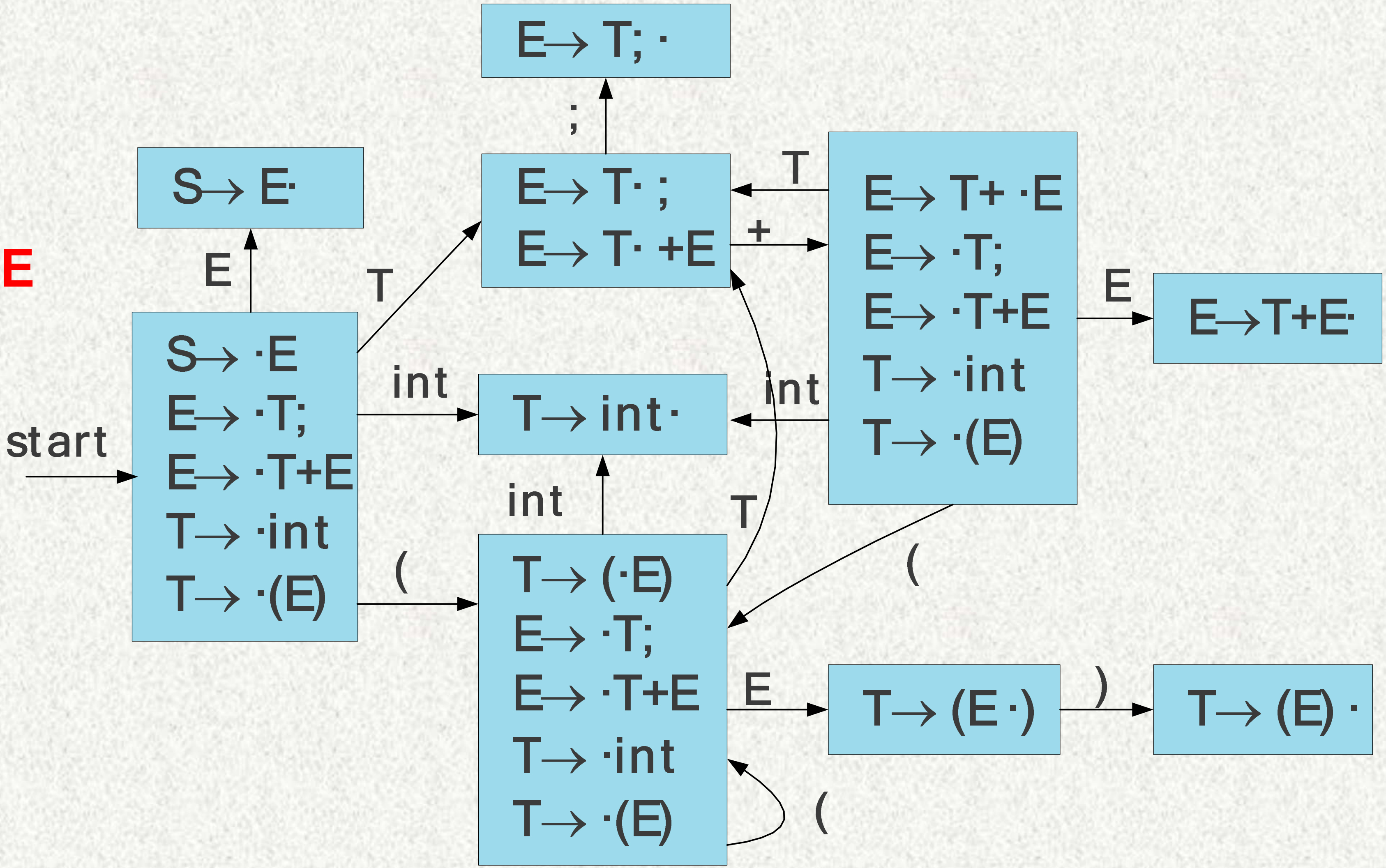
Lecture 10: Bottom-up Analysis

Xiaoyuan Xie 谢晓园
xxie@whu.edu.cn
School of Computer Science E301

LR(0) parser

LR(0) Parsing

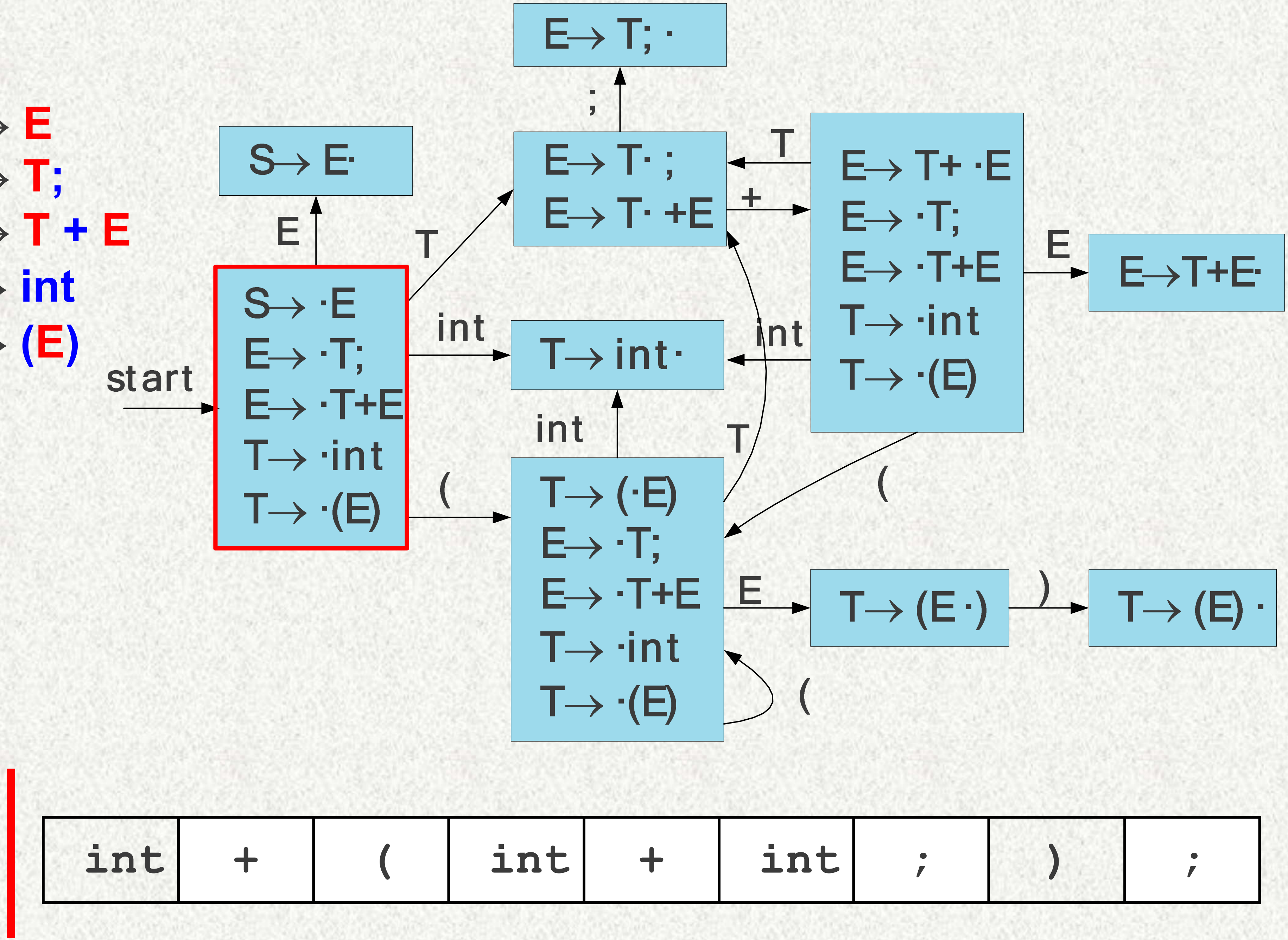
S → E
E → T;
E → T + E
T → int
T → (E)



int	+	(	int	+	int	;	)	;

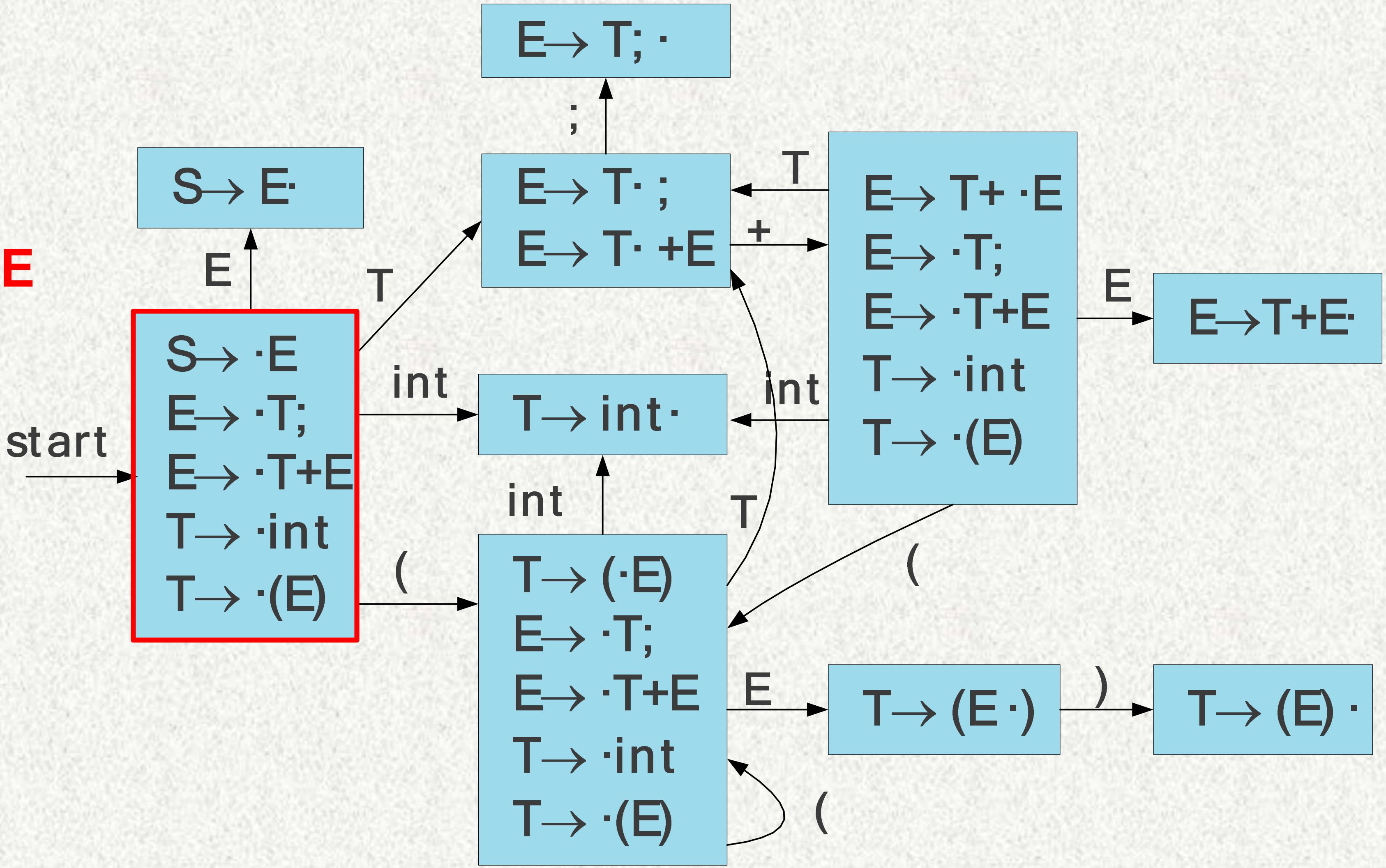
LR(0) Parsing

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



LR(0) Parsing

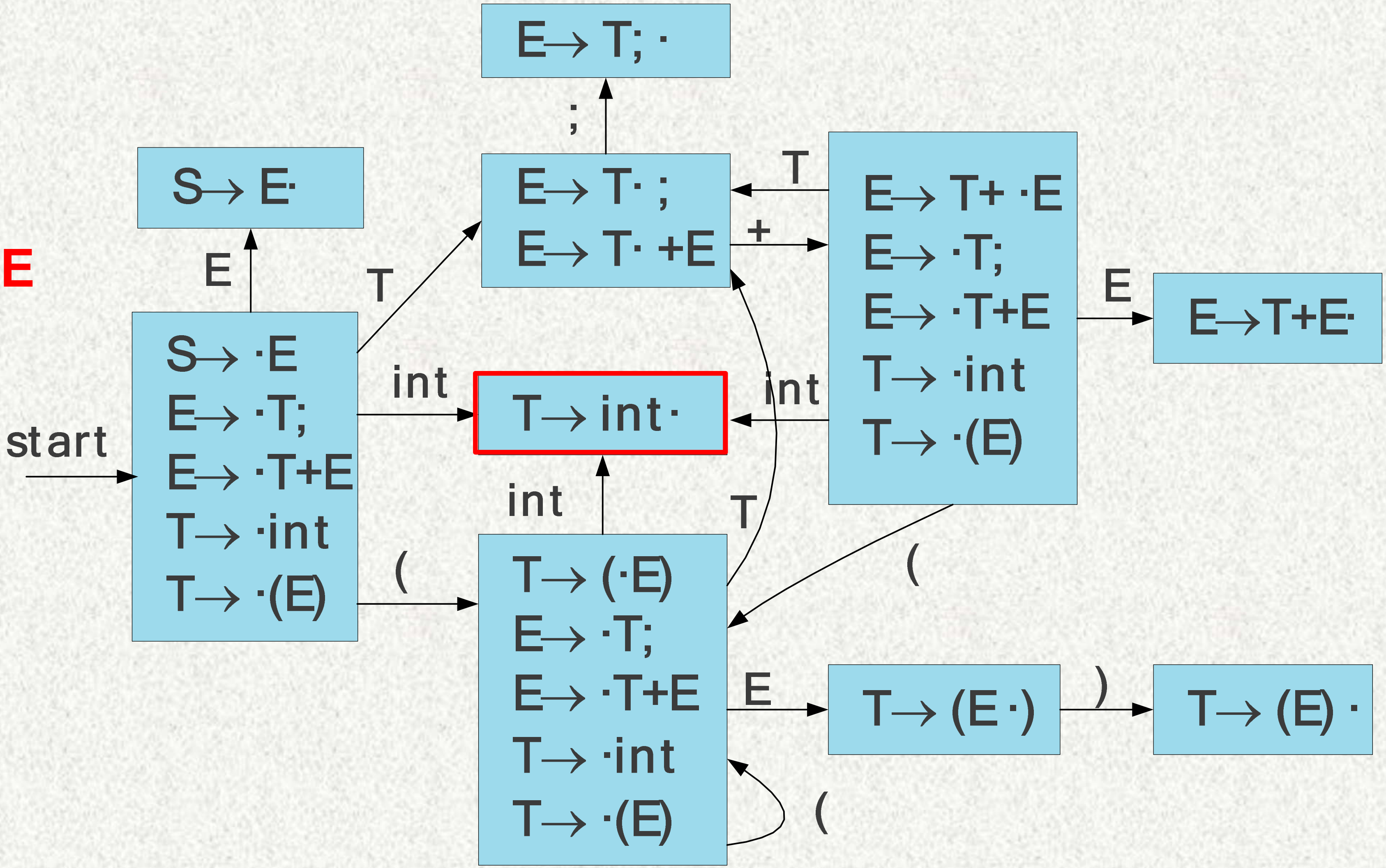
$S \rightarrow E$
$E \rightarrow T;$
$E \rightarrow T + E$
$T \rightarrow int$
$T \rightarrow (E)$



int		+	(	int	+	int	;	)	;

LR(0) Parsing

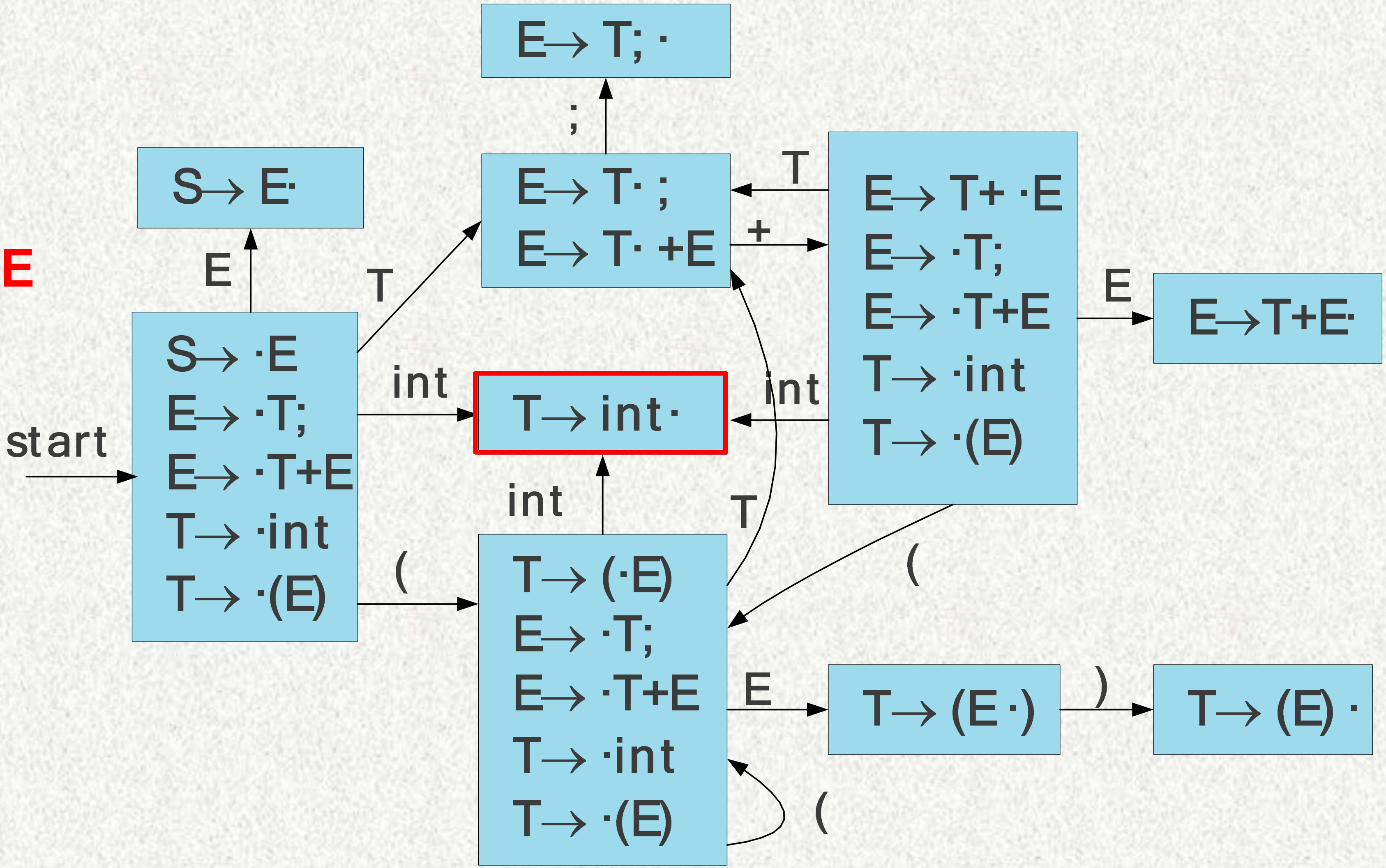
S → E
E → T;
E → T + E
T → int
T → (E)



int	+	(	int	+	int	;	)	;

LR(0) Parsing

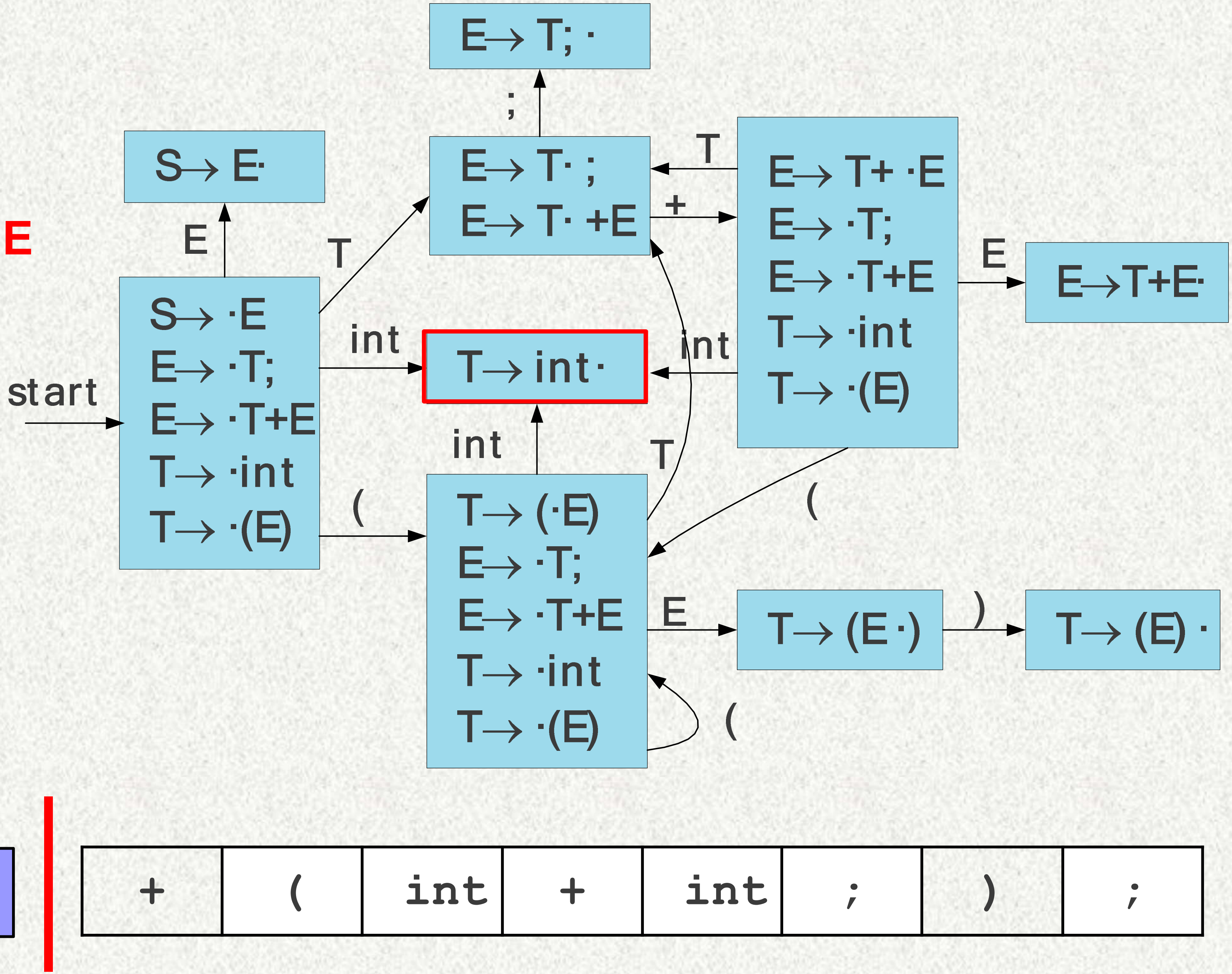
S → E
E → T;
E → T + E
T → int
T → (E)



+	(	int	+	int	;	)	;

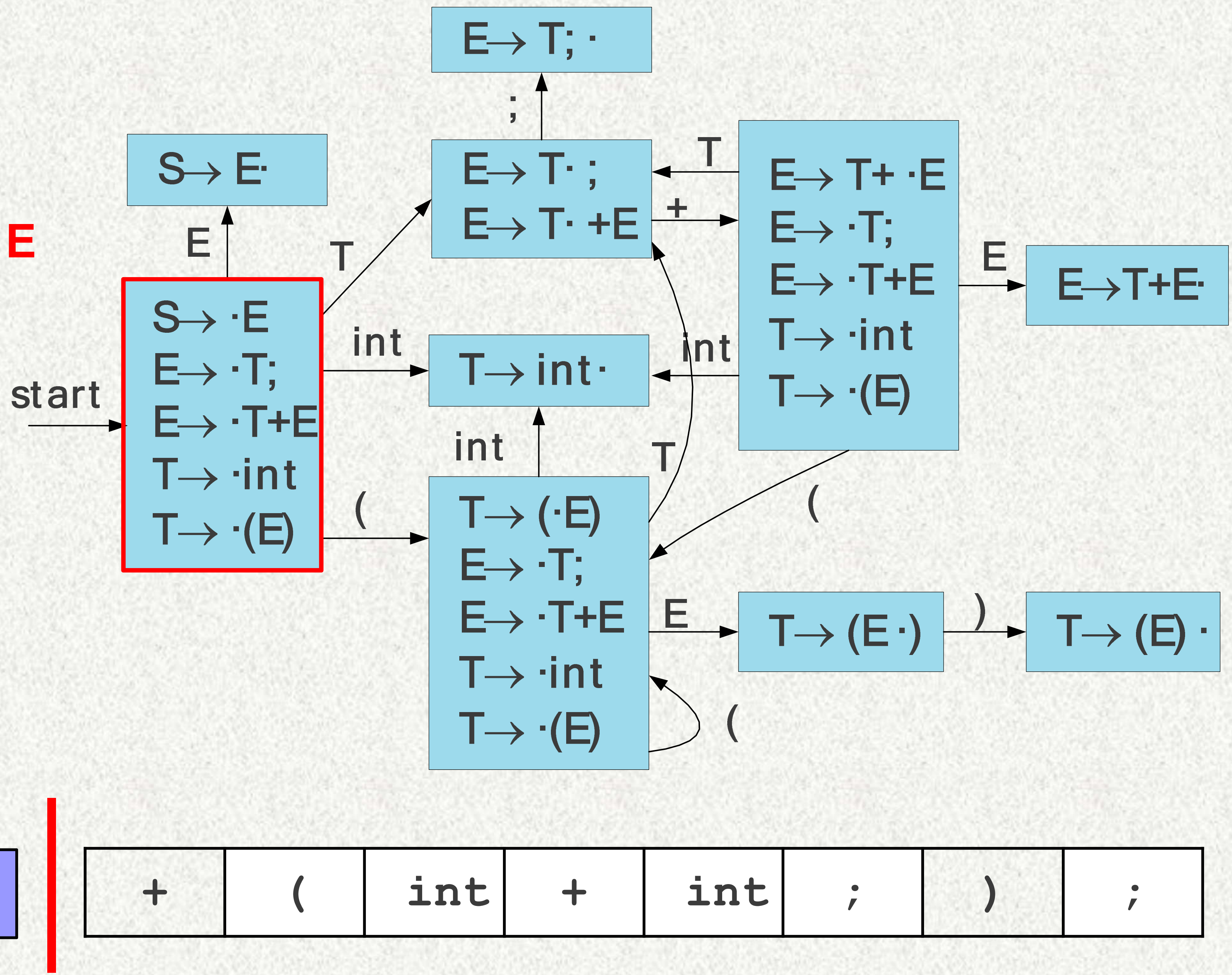
LR(0) Parsing

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



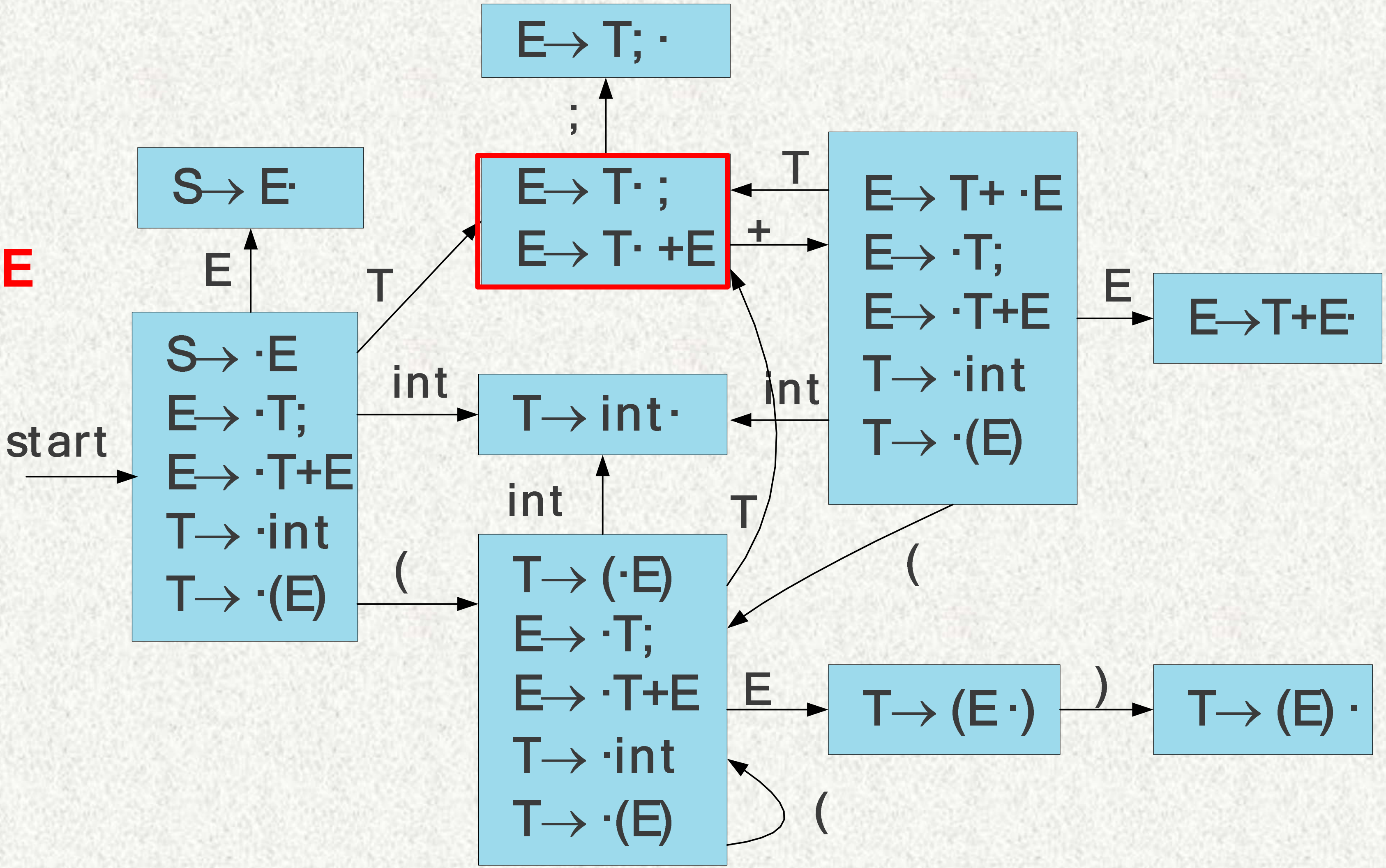
LR(0) Parsing

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



LR(0) Parsing

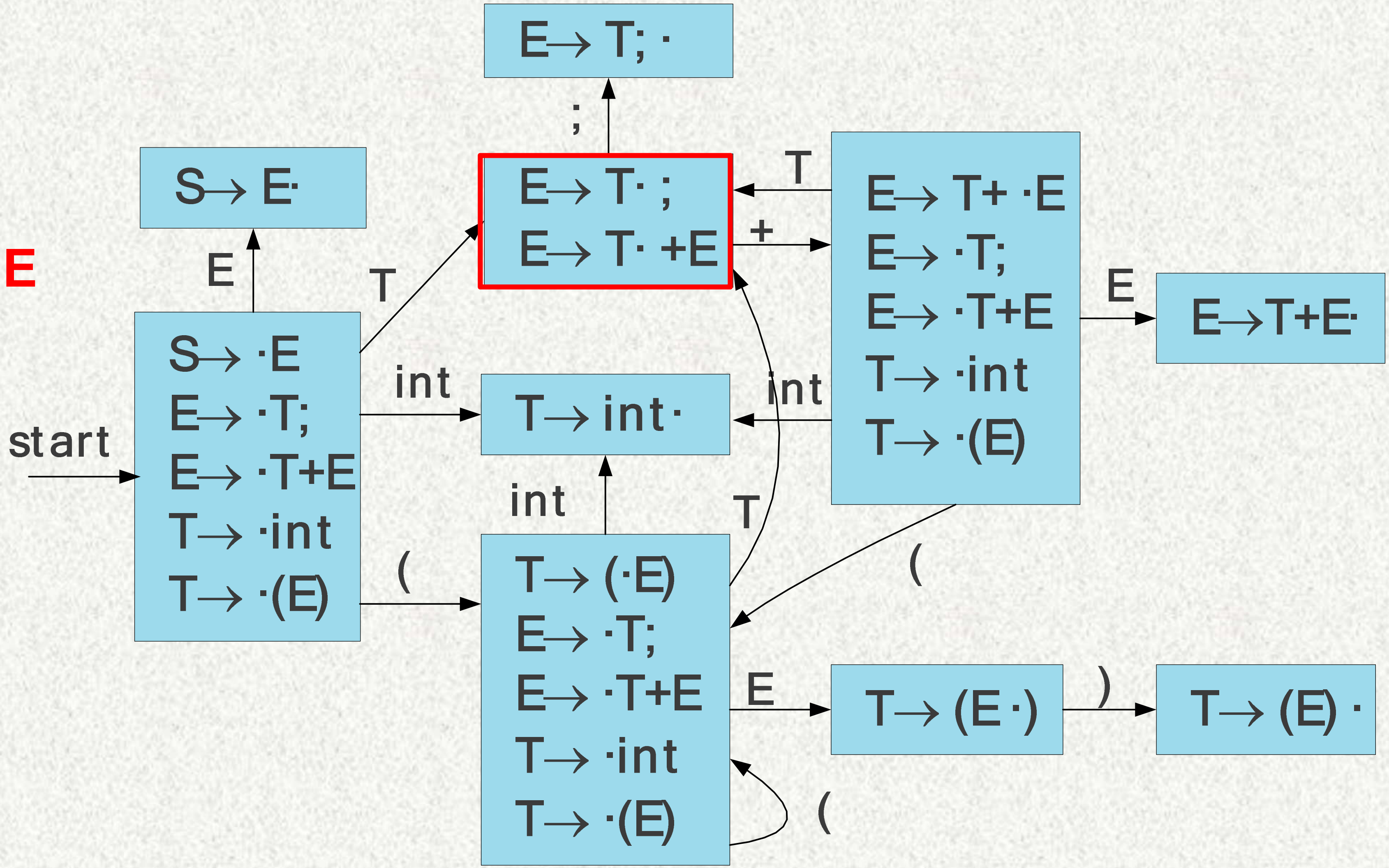
S → E
E → T;
E → T + E
T → int
T → (E)



T	+	(	int	+	int	;	)	;

LR(0) Parsing

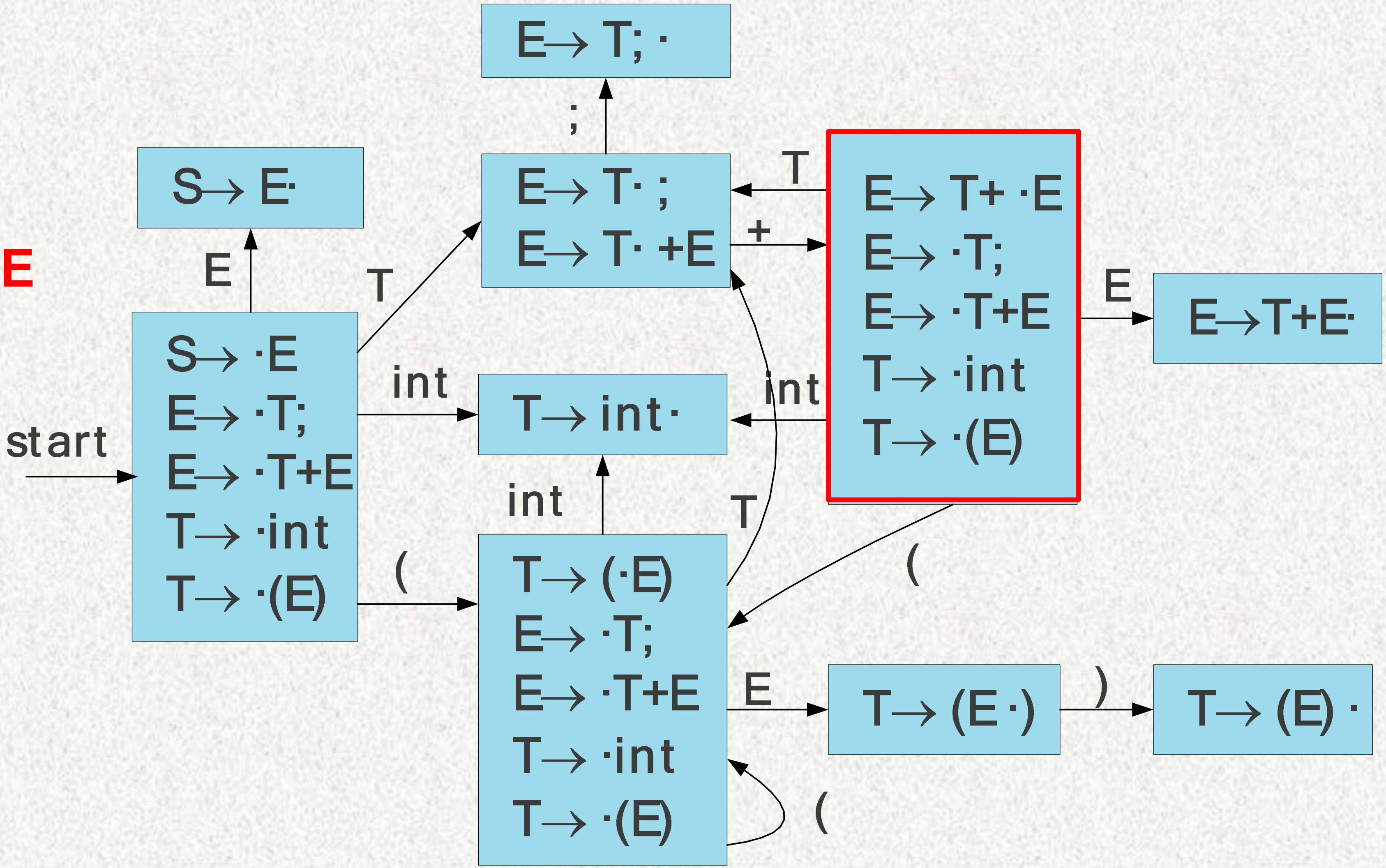
S → E
E → T;
E → T + E
T → int
T → (E)



T	+		(	int	+	int	;	)	;

LR(0) Parsing

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

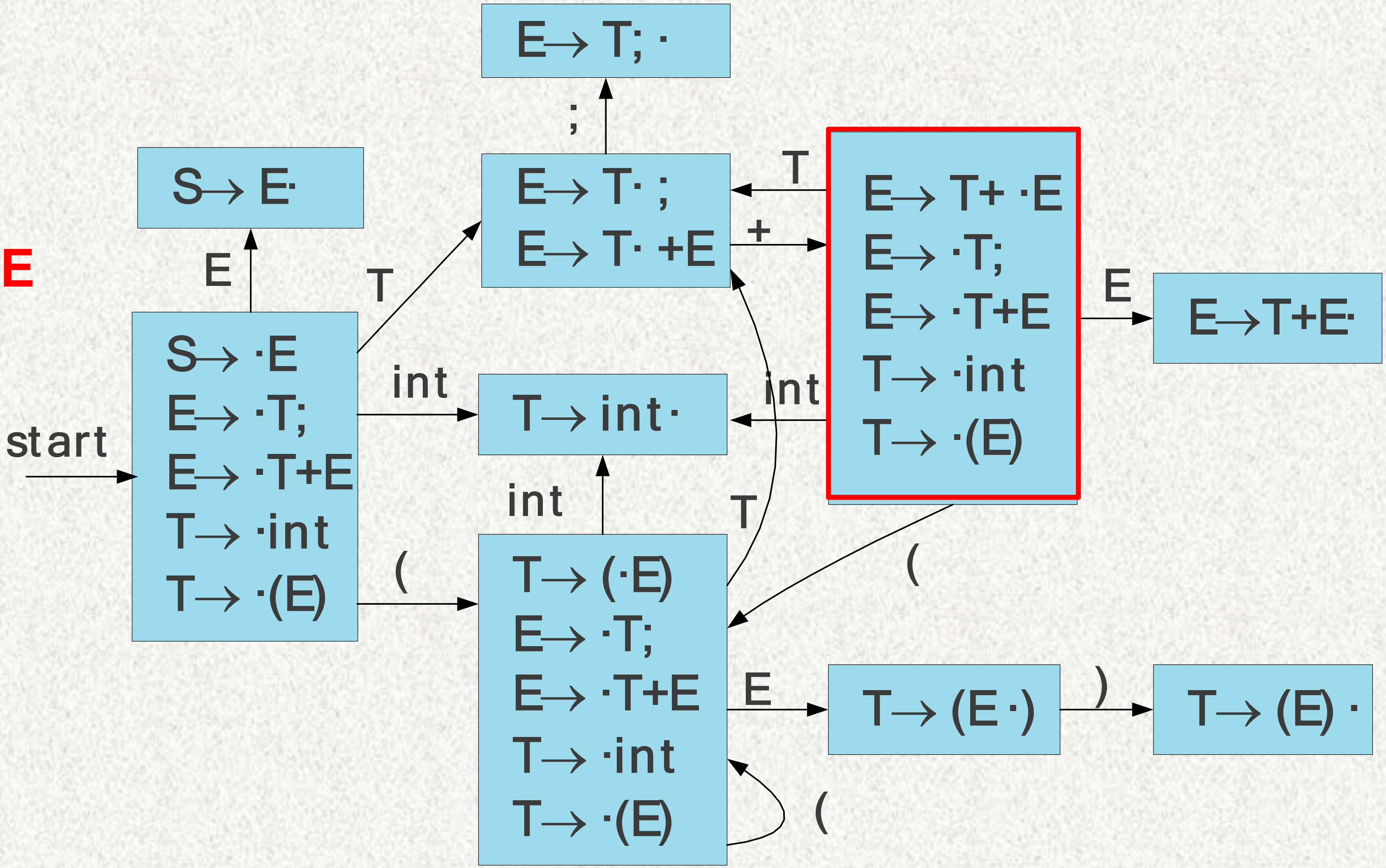


T	+

(	int	+	int	;	)	;

LR(0) Parsing

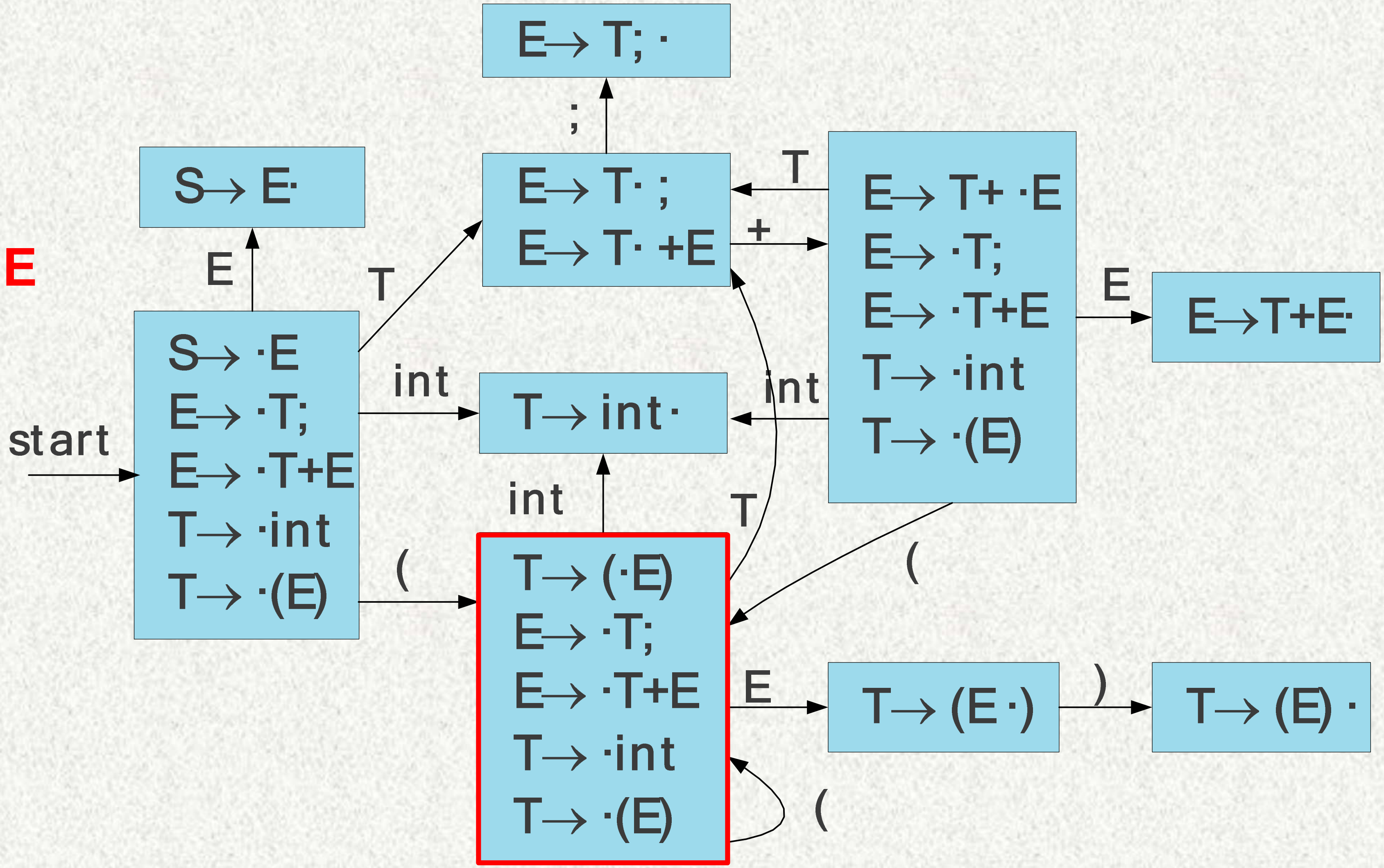
S → E
E → T;
E → T + E
T → int
T → (E)



T	+	(	int	+	int	;	)	;

LR(0) Parsing

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

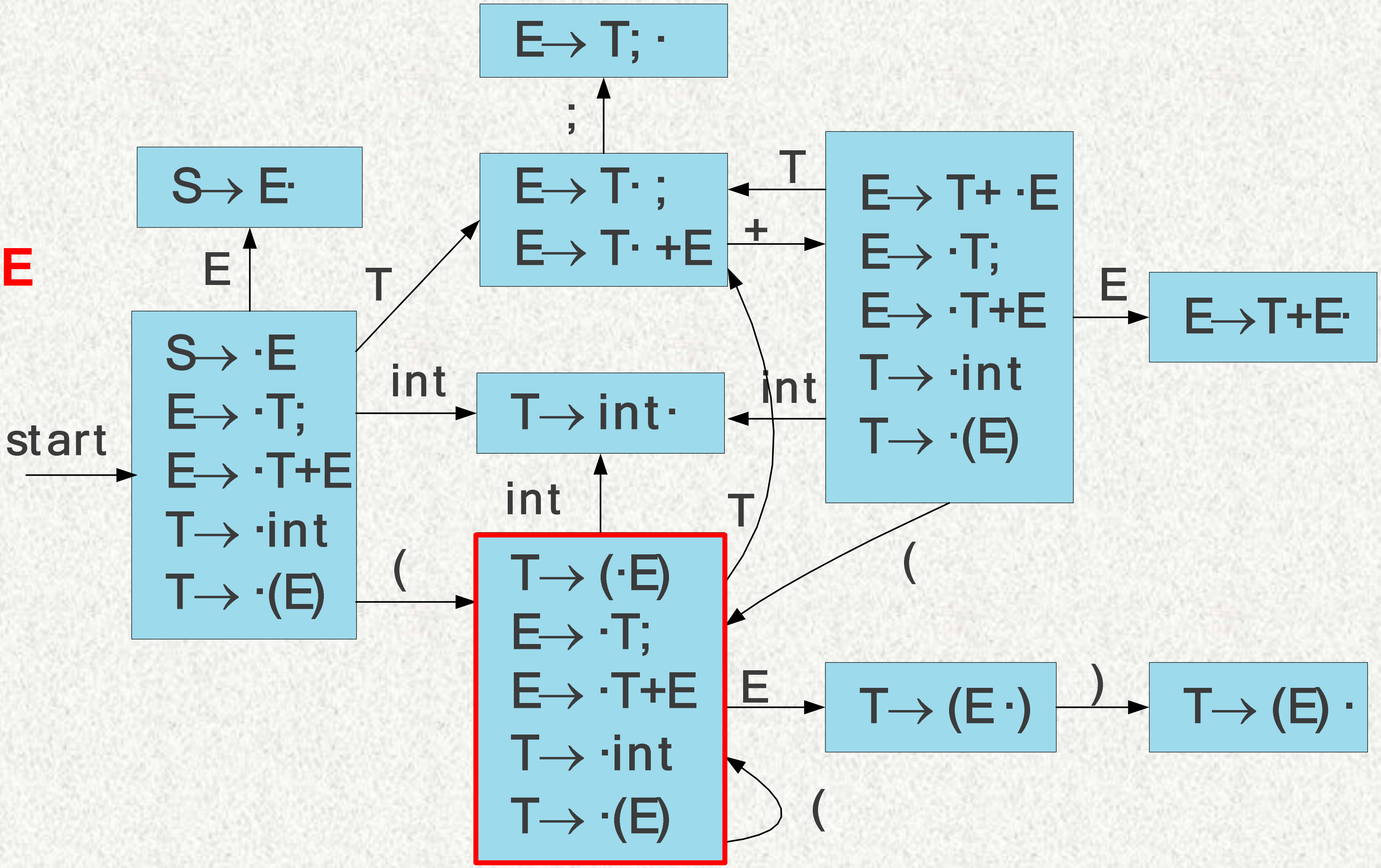


T	+	(

int	+	int	;	)	;

LR(0) Parsing

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

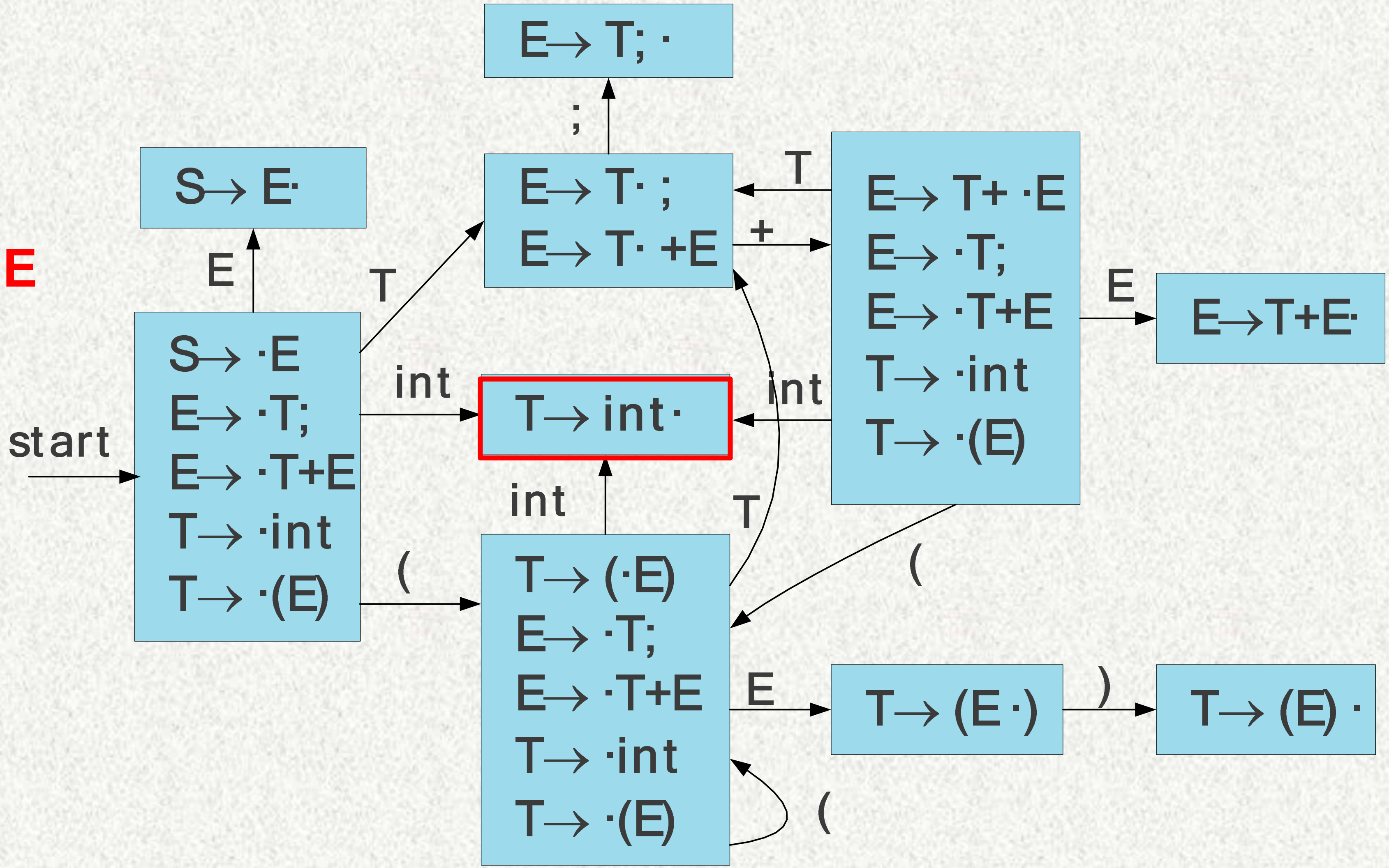


T	+	(	int

+	int	;	)	;

LR(0) Parsing

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

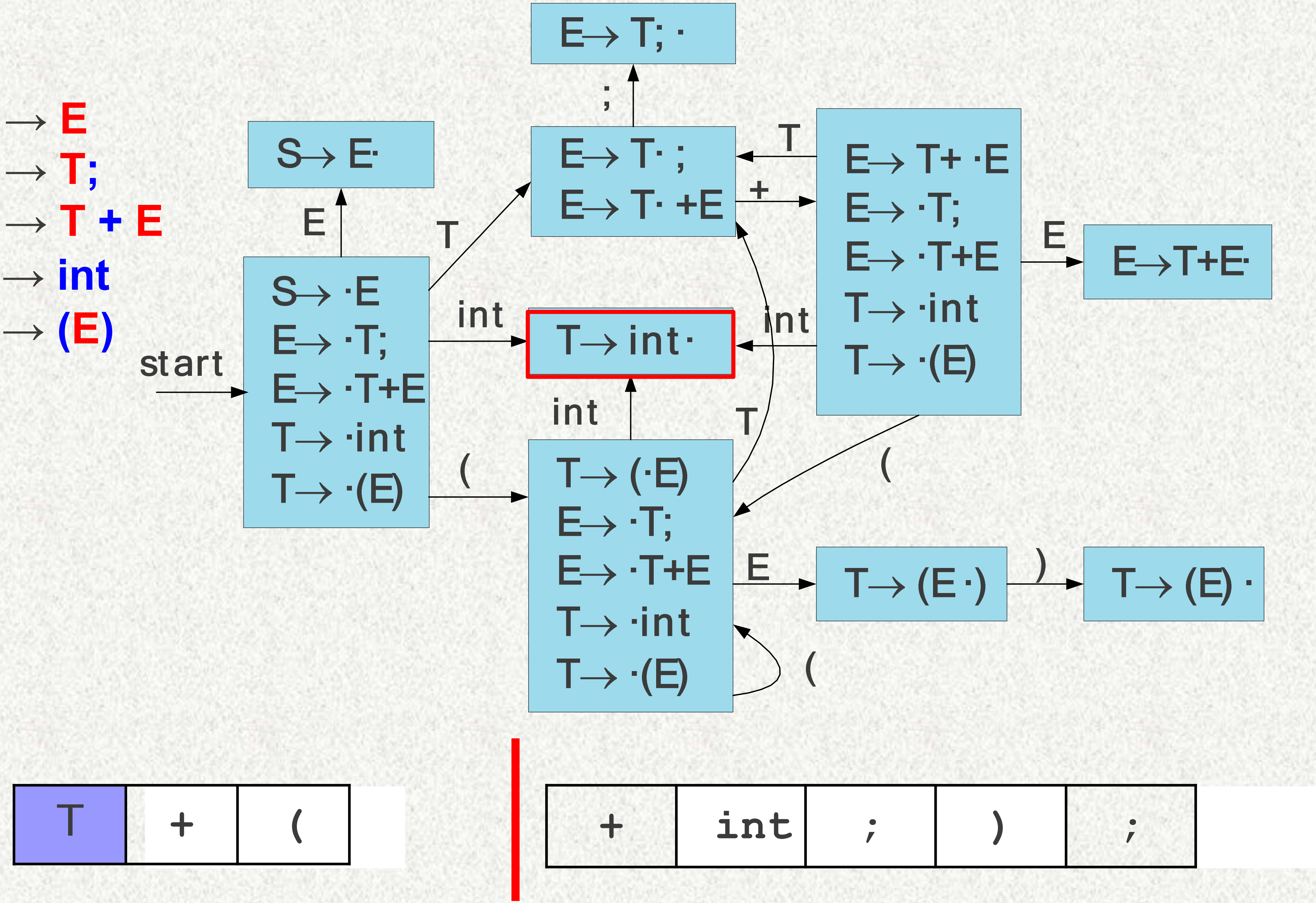


T	+	(	int

+	int	;	)	;

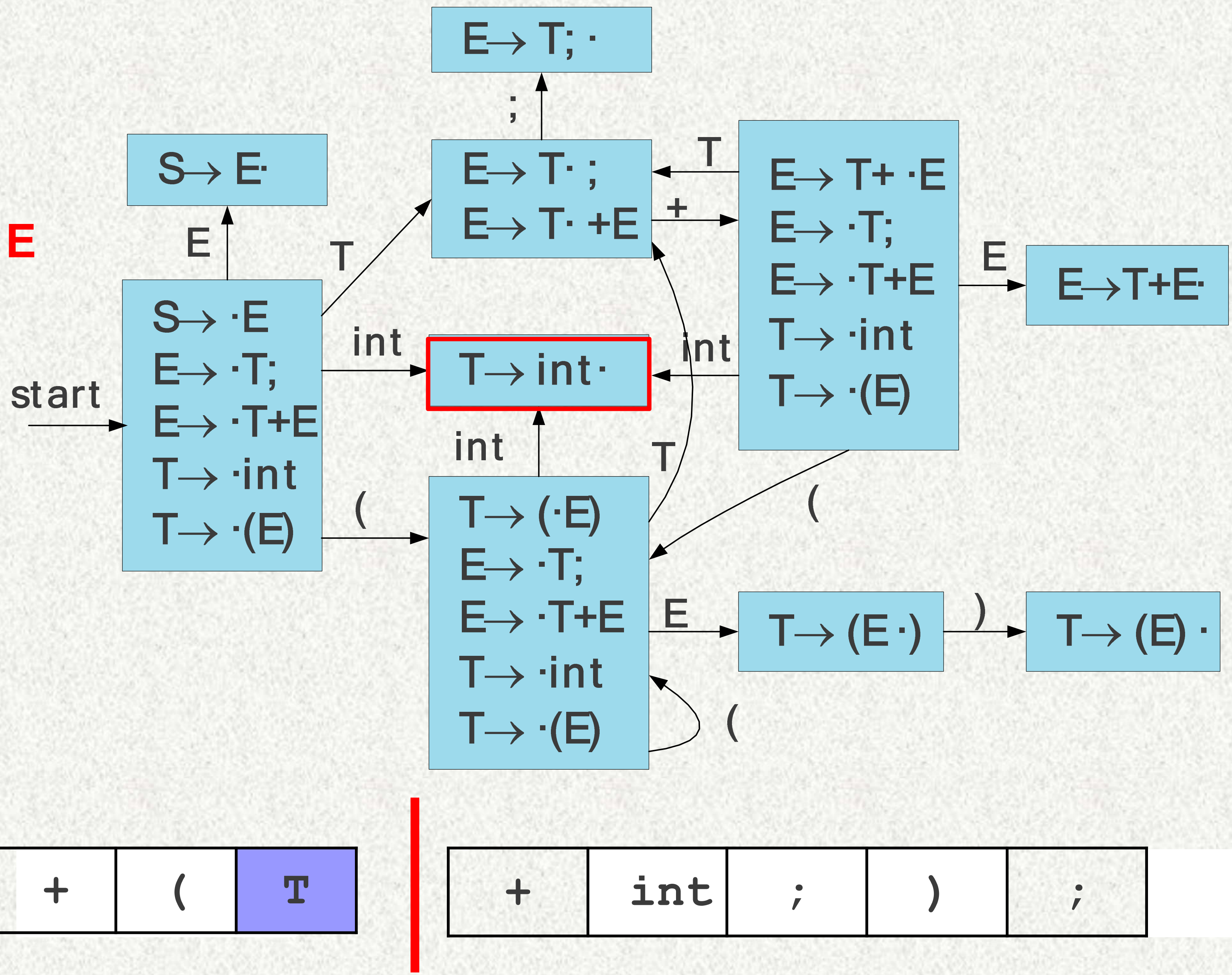
LR(0) Parsing

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



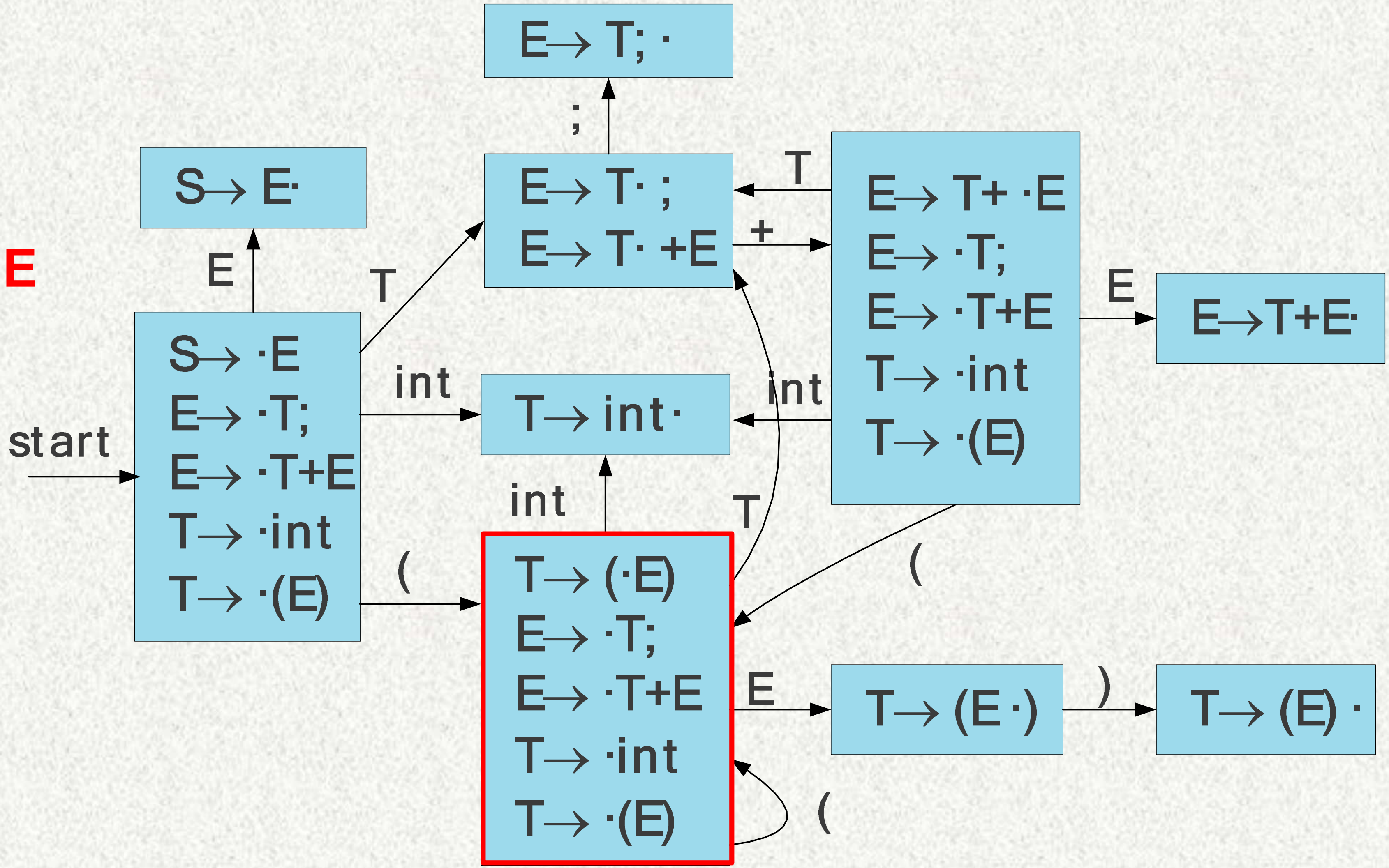
LR(0) Parsing

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



LR(0) Parsing

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

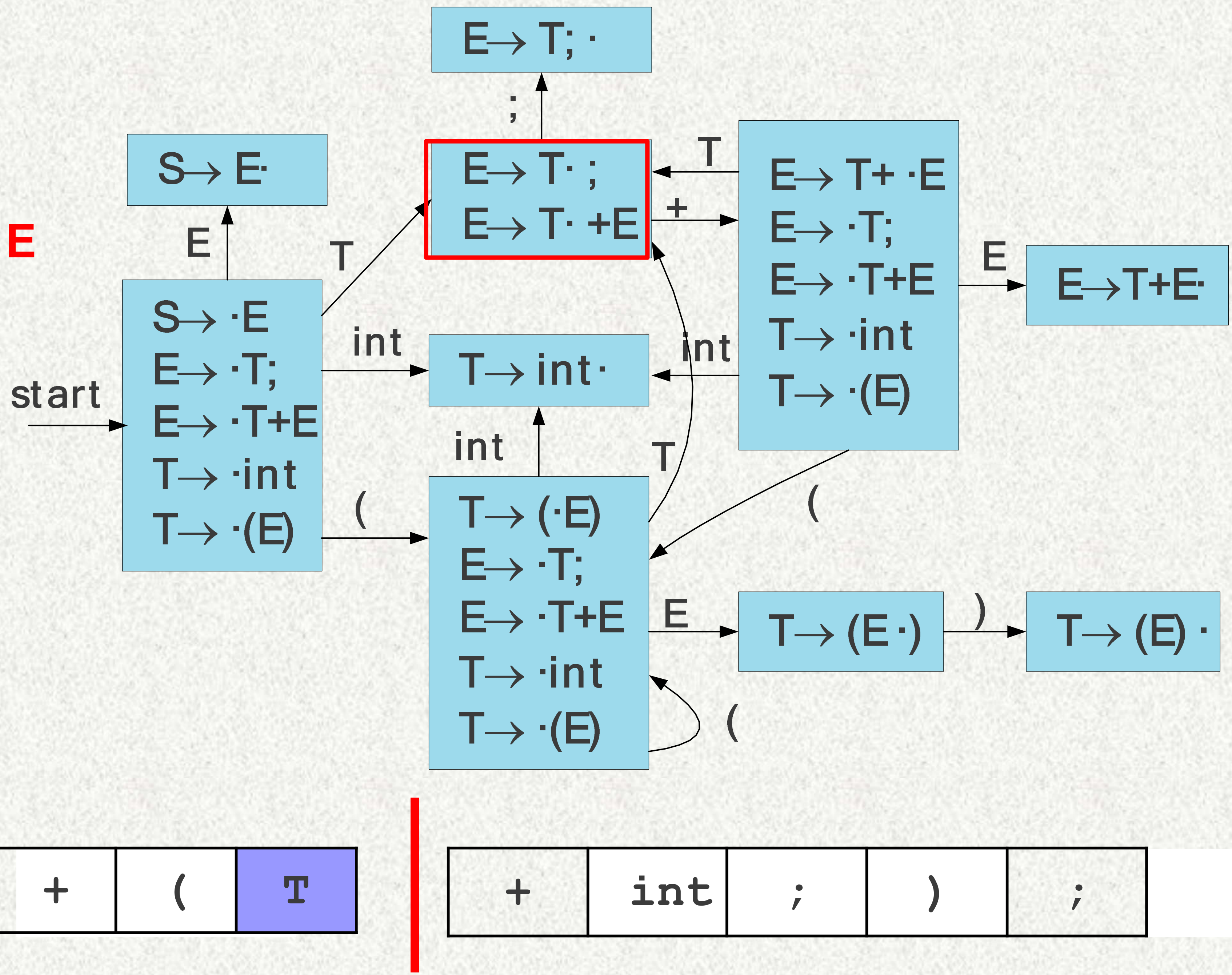


T	+	(	T

+	int	;	)	;

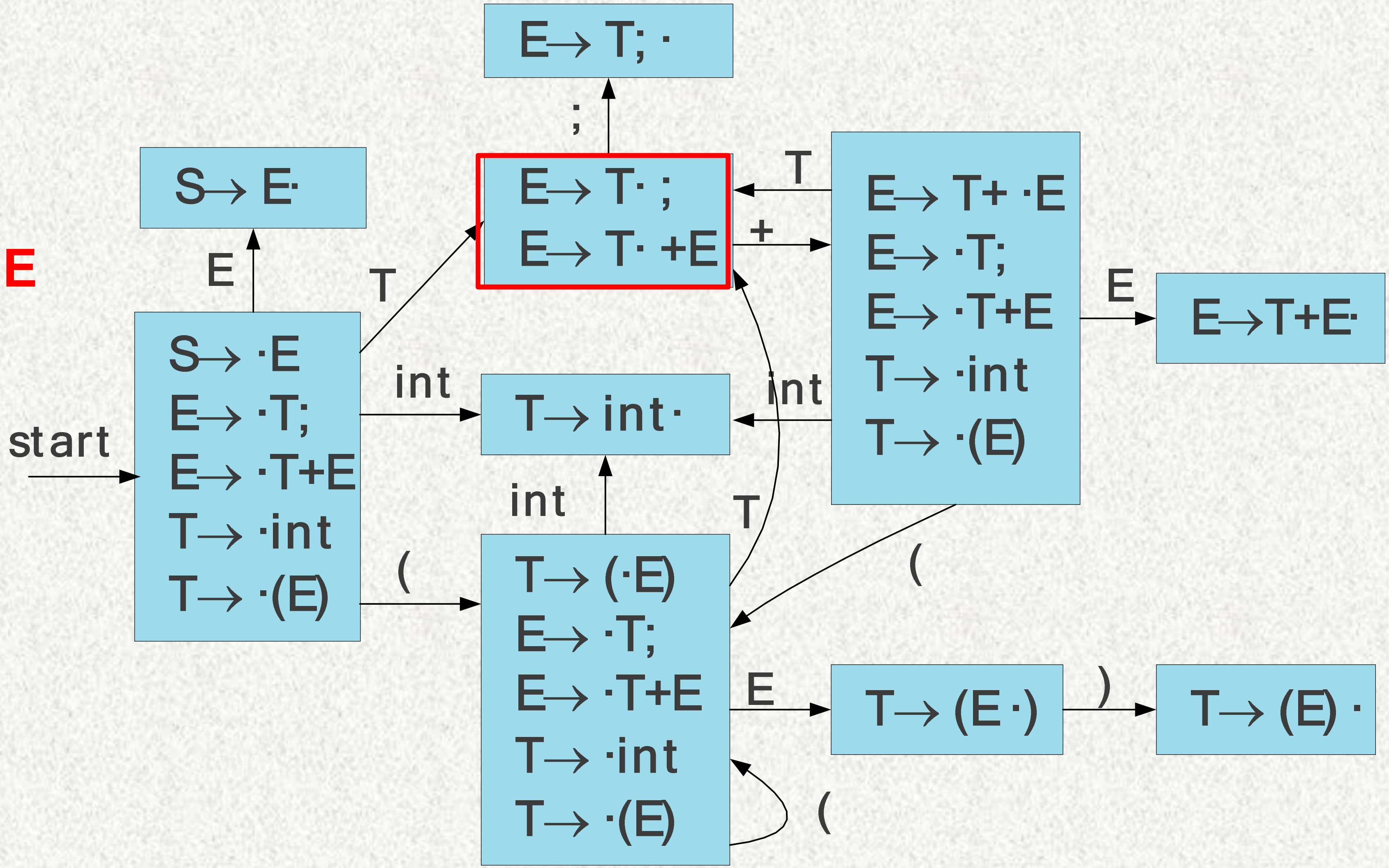
LR(0) Parsing

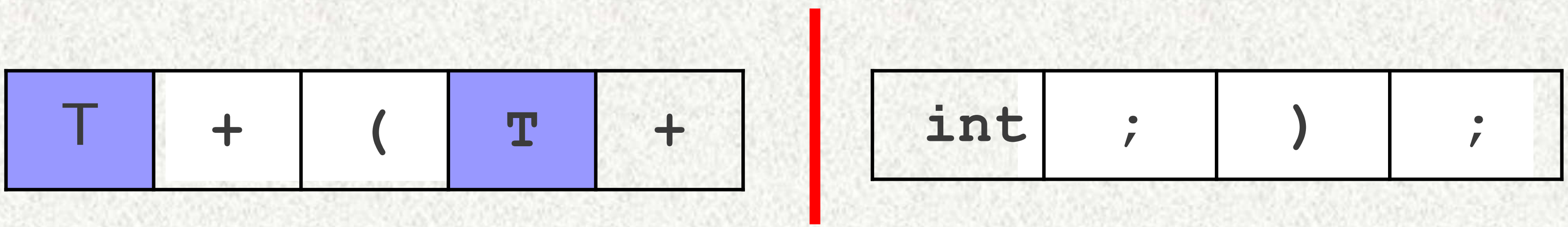
S → E
E → T;
E → T + E
T → int
T → (E)



LR(0) Parsing

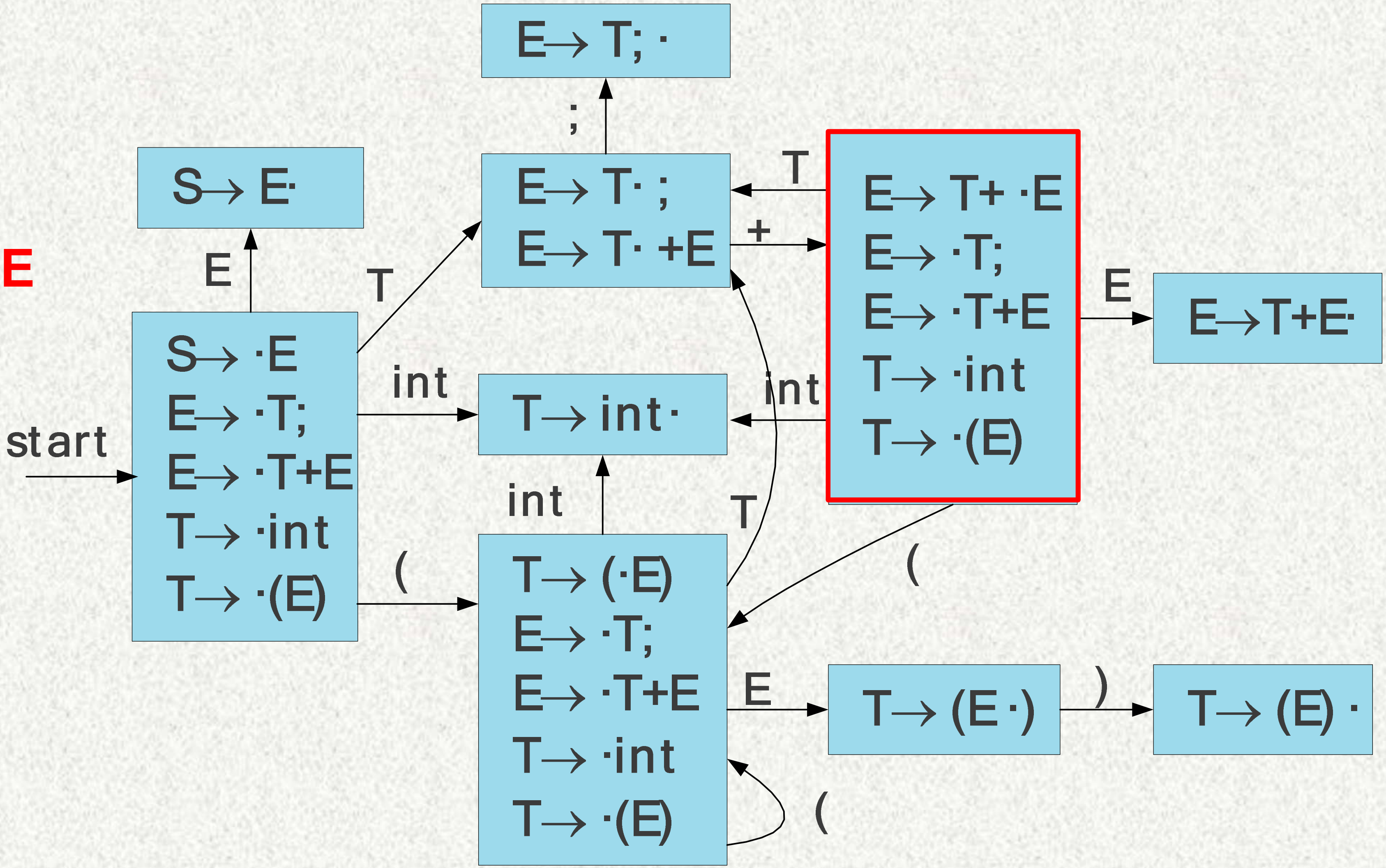
S → E
E → T;
E → T + E
T → int
T → (E)





LR(0) Parsing

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

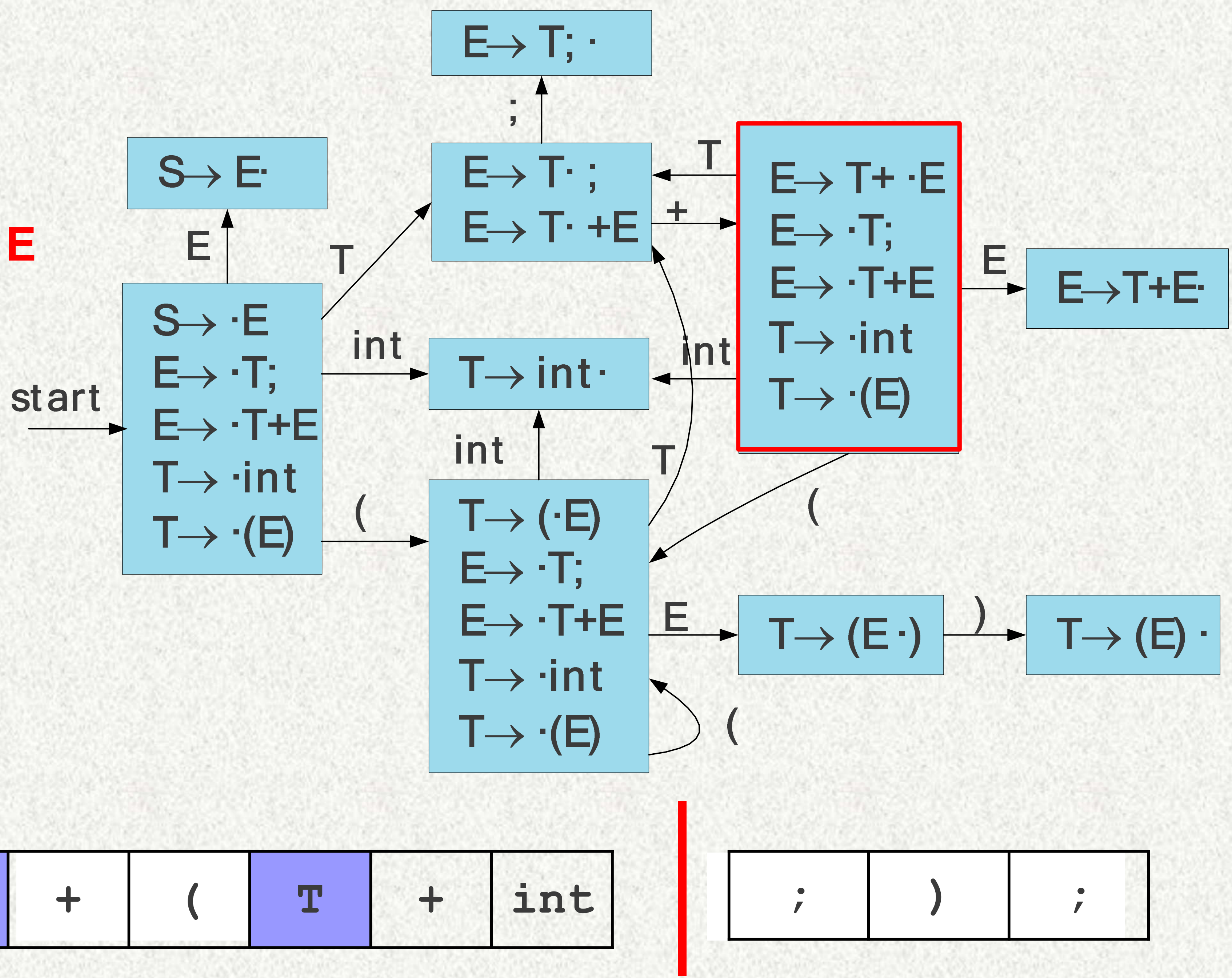


T	+	(	T	+

int	;	)	;

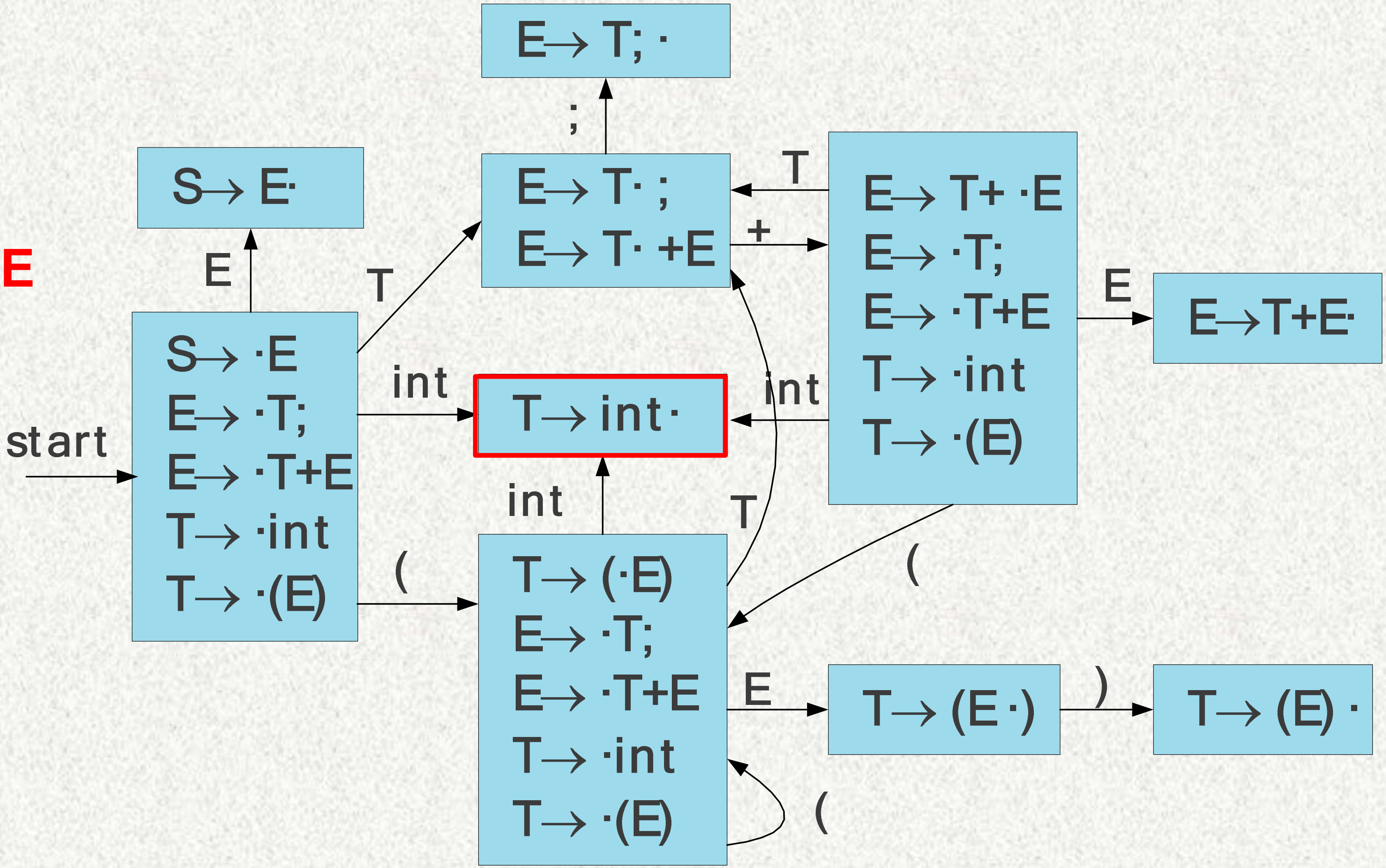
LR(0) Parsing

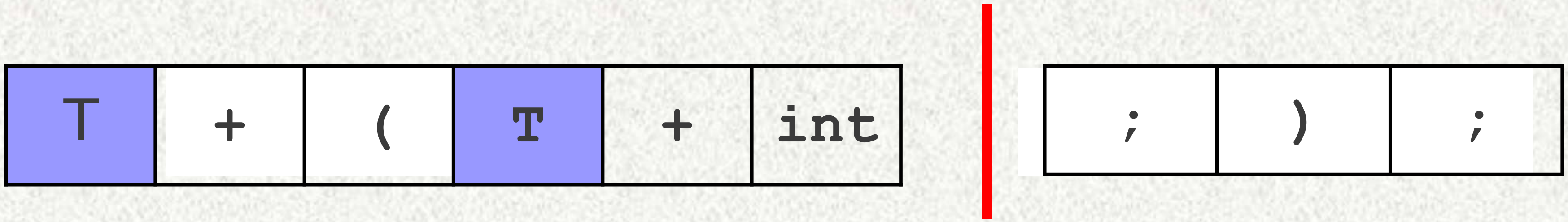
S → E
E → T;
E → T + E
T → int
T → (E)



LR(0) Parsing

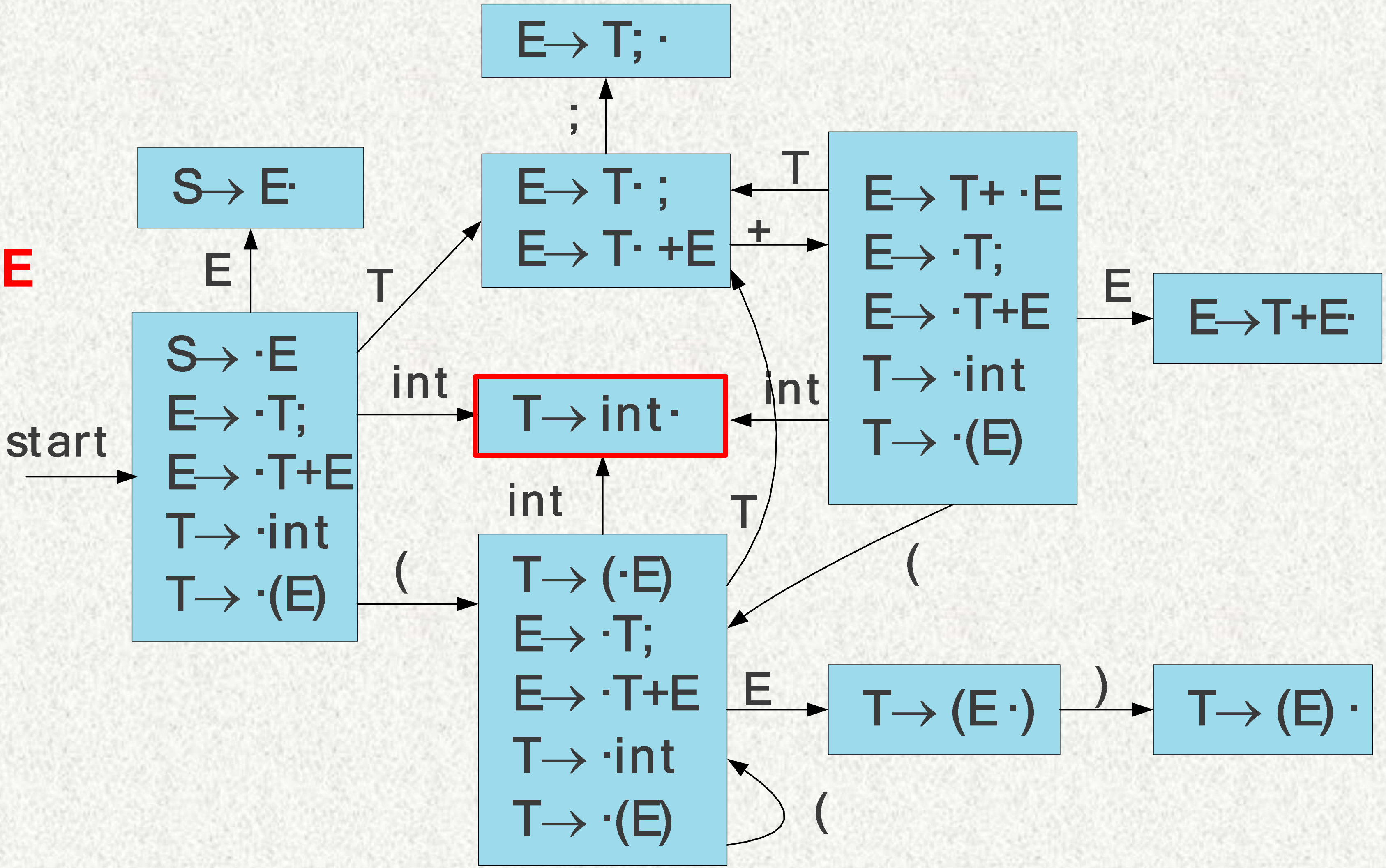
S → E
E → T;
E → T + E
T → int
T → (E)





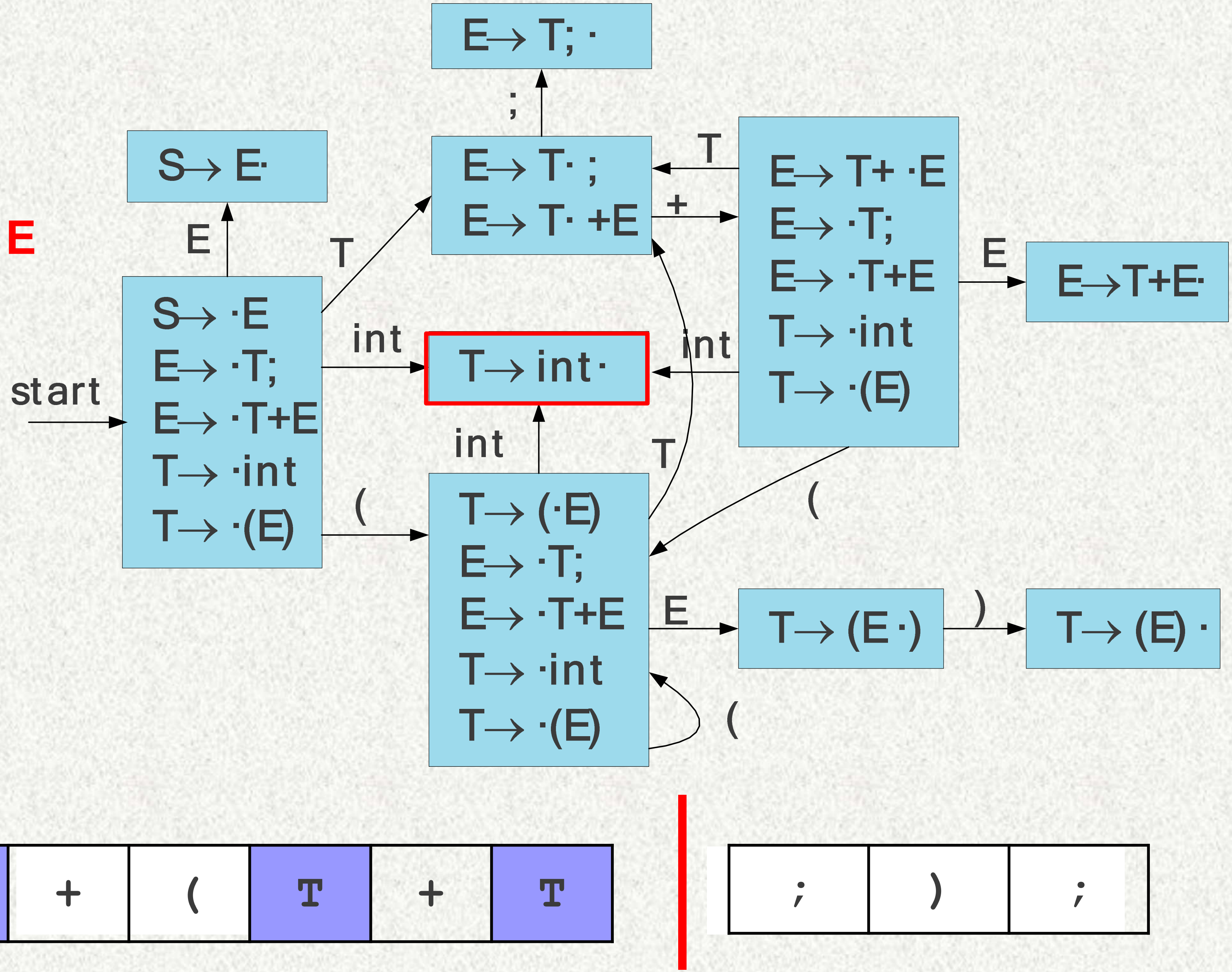
LR(0) Parsing

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



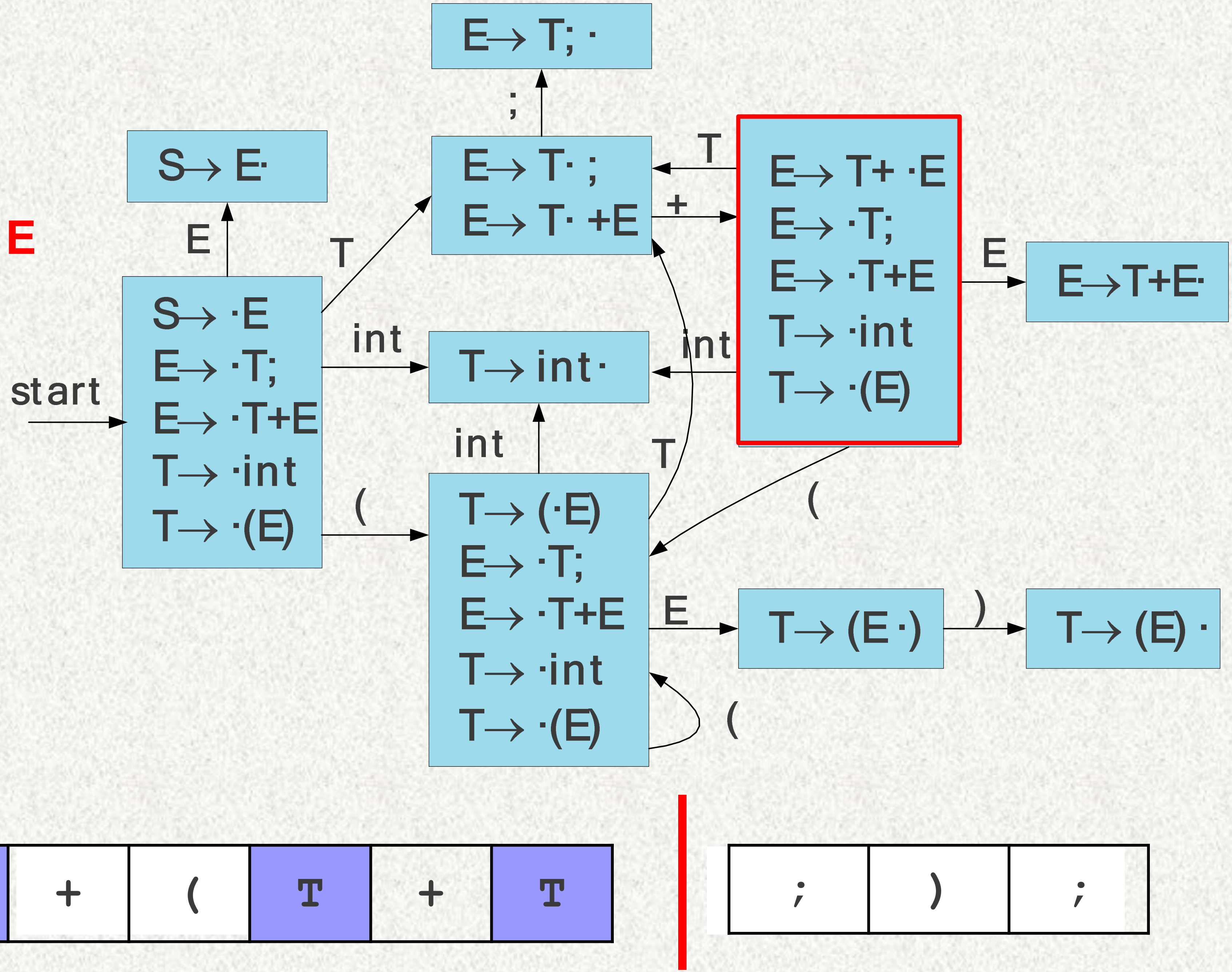
LR(0) Parsing

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



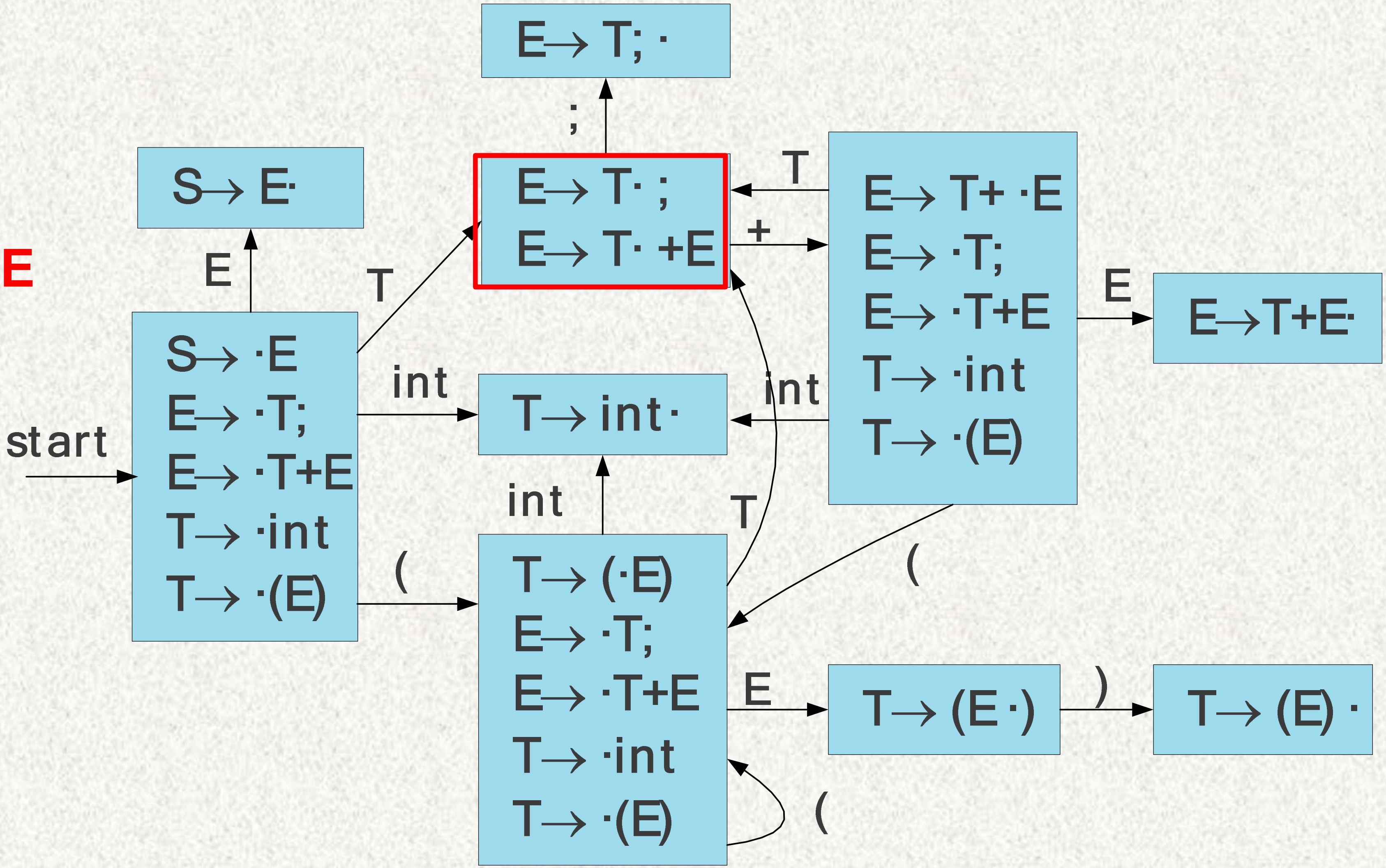
LR(0) Parsing

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



LR(0) Parsing

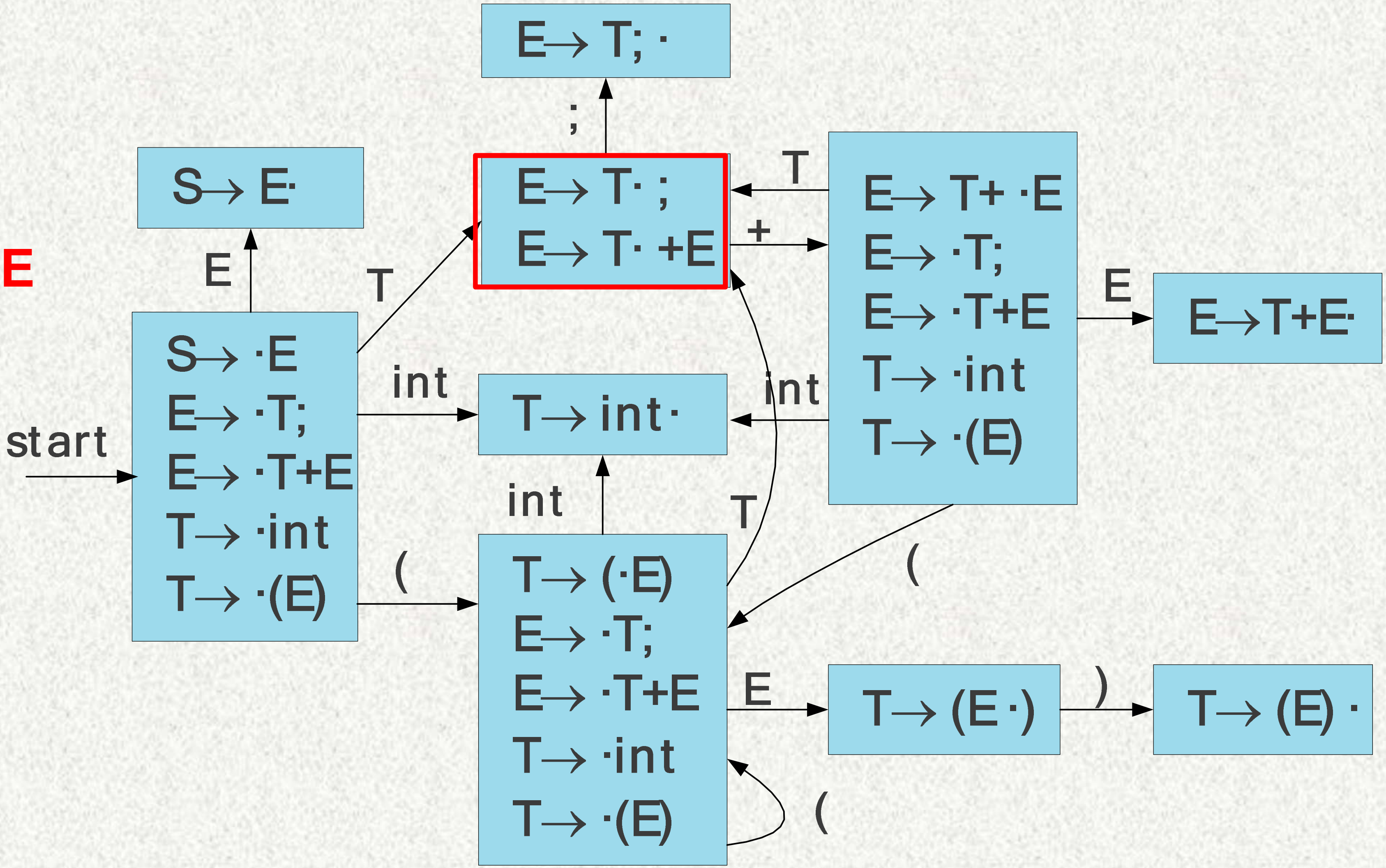
S → E
E → T;
E → T + E
T → int
T → (E)

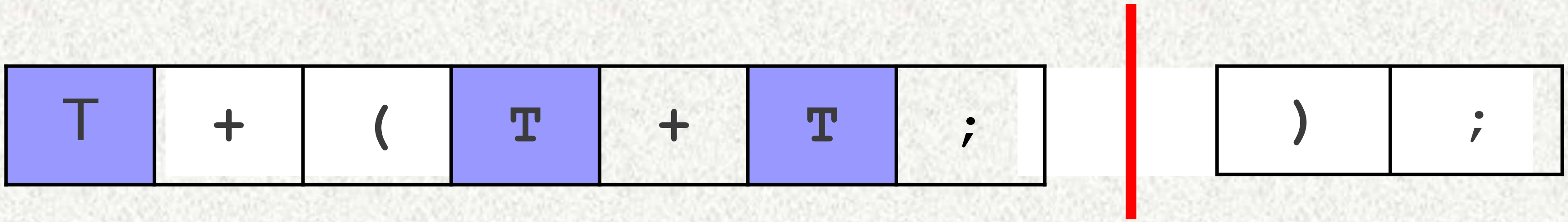


T	+	(	T	+	T	;	)	;

LR(0) Parsing

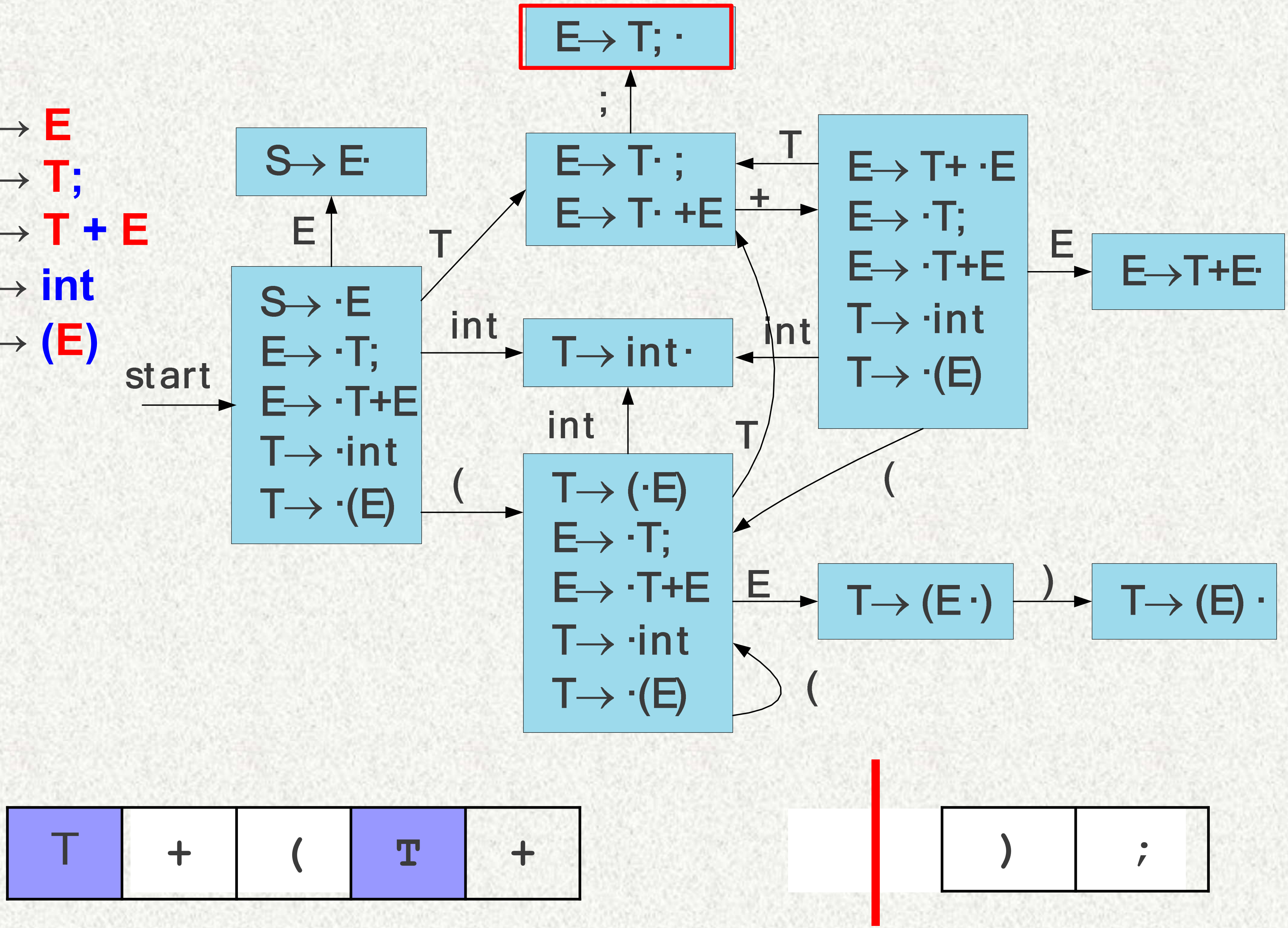
$S \rightarrow E$
$E \rightarrow T;$
$E \rightarrow T + E$
$T \rightarrow \text{int}$
$T \rightarrow (E)$





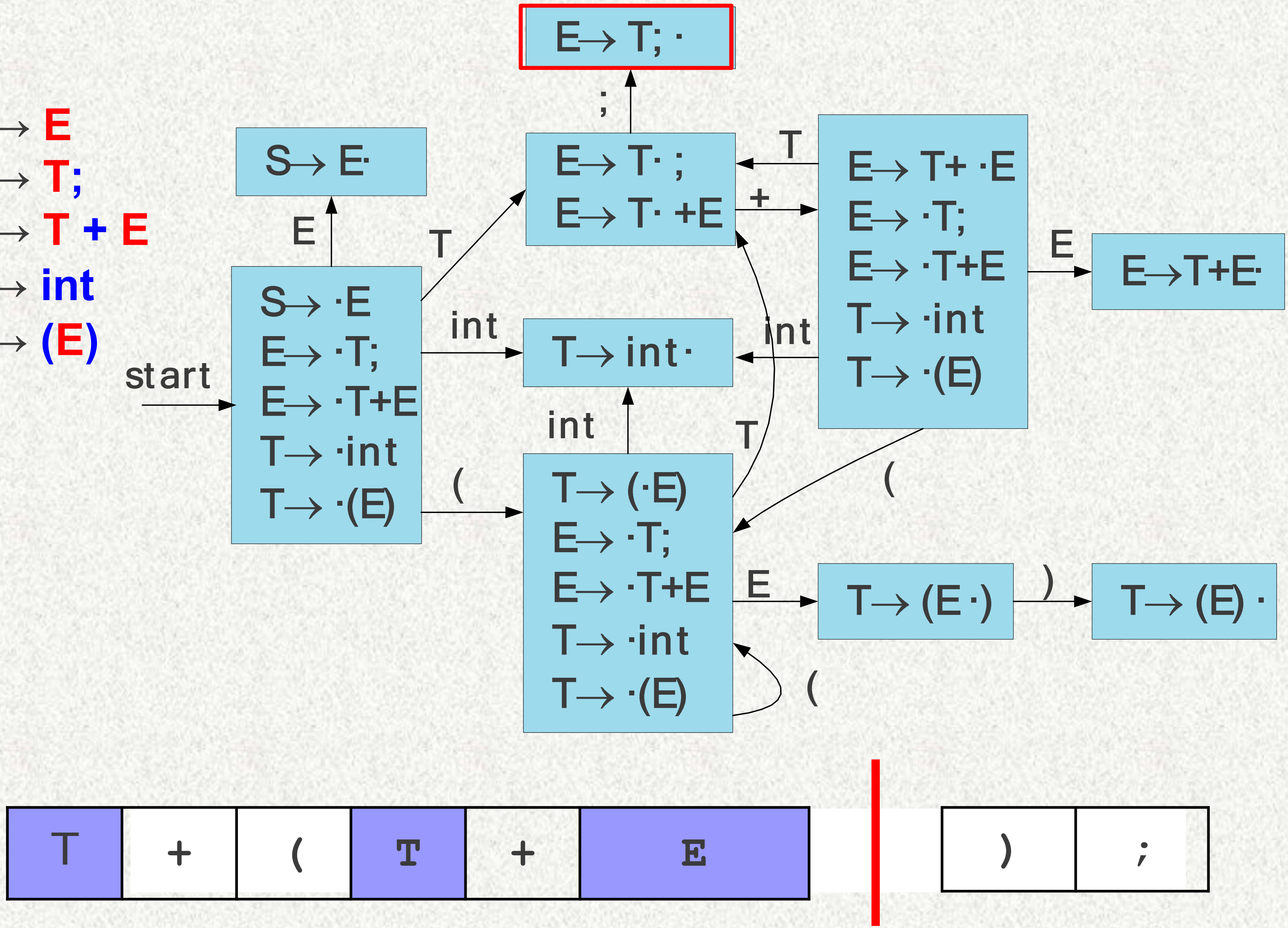
LR(0) Parsing

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



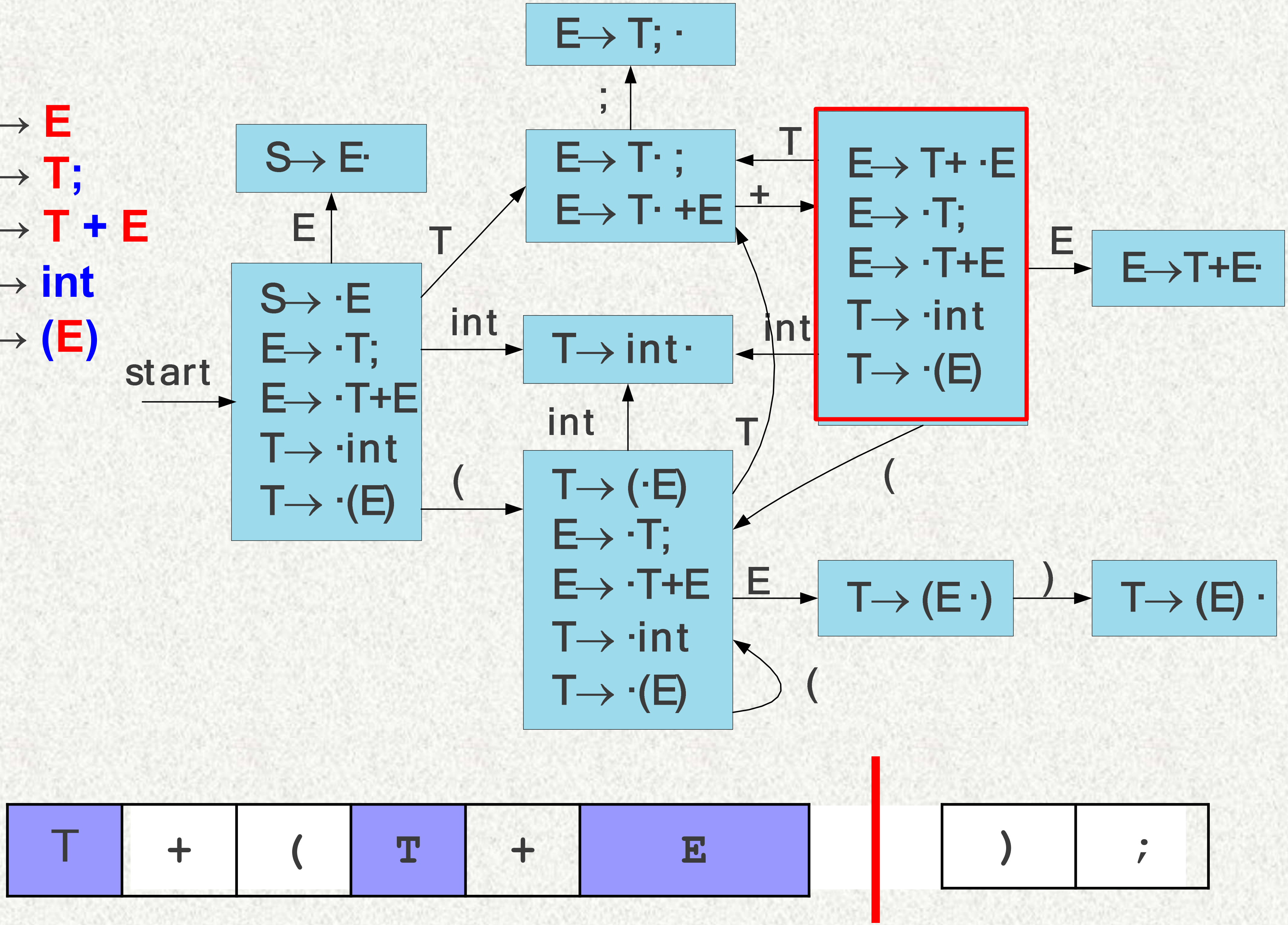
LR(0) Parsing

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



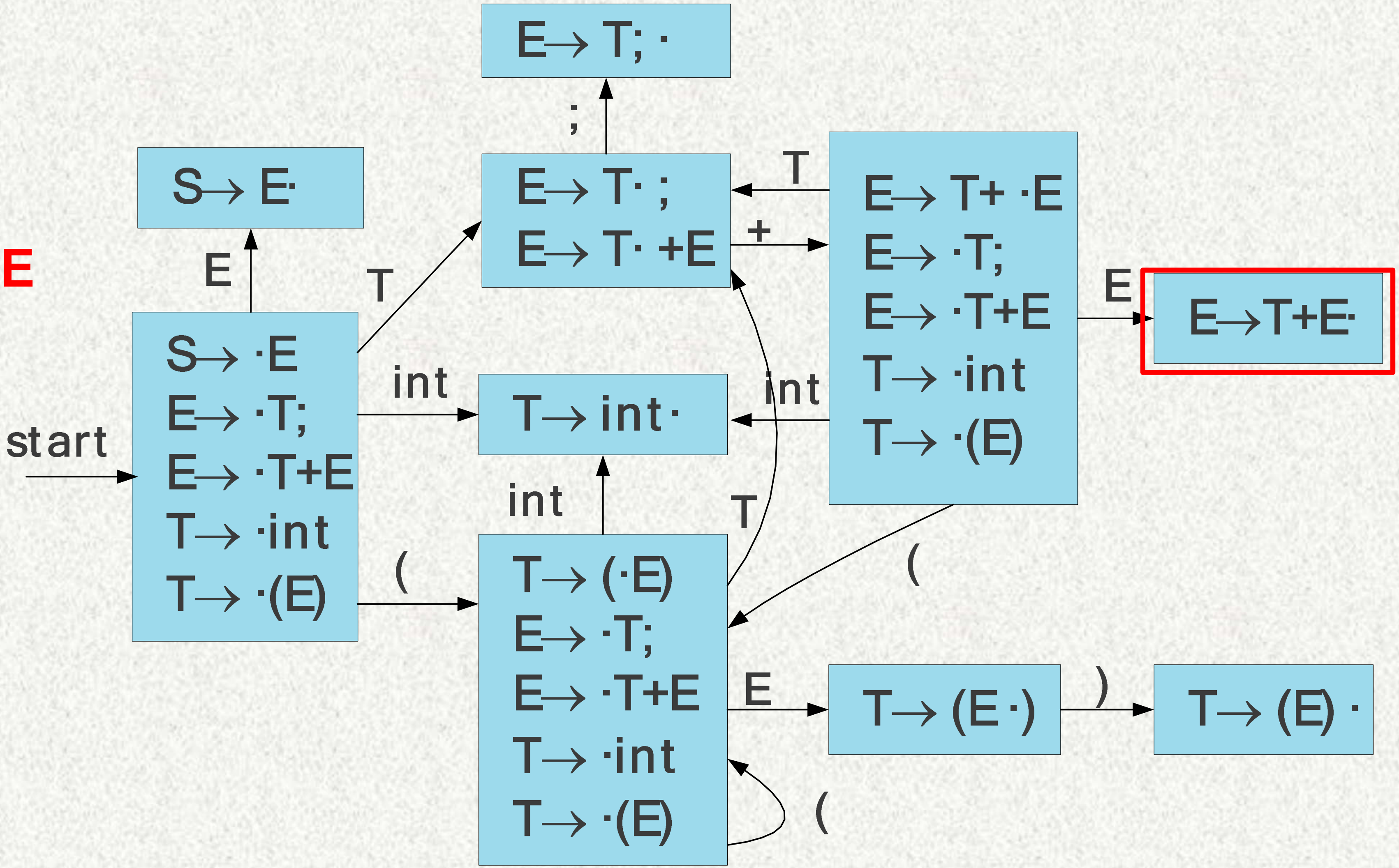
LR(0) Parsing

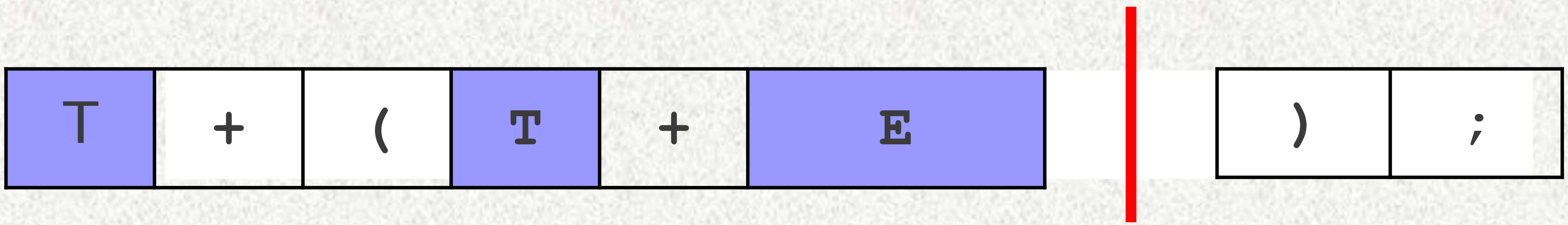
S → E
E → T;
E → T + E
T → int
T → (E)



LR(0) Parsing

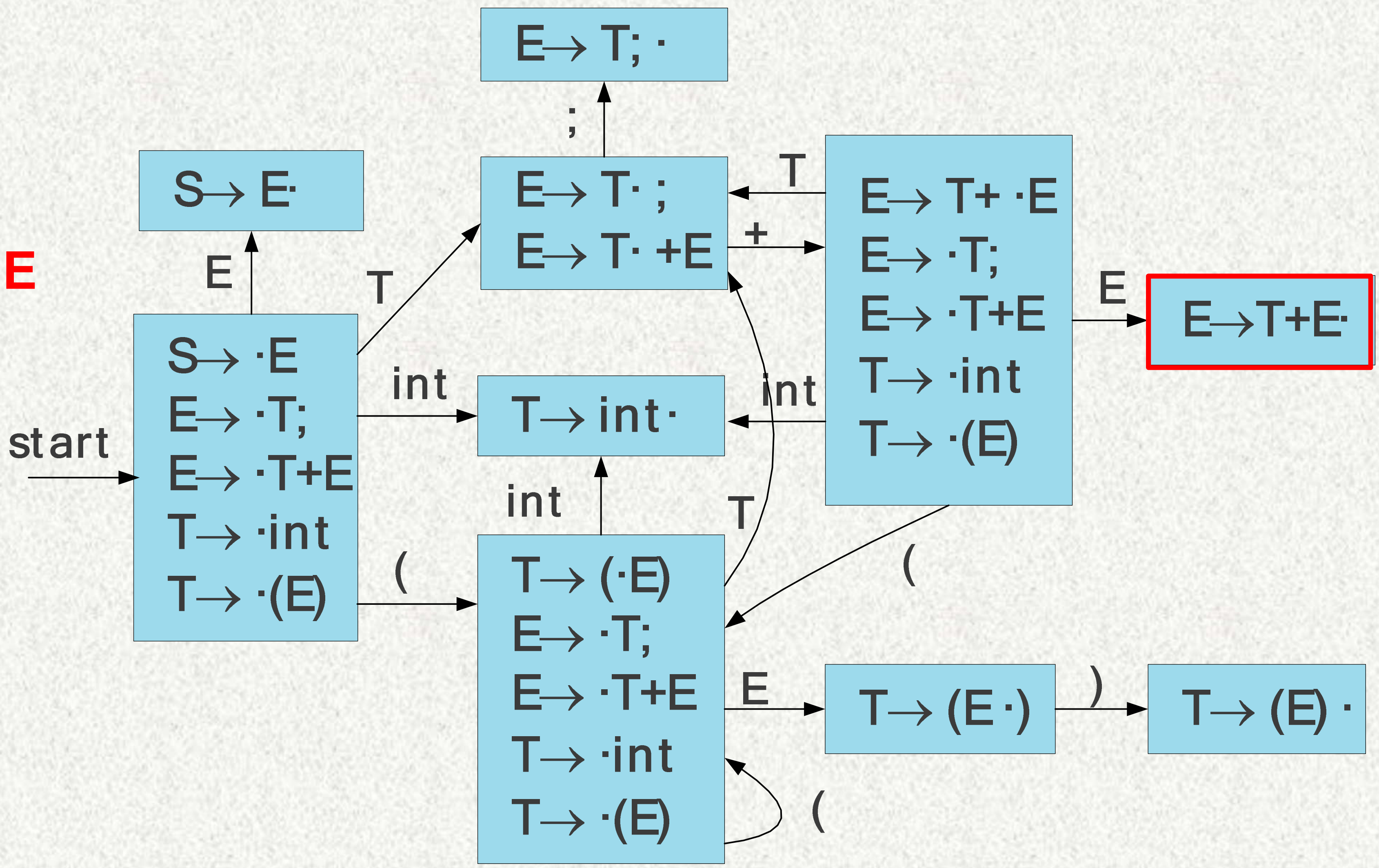
S → E
E → T;
E → T + E
T → int
T → (E)

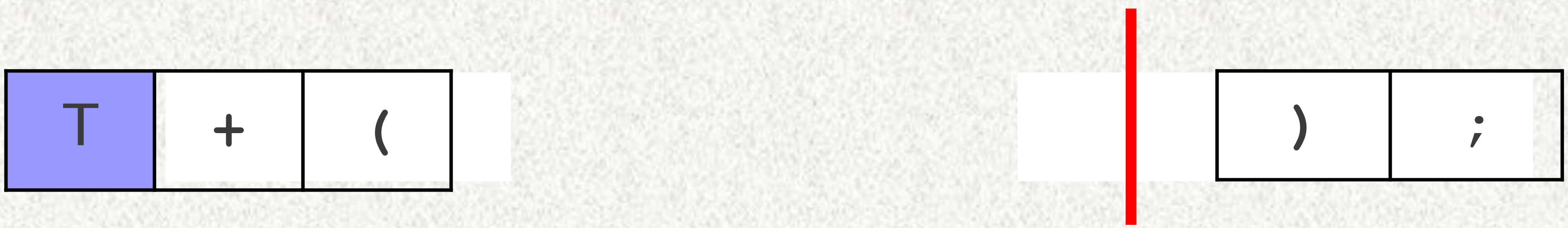




LR(0) Parsing

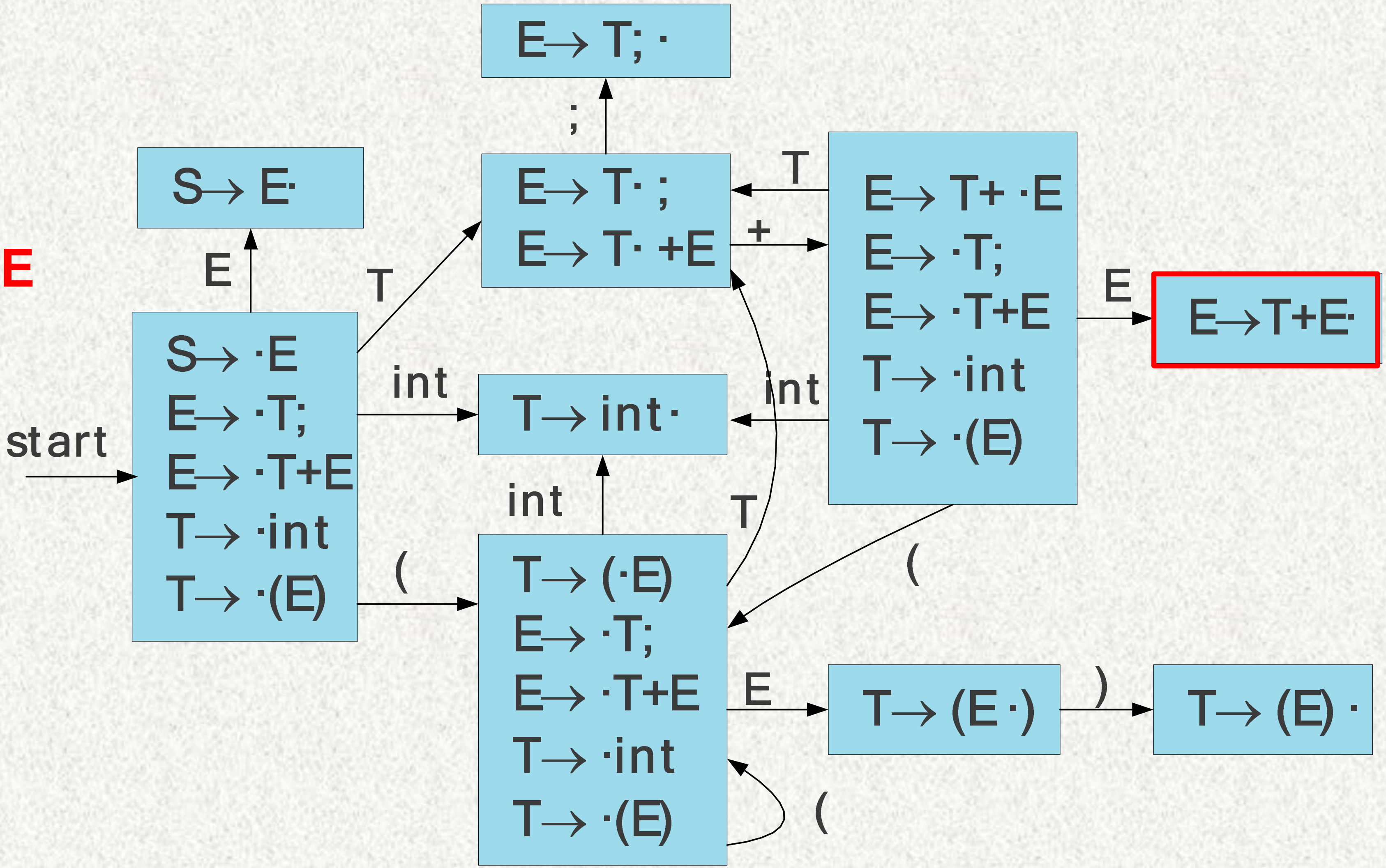
S → E
E → T;
E → T + E
T → int
T → (E)

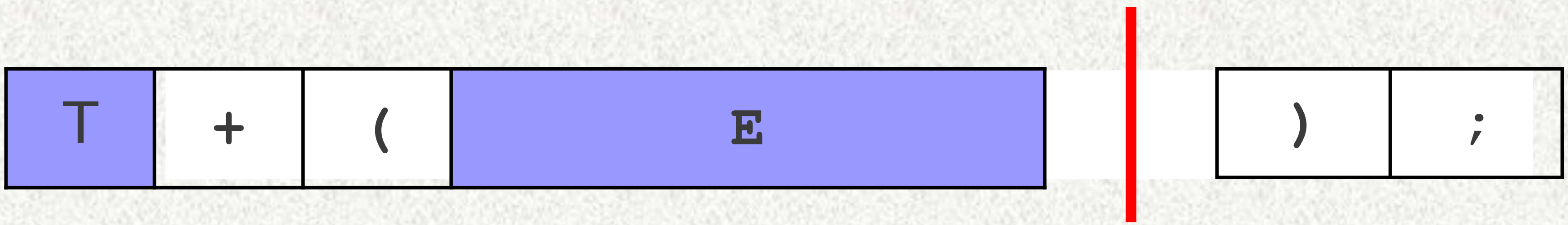




LR(0) Parsing

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



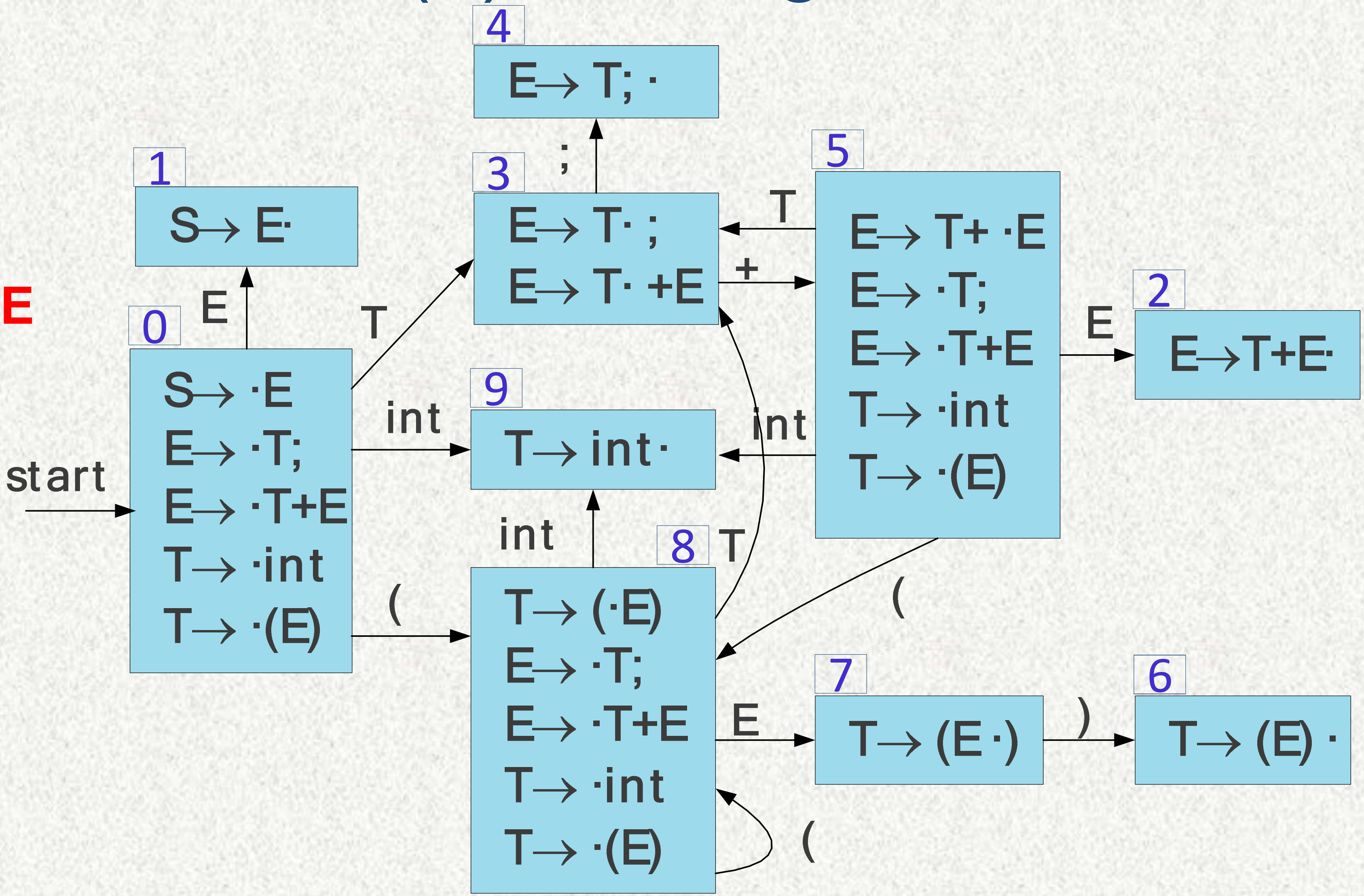


An optimization

- Rather than restart the automaton on each reduction, remember what state we were in for each symbol.
- When applying a reduction, restart the automaton from the last known good state.

LR(0) Parsing

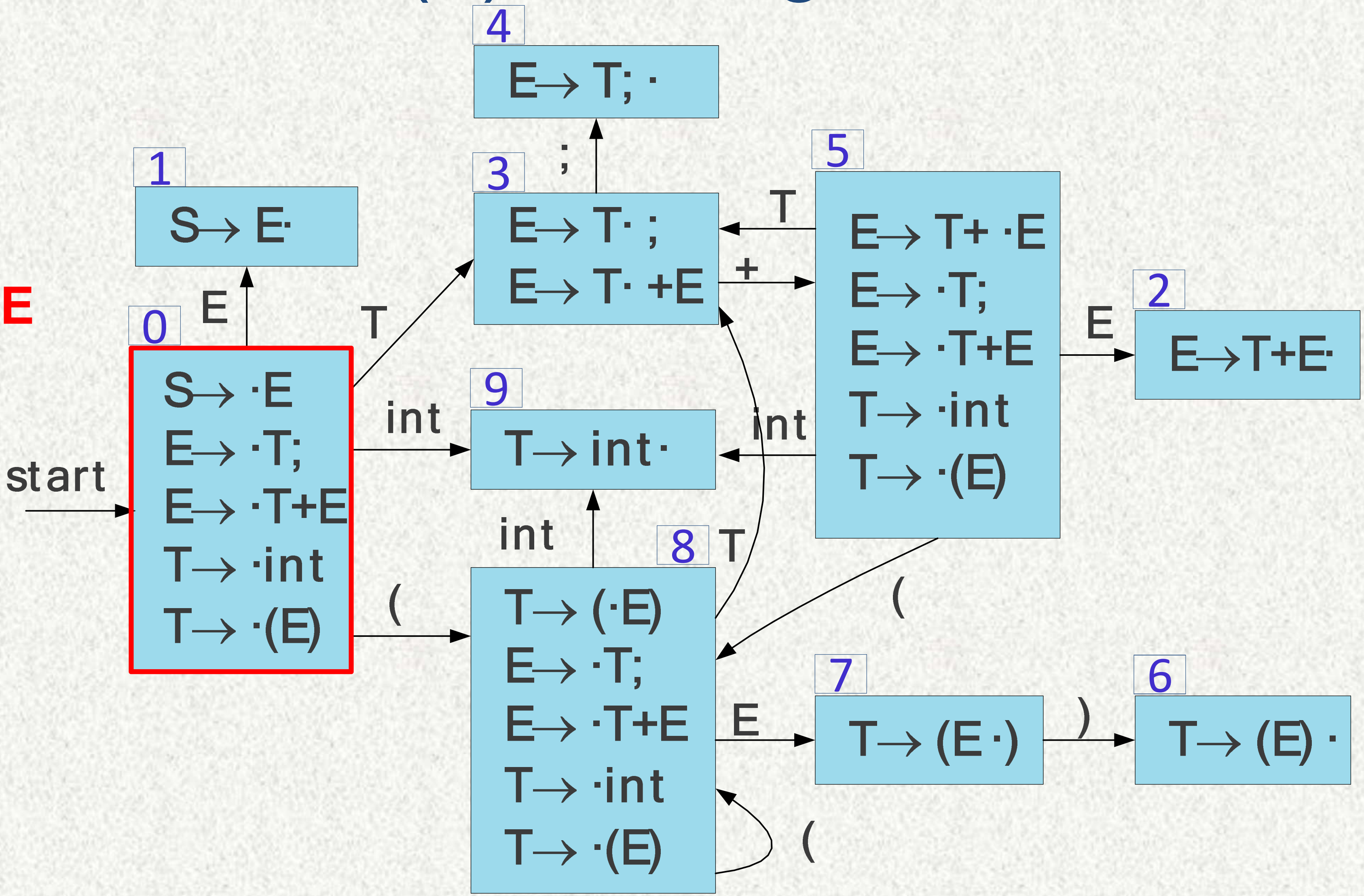
S → E
E → T;
E → T + E
T → int
T → (E)



int	+	(	int	+	int	;	)	;

LR(0) Parsing

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



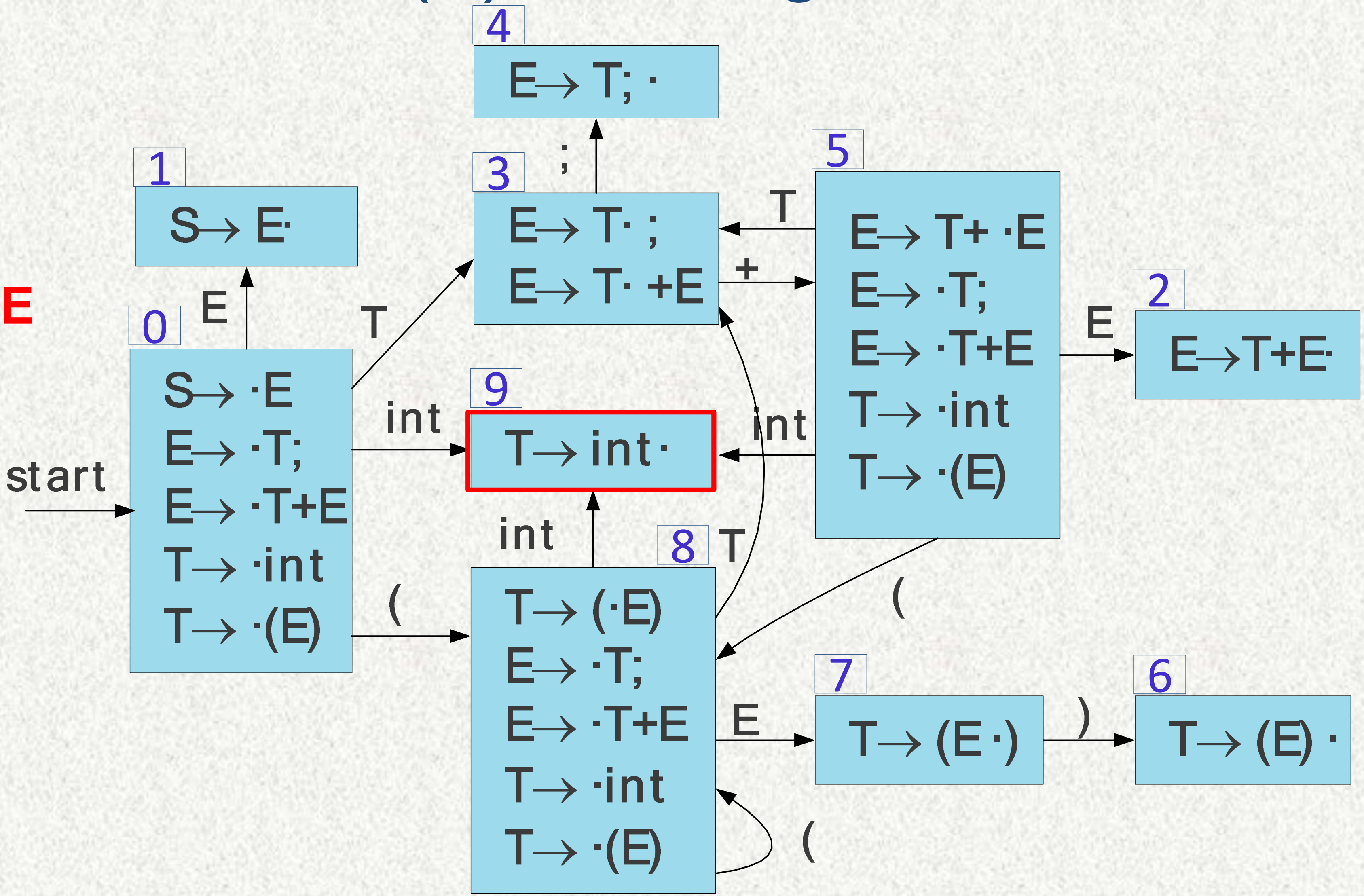
$
0

int	+	(	int	+	int	;	)	;

$

LR(0) Parsing

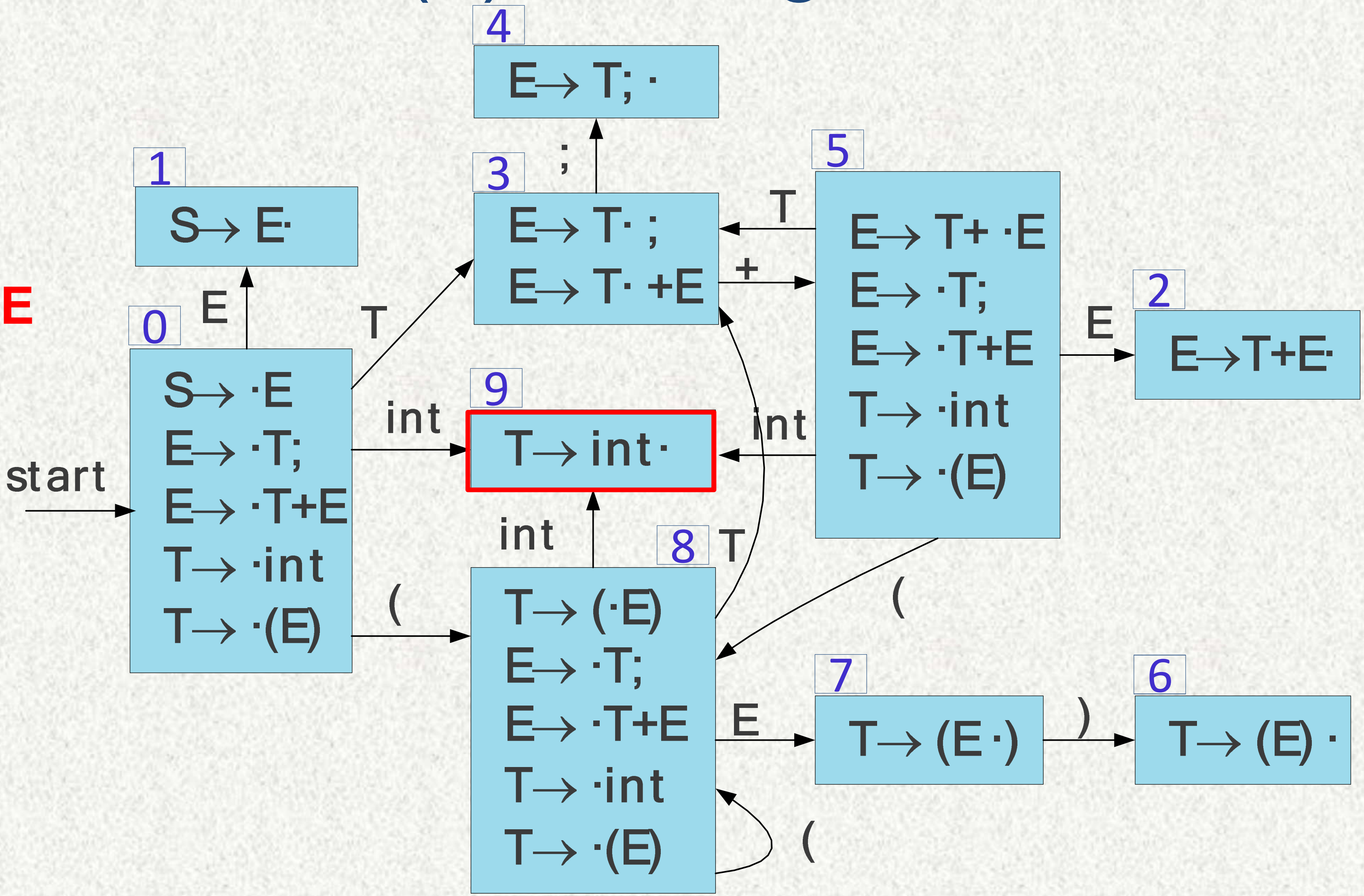
S → E
E → T;
E → T + E
T → int
T → (E)



$	int		+	(	int	+	int	;	)	;	$
0	9										

LR(0) Parsing

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

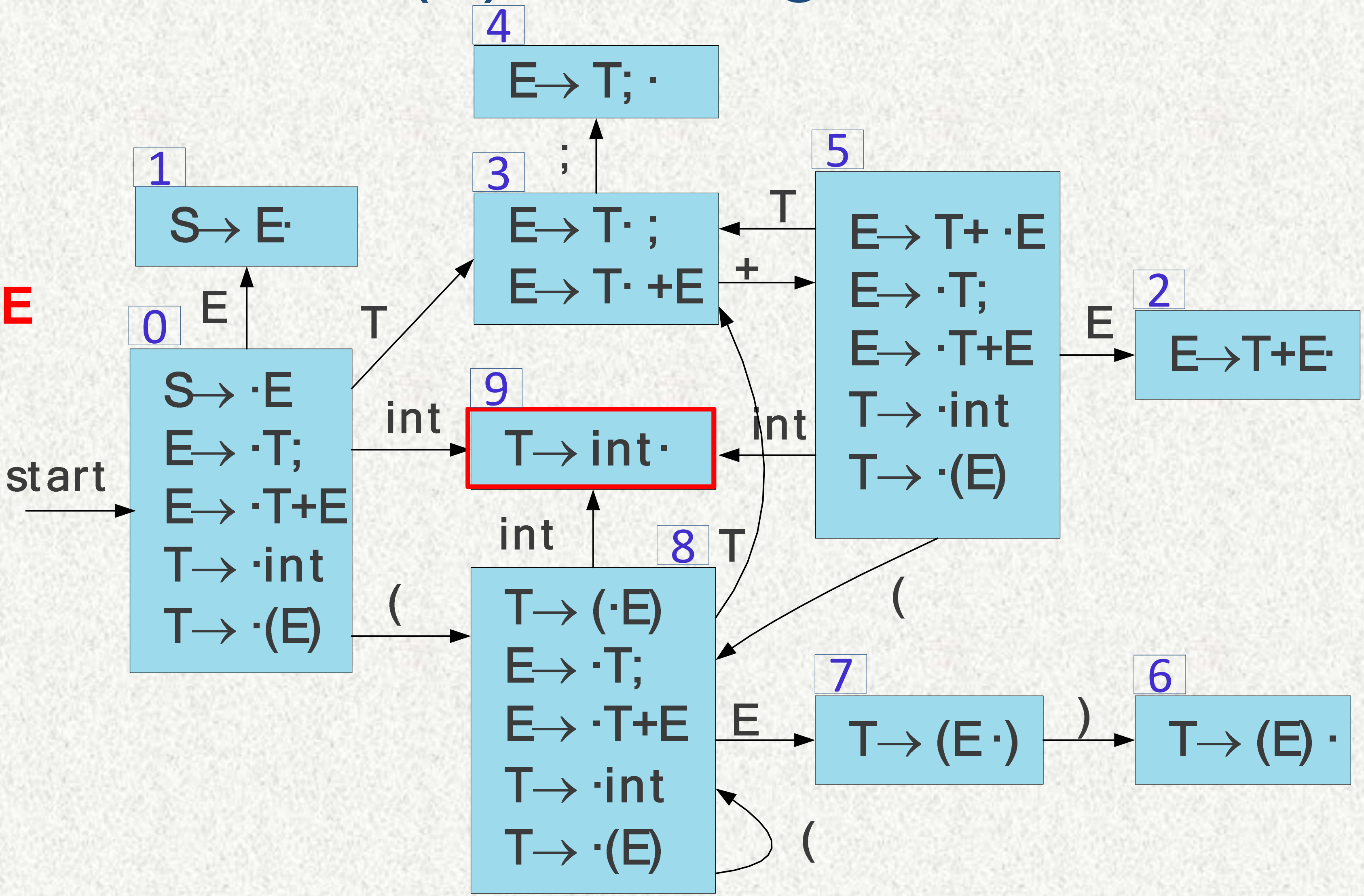


+	(	int	+	int	;	)	;	$

$
0

LR(0) Parsing

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

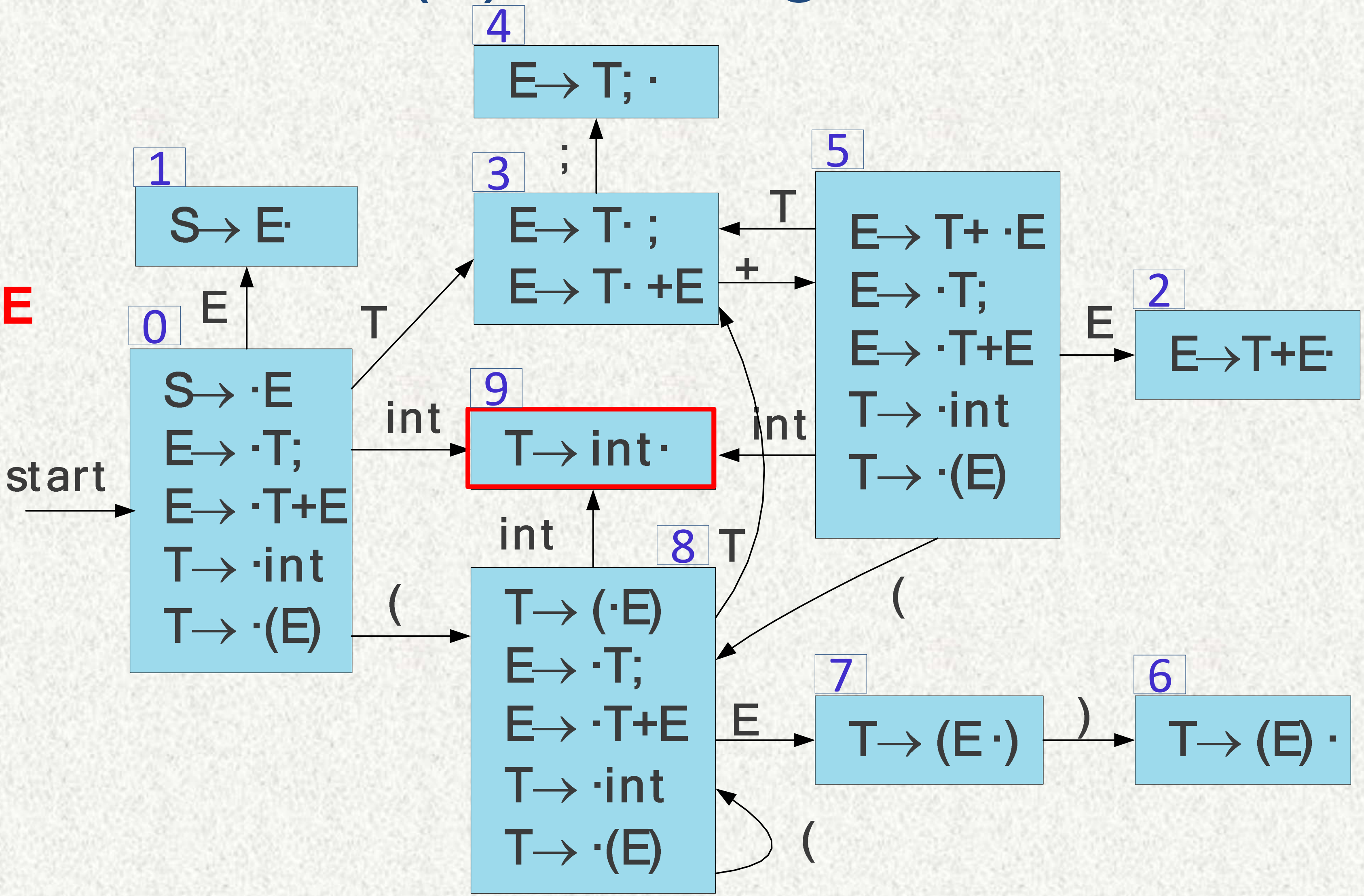


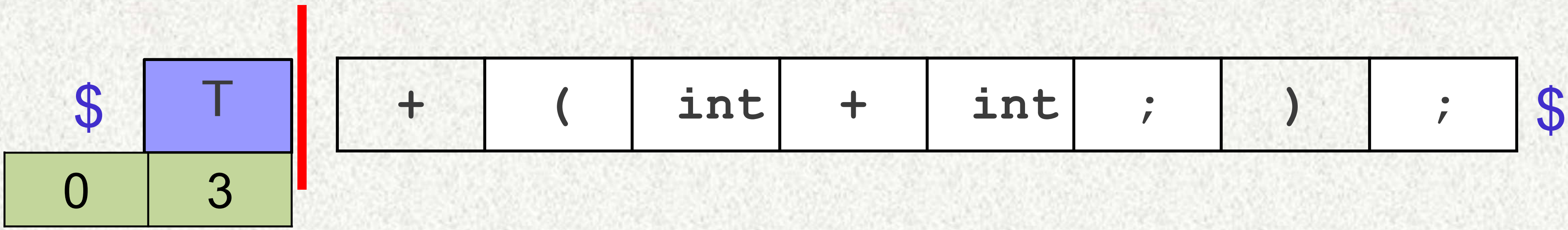
$ 0 T

+	(	int	+	int	;	)	;

LR(0) Parsing

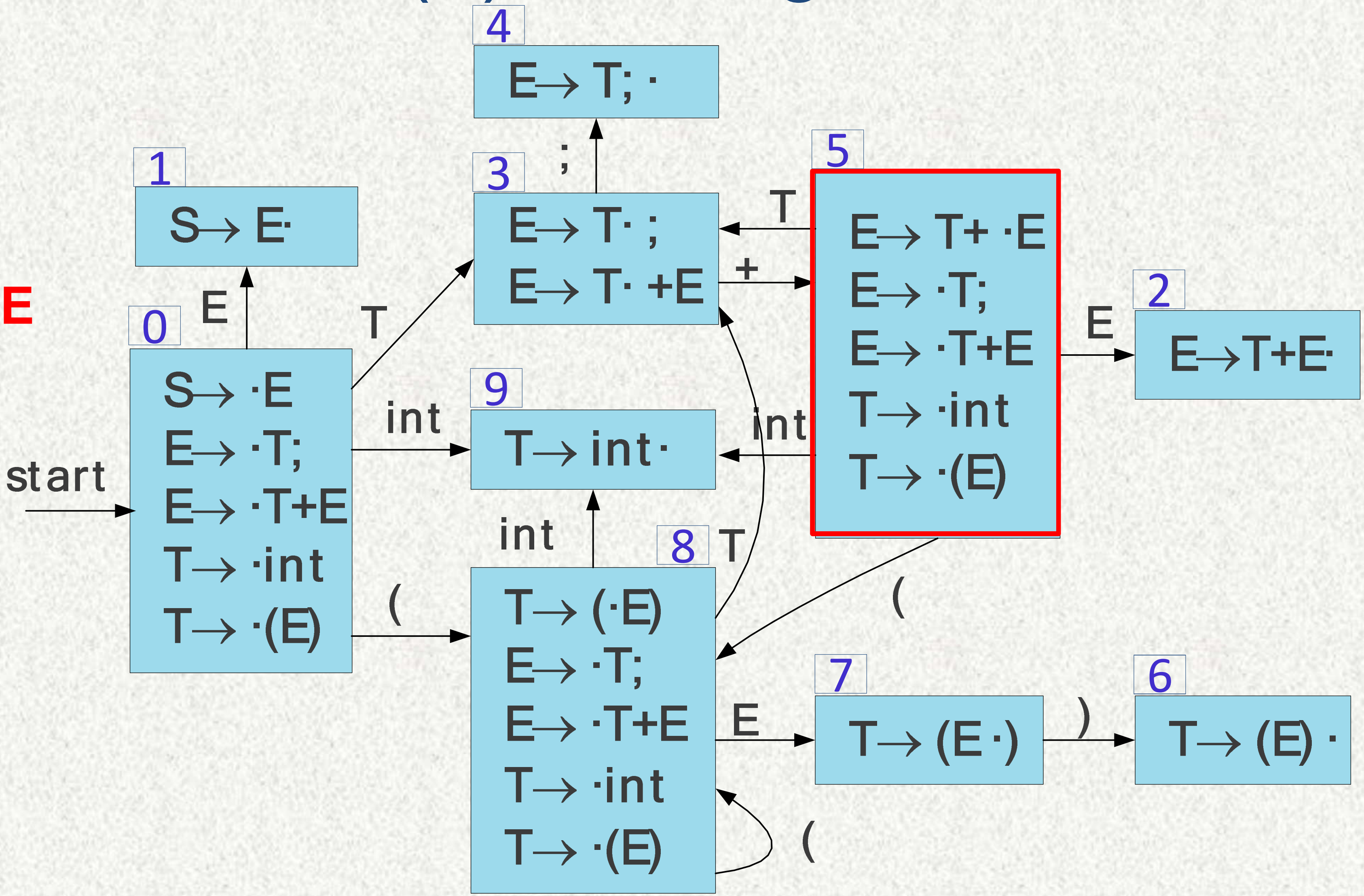
S → E
E → T;
E → T + E
T → int
T → (E)





LR(0) Parsing

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

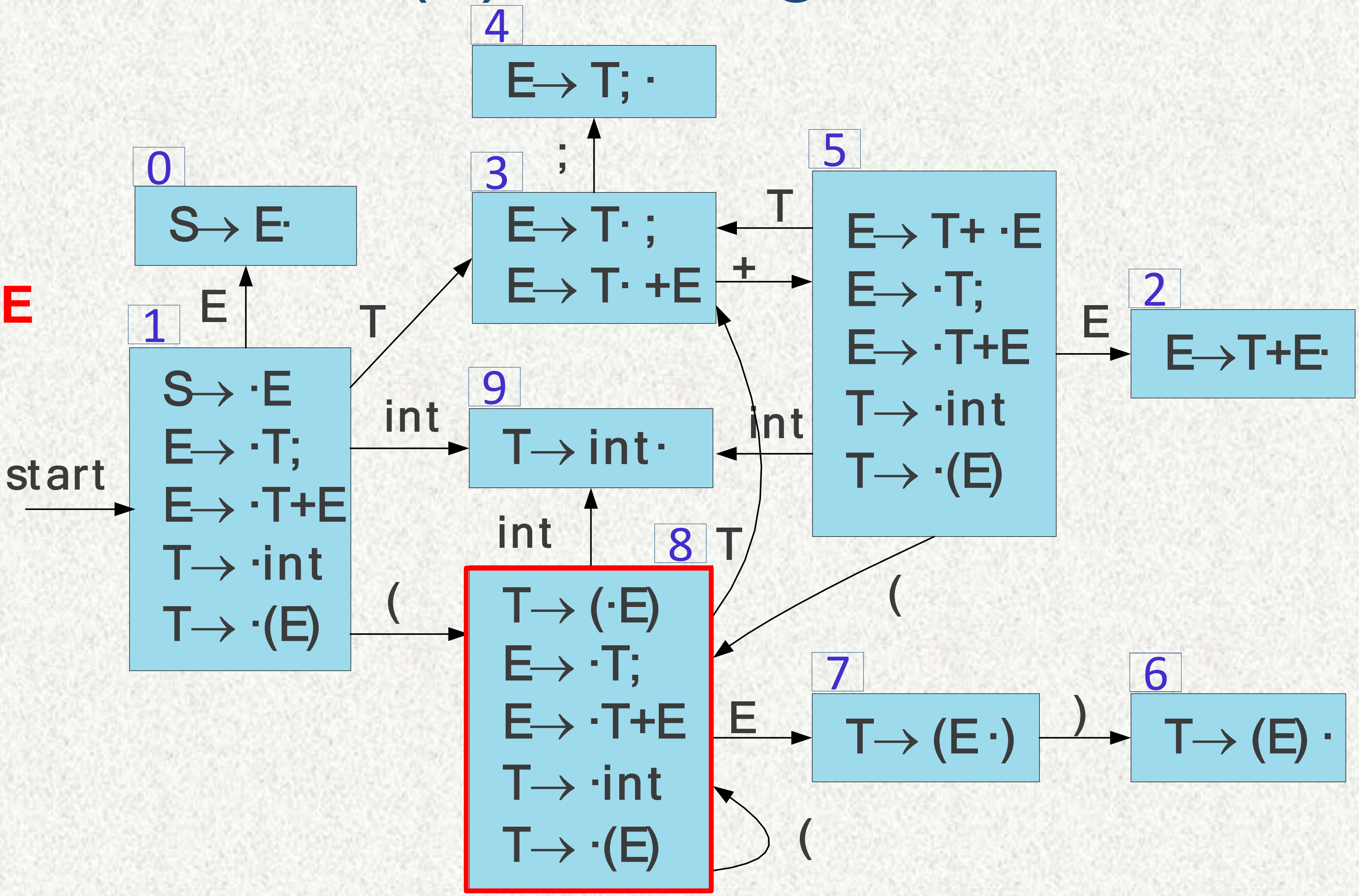


$	T	+
0	3	5

(	int	+	int	;	)	;	$

LR(0) Parsing

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

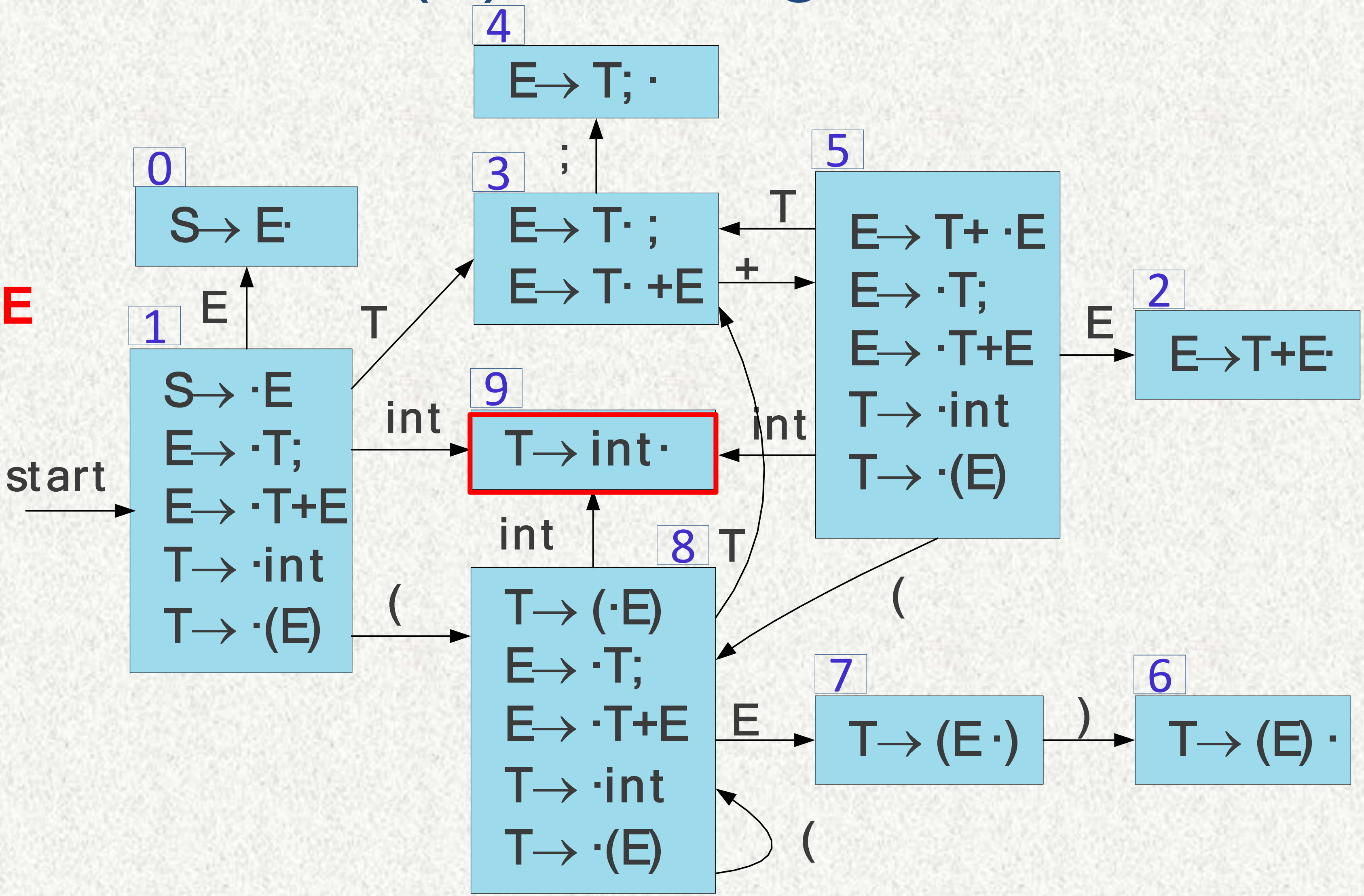


$	T	+	(	
	0	3	5	8

int	+	int	;	)	;	$

LR(0) Parsing

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

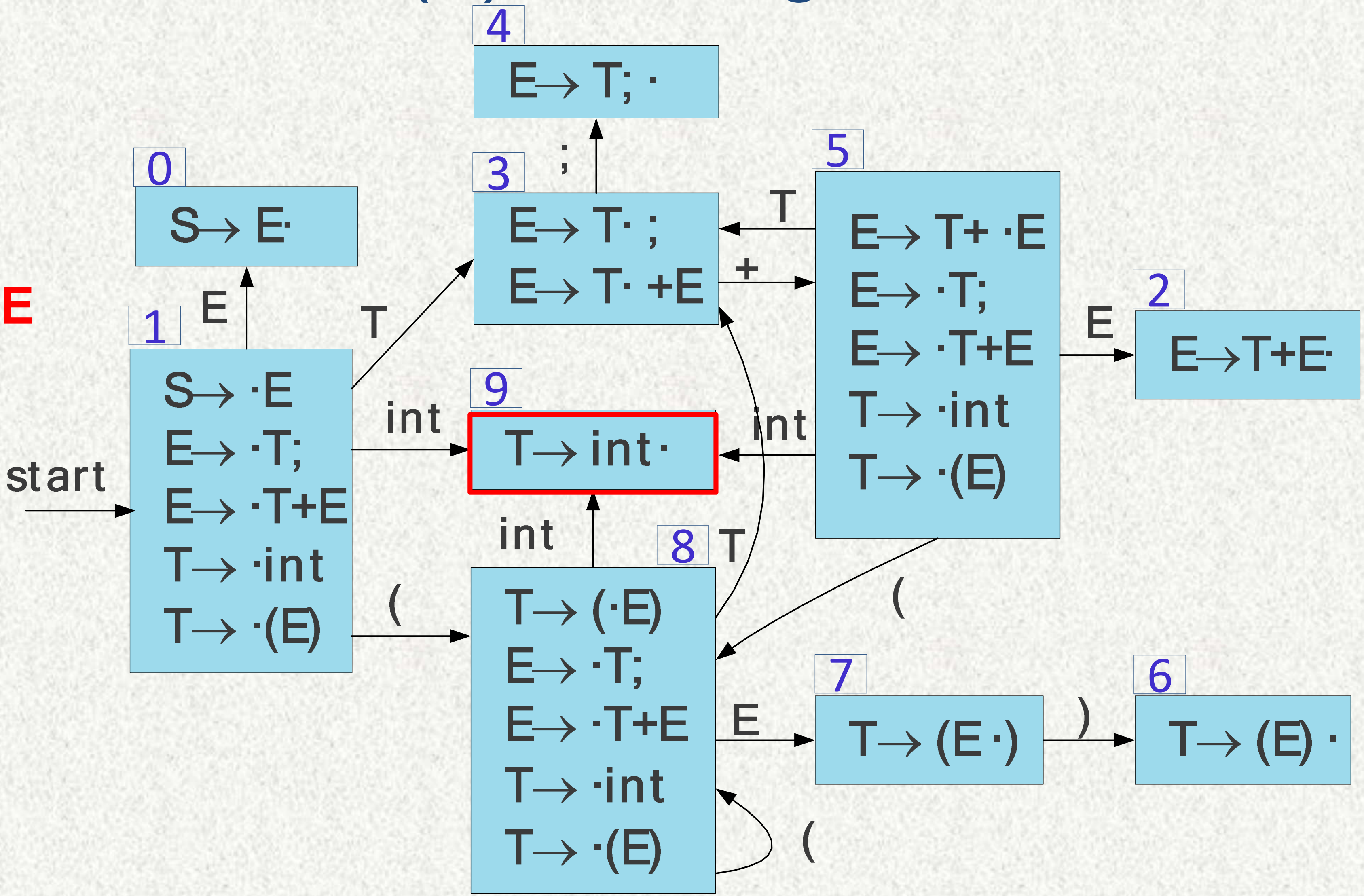


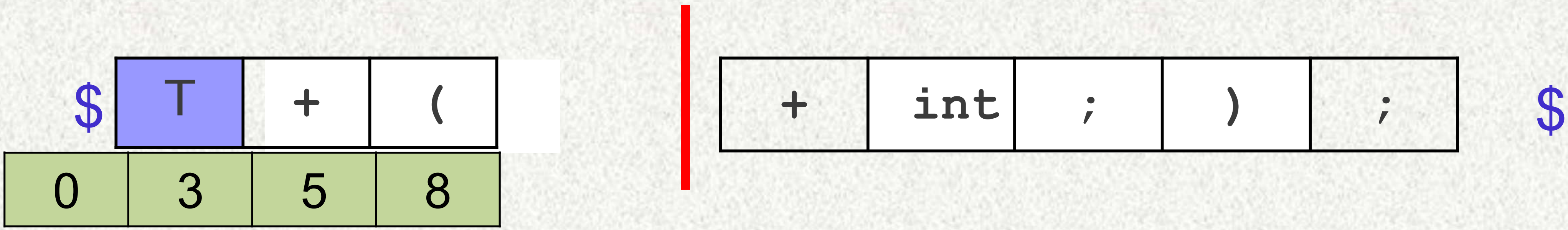
$	T	+	(	int	
	0	3	5	8	9

+	int	;	)	;	$

LR(0) Parsing

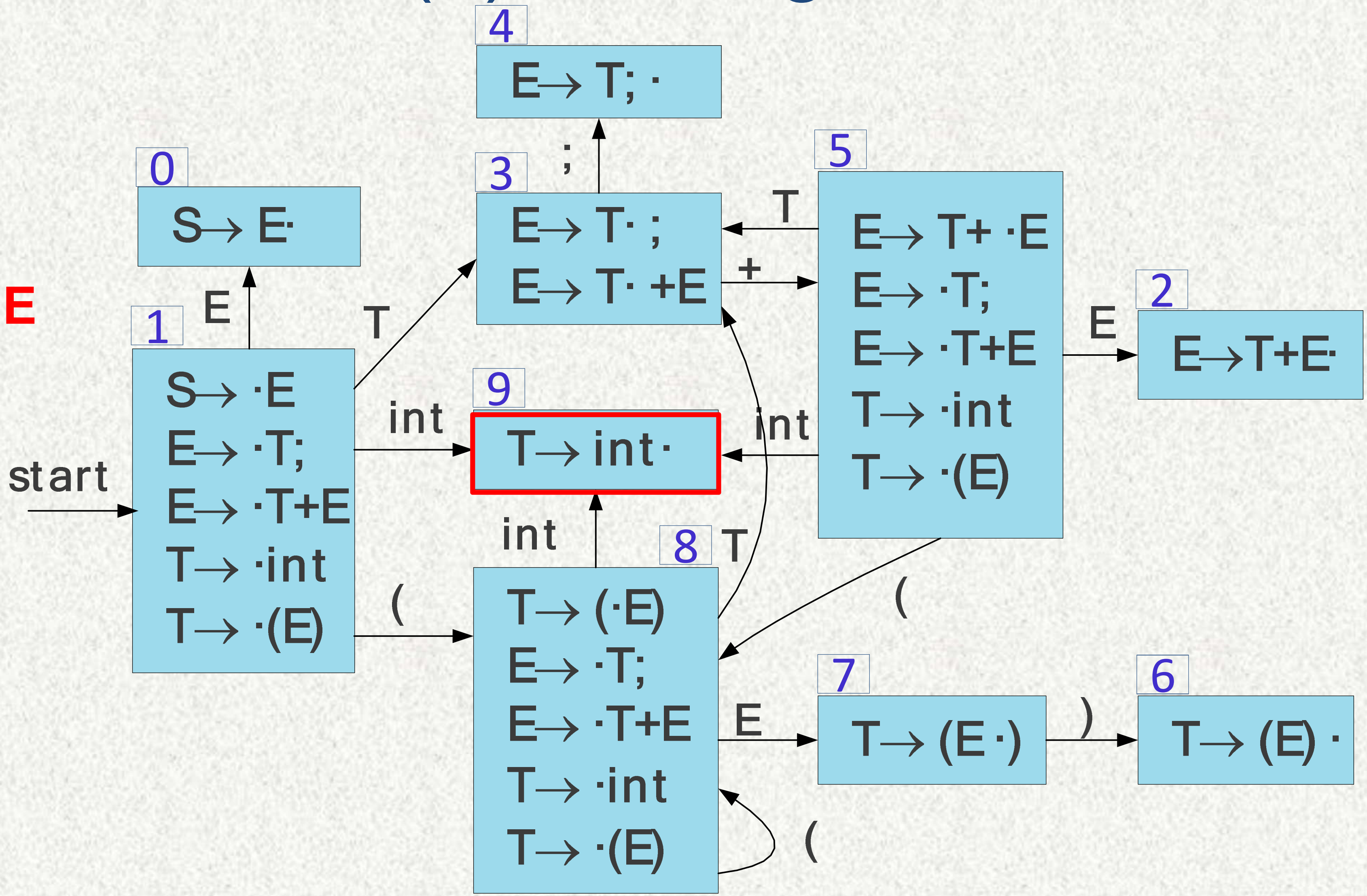
S → E
E → T;
E → T + E
T → int
T → (E)





LR(0) Parsing

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

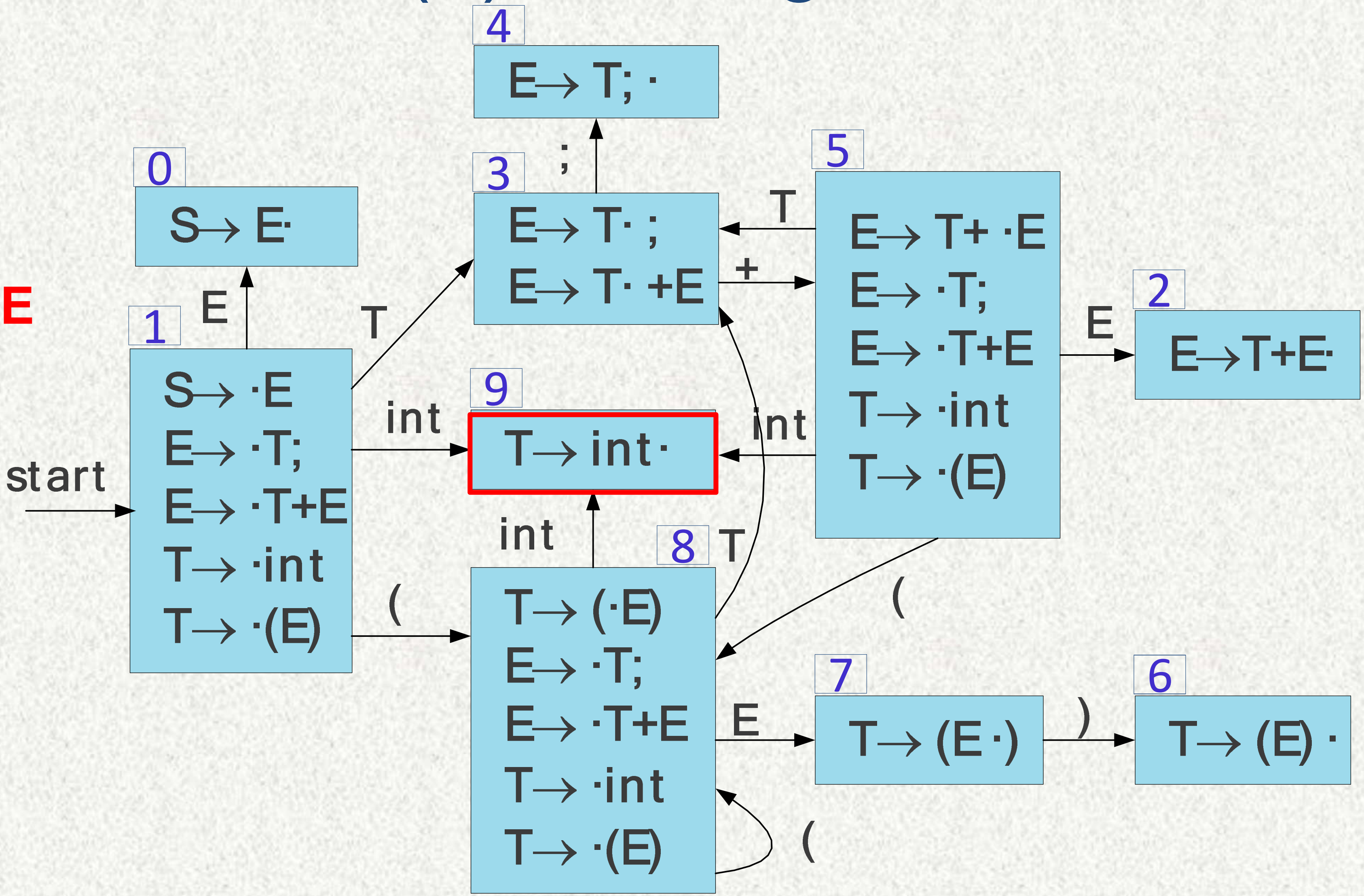


$	T	+	(	T
	0	3	5	8

+	int	;	)	;	$

LR(0) Parsing

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

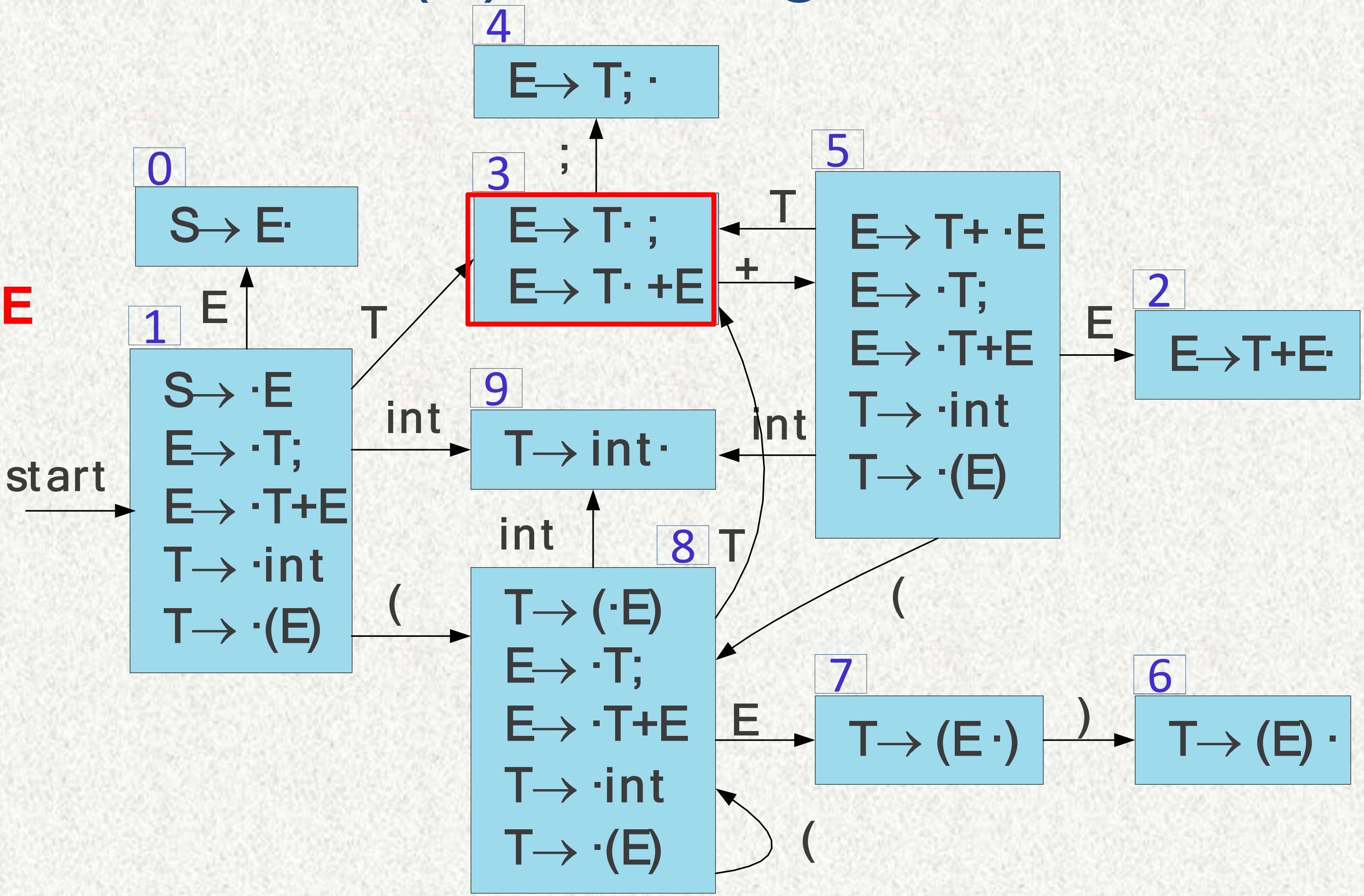


$	T	+	(	T	
	0	3	5	8	3

+	int	;	)	;	$

LR(0) Parsing

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

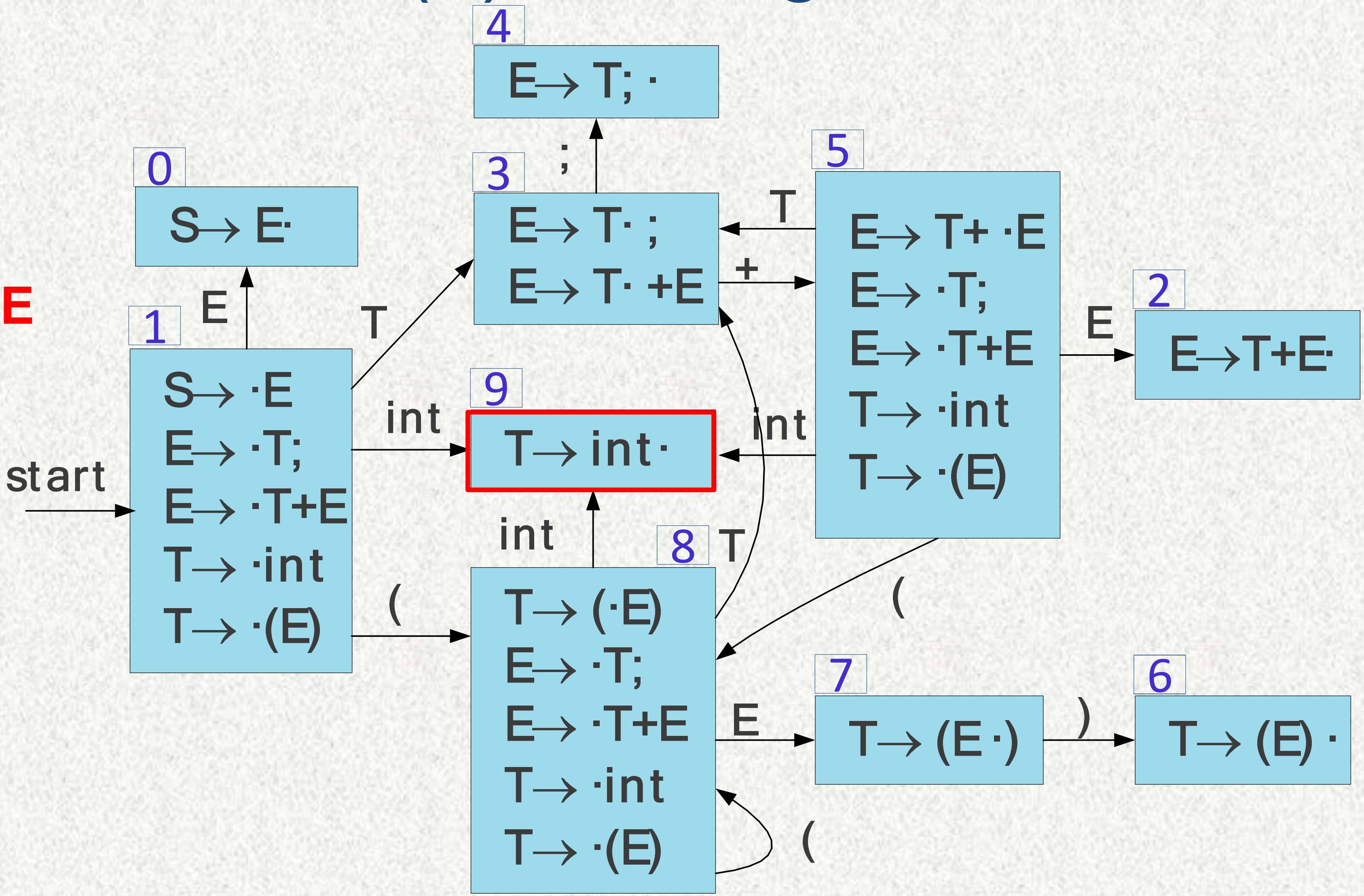


$	T	+	(	T	+
0	3	5	8	3	5

int	;	)	;	$

LR(0) Parsing

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

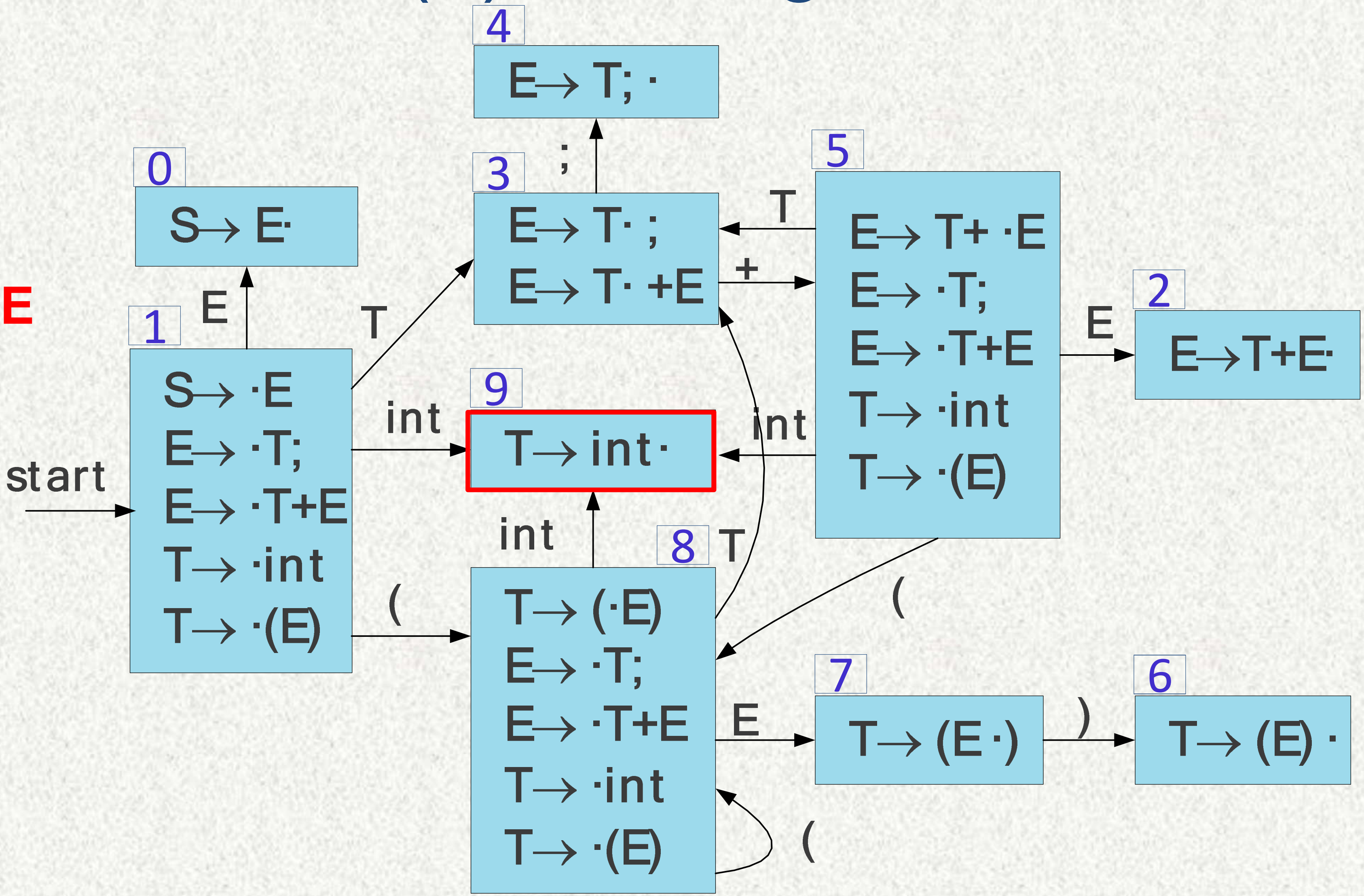


$	T	+	(	T	+	int
0	3	5	8	3	5	9

;	)	;	$

LR(0) Parsing

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

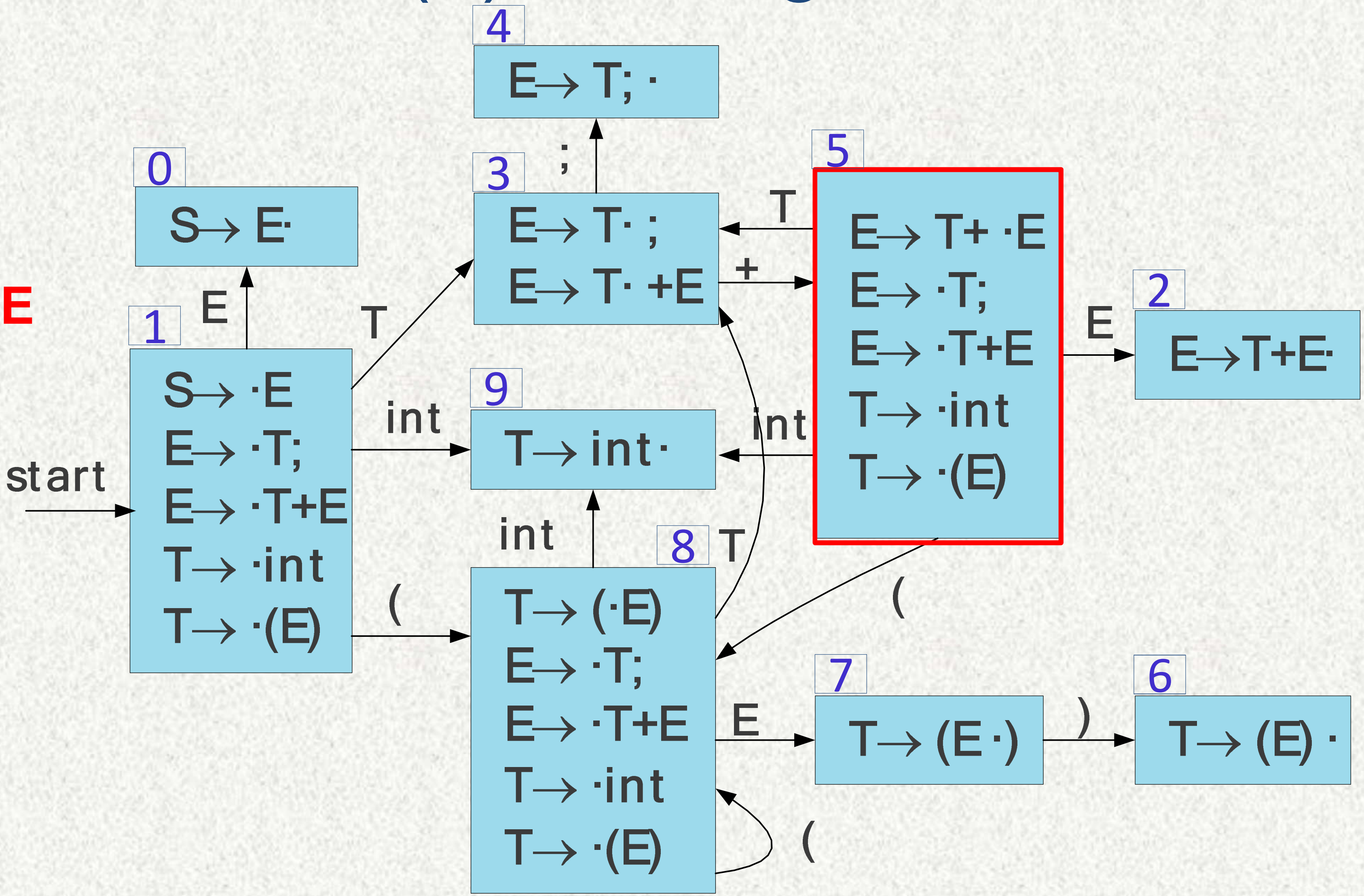


$	T	+	(	T	+
0	3	5	8	3	5

;	)	;	$

LR(0) Parsing

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

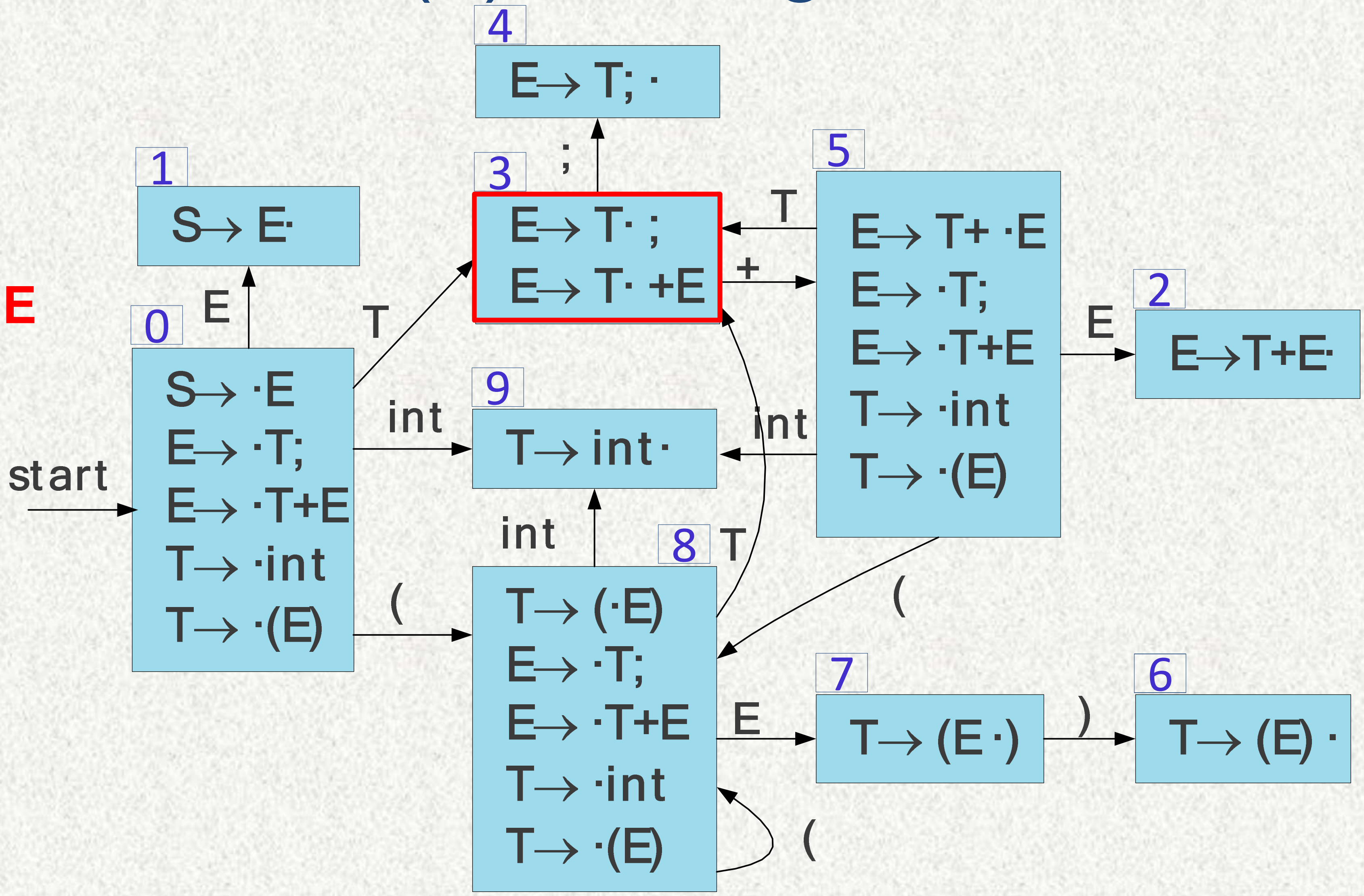


$	T	+	(	T	+	T
0	3	5	8	3	5	

;	)	;	$

LR(0) Parsing

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

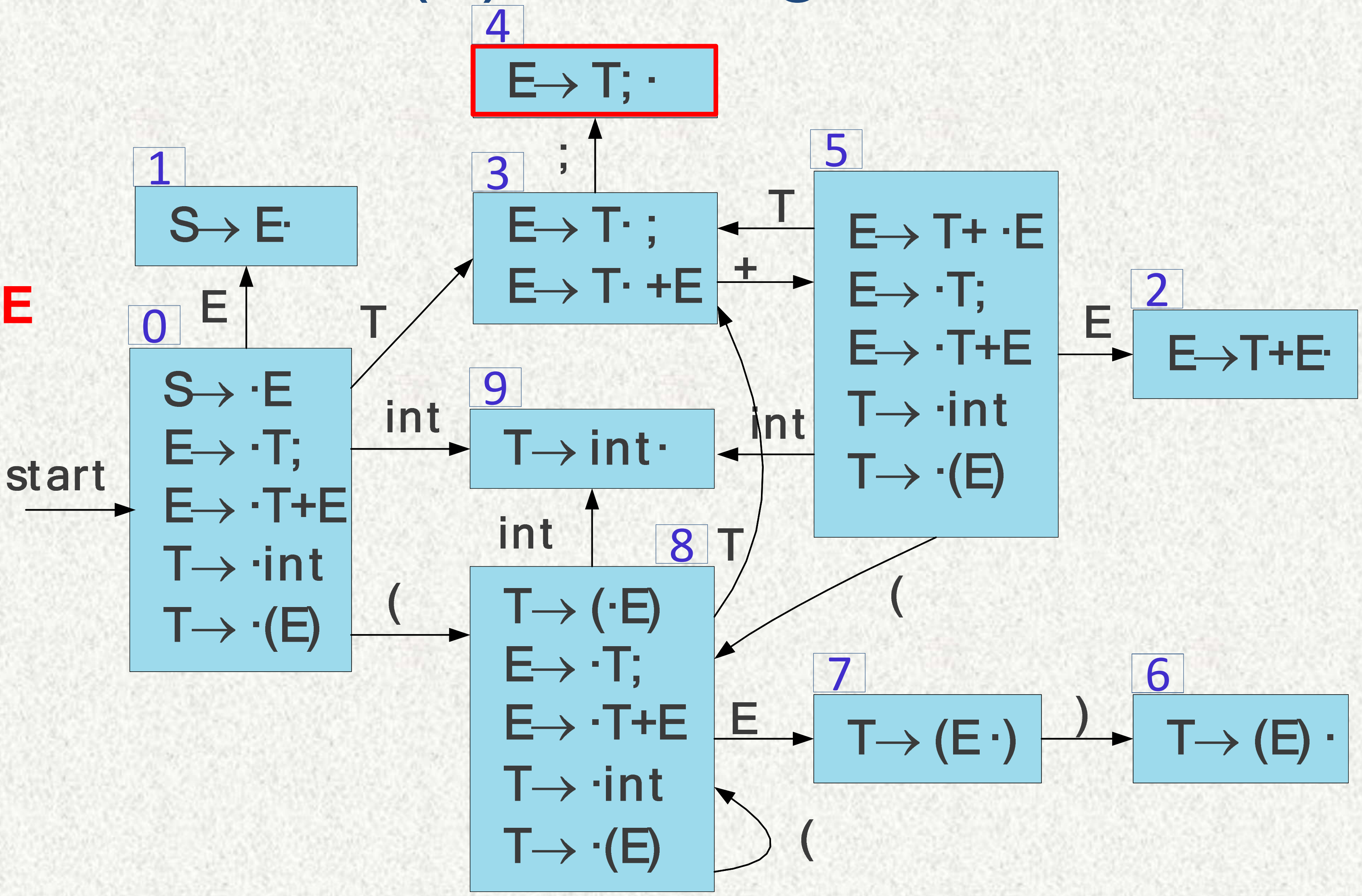


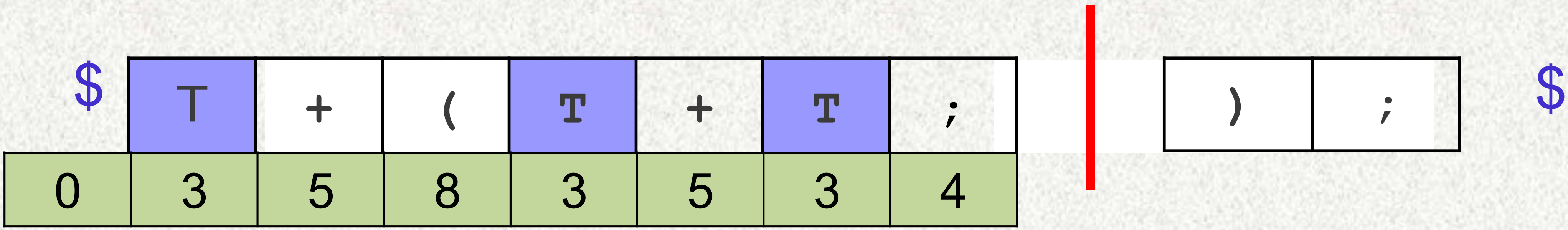
$	T	+	(	T	+	T
0	3	5	8	3	5	3

;	)	;	$

LR(0) Parsing

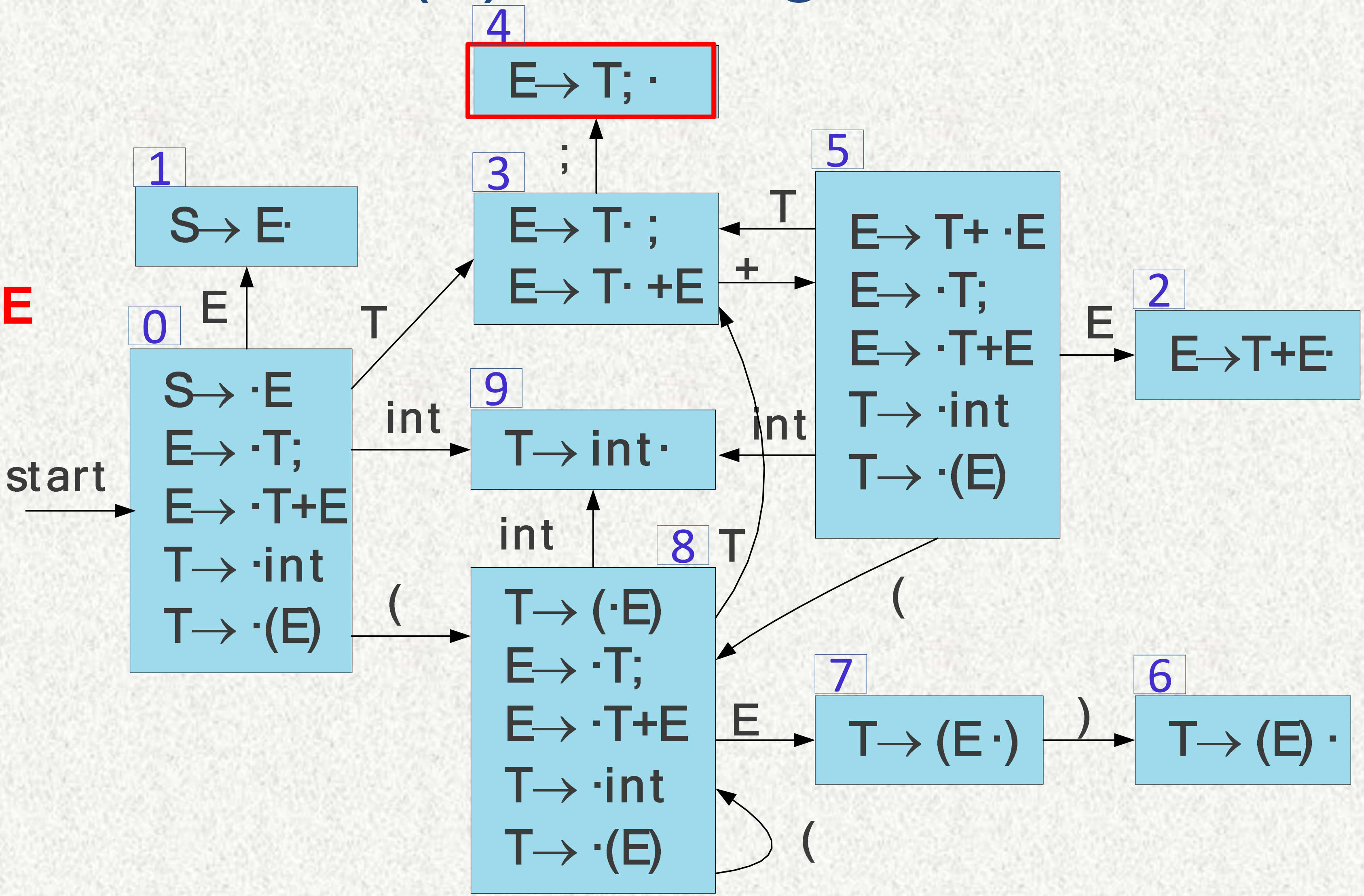
S → E
E → T;
E → T + E
T → int
T → (E)





LR(0) Parsing

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

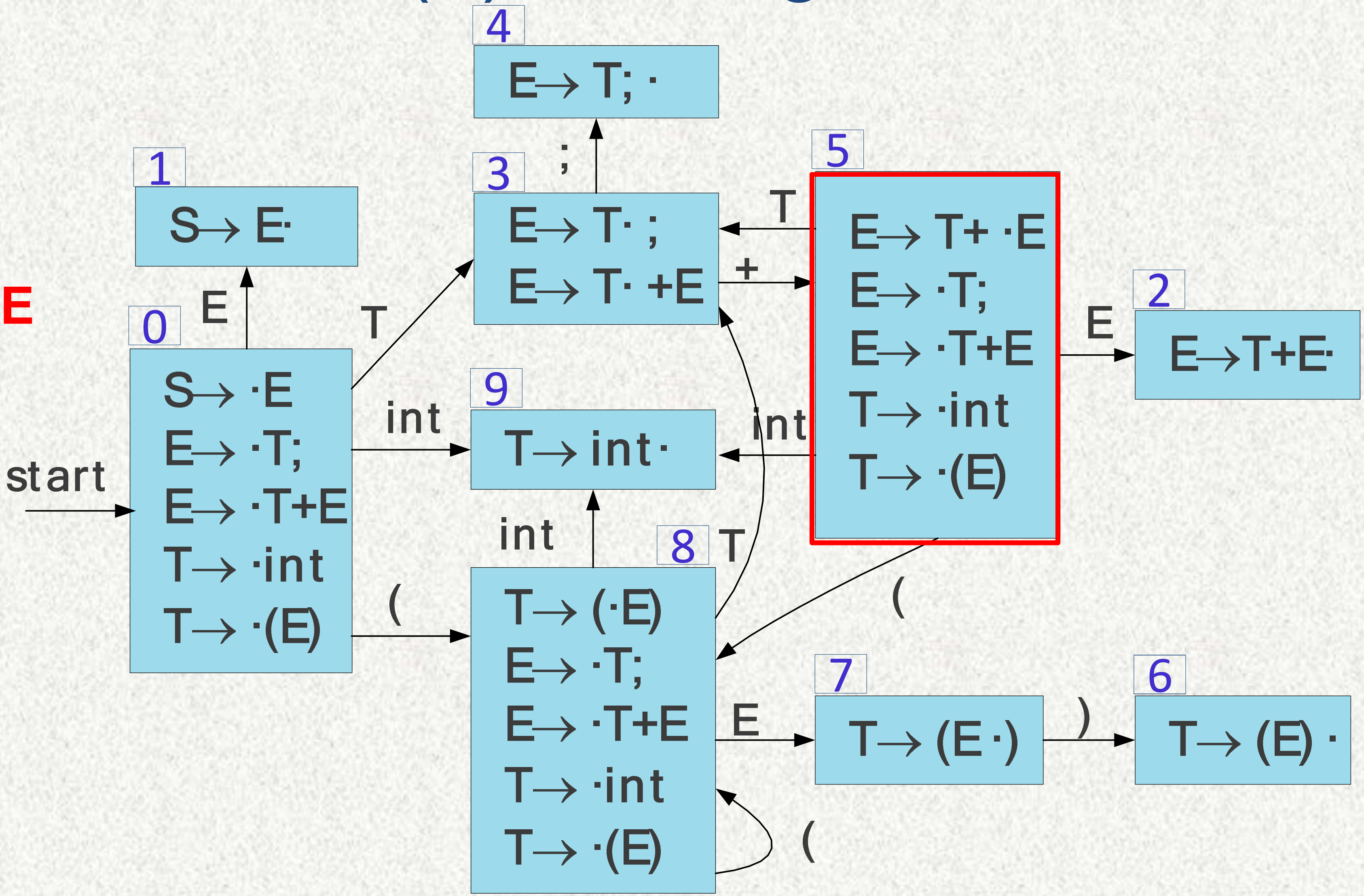


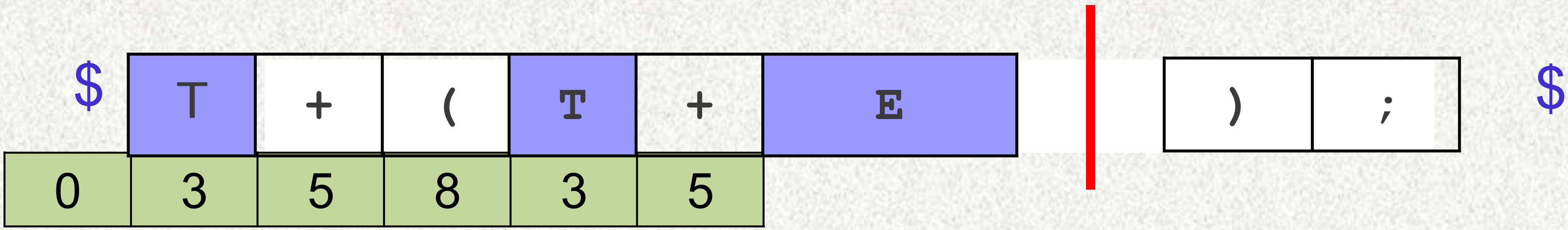
$	T	+	(	T	+	
0	3	5	8	3	5	

)	;	$

LR(0) Parsing

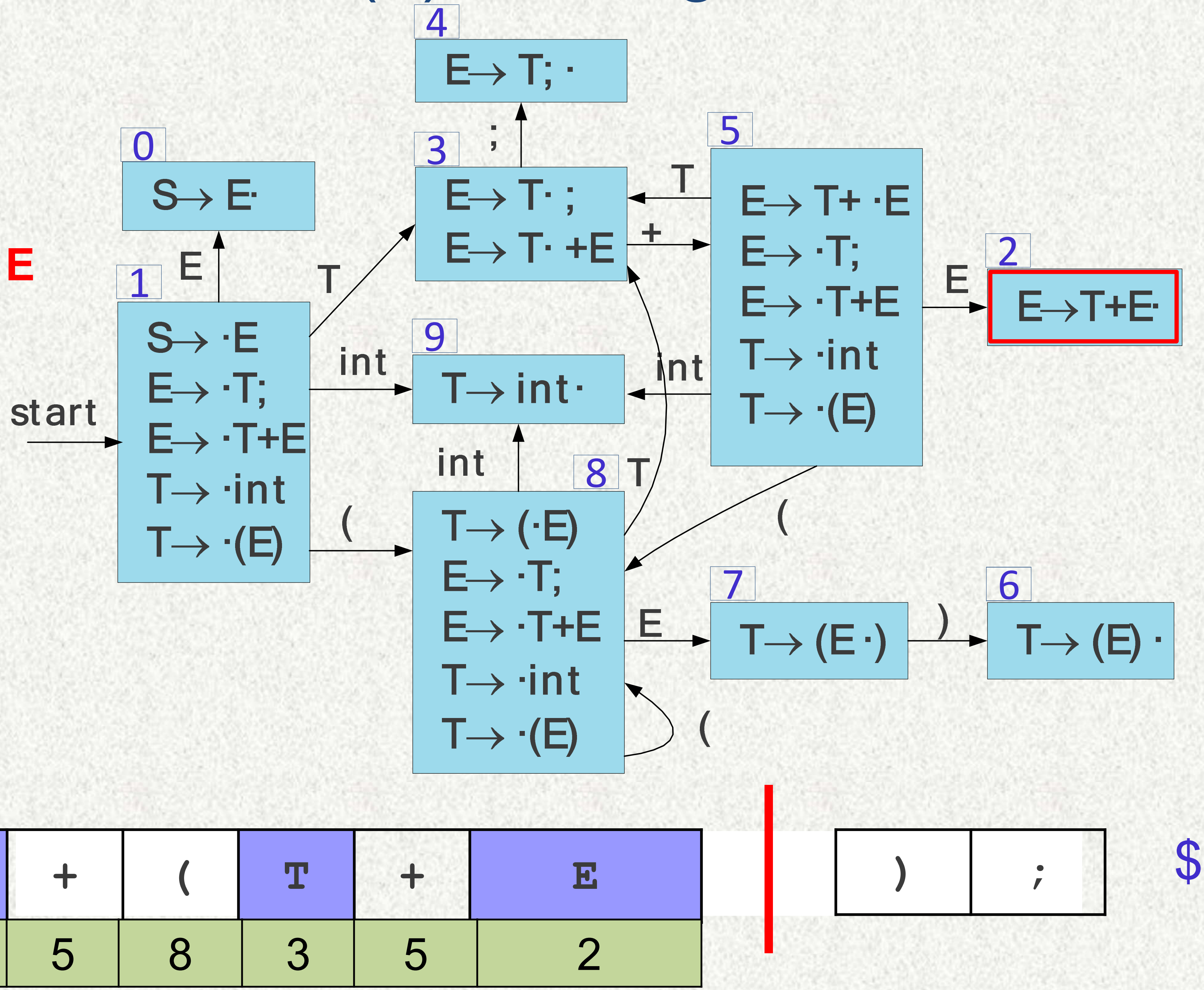
S → E
E → T;
E → T + E
T → int
T → (E)





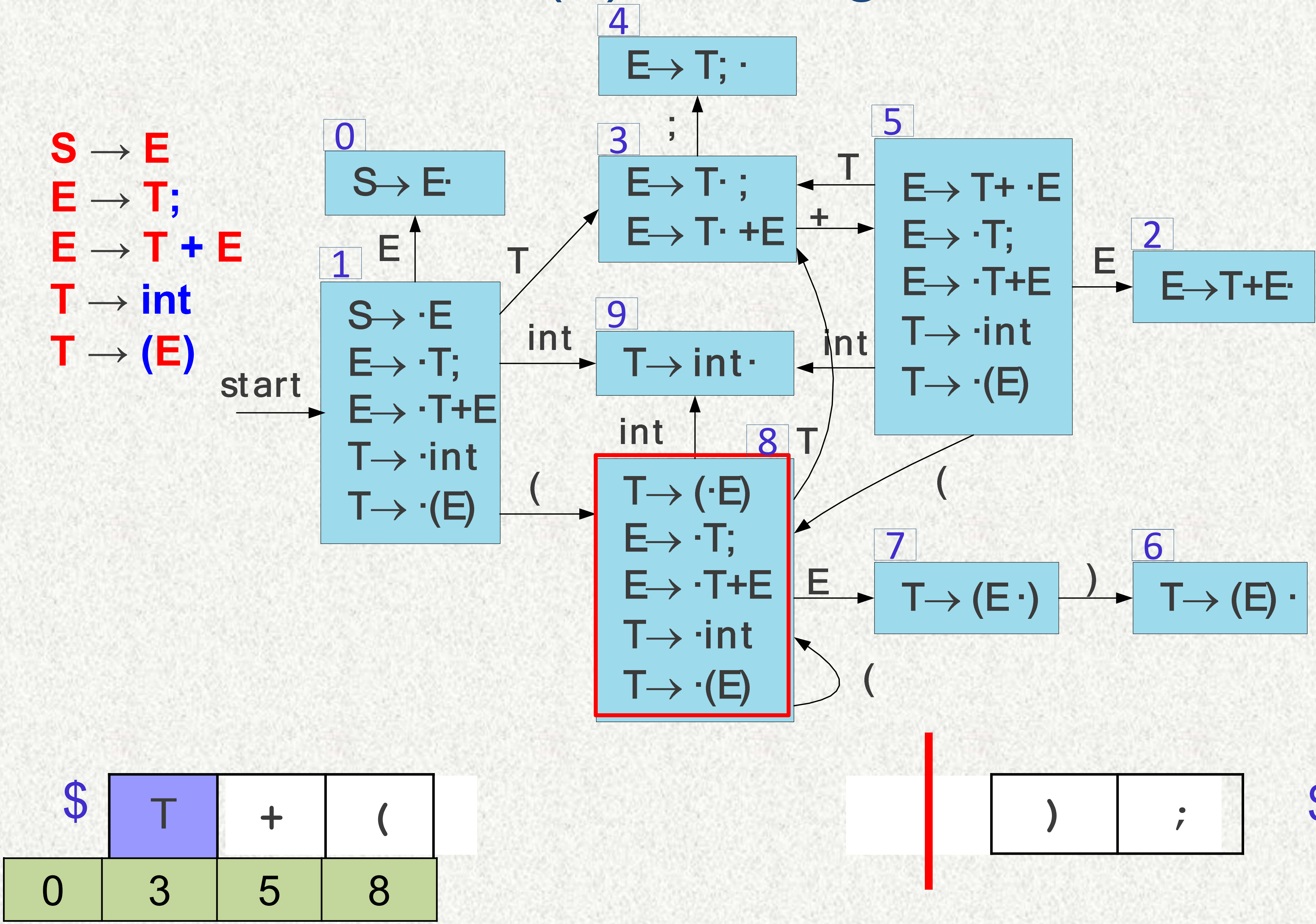
LR(0) Parsing

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



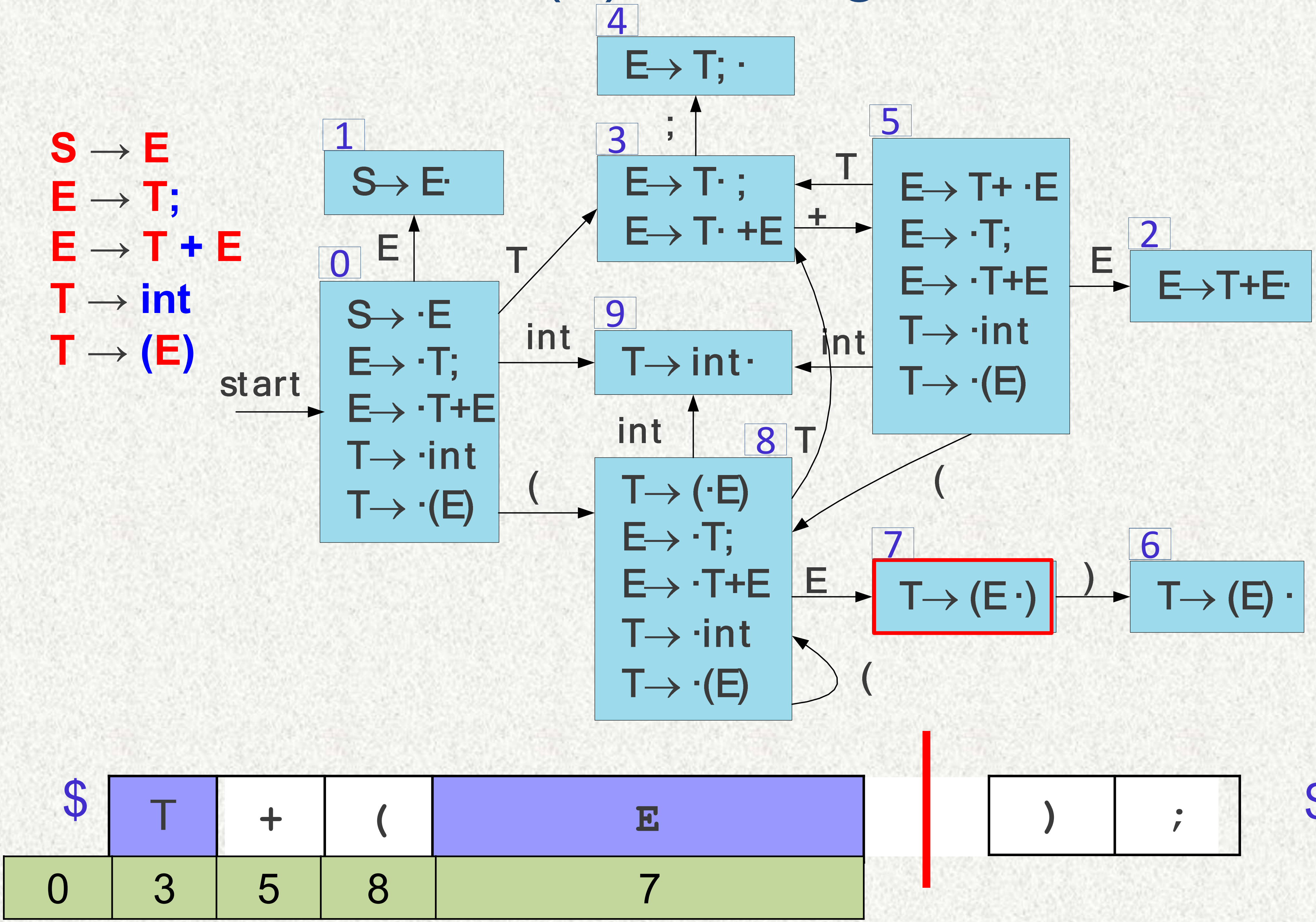
LR(0) Parsing

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



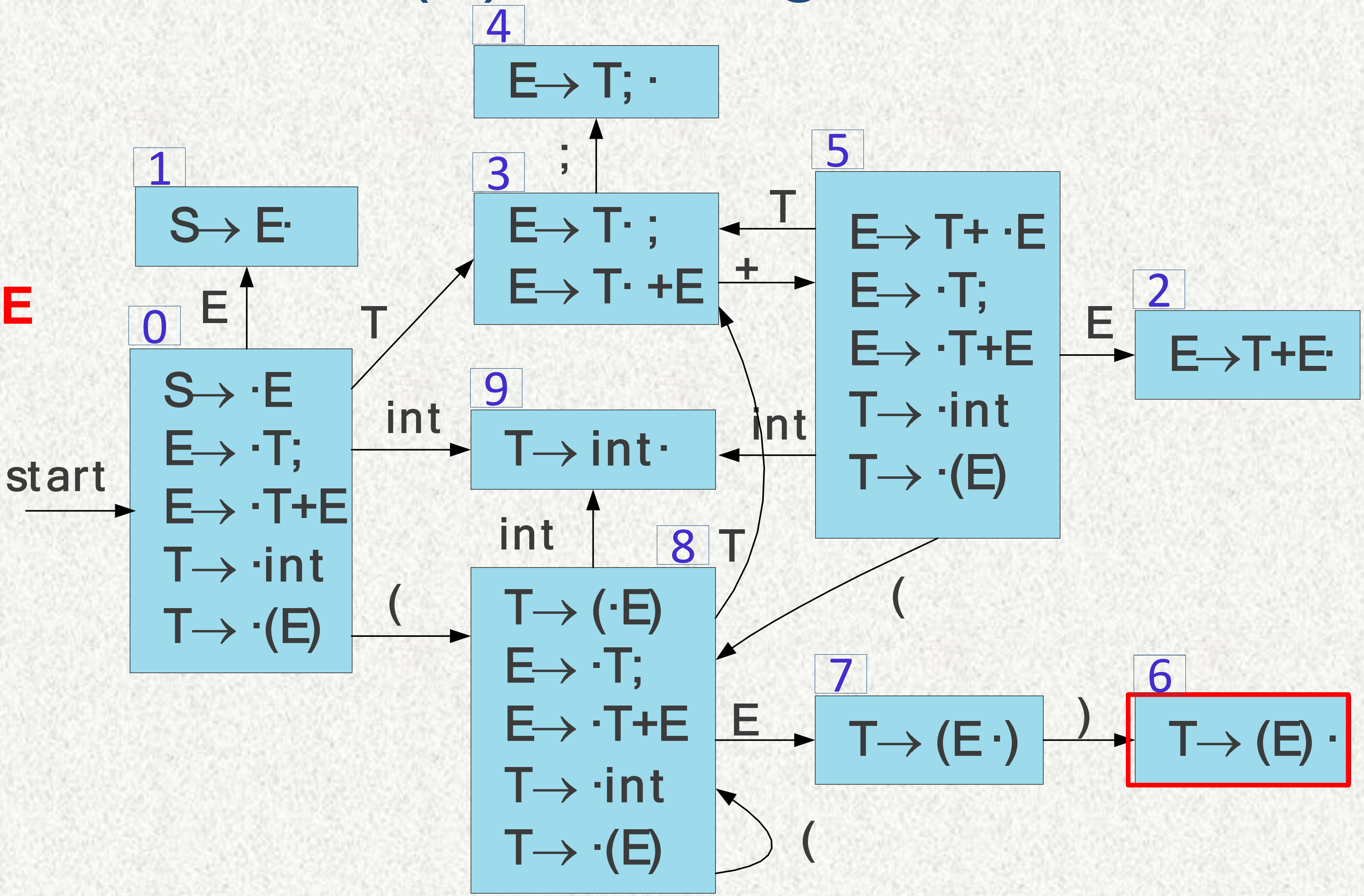
LR(0) Parsing

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



LR(0) Parsing

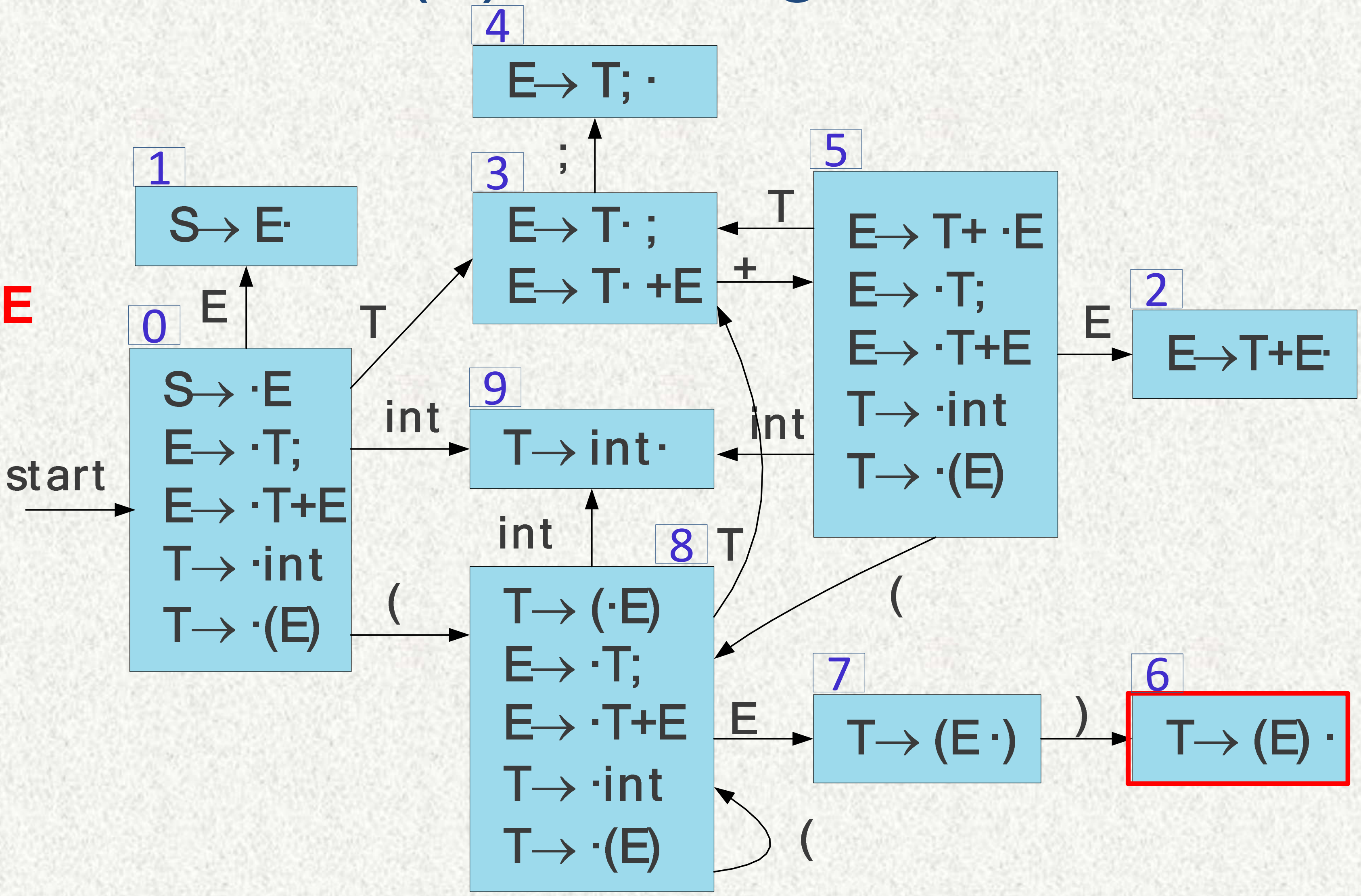
S → **E**
E → **T;**
E → **T + E**
T → **int**
T → **(E)**



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

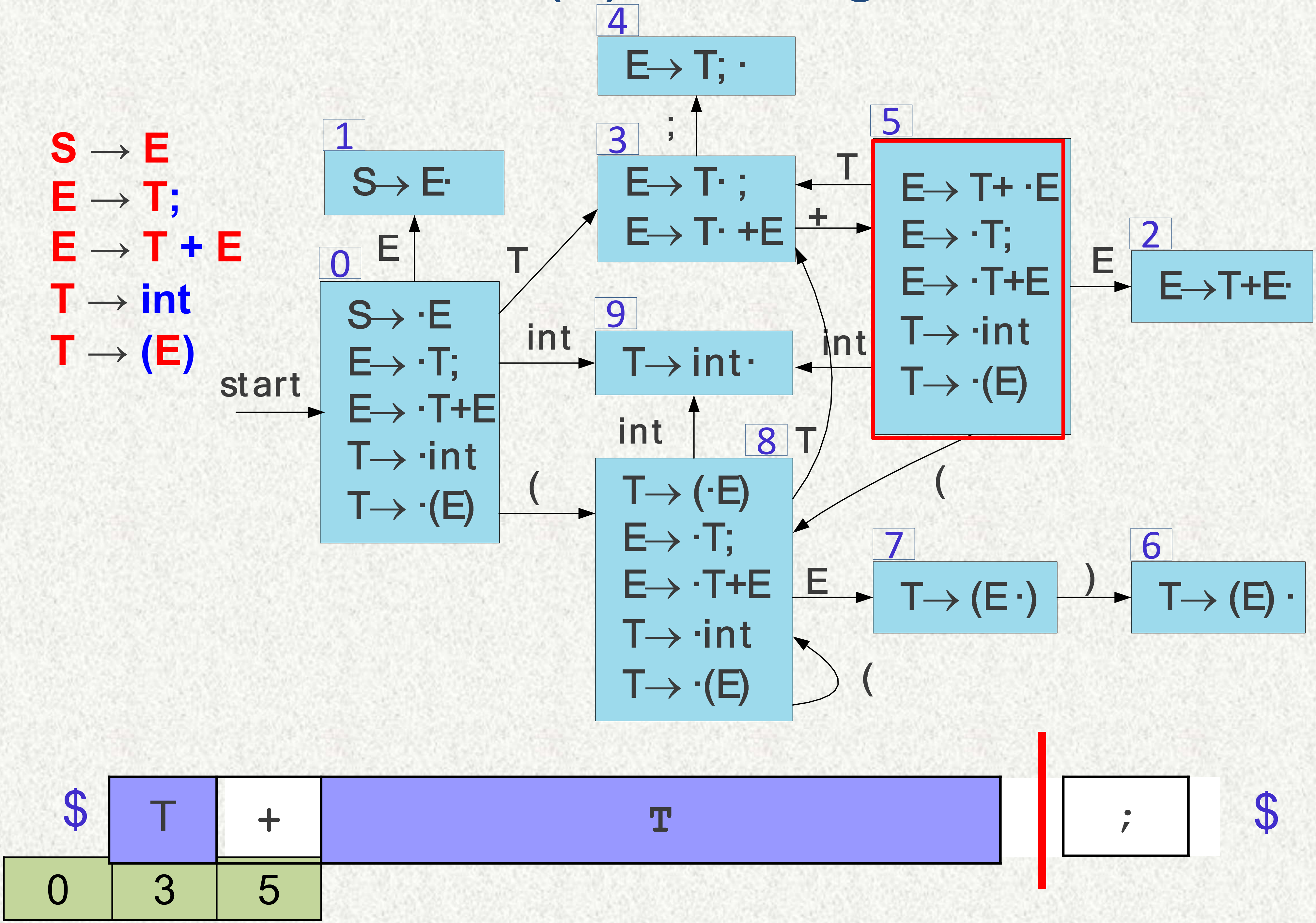
LR(0) Parsing

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



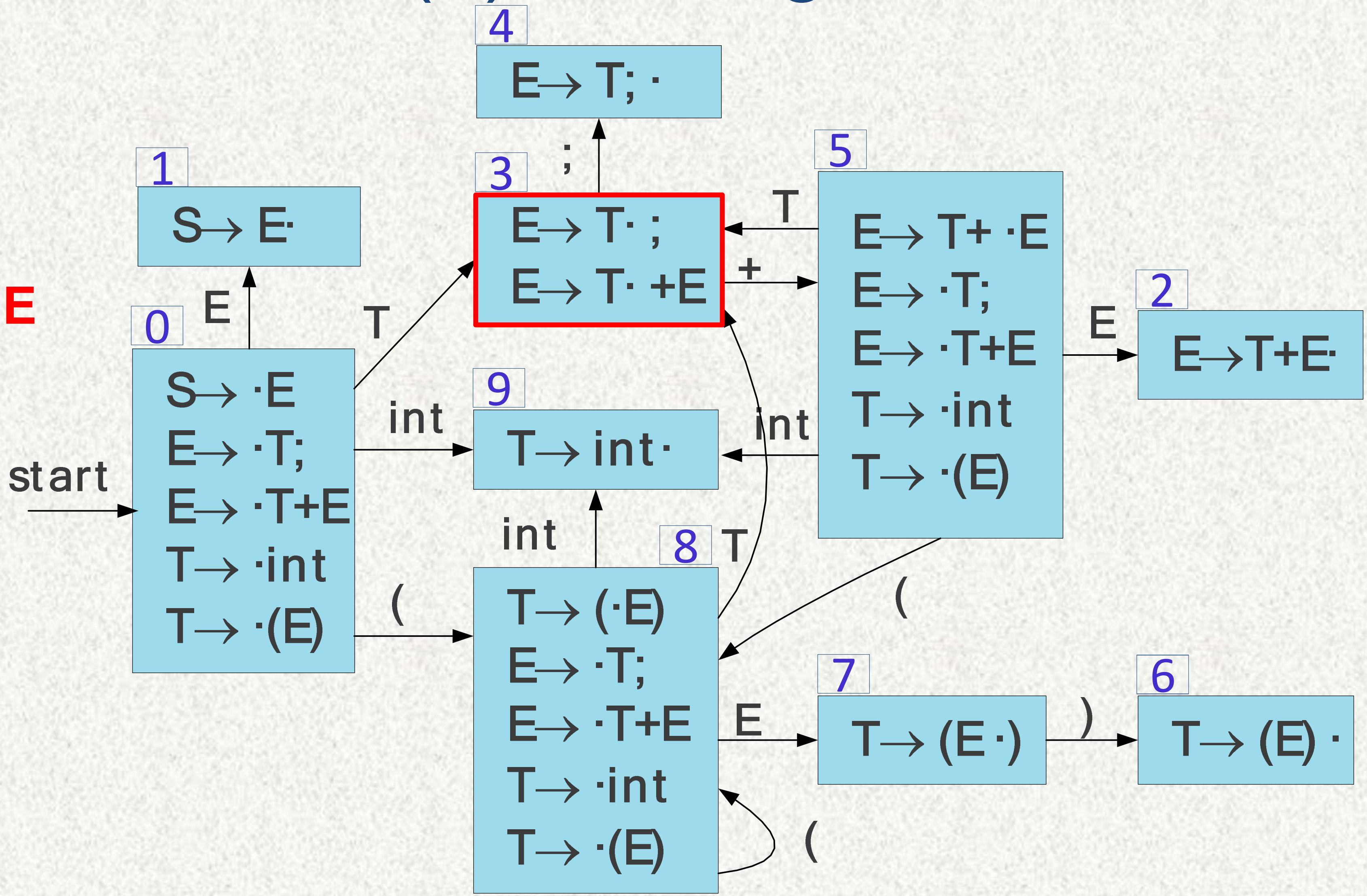
LR(0) Parsing

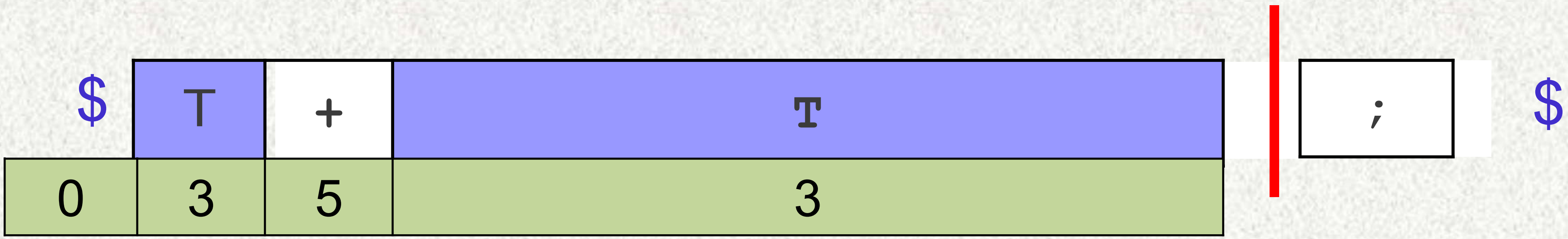
S → E
E → T;
E → T + E
T → int
T → (E)



LR(0) Parsing

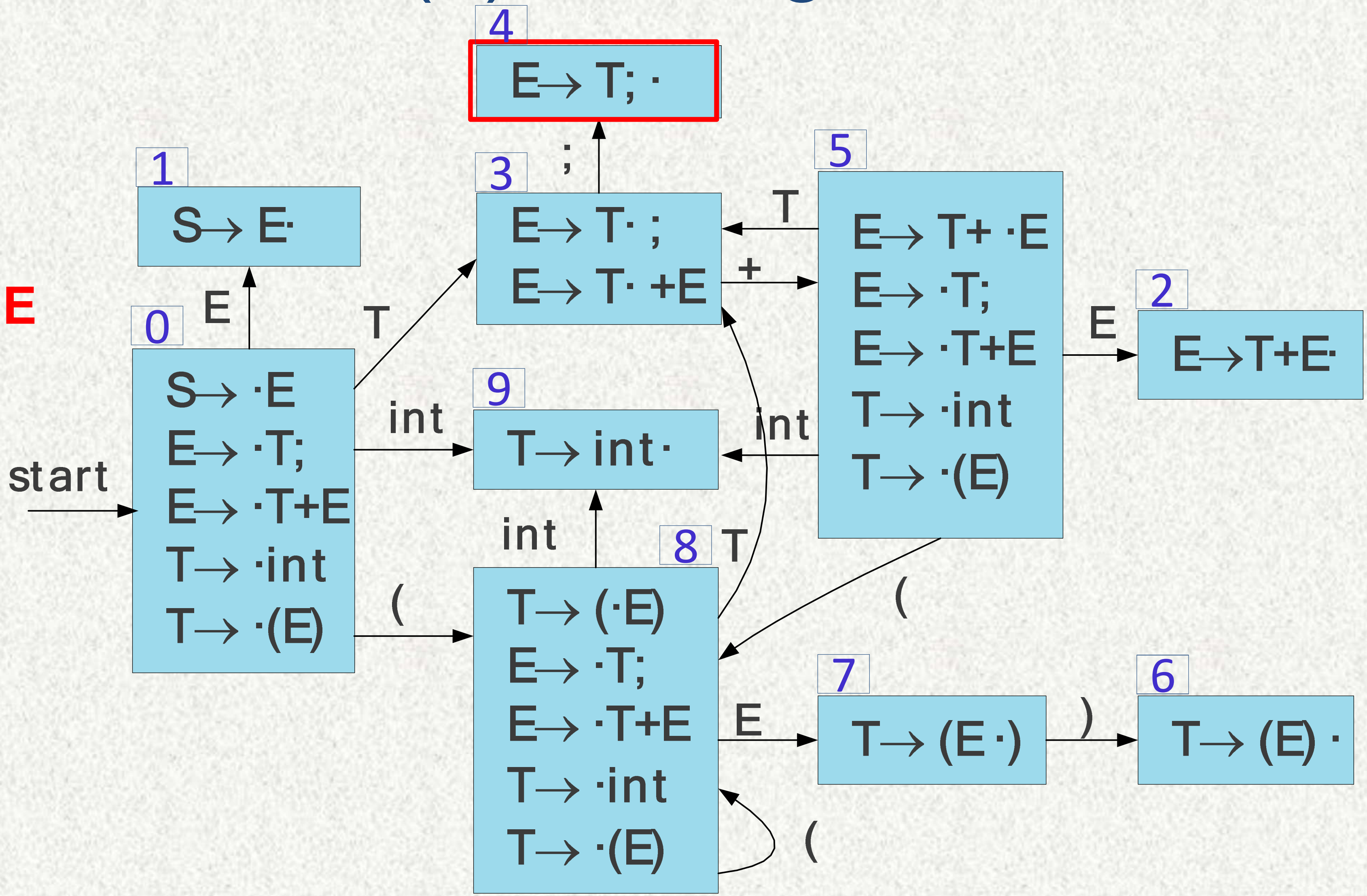
S → E
E → T;
E → T + E
T → int
T → (E)





LR(0) Parsing

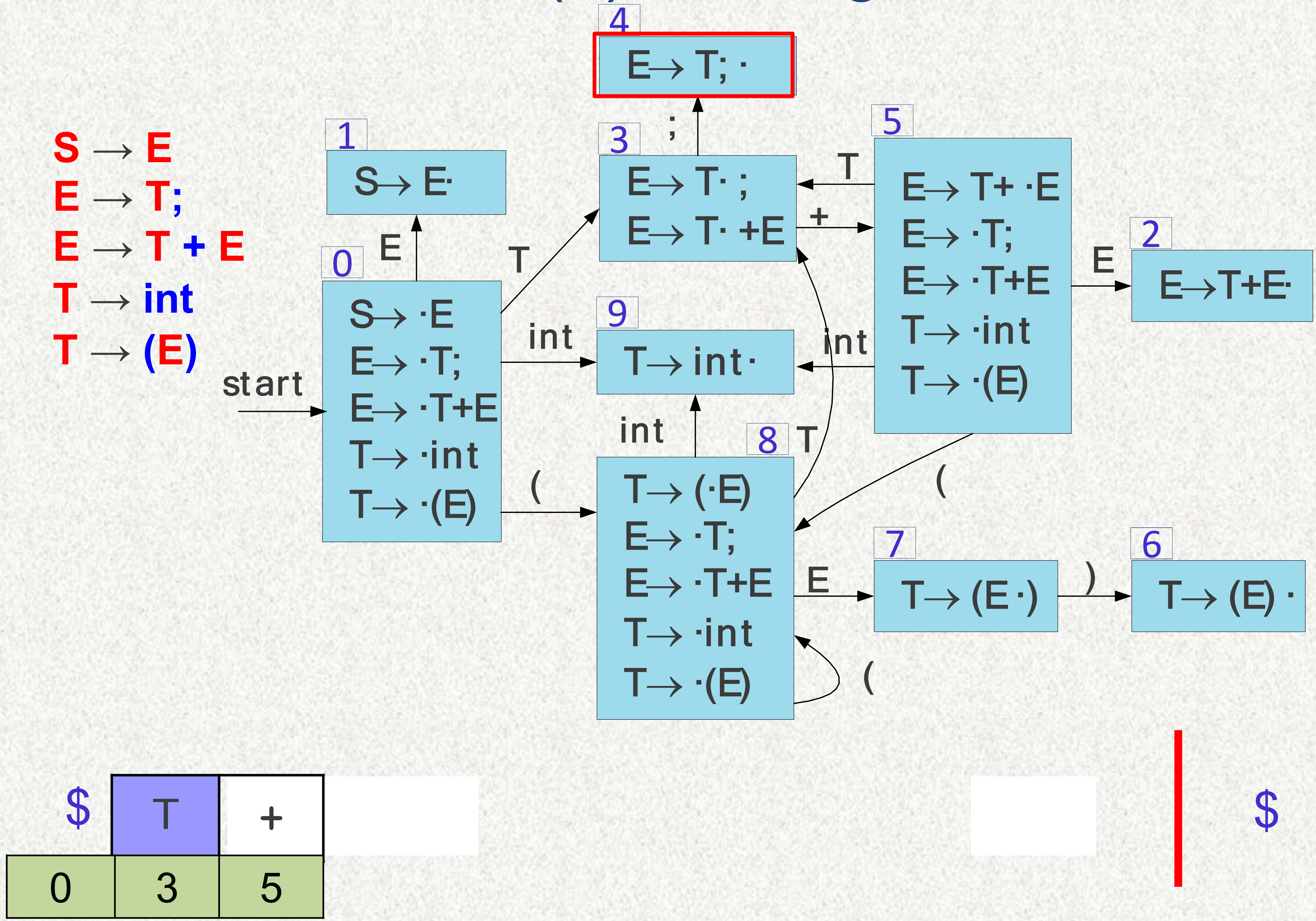
S → E
E → T;
E → T + E
T → int
T → (E)



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

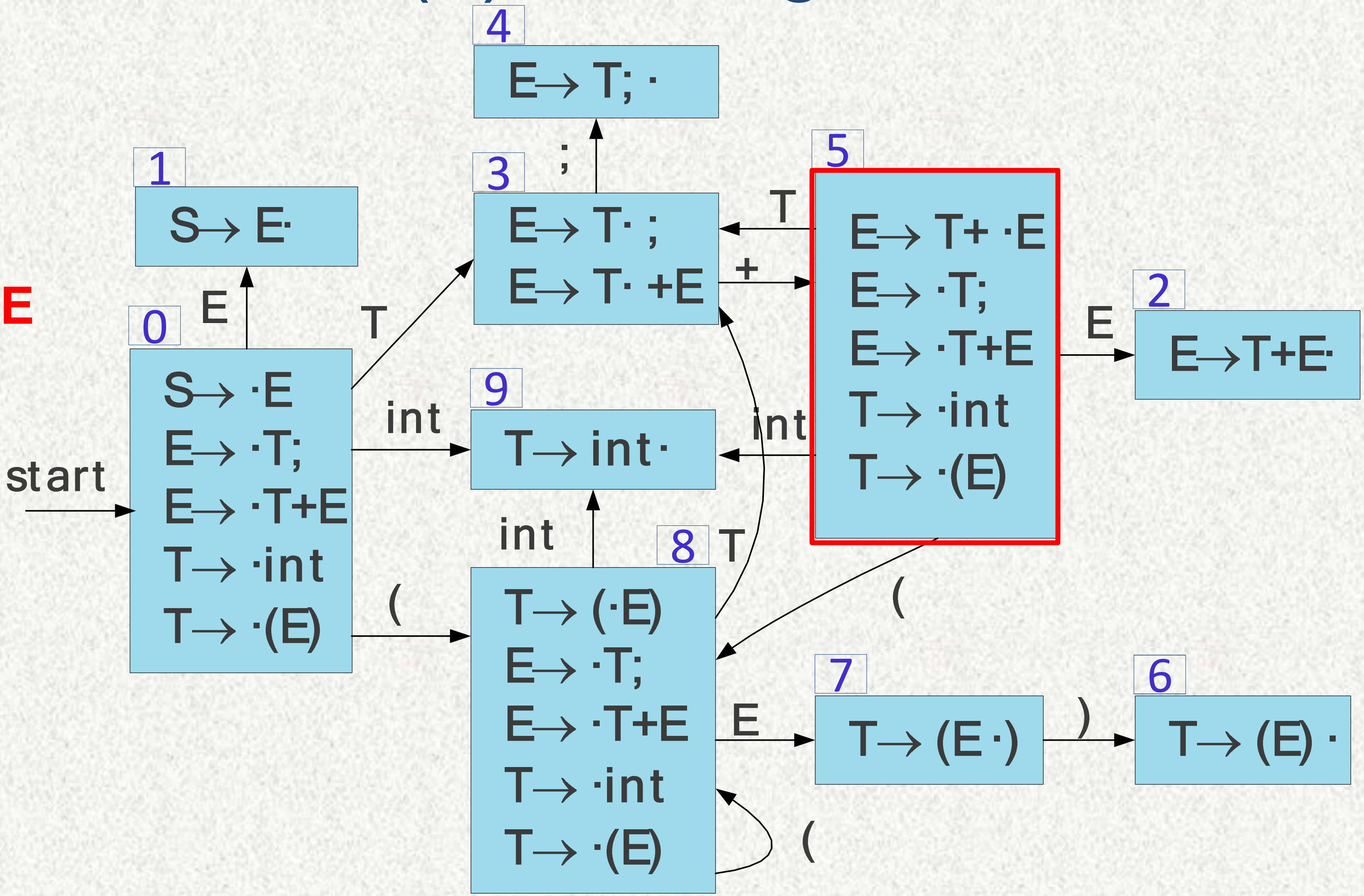
LR(0) Parsing

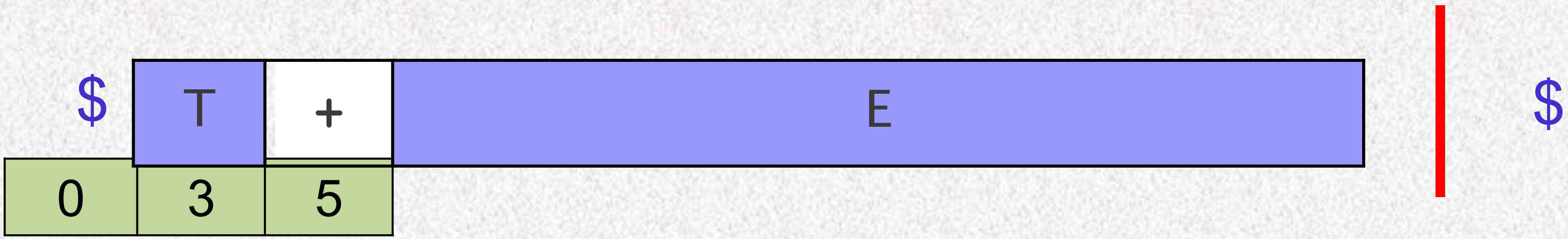
S → E
E → T;
E → T + E
T → int
T → (E)



LR(0) Parsing

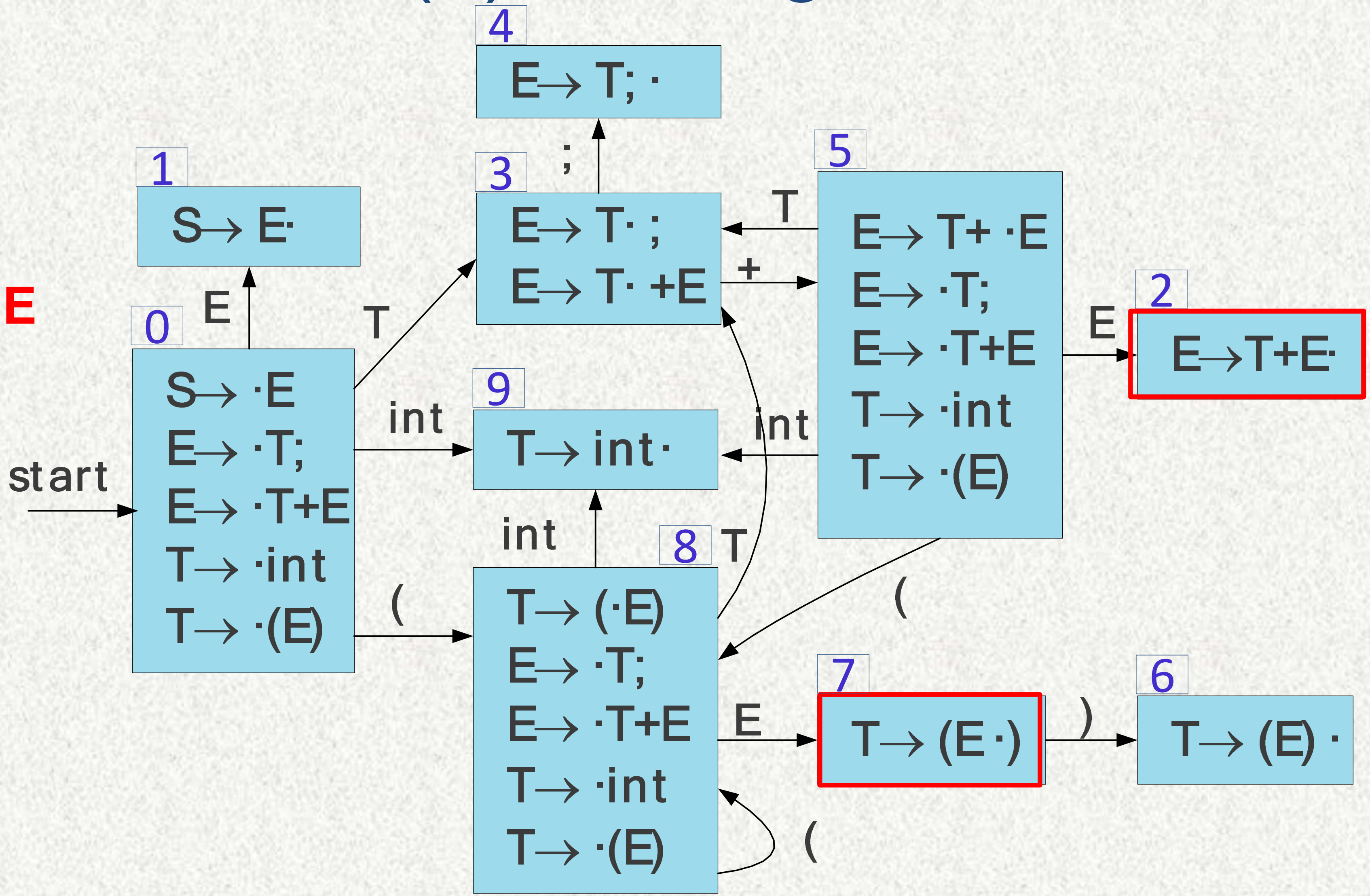
S → E
E → T;
E → T + E
T → int
T → (E)





LR(0) Parsing

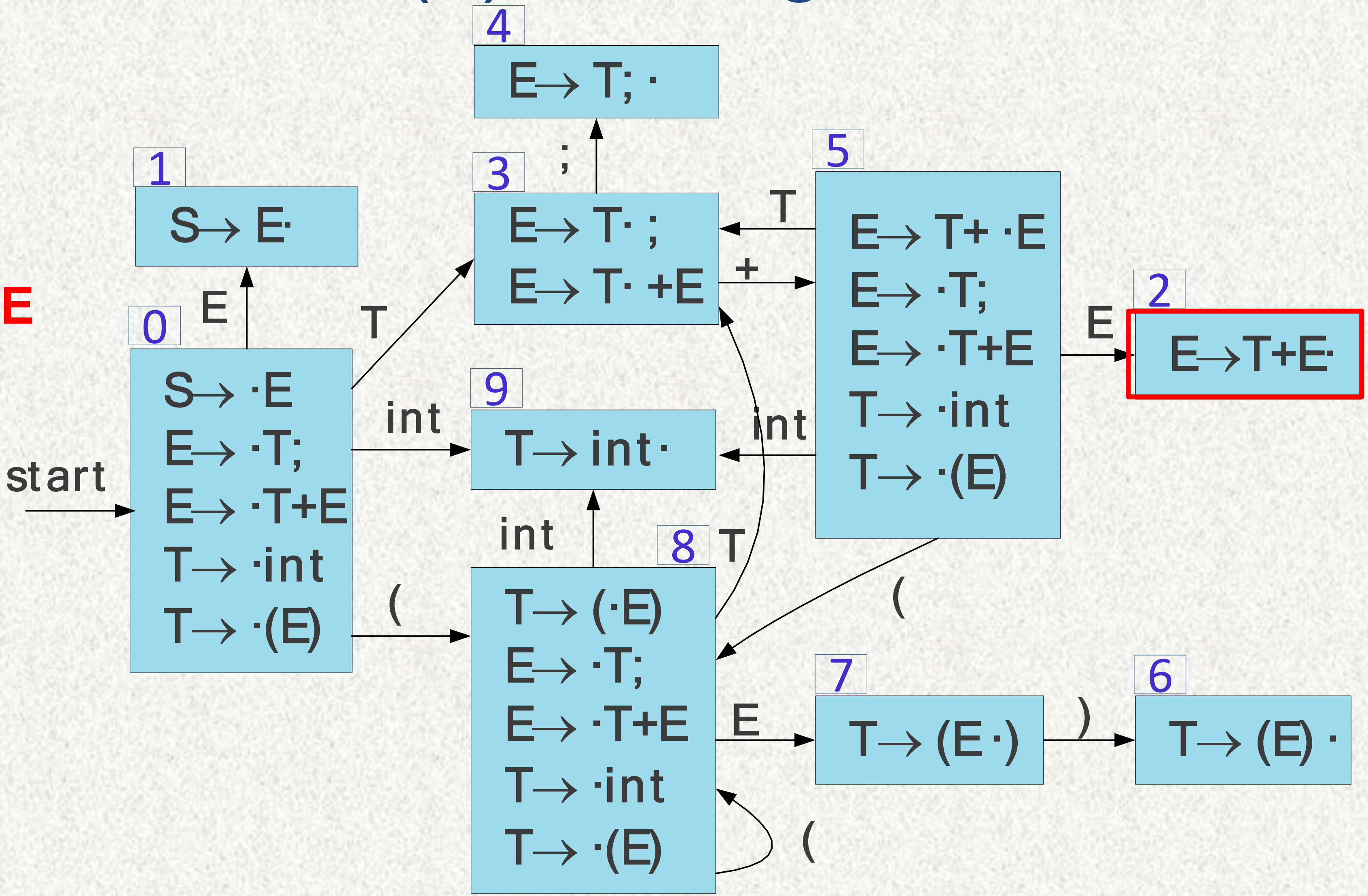
S → E
E → T;
E → T + E
T → int
T → (E)



$	T	+	E	$
0	3	5	2	

LR(0) Parsing

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



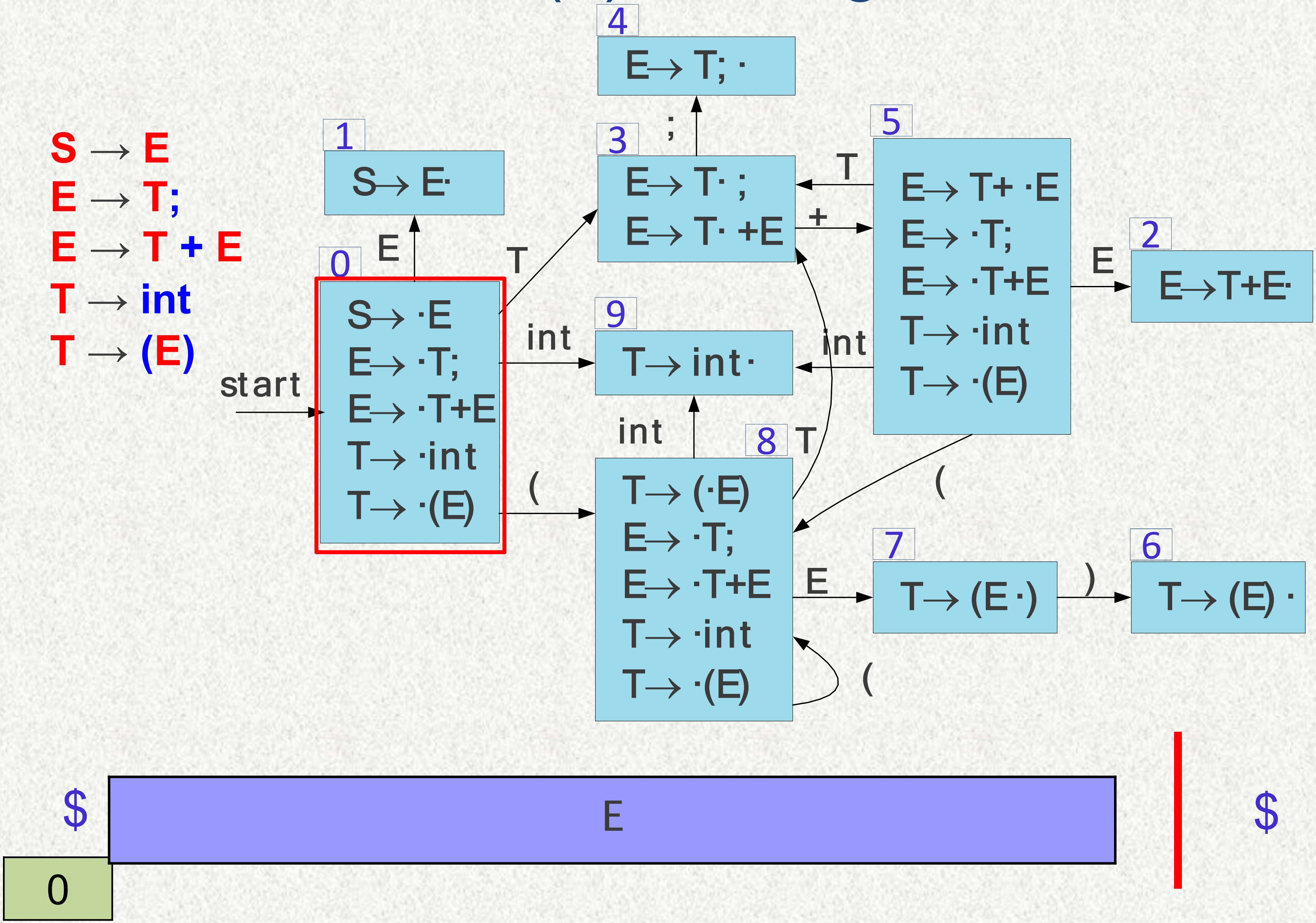
$

0

$

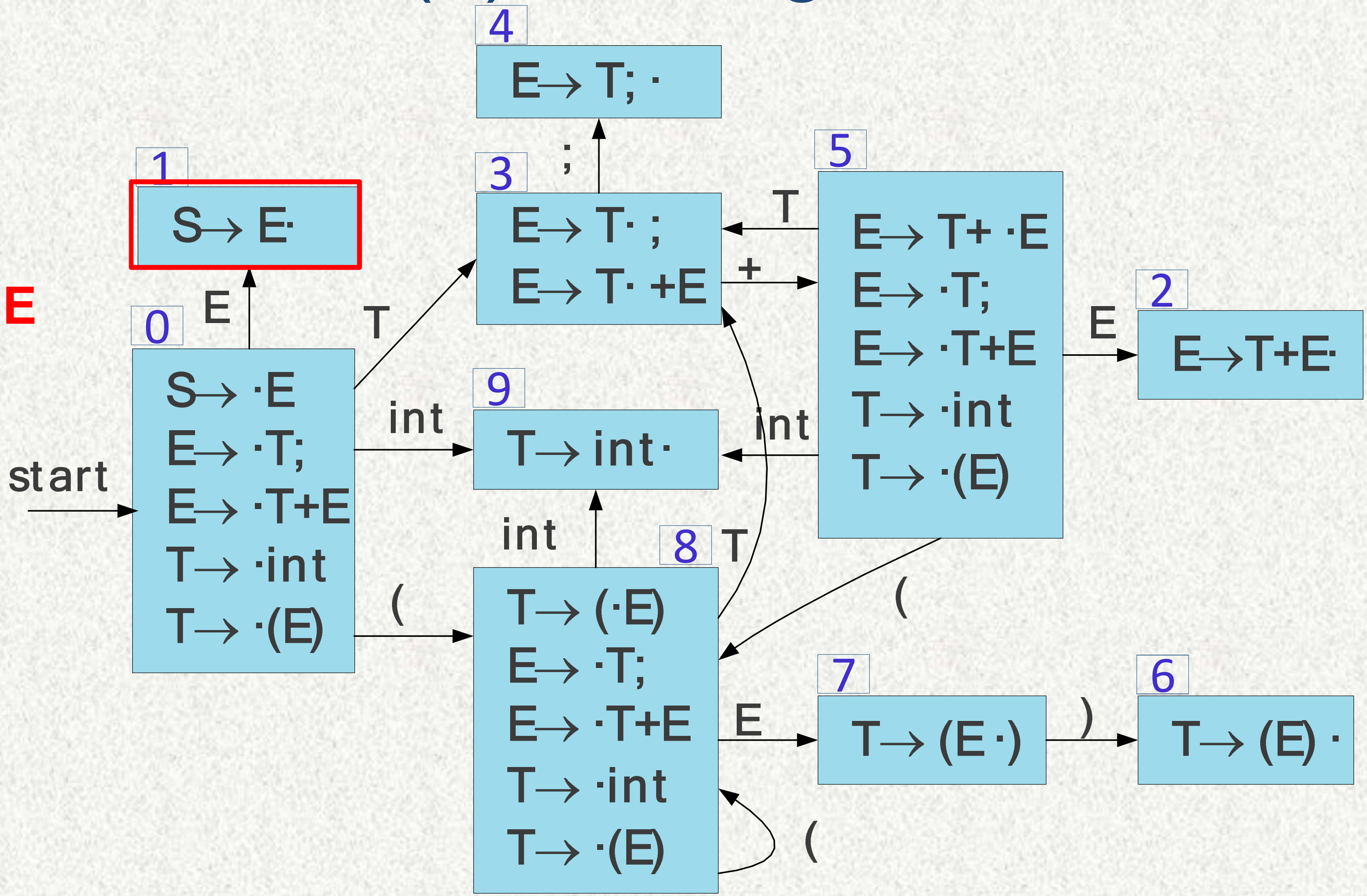
LR(0) Parsing

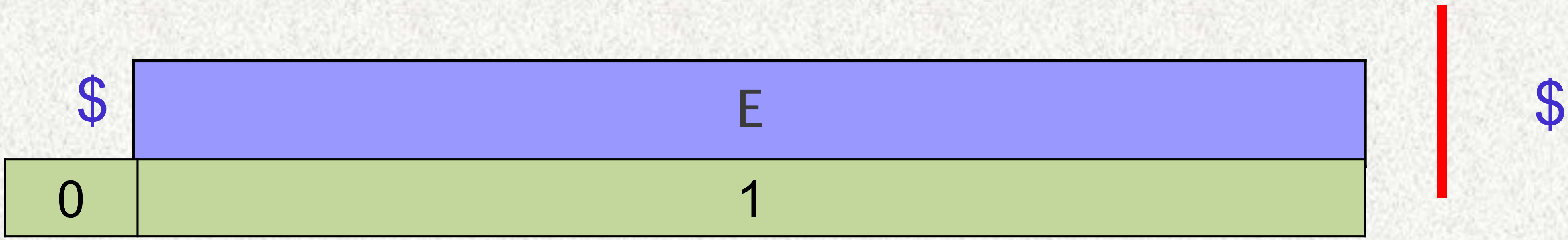
S → E
E → T;
E → T + E
T → int
T → (E)



LR(0) Parsing

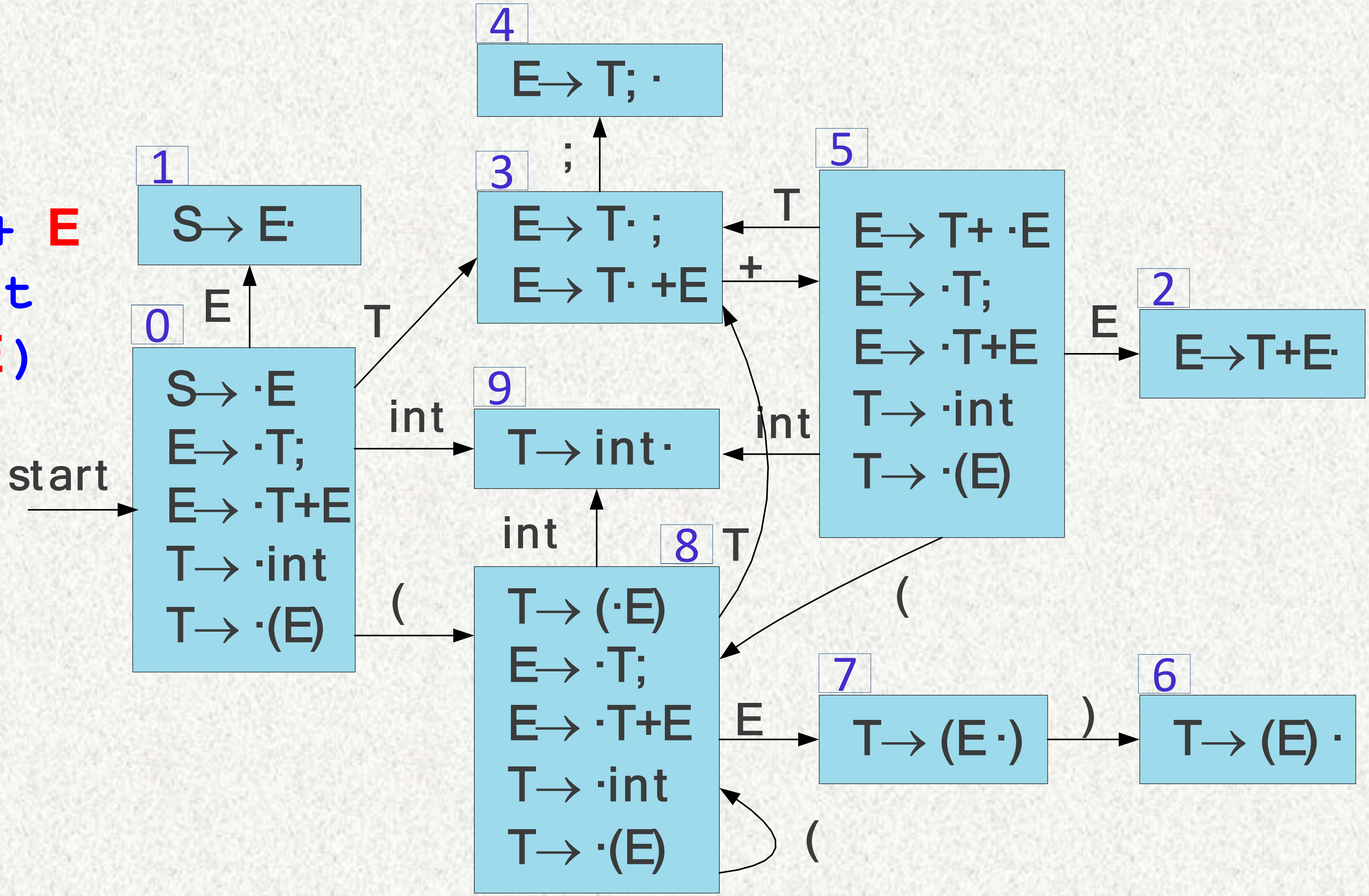
S → E
E → T;
E → T + E
T → int
T → (E)





Building LR(0) Tables

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



LR Tables

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**

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		

Representing the Automaton

- The ACTION function takes as arguments a state i aI1d a terminal a (or $, the input endmarker). The value of ACTION[i, a] can have one of four forms:
  a) Shift j , where j is a state. The action taken by the parser effectively shifts input a to the stack, but uses state j to represent a .
  b) Reduce $A \rightarrow \beta$. The action of the parser effectively reduces β on the top of the stack to head A.
  c) Accept. The parser accepts the input and finishes parsing;
  d) Error.
- We extend the GOTO function, defined on sets of items, to states: if GOTo $[I_i , A] = I_j$, then GOTO also maps a state i and a nonterminal A to state j .

Limit of LR(0)

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

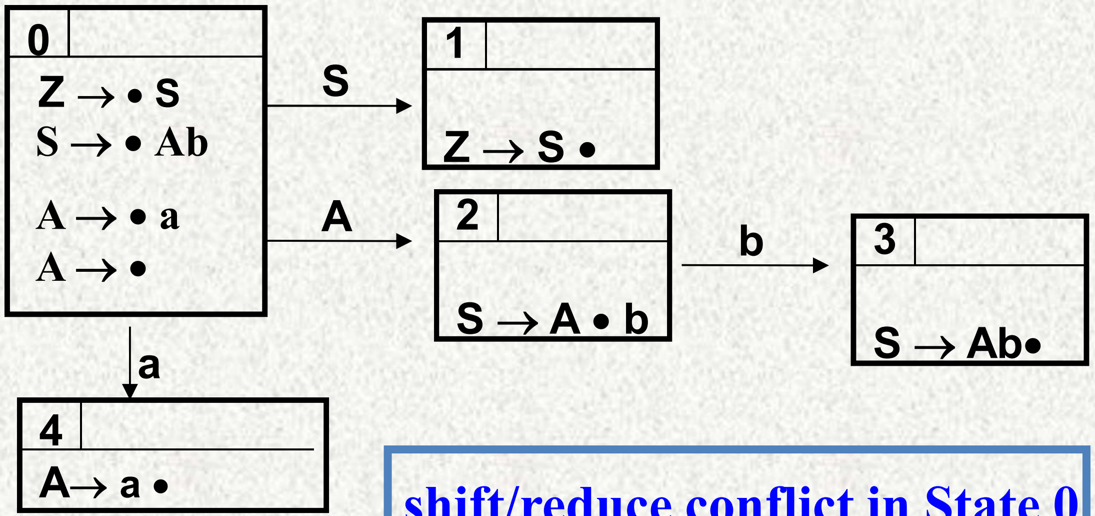
P:

{ (1) S→ Ab

(2) A → ε

(3) A → a

}



shift/reduce conflict in State 0

(1) Shift item: **A → • a**

(2) Reducible item: **A → •**

reduce/reduce conflict

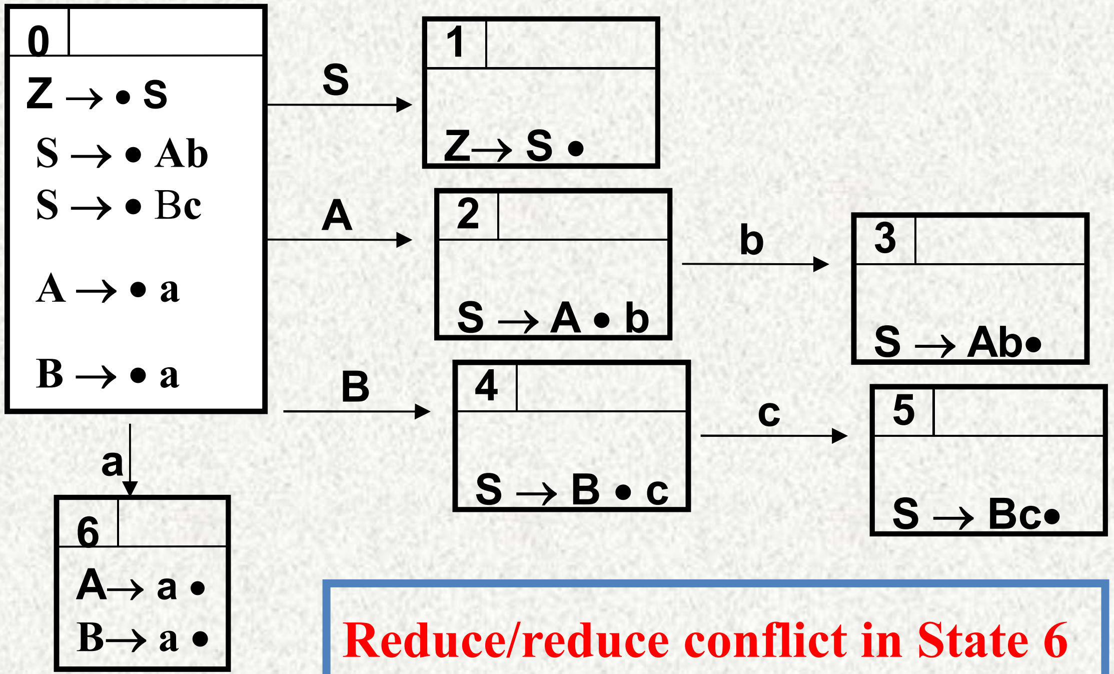
$V_T = \{a, b, c\}$

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

S = S

P:

{(1) S → Ab
(2) S → Bc
(3) A → a
(4) B → a
}



Reduce/reduce conflict in State 6

(1) Reduce item 1: A → a •

(2) Reduce item 2: B → a•

How to resolve?

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

SLR 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) = Φ, 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

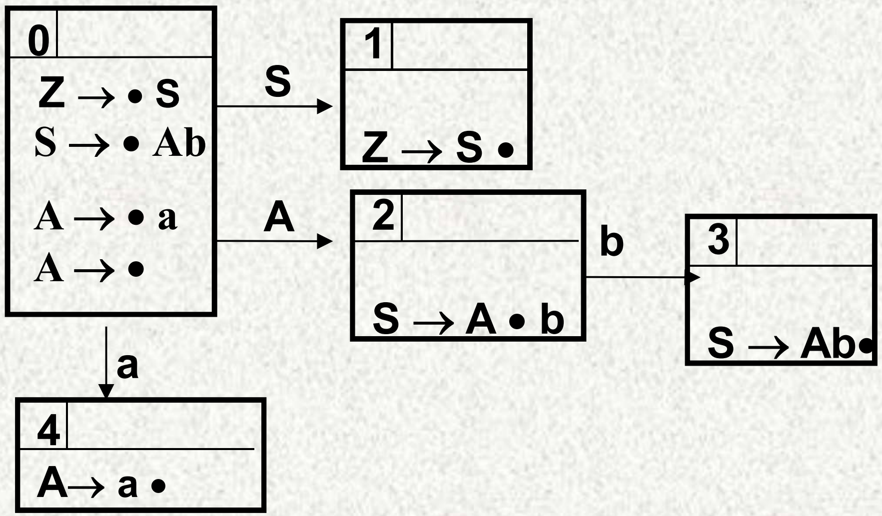
V_T = {a, b}

V_N = {S, A}

S = S

P:

{ (1) S→ Ab
(2) A → ε
(3) A → 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 → • a

(2) reduce: A → •

LR(0) table 1 without S/R conflict

V_T = {a, b}

V_N = {S, A}

S = S

P:
{ (1) S→ Ab
(2) A → ε
(3) A → 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)

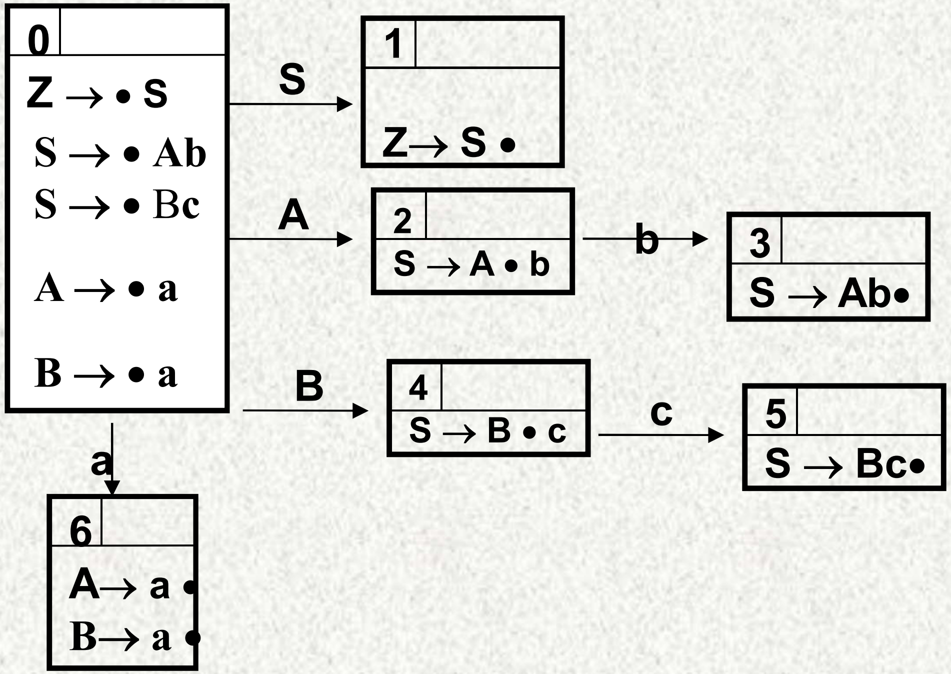
LR(0) table 2 with S/R conflict

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

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

S = S

P:
{(1) S→ Ab
(2) S→ Bc
(3) A → a
(4) B → 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 → a •
(2) Reduce item 2: B → a•

LR(0) table 2 without S/R conflict

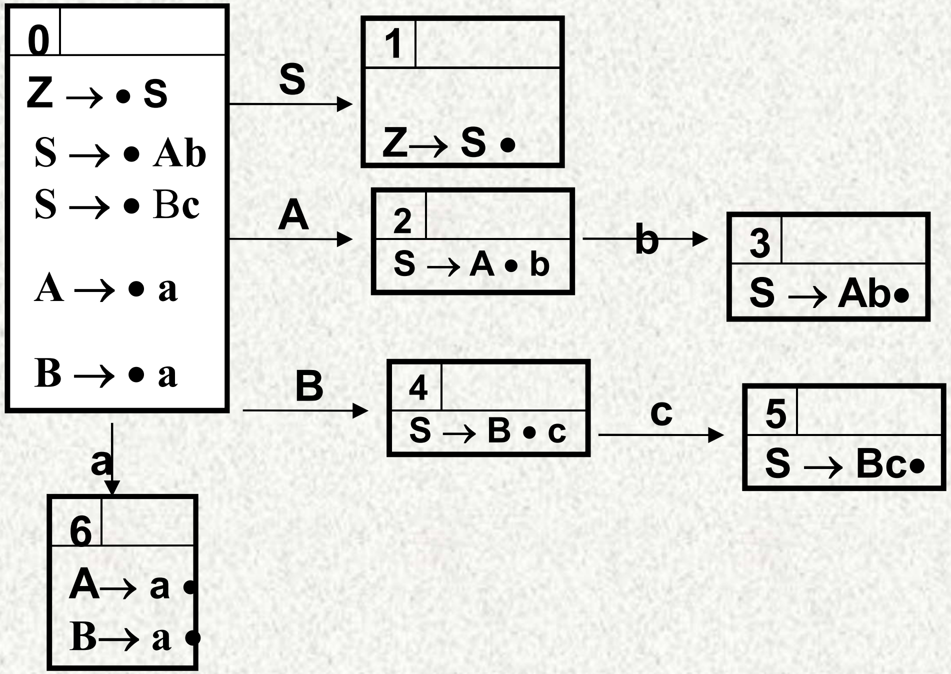
$V_T = \{a, b, c\}$

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

S = S

P:

{(1) S→ Ab
(2) S→ Bc
(3) A → a
(4) B → 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 → a •
(2) Reduce item 2: B → a•

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.

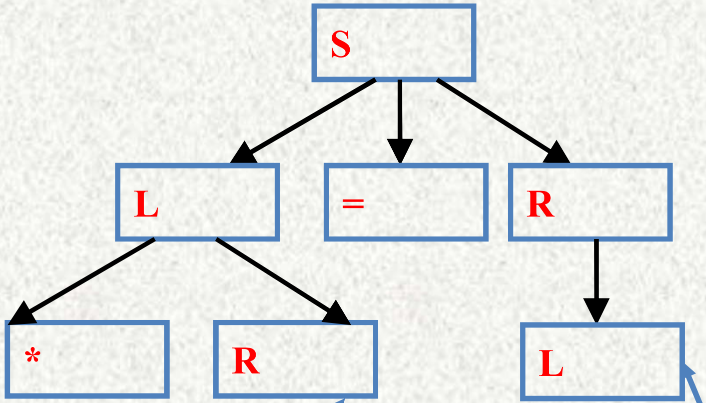
Limitation of SLR(1)

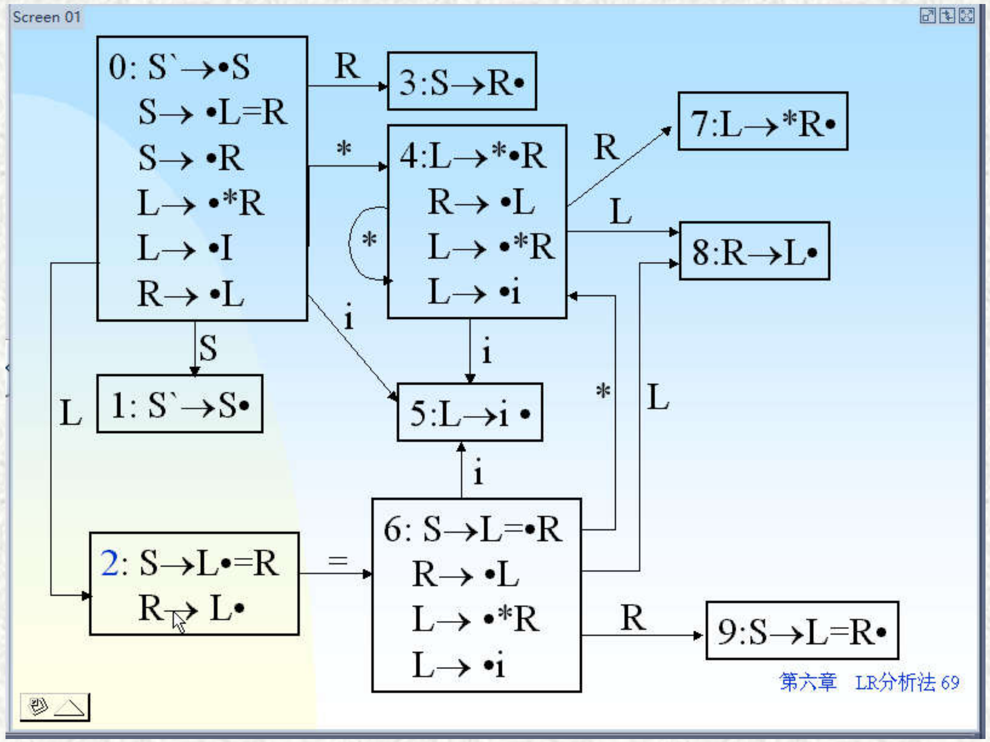
Example 4.51 : Let us reconsider Example 4.48, where in state 2 we had item $R \to L\cdot$, which could correspond to $A \to \alpha$ above, and a could be the = sign, which is in FOLLOW(R). Thus, the SLR parser calls for reduction by $R \to L$ in state 2 with = as the next input (the shift action is 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???

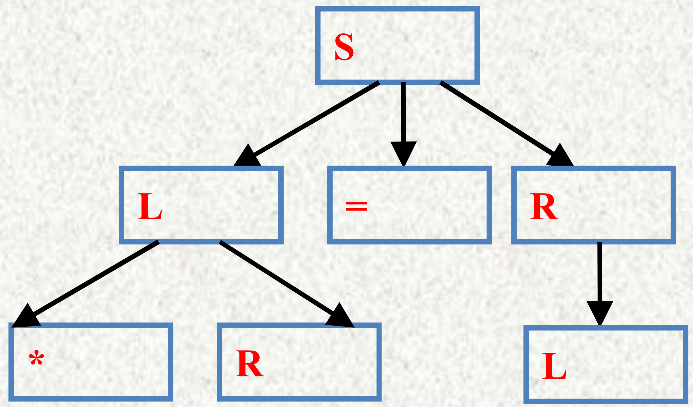
Follow symbol “=“ is actually from here

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.

Homework

Page 258: 4.6.2, 4.6.3;

Page 258: 4.6.4 --- answer the question for 4.2.2(d) (f);

LR(1) Parser

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' → •S, {#})})

- **Type 1**

  S→ α•Xβ, ss —X→ S→ αX•β, ss

- **Type 2**

  S→ α•Aβ, ss
  A → •α1, ss'
  ……..
  A → • αn, ss'
  ……..

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

  ss' = (first(β)-{ε}) ∪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' → • S, \$)}) =
(S' → • S, \$)
(S → • C C, \$)
(C → • c C, c/d)
(C → • d, c/d)

I_1: goto(I_0, S) = (S' → S • , \$)

I_2: goto(I_0, C) =
(S → C • C, \$)
(C → • c C, \$)
(C → • d, \$)

I_3: goto(I_0, c) =
(C → c • C, c/d)
(C → • c C, c/d)
(C → • d, c/d)

I_4: goto(I_0, d) =
(C → d •, c/d)

I_5: goto(I_3, C) =
(S → C C •, \$)

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)

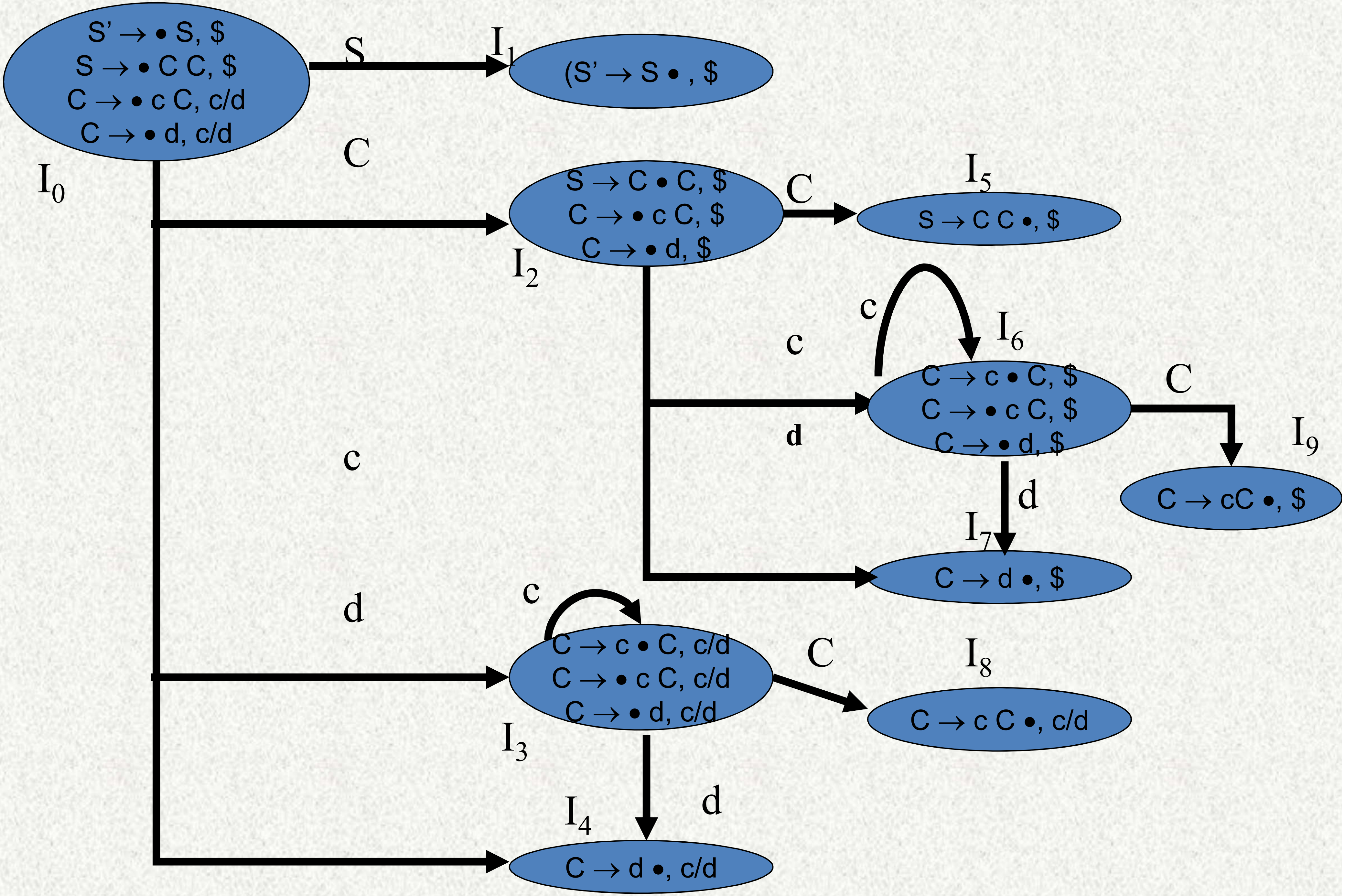
: goto(I_4, c) = I_4

: 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
c
I6
C → c • C, $
C → • c C, $
C → • d, $
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

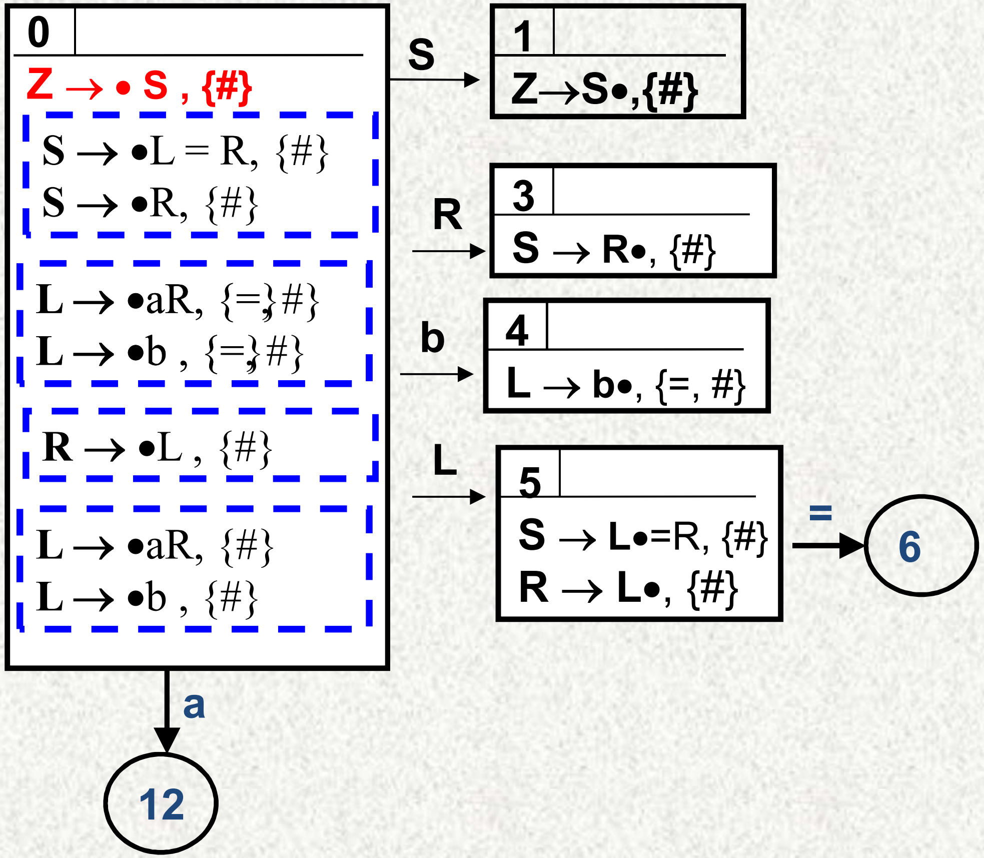
Example 2

V_T = {a, b, =}

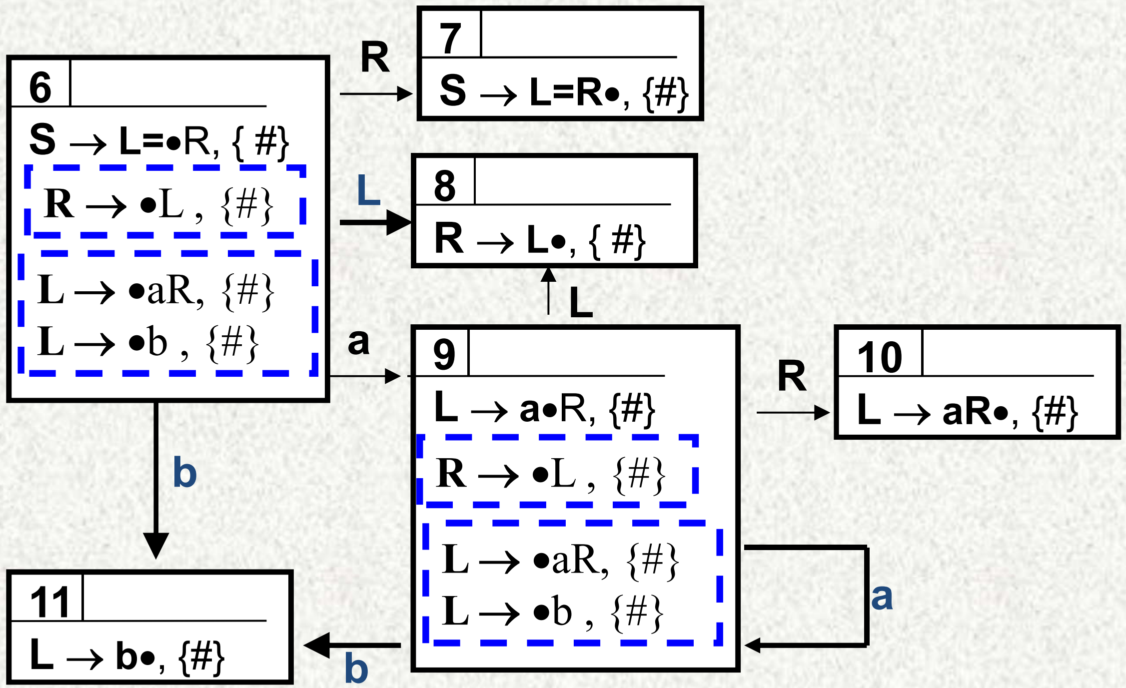
V_N = {S, L, R}

S = S

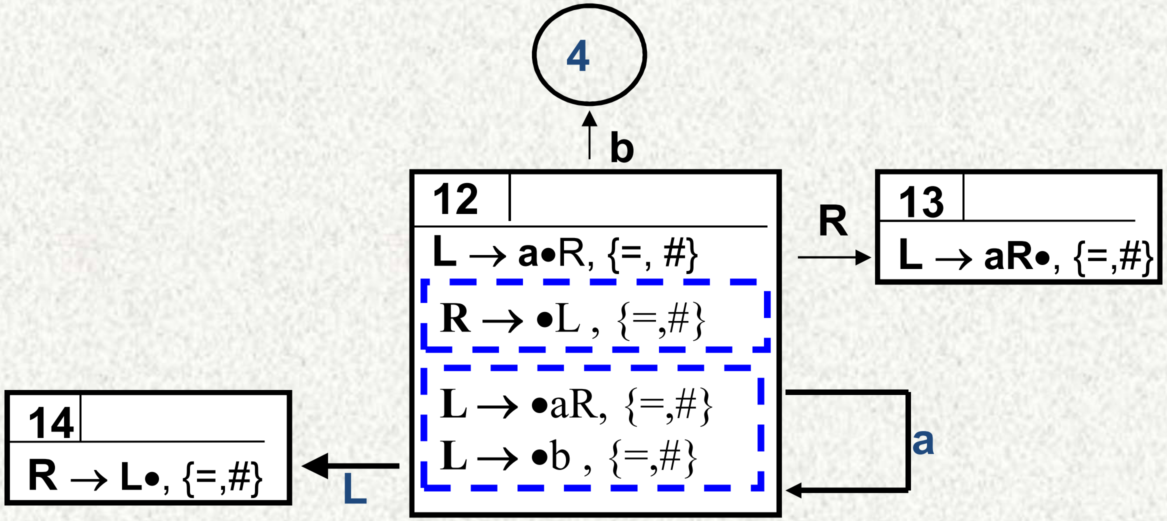
P:
{(1) S → L = R
(2) S → R
(3) L → aR
(4) L → b
(5) R → L
}



Example 2



Example 2



Canonical LR(1) Parsing Table

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

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

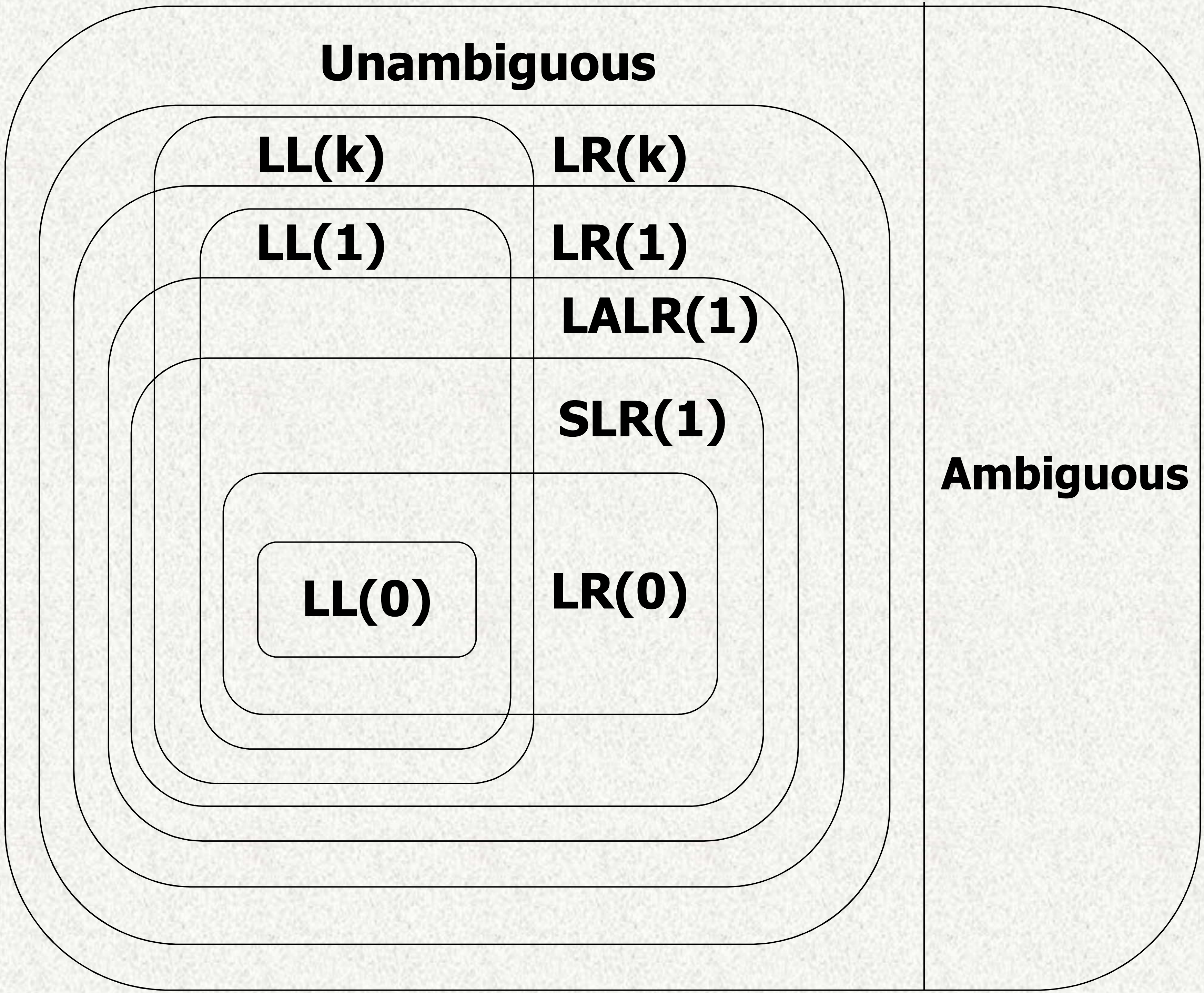
goto (S_i, A) = S_j, if there is an edge from S_i to S_j labeled as A
goto (S_i, A) = error, if there is no edge from S_i to S_j labeled as A

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)