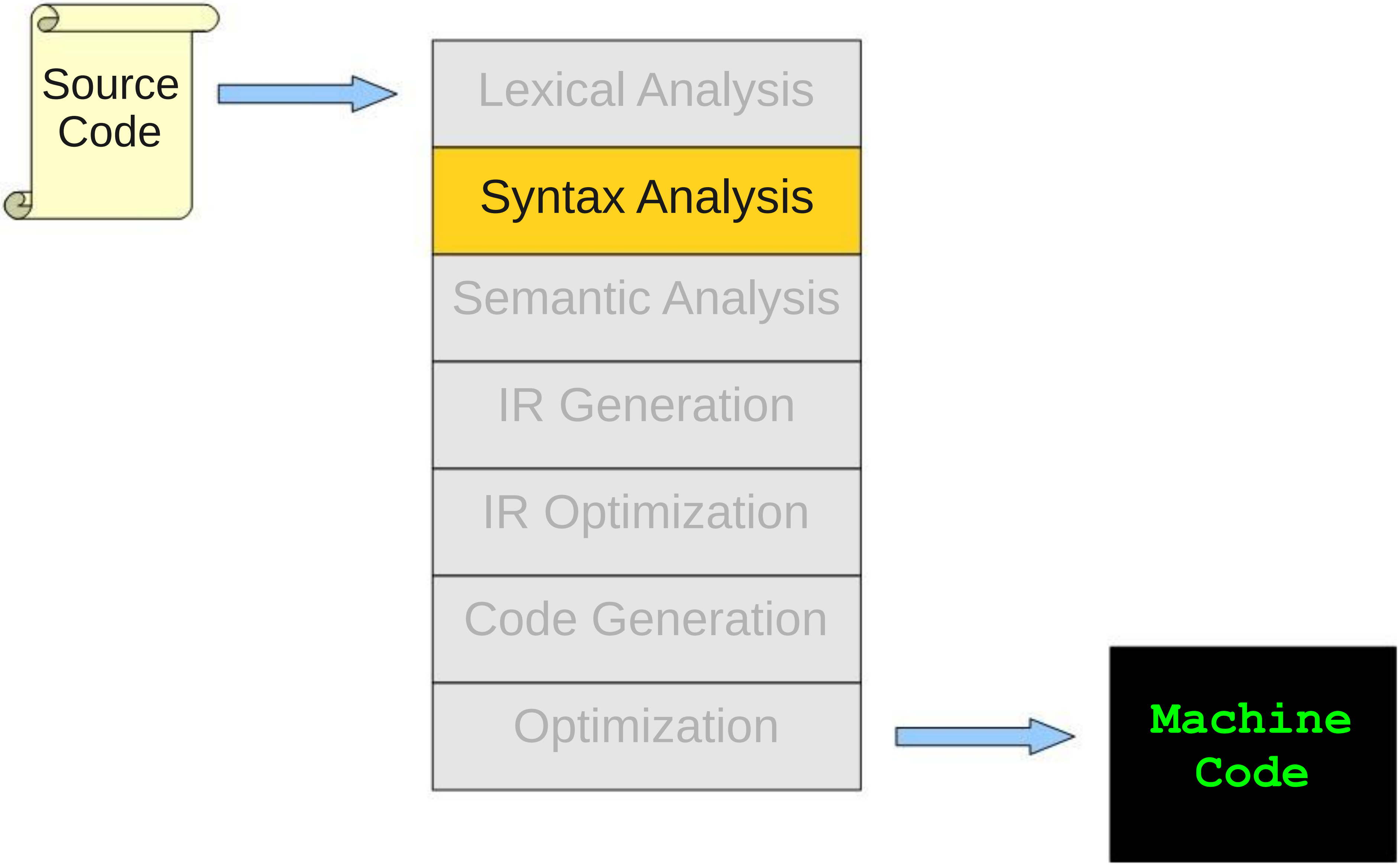
Top-Down Parsing, Part II

Where We Are


Source Code
Lexical Analysis
Syntax Analysis
Semantic Analysis
IR Generation
IR Optimization
Code Generation
Optimization
Machine Code

LL(1) Grammar Construction

- Before constructing a parse table, a **grammar** must be **LL(1)**.

- **LL(1) grammar:** when faced with a choice of several alternatives, we can decide using only the next token, which production to take.

- To make a grammar **LL(1)**
  1. Remove **Left Recursion**
  2. Remove **Left Factoring**

A Grammar that is Not LL(1)

- Consider the following (left-recursive) grammar:

  $A \rightarrow Ab \mid c$

- FIRST(A) = {c}
- However, we cannot build an LL(1) parse table.
- **Why?**

	b	c
A		$A \rightarrow Ab$ $A \rightarrow c$

- **Cannot uniquely predict production!**

Eliminating Left Recursion

- In general, left recursion can be converted into **right recursion** by a mechanical transformation.
- Consider the grammar

  $A \to Av \mid w$
- This will produce *w* followed by some number of *v*'s.
- Can rewrite the grammar as

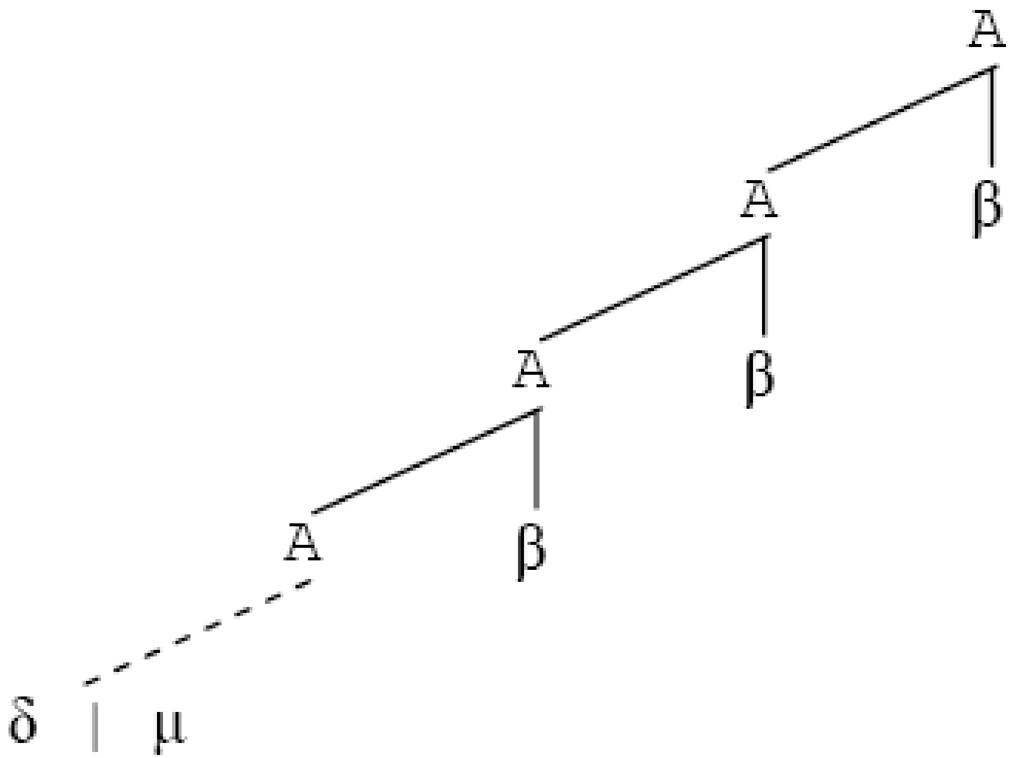
  $A \to wB$

  $B \to \varepsilon \mid vB$

Example: Eliminating Left Recursion

Left recursion

$A \rightarrow A\beta \mid A\gamma \mid \delta \mid \mu$

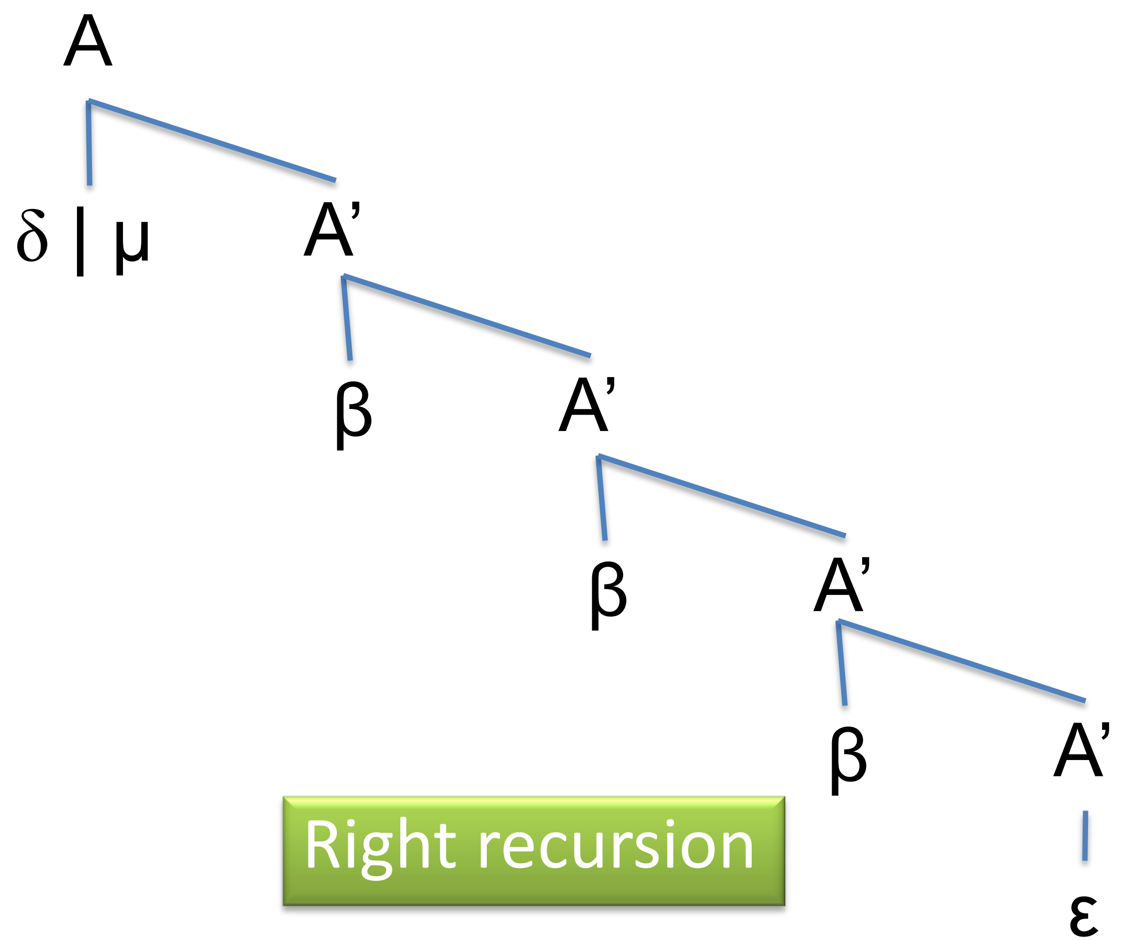


Example: Eliminating Left Recursion

$$A \rightarrow A\beta \mid A\gamma \mid \delta \mid \mu$$

Solution

$$A \rightarrow \delta A' \mid \mu A'$$
$$A' \rightarrow \beta A' \mid \gamma A' \mid \varepsilon$$



Another Non-LL(1) Grammar

- Consider the following grammar:

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

<u>**Left factoring** problem:</u> How do you predict which of these to use?

- FIRST(E) = { **int**, **(** }
- FIRST(T) = { **int**, **(** }
- This grammar is not LL(1).

Eliminating Left-Factoring

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

$S \rightarrow E$
$E \rightarrow TY$
$Y \rightarrow$ **+** $E \mid \varepsilon$
$T \rightarrow$ **`int`**
$T \rightarrow$ `(E)`

LL(1) Table Construction

- When our grammar is LL(1), we can build LL(1) parsing table.
- Compute FIRST(A) and FOLLOW(A) for all nonterminal A.
- For each rule $A \rightarrow w$ and for each **terminal** t in FIRST(w), set T[A, t] = w.
- For each rule $A \rightarrow w$ with ε in FIRST(w), set T[A, t] = w for each t in FOLLOW(A).

Construct LL(1) Parse Table

S → E		1
E → TY		2
T → `int`		3
T → (E)		4
Y → + E		5
Y → ε		6

FIRST			
S	E	T	Y
`int` (	`int` (	`int` (	+ ε

FOLLOW			
S	E	T	Y
$	$)	+ $)	$)

	`int`	(	)	+	$
S					
E					
T					
Y					

Construct LL(1) Parse Table

$S \rightarrow E$		1
$E \rightarrow TY$		2
$T \rightarrow$ **`int`**		3
$T \rightarrow$ **`(E)`**		4
$Y \rightarrow$ **+** E		5
$Y \rightarrow \varepsilon$		6

FIRST			
S	E	T	Y
`int` (	**`int`** (	**`int`** (	**+** **ε**

FOLLOW			
S	E	T	Y
\$	\$)	+ \$)	\$)

	`int`	(	)	+	\$
S	1	1			
E	2	2			
T	3	4			
Y			6	5	6

Another example : LL(1) Parse Table

statement → **if-stmt** | other

if-stmt → if (**exp**) **statement else-part**

else-part → else **statement** | ε

exp → 0 | 1

First sets

statement	if-stmt	else-part	exp
if	if	else	0
other		ε	1

Follow sets

statement	if-stmt	else-part	exp
$	$	$	)
else	else	else	

Another example : LL(1) Parse Table

statement → **if-stmt**	1
statement → other	2
if-stmt → if (**exp**) **statement else-part**	3
else-part → else **statement**	4
else-part → ε	5
exp → 0	6
exp → 1	7

First sets

statement	if-stmt	else-part	exp
if	if	else	0
other		ε	1

Follow sets

statement	if-stmt	else-part	exp
$	$	$	)
else	else	else	

	if	other	else	0	1	$
statement	1	2				
if-stmt	3					
else-part			4			
exp						

Another example : LL(1) Parse Table

statement → **if-stmt**	1
statement → other	2
if-stmt → if (**exp**) **statement else-part**	3
else-part → else **statement**	4
else-part → ε	5
exp → 0	6
exp → 1	7

First sets

statement	if-stmt	else-part	exp
if	if	else	0
other		ε	1

Follow sets

statement	if-stmt	else-part	exp
$	$	$	)
else	else	else	

	if	other	else	0	1	$
statement	1	2				
if-stmt	3					
else-part			4 5			5
exp						

Another example : LL(1) Parse Table

statement → **if-stmt**	1
statement → other	2
if-stmt → if (**exp**) **statement else-part**	3
else-part → else **statement**	4
else-part → ε	5
exp → 0	6
exp → 1	7

First sets

statement	if-stmt	else-part	exp
if	if	else	0
other		ε	1

Follow sets

statement	if-stmt	else-part	exp
$	$	$	)
else	else	else	

	if	other	else	0	1	$
statement	1	2				
if-stmt	3					
else-part			4 5			5
exp				6	7	

The grammar is ambiguous

LL(1) is Realistic

- Some real-world programming languages are LL(1)-parsable or parsable with a minor modification on LL(1).

- Examples:
  - LISP
  - Python
  - JavaScript

Another way to implement LL(1) parser

- Can be implemented quickly with a table-driven design.
- Can also be implemented by **recursive descent**:
  - Define a function for each nonterminal.
  - Have these functions call each other based on the lookahead token.

Recursive Descent Program

Consider the following LL(1) grammar:

$E \rightarrow TE'$
$E' \rightarrow \varepsilon \mid +E$
$T \rightarrow FT'$
$T' \rightarrow \varepsilon \mid *T$
$F \rightarrow id \mid (E)$

```
Procedure E()
   if T() then
      if E'() then return true;
   error;
end;
```

```
Procedure E'()
 if match('+') then
    if E() then return true;
    else error;
 return true;     // match ε
end;
```

```
Procedure F()
  if match(id) then
     return true;
  if match('(') then
     if E() then
      if match (')') then return true;
      error;
     error;
   error;
end;
```

Comparison :
Table driven VS Recursive descent

Table driven	Recusive descent
Small/ Fast	Expressive / Fast
Hard to debug	Bulky

Summary

(Self review)

- **Top-down parsing** tries to derive the user's program from the start symbol.
- **Leftmost BFS** is one approach to top-down parsing; it consumes time and space.
- **LL(1)** parsing scans from left-to-right, using one token of look ahead to find a leftmost derivation.
- **FIRST sets** contain terminals that may be the first symbol of a production.
- **FOLLOW sets** contain terminals that may follow a nonterminal in a production.
- **Left recursion** and **left factoring** cause LL(1) to fail and can be mechanically eliminated.
- Two main approaches of **Top-down parsing: Table driven, Recursive Descent program**.

Bottom-Up Parsing

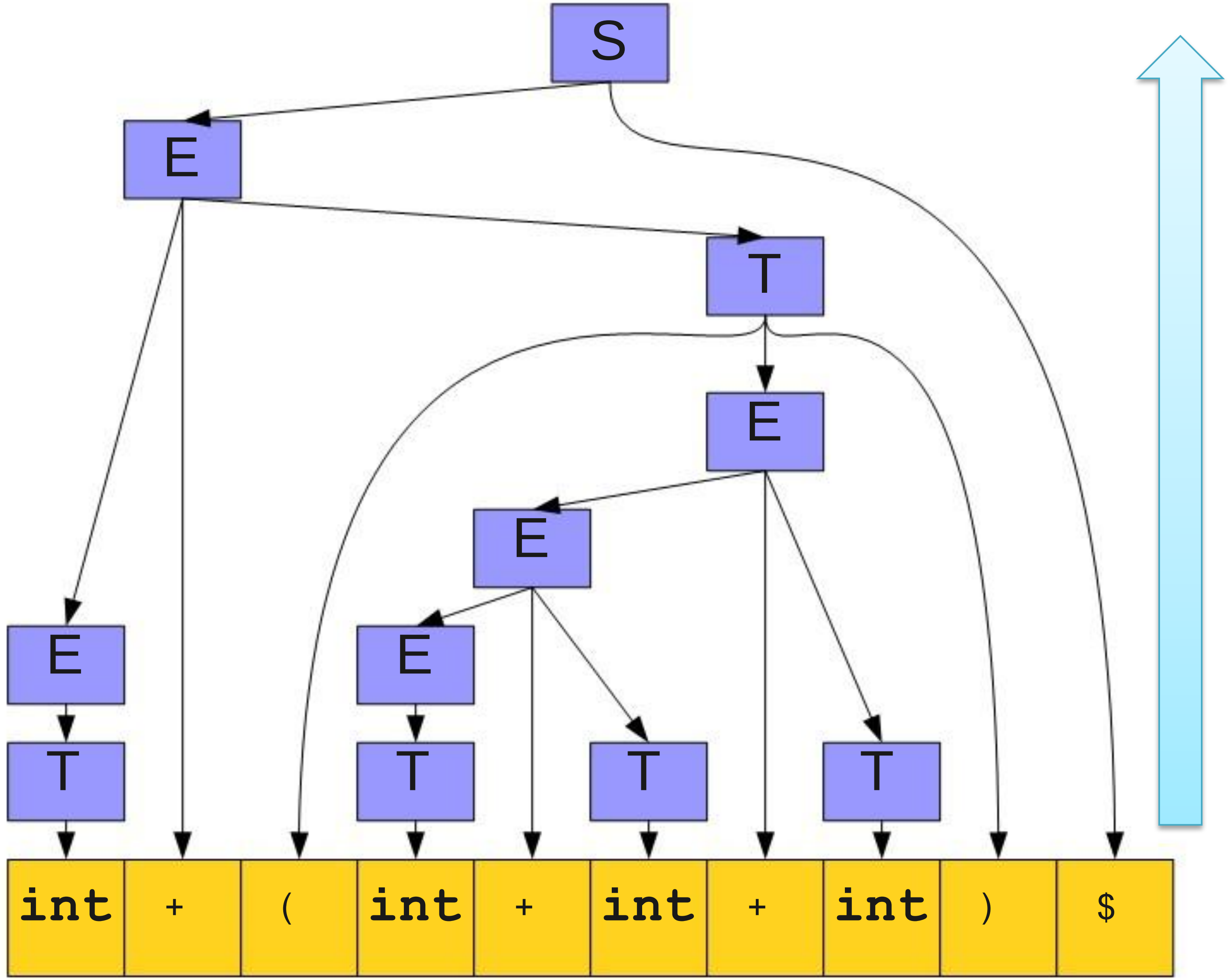
Why need Bottom-Up Parsing?

- If a language grammar has **multiple rules** that may start with the **same leftmost symbols** but have **different endings**, then that grammar can be more efficiently handled by a bottom-up parsing.

- So bottom-up parsers handle a somewhat **larger range of computer language grammars** than do deterministic top-down parsers.

View of a Bottom-Up Parsing

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



Handles

- Suppose our grammar contains a production **A** → ***w*** and there is a derivation *u**w**v* → *u**A**v* → … → **S** (where S is the start symbol). We call the substring **A** as a **handle**.

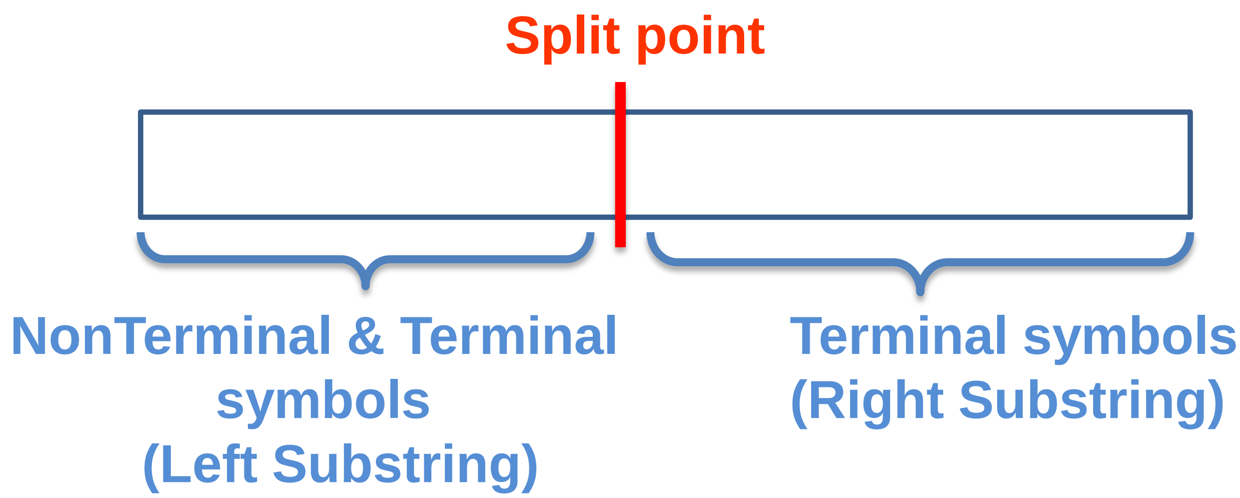
- When the parser finds the handle at the top of the parsing stack it performs a **reduction**.

- The rightmost derivation is required for the bottom-up parser.

Shift/Reduce Parsing

- Shift/reduce parsing is the most commonly used and the most powerful of the bottom-up techniques.
- Idea: Split the input into two parts:
  - **Left** substring is our work area.
  - **Right** substring is input we have not yet processed.
- All **handles** are reduced in the **left substring**.
- **Right substring** (of the split point) consists only of **terminals**.
- At each point, decide whether to:
  - Move a terminal across the split (**shift**)
  - Reduce a handle (**reduce**)

A Sample Shift/Reduce Parse



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

Note: Don't worry about when to shift or when to reduce. We will learn that soon.

int	+	int	*	int	+	int	$

A Sample Shift/Reduce Parse

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

Shift **int**

int	|	+	**int**	*	**int**	+	**int**	$

A Sample Shift/Reduce Parse

S → E**$**

E → F

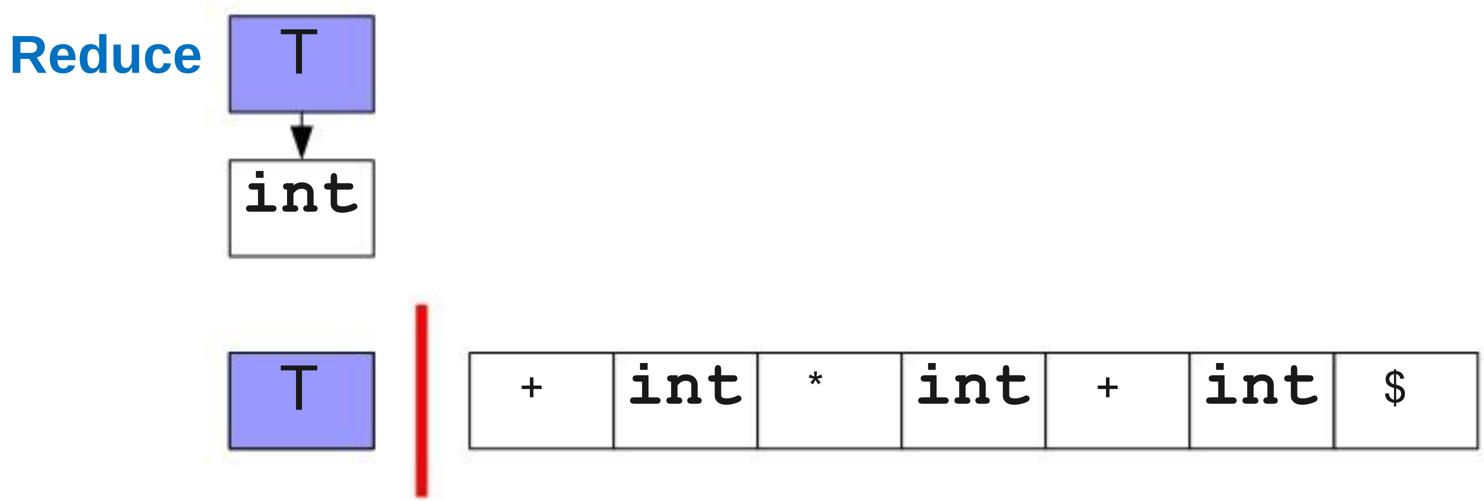
E → E **+** F

F → F ***** T

F → T

T → **int**

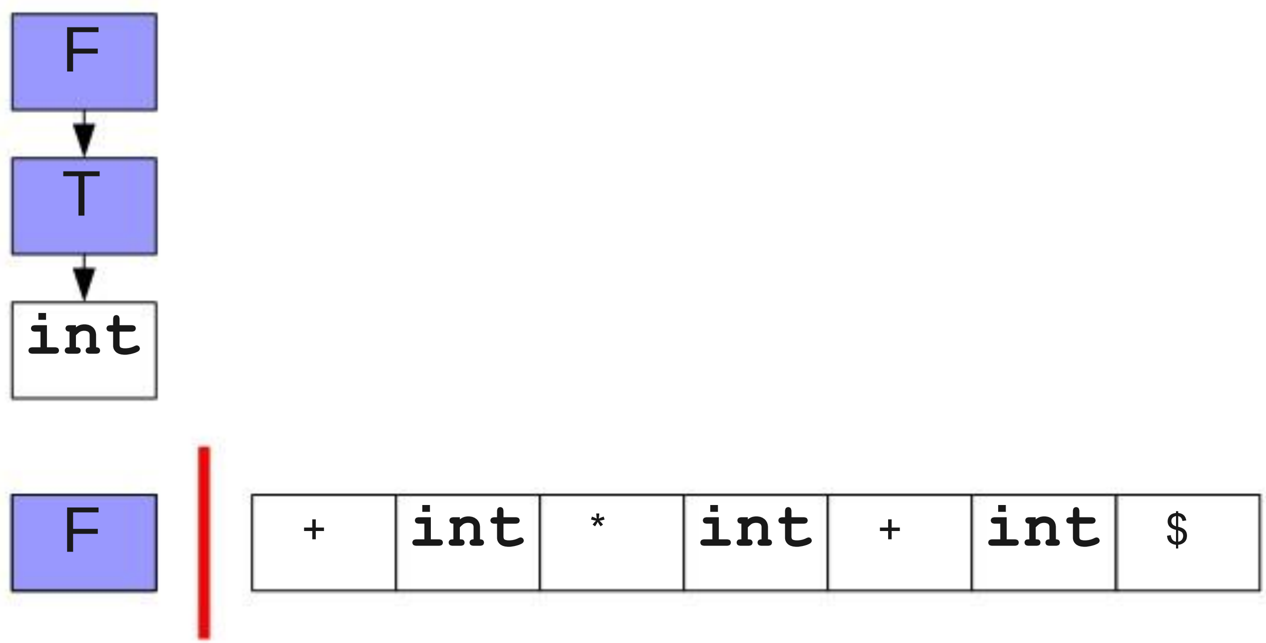
T → **(**E**)**



A Sample Shift/Reduce Parse

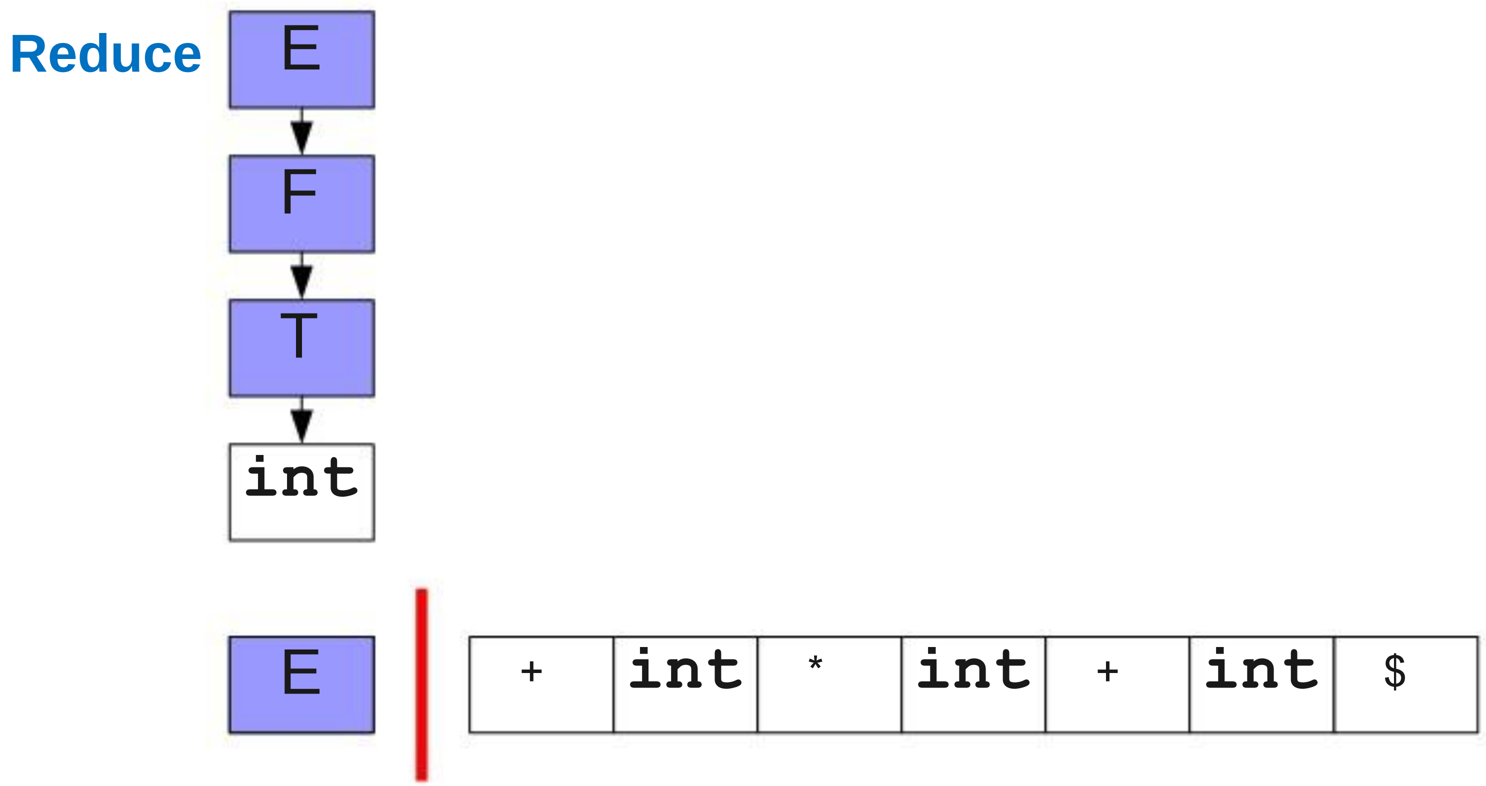
S → E**$**
E → F
E → E **+** F
F → F ***** T
F → T
T → **int**
T → **(**E**)**

Reduce



A Sample Shift/Reduce Parse

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



A Sample Shift/Reduce Parse

$S \rightarrow E\$$

$E \rightarrow F$

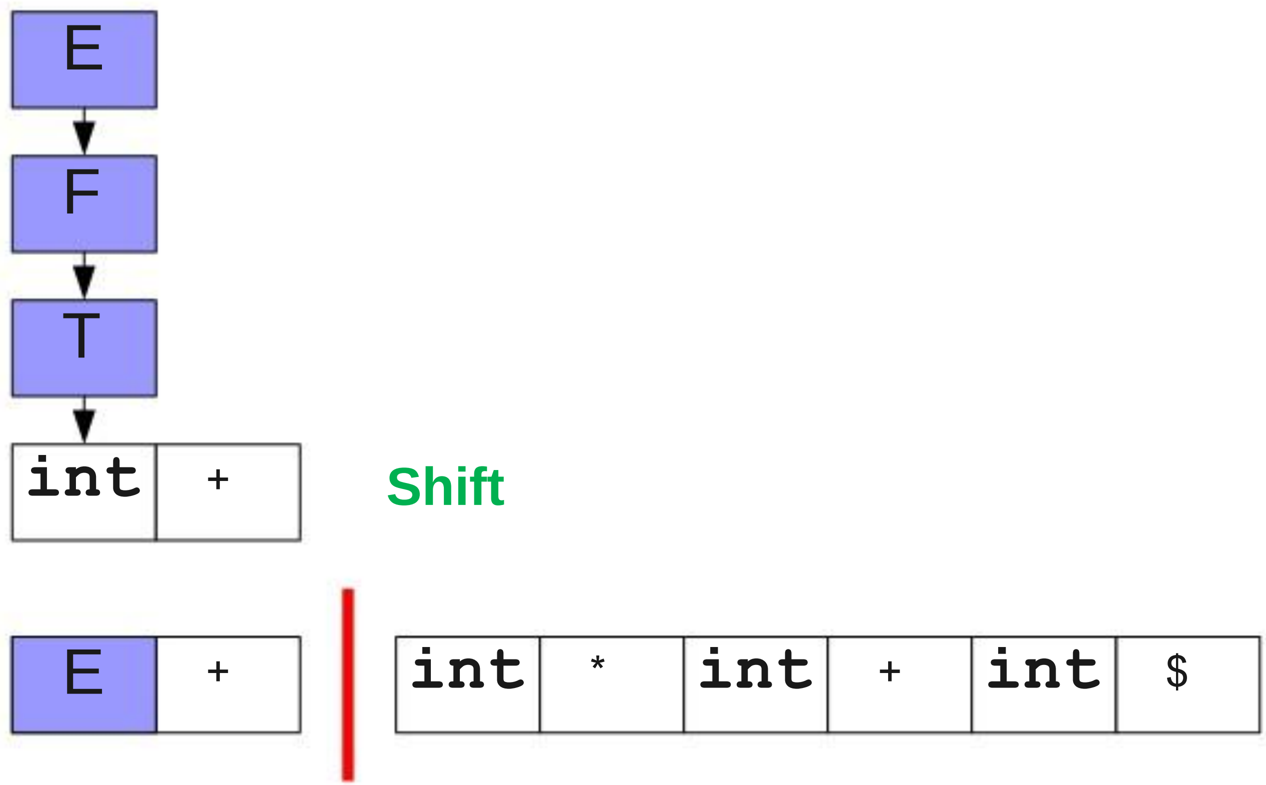
$E \rightarrow E + F$

$F \rightarrow F * T$

$F \rightarrow T$

$T \rightarrow \mathtt{int}$

$T \rightarrow (E)$



A Sample Shift/Reduce Parse

$S \rightarrow E\$$

$E \rightarrow F$

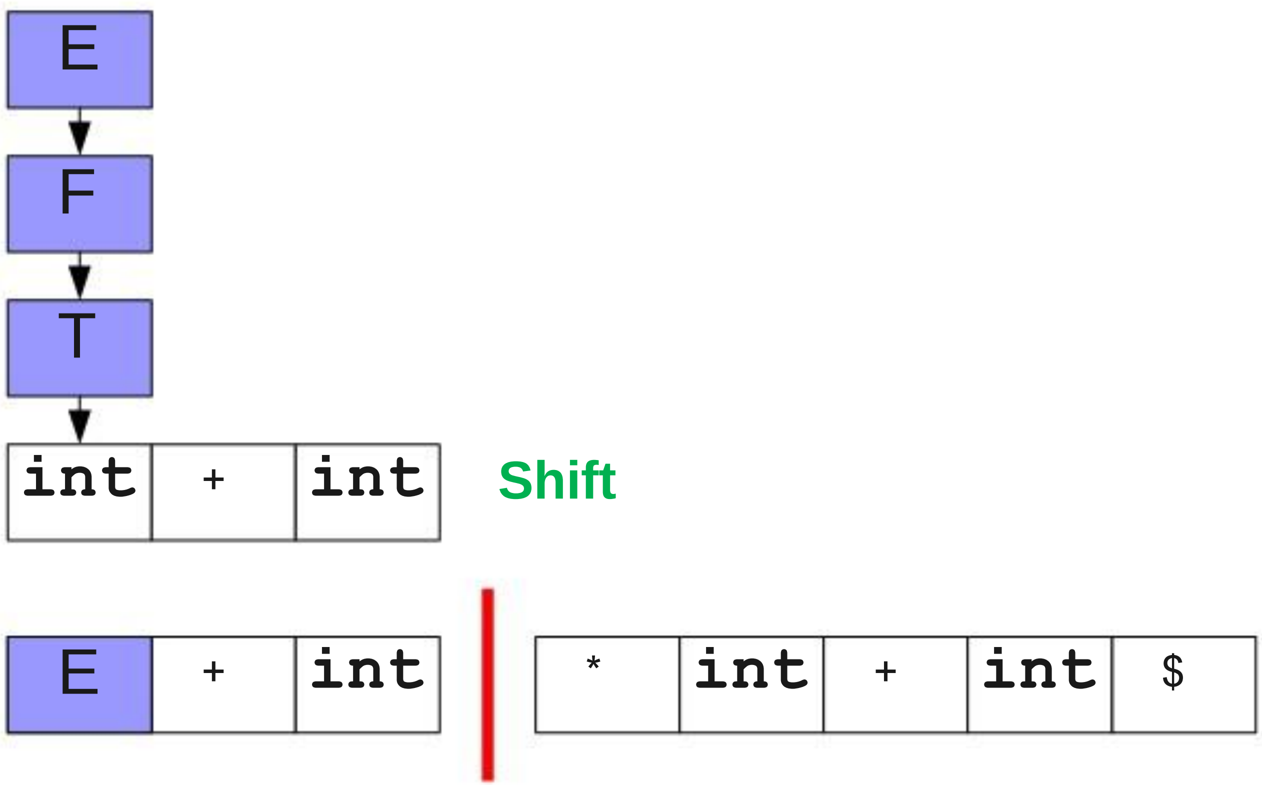
$E \rightarrow E + F$

$F \rightarrow F * T$

$F \rightarrow T$

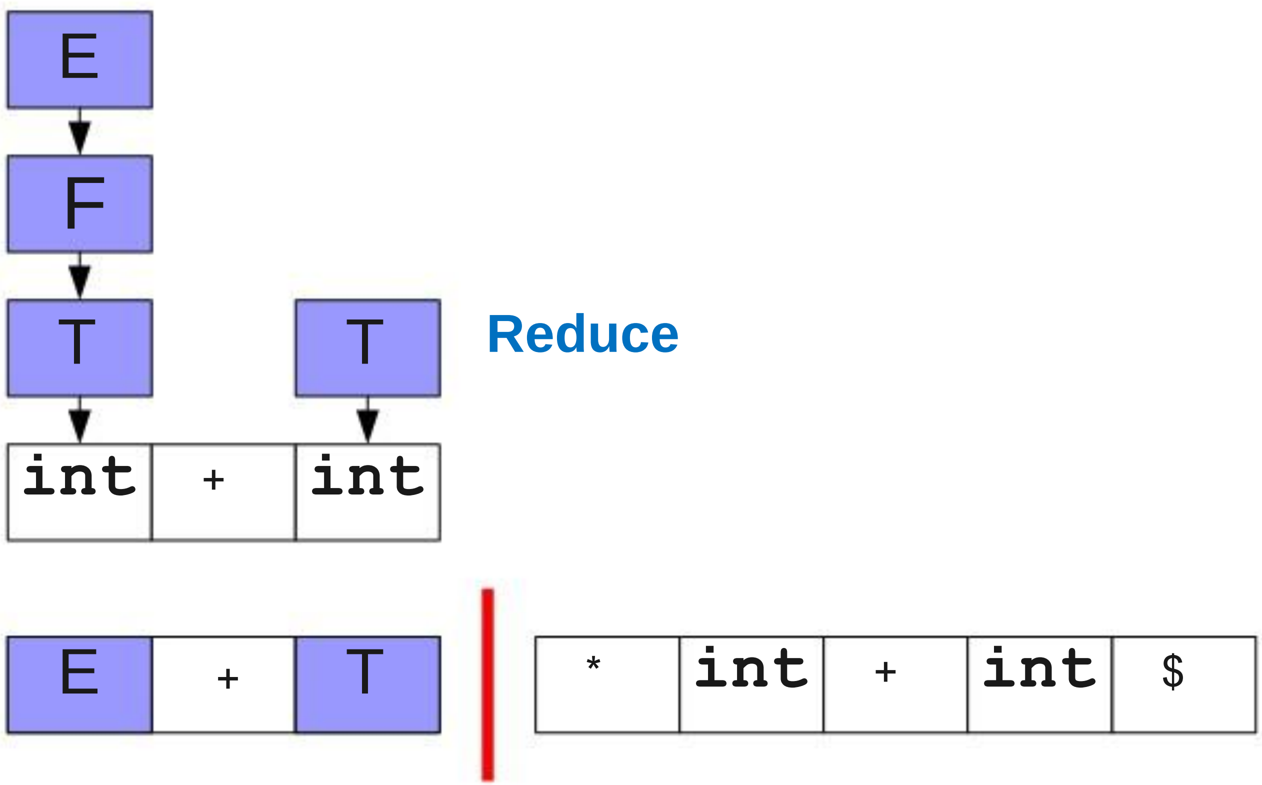
$T \rightarrow \texttt{int}$

$T \rightarrow (E)$



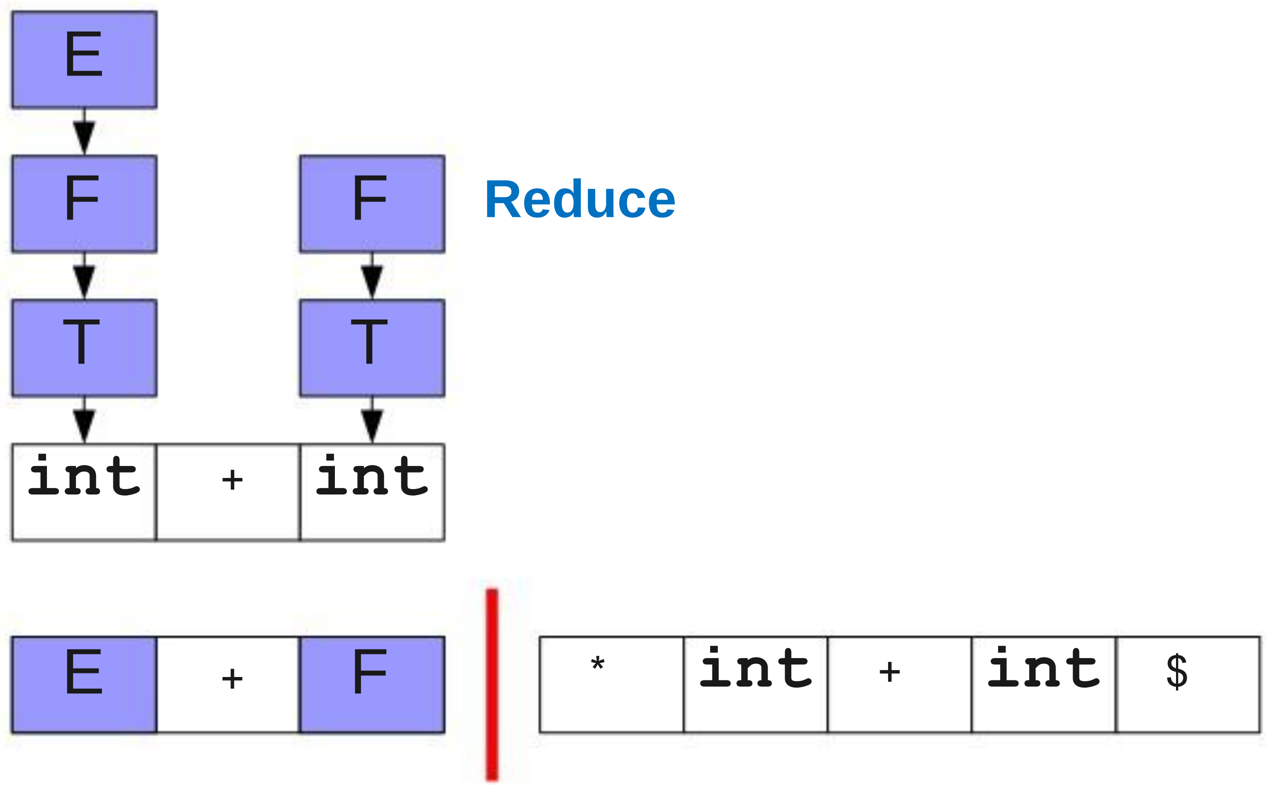
A Sample Shift/Reduce Parse

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



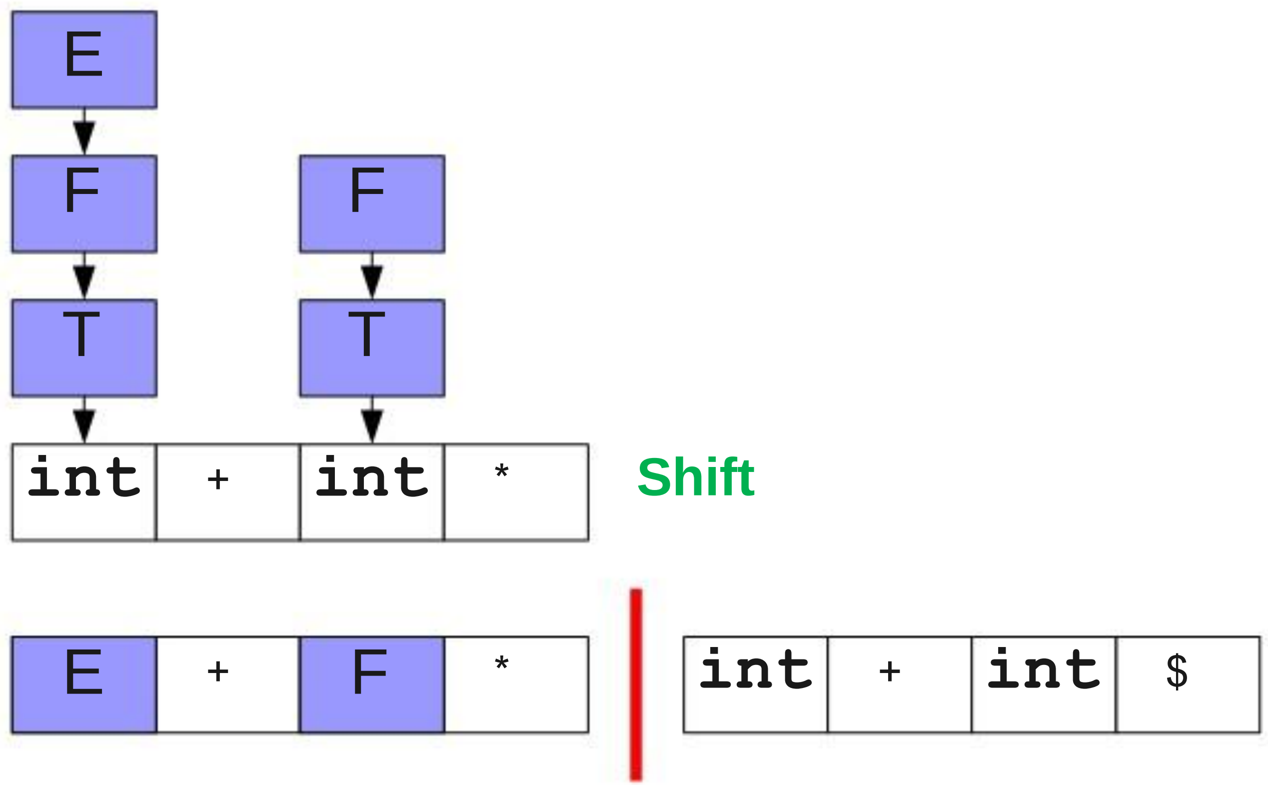
A Sample Shift/Reduce Parse

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



A Sample Shift/Reduce Parse

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



A Sample Shift/Reduce Parse

S → E$

E → F

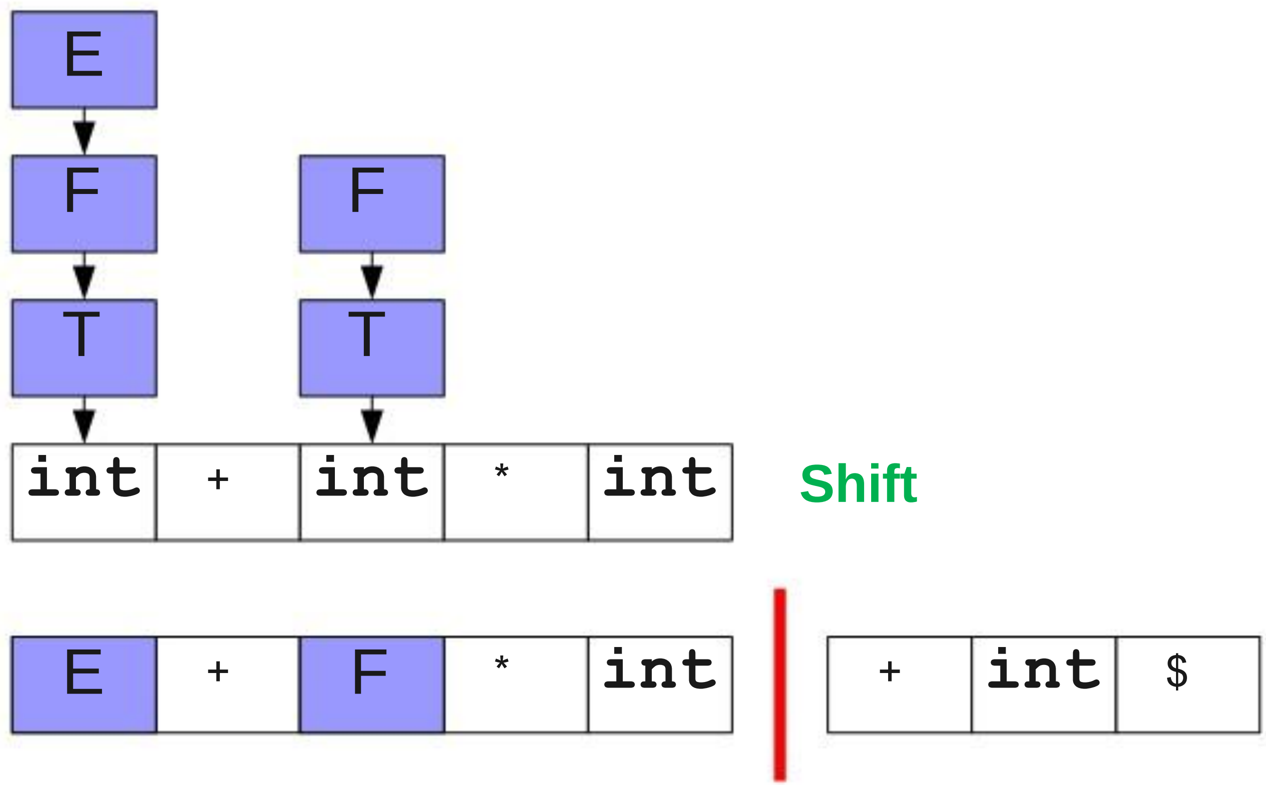
E → E + F

F → F * T

F → T

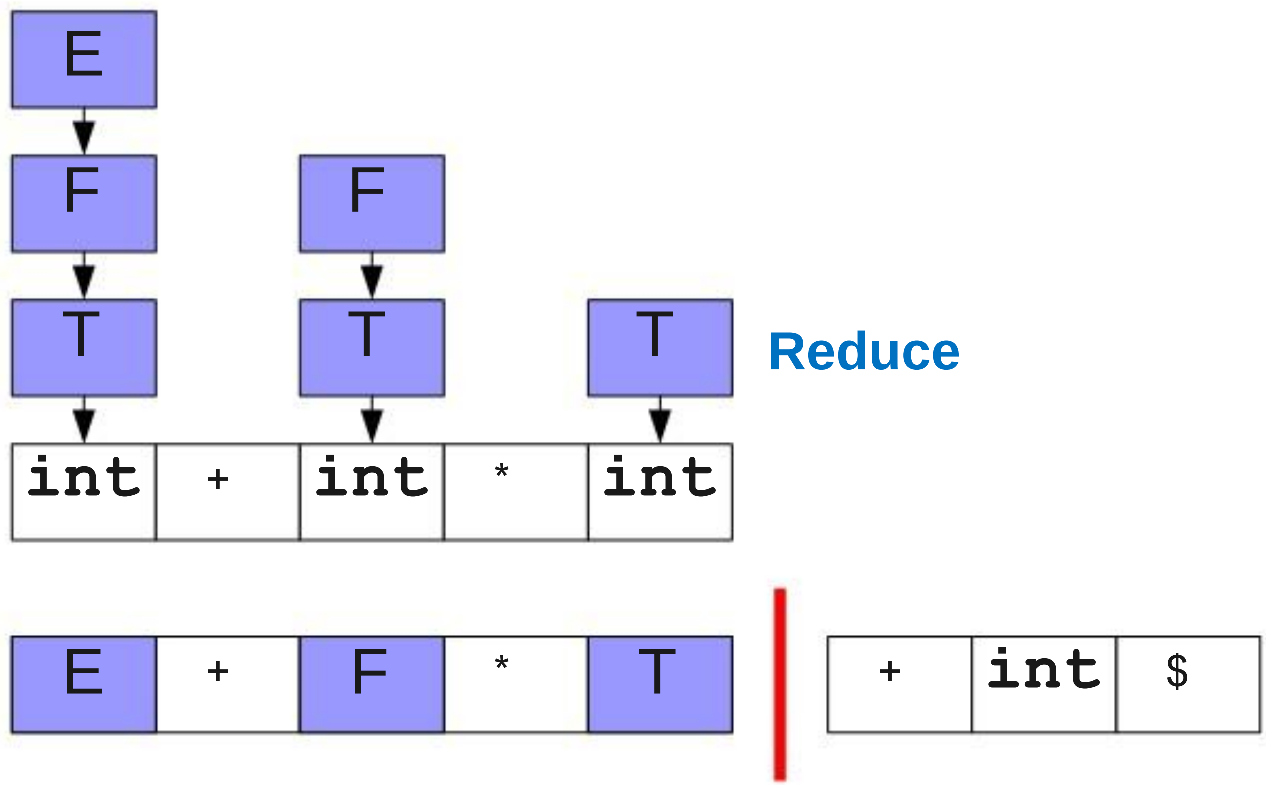
T → **int**

T → (E)



A Sample Shift/Reduce Parse

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



A Sample Shift/Reduce Parse

S → E$

E → F

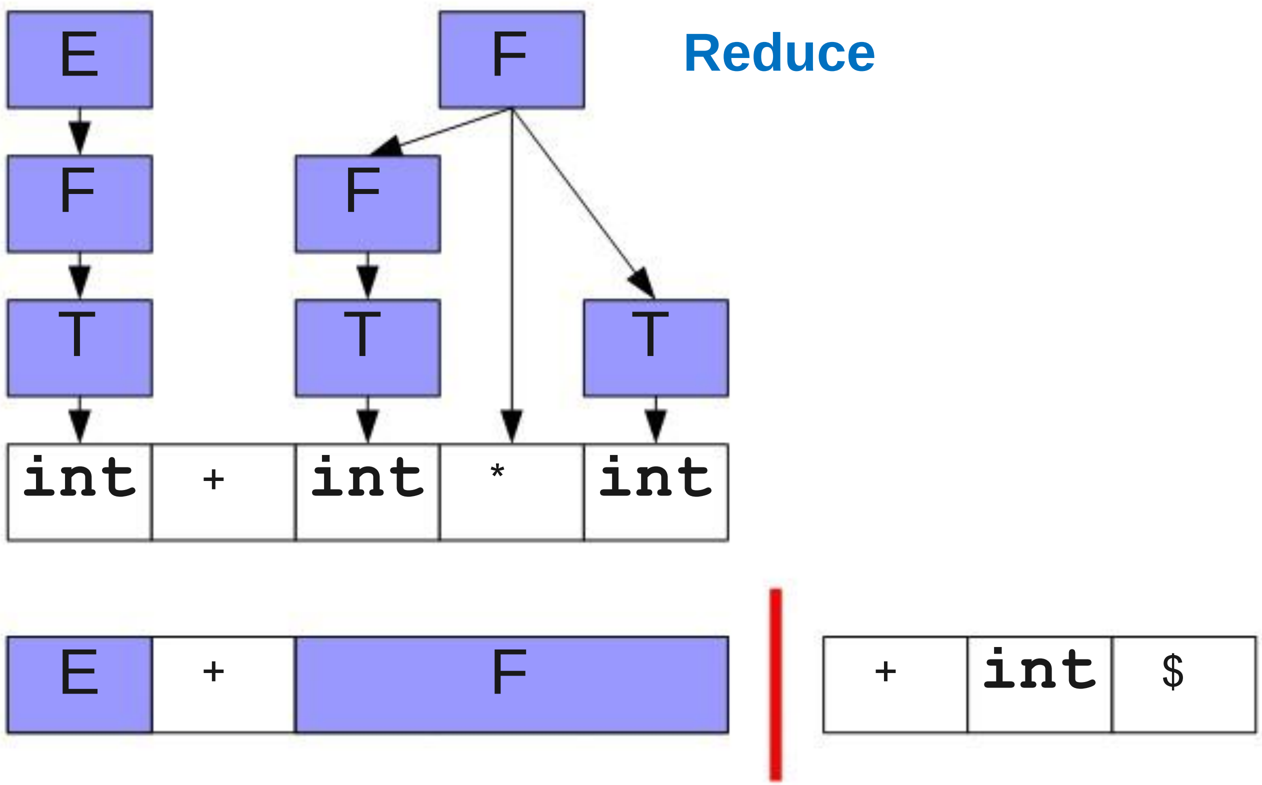
E → E + F

F → F * T

F → T

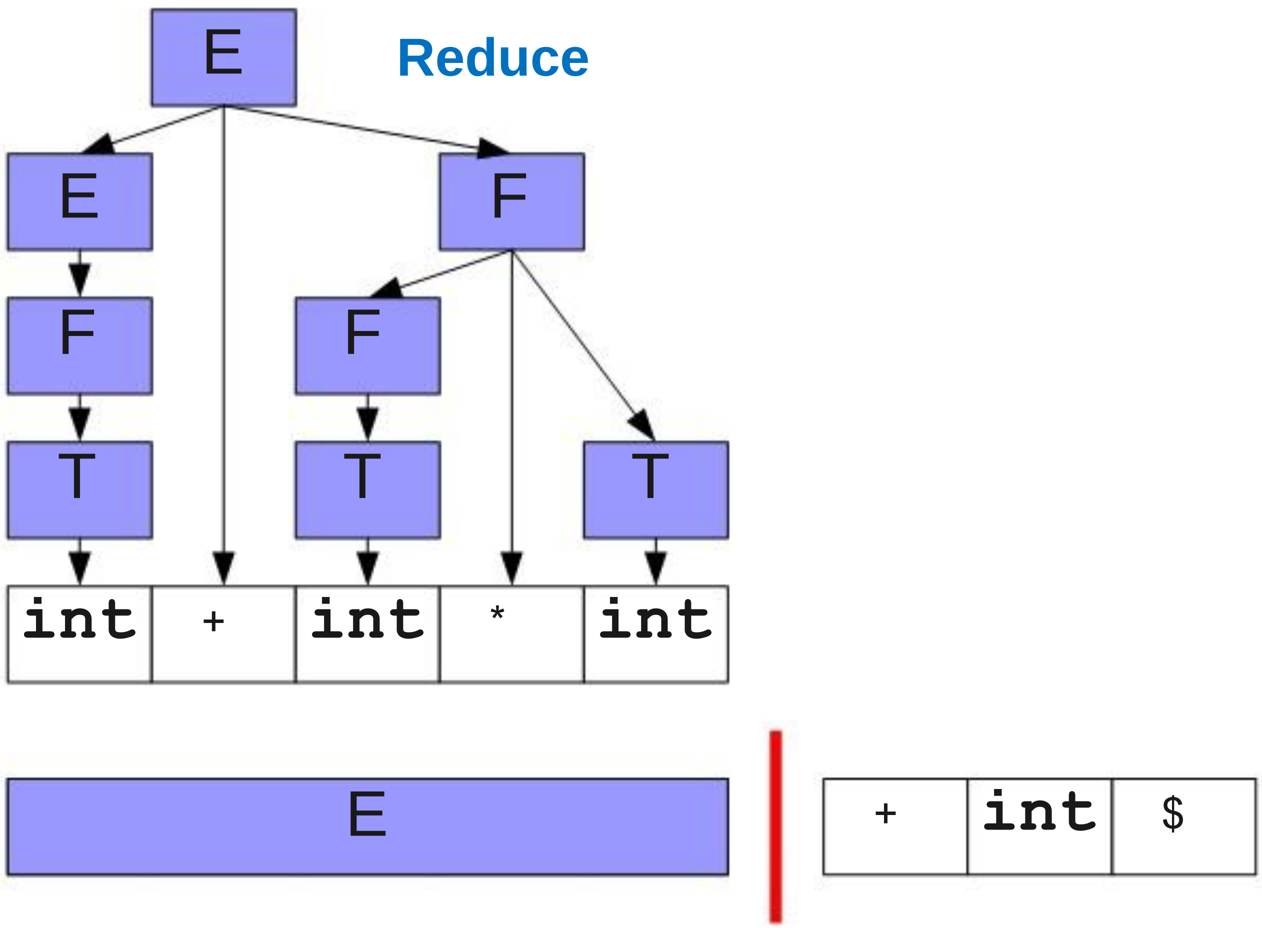
T → int

T → (E)



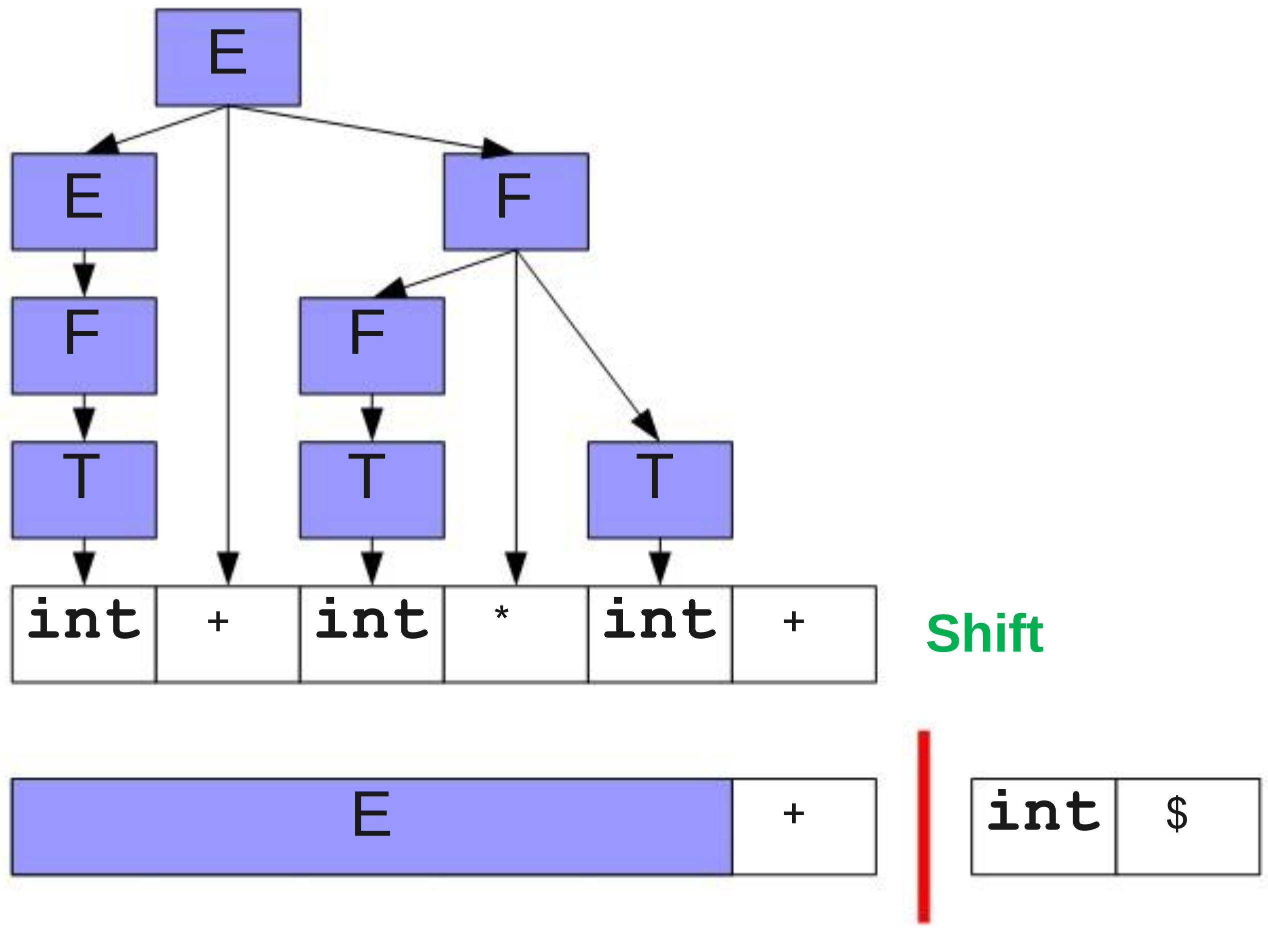
A Sample Shift/Reduce Parse

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



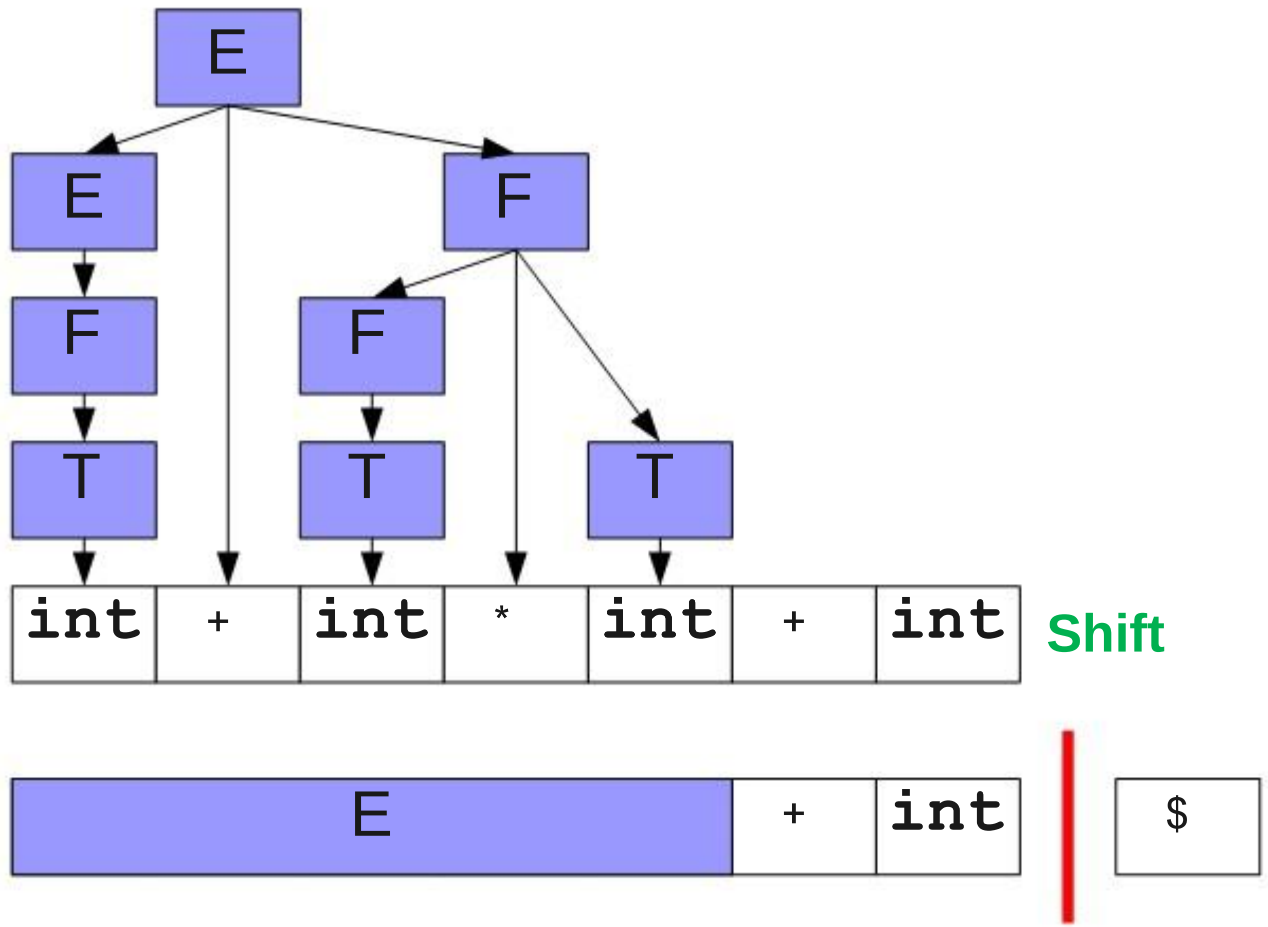
A Sample Shift/Reduce Parse

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



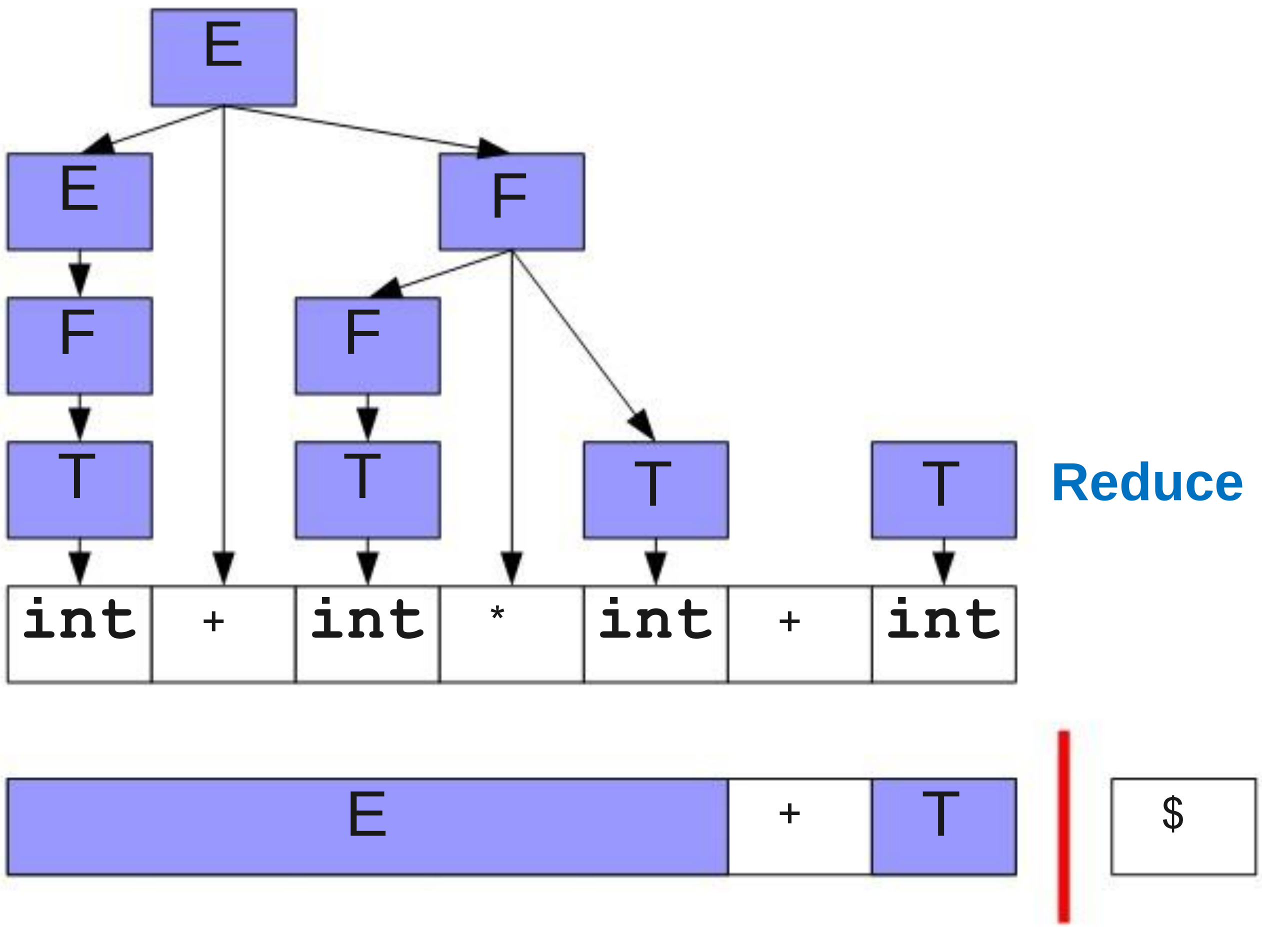
A Sample Shift/Reduce Parse

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



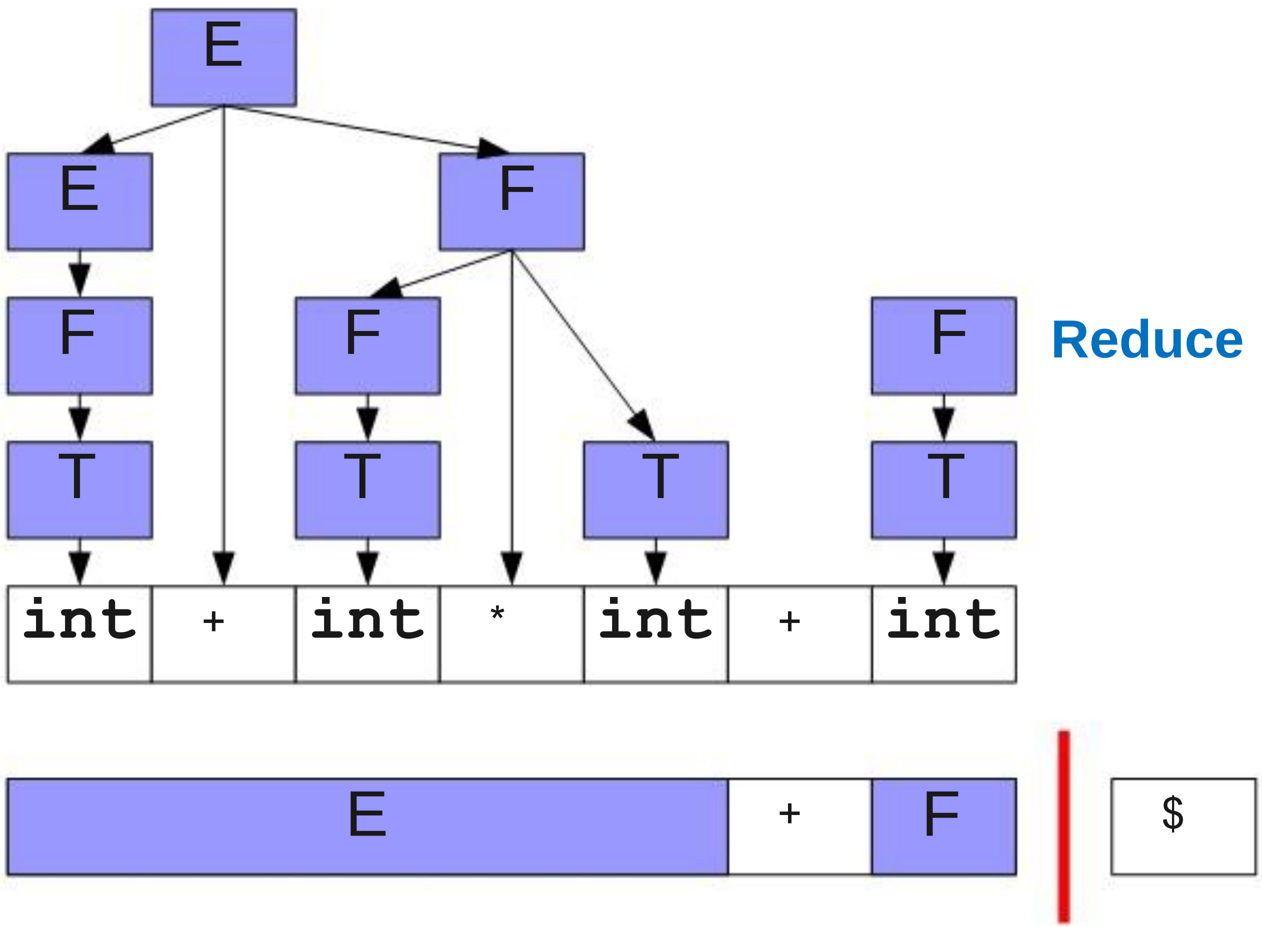
A Sample Shift/Reduce Parse

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



A Sample Shift/Reduce Parse

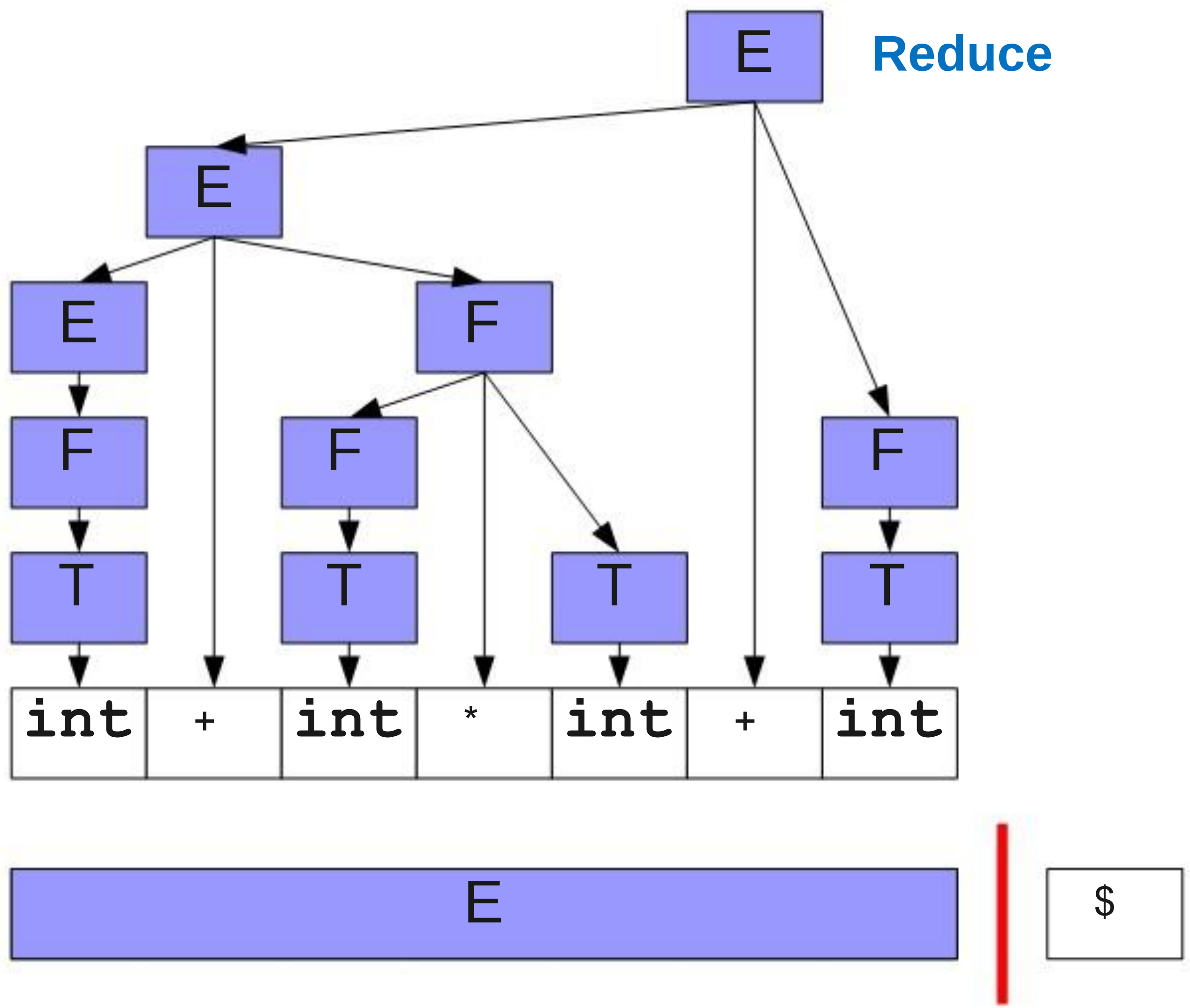
$S \rightarrow E\$$
$E \rightarrow F$
$E \rightarrow E + F$
$F \rightarrow F * T$
$F \rightarrow T$
$T \rightarrow$ **int**
$T \rightarrow (E)$



Reduce

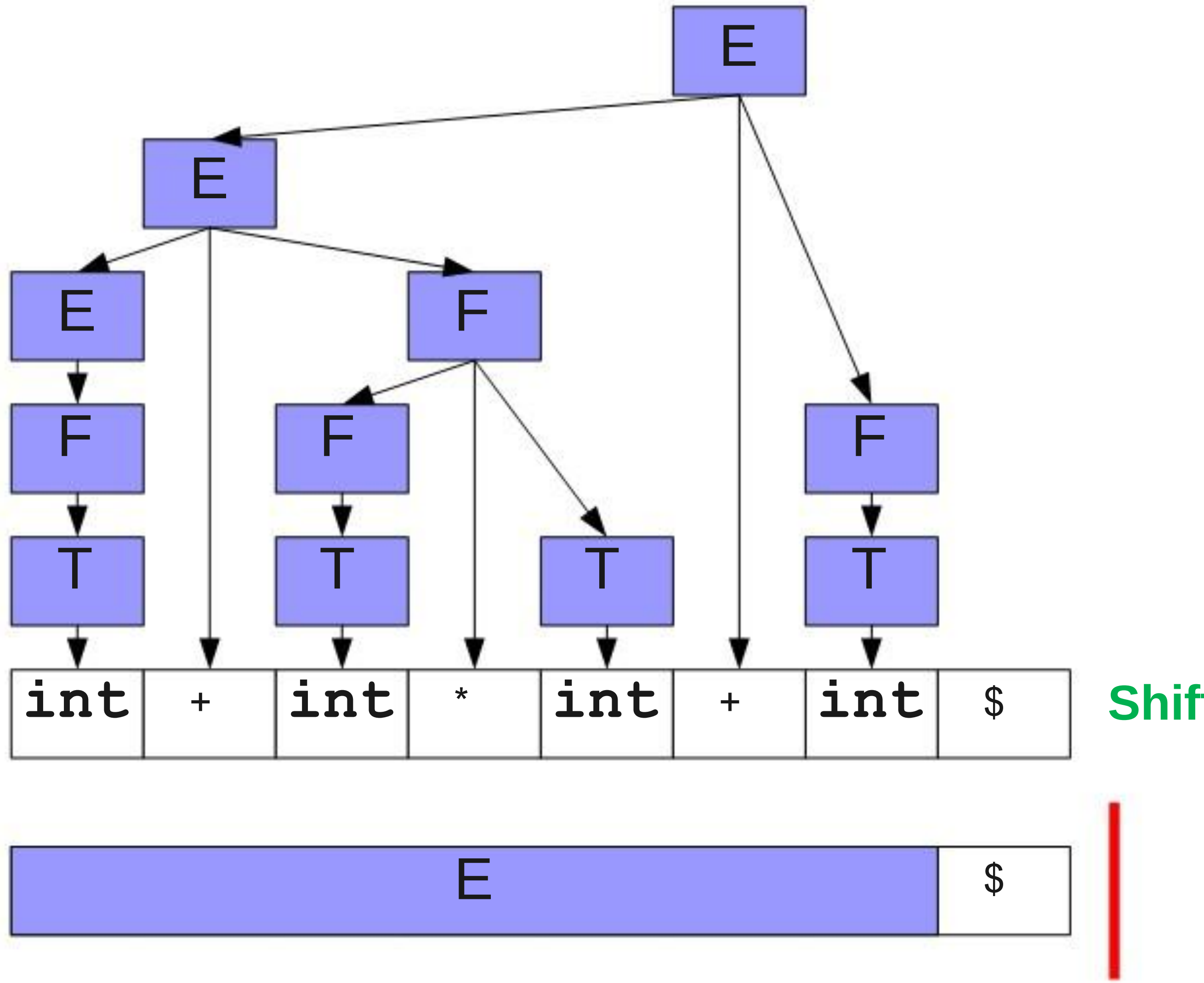
A Sample Shift/Reduce Parse

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



A Sample Shift/Reduce Parse

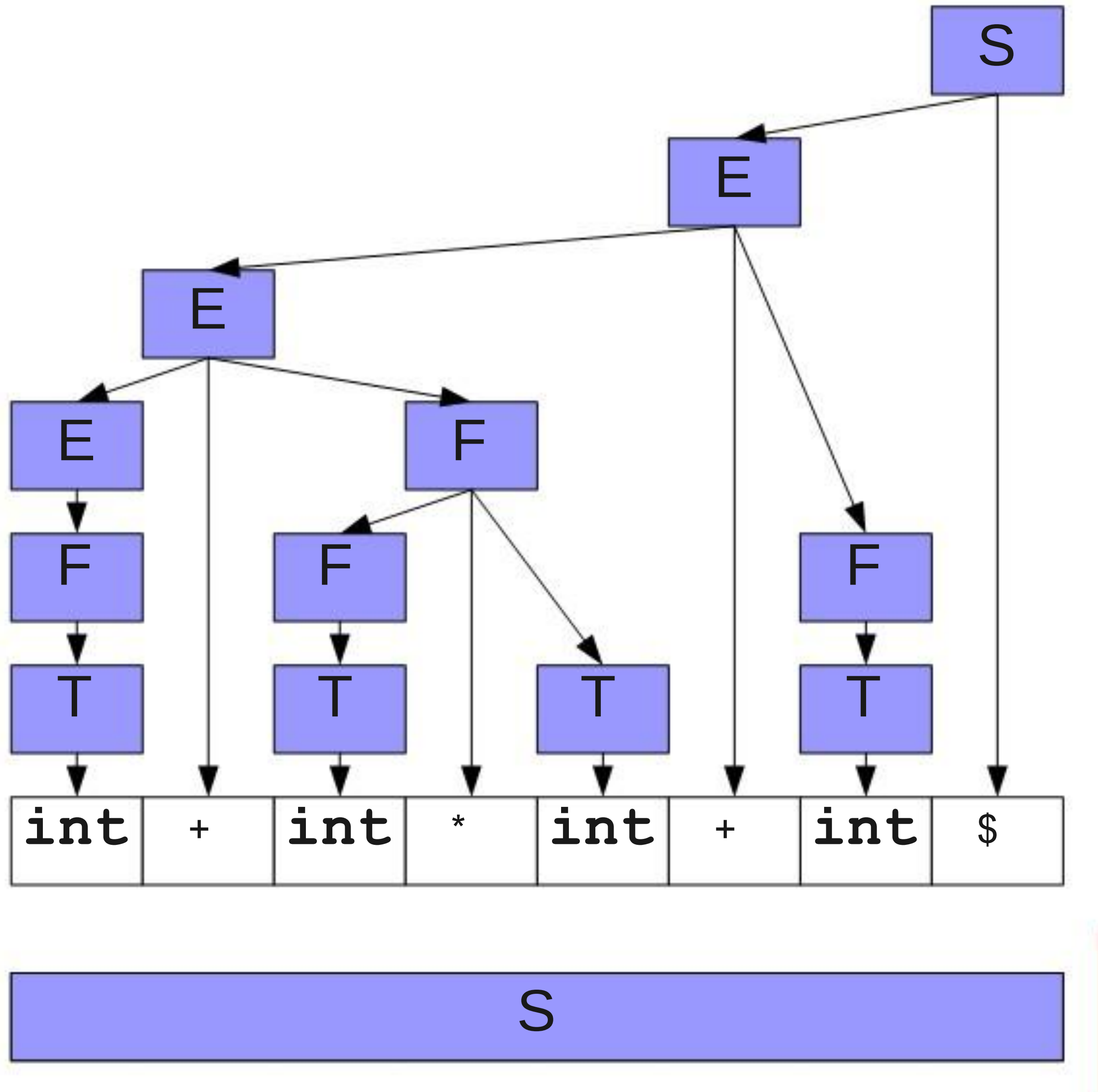
$S \rightarrow E\$$
$E \rightarrow F$
$E \rightarrow E + F$
$F \rightarrow F * T$
$F \rightarrow T$
$T \rightarrow$ **`int`**
$T \rightarrow (E)$



A Sample Shift/Reduce Parse

Reduce

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



Accept

Steps in shift/reduce parsing

- **Shift**: Move a terminal from the right to the left area.
- **Reduce**: Replace some symbols at the right side of the left area.

By representing the left area as a **stack**.

- **Shift** : Push the next terminal onto the stack.
- **Reduce**: Pop some number of symbols from the stack, then push the appropriate nonterminal.

When to Reduce, When to Shift

- **Reduce** when we have a **handle** at the top of stack.
- **Shift** , otherwise.

Right-hand side

- **Handle** is the substring that matches the rhs of a production whose reduction to the non-terminal on the lhs represents the reverse step of the right-most derivation.

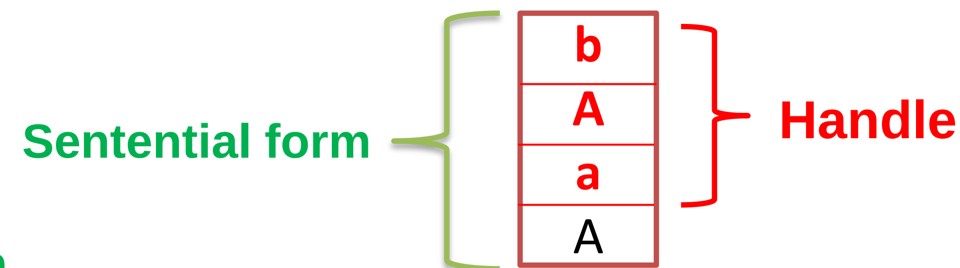
Left-hand side

Example

$S \rightarrow A$

$A \rightarrow AB \mid B$

$B \rightarrow aAb \mid ab$



Sentential form

$$S \Rightarrow A \Rightarrow AB \Rightarrow \boxed{A\ \underbrace{aAb}_{\text{Handle}}} \Rightarrow \cdots$$

Bottom-up parsing

(Rightmost Derivation)

- **Viable prefixes** is the set of all possible prefixes of the right sentential forms that can appear on the top of the parsing stack.

Example: **Viable prefixes** of AaAb = **{** ε, **A, Aa, AaA, AaAb }**

The use of Viable prefix and Handle

Shift-Reduce Parser

- **Shift** if doing so leaves the stack containing a viable prefix.
- Otherwise, **reduce** if we have found a handle and **error** if the input is malformed.

Facts

- The set of viable prefixes for any grammar is a **regular language**.
- Therefore, we can build a **finite automata** to handle them.

A Deterministic Automaton

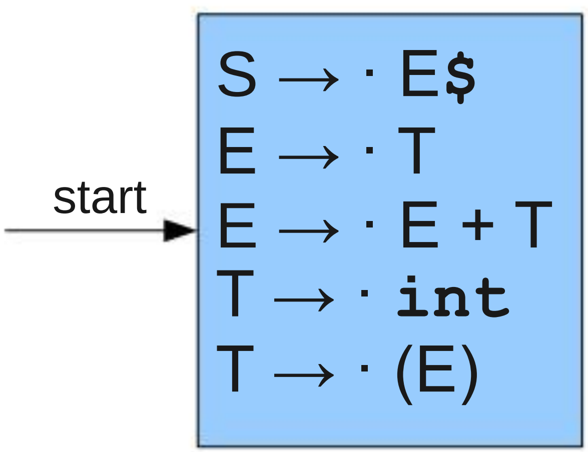
$S \rightarrow E\$$

$E \rightarrow T$

$E \rightarrow E + T$

$T \rightarrow$ **`int`**

$T \rightarrow (E)$



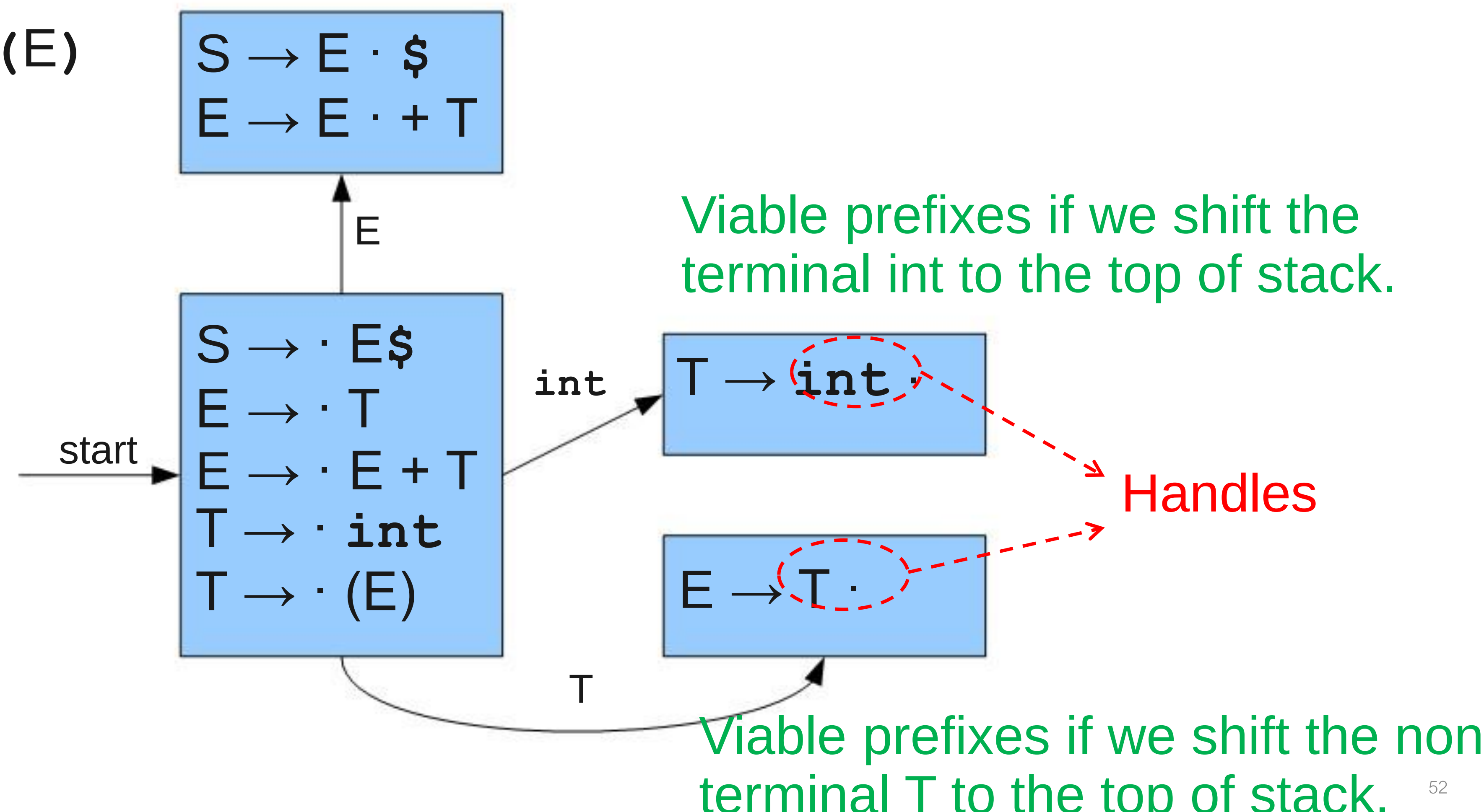
These are viable prefixes if we shift an empty string to the top of stack.

Note: Dot (·) represents a split point

A Deterministic Automaton

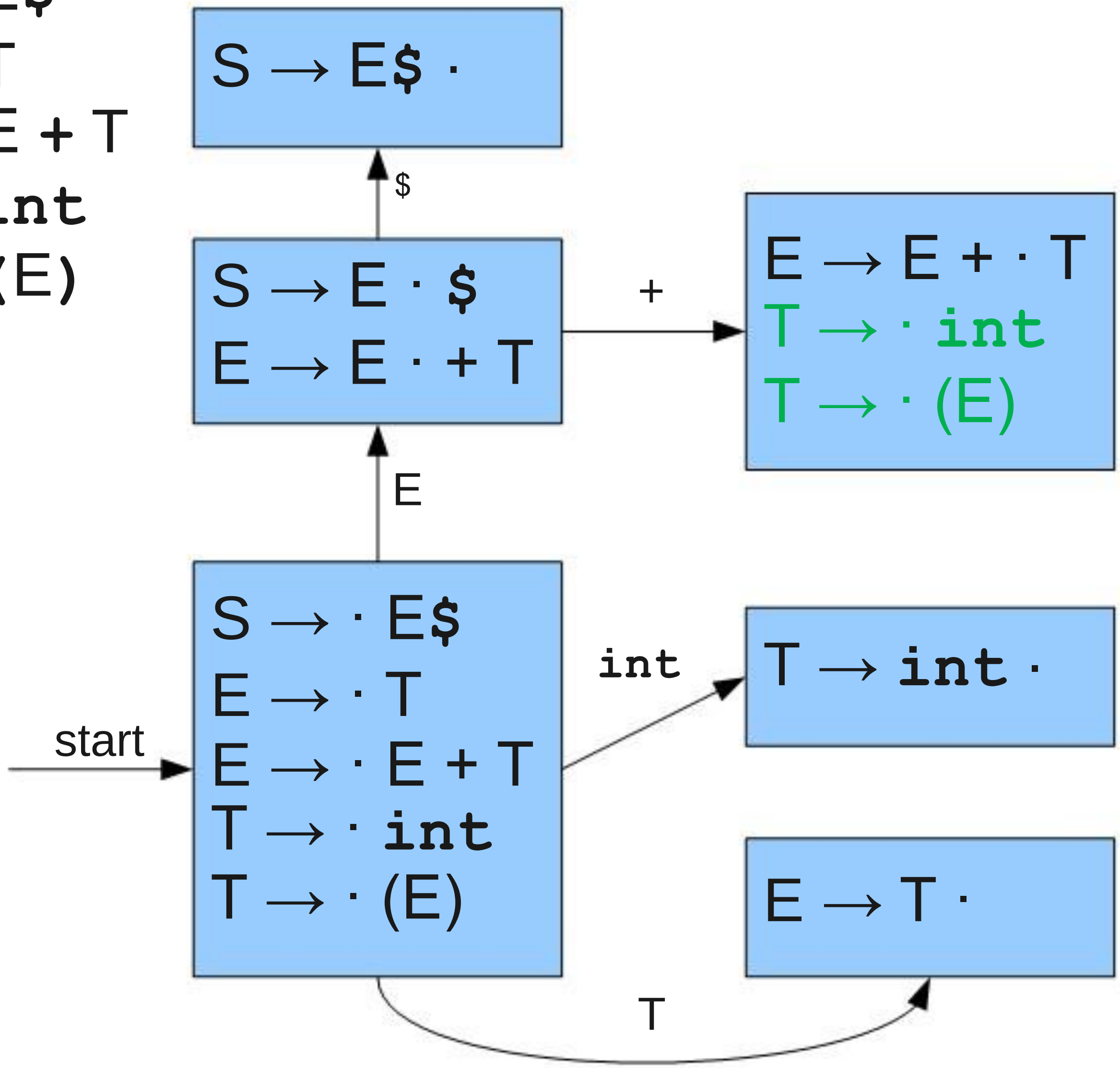
$S \rightarrow E\$$
$E \rightarrow T$
$E \rightarrow E + T$
$T \rightarrow \texttt{int}$
$T \rightarrow (E)$

Viable prefixes if we shift the non-terminal E to the top of stack.



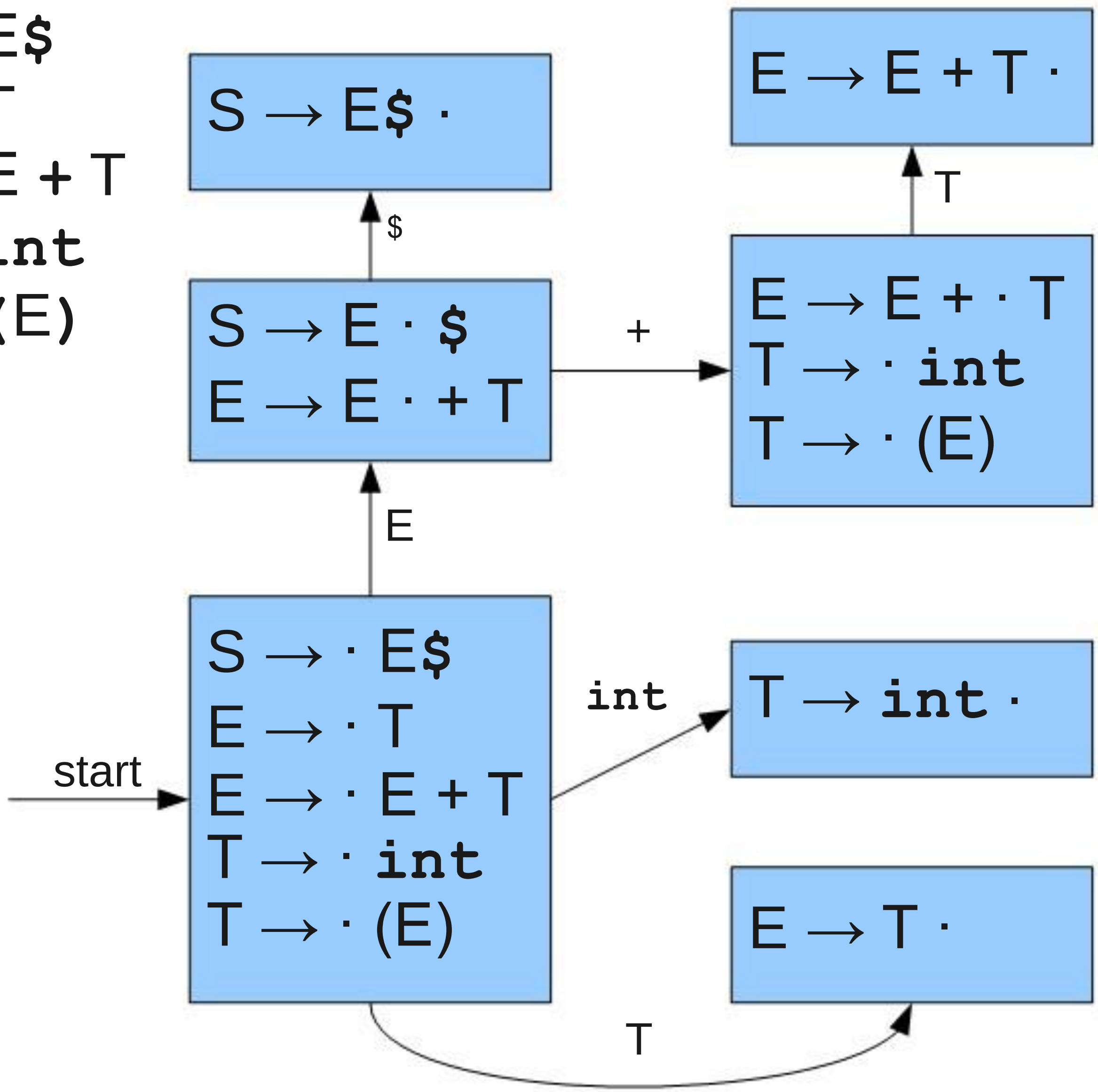
A Deterministic Automaton

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



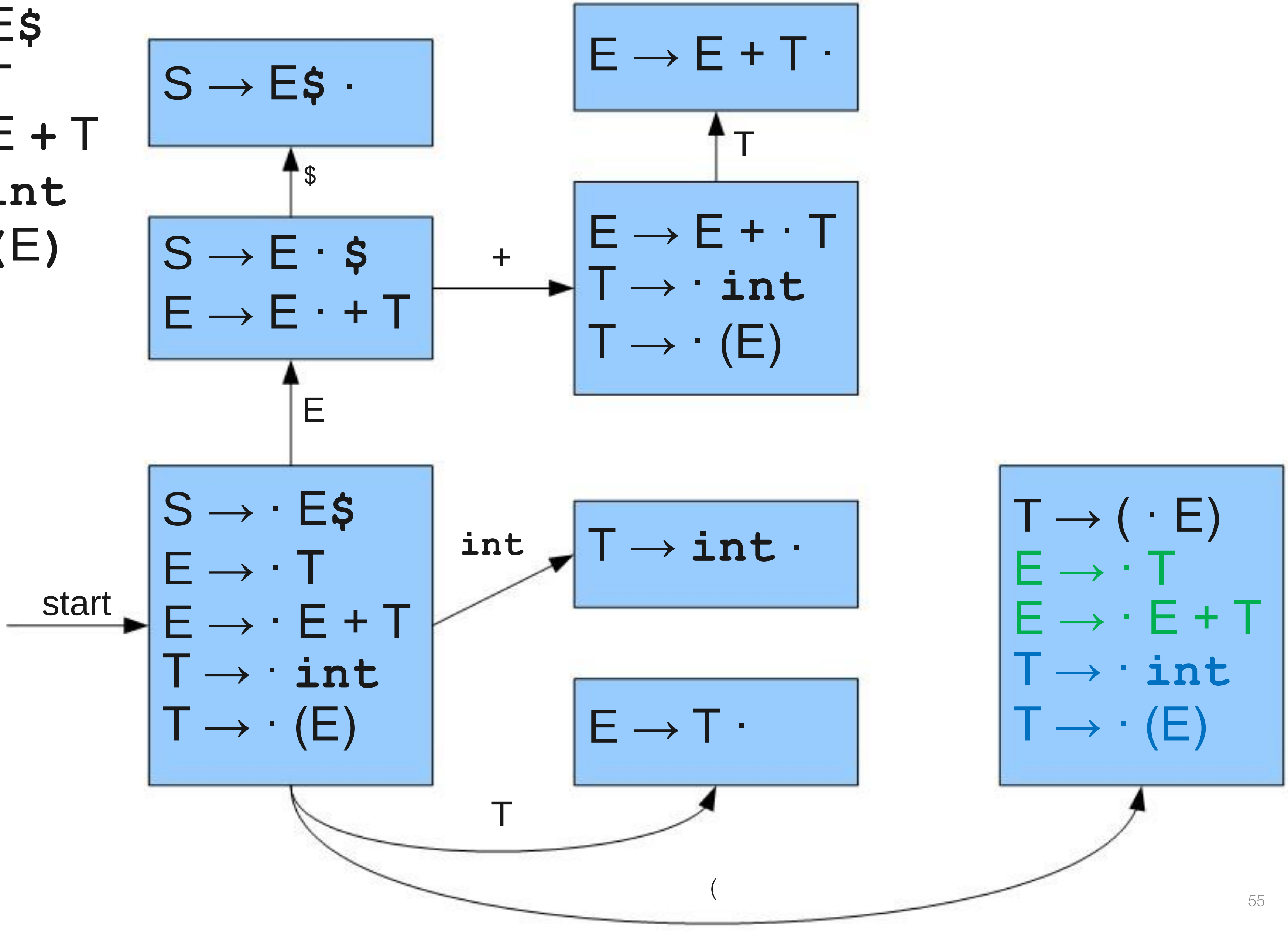
A Deterministic Automaton

$S \to E\$$
$E \to T$
$E \to E + T$
$T \to \texttt{int}$
$T \to (E)$



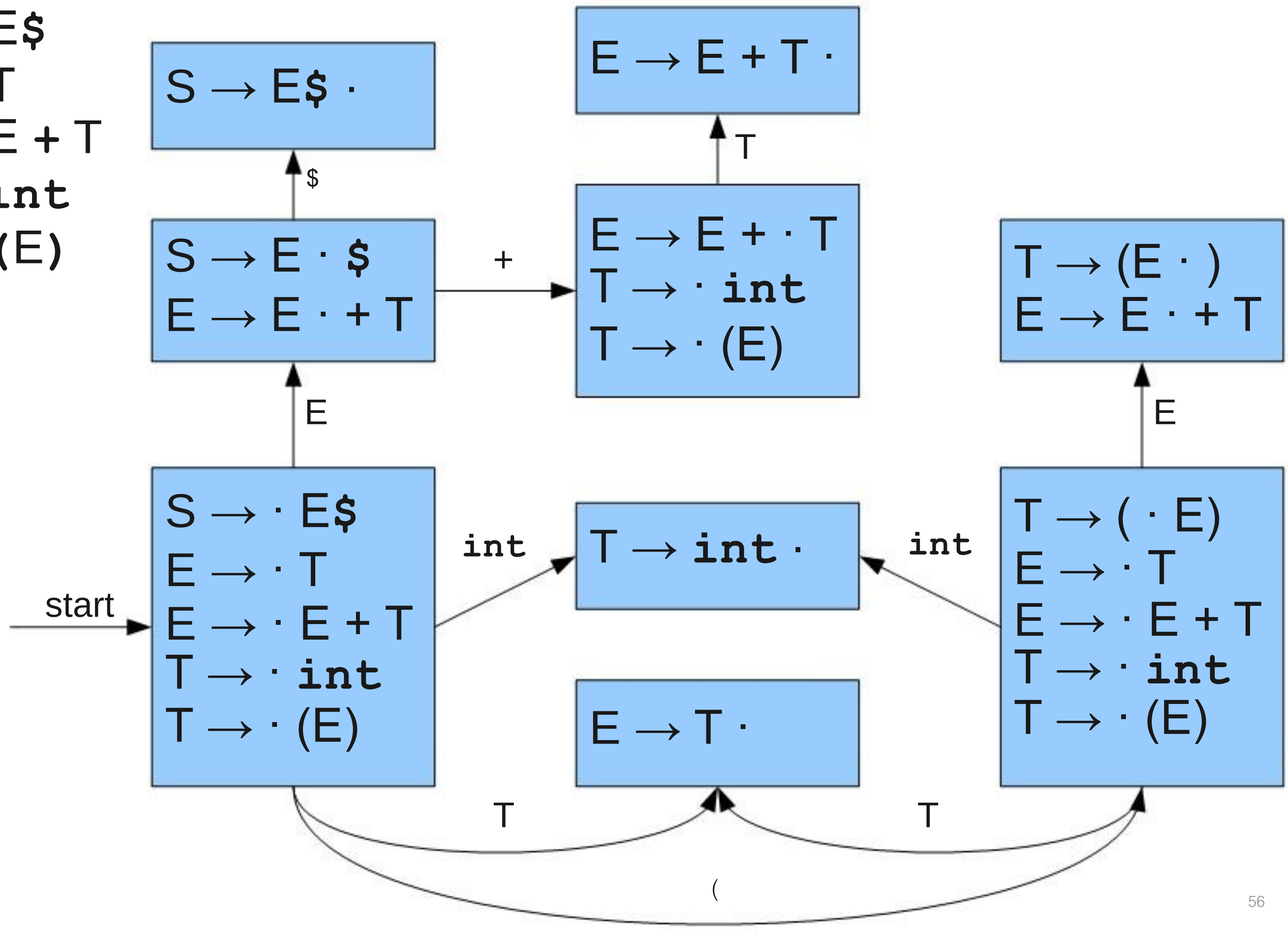
A Deterministic Automaton

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



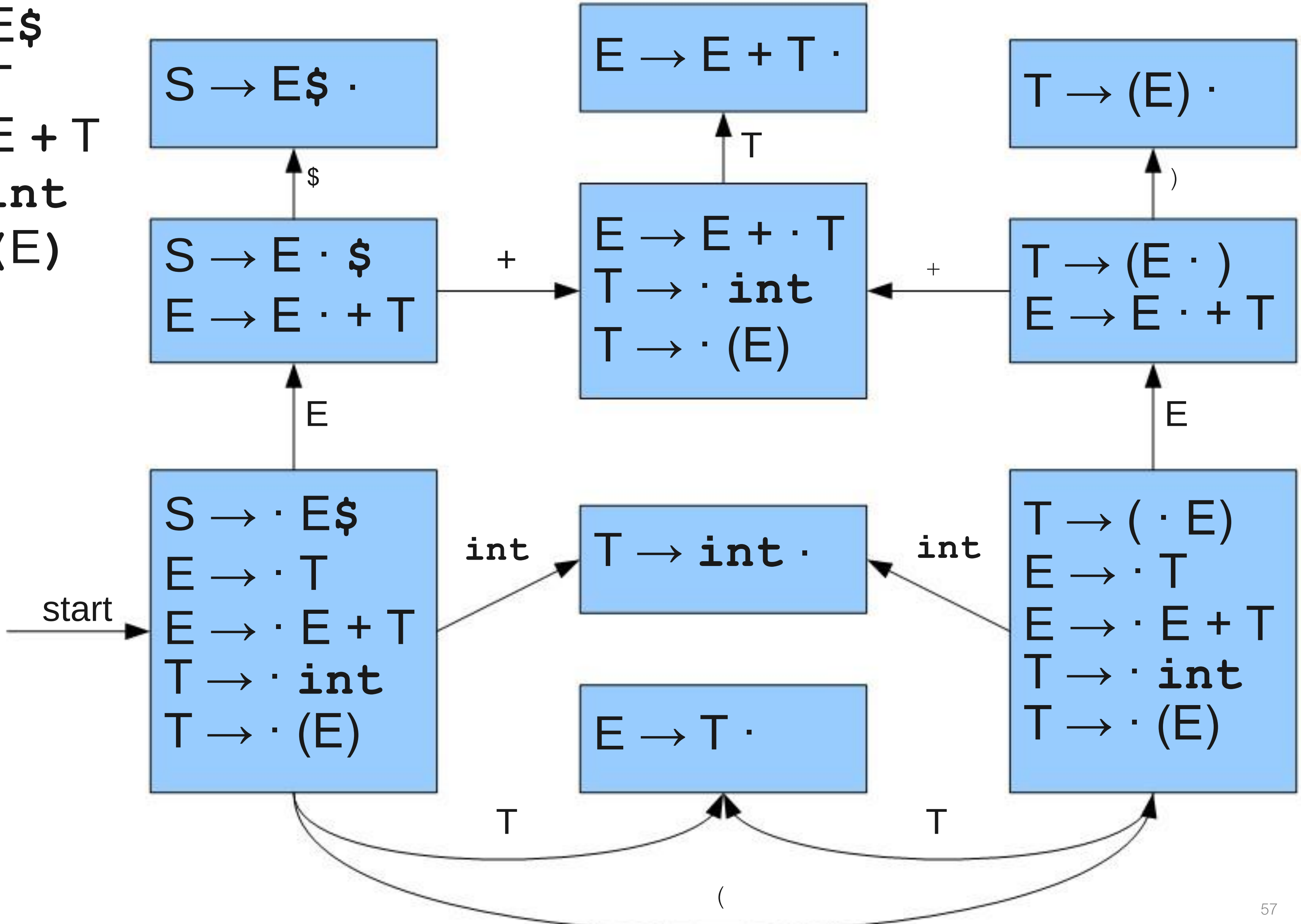
A Deterministic Automaton

$S \to E\$$
$E \to T$
$E \to E + T$
$T \to \texttt{int}$
$T \to (E)$



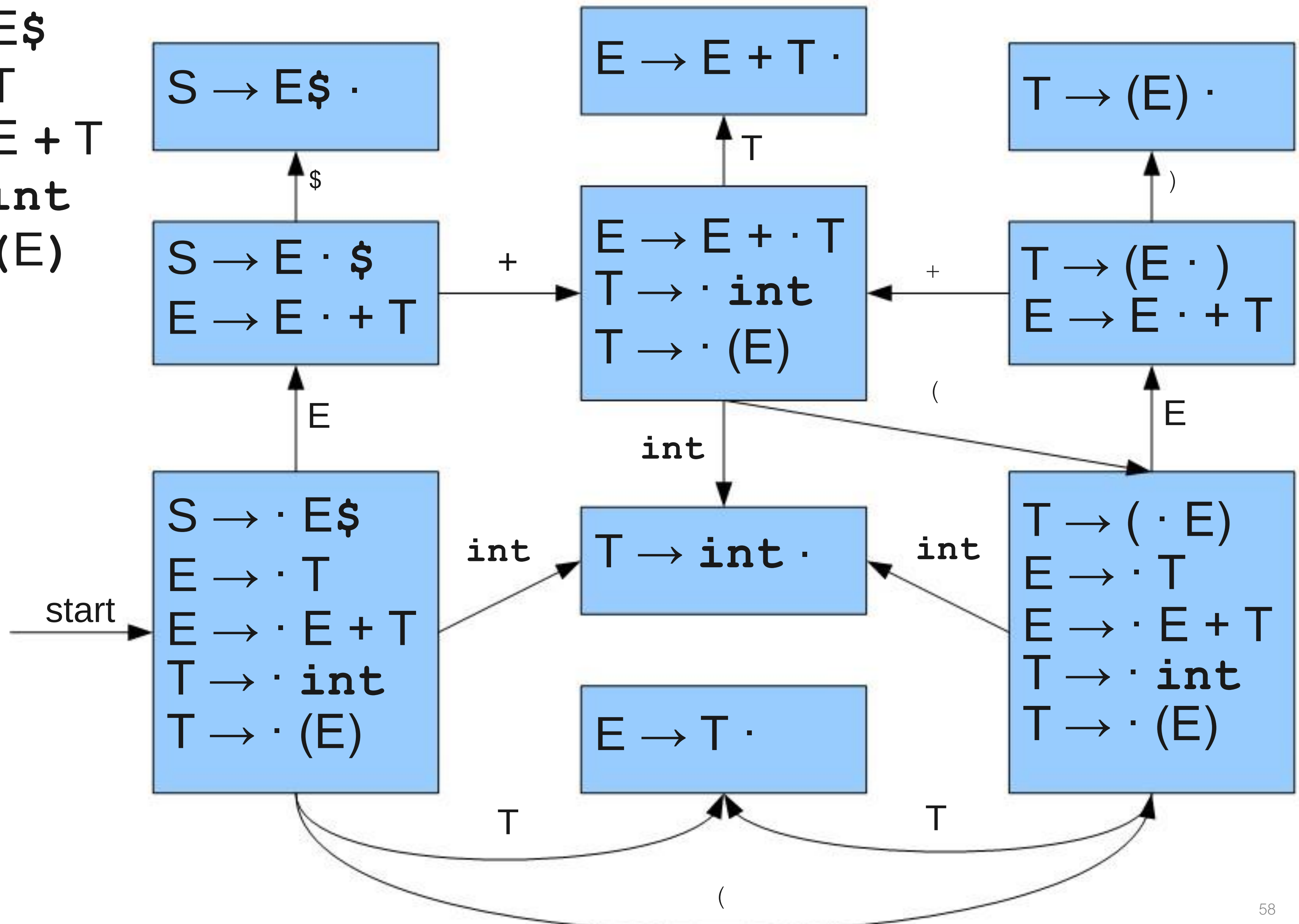
A Deterministic Automaton

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



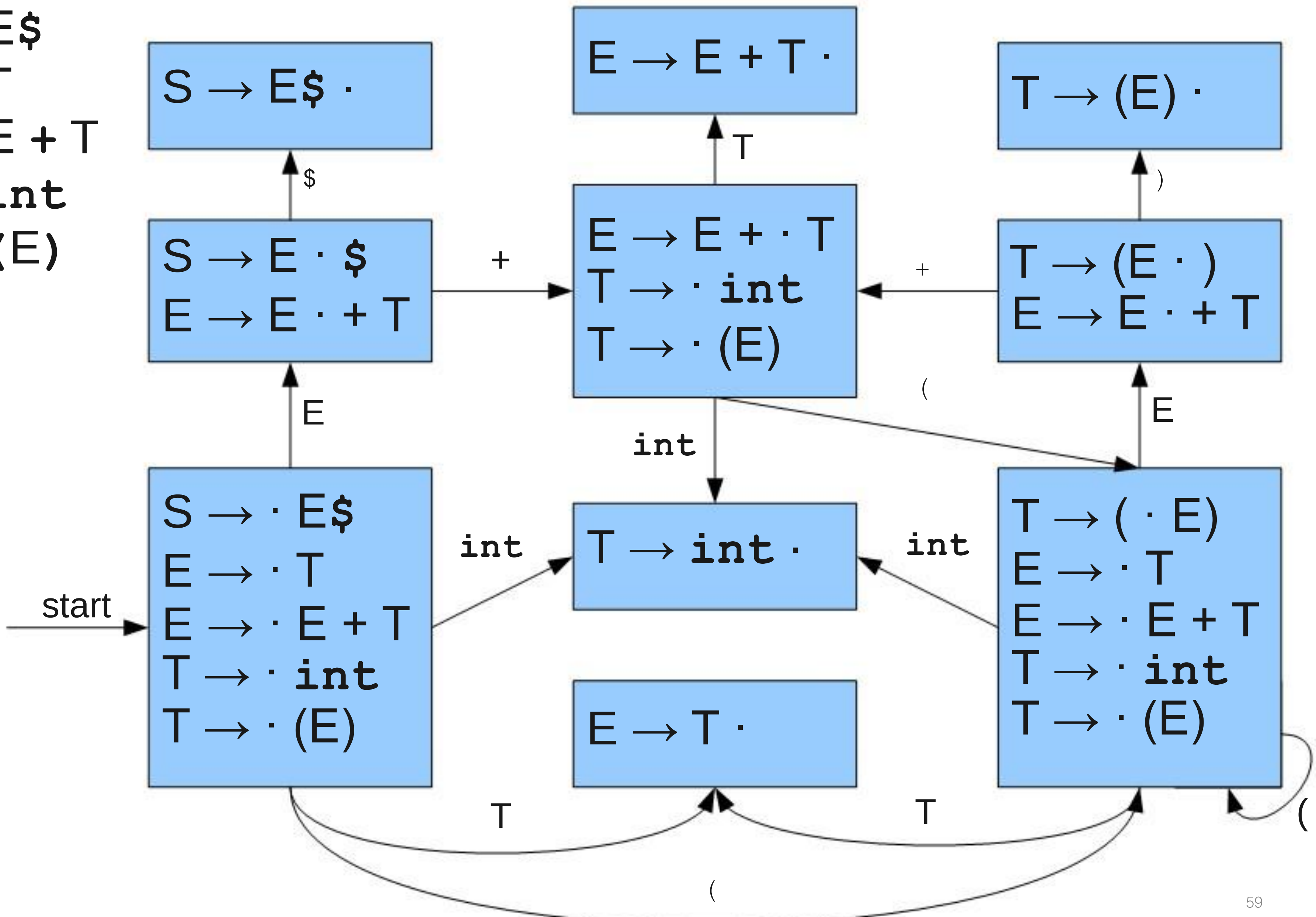
A Deterministic Automaton

$S \to E\$$
$E \to T$
$E \to E + T$
$T \to \texttt{int}$
$T \to (E)$

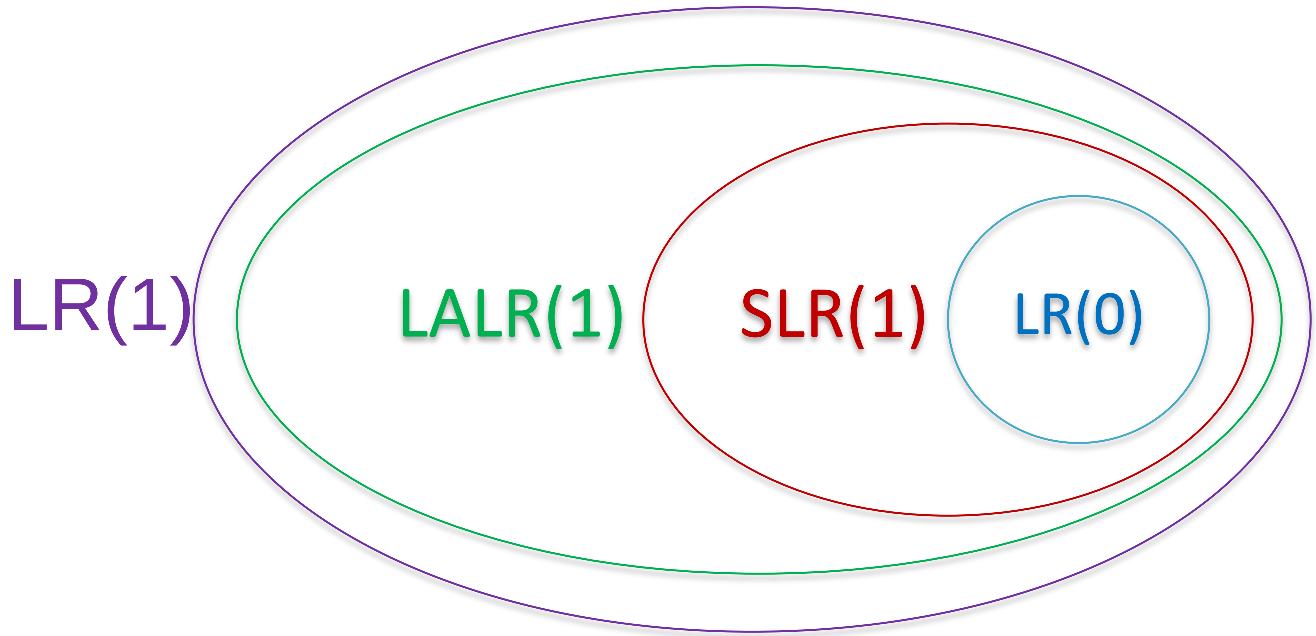


A Deterministic Automaton

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



Hierarchy of Shift/Reduce Parser

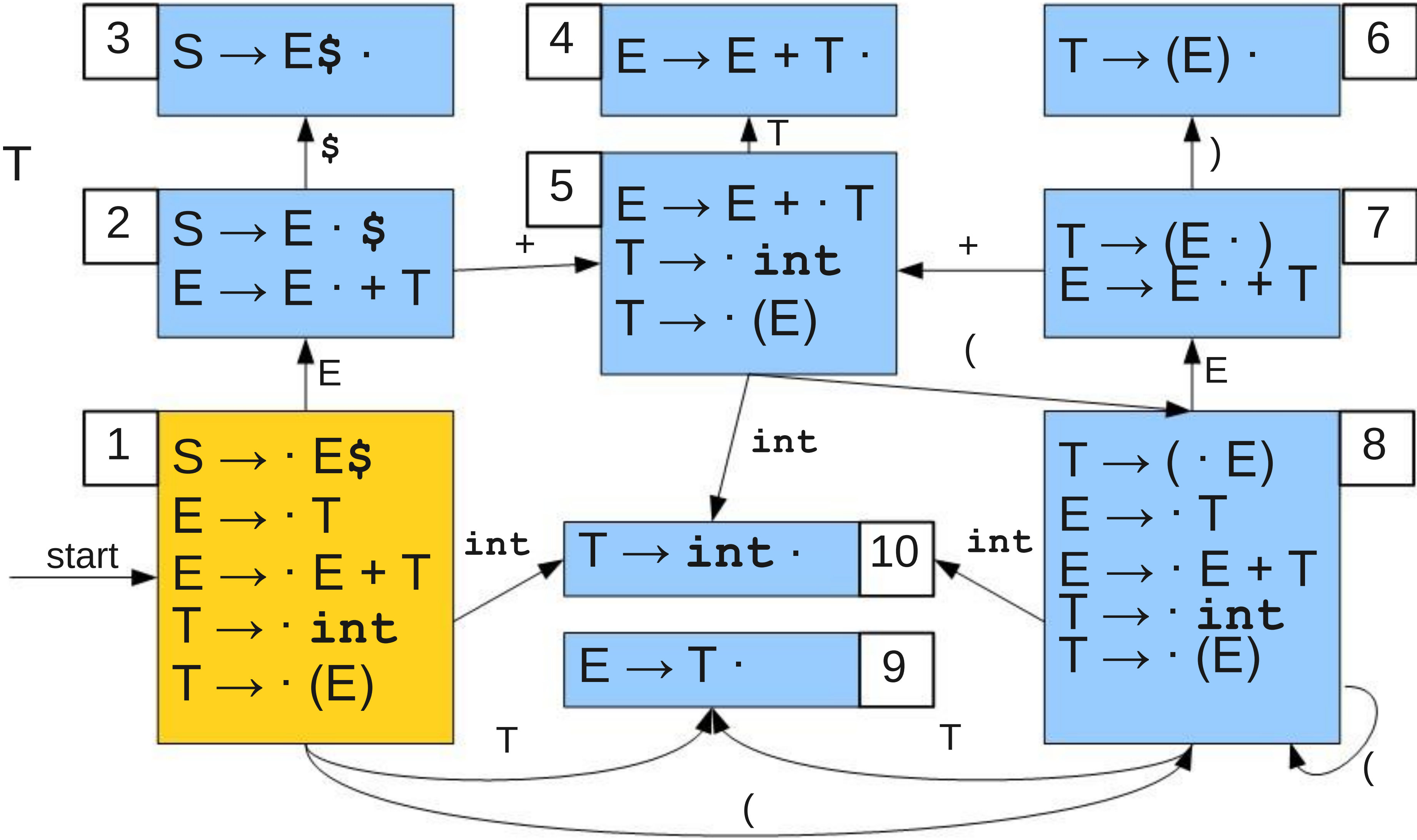


LR(0)

- Predictive bottom-up parsing with:
  - **L**: **L**eft-to-right scan of the input.
  - **R**: **R**ightmost derivation.
  - (0): Zero tokens of look ahead.
- Use the handle-finding automaton, without any lookahead, to predict where handles are.

LR(0) Parsing

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



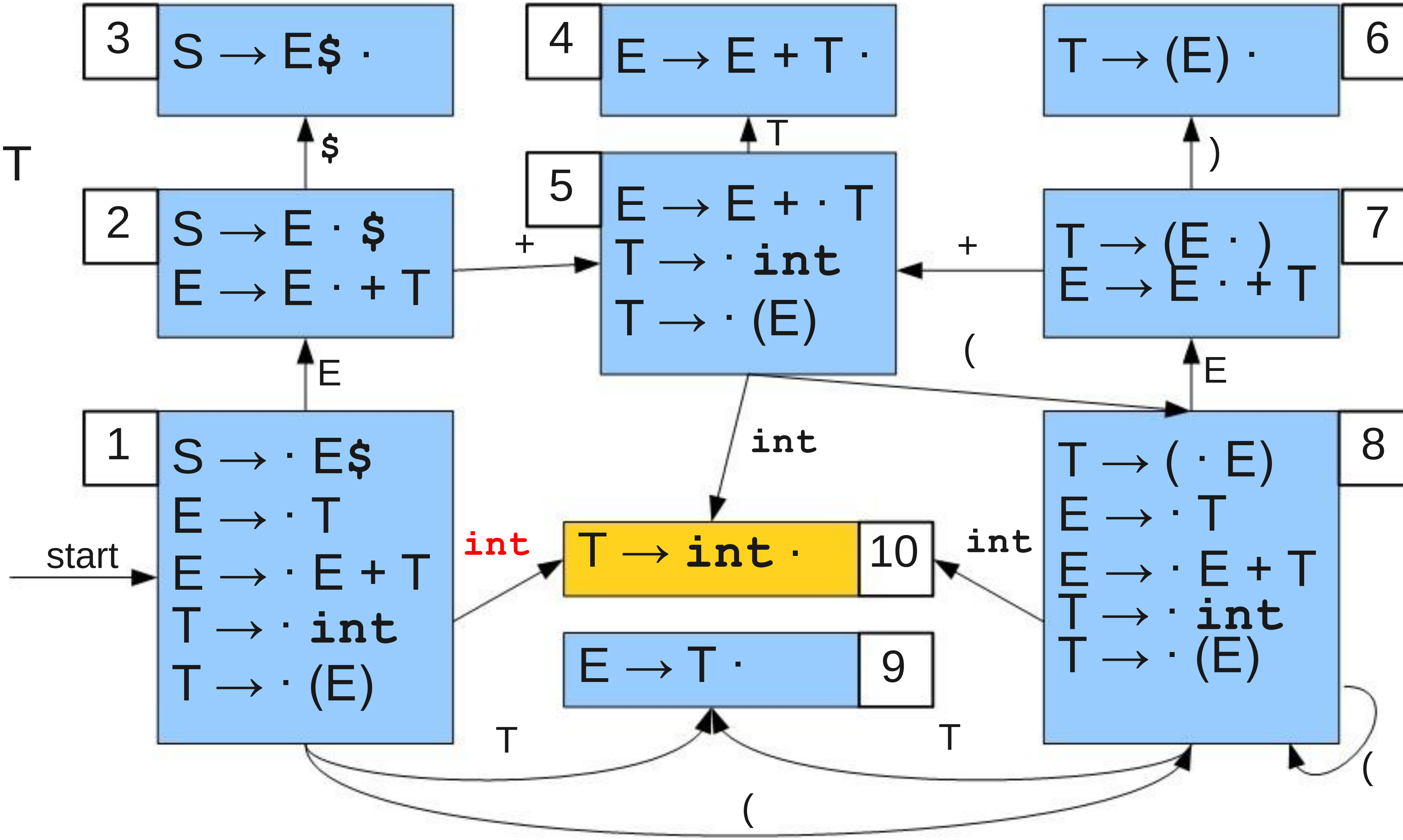
int	+	(	int	+	int	+	int	)	$

Current state

1

LR(0) Parsing

$S \to E\$$
$E \to T$
$E \to E + T$
$T \to \texttt{int}$
$T \to (E)$



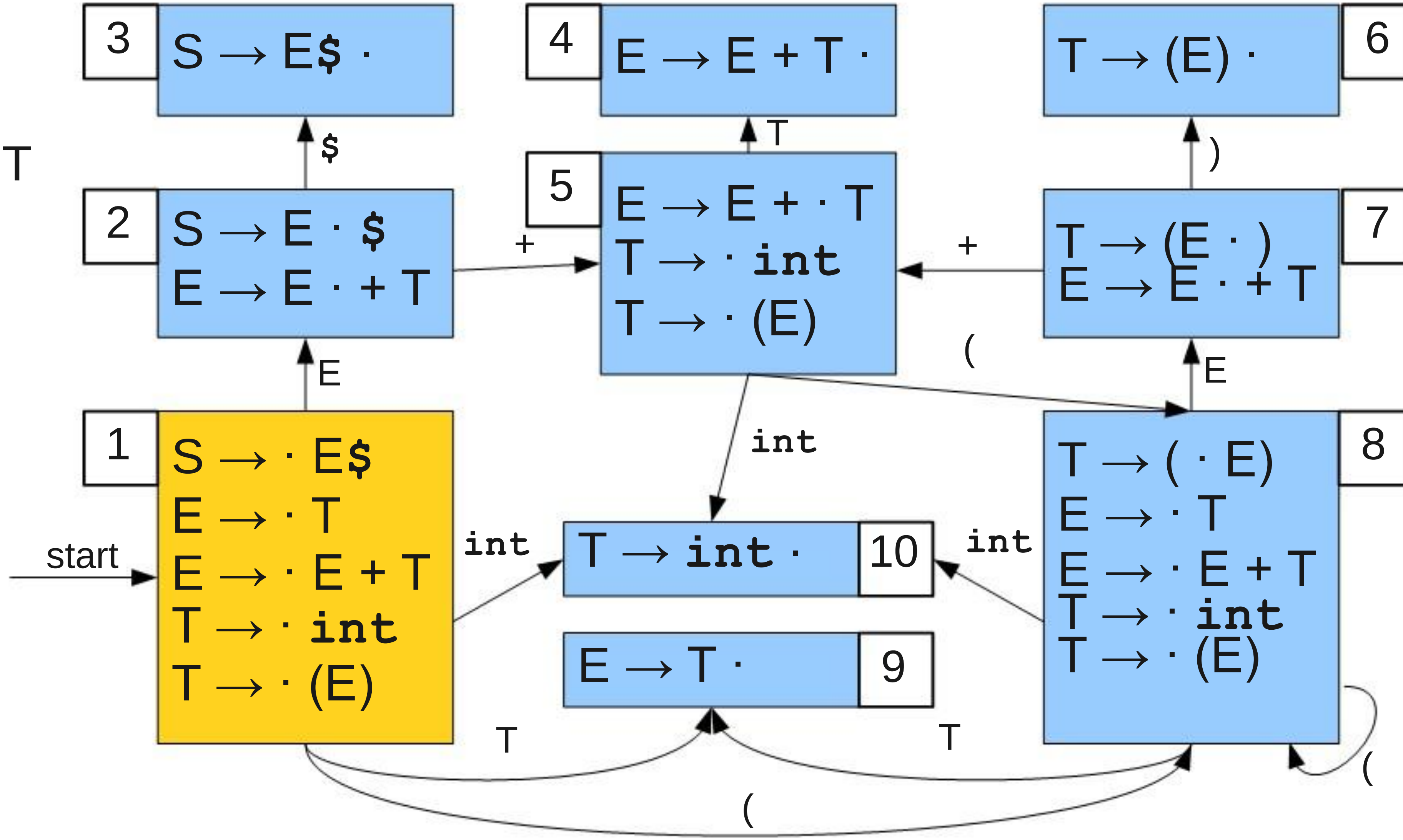
int

1	10

+	(	int	+	int	+	int	)	$

LR(0) Parsing

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



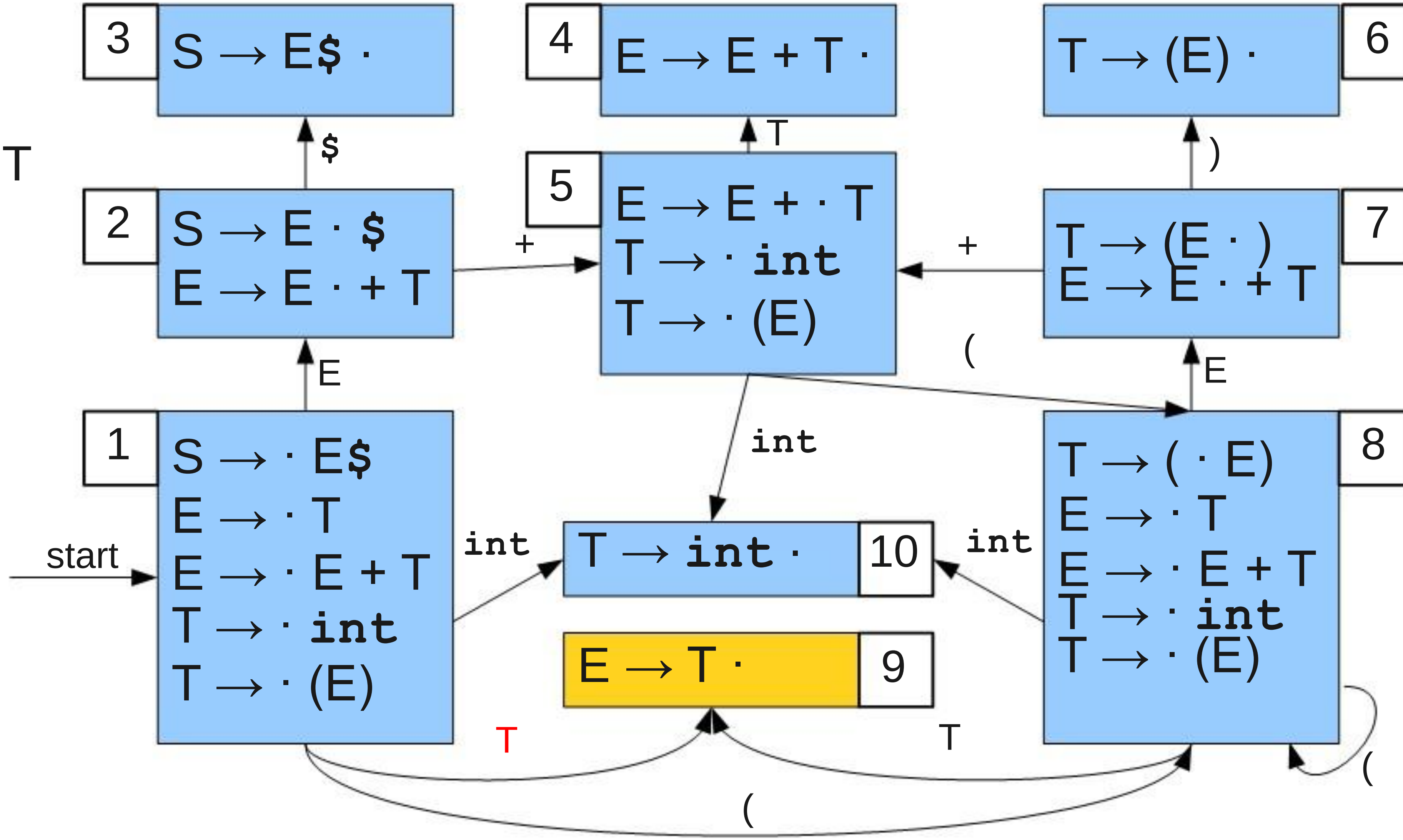
T

1

+	(	int	+	int	+	int	)	$

LR(0) Parsing

$S \to E\$$
$E \to T$
$E \to E + T$
$T \to \texttt{int}$
$T \to (E)$



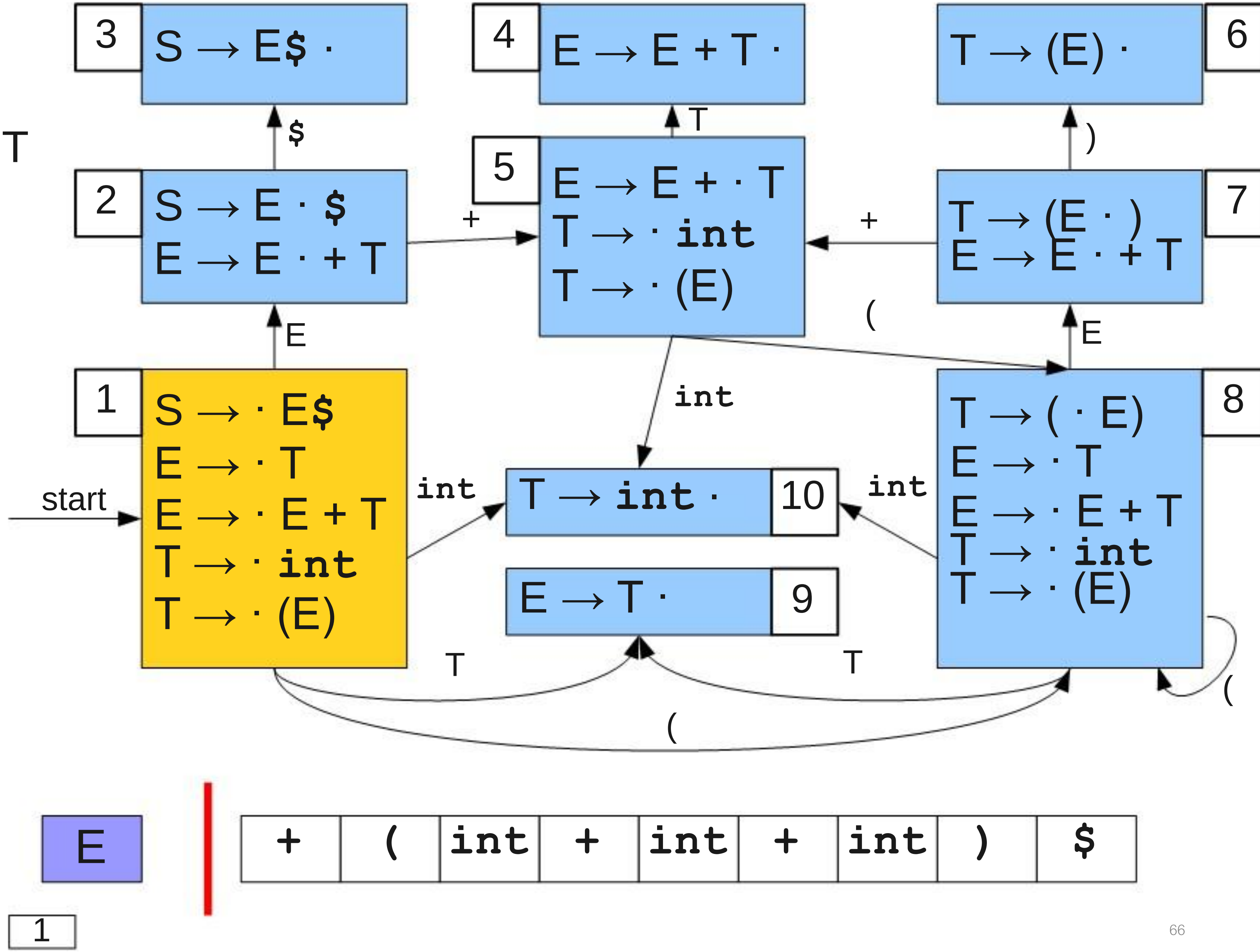
+	(	int	+	int	+	int	)	$

T

1 9

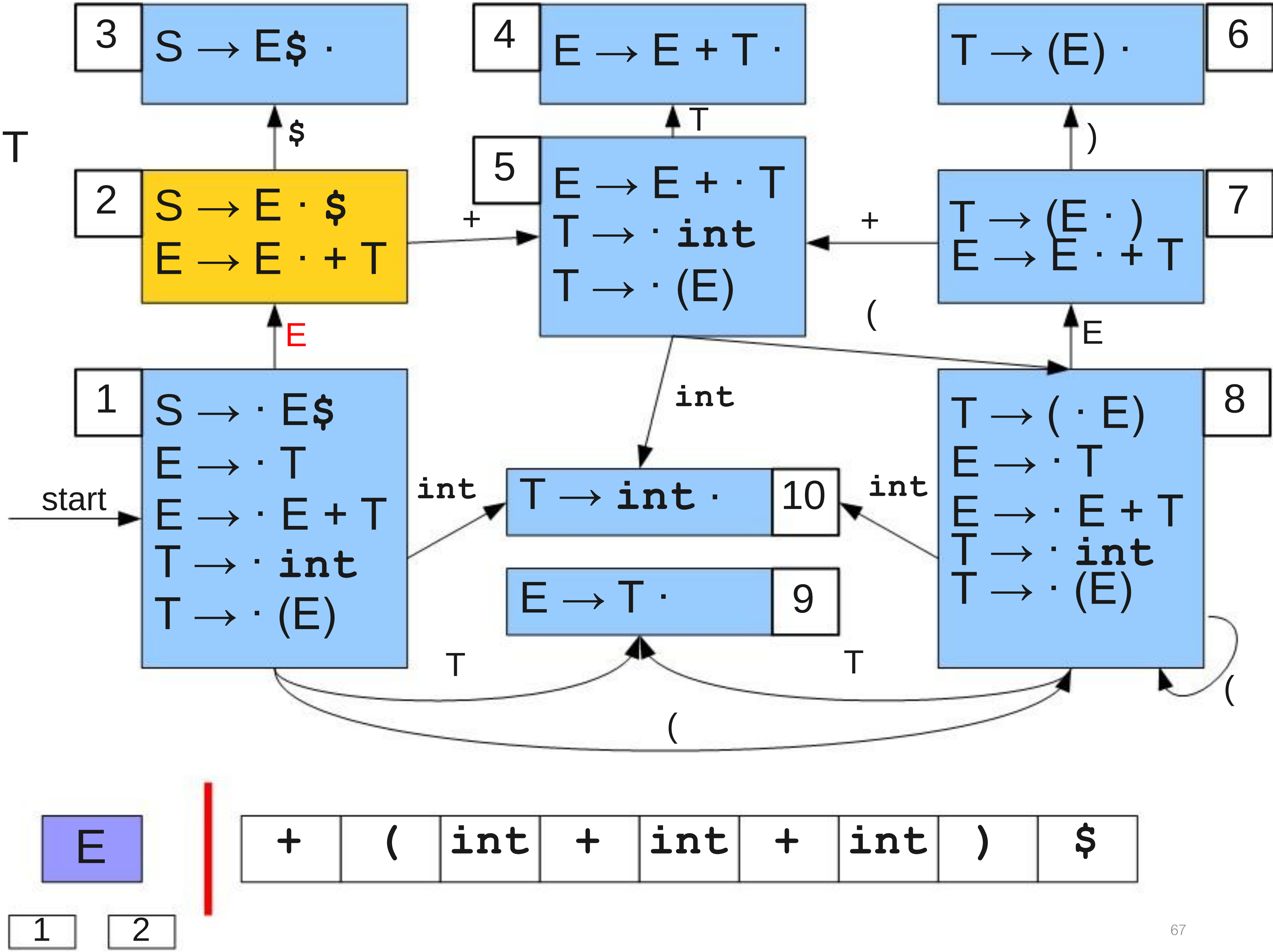
LR(0) Parsing

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



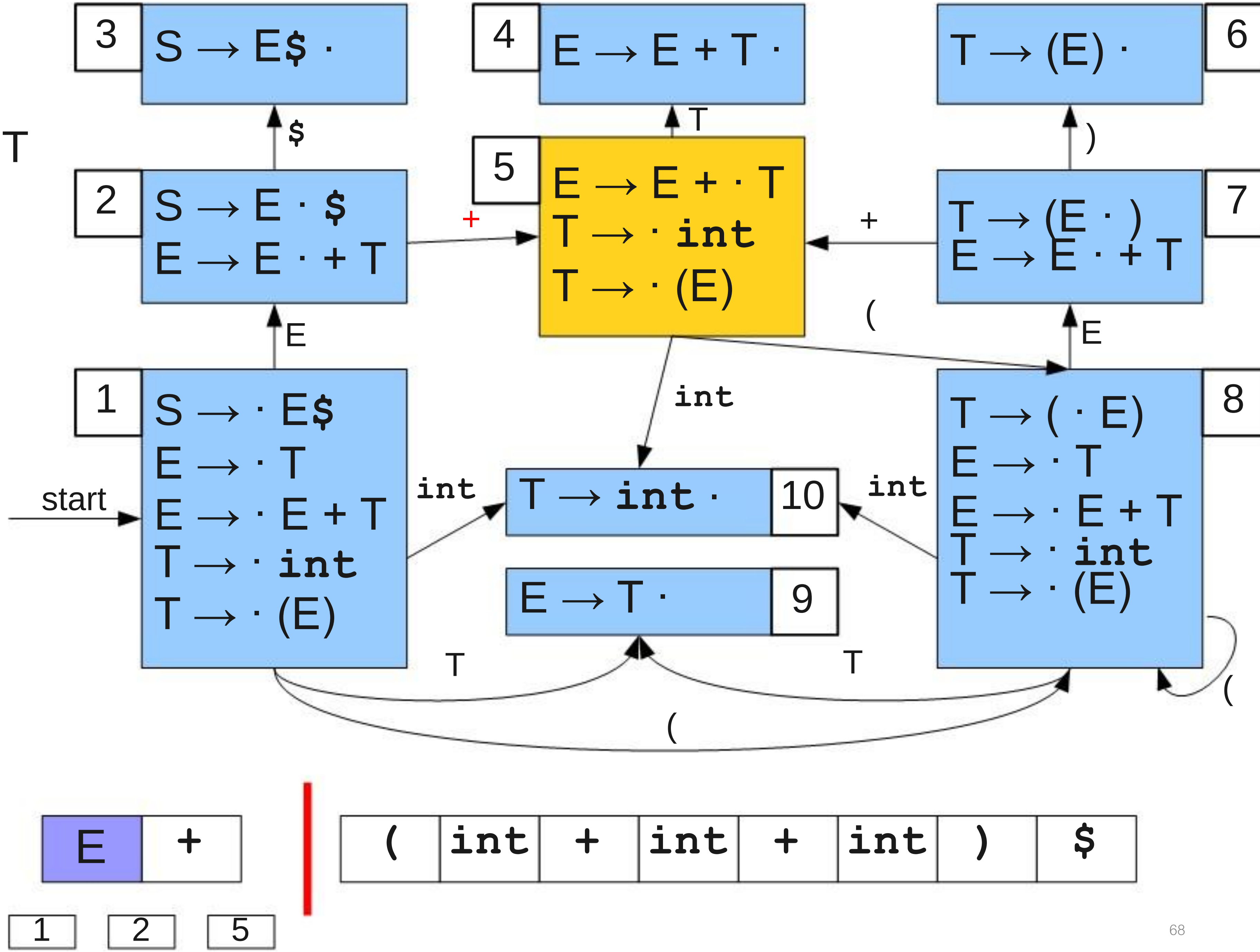
LR(0) Parsing

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



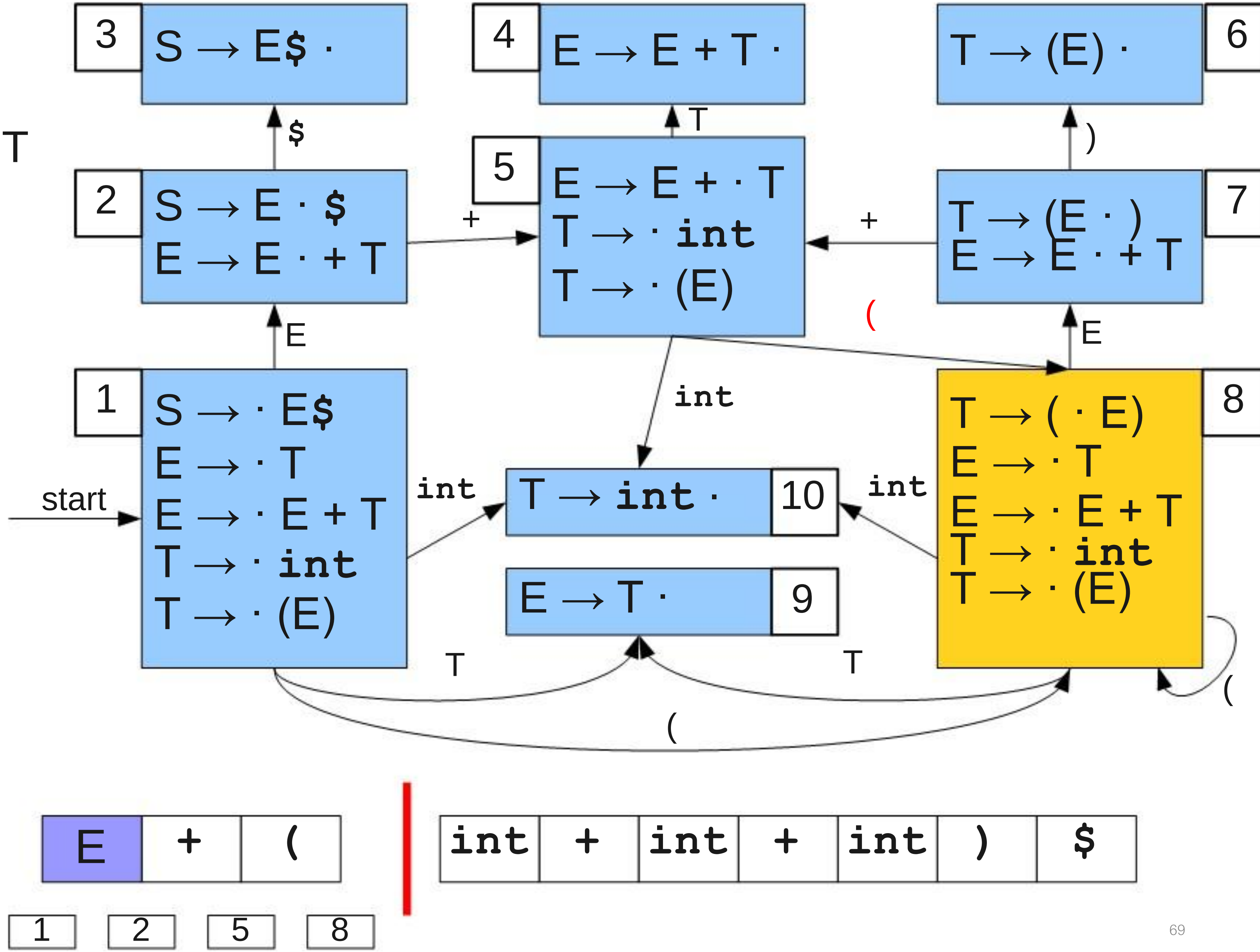
LR(0) Parsing

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



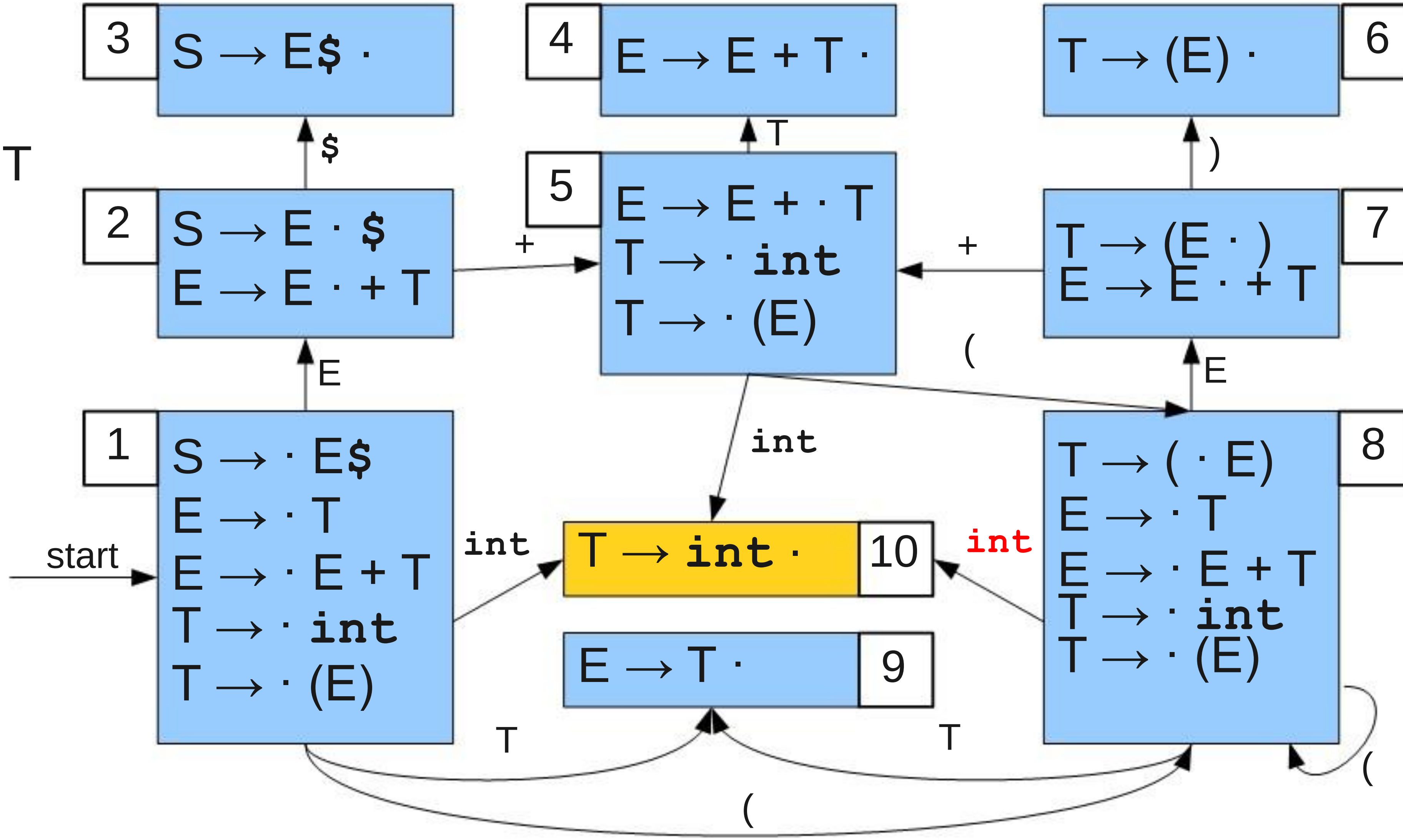
LR(0) Parsing

$S \to E\$$
$E \to T$
$E \to E + T$
$T \to \texttt{int}$
$T \to (E)$



LR(0) Parsing

$S \to E\$$
$E \to T$
$E \to E + T$
$T \to \texttt{int}$
$T \to (E)$



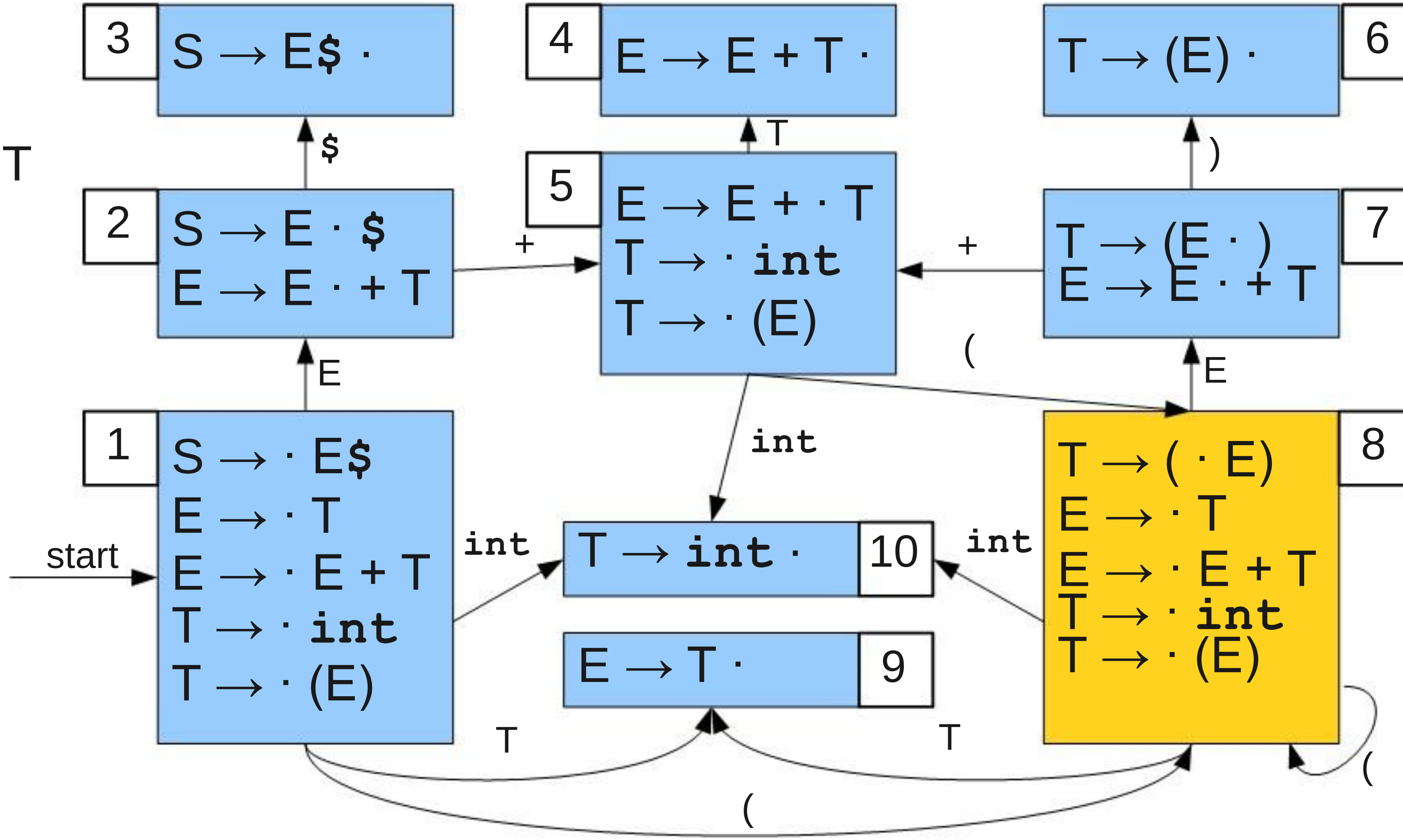
E	+	(	int
1	2	5	8

10

+	int	+	int	)	$

LR(0) Parsing

$S \to E\$$
$E \to T$
$E \to E + T$
$T \to \texttt{int}$
$T \to (E)$



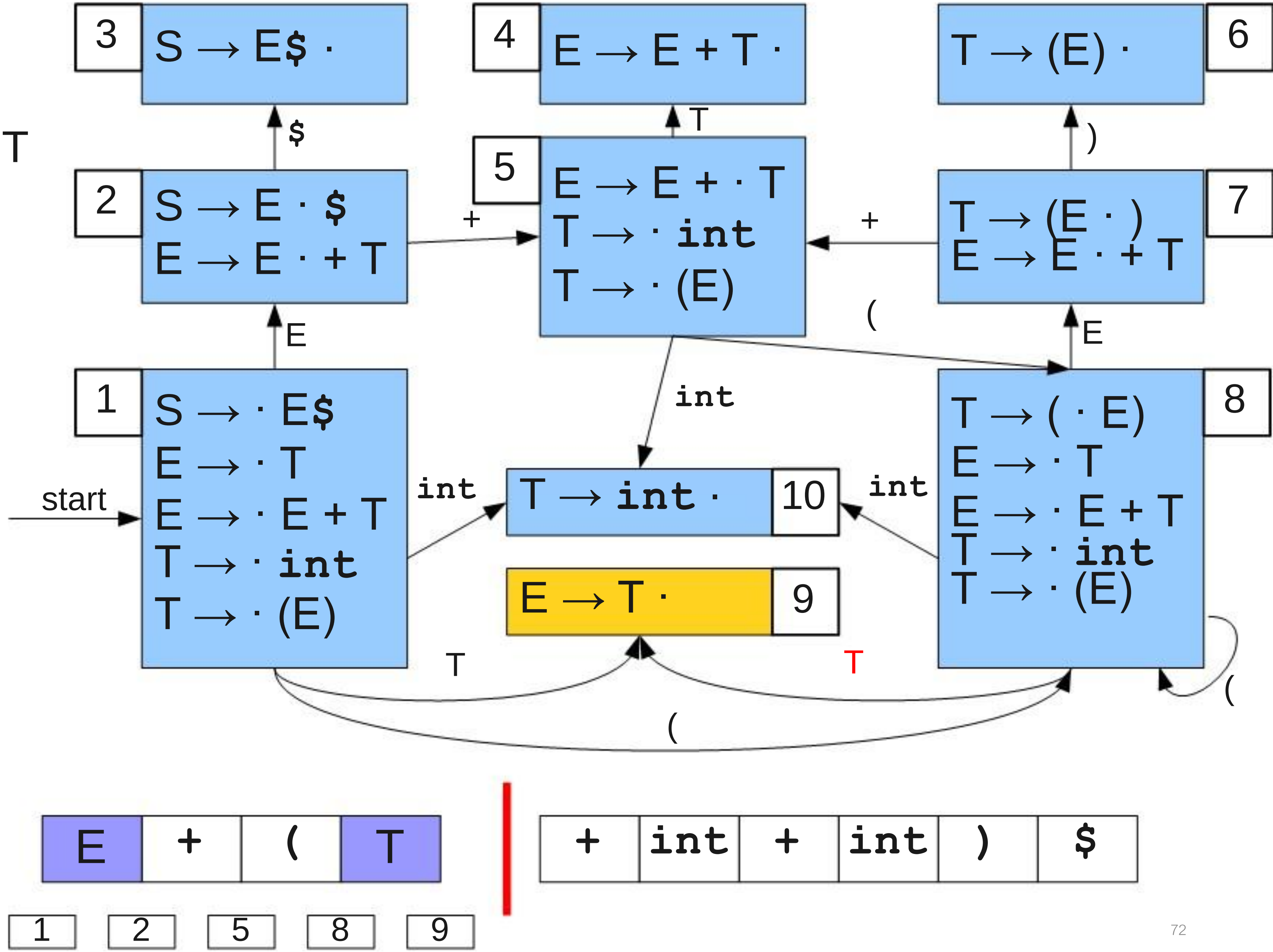
E	+	(

1	2	5	8

+	int	+	int	)	$

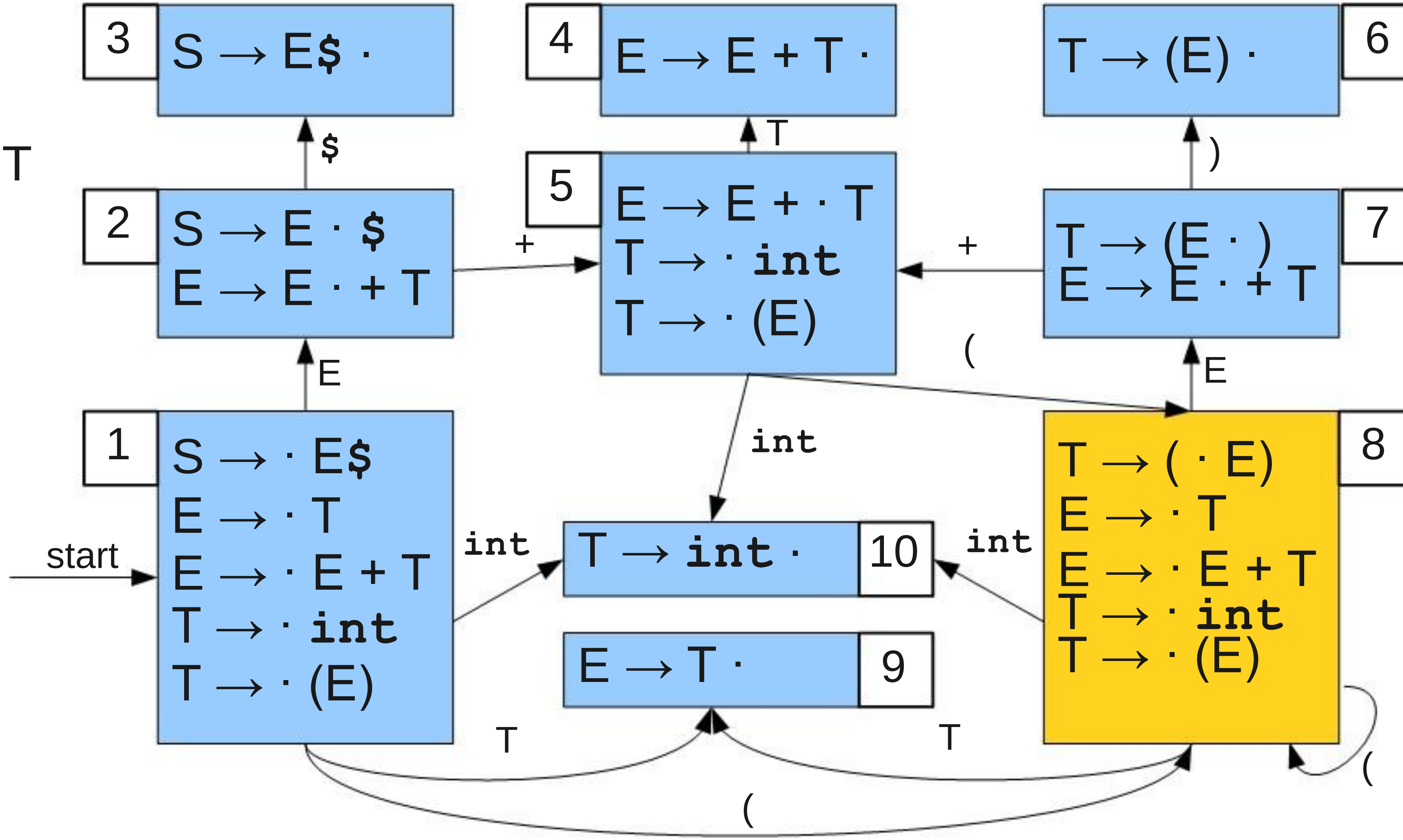
LR(0) Parsing

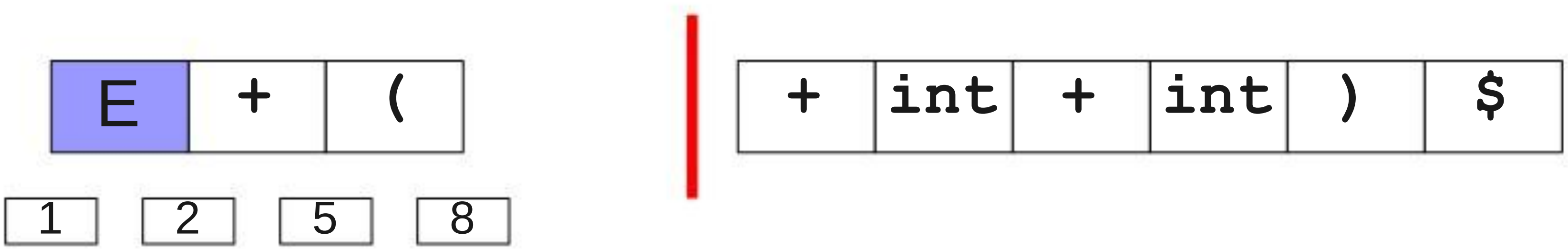
S → E$
E → T
E → E + T
T → int
T → (E)



LR(0) Parsing

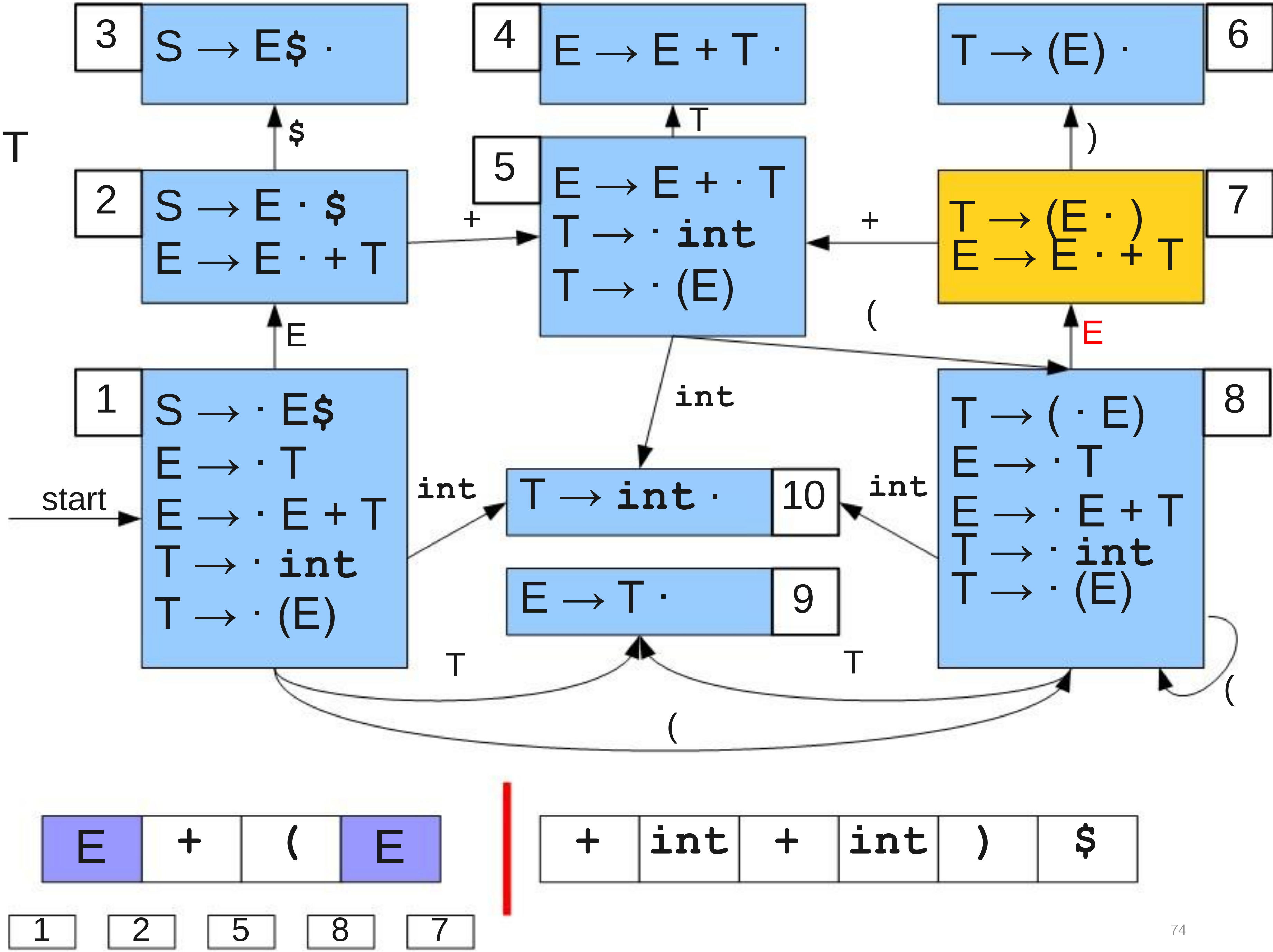
S → E$
E → T
E → E + T
T → int
T → (E)





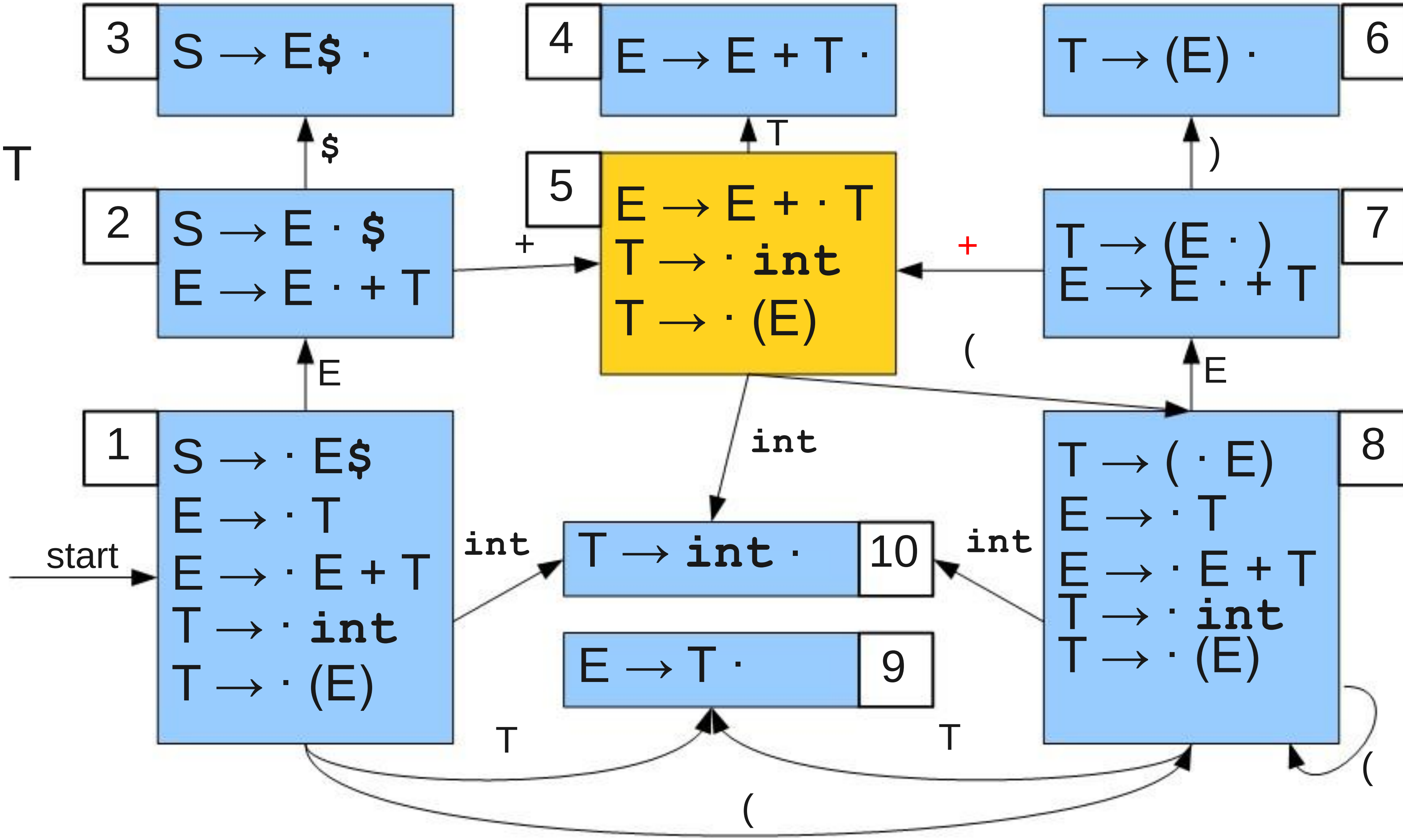
LR(0) Parsing

$S \to E\$$
$E \to T$
$E \to E + T$
$T \to \texttt{int}$
$T \to (E)$



LR(0) Parsing

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



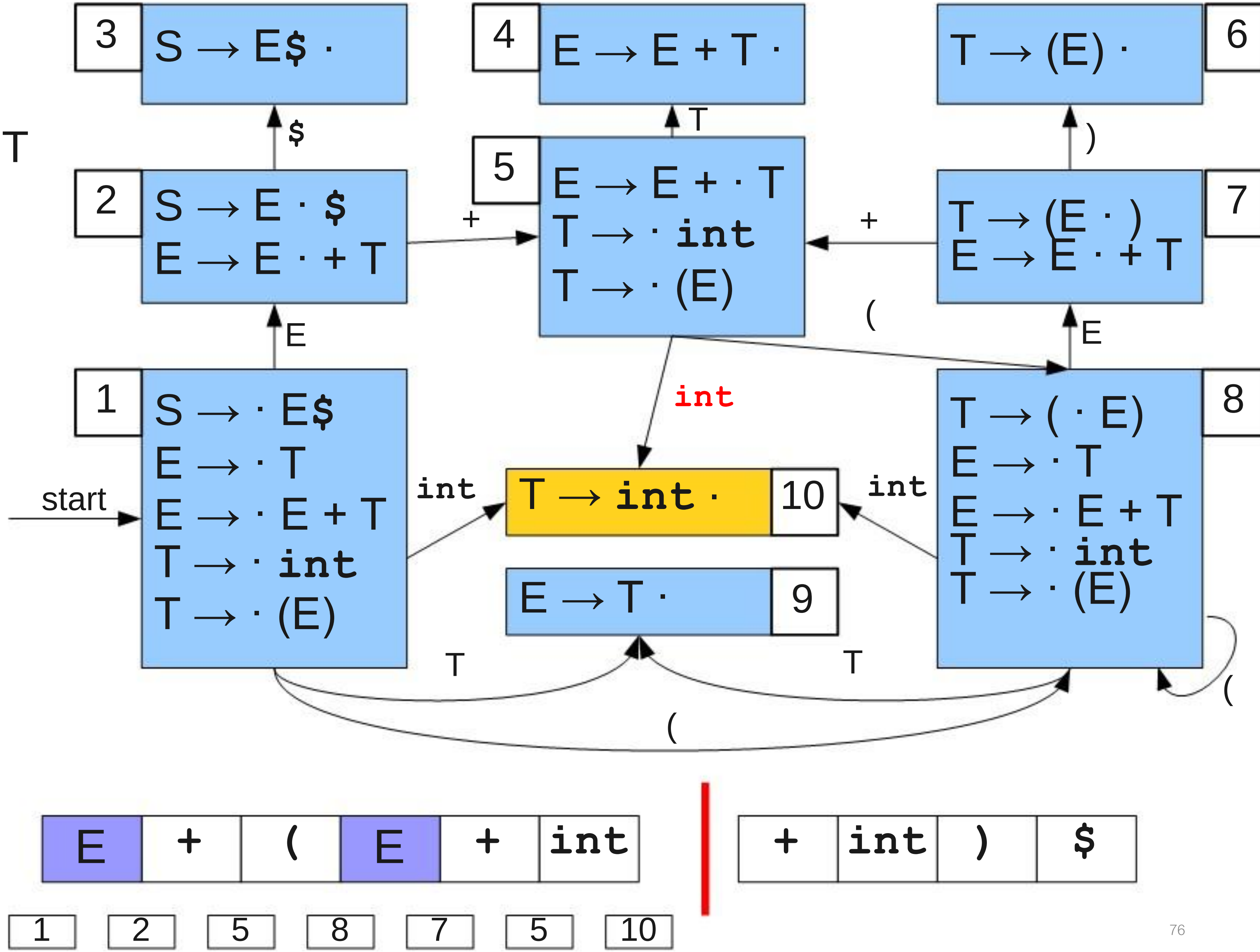
E	+	(	E	+

int	+	int	)	$

1	2	5	8	7	5

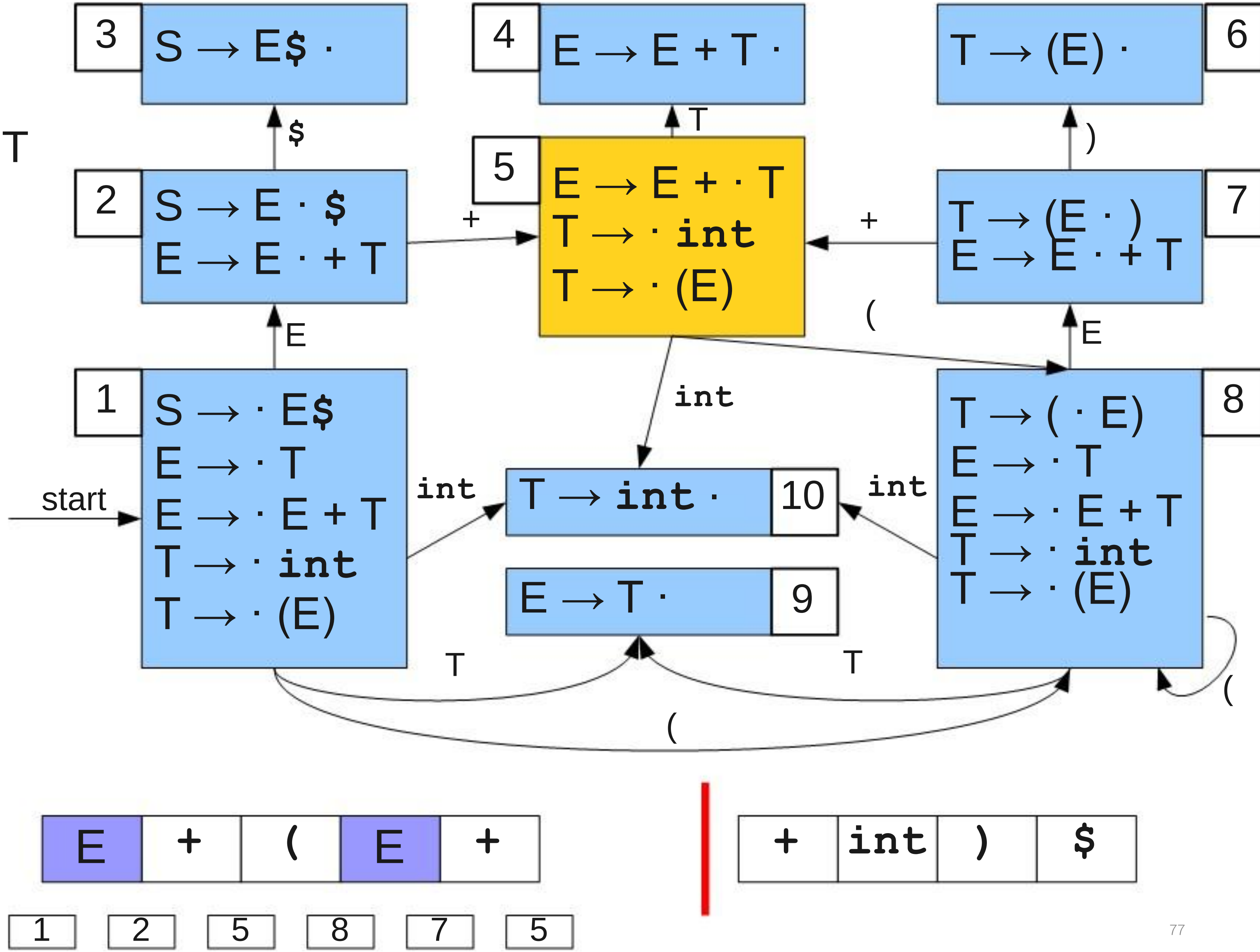
LR(0) Parsing

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



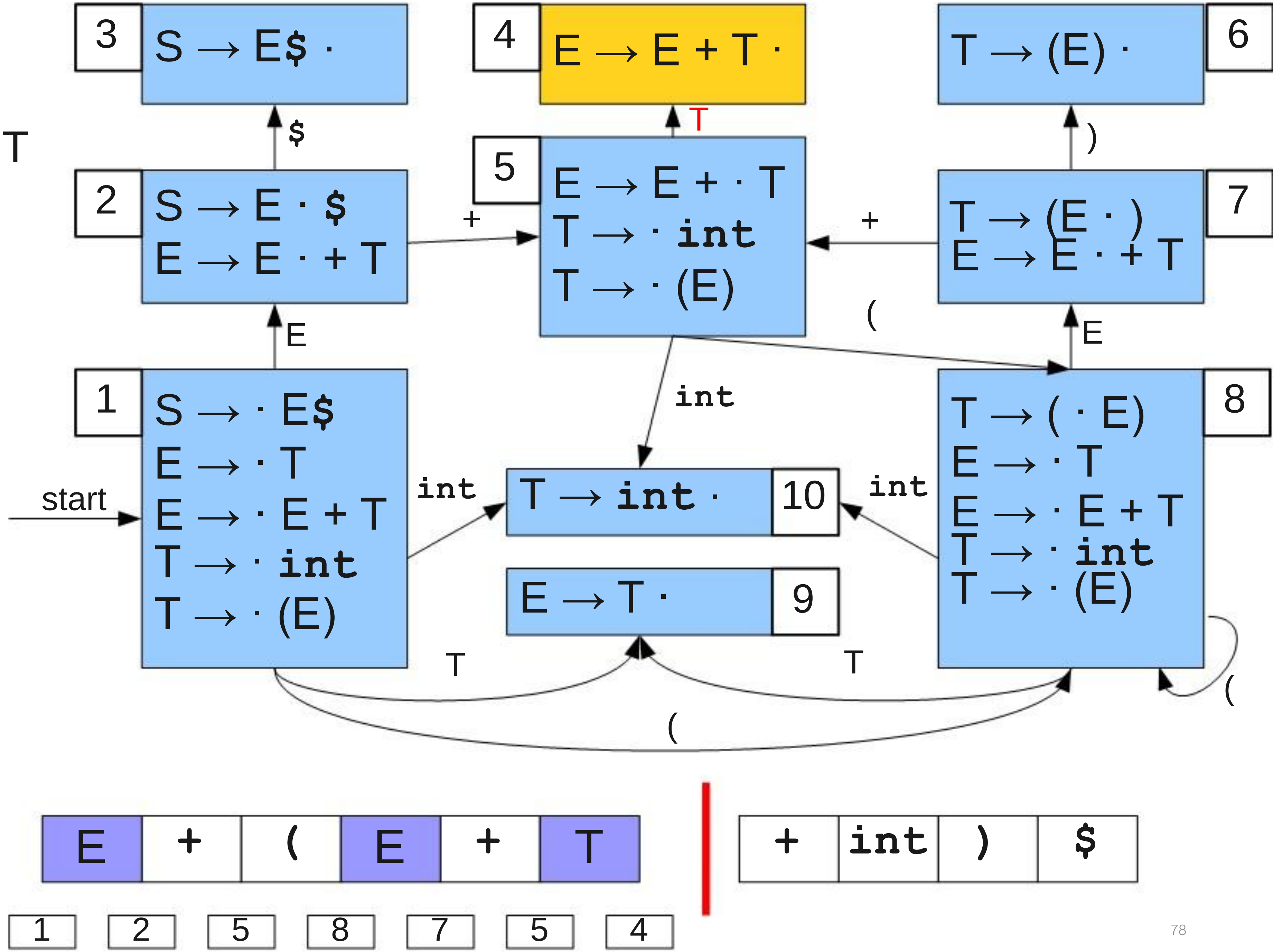
LR(0) Parsing

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



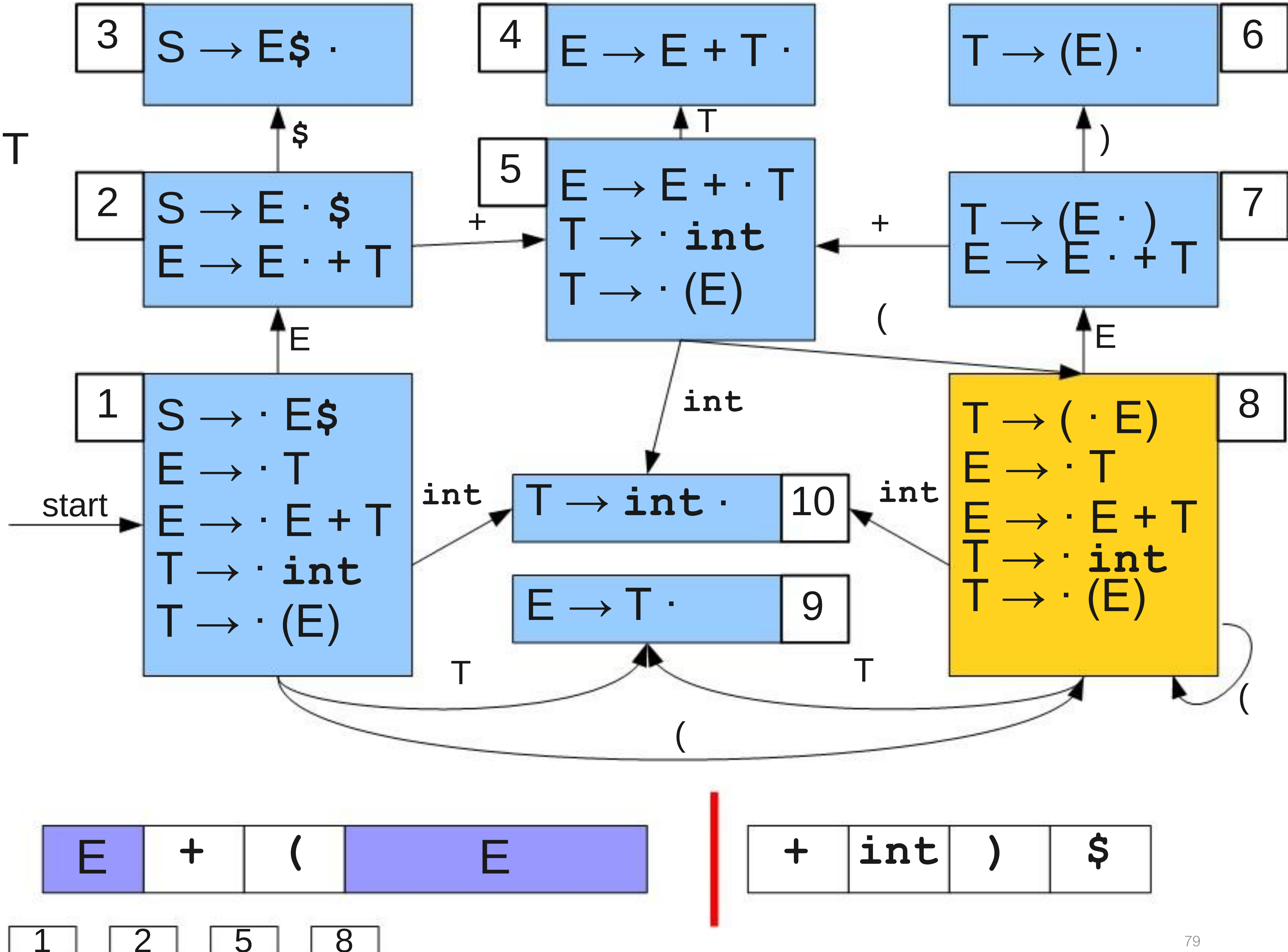
LR(0) Parsing

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



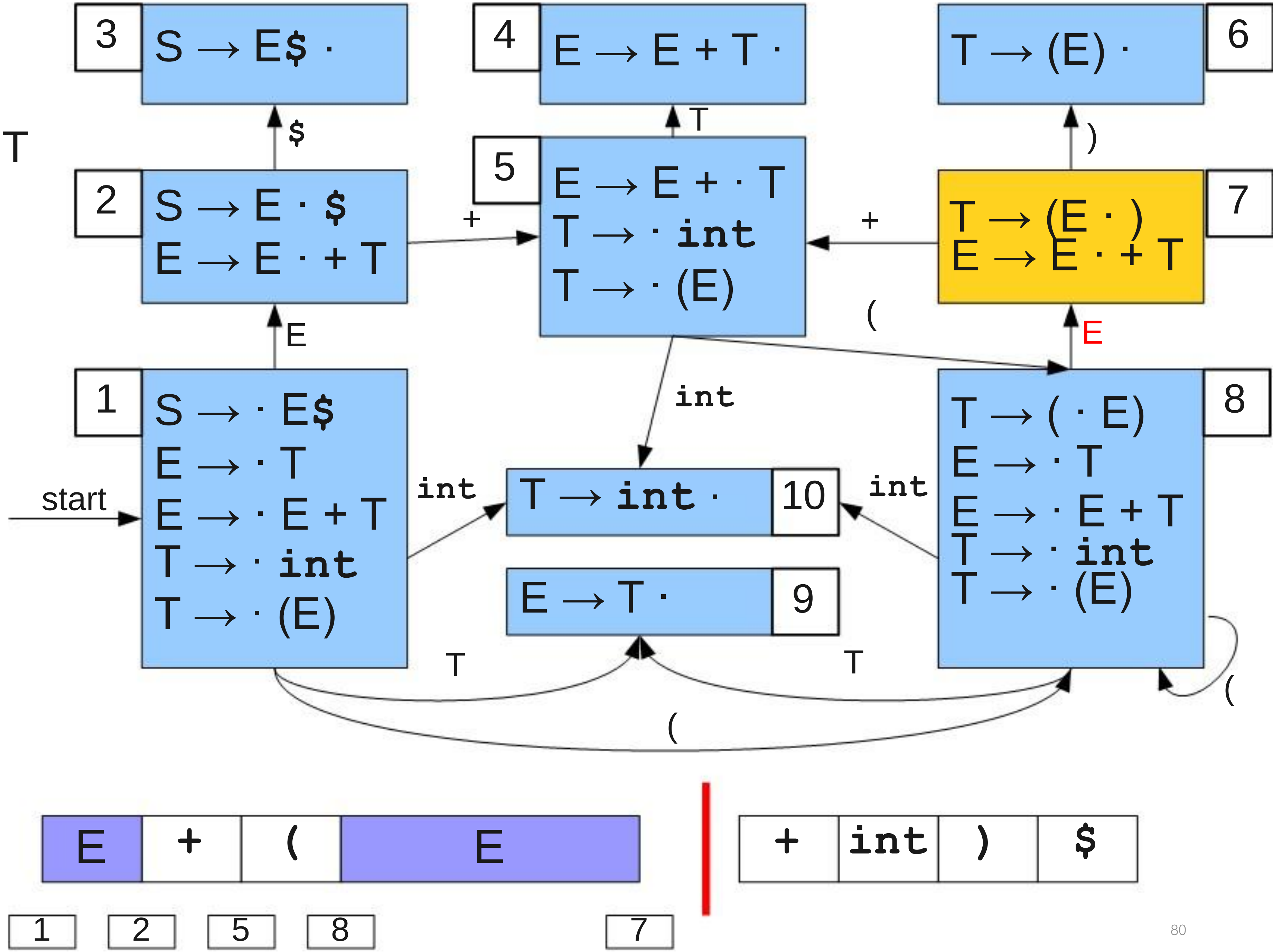
LR(0) Parsing

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

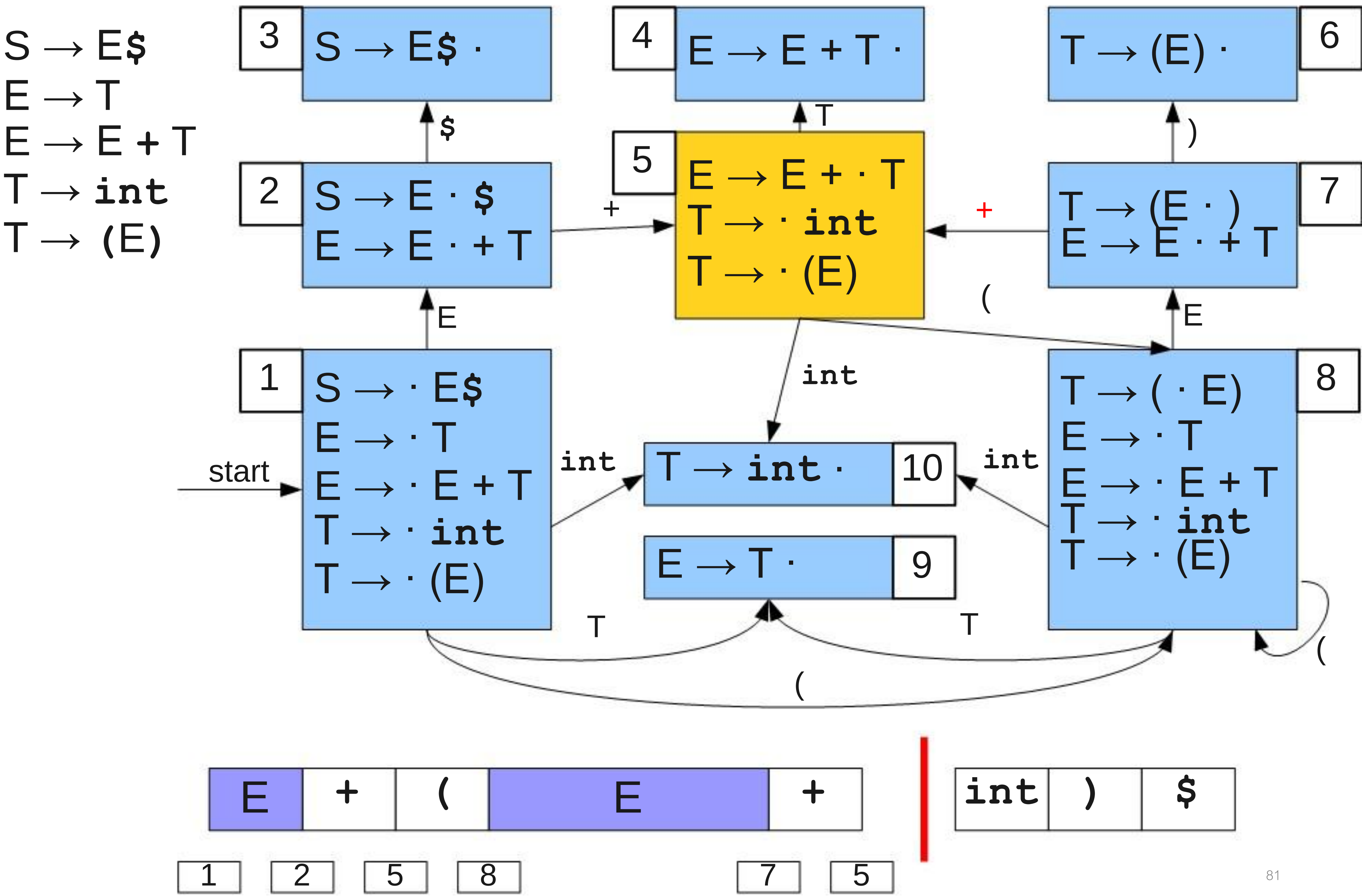


LR(0) Parsing

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



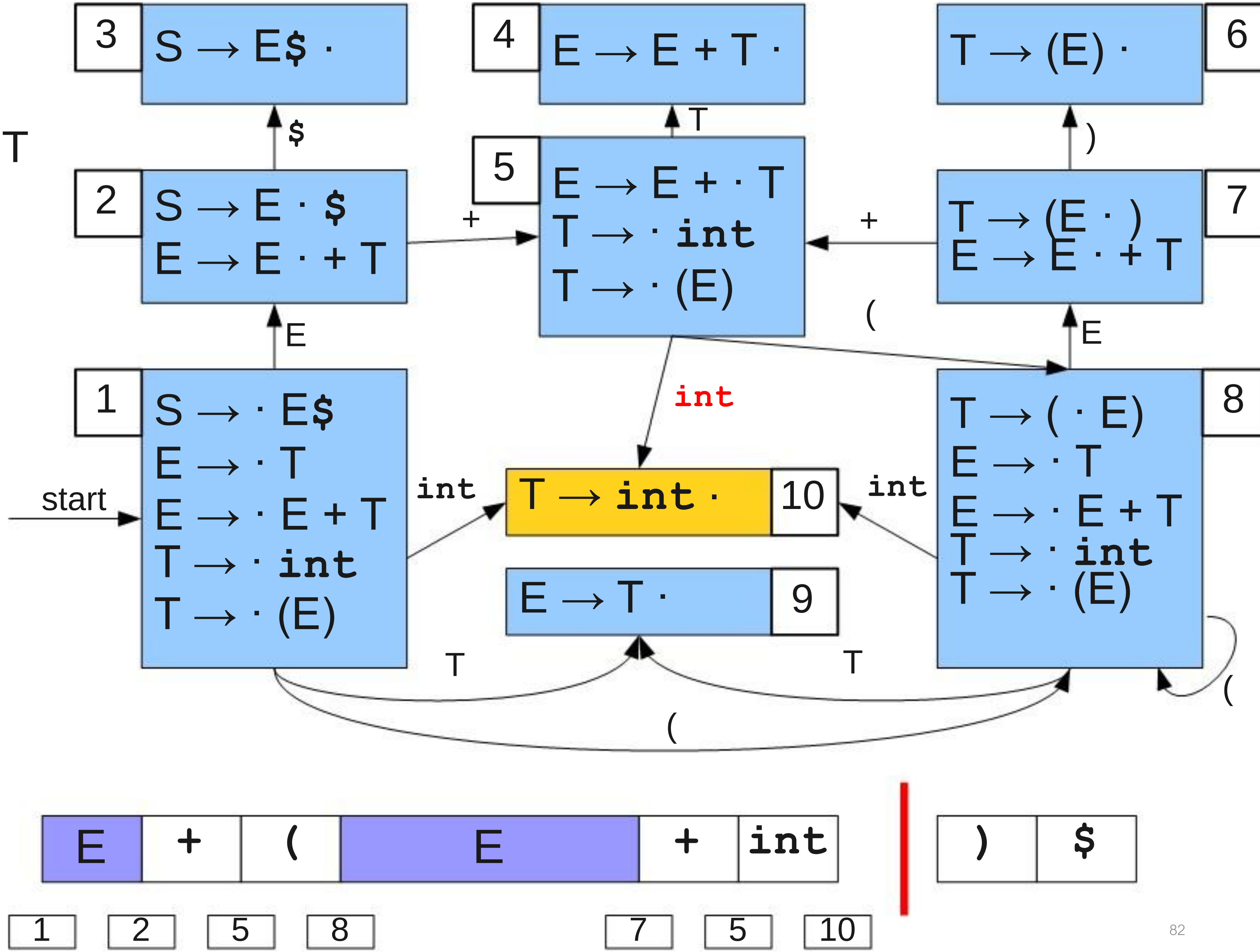
LR(0) Parsing



S → E$
E → T
E → E + T
T → int
T → (E)
3
S → E$ ·
4
E → E + T ·
6
T → (E) ·
$
T
)
5
E → E + · T
T → · int
T → · (E)
2
S → E · $
E → E · + T
+
+
7
T → (E ·)
E → E · + T
E
(
E
int
1
S → · E$
E → · T
E → · E + T
T → · int
T → · (E)
8
T → (· E)
E → · T
E → · E + T
T → · int
T → · (E)
start
int
T → int ·
10
int
E → T ·
9
T
T
(
(
E
+
(
E
+
int
)
$
1
2
5
8
7
5

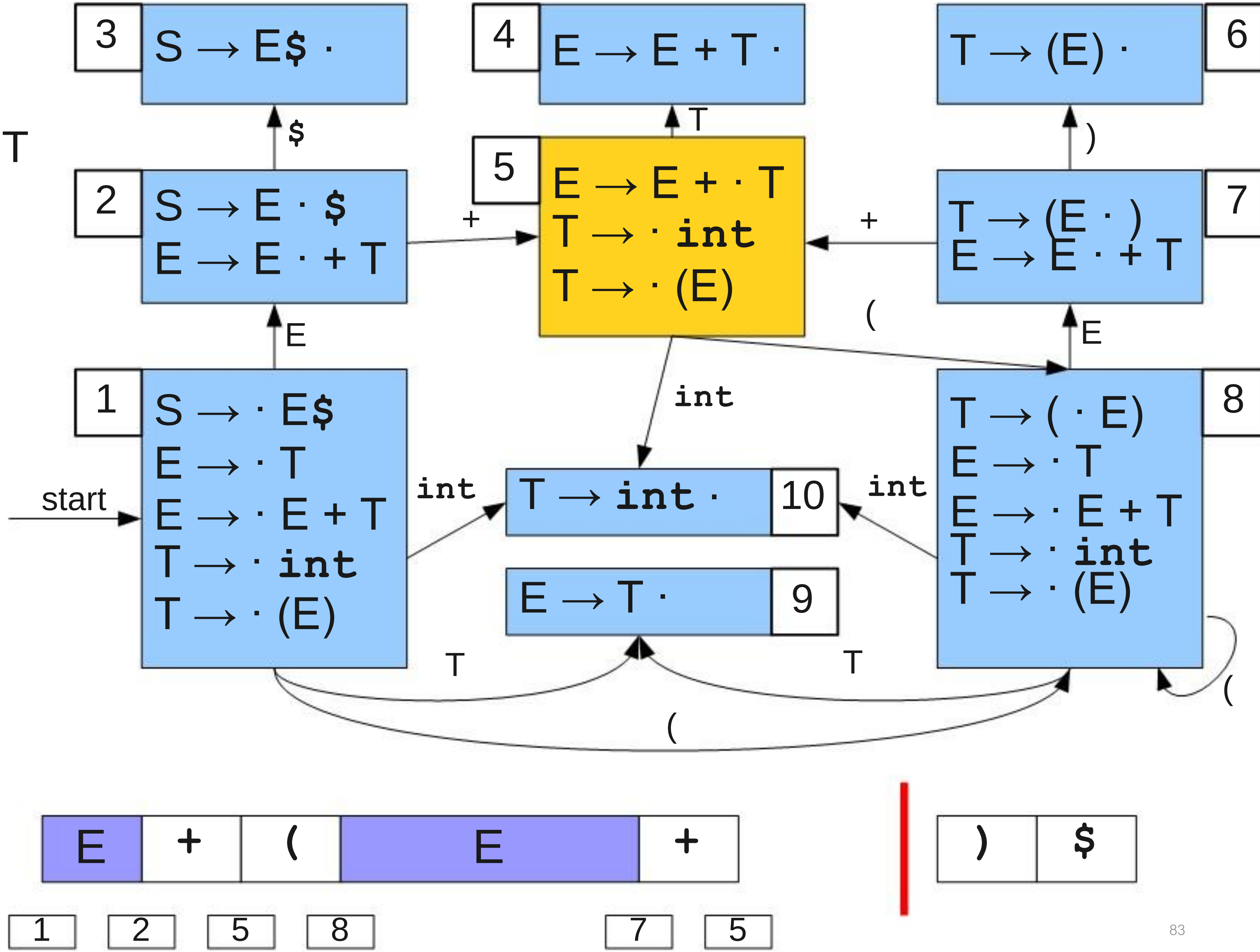
LR(0) Parsing

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



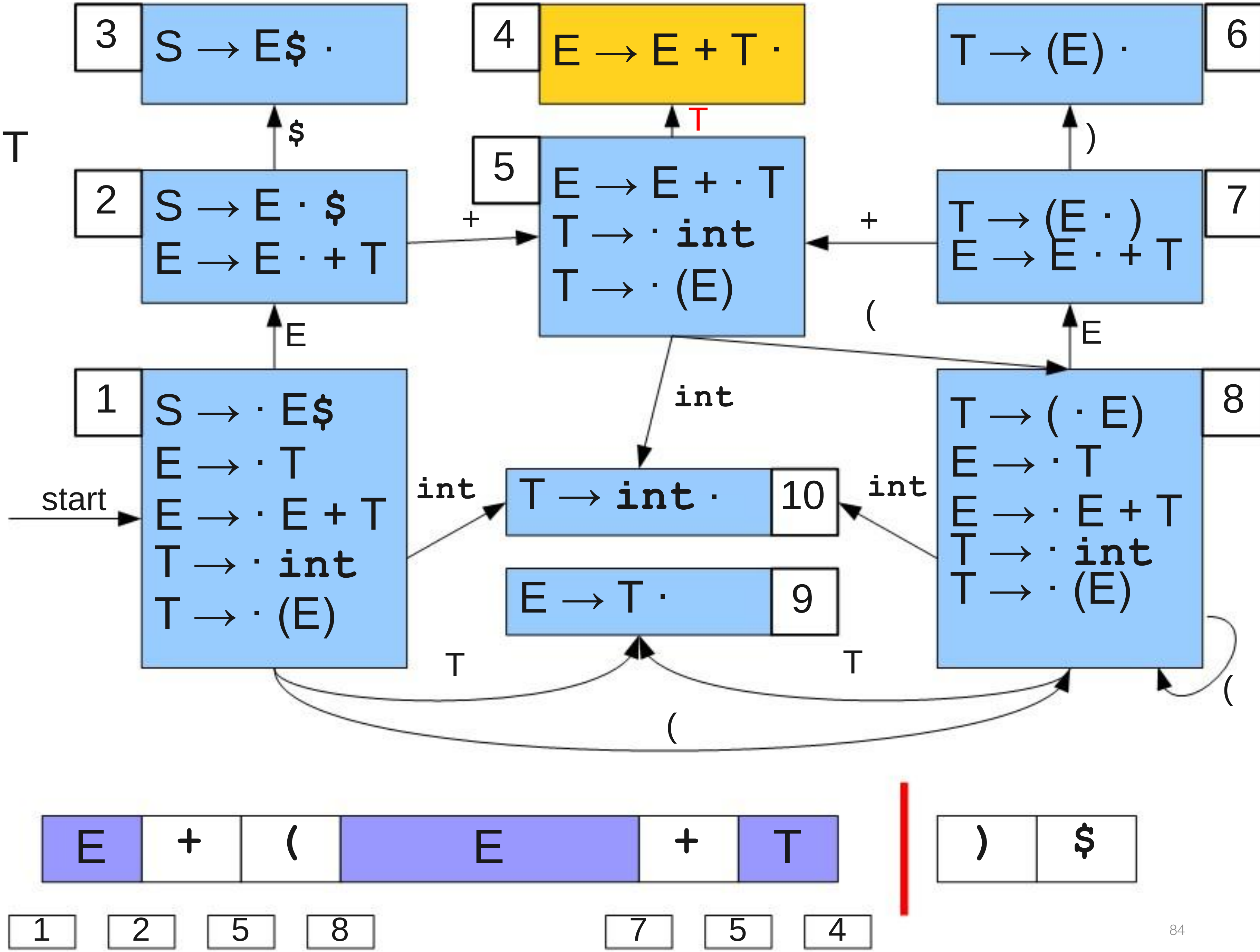
LR(0) Parsing

$S \to E\$$
$E \to T$
$E \to E + T$
$T \to \texttt{int}$
$T \to (E)$



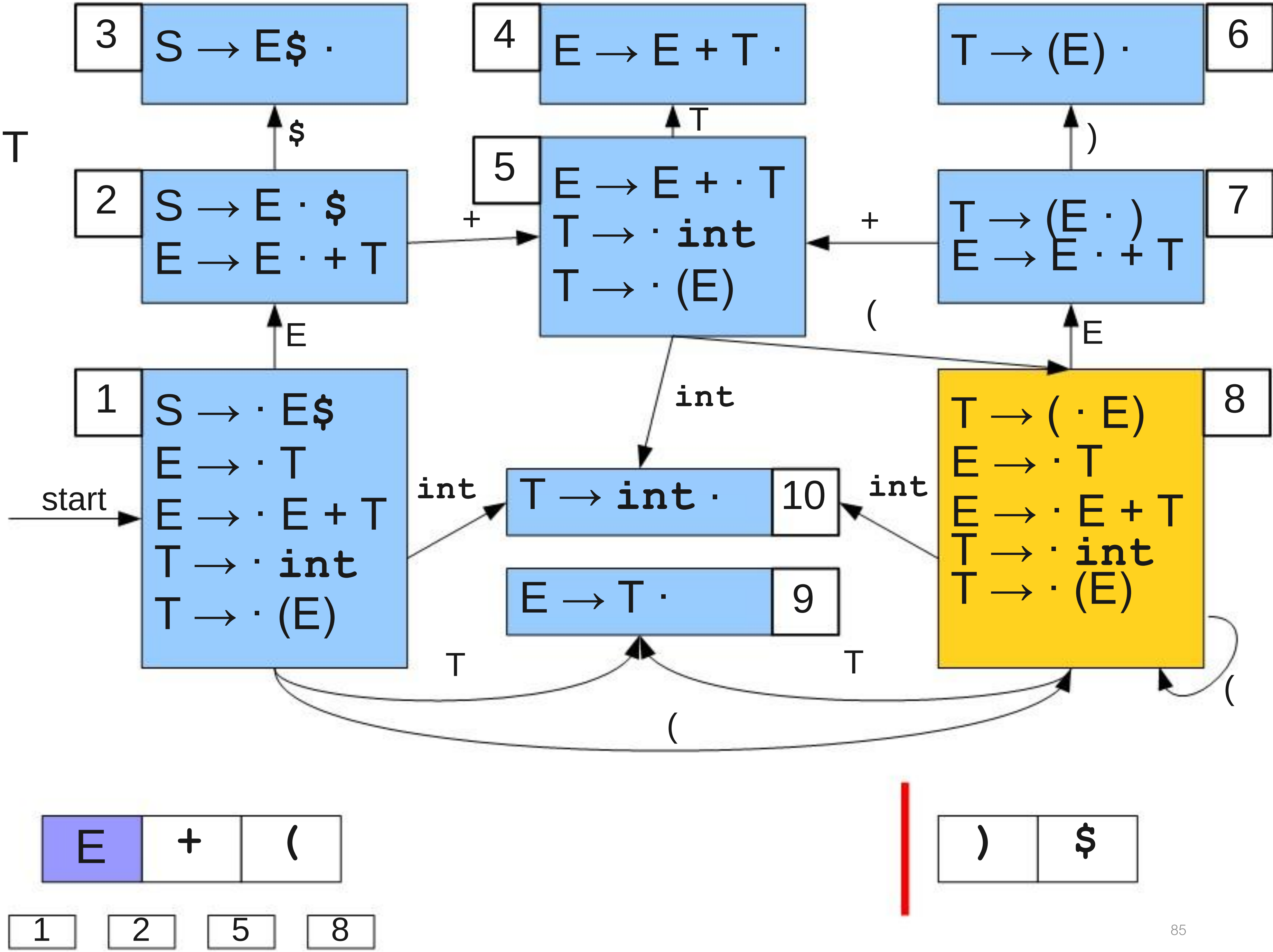
LR(0) Parsing

$S \to E\$$
$E \to T$
$E \to E + T$
$T \to \texttt{int}$
$T \to (E)$

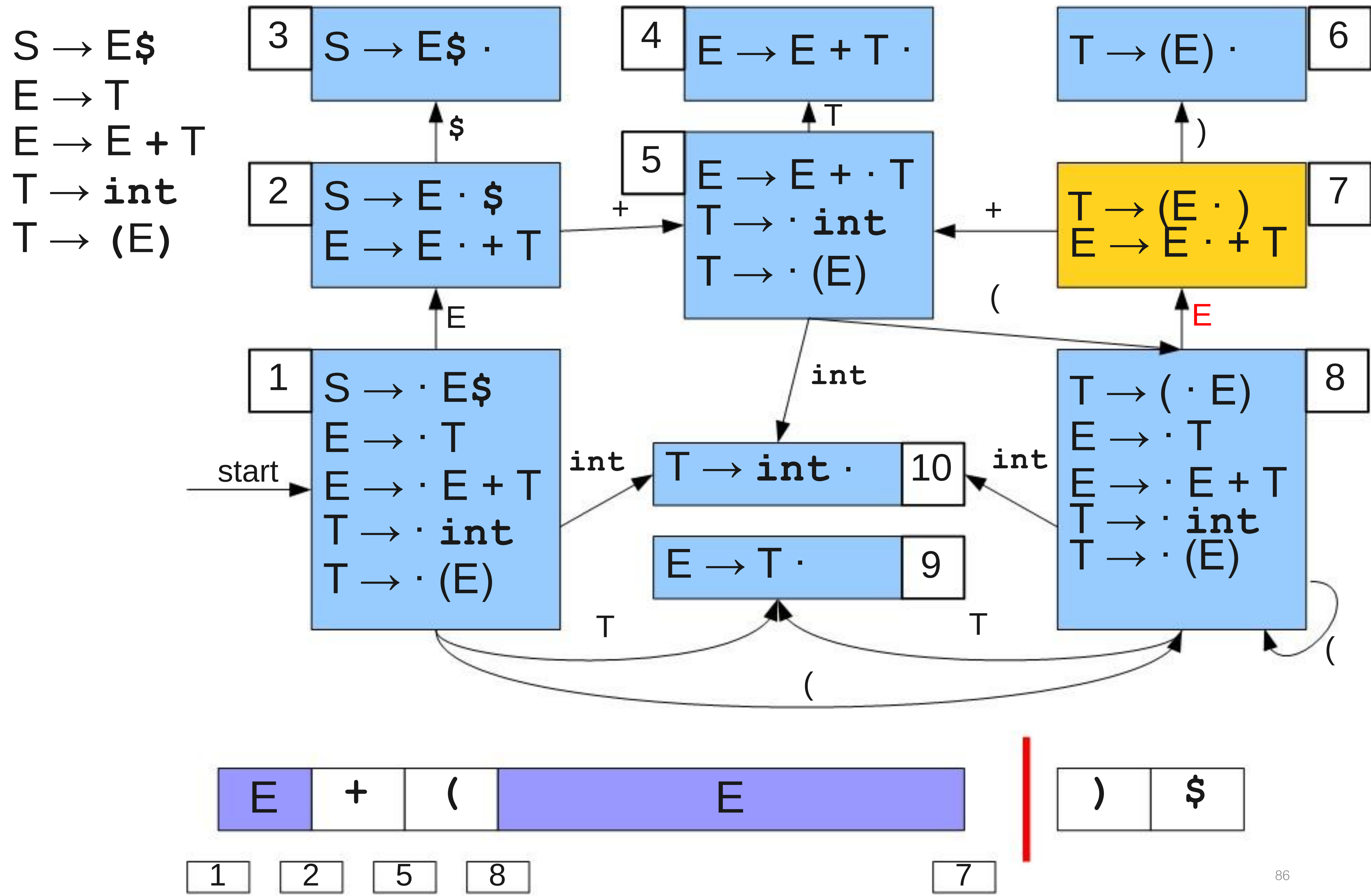


LR(0) Parsing

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



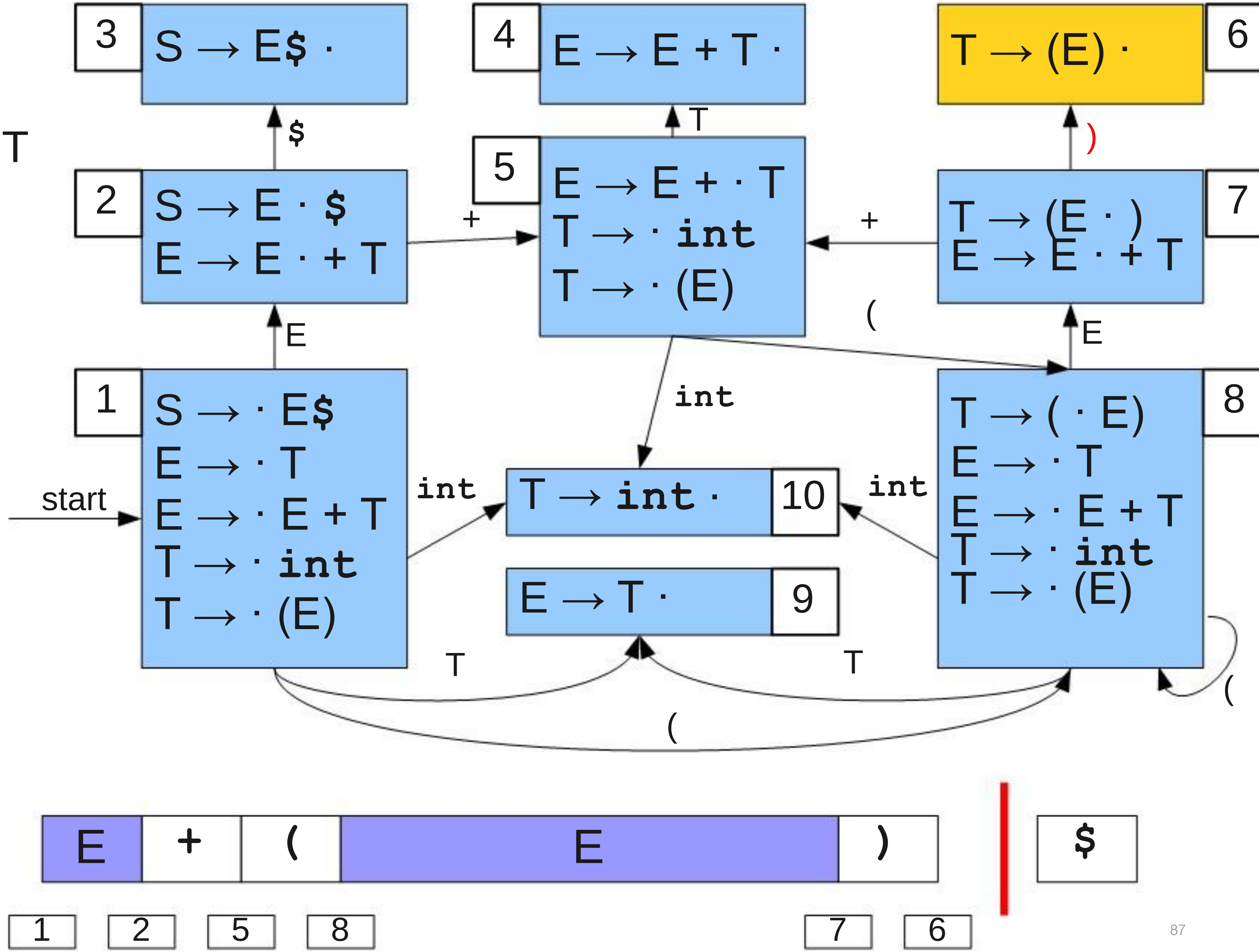
LR(0) Parsing



S → E$
E → T
E → E + T
T → int
T → (E)
3
S → E$ ·
$
2
S → E · $
E → E · + T
E
1
S → · E$
E → · T
E → · E + T
T → · int
T → · (E)
start
4
E → E + T ·
T
5
E → E + · T
T → · int
T → · (E)
+
int
T → int ·
10
E → T ·
9
6
T → (E) ·
)
7
T → (E ·)
E → E · + T
+
(
E
8
T → (· E)
E → · T
E → · E + T
T → · int
T → · (E)
int
int
T
T
(
(
E
+
(
E
)
$
1
2
5
8
7

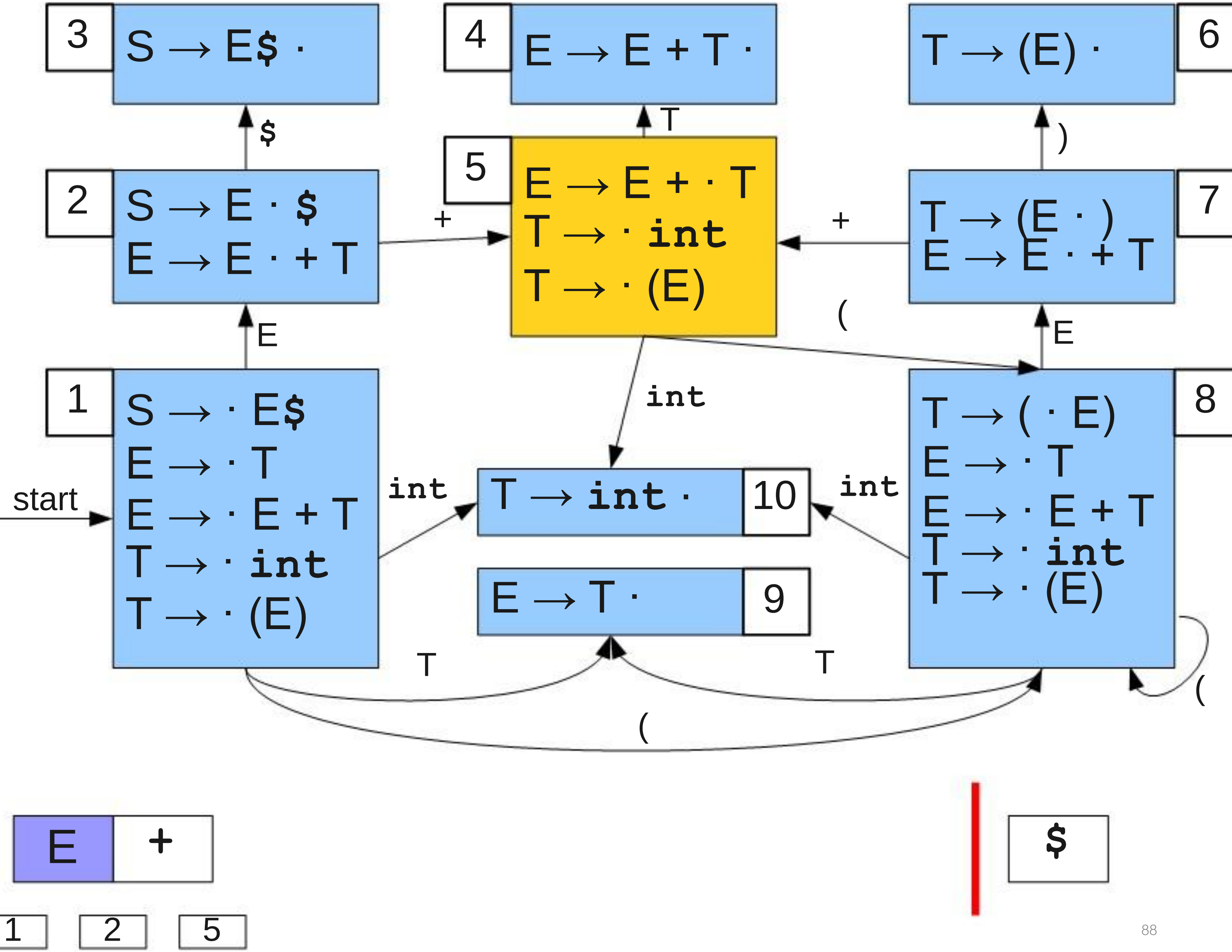
LR(0) Parsing

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



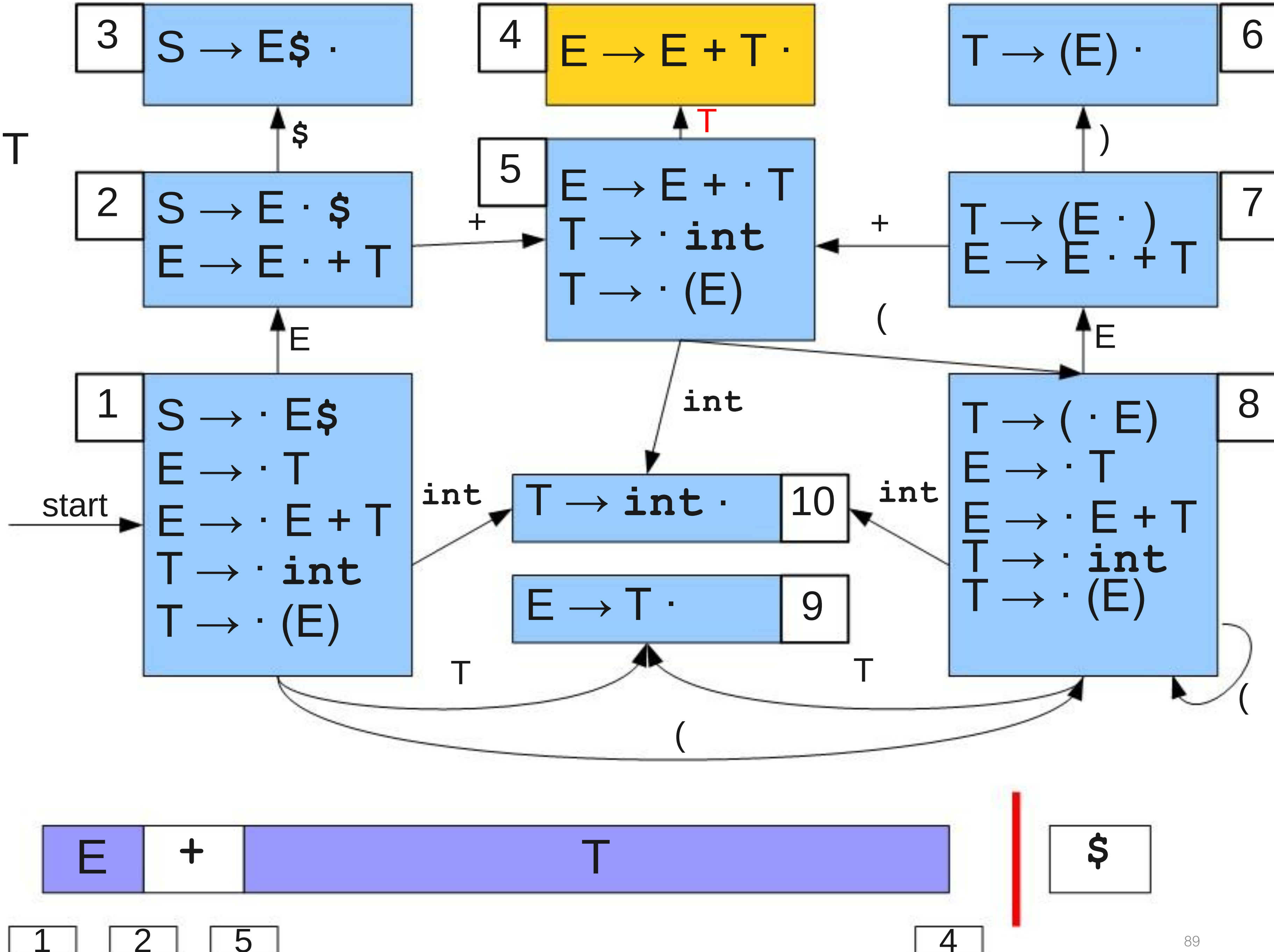
LR(0) Parsing

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



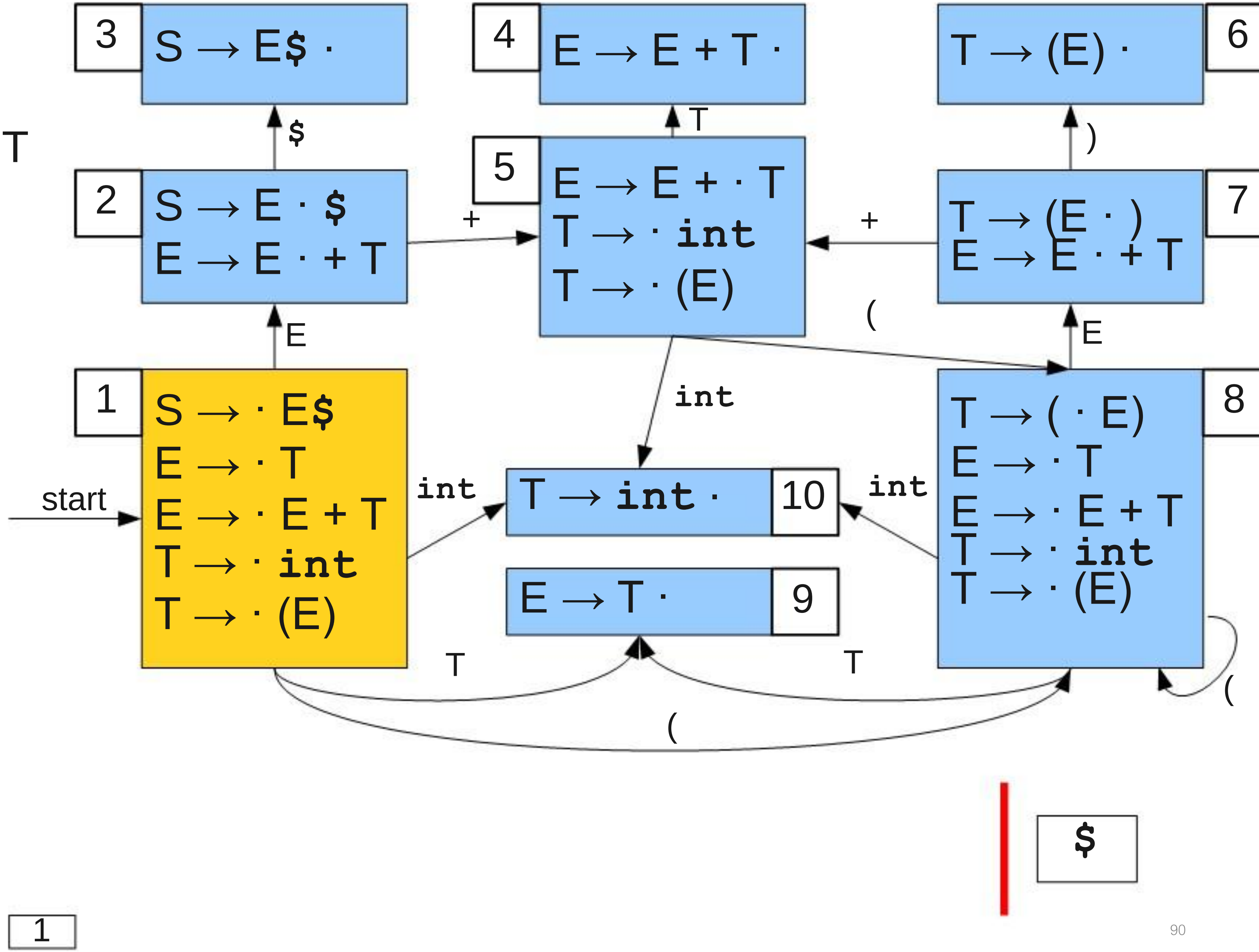
LR(0) Parsing

$S \to E\$$
$E \to T$
$E \to E + T$
$T \to \texttt{int}$
$T \to (E)$



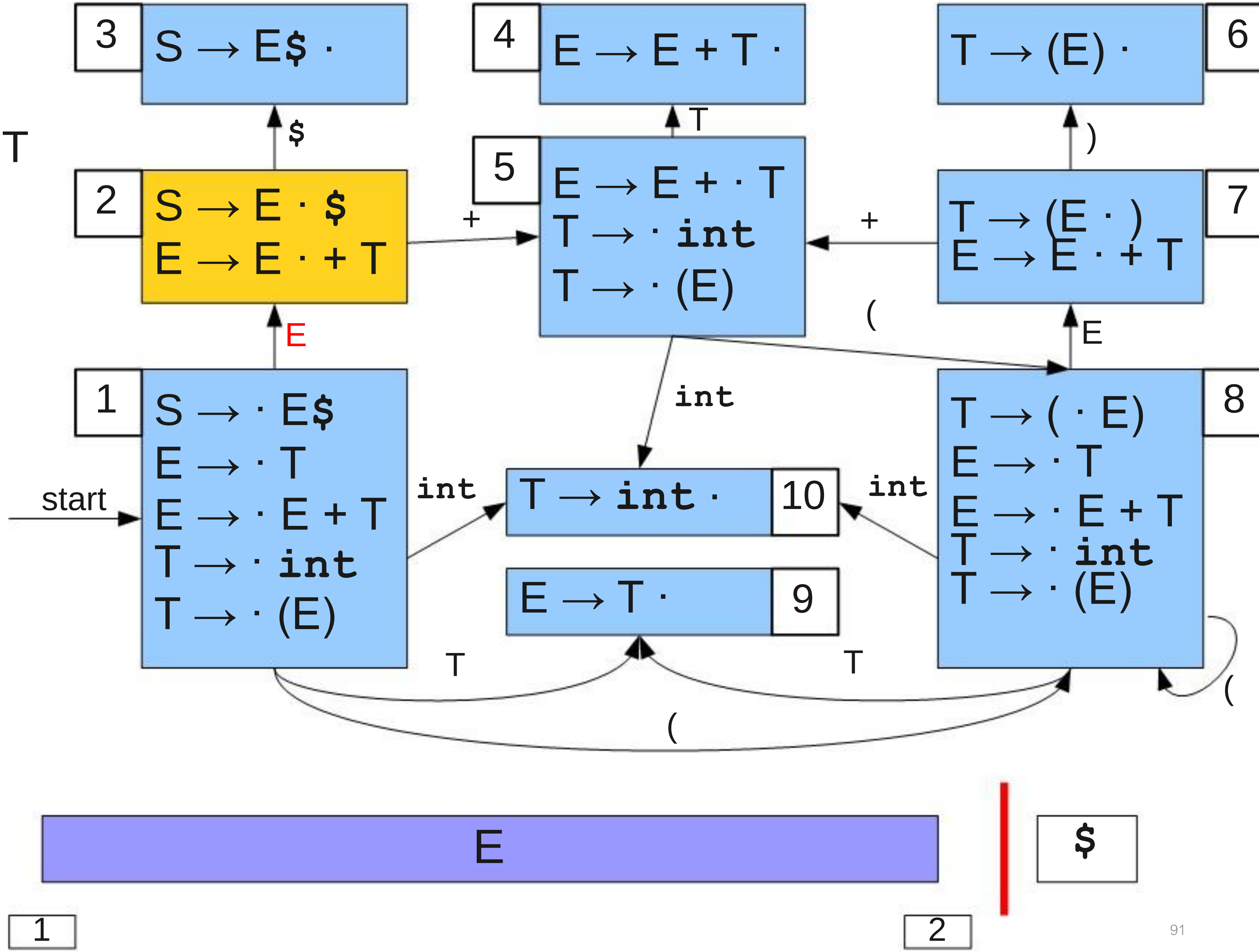
LR(0) Parsing

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



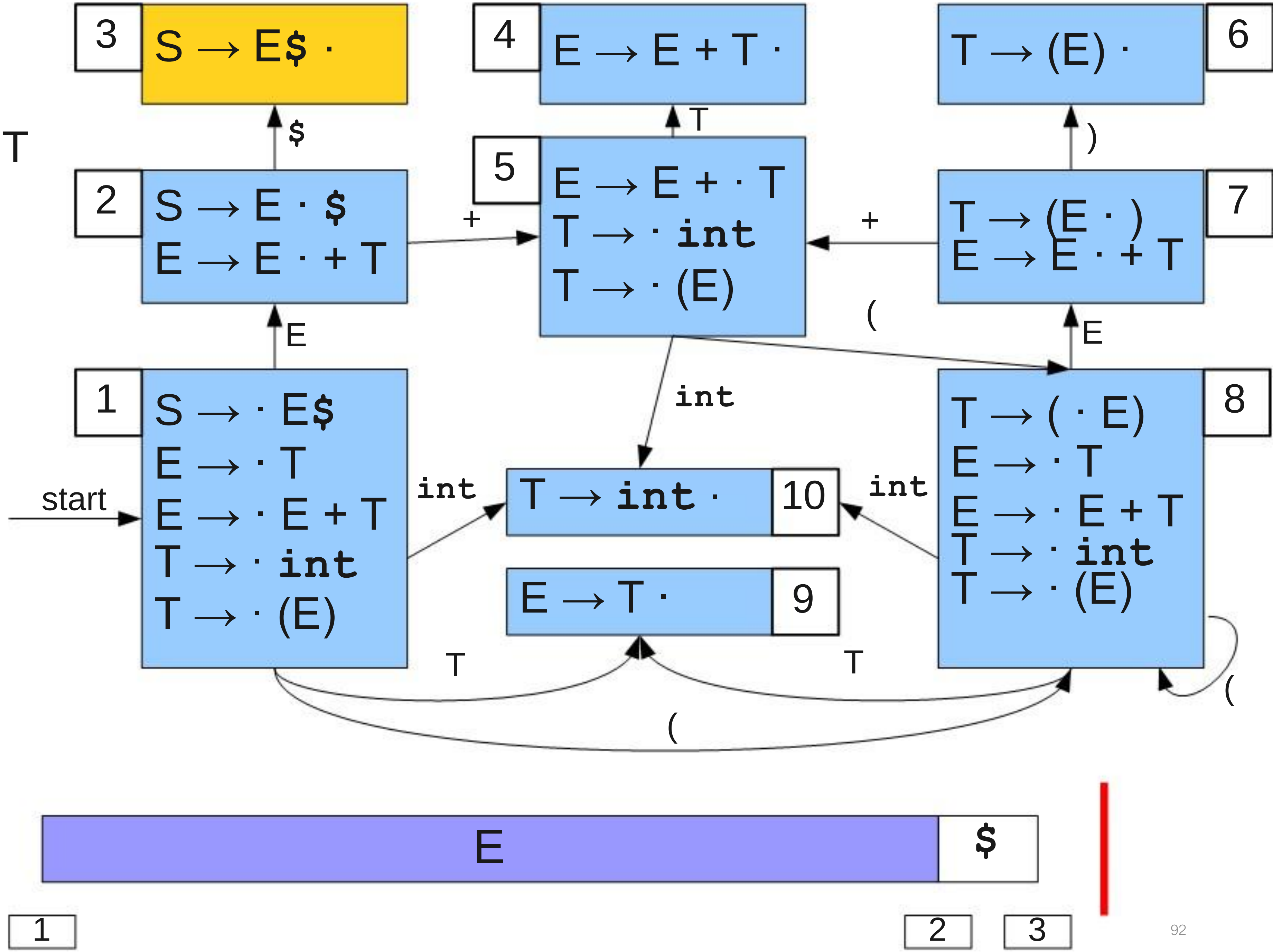
LR(0) Parsing

S → E$
E → T
E → E + T
T → `int`
T → (E)



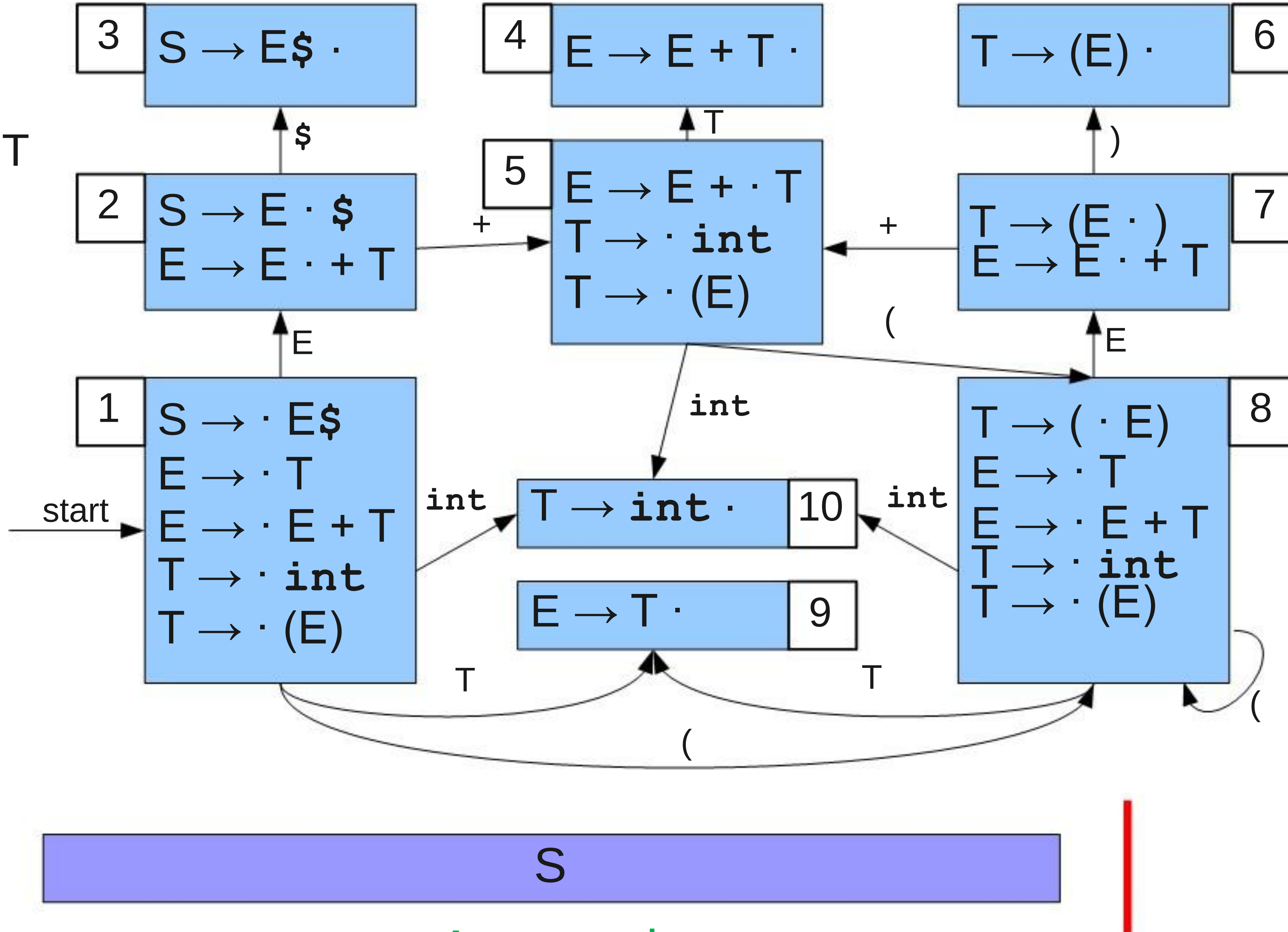
LR(0) Parsing

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



LR(0) Parsing

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



Accepted