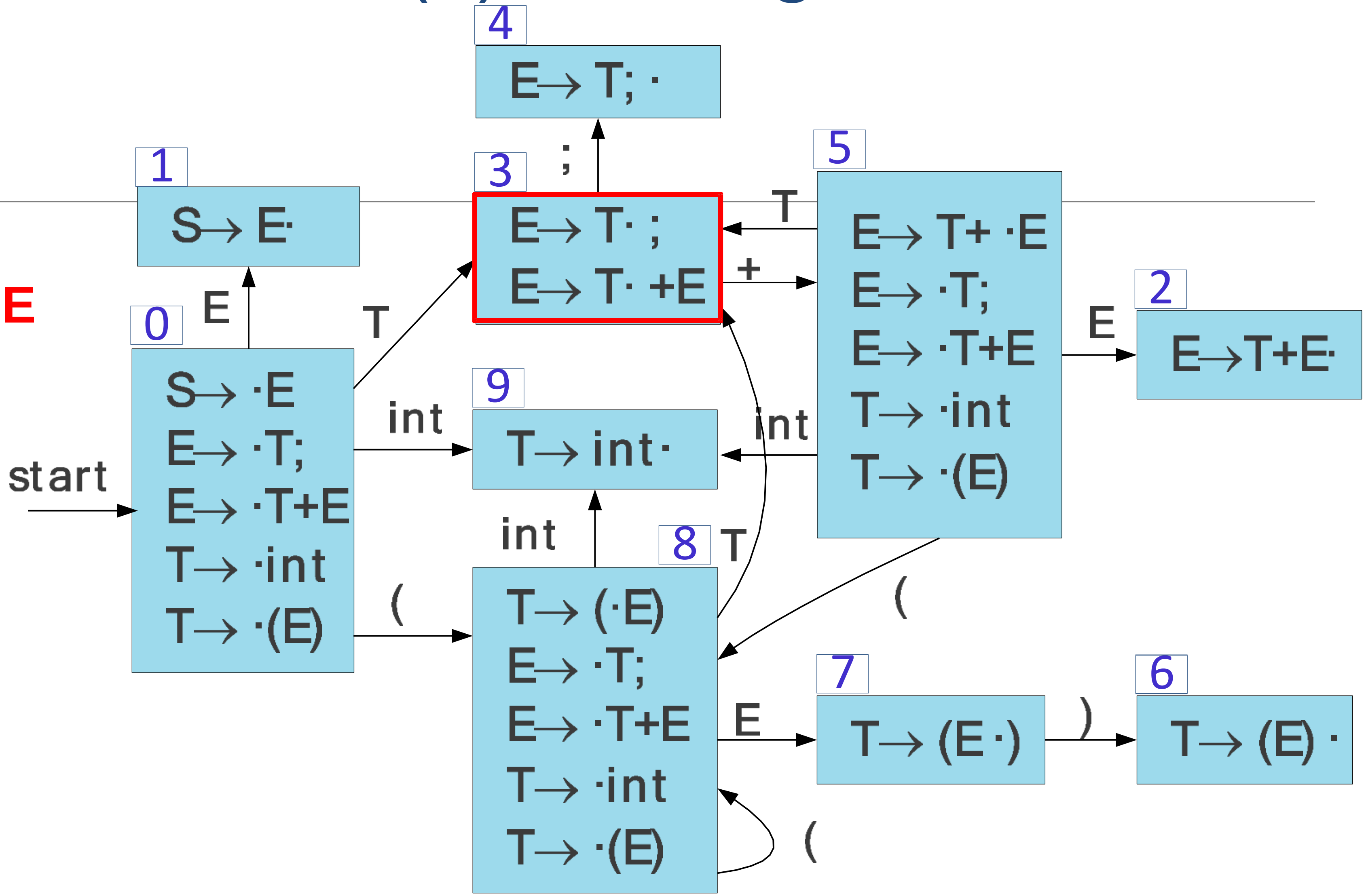
LR(0) Parsing

$S \to E$
$E \to T;$
$E \to T + E$
$T \to int$
$T \to (E)$



LR(0) Parsing

$S \rightarrow E$
$E \rightarrow T;$
$E \rightarrow T + E$
$T \rightarrow int$
$T \rightarrow (E)$

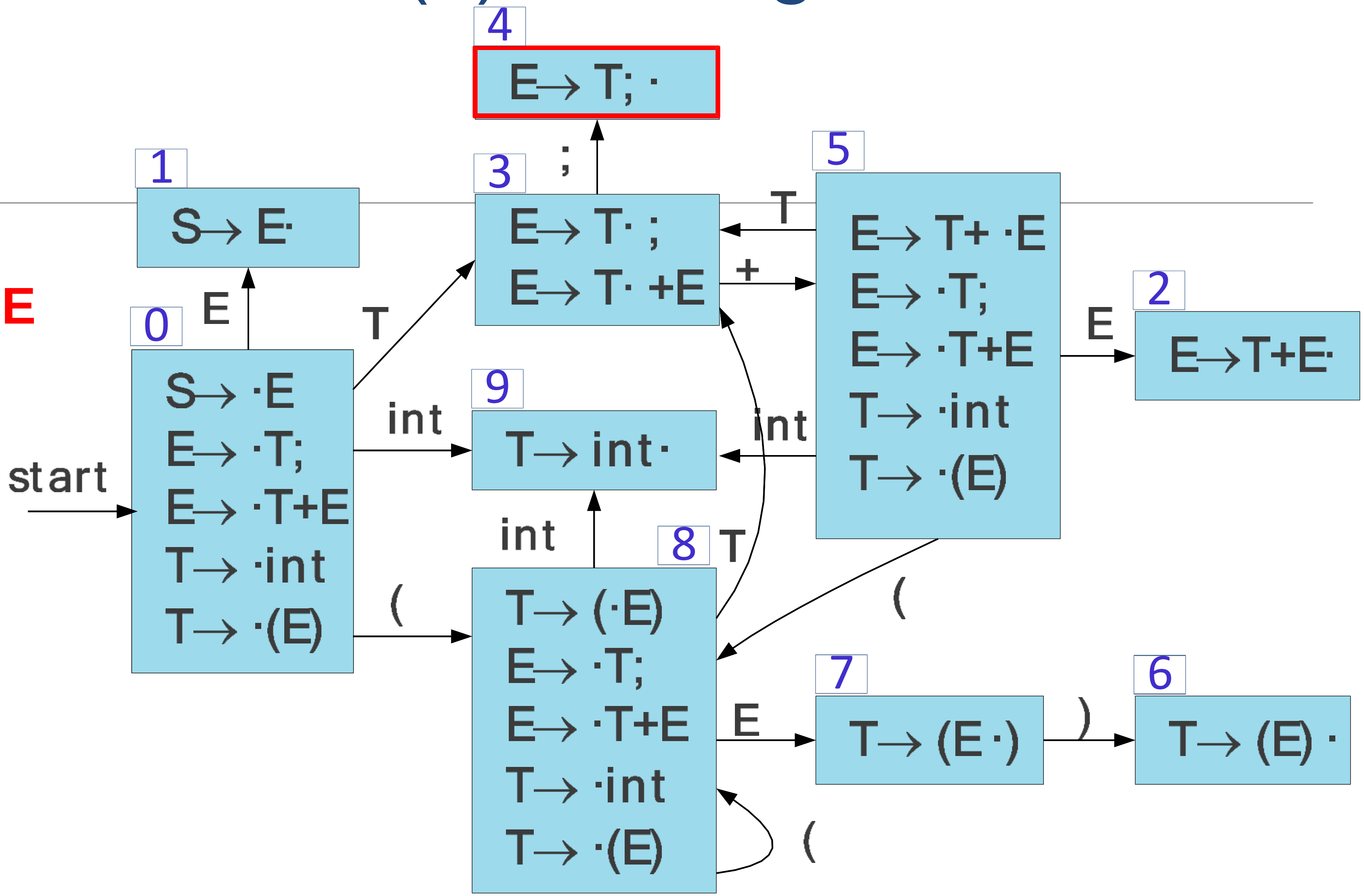


$	T	+	(	T	+	T
0	3	5	8	3	5	3

;	)	;	$

LR(0) Parsing

$S \rightarrow E$
$E \rightarrow T;$
$E \rightarrow T + E$
$T \rightarrow \text{int}$
$T \rightarrow (E)$



$

T	+	(	T	+	T	;

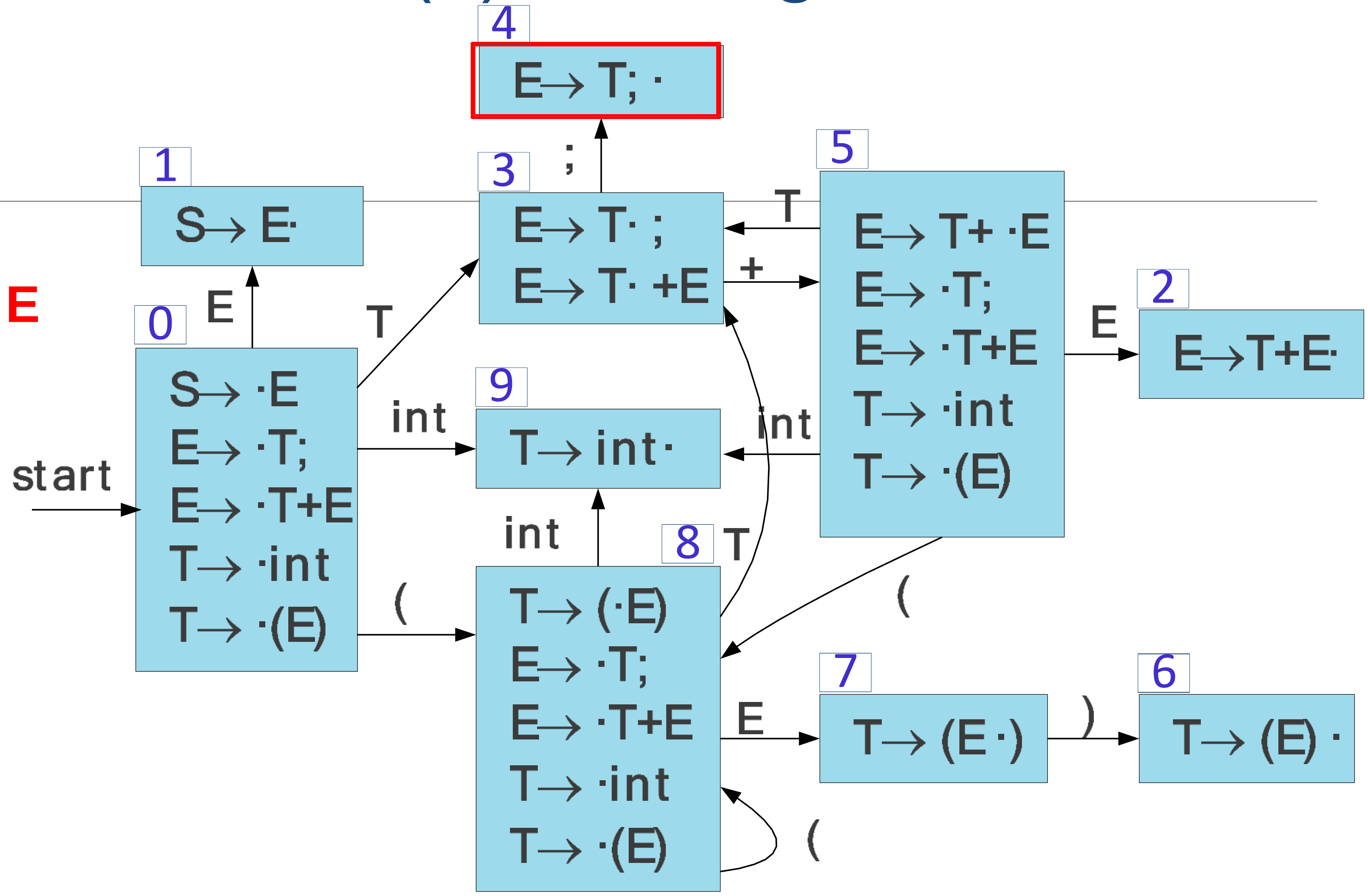
0	3	5	8	3	5	3	4

)	;

$

LR(0) Parsing

S → **E**
E → **T;**
E → **T + E**
T → **int**
T → **(E)**

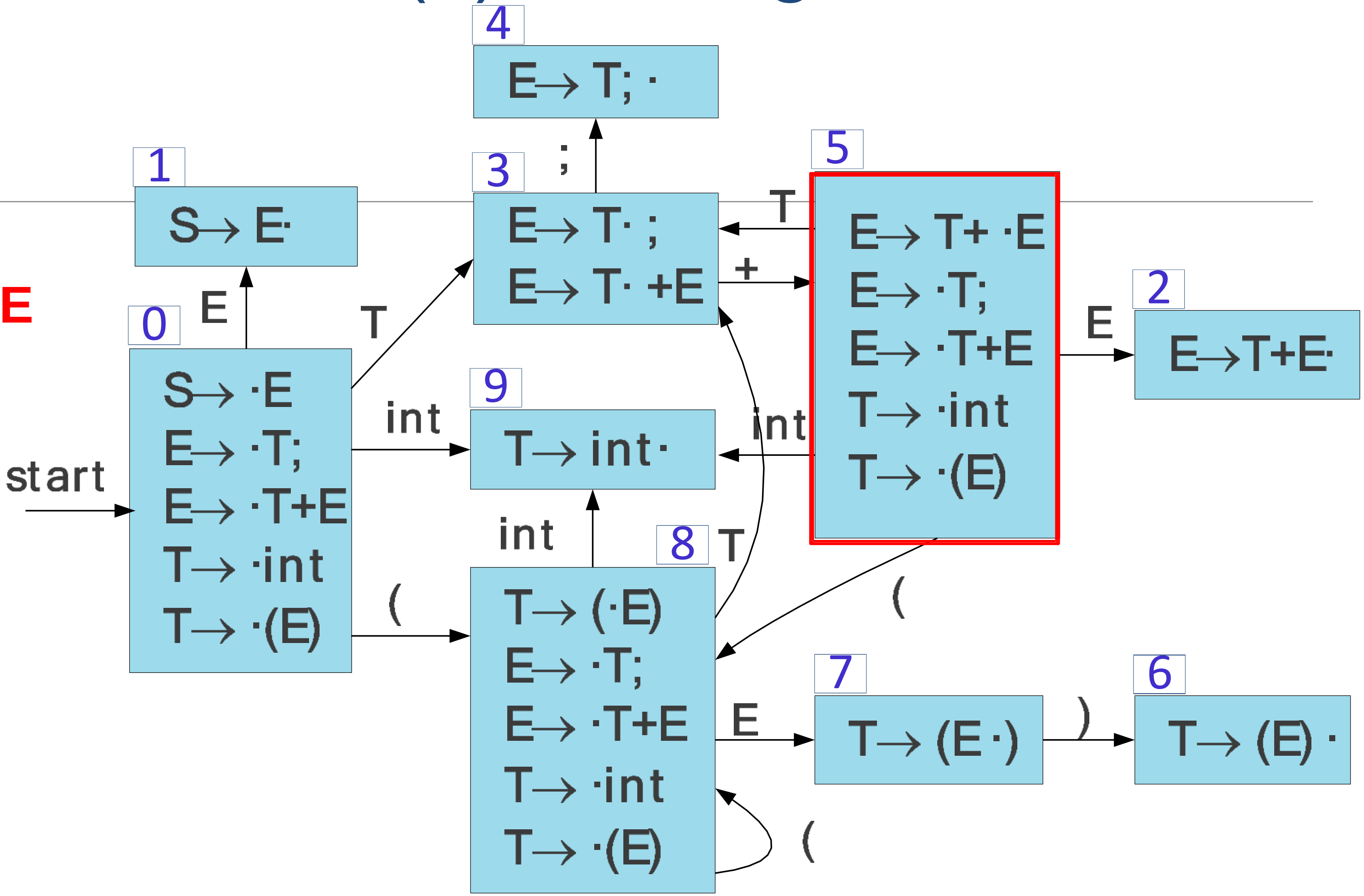


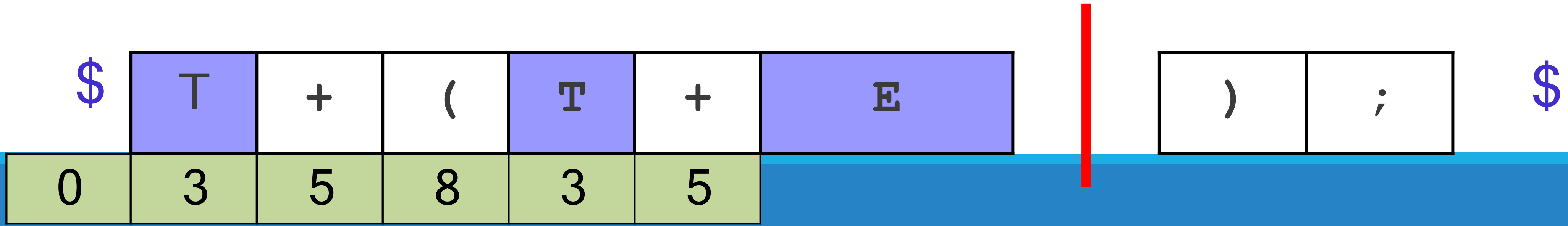
$	T	+	(	T	+
	0	3	5	8	3	5

)	;	$

LR(0) Parsing

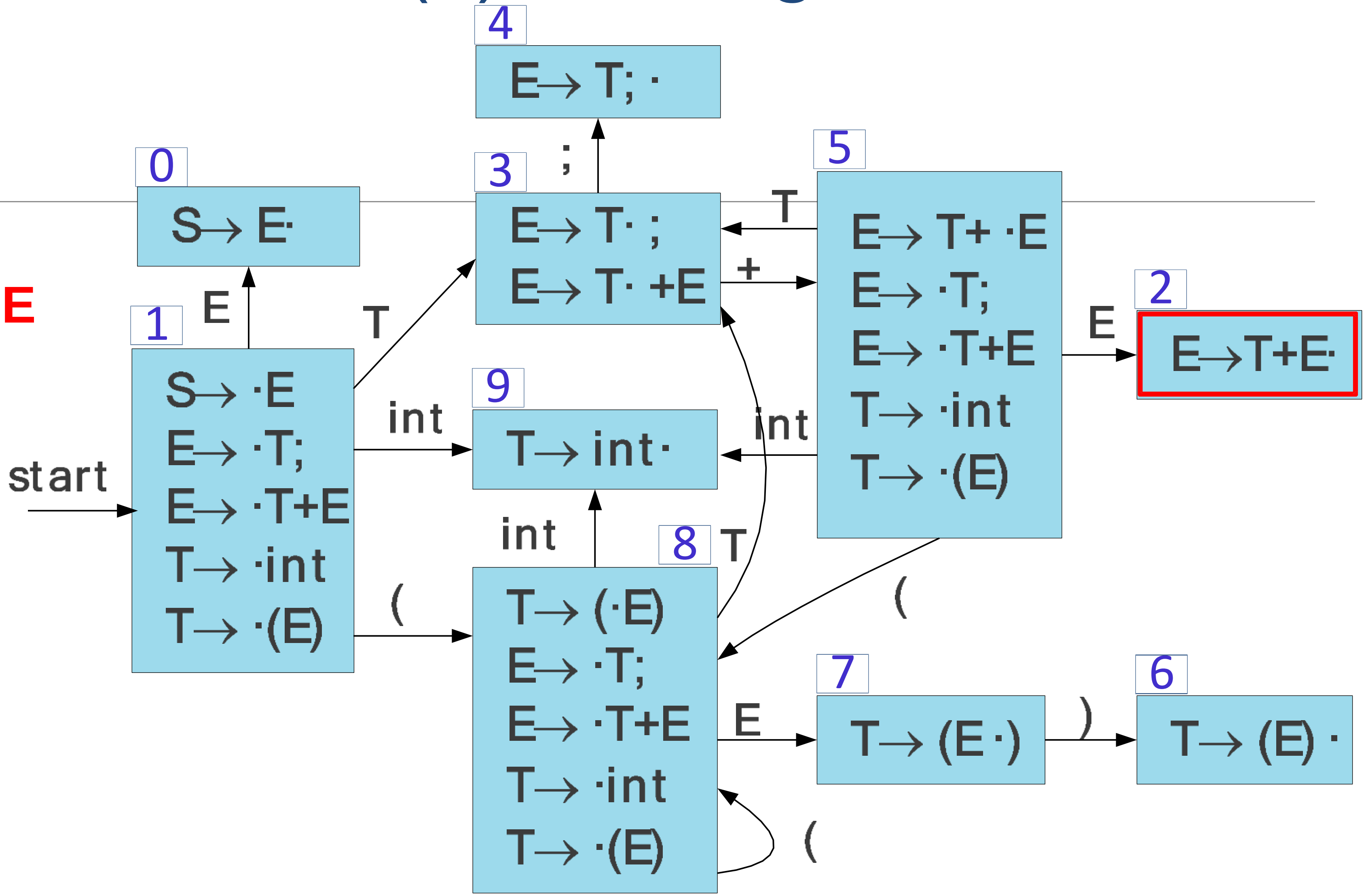
$S \rightarrow E$
$E \rightarrow T;$
$E \rightarrow T + E$
$T \rightarrow \mathbf{int}$
$T \rightarrow (E)$





LR(0) Parsing

S → **E**
E → **T;**
E → **T + E**
T → **int**
T → **(E)**

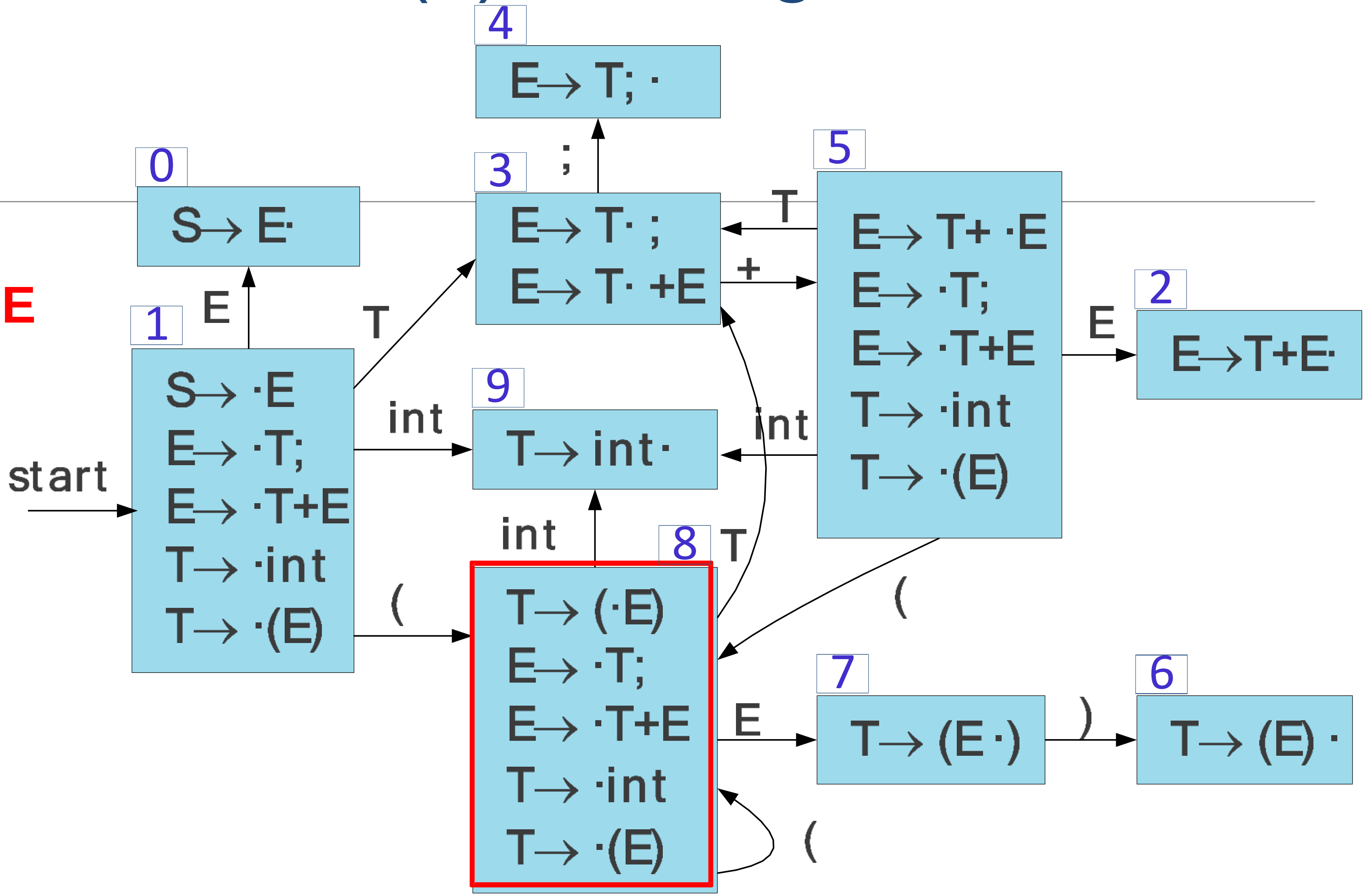


$	T	+	(	T	+	E
0	3	5	8	3	5	2

)	;	$

LR(0) Parsing

$S \rightarrow E$
$E \rightarrow T;$
$E \rightarrow T + E$
$T \rightarrow int$
$T \rightarrow (E)$

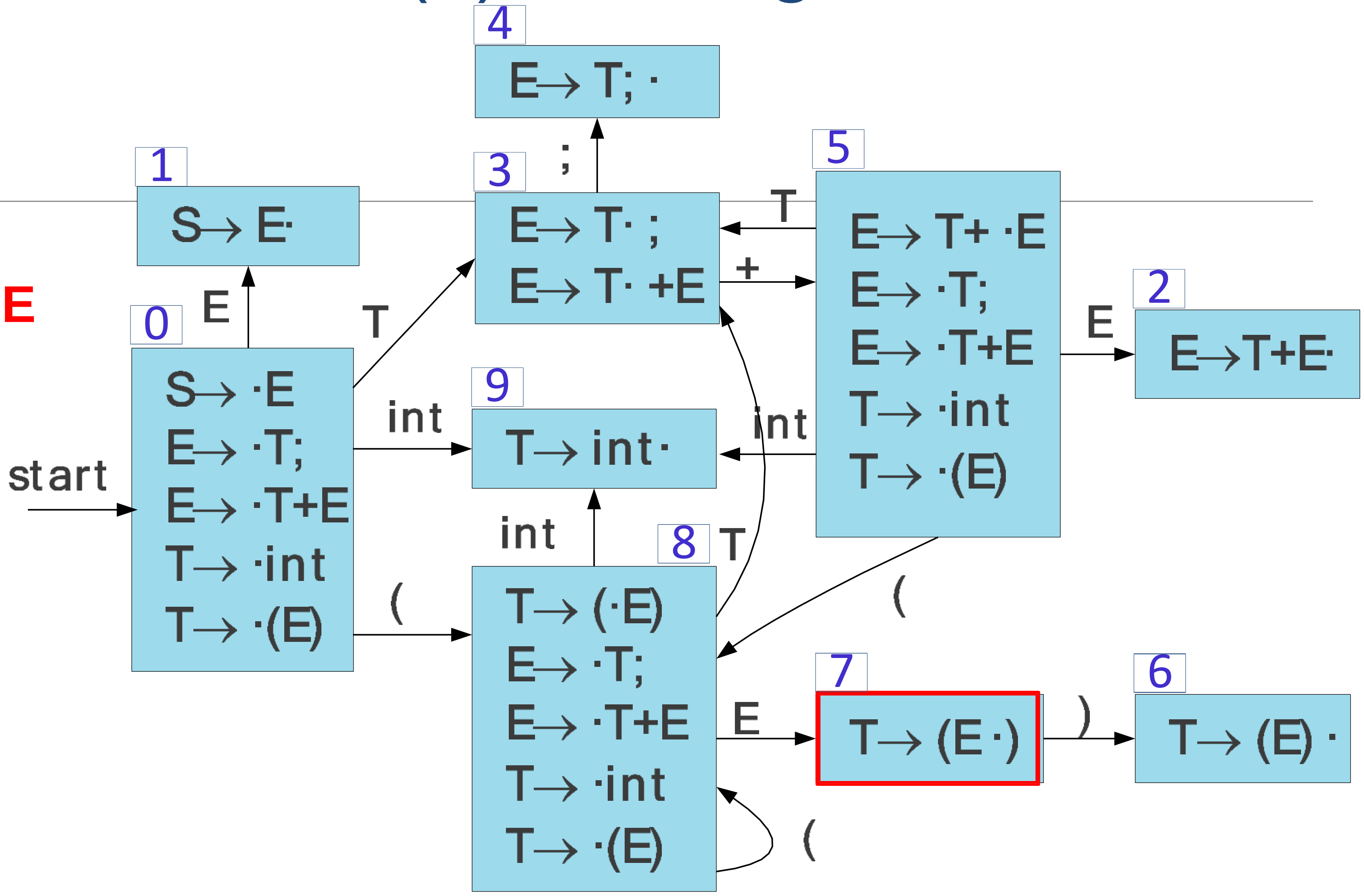


$	T	+	(
0	3	5	8

)	;	$

LR(0) Parsing

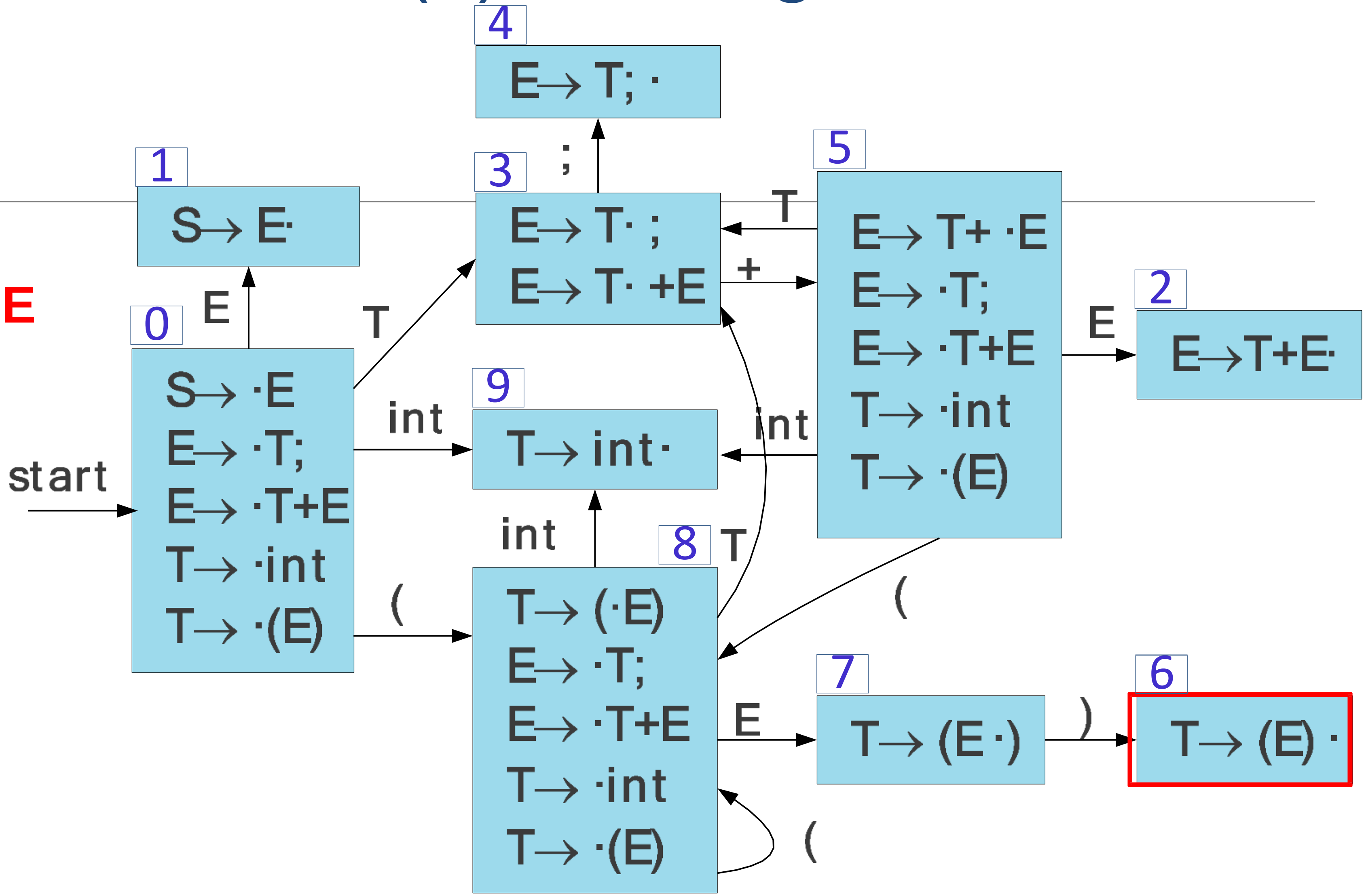
S → E
E → T;
E → T + E
T → int
T → (E)



$	T	+	(	E	|	)	;	$
0	3	5	8	7				

LR(0) Parsing

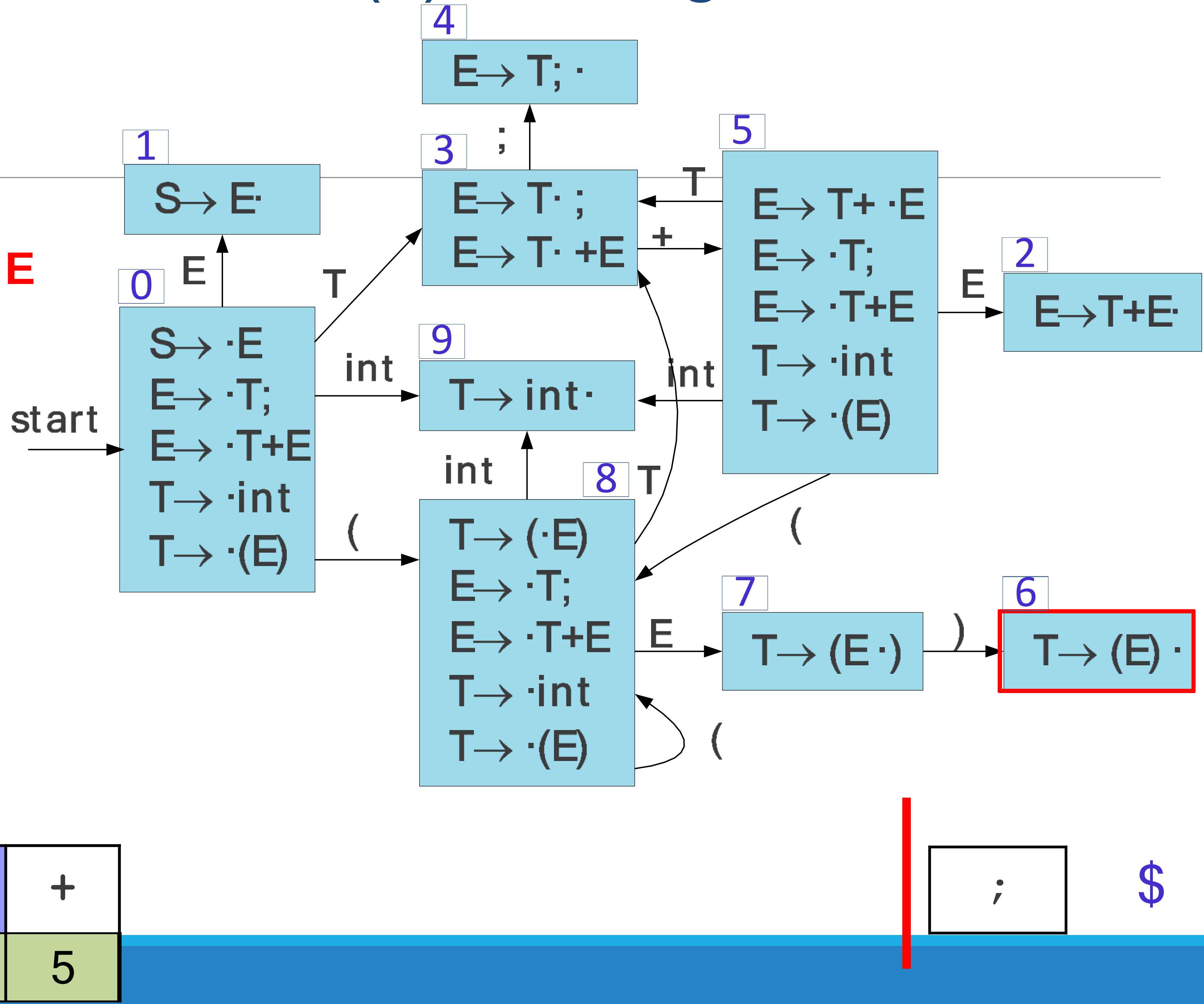
$S \rightarrow E$
$E \rightarrow T;$
$E \rightarrow T + E$
$T \rightarrow int$
$T \rightarrow (E)$



\$	T	+	(	E	)		; \$
0	3	5	8	7	6		

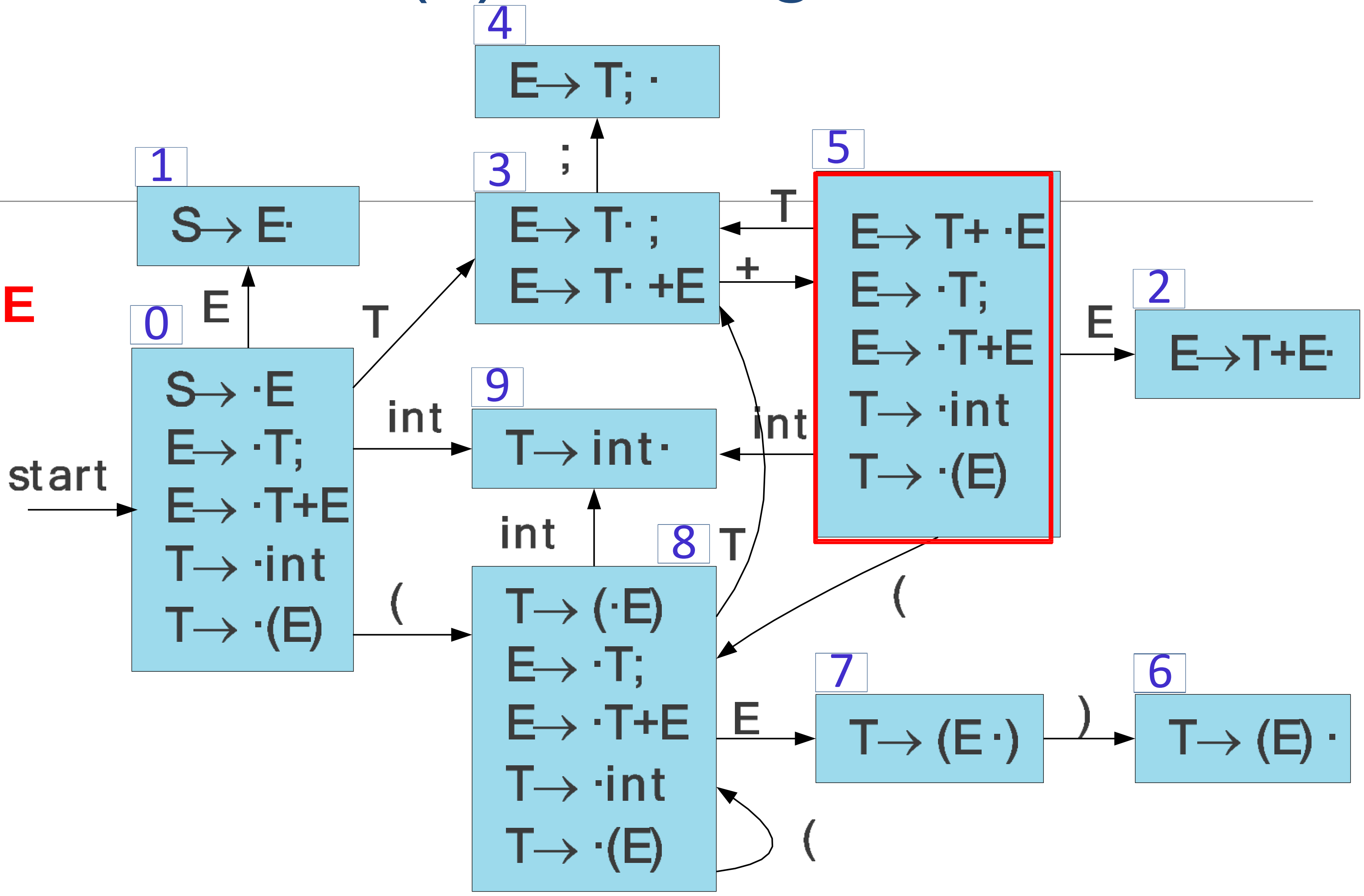
LR(0) Parsing

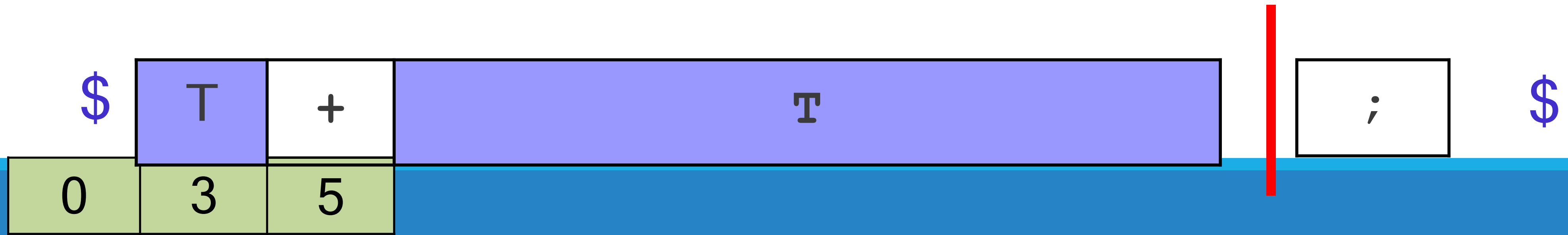
S → E
E → T;
E → T + E
T → int
T → (E)



LR(0) Parsing

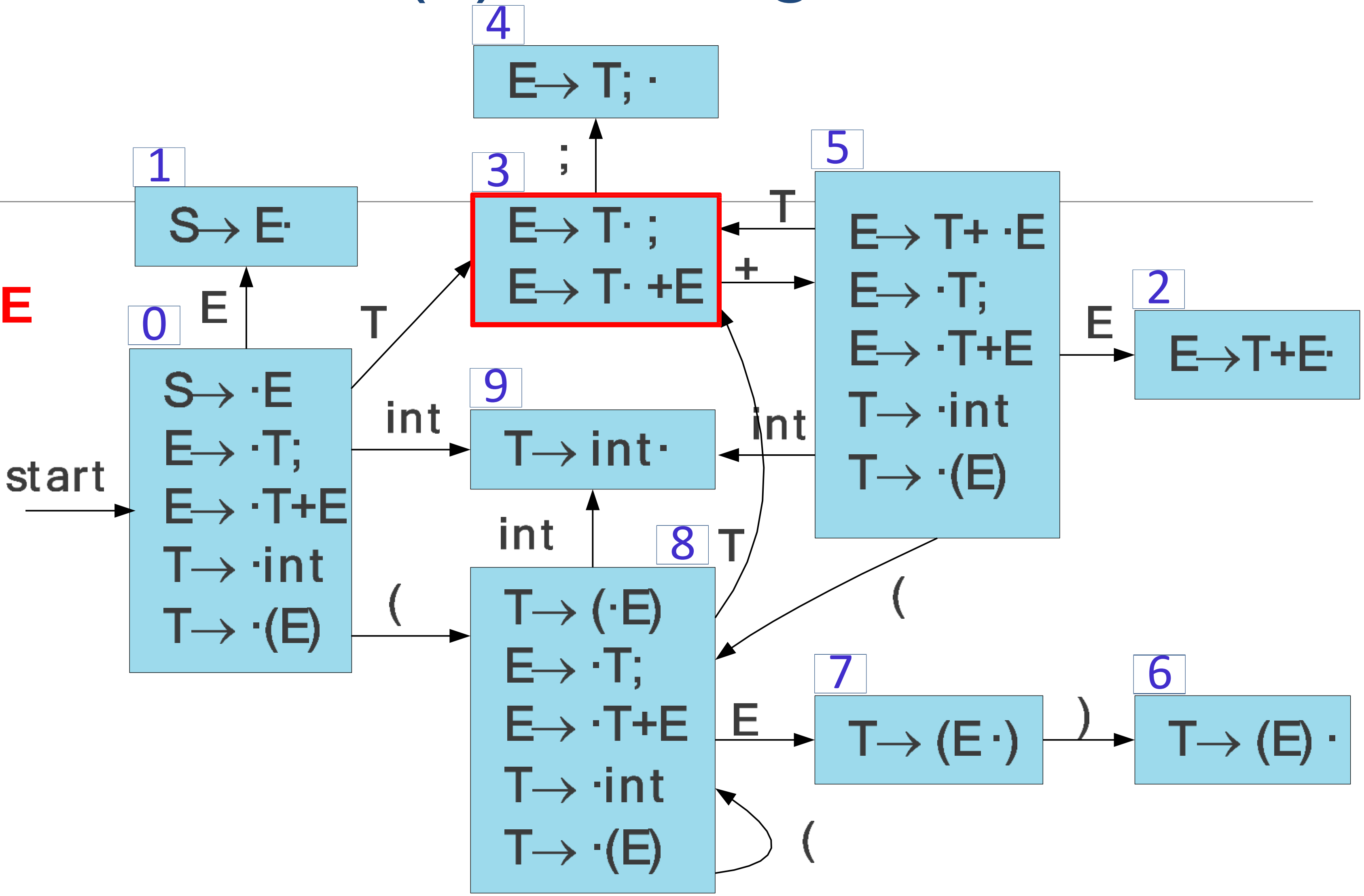
$S \rightarrow E$
$E \rightarrow T;$
$E \rightarrow T + E$
$T \rightarrow \text{int}$
$T \rightarrow (E)$





LR(0) Parsing

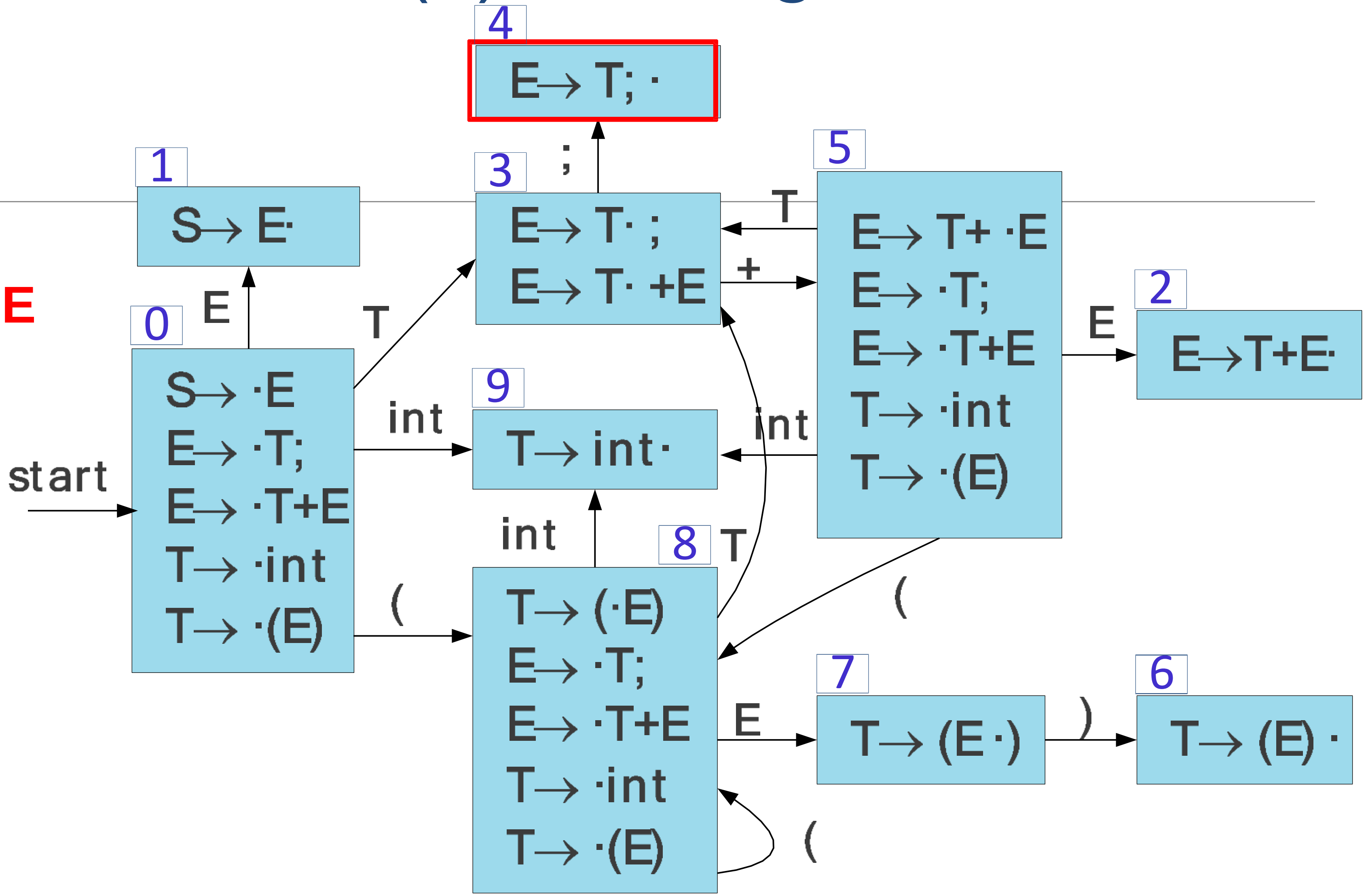
$S \rightarrow E$
$E \rightarrow T;$
$E \rightarrow T + E$
$T \rightarrow int$
$T \rightarrow (E)$



$	T	+	T		;	$
0	3	5	3			

LR(0) Parsing

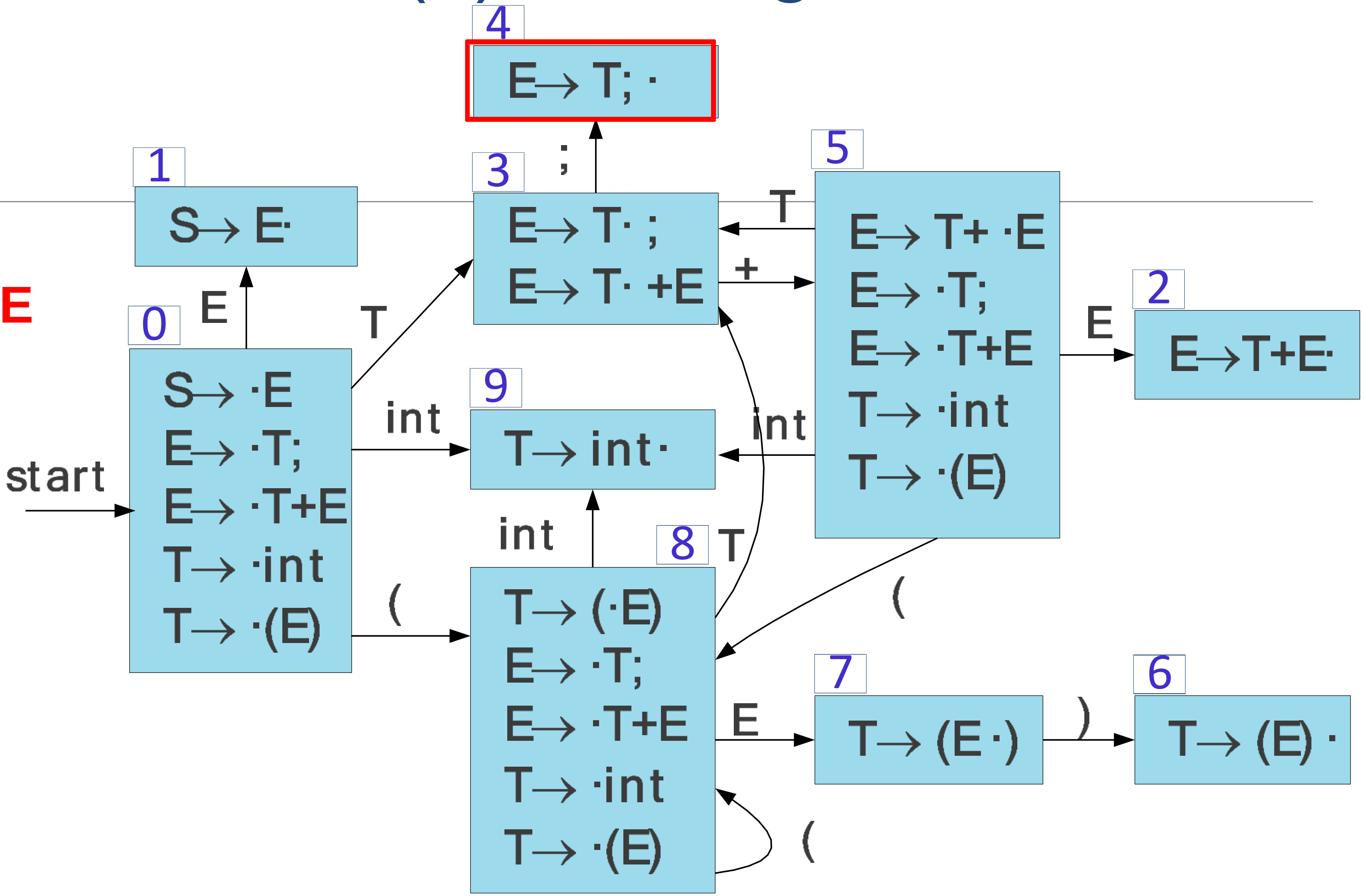
$S \rightarrow E$
$E \rightarrow T;$
$E \rightarrow T + E$
$T \rightarrow int$
$T \rightarrow (E)$



\$	T	+	T	;	\$
0	3	5	3	4	

LR(0) Parsing

S → E
E → T;
E → T + E
T → int
T → (E)

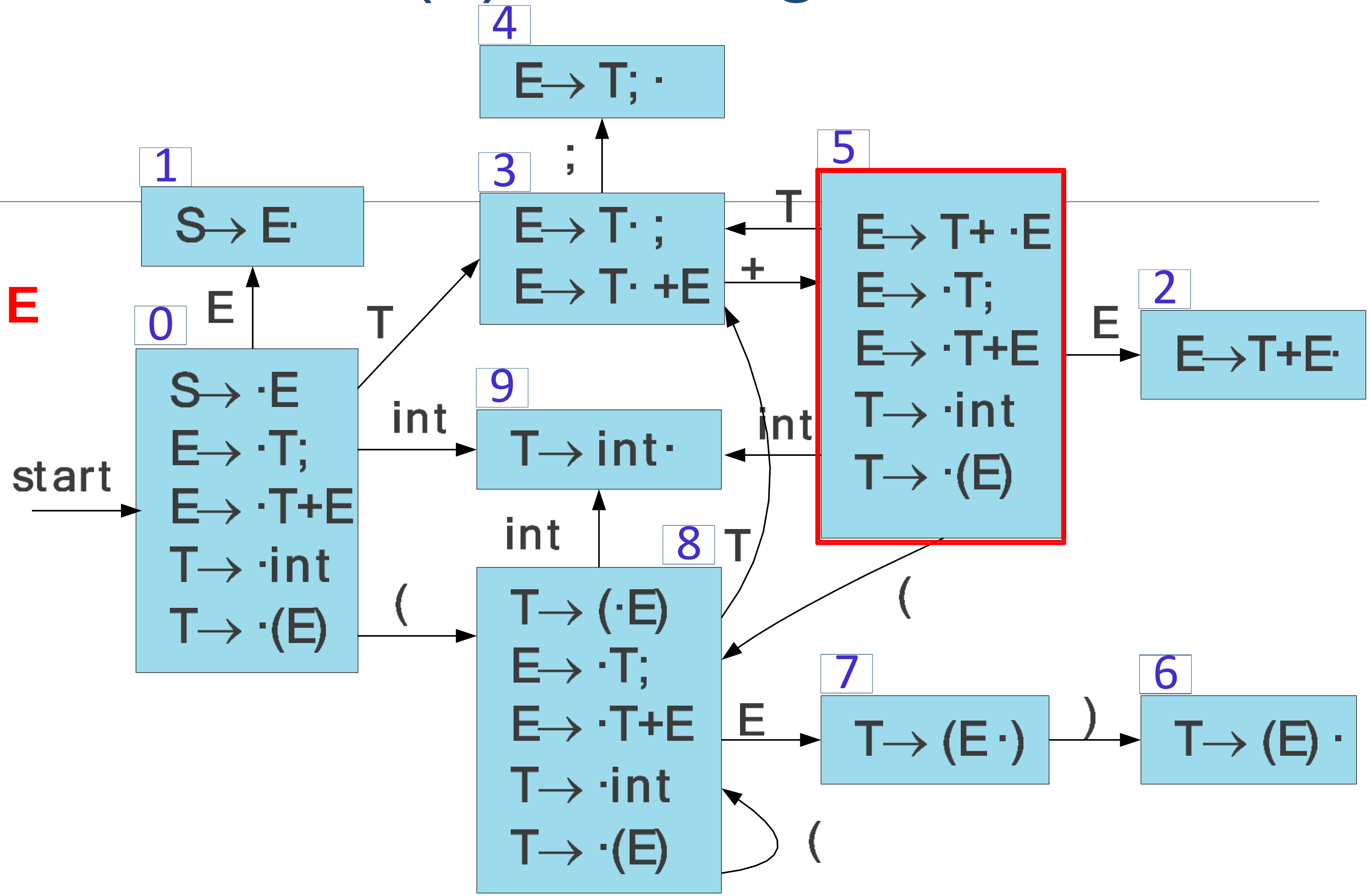


$	T	+
0	3	5

$

LR(0) Parsing

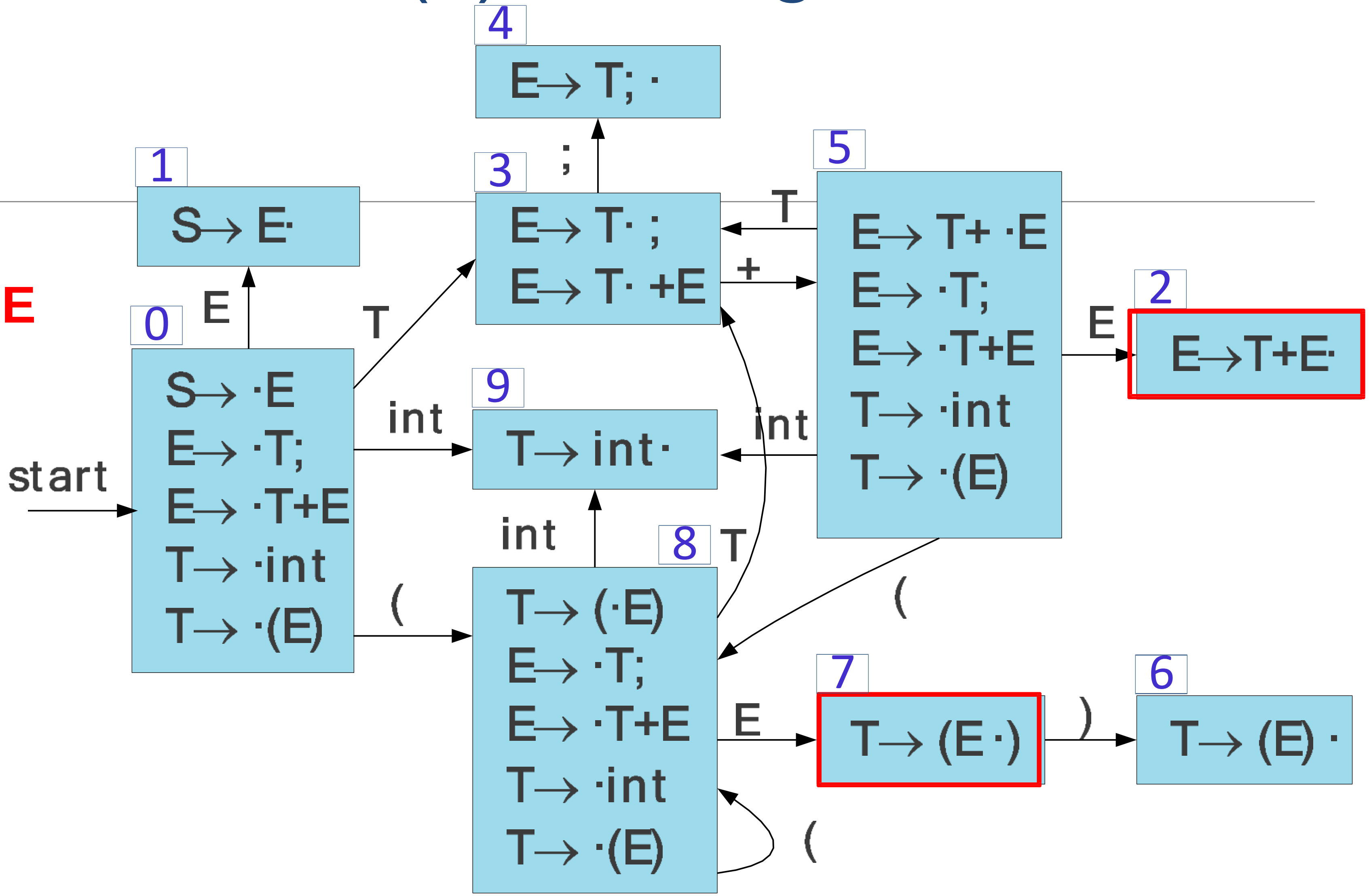
$S \rightarrow E$
$E \rightarrow T;$
$E \rightarrow T + E$
$T \rightarrow \text{int}$
$T \rightarrow (E)$



$	T	+	E	$
0	3	5		

LR(0) Parsing

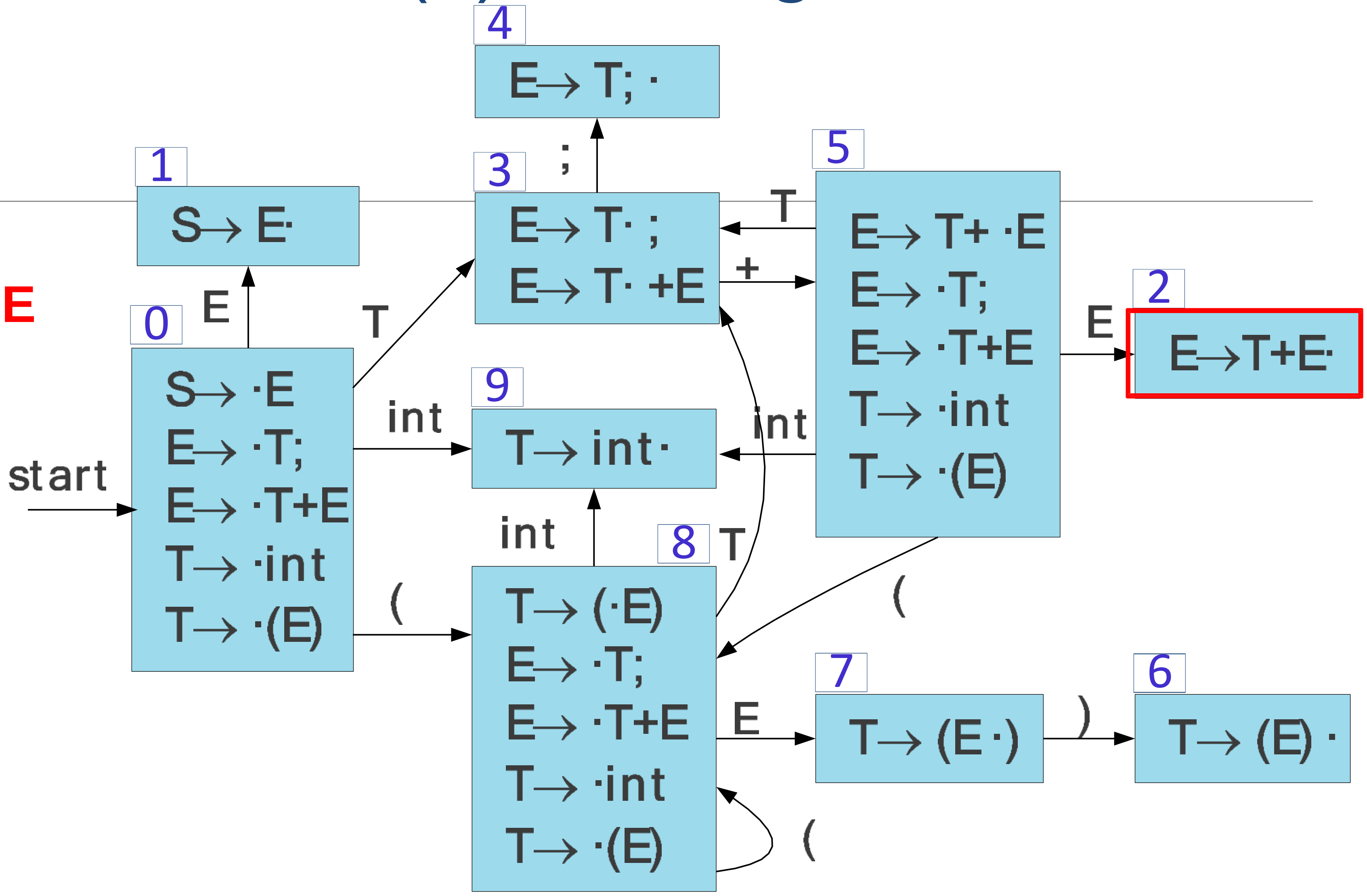
$S \rightarrow E$
$E \rightarrow T;$
$E \rightarrow T + E$
$T \rightarrow \text{int}$
$T \rightarrow (E)$



$	T	+	E	|	$
0	3	5	2		

LR(0) Parsing

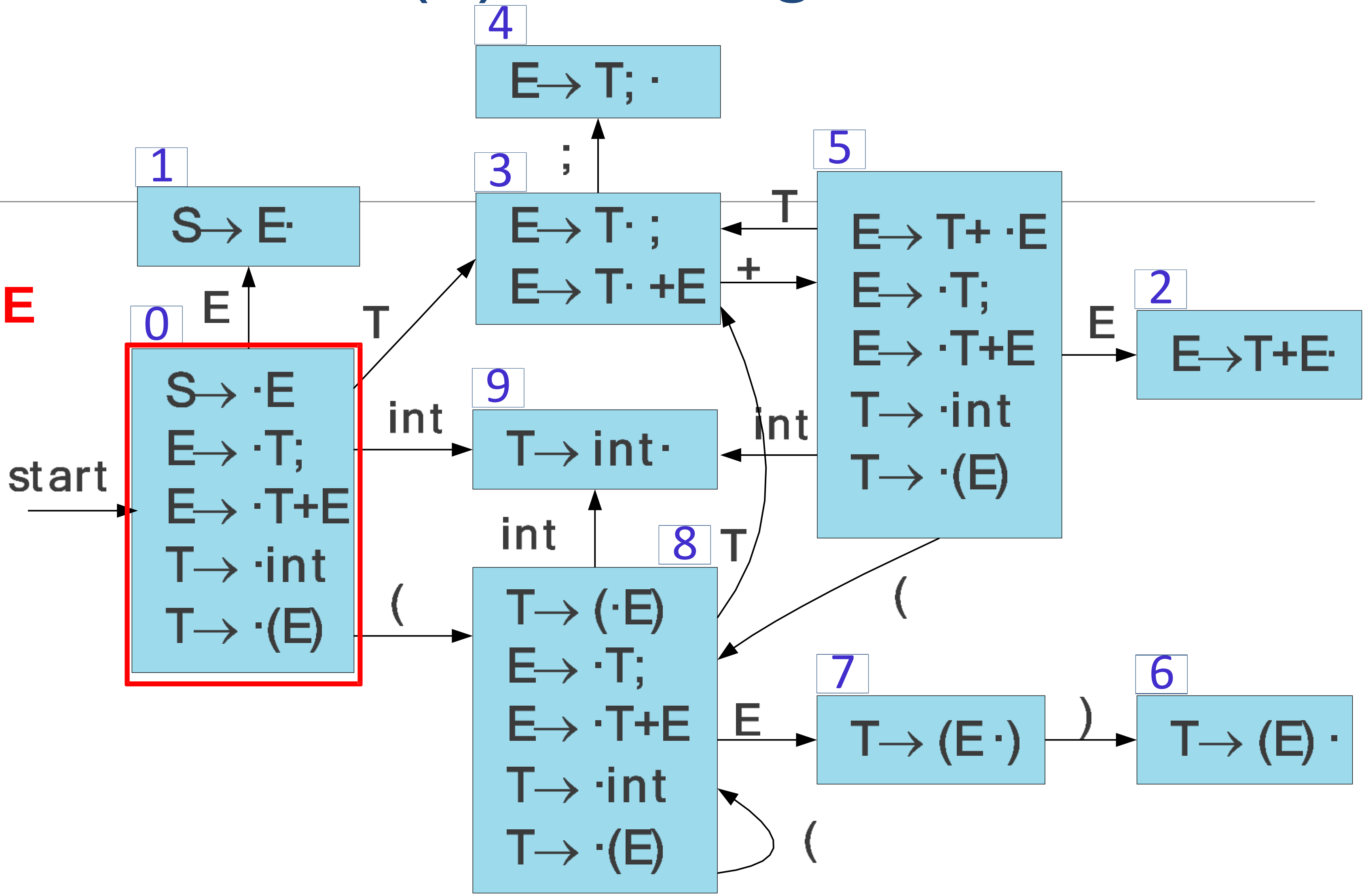
S → E
E → T;
E → T + E
T → int
T → (E)

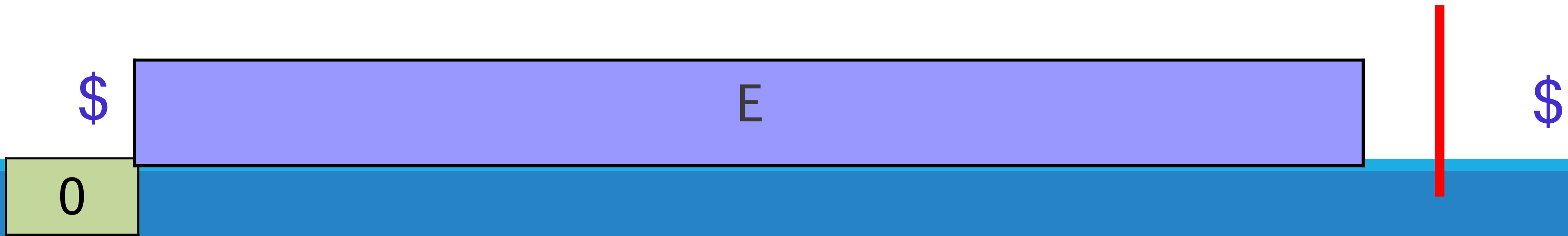


$ 0 $

LR(0) Parsing

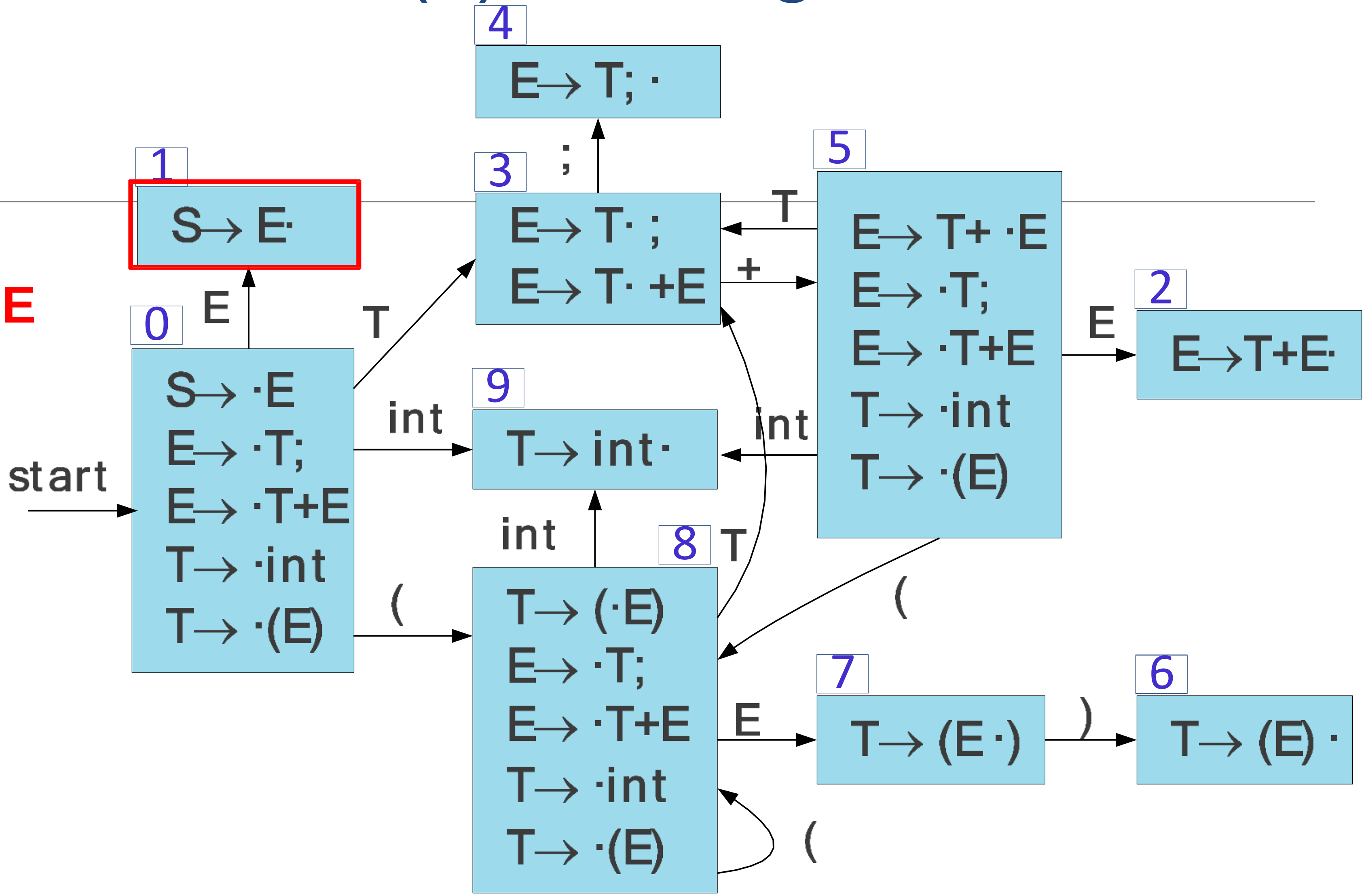
$S \rightarrow E$
$E \rightarrow T;$
$E \rightarrow T + E$
$T \rightarrow \text{int}$
$T \rightarrow (E)$

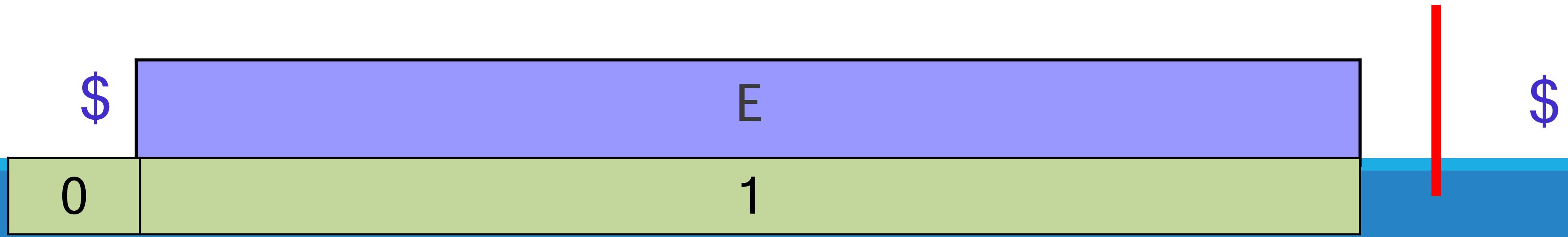




LR(0) Parsing

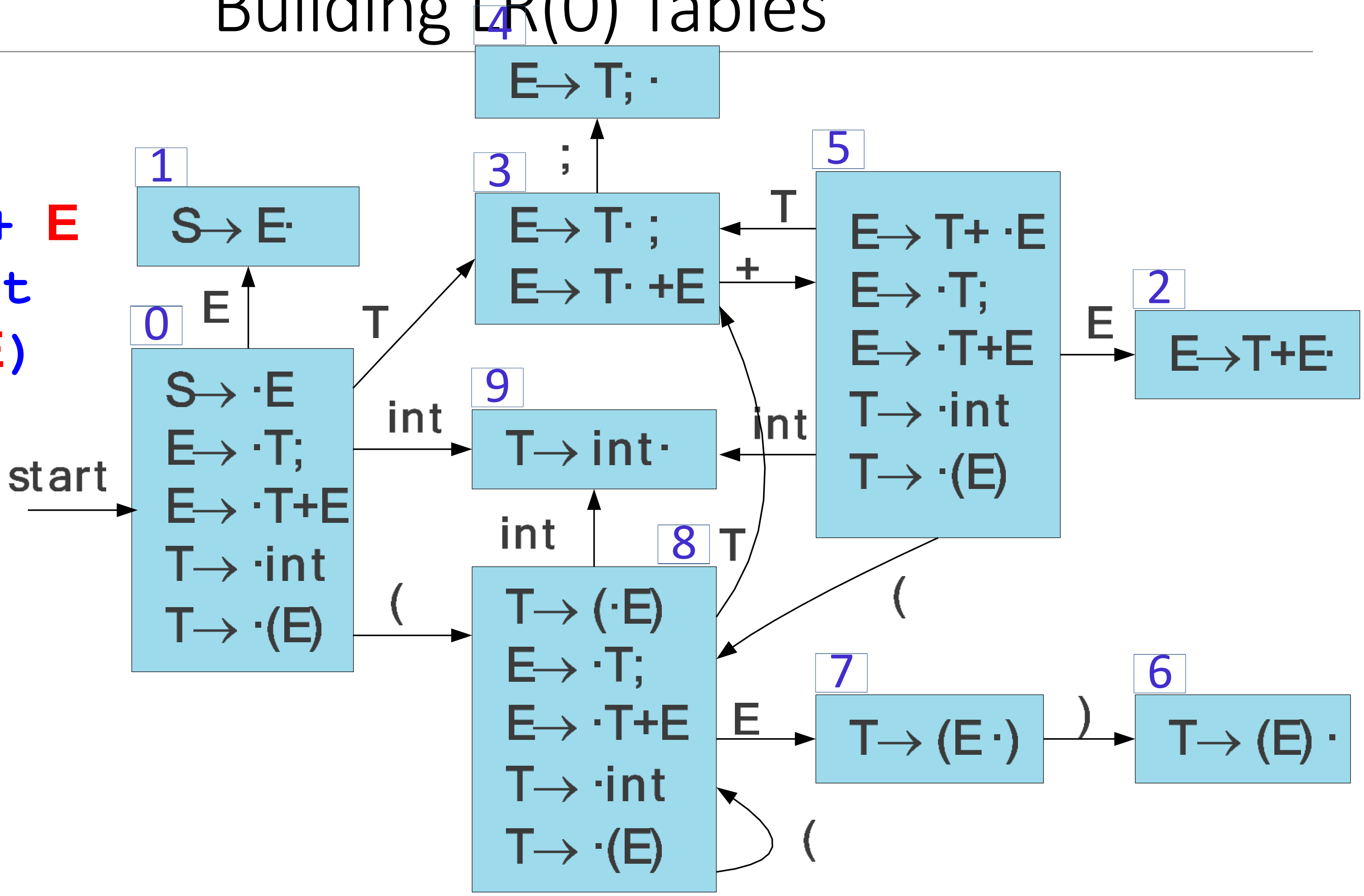
S → E
E → T;
E → T + E
T → int
T → (E)





Building LR(0) Tables

1. S → E
2. E → T;
3. E → T + E
4. T → int
5. T → (E)



LR Tables

	ACTION								GOTO	
state	int	+	;	(	)	E	T	$	E	T
0	s9			s8		s1	s3		1	3
1								acc		
2	r3	r3	r3	r3	r3	r3	r3	r3		
3		s5	s4							
4	r2	r2	r2	r2	r2	r2	r2	r2		
5	s9			s8		s2	s3		2	3
6	r5	r5	r5	r5	r5	r5	r5	r5		
7					s6					
8	s9			s8		s7	s3		7	3
9	r4	r4	r4	r4	r4	r4	r4	r4		

LR Tables

state	int	+	;	(	)	E	T	$	Action
0	9			8		1	3		Shift
1								acc	Accept
2									Reduce **E** → **T + E**
3		5	4						Shift
4									Reduce **E** → **T ;**
5	9			8		2	3		Shift
6									Reduce **T** → **(E)**
7					6				Shift
8	9			8		7	3		Shift
9									Reduce **T** → **int**

Limitations of LR- LR Conflicts

A **shift/reduce conflict** is an error where a shift/reduce parser cannot tell whether to shift a token or perform a reduction.

A **reduce/reduce conflict** is an error where a shift/reduce parser cannot tell which of many reductions to perform.

A grammar whose handle-finding automaton contains a shift/reduce conflict or a reduce/reduce conflict is not LR(0).

shift/reduce conflict

$V_T = \{a, b\}$

$V_N = \{S, A\}$

$S = S$

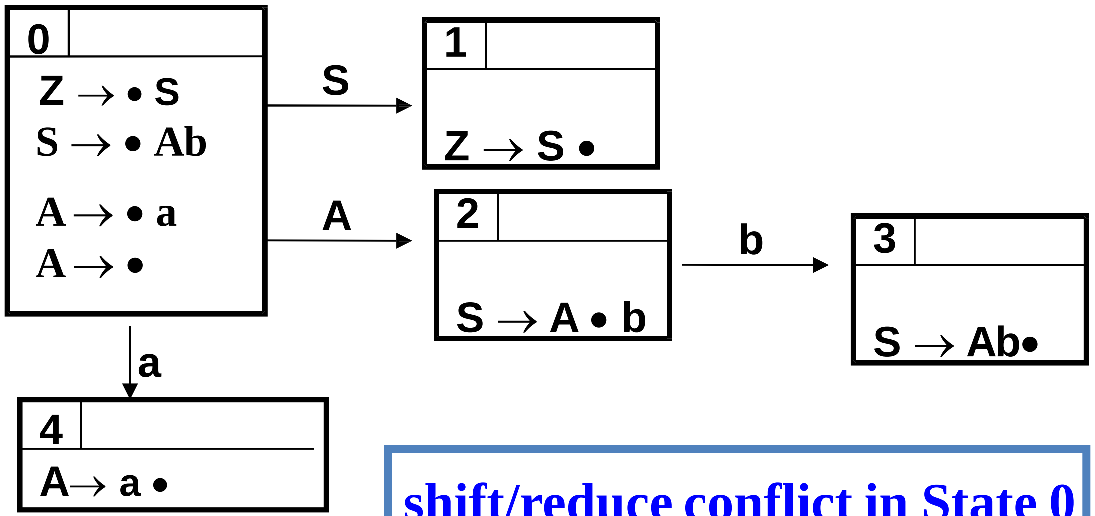
P:

{ (1) $S \rightarrow Ab$

(2) $A \rightarrow \varepsilon$

(3) $A \rightarrow a$

}



shift/reduce conflict in State 0

(1) Shift item: **A → • a**

(2) Reducible item: **A → •**

reduce/reduce conflict

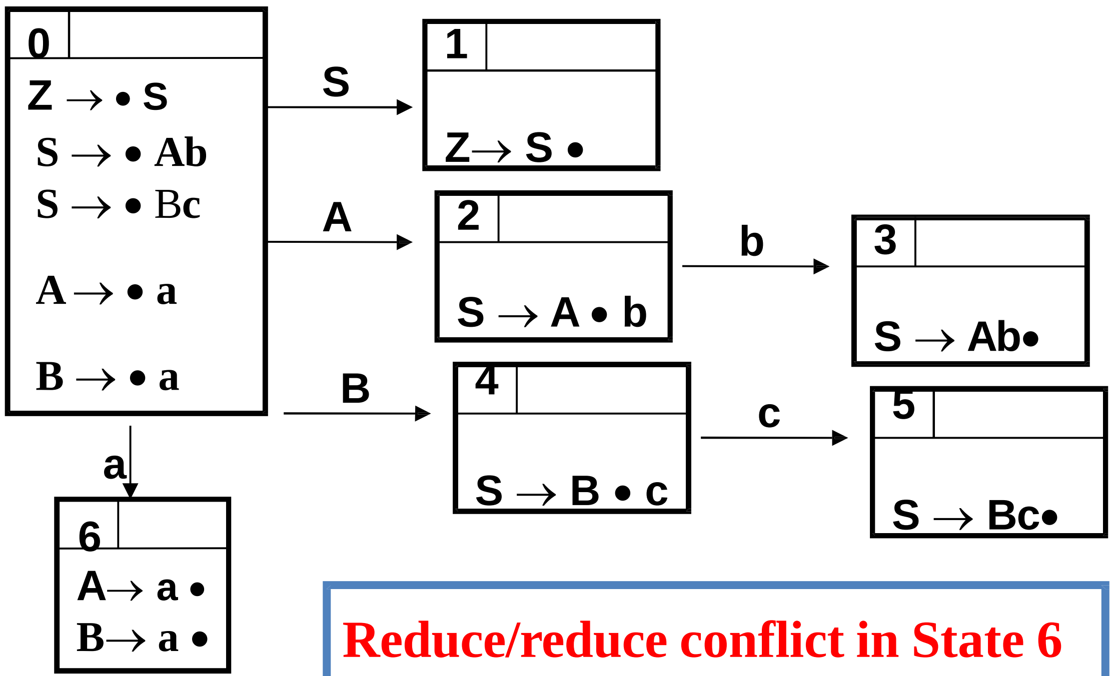
$V_T = \{a, b, c\}$

$V_N = \{S, A, B\}$

$S = S$

P:

{(1) $S \rightarrow Ab$
(2) $S \rightarrow Bc$
(3) $A \rightarrow a$
(4) $B \rightarrow a$
}



Reduce/reduce conflict in State 6
(1) Reduce item 1: $A \rightarrow a \bullet$
(2) Reduce item 2: $B \rightarrow a \bullet$

How to resolve?

- **Improve LR(0)**
  - **SLR** - simple LR parser
  - **LR** - most general LR parser
  - **LALR** - intermediate LR parser

SLR(1)

SLR(1), simple LR(1) parsing, uses the DFA of sets of LR(0) items as constructed in the previous section

SLR(1) increases the power of LR(0) parsing significant by using the next token in the input string

- First, it consults the input token *before* a shift to make sure that an appropriate DFA transition exists
- Second, it uses the **Follow set** of a non-terminal to decide if a reduction should be performed

SLR(1)

- **Choose the action by looking ahead of a symbol**
  - For LR(0) itemset $I=\{X \rightarrow \gamma \bullet a\beta,\ \ A \rightarrow \pi \bullet,\ B \rightarrow \pi' \bullet\}$, denoted as state Si:
    - Conflict in cell (Si, a): Reduce or shift?
    - What if $Follow(A) \cap Follow(B) = \Phi$, specifically,

      $a \notin Follow(A)$, $a \notin Follow(B)$, what can we do?

SLR(1)

- **Choose the action by looking ahead of a symbol, for cell (Si, a)**
  - S/R conflict:
    - Choose shift: if there exist $A\rightarrow \alpha\bullet a\beta$
    - Choose reduce: if there exist $B\rightarrow \pi\bullet$, and $a \in follow(B)$
  - R/R conflict
    - Choose reduce with P1: if there exist $A\rightarrow \pi\bullet$, $a\in follow(A)$, where $P1= A\rightarrow \pi$
    - Choose reduce with P2, if there exist $B\rightarrow \pi'\bullet$, $a\in follow(B)$, where $P2= B\rightarrow\pi'$

LR(0) table 1 with S/R conflict

$V_T = \{a, b\}$

$V_N = \{S, A\}$

S = S

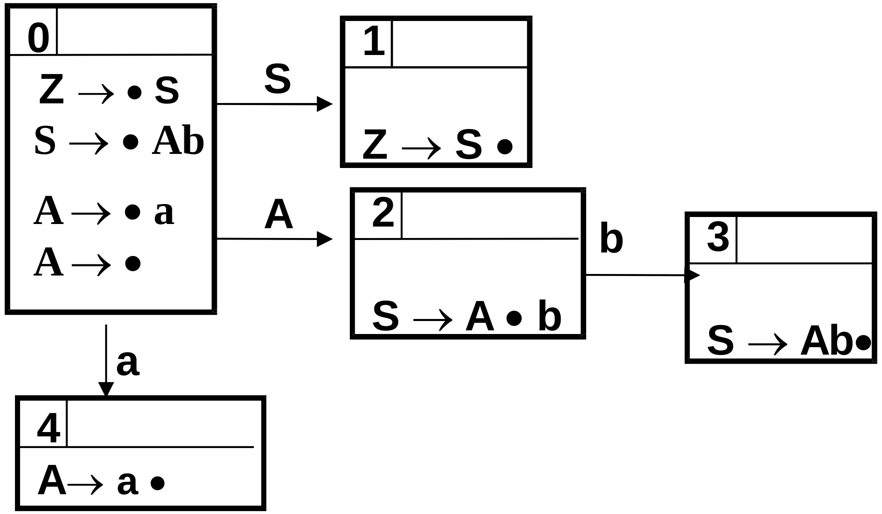
P:

{ (1) $S \rightarrow Ab$

(2) $A \rightarrow \varepsilon$

(3) $A \rightarrow a$

}



	Action				Goto	
	a	b	#		S	A
0	S4;R2	R2	R2		1	3
1			Accept			
2		S3				
3	R1	R1	R1			
4	R3	R3	R3			

In state 0:

(1) shift: $A \rightarrow \bullet a$

(2) reduce: $A \rightarrow \bullet$

LR(0) table 1 without S/R conflict

$V_T = \{a, b\}$

$V_N = \{S, A\}$

S = S

P:

{ (1) $S \rightarrow Ab$
(2) $A \rightarrow \varepsilon$
(3) $A \rightarrow a$
}

	Action				Goto	
	a	b	#		S	A
0	S4	R2			1	3
1			Accept			
2		S3				
3			R1			
4		R3				

Resolve conflict with follow(A)

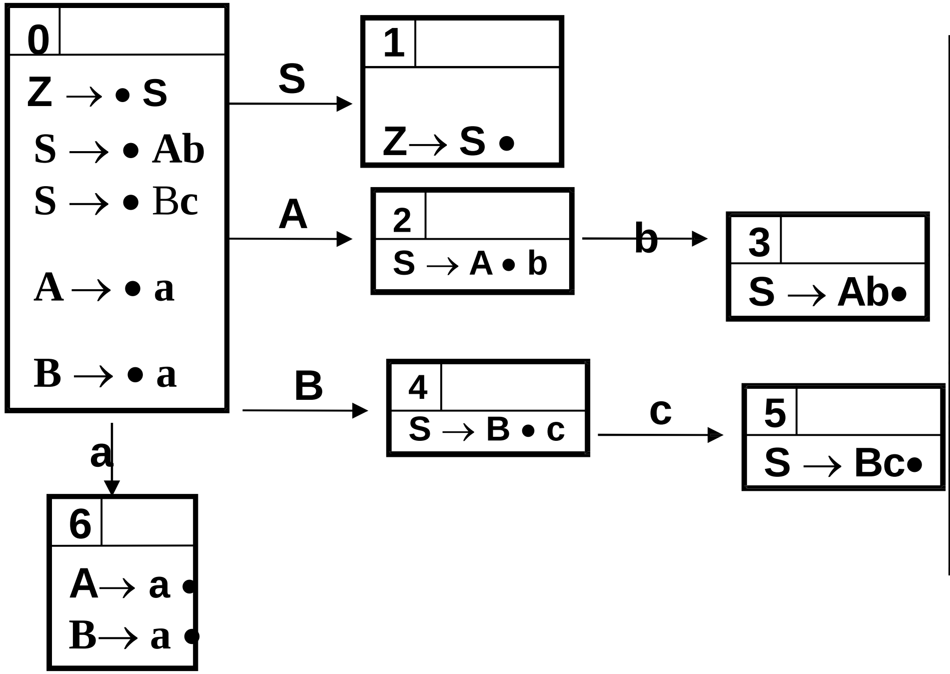
LR(0) table 2 with S/R conflict

$V_T = \{a, b, c\}$

$V_N = \{S, A, B\}$

S = S

P:
{(1) $S \rightarrow Ab$
(2) $S \rightarrow Bc$
(3) $A \rightarrow a$
(4) $B \rightarrow a$
}



	Action					Goto		
	a	b	c	#		S	A	B
0	S7					1	3	5
1				Accept				
2		S4						
3	R1	R1	R1	R1				
4			S6					
5	R2	R2	R2	R2				
6	R3 R4	R3 R4	R3 R4	R3 R4				

Reduce/reduce conflict in State 6
(1) Reduce item 1: $A \rightarrow a \bullet$
(2) Reduce item 2: $B \rightarrow a \bullet$

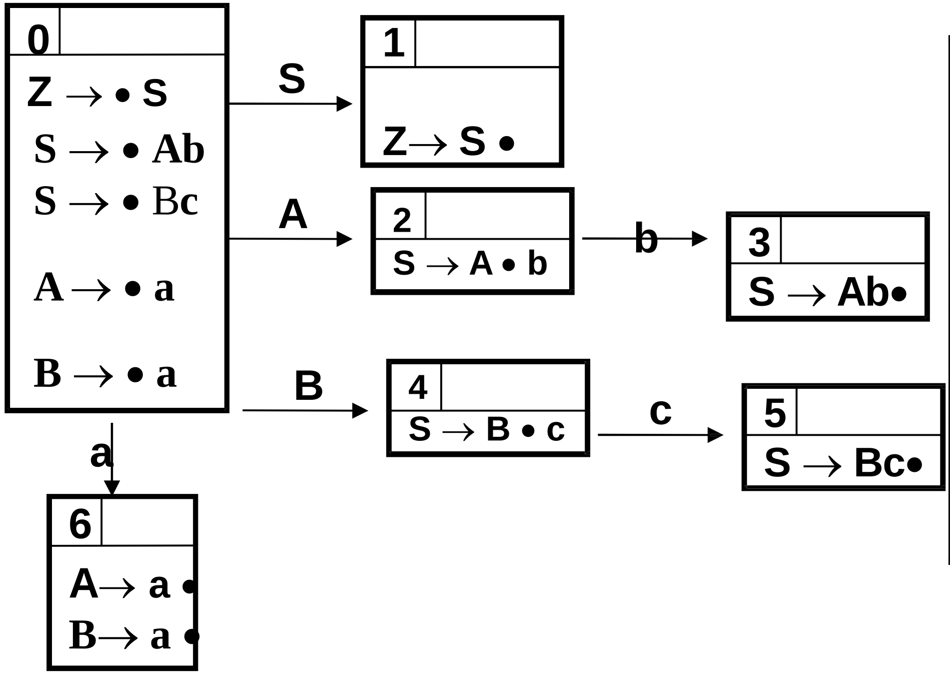
LR(0) table 2 without S/R conflict

$V_T = \{a, b, c\}$

$V_N = \{S, A, B\}$

S = S

P:
{(1) $S \rightarrow Ab$
(2) $S \rightarrow Bc$
(3) $A \rightarrow a$
(4) $B \rightarrow a$
}



	Action					Goto		
	a	b	c	#		S	A	B
0	S7					1	3	5
1				Accept				
2		S4						
3	R1	R1	R1	R1				
4			S6					
5	R2	R2	R2	R2				
6		R3	R4					

Reduce/reduce conflict in State 6
(1) Reduce item 1: $A \rightarrow a \bullet$
(2) Reduce item 2: $B \rightarrow a \bullet$

Resolve conflict with follow(A) and follow(B)

Limitation of SLR(1)

- In SLR method, the state i makes a reduction by $A \rightarrow \alpha$ when the current token is a:
  - if the $A \rightarrow \alpha.$ in the I_i and a is FOLLOW(A)
- In some situations, βA cannot be followed by the terminal a in a right-sentential form when $\beta\alpha$ and the state i are on the top stack.
- This means that making reduction in this case is not correct.

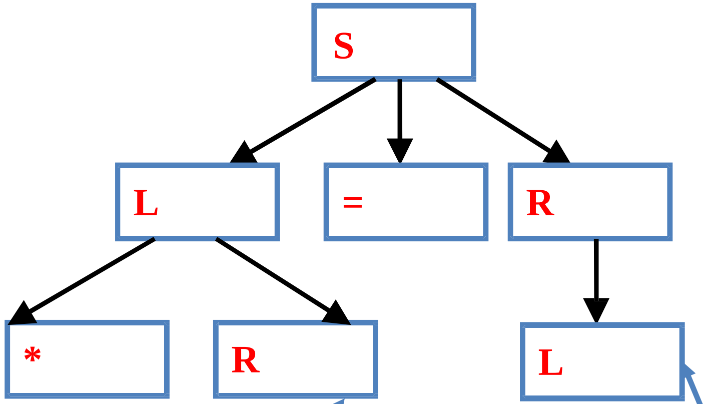
Limitation of SLR(1)

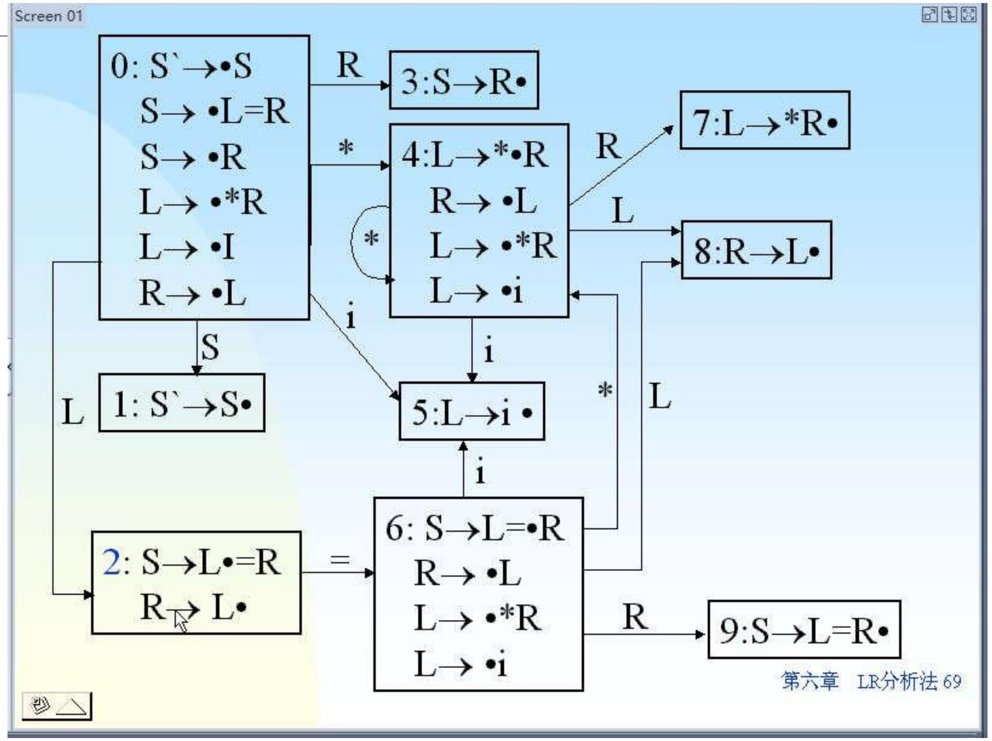
Example 4.51 : Let us reconsider Example 4.48, where in state 2 we had item $R \to L\cdot$, which could correspond to $A \to \alpha$ above, and a could be the = sign, which is in FOLLOW(R). Thus, the SLR parser calls for reduction by $R \to L$ in state 2 with = as the next input (the shift action is also called for, because of item $S \to L\cdot{=}R$ in state 2). However, there is no right-sentential form of the grammar in Example 4.48 that begins $R = \cdots$. Thus state 2, which is the state corresponding to viable prefix L only, should not really call for reduction of that L to R. □

Limitation of SLR(1)

- 1. S` →S   2. S →L=R
- 3. S →R   4. L →*R
- 5. L → i   6. R →L

follow(R) = {#, =}





Reducible???

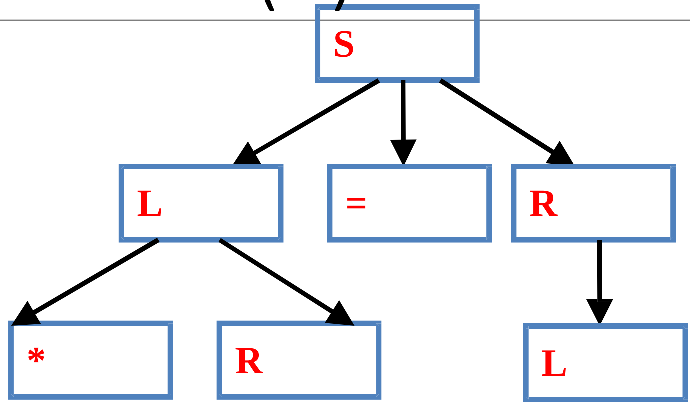
Follow symbol "=" is actually from here

How exactly does "R=" come from: S' => L=R =>*R=R

We must have a * before R.

Limitation of SLR(1)

- 1. S` →S 2. S →L=R
- 3. S →R 4. L →*R
- 5. L → i 6. R →L



Solution: LR(1), not consider ALL follow symbols, instead, we consider all feasible follow symbols

To avoid some of invalid reductions, the states need to carry more information. Extra information is put into a state by including a terminal symbol as a second component in an item.

LR(1) Item

- A LR(1) item is: $A \rightarrow \alpha.\beta$**, a**,

  where **a** is the look-ahead of the LR(1) item (**a** is a terminal or end-marker.)

Constructing LR(1) automaton

```
SetOfItems CLOSURE(I) {
    repeat
        for ( each item [A → α·Bβ, a] in I )
            for ( each production B → γ in G' )
                for ( each terminal b in FIRST(βa) )
                    add [B → ·γ, b] to set I;
    until no more items are added to I;
    return I;
}

SetOfItems GOTO(I, X) {
    initialize J to be the empty set;
    for ( each item [A → α·Xβ, a] in I )
        add item [A → αX·β, a] to set J;
    return CLOSURE(J);
}

void items(G') {
    initialize C to CLOSURE({[S' → ·S, $]});
    repeat
        for ( each set of items I in C )
            for ( each grammar symbol X )
                if ( GOTO(I, X) is not empty and not in C )
                    add GOTO(I, X) to C;
    until no new sets of items are added to C;
}
```

Key about look-ahead symbols

S0 = CLOSURE$(\{(S' \rightarrow \bullet S, \{\#\})\})$

- **Type 1**

$$S \rightarrow \alpha \bullet X\beta,\ ss \xrightarrow{X} S \rightarrow \alpha X \bullet \beta,\ ss$$

- **Type 2**

$S \rightarrow \alpha \bullet A\beta$, ss

A $\rightarrow \bullet\alpha 1$, ss'

…….

A $\rightarrow \bullet\ \alpha n$, ss'

…….

ss' = first(β), if β does not derive empty;

ss' = (first(β)-{ε}) $\cup$ss, if β derives empty;

An Example

1. $S' \to S$
2. $S \to C\ C$
3. $C \to c\ C$
4. $C \to d$

I_0: closure($\{(S' \to \bullet\ S, \$)\}$) =

$(S' \to \bullet\ S, \$)$
$(S \to \bullet\ C\ C, \$)$
$(C \to \bullet\ c\ C, c/d)$
$(C \to \bullet\ d, c/d)$

I_1: goto(I_0, S) = $(S' \to S\ \bullet, \$)$

I_2: goto(I_0, C) =

$(S \to C\ \bullet\ C, \$)$
$(C \to \bullet\ c\ C, \$)$
$(C \to \bullet\ d, \$)$

I_3: goto(I_0, c) =

$(C \to c\ \bullet\ C, c/d)$
$(C \to \bullet\ c\ C, c/d)$
$(C \to \bullet\ d, c/d)$

I_4: goto(I_0, d) =

$(C \to d\ \bullet, c/d)$

I_5: goto(I_3, C) =

$(S \to C\ C\ \bullet, \$)$

An Example

I_6: goto(I_3, c) =
(C $\rightarrow$ c $\bullet$ C, \$)
(C $\rightarrow$ $\bullet$ c C, \$)
(C $\rightarrow$ $\bullet$ d, \$)

I_7: goto(I_3, d) =
(C $\rightarrow$ d $\bullet$, \$)

I_8: goto(I_4, C) =
(C $\rightarrow$ c C $\bullet$, c/d)

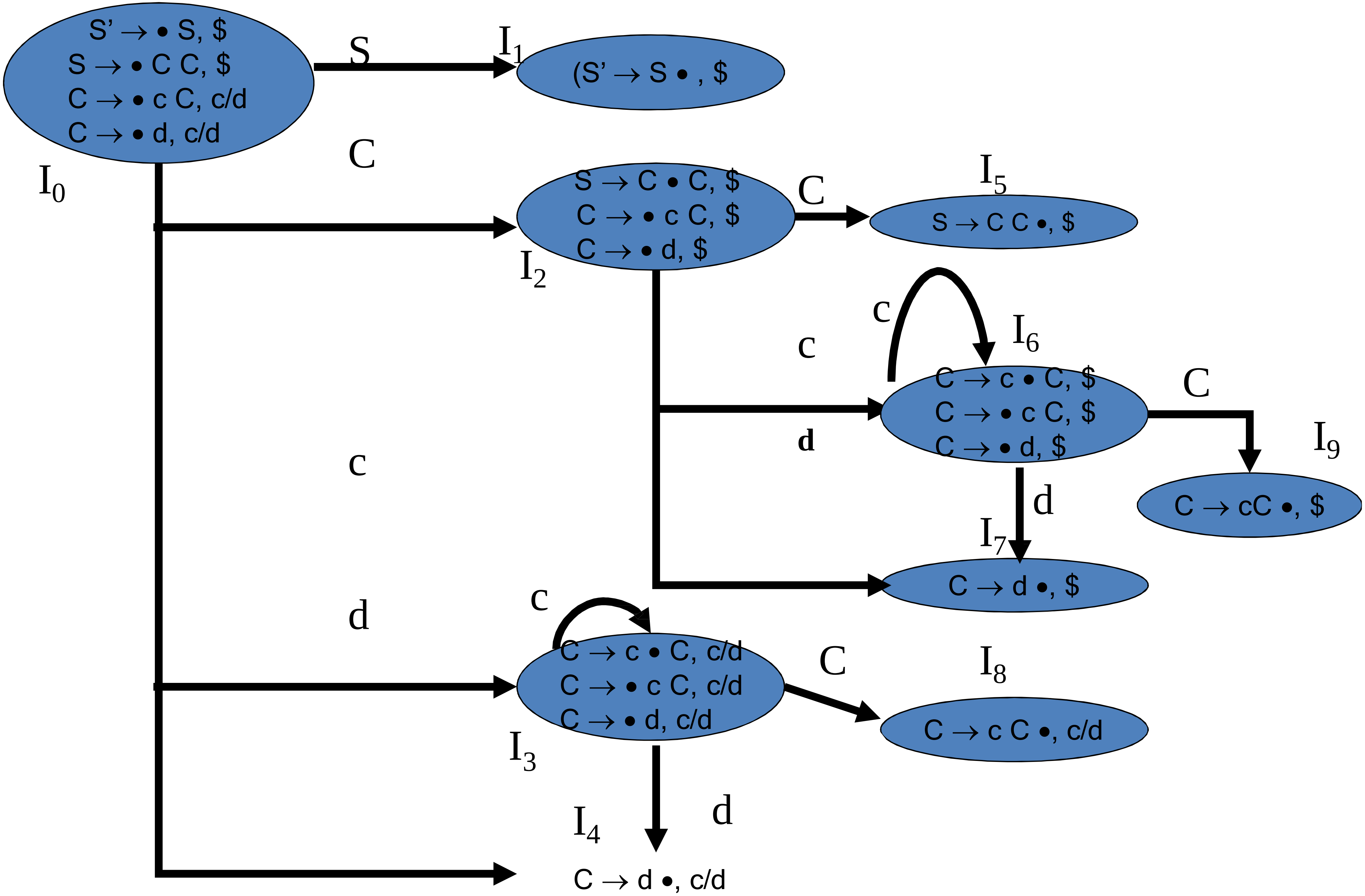
: goto(I_4, c) = I_4

: goto(I_4, d) = I_5

I_9: goto(I_7, c) =
(C $\rightarrow$ c C $\bullet$, \$)

: goto(I_7, c) = I_7

: goto(I_7, d) = I_8



S’ → • S, $
S → • C C, $
C → • c C, c/d
C → • d, c/d
I0
S
I1
(S’ → S • , $
C
S → C • C, $
C → • c C, $
C → • d, $
I2
C
I5
S → C C •, $
c
I6
C → c • C, $
C → • c C, $
C → • d, $
c
d
C
I9
C → cC •, $
d
I7
C → d •, $
c
d
c
C → c • C, c/d
C → • c C, c/d
C → • d, c/d
I3
C
I8
C → c C •, c/d
I4
d
C → d •, c/d

Example 2

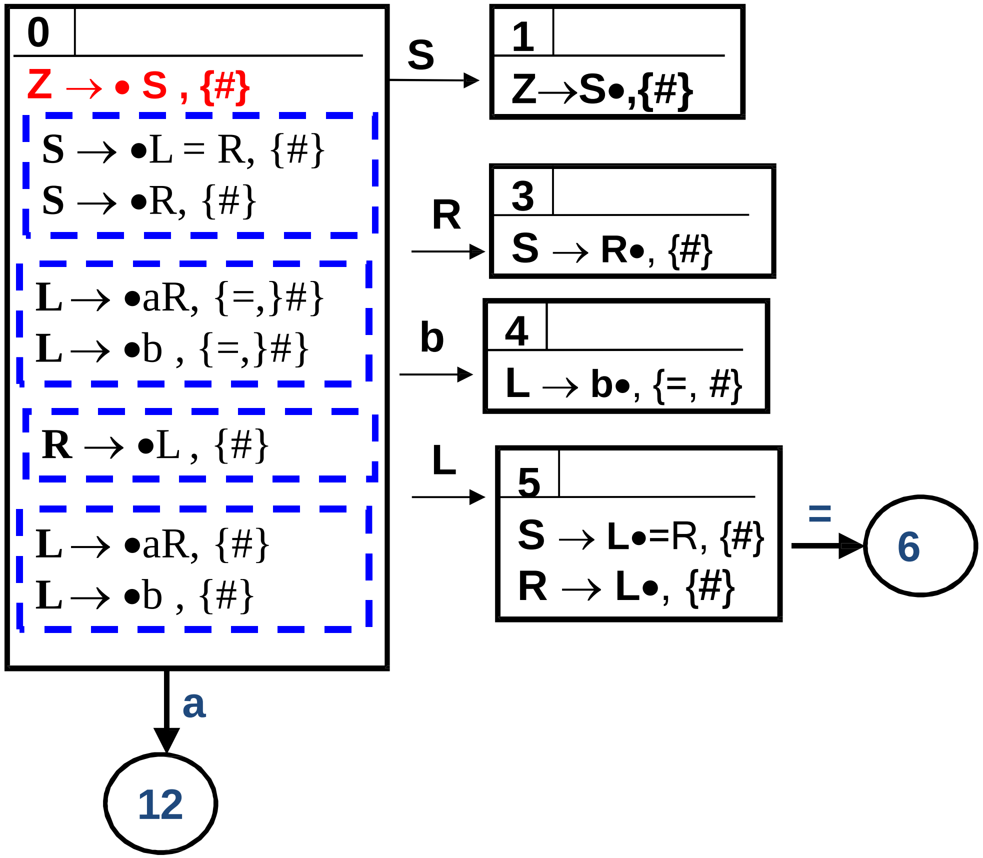
V_T = {a, b, =}

V_N = {S, L, R}

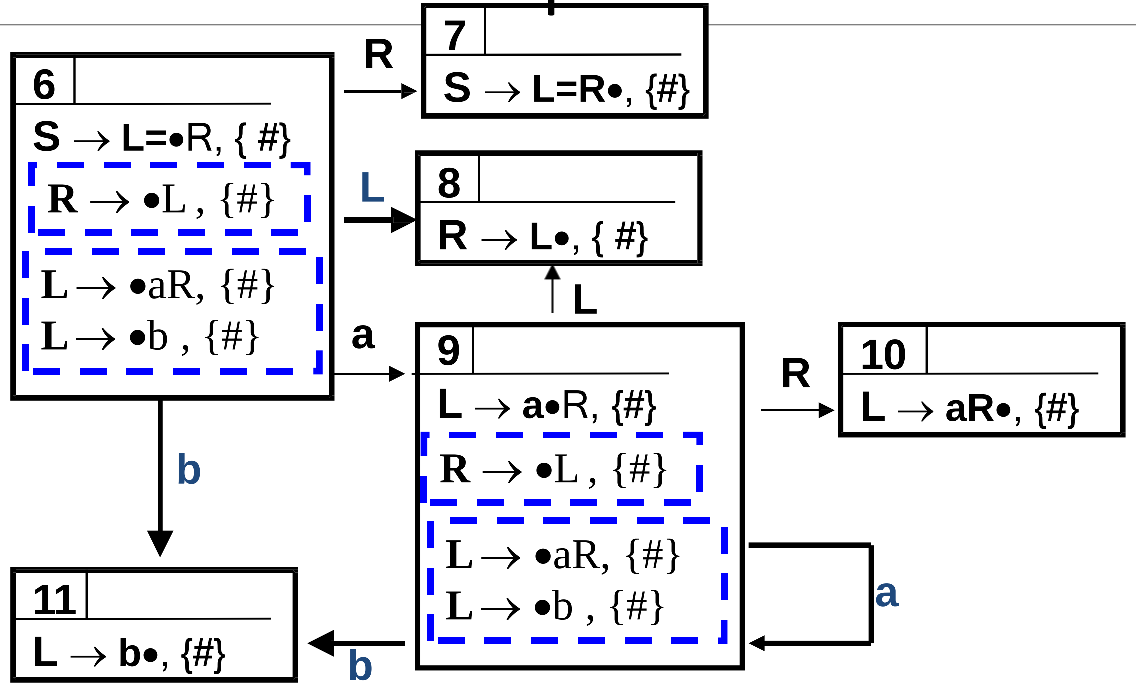
S = S

P:

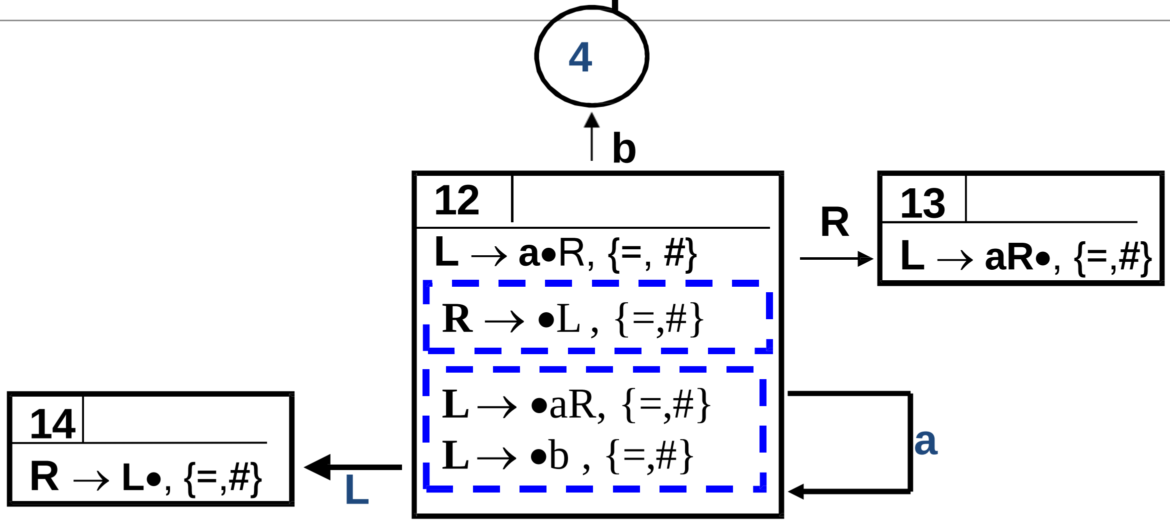
{(1) S → L = R
(2) S → R
(3) L → aR
(4) L → b
(5) R → L
}



Example 2



Example 2



Algorithm 4.56: Construction of canonical-LR parsing tables.

INPUT: An augmented grammar G'.

OUTPUT: The canonical-LR parsing table functions ACTION and GOTO for G'.

METHOD:

1. Construct $C' = \{I_0, I_1, \cdots, I_n\}$, the collection of sets of LR(1) items for G'.

2. State i of the parser is constructed from I_i. The parsing action for state i is determined as follows.

   (a) If $[A \to \alpha{\cdot}a\beta, b]$ is in I_i and GOTO$(I_i, a) = I_j$, then set ACTION$[i, a]$ to "shift j." Here a must be a terminal.

   (b) If $[A \to \alpha\cdot, a]$ is in I_i, $A \neq S'$, then set ACTION$[i, a]$ to "reduce $A \to \alpha$."

   (c) If $[S' \to S\cdot, \$]$ is in I_i, then set ACTION$[i, \$]$ to "accept."

   If any conflicting actions result from the above rules, we say the grammar is not LR(1). The algorithm fails to produce a parser in this case.

3. The goto transitions for state i are constructed for all nonterminals A using the rule: If GOTO$(I_i, A) = I_j$, then GOTO$[i, A] = j$.

4. All entries not defined by rules (2) and (3) are made "error."

5. The initial state of the parser is the one constructed from the set of items containing $[S' \to \cdot S, \$]$.

Building the Action Table

Action Table

action(S_i,a) = S_j, if there is an edge from S_i to S_j labeled as a

action(S_i,a) = R_p, only if S_i contains LR(1) item ($A \rightarrow \alpha \bullet$, ss)

Where $A \rightarrow \alpha$ is production P, 且 $a \in ss$;

action(S_i,#) = accept, if S_i is acceptance state

action(S_i,a) = error, otherwise

Terminal symbols / States	a_1	…	#
S_1			
…			
S_n			

Building the Goto Table-same as LR(0)

GOTO Table

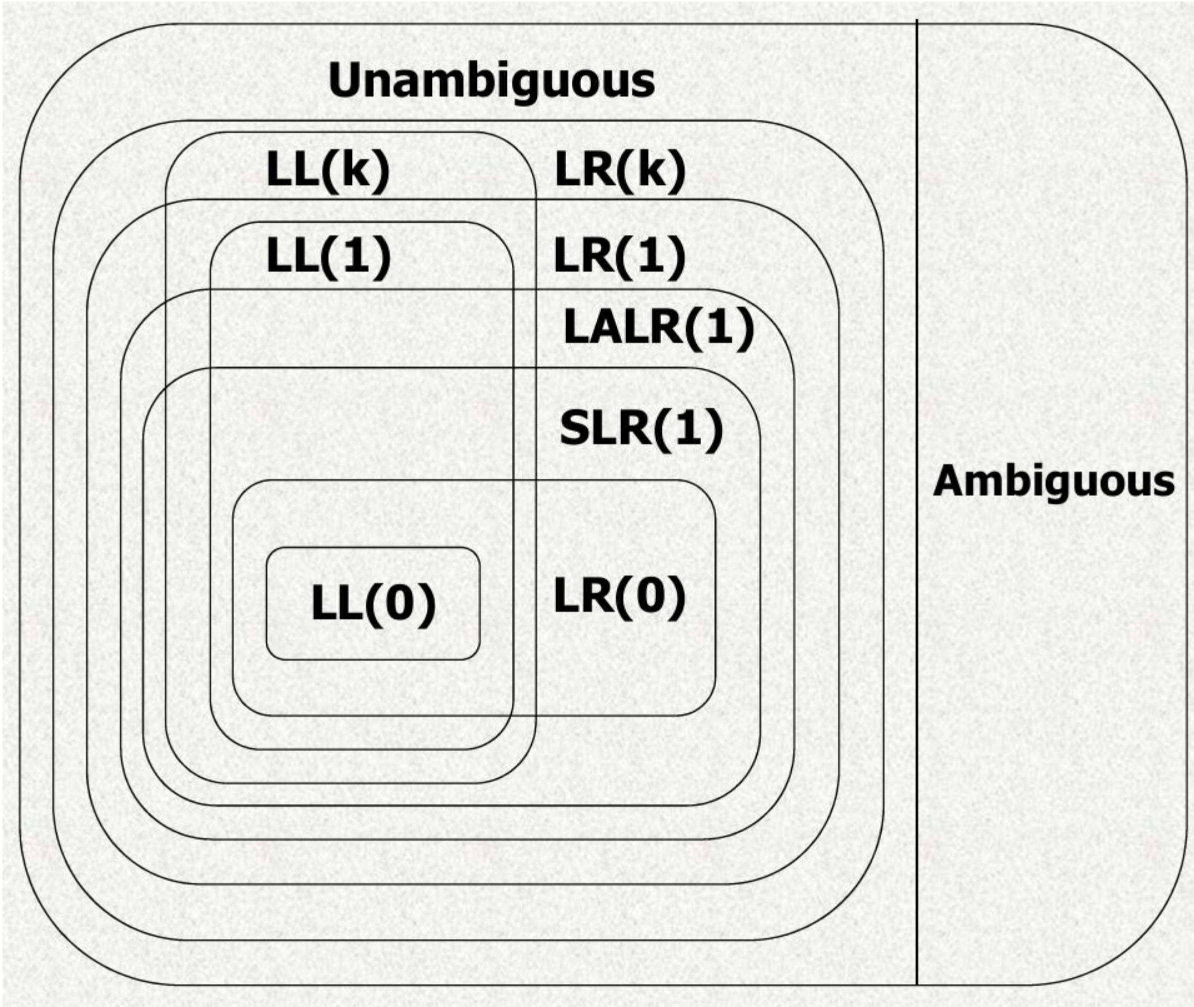
goto (S_i, A) = S_j, if there is an edge from S_i to S_j labeled as A
goto (S_i, A) = error, if there is no edge from S_i to S_j labeled as A

non-terminal / **State**	A_1	…	#
S_1			
…			
S_n			

LR Family

- **LR Family**
  - covers wide range of grammars.
  - SLR – simple LR parser
  - LR – most general LR parser
  - LALR – intermediate LR parser (look-head LR parser)

  - SLR, LR and LALR work same (they used the same algorithm), only their parsing tables are different.

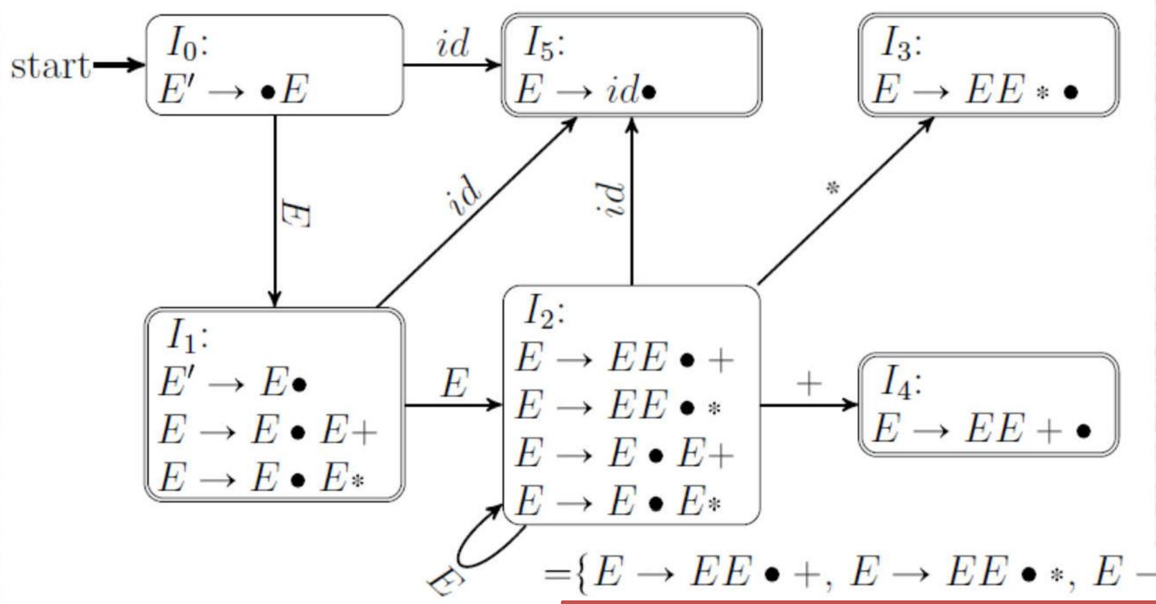

Unambiguous
LL(k)
LR(k)
LL(1)
LR(1)
LALR(1)
SLR(1)
Ambiguous
LL(0)
LR(0)

Sample exercises

$$
\begin{aligned}
E' &\to E \\
E &\to EE+ \\
&\mid\ EE* \\
&\mid\ id
\end{aligned}
$$

First(E) = {id}; Follow(E) = {+, *, id, $}

	action				goto
状态	$*$	$+$	id	$\$$	E
0			s5		1
1			s5	acc	2
2	s3	s4	s5		2
3	r2	r2	r2	r2	
4	r1	r1	r1	r1	
5	r3	r3	r3	r3	



$$=\{E \to EE\bullet +,\ E \to EE\bullet *,\ E \to E\bullet E+,\ E \to E\bullet E*,$$

$$\boxed{E \to \bullet EE+,\ E \to \bullet EE+,\ E \to \bullet id\}}$$

Sample exercises

G(S) is defined as follow:
S → SS | (S) | ε
Where (and) are terminal symbols, S is the starting symbol;

S'	→	S	(0)
S	→	$S\,S$	(1)
	|	(S)	(2)
	|	ε	(3)

	action			goto
状态	(	)	$	S
0	s1/r3	r3	r3	2
1	s1/r3	r3	r3	3
2	s1/r3	r3	acc/r3	4
3	s1/r3	s5/r3	r3	4
4	r1/s1/r3	r1/r3	r1/r3	4
5	r2	r2	r2	

()()

Prefix	Remained string	State Stack	Action
	()()$	0	S1
$(	)()$	01	R3, goto(1, S)=3
$(S	)()$	013	S5
$(S)	()$	0135	R2, goto(0, S)=2
$S	()$	02	S1
$S(	)$	021	R3, goto(1, S)=3
$S(S	)$	0213	S5
$S(S)	$	02135	R2, goto(2, S)=4
$SS	$	022	R1, goto(0, S)=2
$S	$	02	accept