Bottom-Up/Shift-Reduce Parsing

Parsing with CFGs

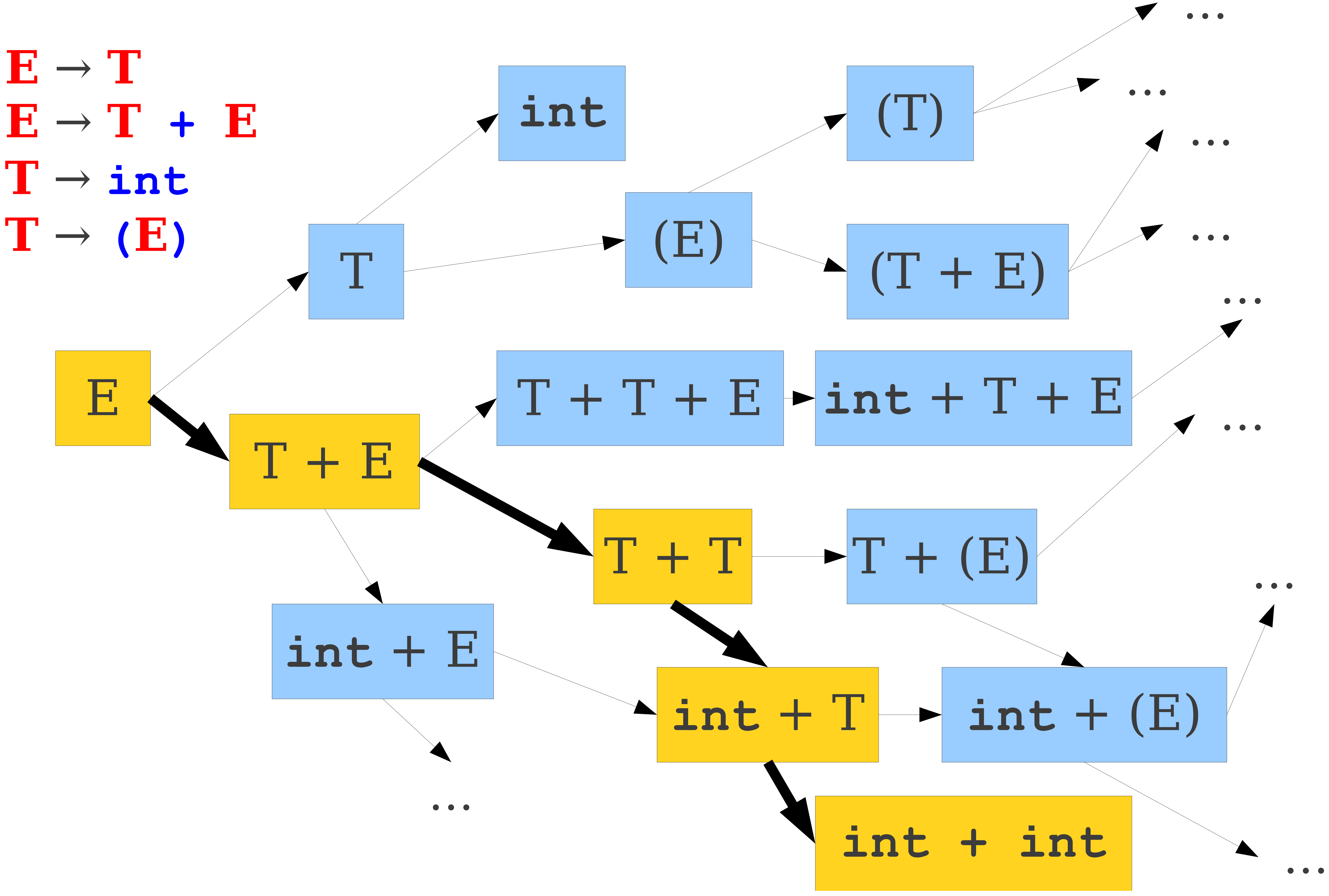
- Last Time: Top-down/Predictive/LL(1)
  - Fast, simple enough for hand-written parsers
  - Too weak for many practical languages
- This week: Bottom-up/Shift-Reduce/LR(0)/SLR/LR(1)
  - More Expressive
  - Too complicated for hand-written, instead use parser generators

Top-Down vs Bottom-Up

- Top-Down
  - Start from the start symbol S
  - Based on the input, rewrite to a new state aSb -> agb
  - Succeed when all input is read, the state is the input sequence

- Bottom-Up
  - Start from the input sequence w
  - Based on the input, rewrite
  - Succeed when all input is read, the state is the start symbol

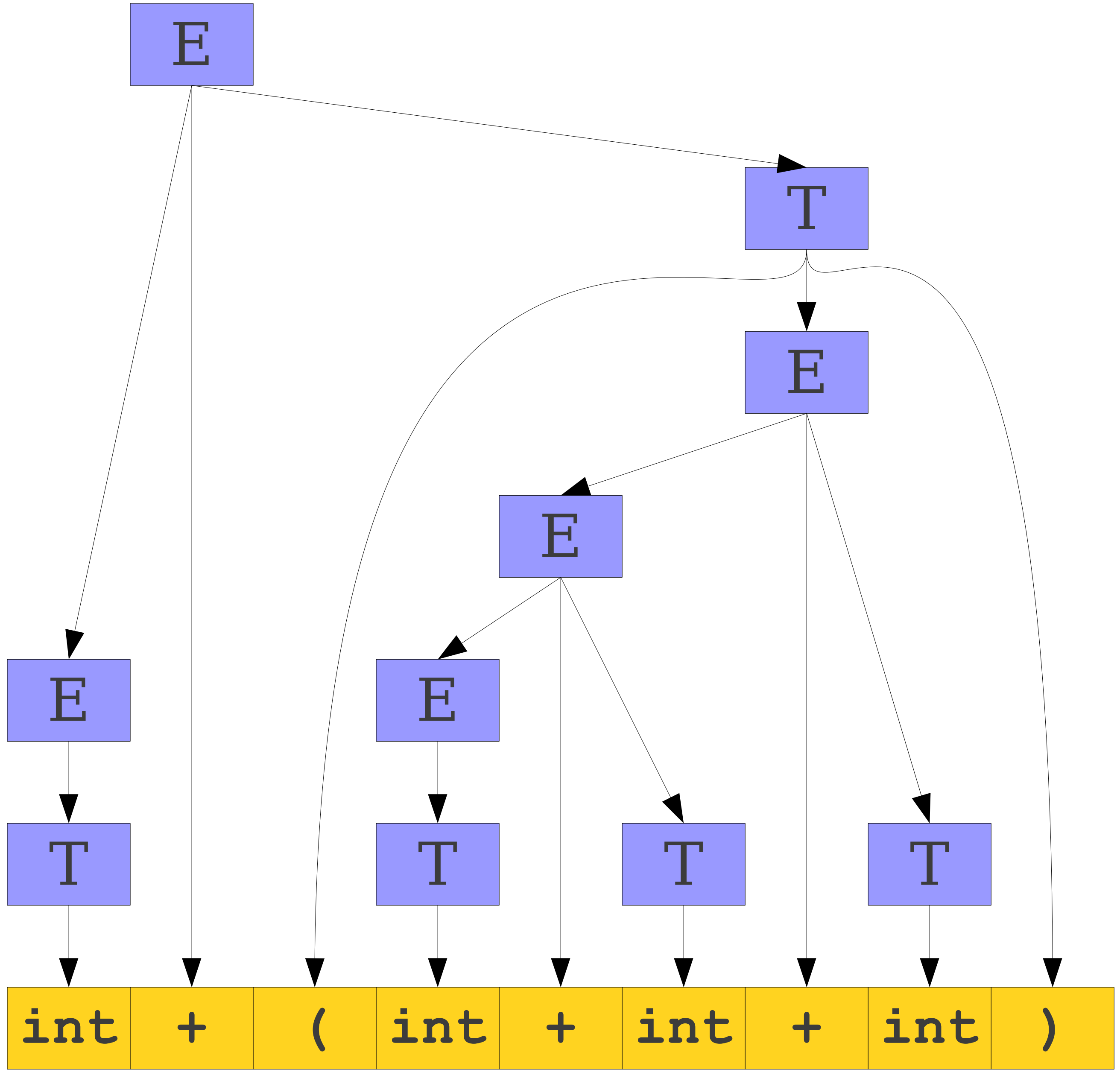
Top-Down



E → T
E → T + E
T → int
T → (E)
E
T
T + E
int
(E)
T + T + E
int + E
T + T
(T)
(T + E)
int + T + E
T + (E)
int + T
int + (E)
int + int
...

One View of a Bottom-Up Parse

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



A Second View of a Bottom-Up Parse

$\mathbf{E} \rightarrow \mathbf{T}$

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

$\mathbf{T} \rightarrow \texttt{int}$

$\mathbf{T} \rightarrow (\mathbf{E})$

$$
\begin{aligned}
& \texttt{int + (int + int + int)} \\
\Rightarrow\ & \mathbf{T}\ \texttt{+ (int + int + int)} \\
\Rightarrow\ & \mathbf{E}\ \texttt{+ (int + int + int)} \\
\Rightarrow\ & \mathbf{E}\ \texttt{+ (}\mathbf{T}\ \texttt{+ int + int)} \\
\Rightarrow\ & \mathbf{E}\ \texttt{+ (}\mathbf{E}\ \texttt{+ int + int)} \\
\Rightarrow\ & \mathbf{E}\ \texttt{+ (}\mathbf{E}\ \texttt{+}\ \mathbf{T}\ \texttt{+ int)} \\
\Rightarrow\ & \mathbf{E}\ \texttt{+ (}\mathbf{E}\ \texttt{+ int)} \\
\Rightarrow\ & \mathbf{E}\ \texttt{+ (}\mathbf{E}\ \texttt{+}\ \mathbf{T}\texttt{)} \\
\Rightarrow\ & \mathbf{E}\ \texttt{+ (}\mathbf{E}\texttt{)} \\
\Rightarrow\ & \mathbf{E}\ \texttt{+}\ \mathbf{T} \\
\Rightarrow\ & \mathbf{E}
\end{aligned}
$$

A left-to-right, bottom-up parse is a rightmost derivation traced in reverse.

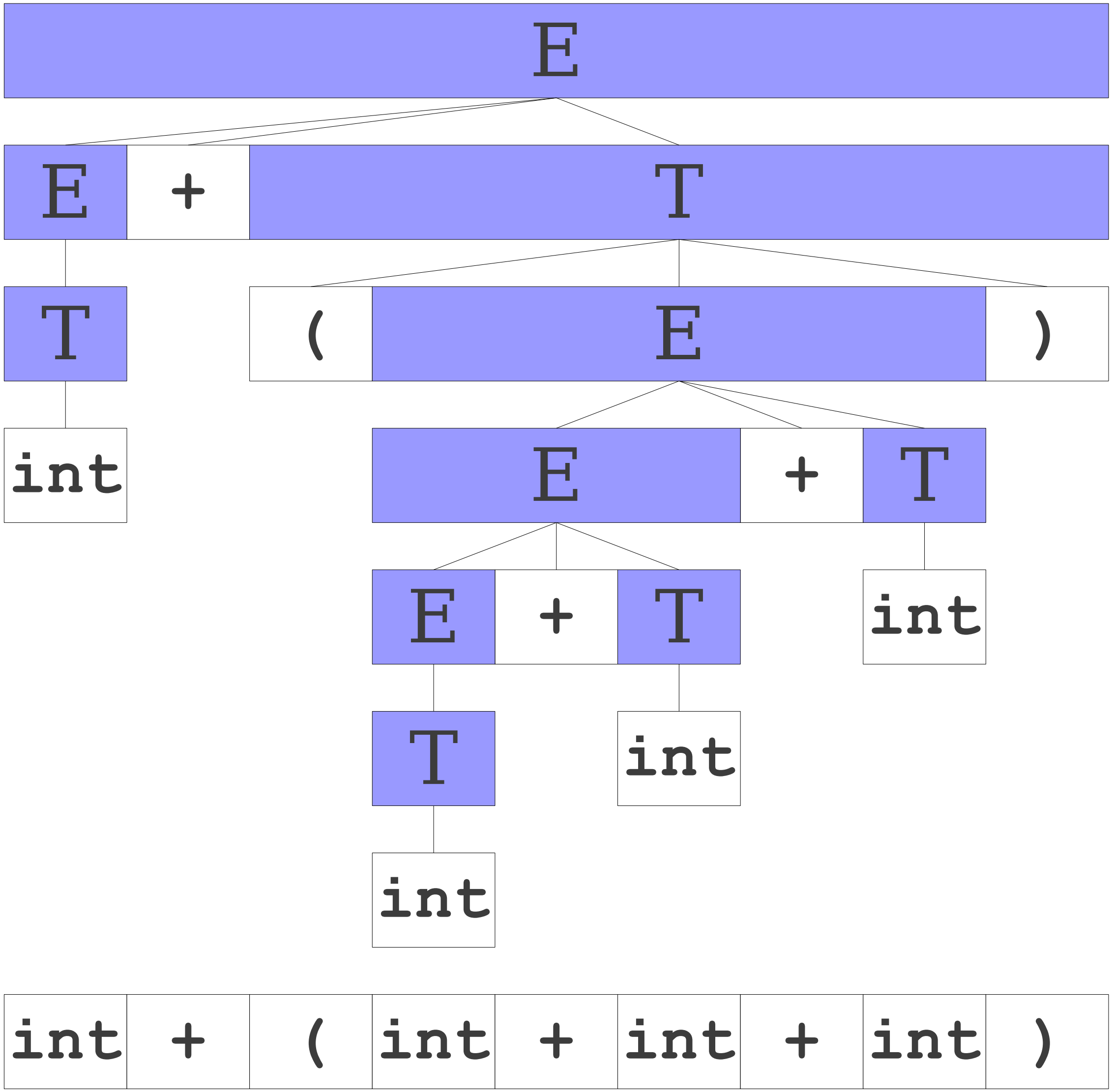
A Third View of a Bottom-Up Parse

```
  int + (int + int + int)
⇒ T + (int + int + int)
⇒ E + (int + int + int)
⇒ E + (T + int + int)
⇒ E + (E + int + int)
⇒ E + (E + T + int)
⇒ E + (E + int)
⇒ E + (E + T)
⇒ E + (E)
⇒ E + T
⇒ E
```

Each step in this bottom-up parse is called a **reduction**. We **reduce** a substring of the sentential form back to a nonterminal.

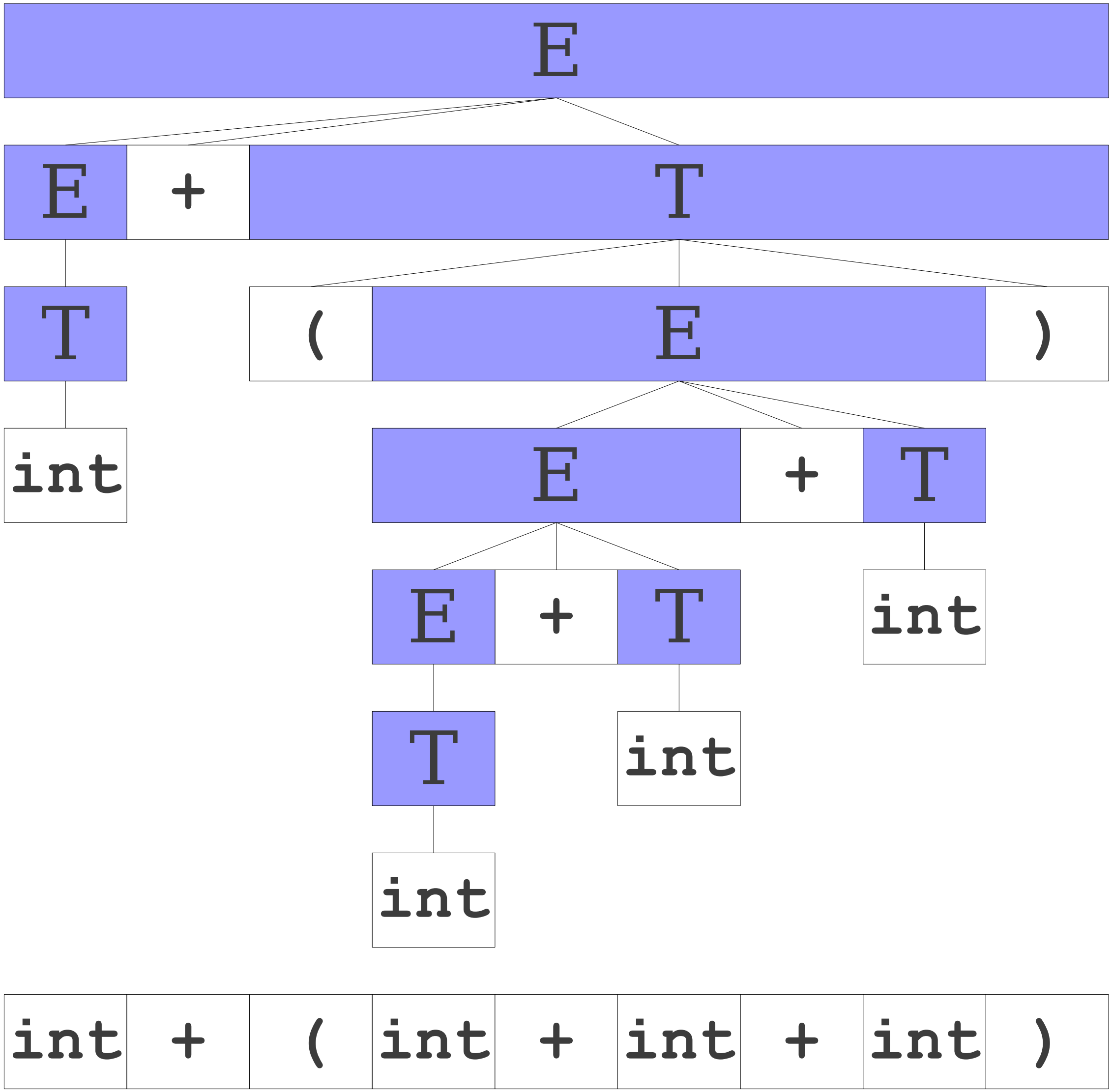
A Third View of a Bottom-Up Parse

```
  int + (int + int + int)
⇒ T + (int + int + int)
⇒ E + (int + int + int)
⇒ E + (T + int + int)
⇒ E + (E + int + int)
⇒ E + (E + T + int)
⇒ E + (E + int)
⇒ E + (E + T)
⇒ E + (E)
⇒ E + T
⇒ E
```



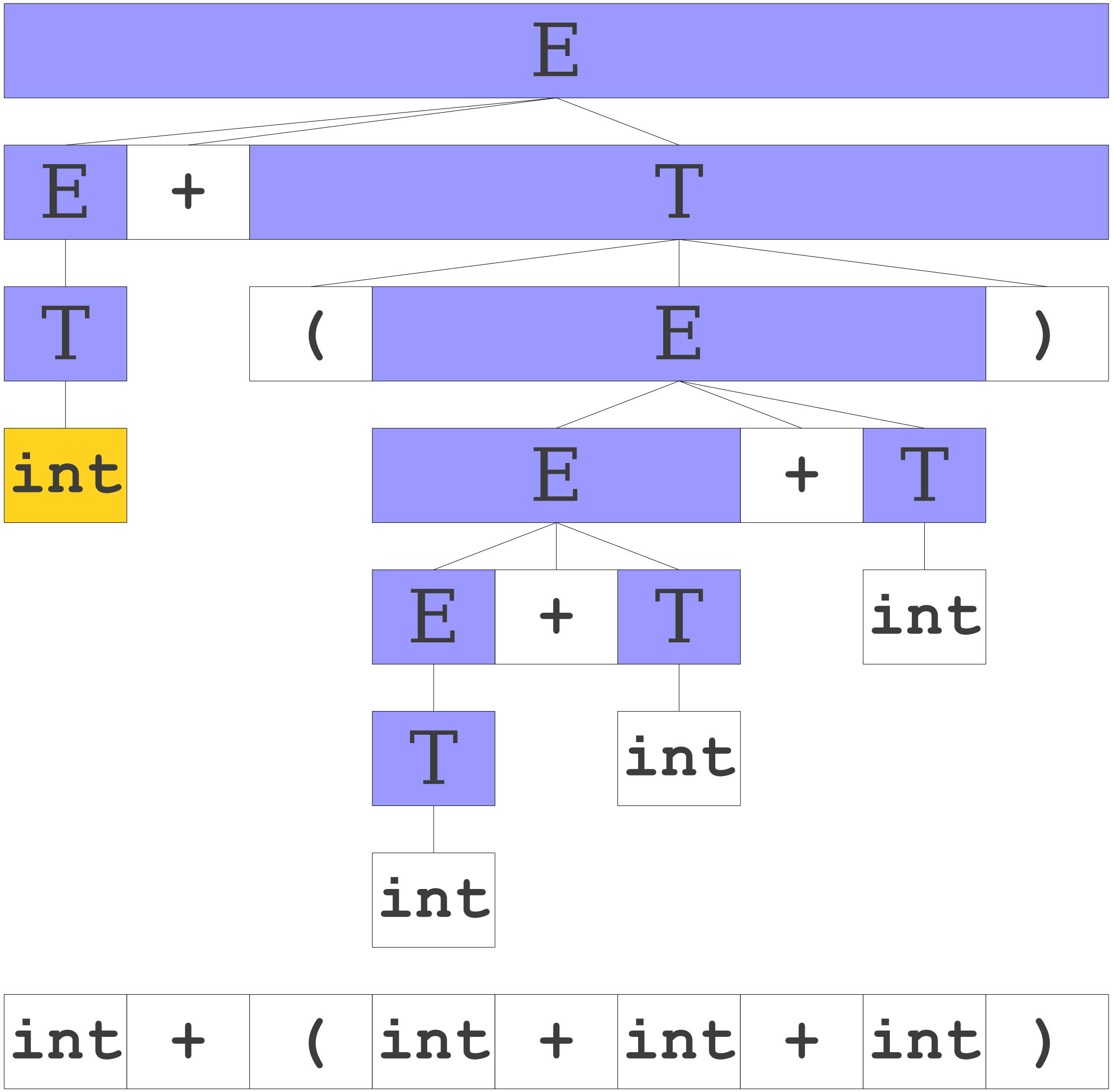
A Third View of a Bottom-Up Parse

int + (int + int + int)
⇒ T + (int + int + int)
⇒ E + (int + int + int)
⇒ E + (T + int + int)
⇒ E + (E + int + int)
⇒ E + (E + T + int)
⇒ E + (E + int)
⇒ E + (E + T)
⇒ E + (E)
⇒ E + T
⇒ E



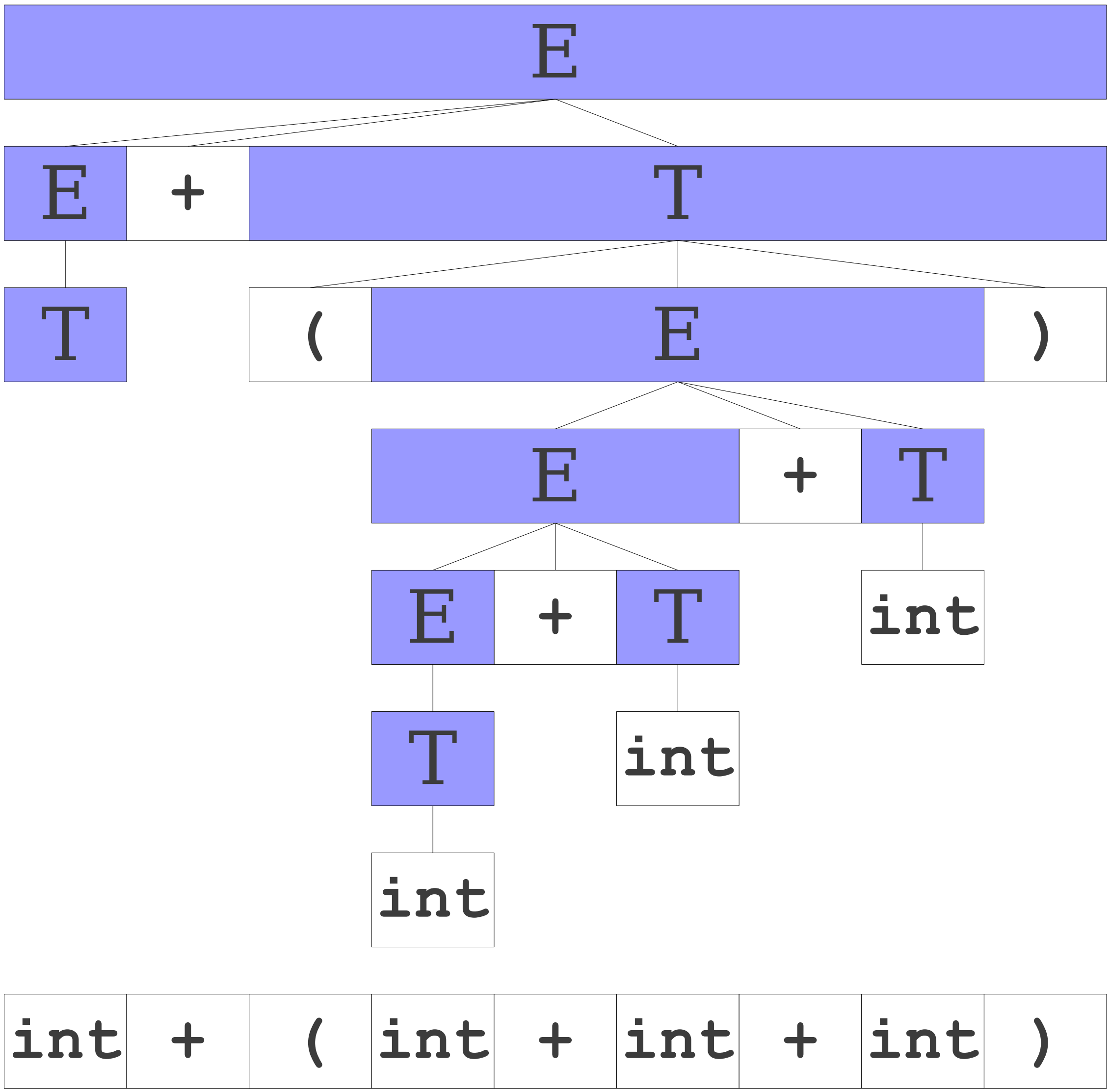
A Third View of a Bottom-Up Parse

```
  int + (int + int + int)
⇒ T + (int + int + int)
⇒ E + (int + int + int)
⇒ E + (T + int + int)
⇒ E + (E + int + int)
⇒ E + (E + T + int)
⇒ E + (E + int)
⇒ E + (E + T)
⇒ E + (E)
⇒ E + T
⇒ E
```



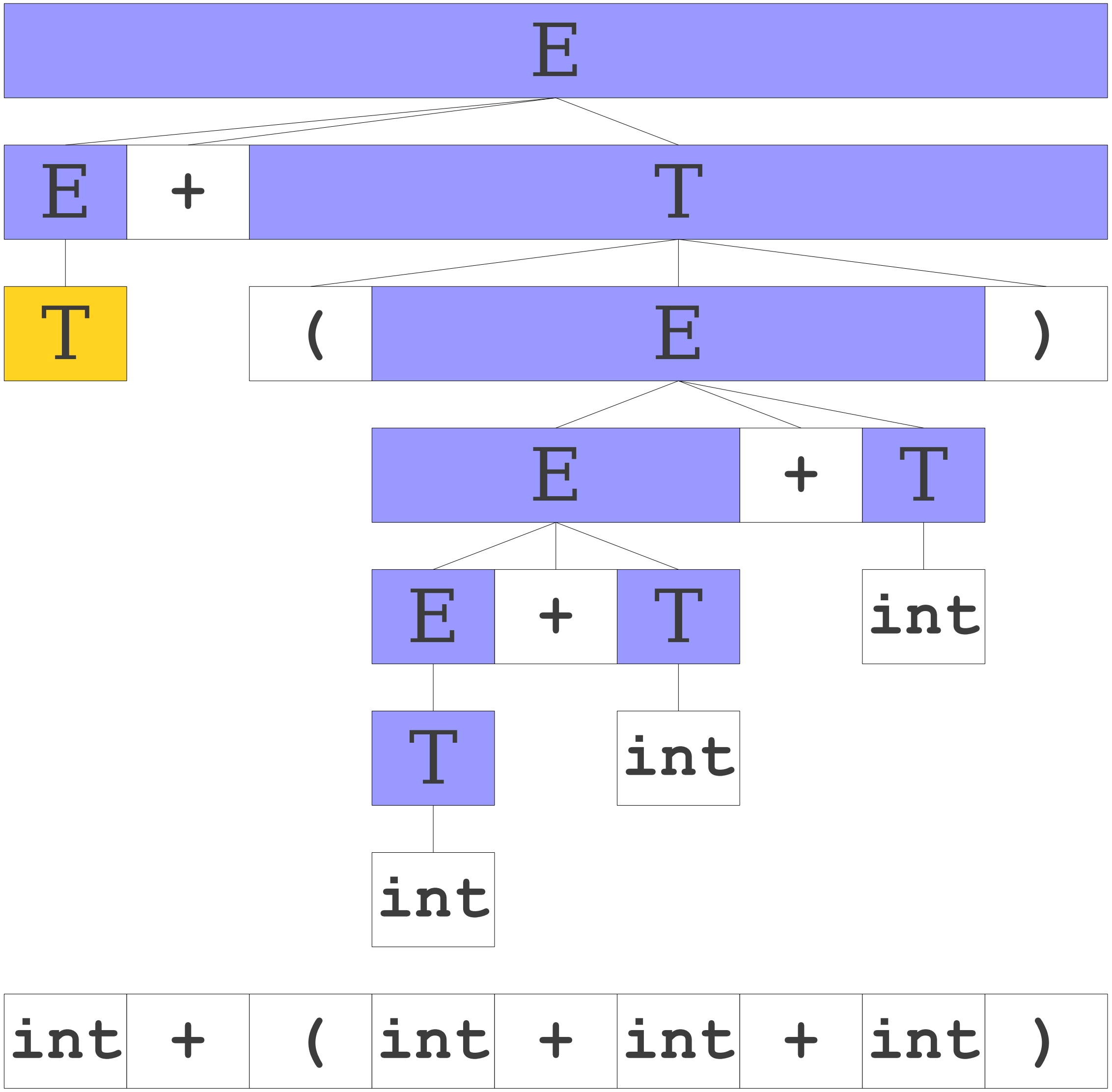
A Third View of a Bottom-Up Parse

⇒ **T** + (int + int + int)
⇒ **E** + (int + int + int)
⇒ **E** + (**T** + int + int)
⇒ **E** + (**E** + int + int)
⇒ **E** + (**E** + **T** + int)
⇒ **E** + (**E** + int)
⇒ **E** + (**E** + **T**)
⇒ **E** + (**E**)
⇒ **E** + **T**
⇒ **E**



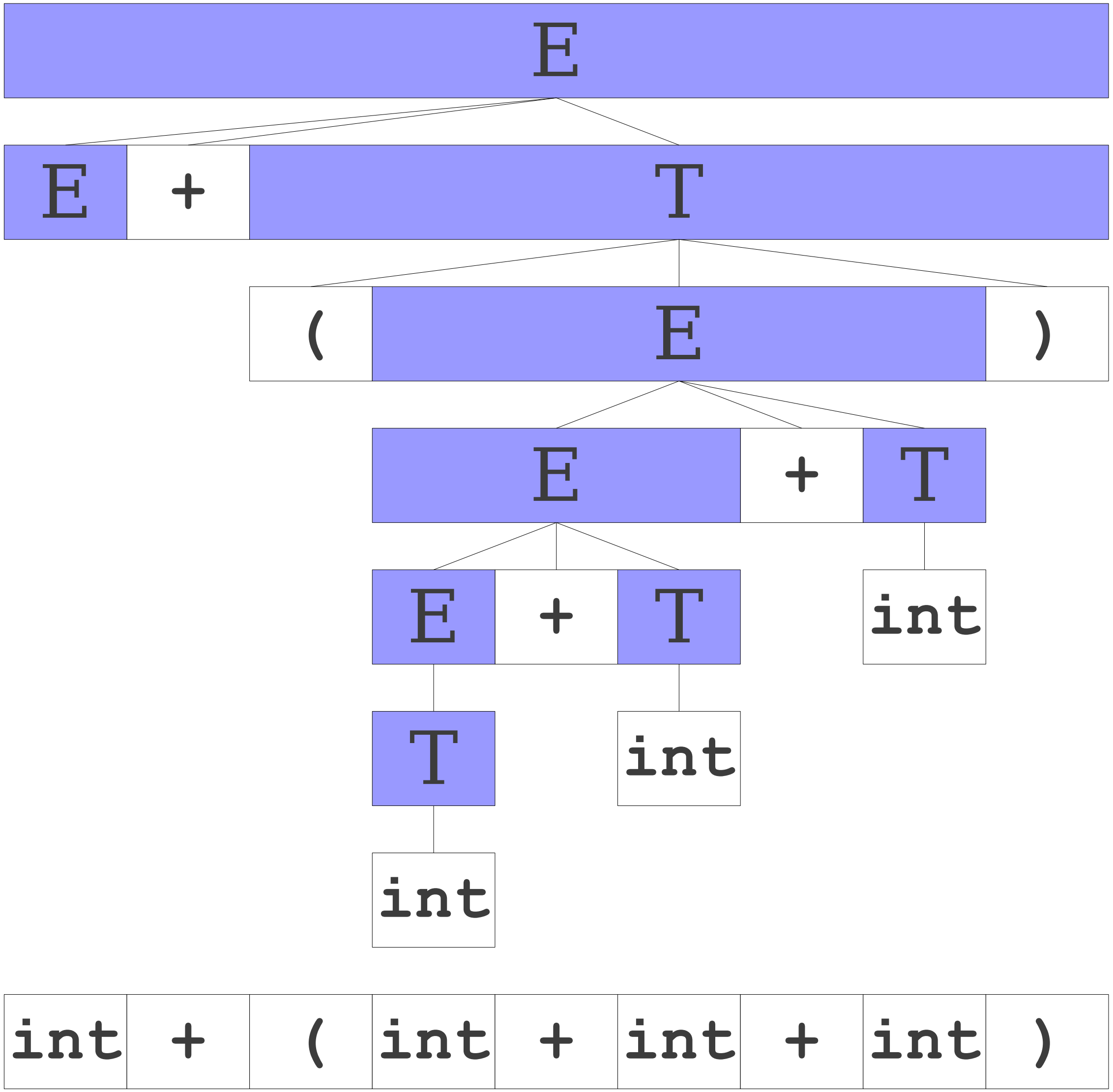
A Third View of a Bottom-Up Parse

⇒ **T** + (int + int + int)
⇒ **E** + (int + int + int)
⇒ **E** + (**T** + int + int)
⇒ **E** + (**E** + int + int)
⇒ **E** + (**E** + **T** + int)
⇒ **E** + (**E** + int)
⇒ **E** + (**E** + **T**)
⇒ **E** + (**E**)
⇒ **E** + **T**
⇒ **E**



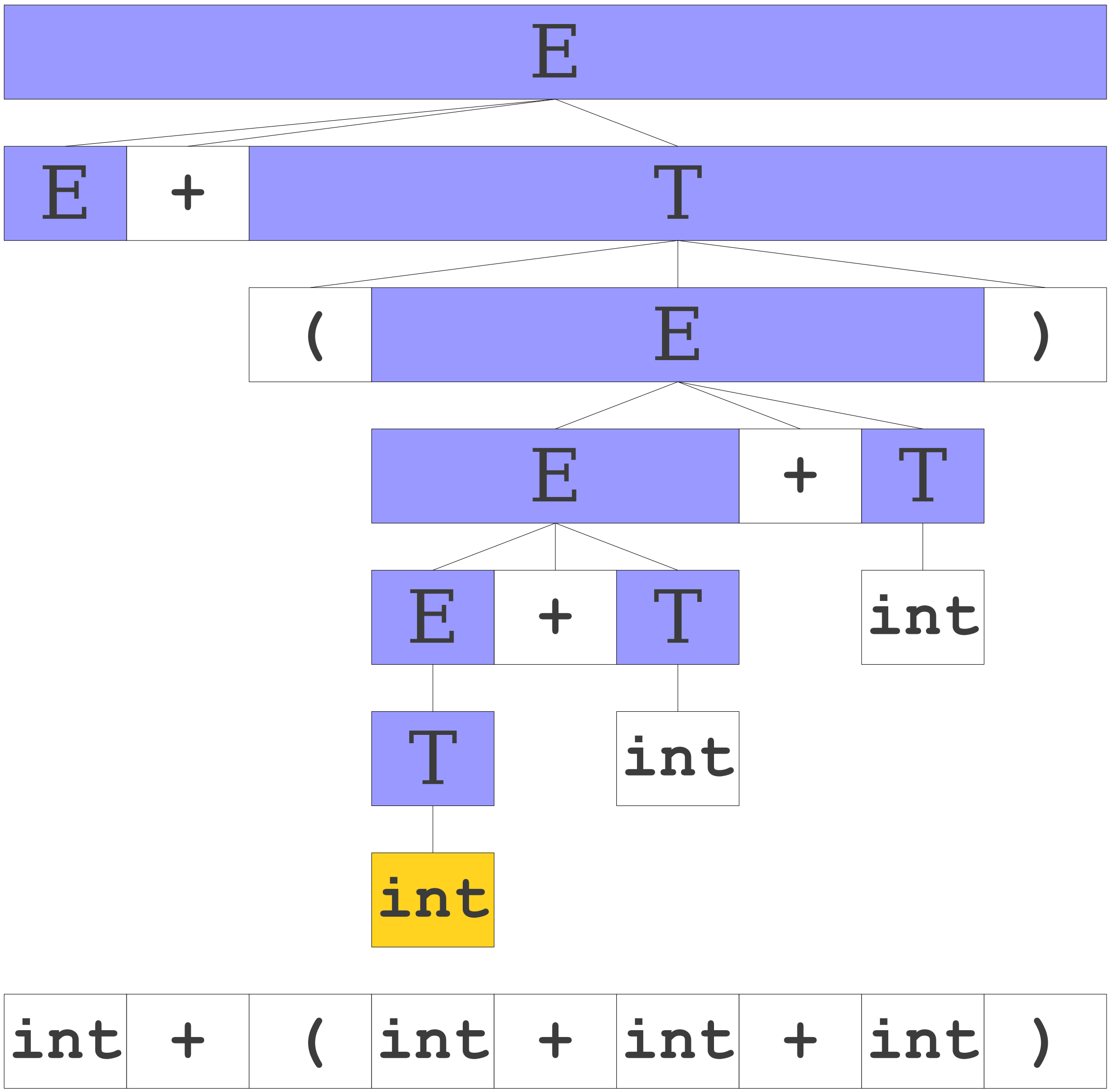
A Third View of a Bottom-Up Parse

⇒ **E** + (int + int + int)
⇒ **E** + (**T** + int + int)
⇒ **E** + (**E** + int + int)
⇒ **E** + (**E** + **T** + int)
⇒ **E** + (**E** + int)
⇒ **E** + (**E** + **T**)
⇒ **E** + (**E**)
⇒ **E** + **T**
⇒ **E**



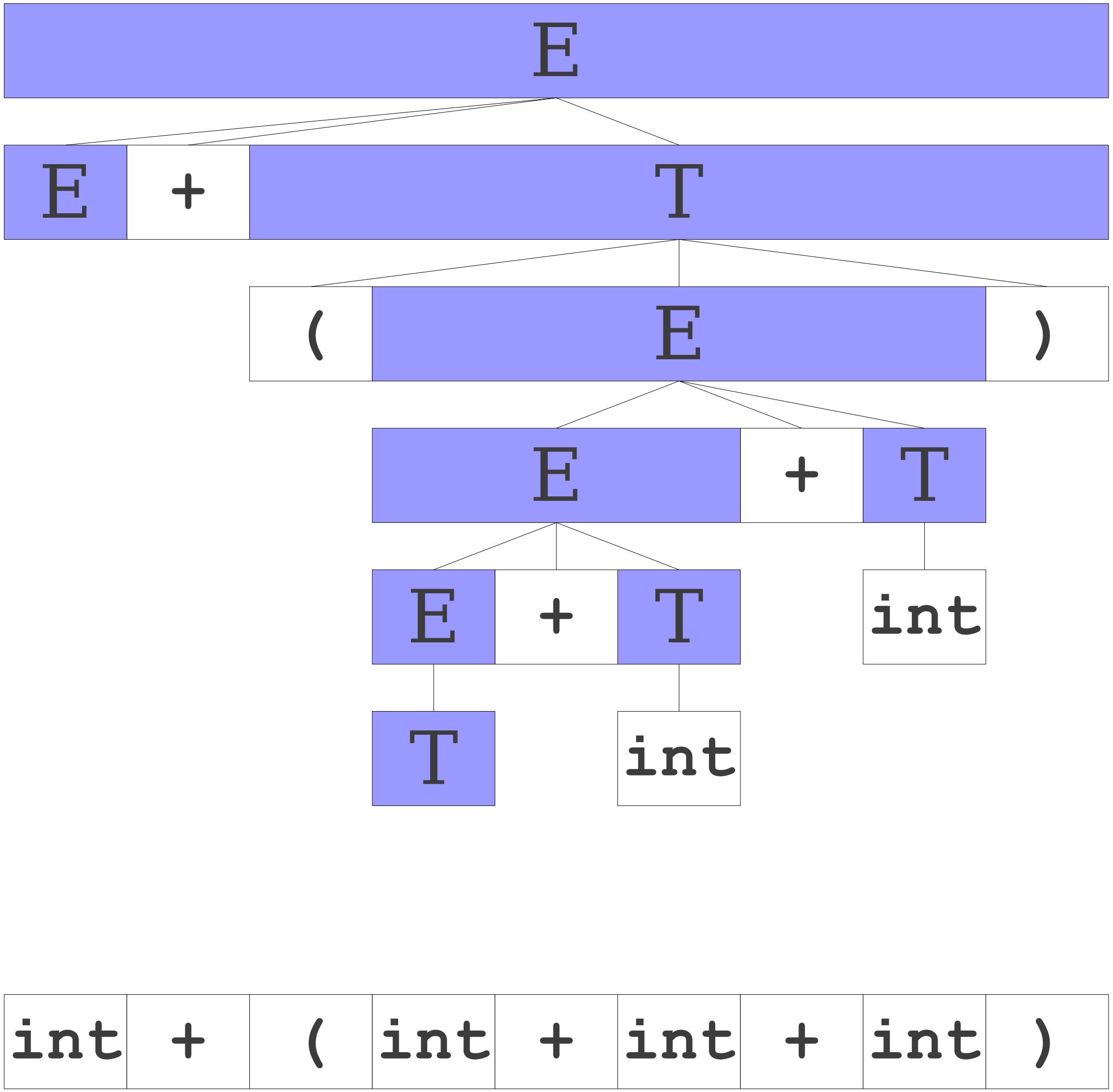
A Third View of a Bottom-Up Parse

⇒ **E** + (int + int + int)
⇒ **E** + (**T** + int + int)
⇒ **E** + (**E** + int + int)
⇒ **E** + (**E** + **T** + int)
⇒ **E** + (**E** + int)
⇒ **E** + (**E** + **T**)
⇒ **E** + (**E**)
⇒ **E** + **T**
⇒ **E**



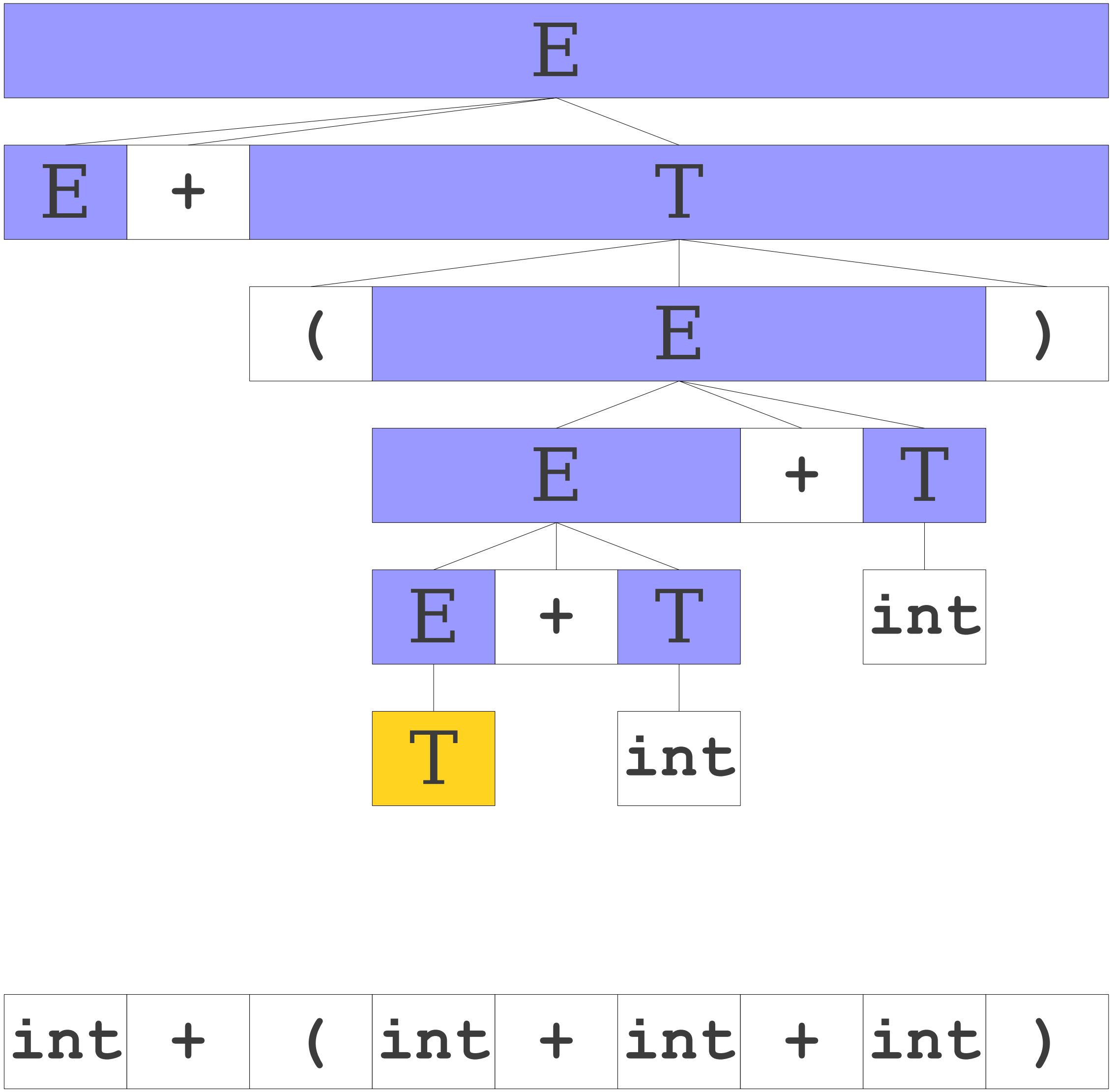
A Third View of a Bottom-Up Parse

⇒ **E** + (**T** + int + int)
⇒ **E** + (**E** + int + int)
⇒ **E** + (**E** + **T** + int)
⇒ **E** + (**E** + int)
⇒ **E** + (**E** + **T**)
⇒ **E** + (**E**)
⇒ **E** + **T**
⇒ **E**



A Third View of a Bottom-Up Parse

⇒ **E** + (**T** + int + int)
⇒ **E** + (**E** + int + int)
⇒ **E** + (**E** + **T** + int)
⇒ **E** + (**E** + int)
⇒ **E** + (**E** + **T**)
⇒ **E** + (**E**)
⇒ **E** + **T**
⇒ **E**



A Third View of a Bottom-Up Parse

⇒ **E** + (**E** + int + int)

⇒ **E** + (**E** + **T** + int)

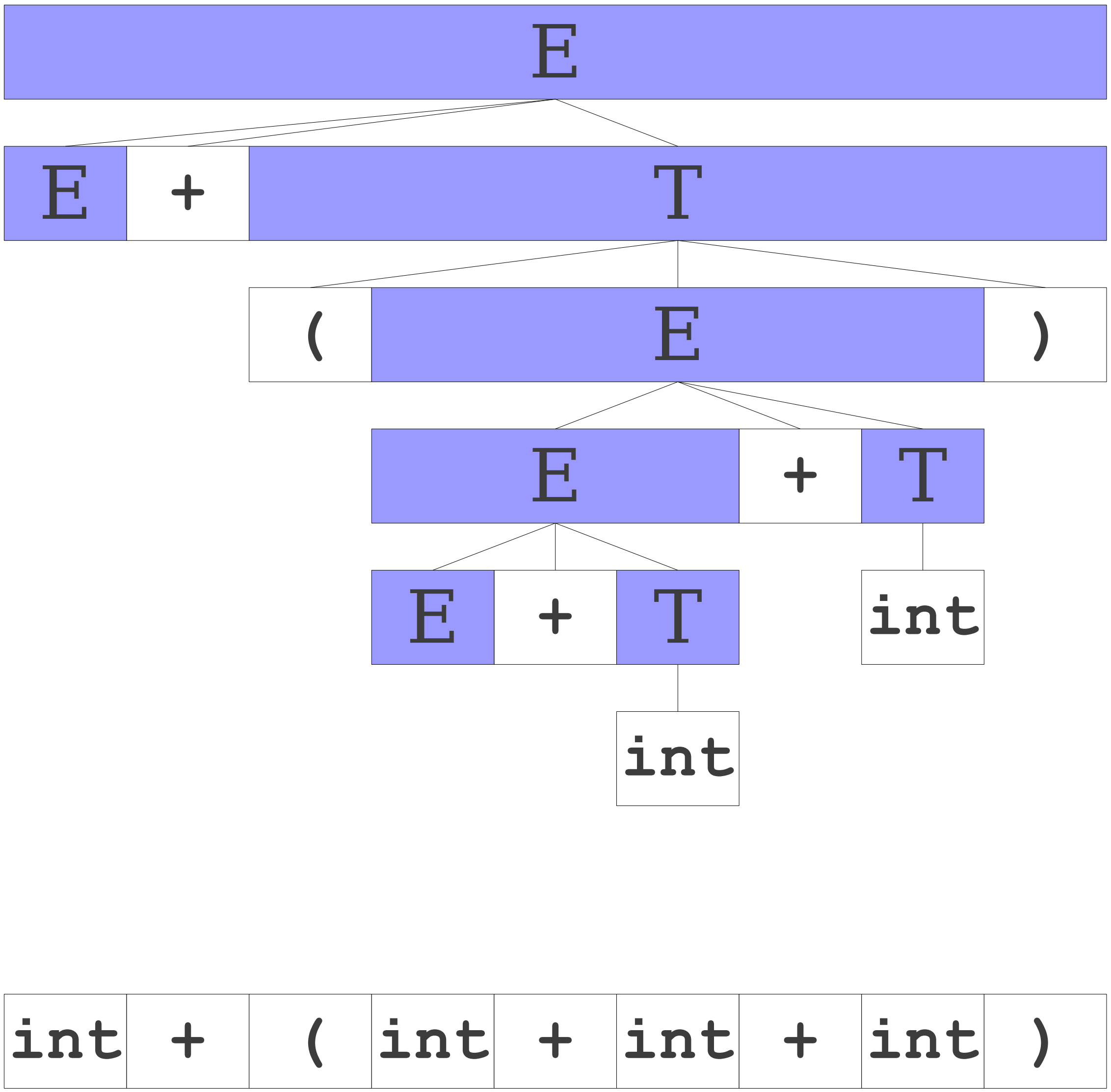
⇒ **E** + (**E** + int)

⇒ **E** + (**E** + **T**)

⇒ **E** + (**E**)

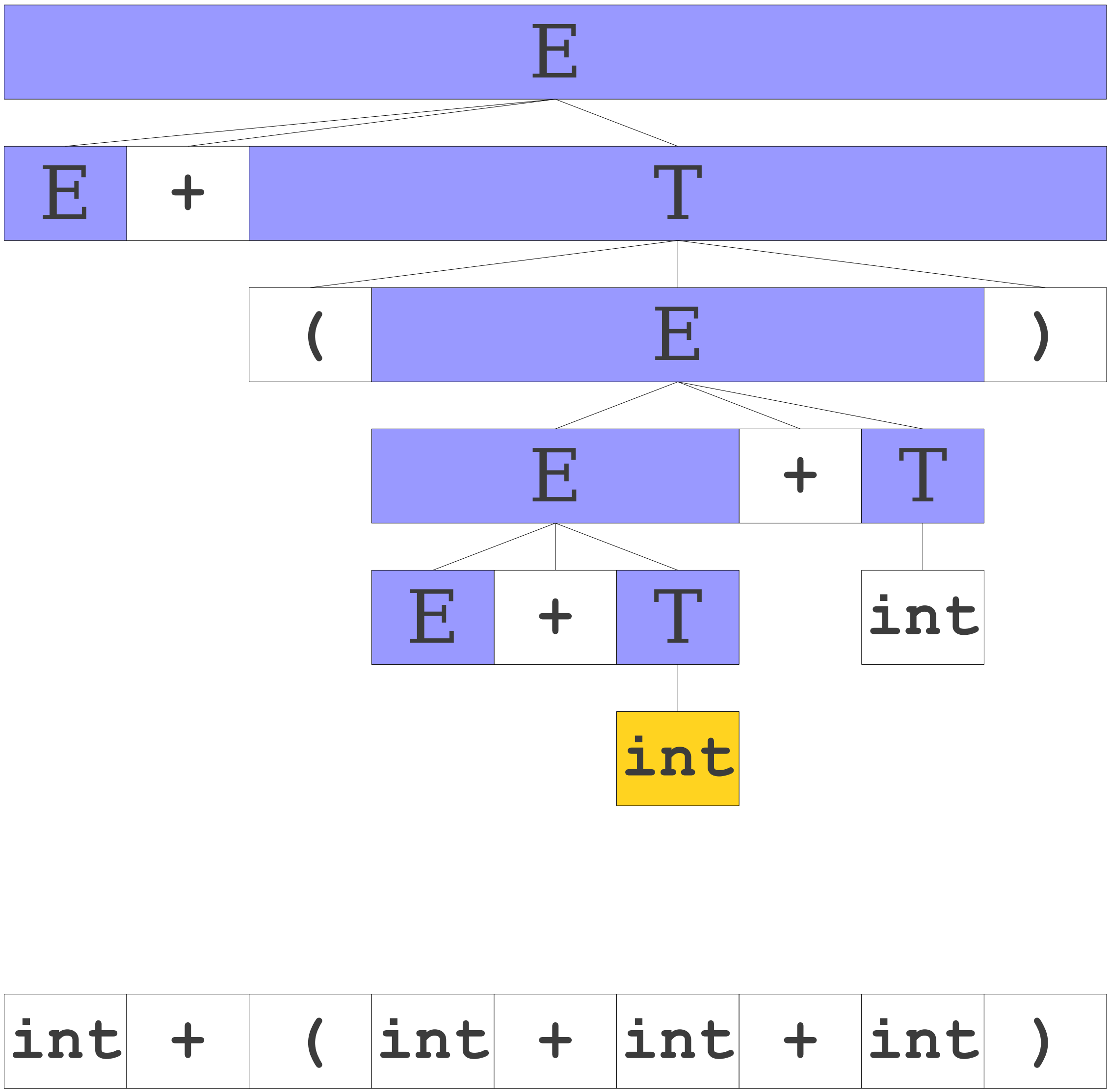
⇒ **E** + **T**

⇒ **E**

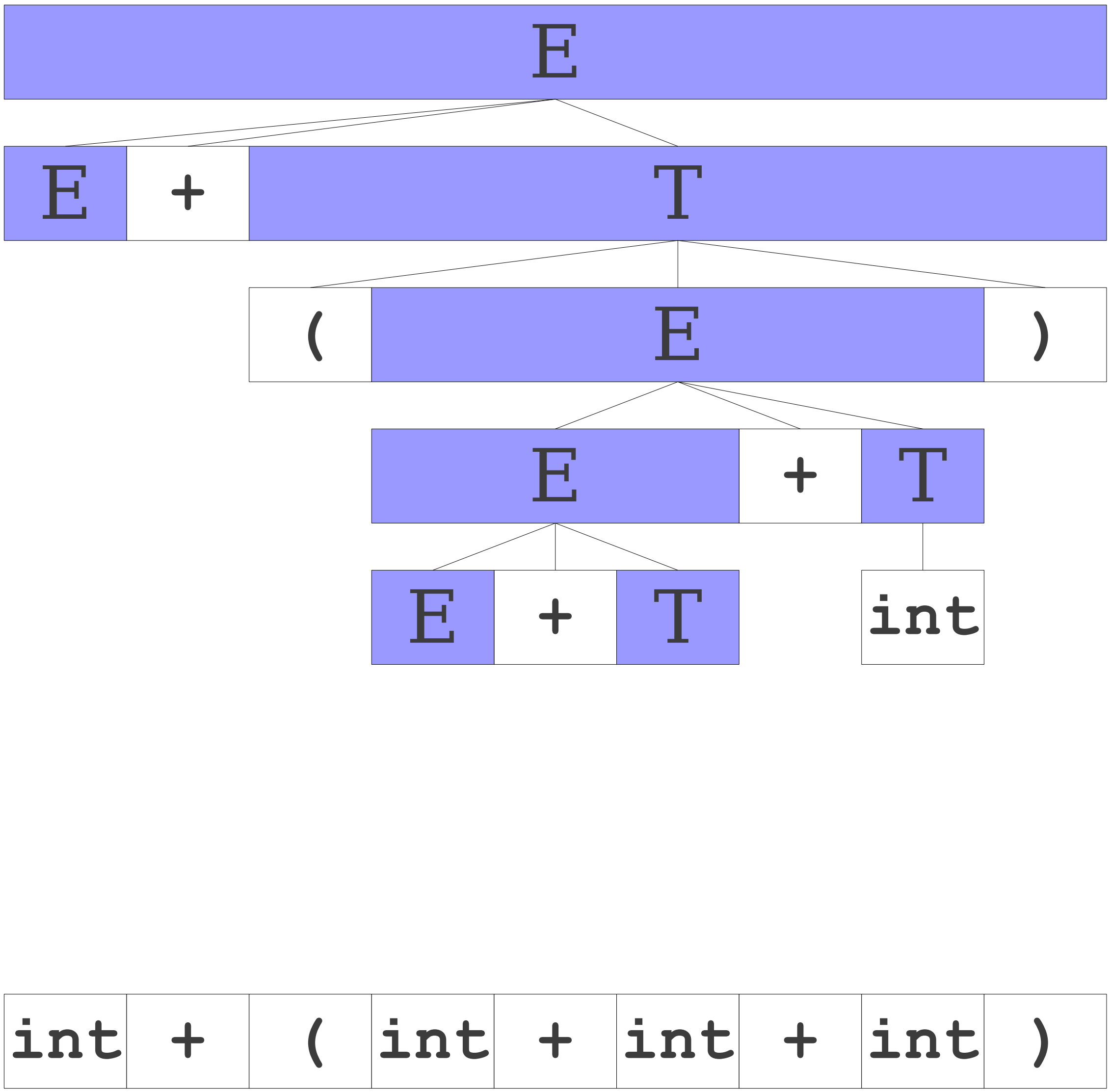


A Third View of a Bottom-Up Parse

⇒ **E** + (**E** + int + int)
⇒ **E** + (**E** + **T** + int)
⇒ **E** + (**E** + int)
⇒ **E** + (**E** + **T**)
⇒ **E** + (**E**)
⇒ **E** + **T**
⇒ **E**

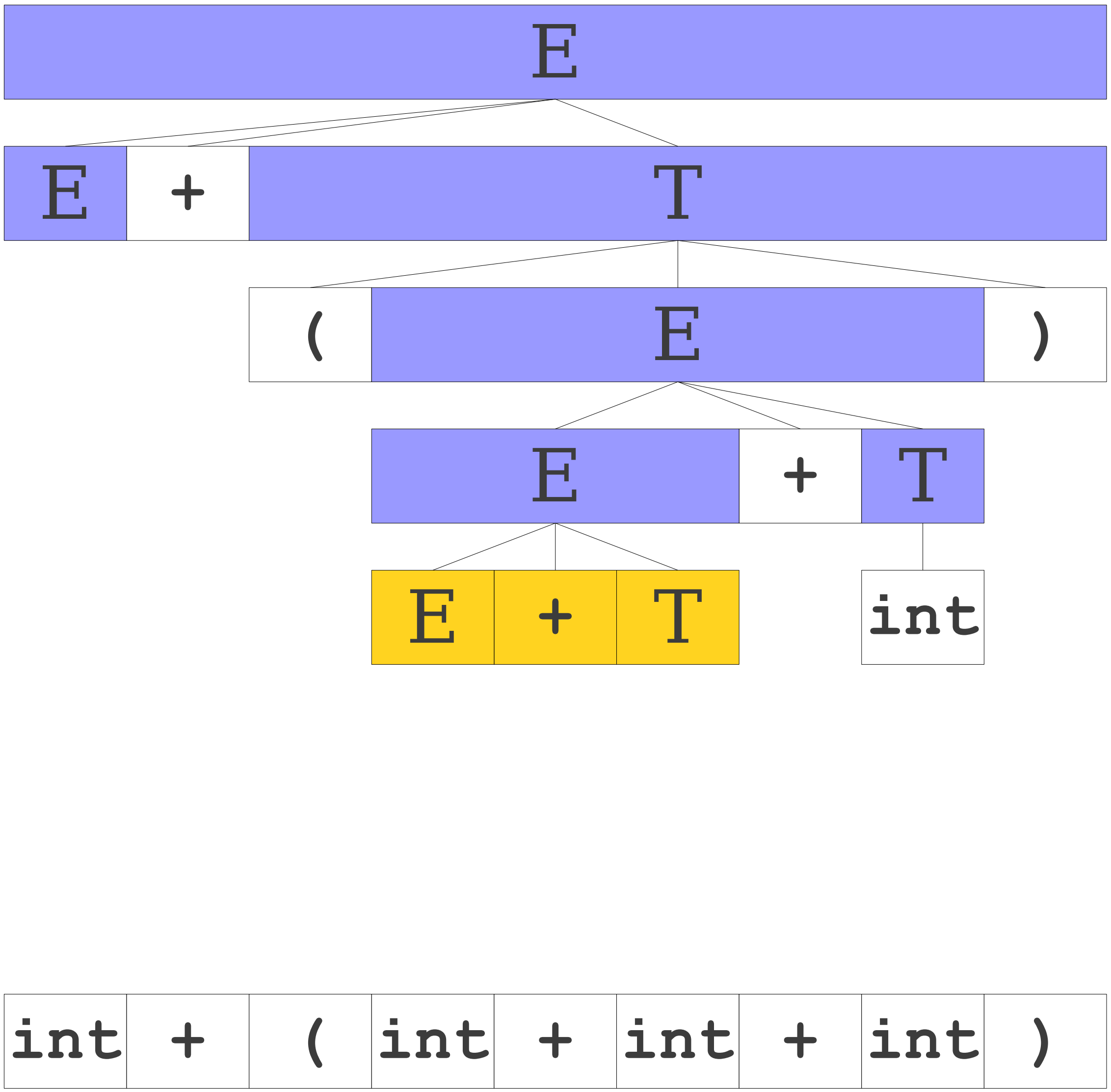


A Third View of a Bottom-Up Parse



⇒ **E** + (**E** + **T** + int)
⇒ **E** + (**E** + int)
⇒ **E** + (**E** + **T**)
⇒ **E** + (**E**)
⇒ **E** + **T**
⇒ **E**

A Third View of a Bottom-Up Parse



⇒ **E + (E + T + int)**

⇒ **E + (E + int)**

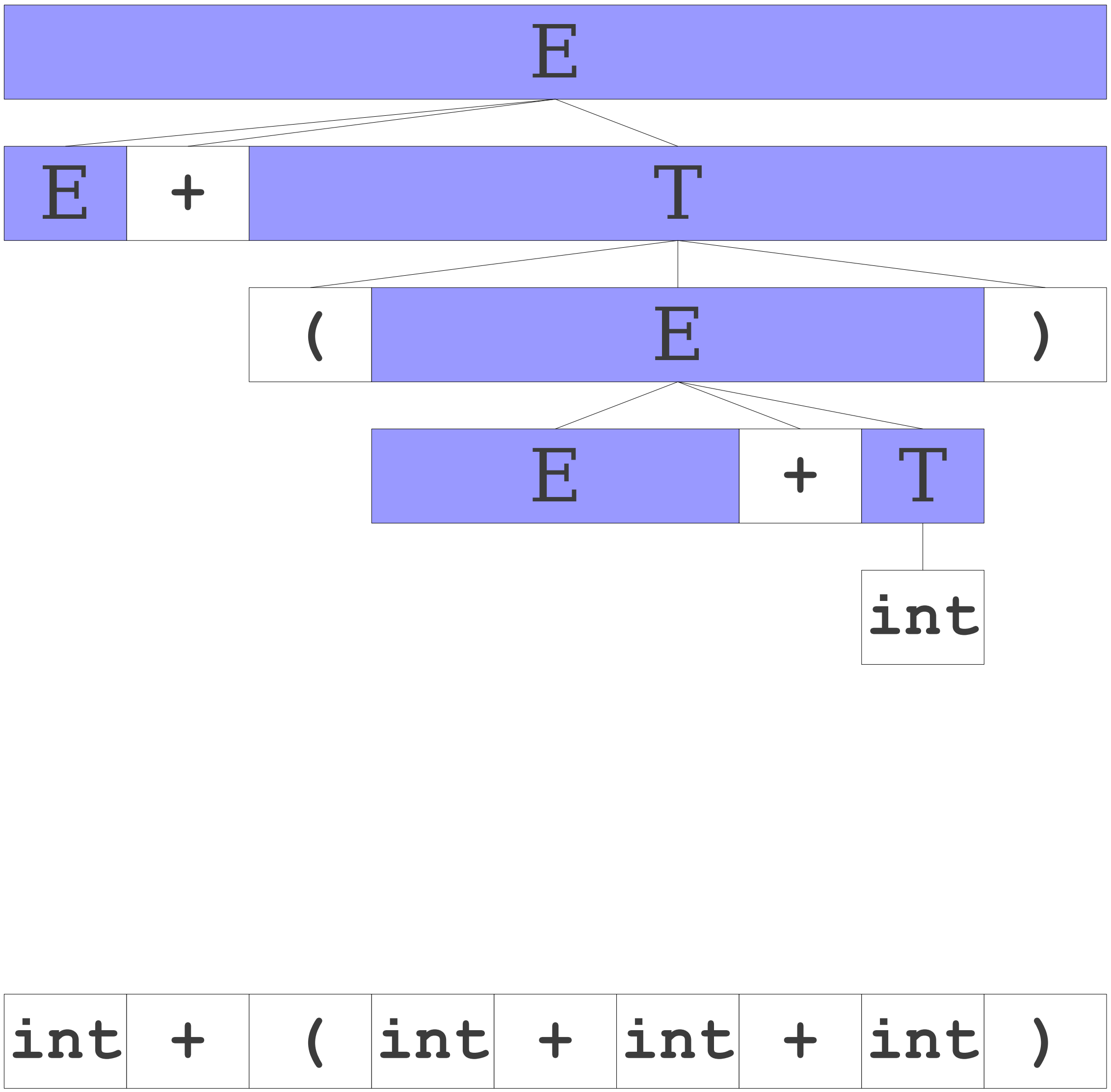
⇒ **E + (E + T)**

⇒ **E + (E)**

⇒ **E + T**

⇒ **E**

A Third View of a Bottom-Up Parse



$\Rightarrow$ **E + (E + int)**

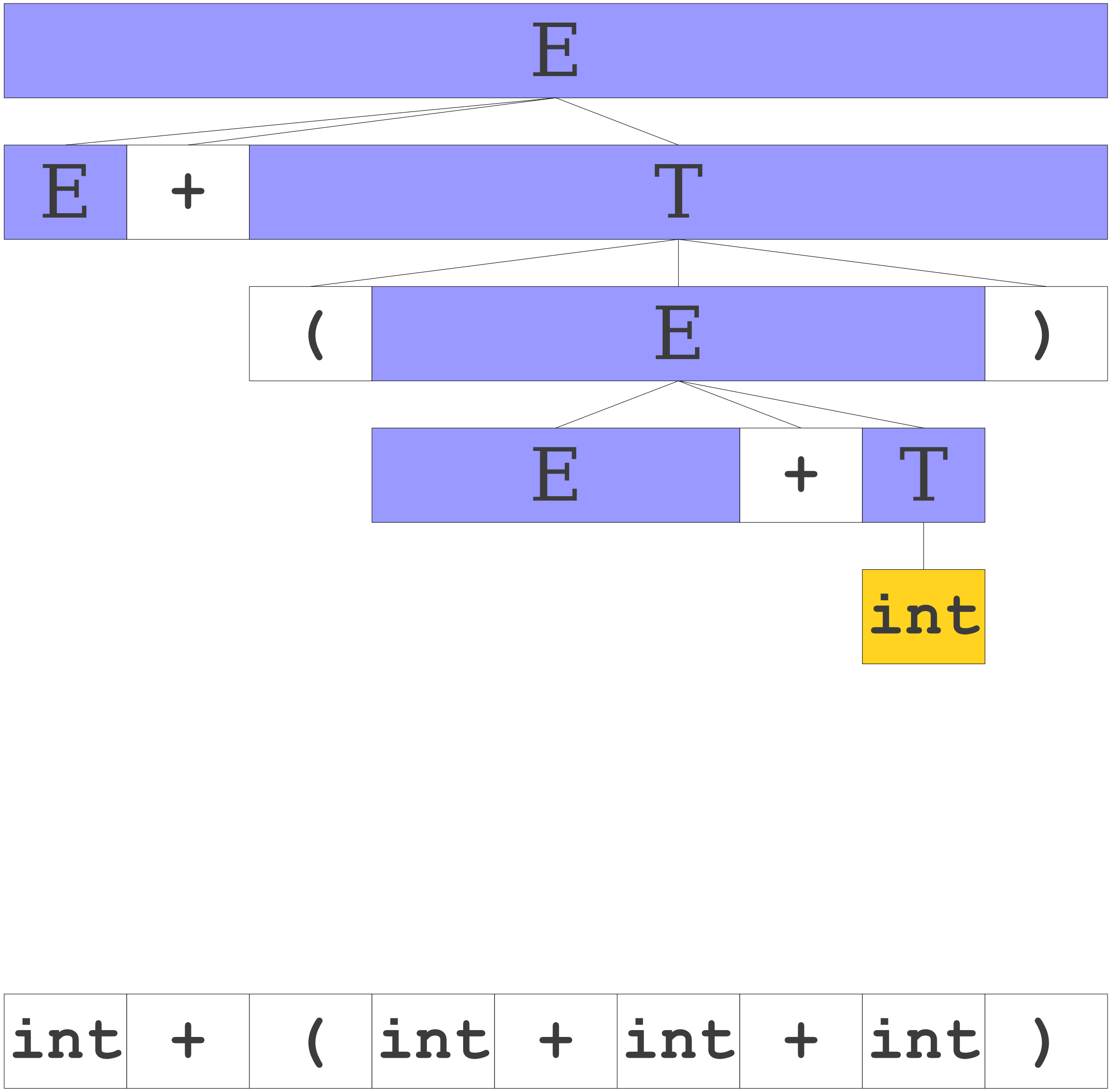
$\Rightarrow$ **E + (E + T)**

$\Rightarrow$ **E + (E)**

$\Rightarrow$ **E + T**

$\Rightarrow$ **E**

A Third View of a Bottom-Up Parse



⇒ **E** + (**E** + int)

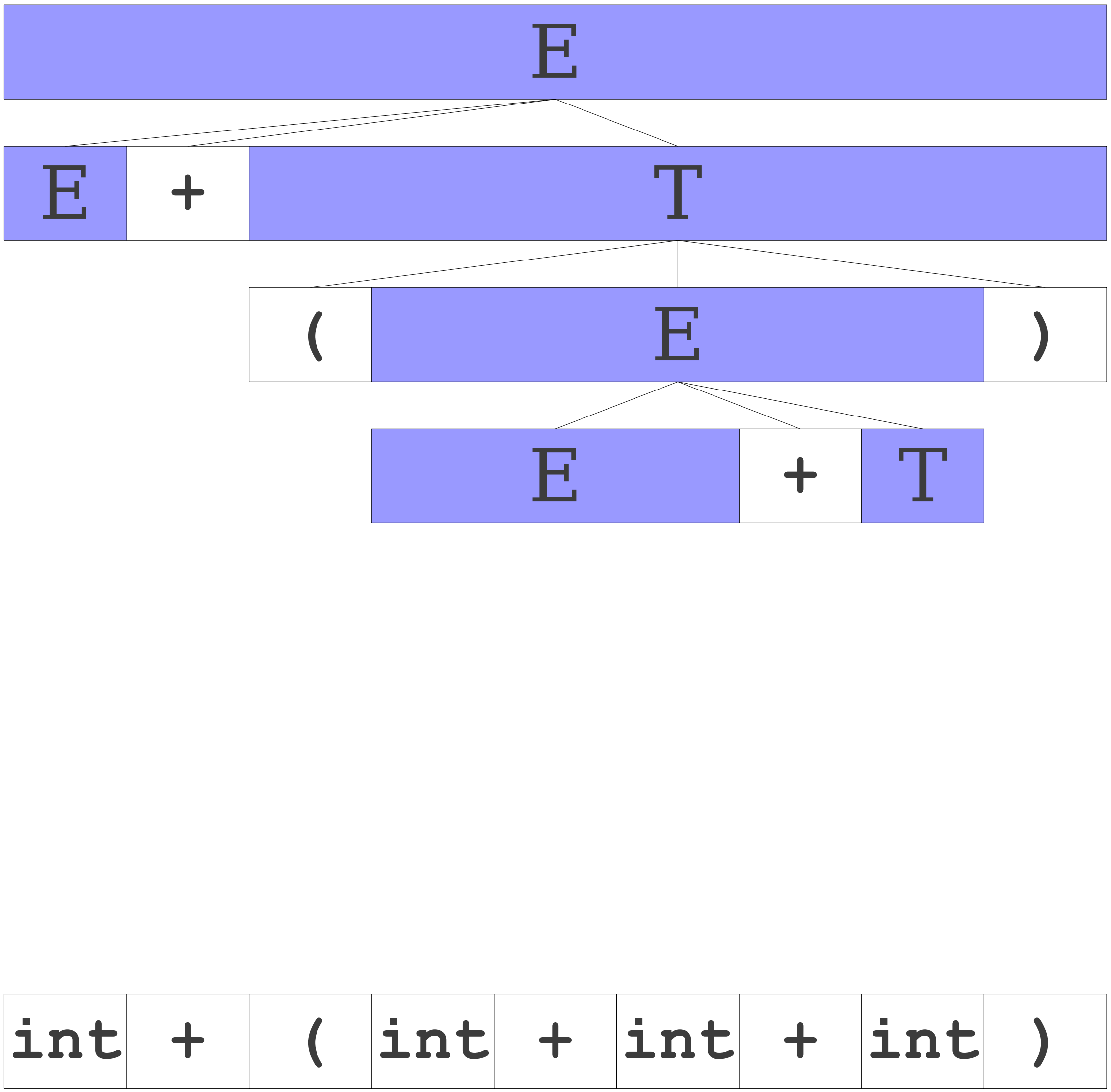
⇒ **E** + (**E** + **T**)

⇒ **E** + (**E**)

⇒ **E** + **T**

⇒ **E**

A Third View of a Bottom-Up Parse



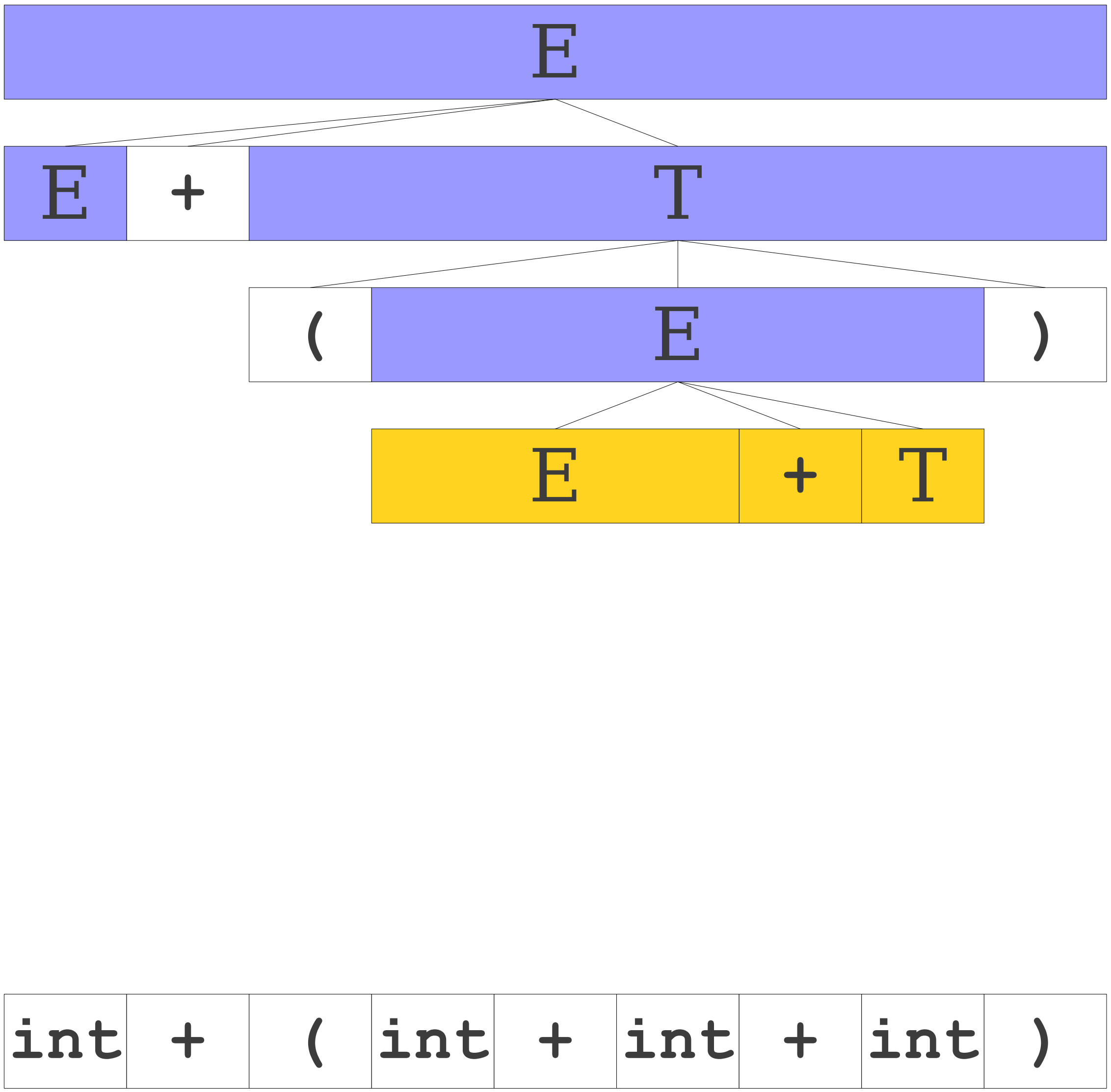
⇒ **E + (E + T)**

⇒ **E + (E)**

⇒ **E + T**

⇒ **E**

A Third View of a Bottom-Up Parse



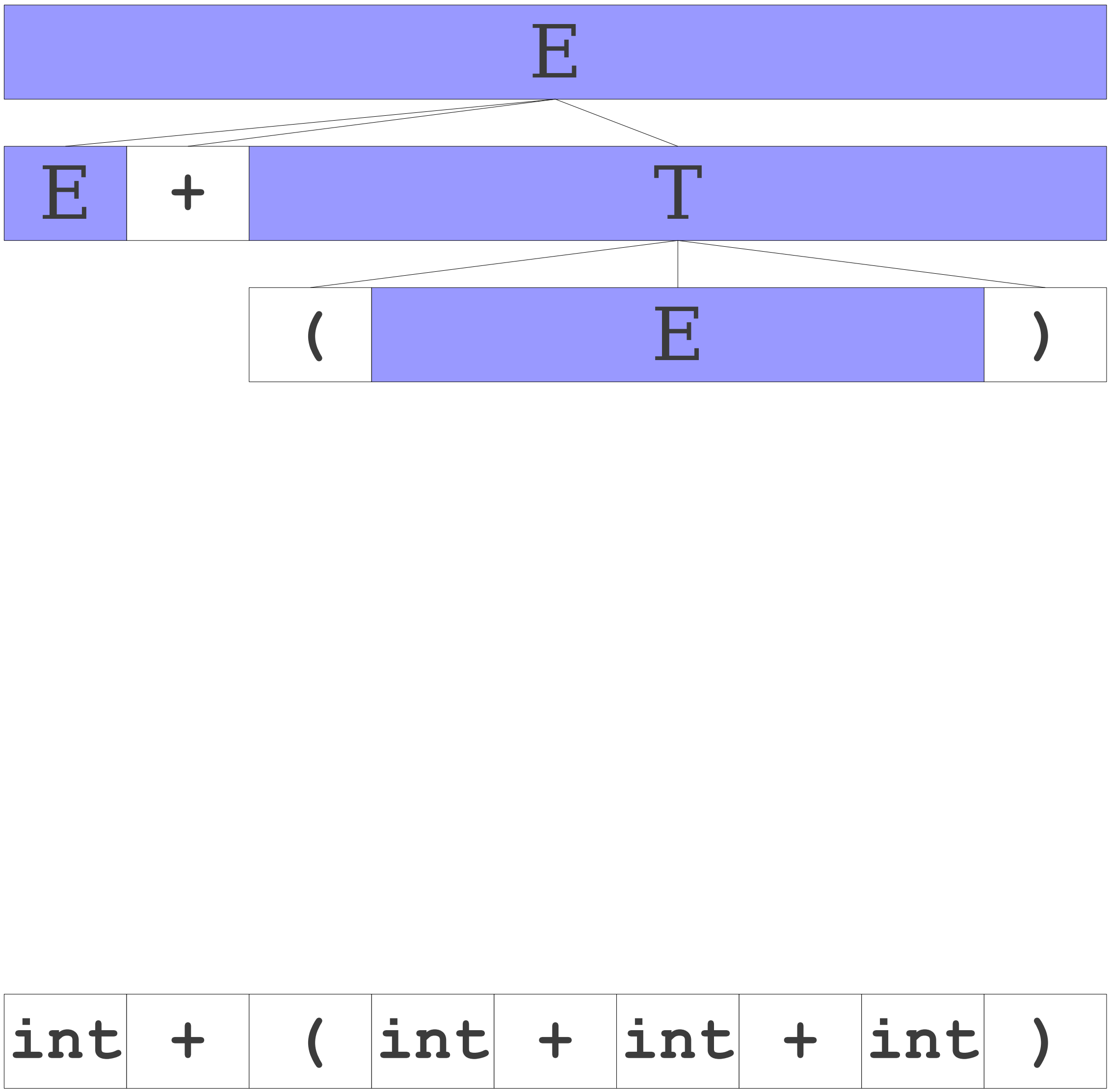
⇒ **E + (E + T)**

⇒ **E + (E)**

⇒ **E + T**

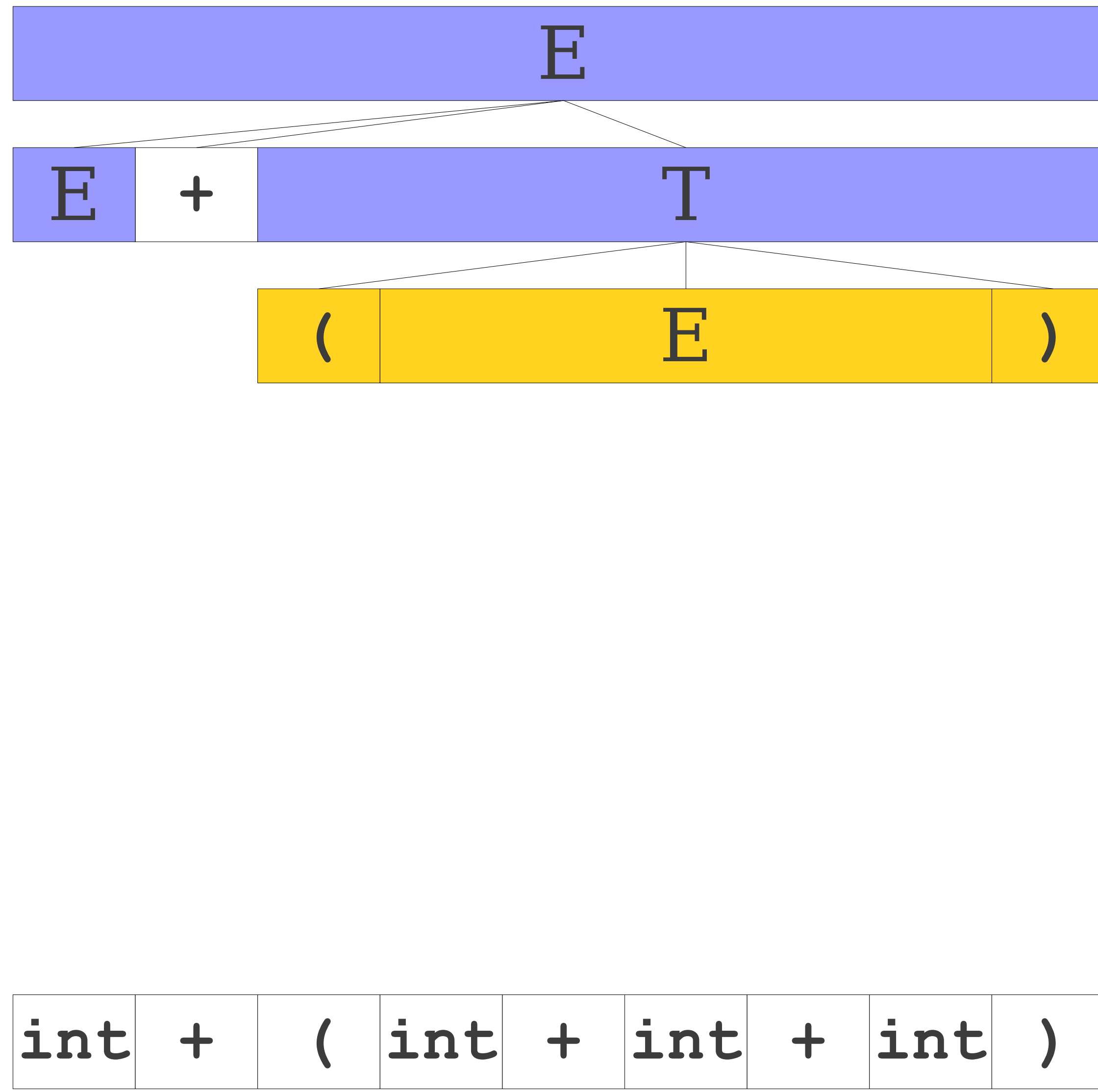
⇒ **E**

A Third View of a Bottom-Up Parse



⇒ **E** + (**E**)
⇒ **E** + **T**
⇒ **E**

A Third View of a Bottom-Up Parse

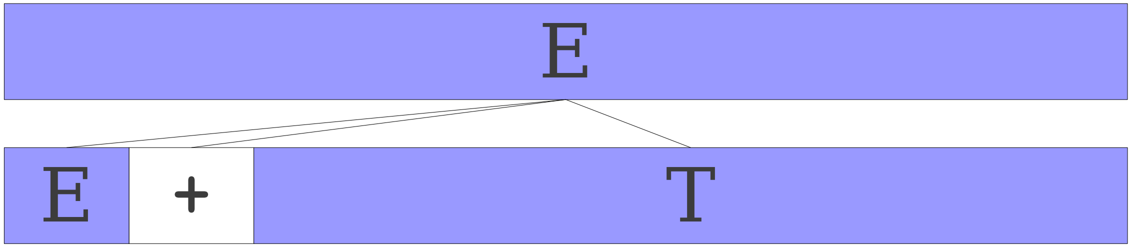


$\Rightarrow$ **E + (E)**

$\Rightarrow$ **E + T**

$\Rightarrow$ **E**

A Third View of a Bottom-Up Parse

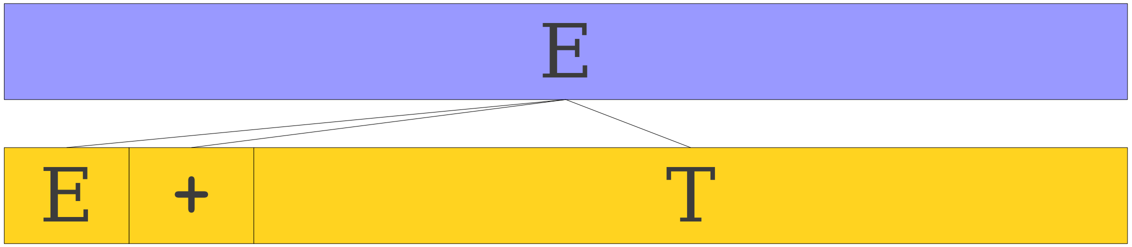


⇒ **E + T**

⇒ **E**

`int`	`+`	`(`	`int`	`+`	`int`	`+`	`int`	`)`

A Third View of a Bottom-Up Parse



⇒ **E + T**

⇒ **E**

`int`	`+`	`(`	`int`	`+`	`int`	`+`	`int`	`)`

A Third View of a Bottom-Up Parse

E

⇒ **E**

`int`	`+`	`(`	`int`	`+`	`int`	`+`	`int`	`)`

Handles

- Informally, a "handle" is a substring that matches the body of a production, and whose reduction represents one step along the reverse of a rightmost derivation.

- A left-to-right, bottom-up parse works by iteratively searching for a handle, then reducing the handle.

Shift/Reduce Parsing

- The bottom-up parsers we will consider are called **shift/reduce** parsers.
  - Contrast with the LL(1) **predict/match** parser.
- Idea: Split the input into two parts:
  - Left substring is our work area; all handles must be here.
  - Right substring is input we have not yet processed; consists purely of terminals.
- At each point, decide whether to:
  - Move a terminal across the split (**shift**)
  - Reduce a handle (**reduce**)

A Sample Shift/Reduce Parse

$\mathbf{E} \to \mathbf{F}$
$\mathbf{E} \to \mathbf{E}$ `+` $\mathbf{F}$
$\mathbf{F} \to \mathbf{F}$ `*` $\mathbf{T}$
$\mathbf{F} \to \mathbf{T}$
$\mathbf{T} \to$ `int`
$\mathbf{T} \to$ `(`$\mathbf{E}$`)`

`int`	`+`	`int`	`*`	`int`	`+`	`int`

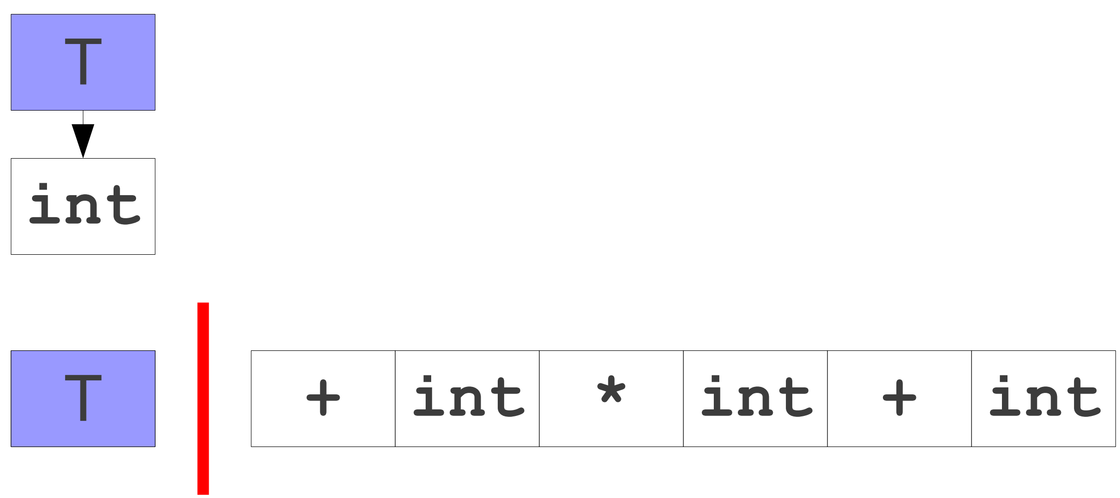
A Sample Shift/Reduce Parse

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

int	|	+	int	*	int	+	int

A Sample Shift/Reduce Parse

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



A Sample Shift/Reduce Parse

$\mathbf{E} \rightarrow \mathbf{F}$

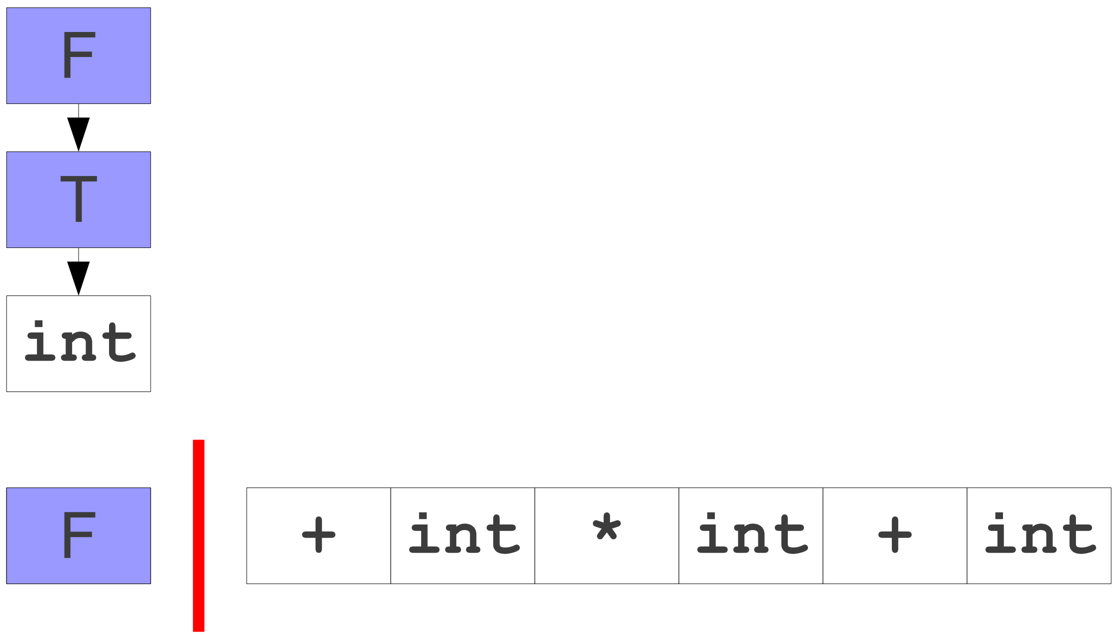
$\mathbf{E} \rightarrow \mathbf{E}$ **+** $\mathbf{F}$

$\mathbf{F} \rightarrow \mathbf{F}$ ***** $\mathbf{T}$

$\mathbf{F} \rightarrow \mathbf{T}$

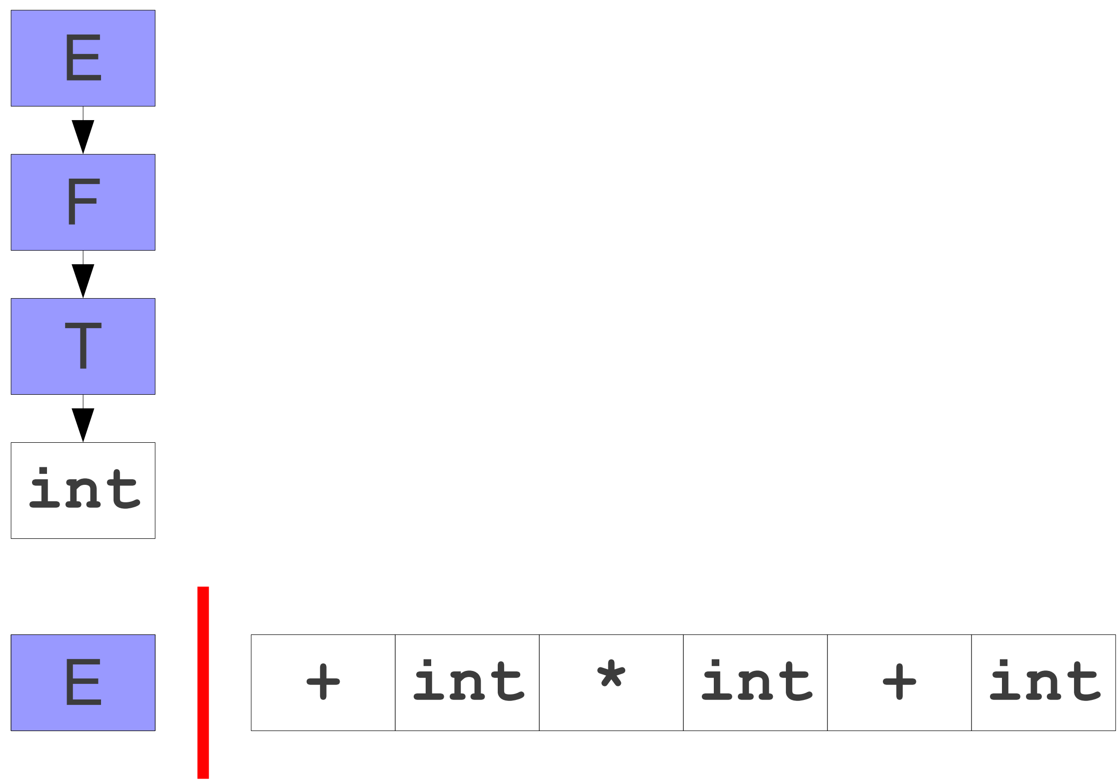
$\mathbf{T} \rightarrow$ **int**

$\mathbf{T} \rightarrow$ **(E)**



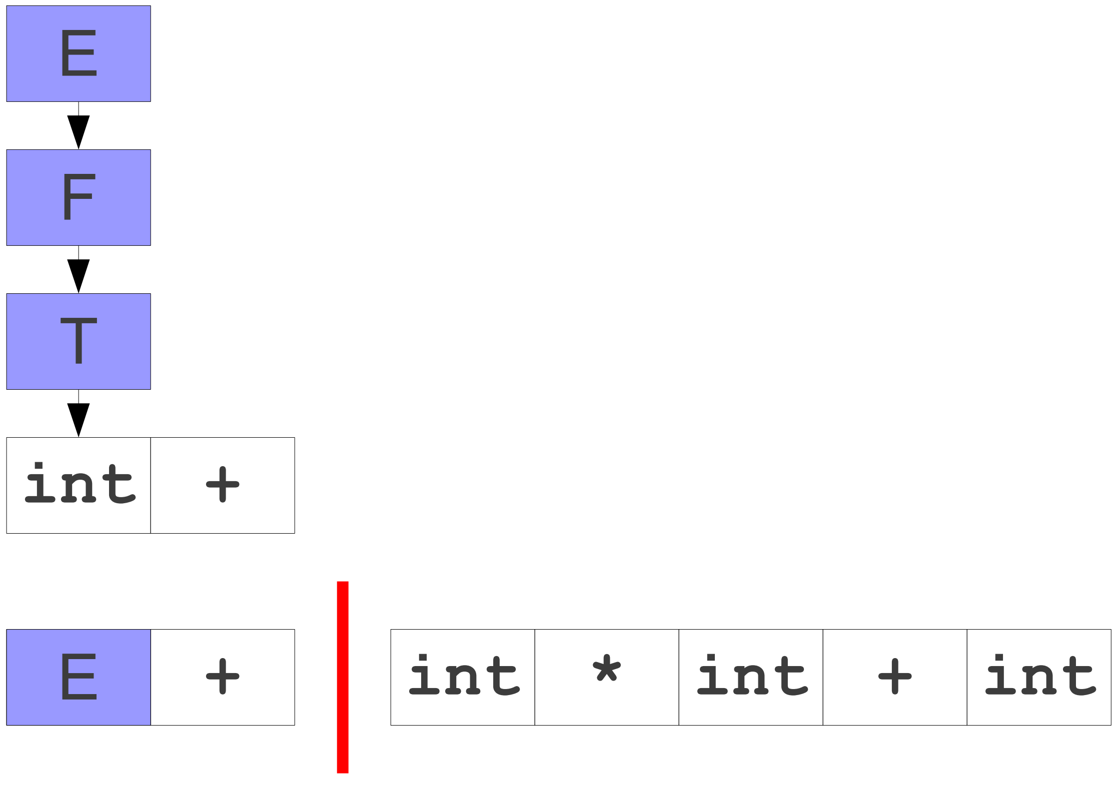
A Sample Shift/Reduce Parse

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



A Sample Shift/Reduce Parse

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



A Sample Shift/Reduce Parse

$\mathbf{E} \rightarrow \mathbf{F}$

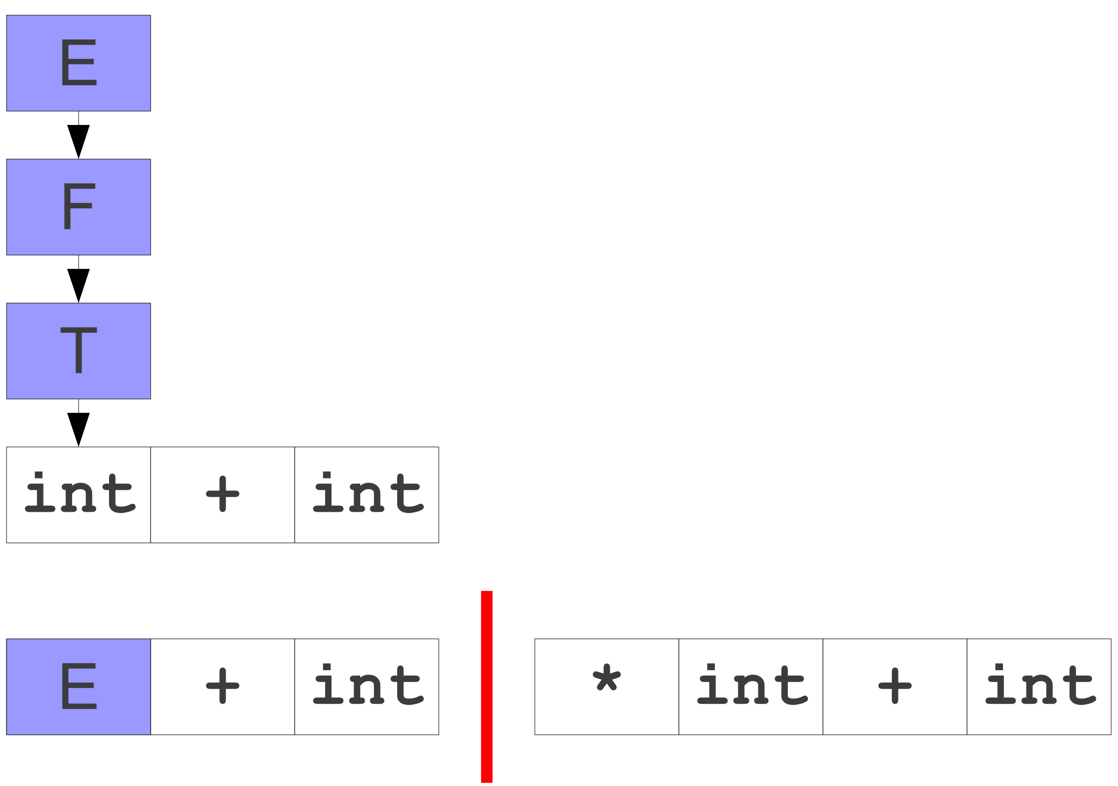
$\mathbf{E} \rightarrow \mathbf{E} + \mathbf{F}$

$\mathbf{F} \rightarrow \mathbf{F} * \mathbf{T}$

$\mathbf{F} \rightarrow \mathbf{T}$

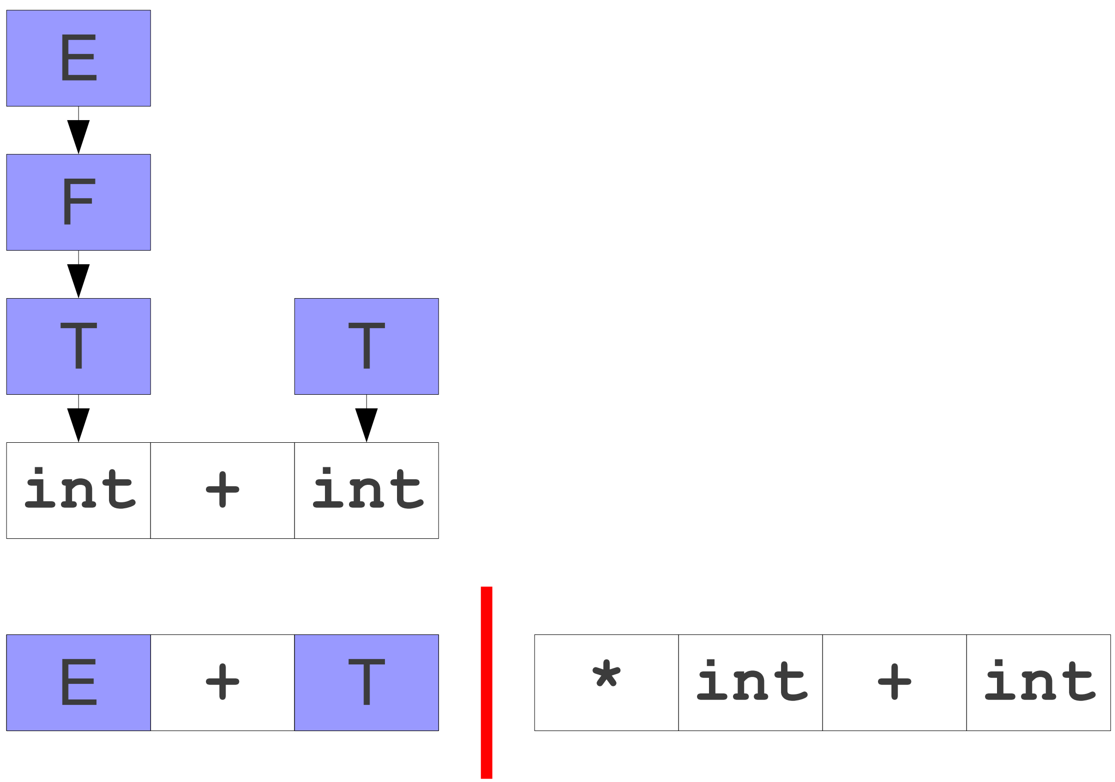
$\mathbf{T} \rightarrow$ **int**

$\mathbf{T} \rightarrow$ **(E)**



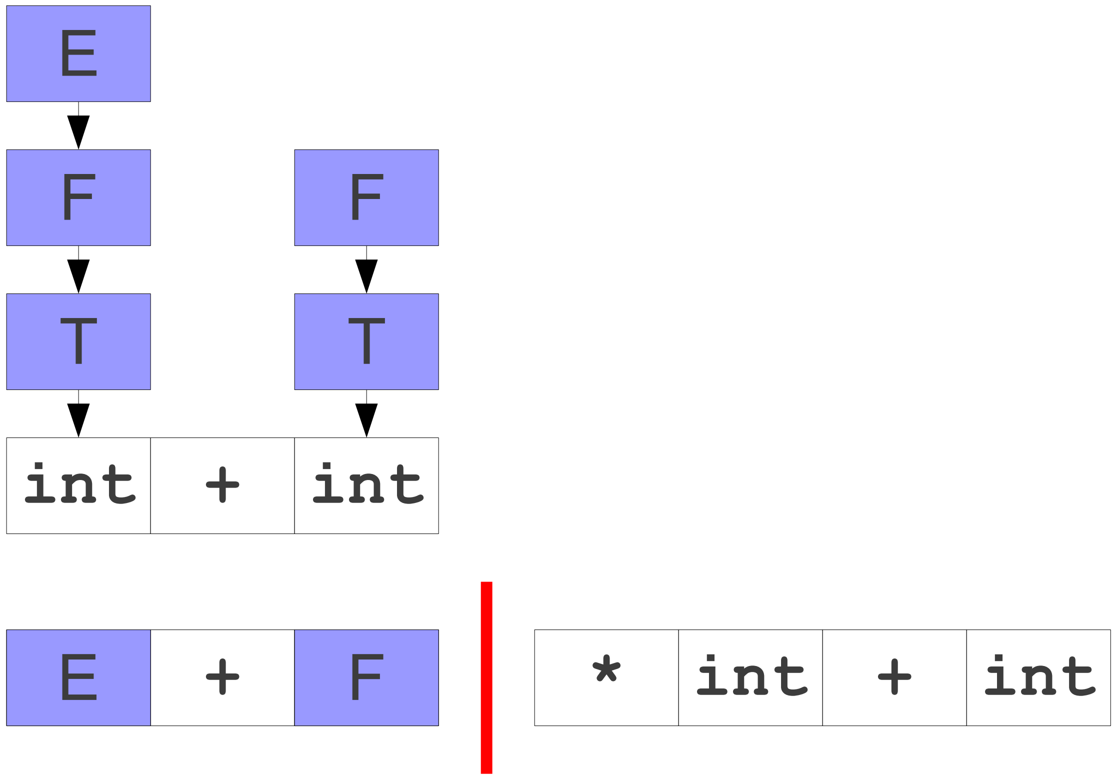
A Sample Shift/Reduce Parse

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



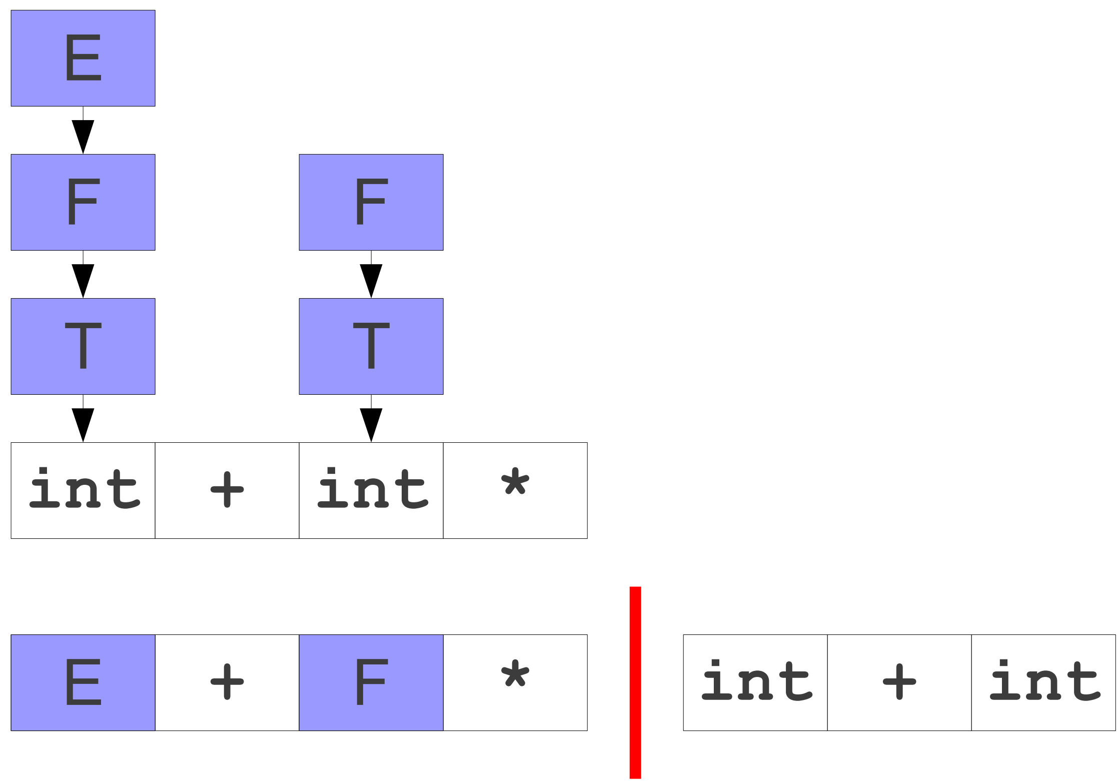
A Sample Shift/Reduce Parse

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



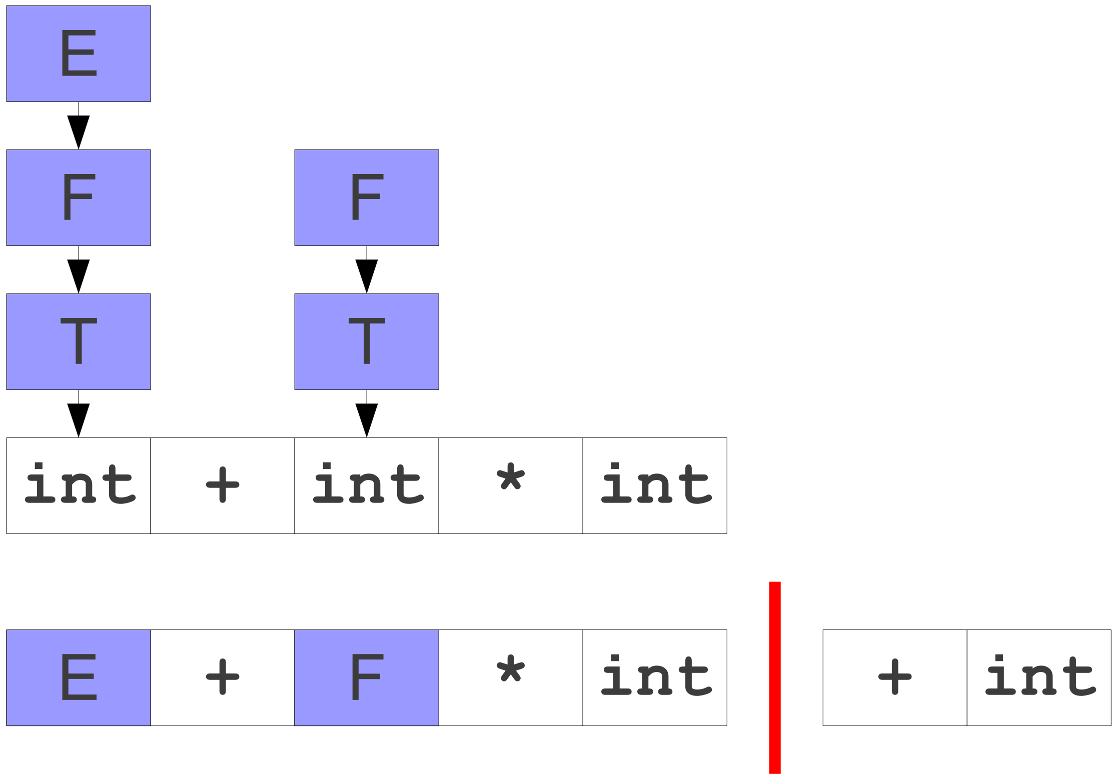
A Sample Shift/Reduce Parse

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



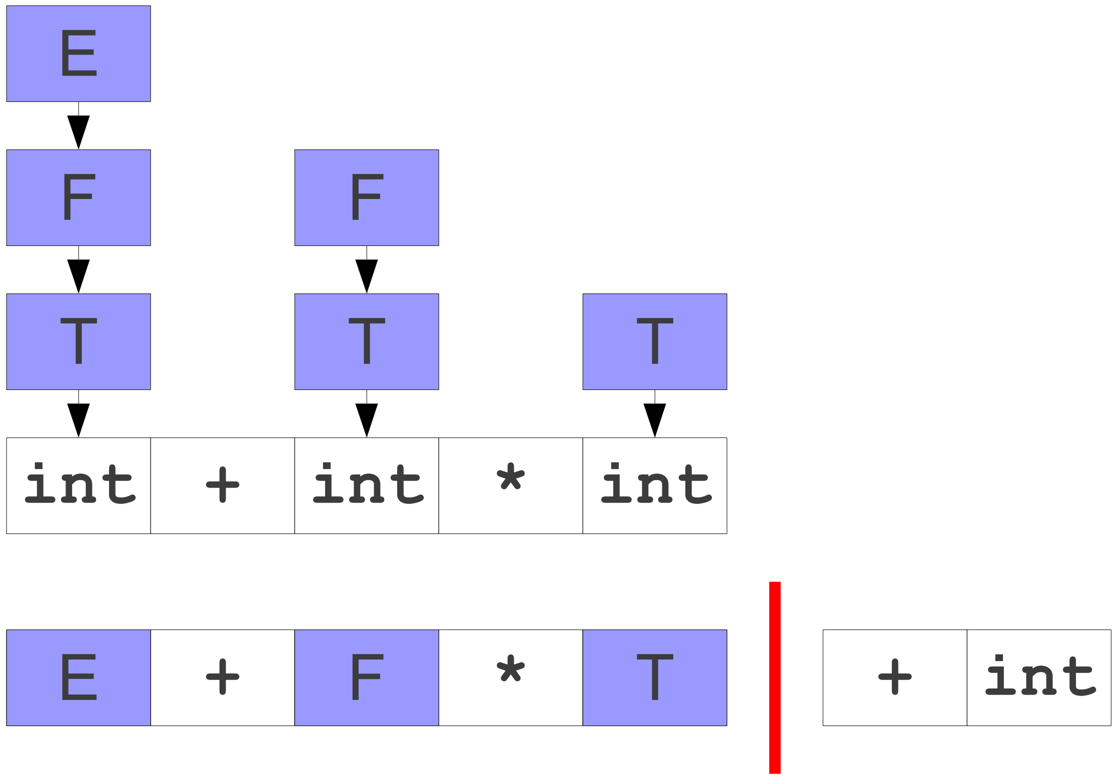
A Sample Shift/Reduce Parse

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



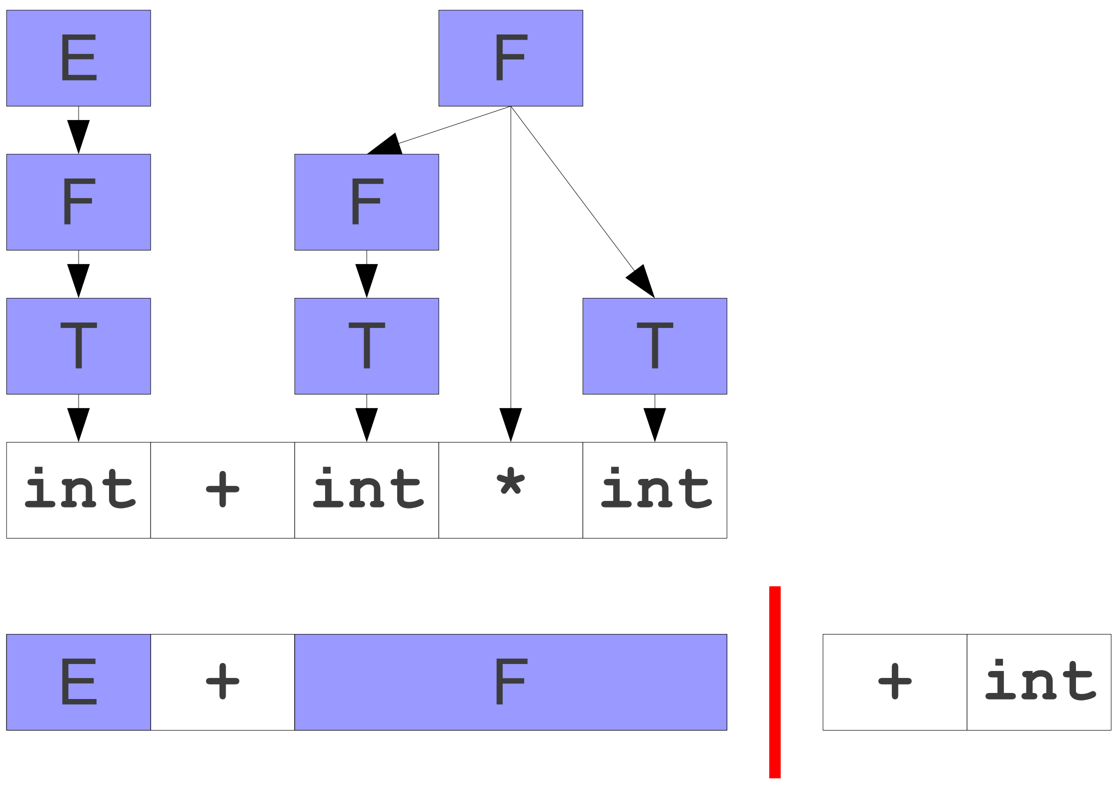
A Sample Shift/Reduce Parse

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



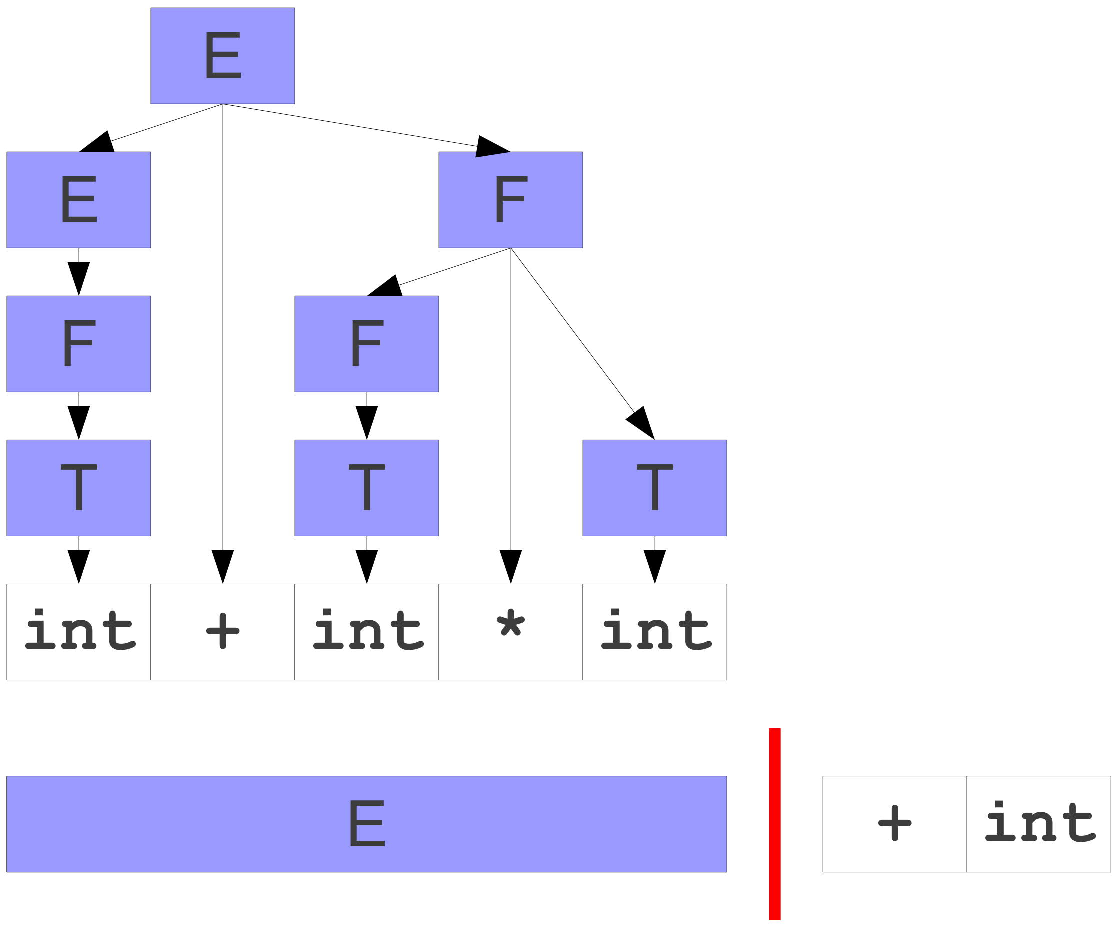
A Sample Shift/Reduce Parse

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



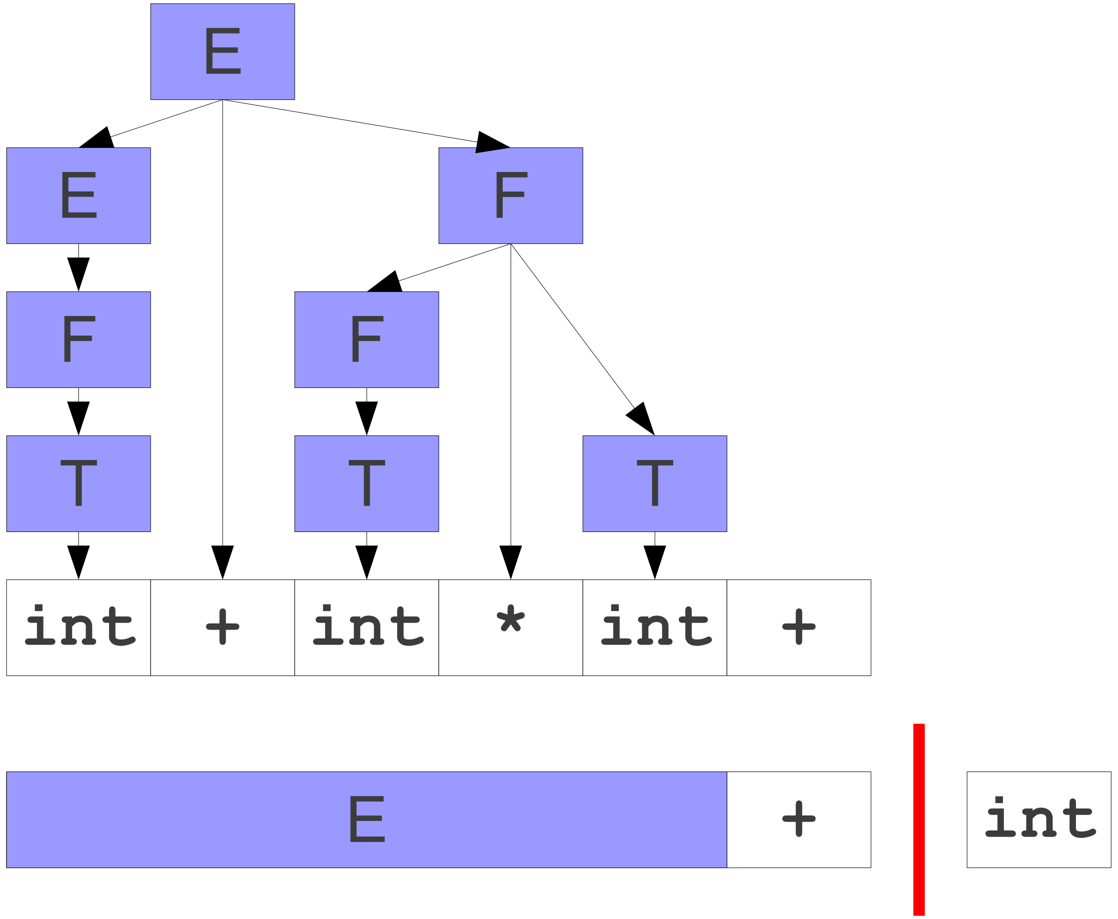
A Sample Shift/Reduce Parse

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



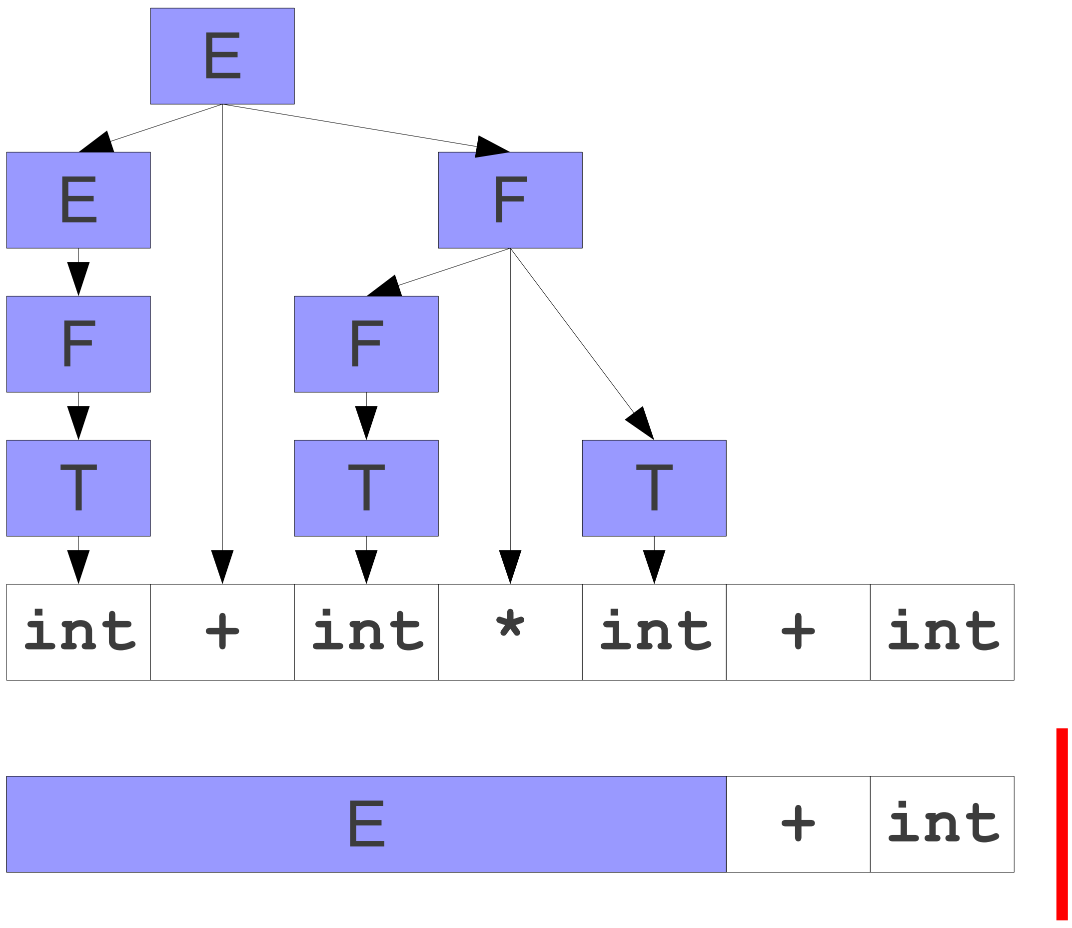
A Sample Shift/Reduce Parse

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



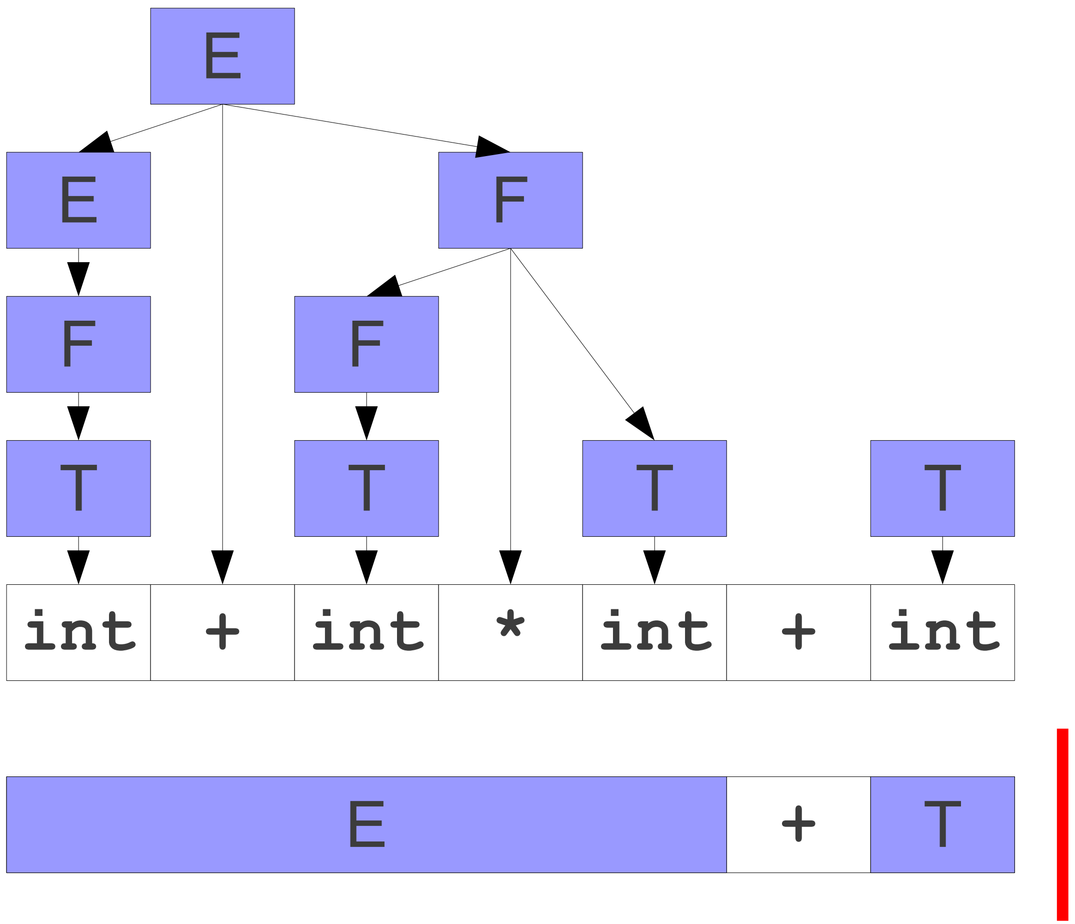
A Sample Shift/Reduce Parse

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



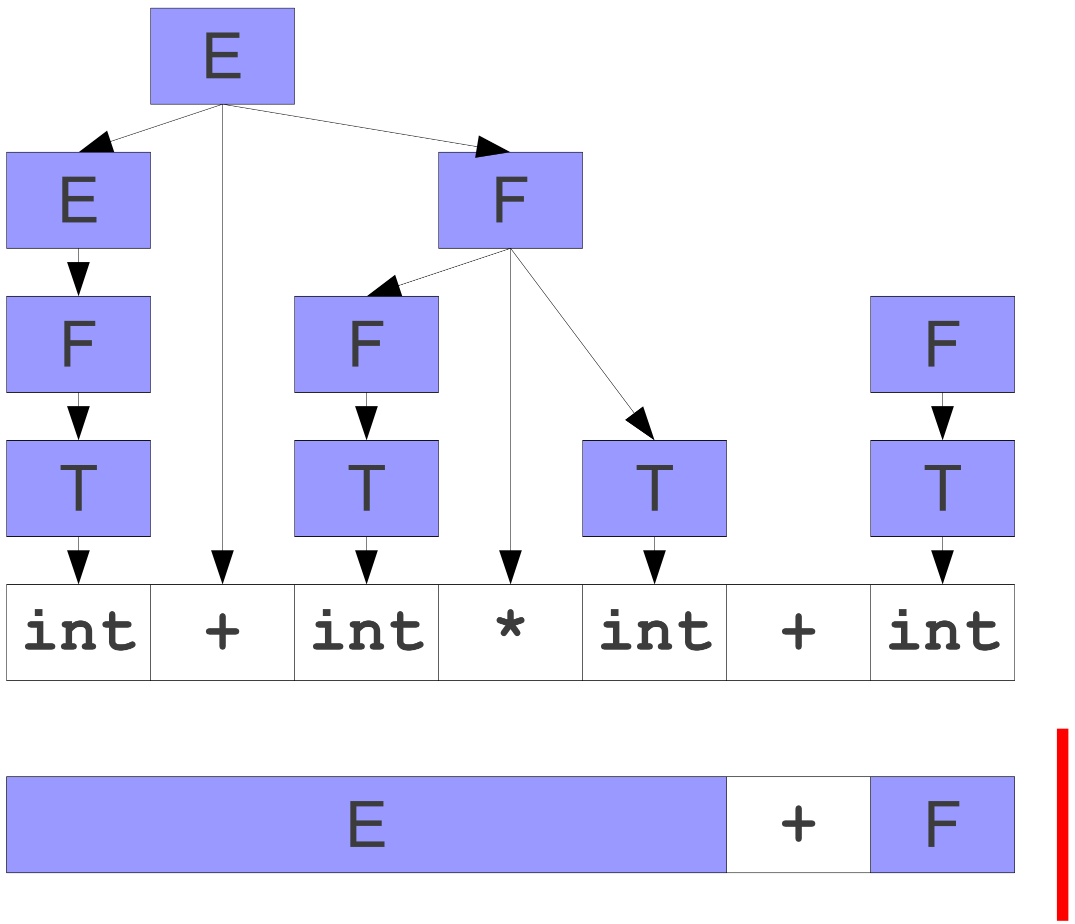
A Sample Shift/Reduce Parse

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



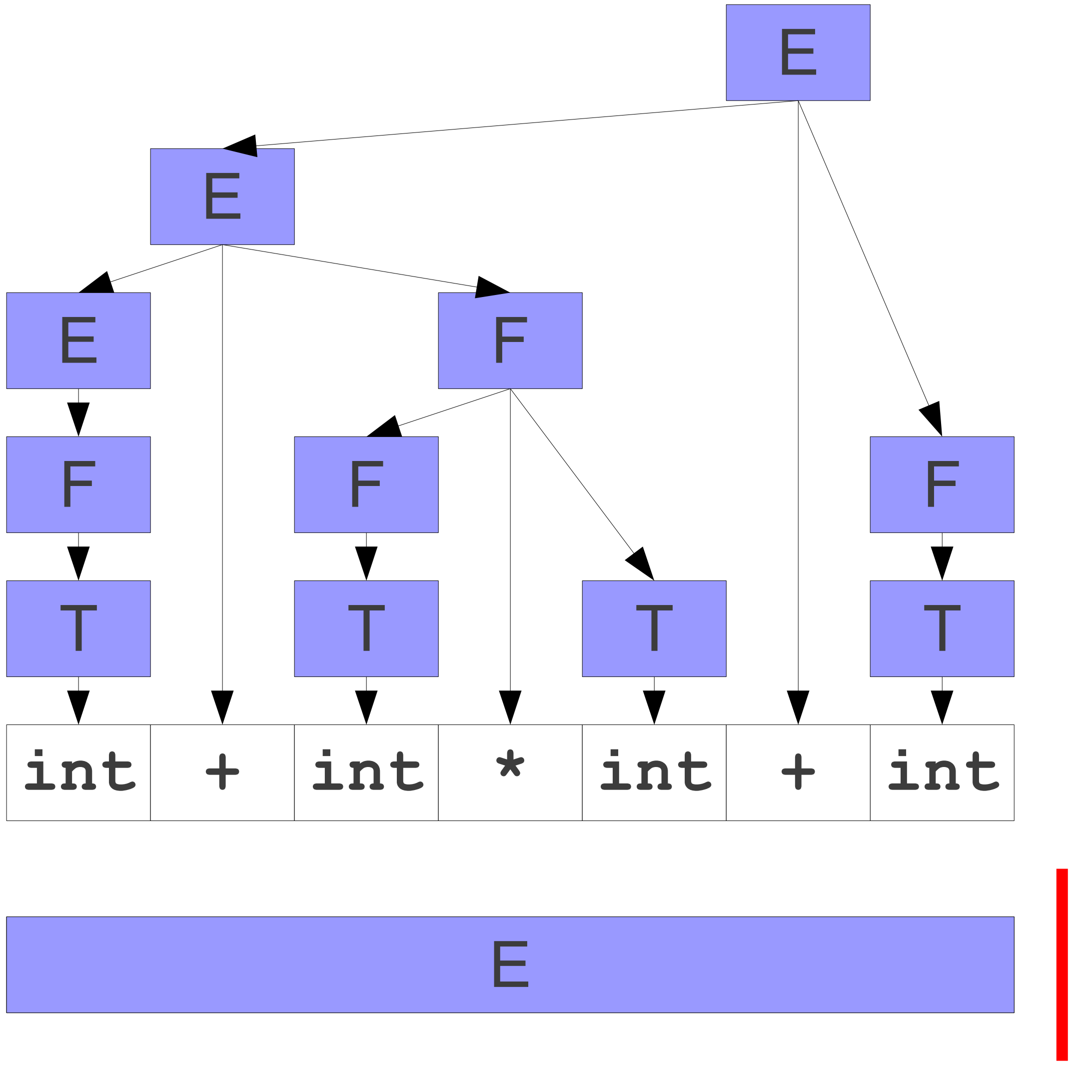
A Sample Shift/Reduce Parse

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



A Sample Shift/Reduce Parse

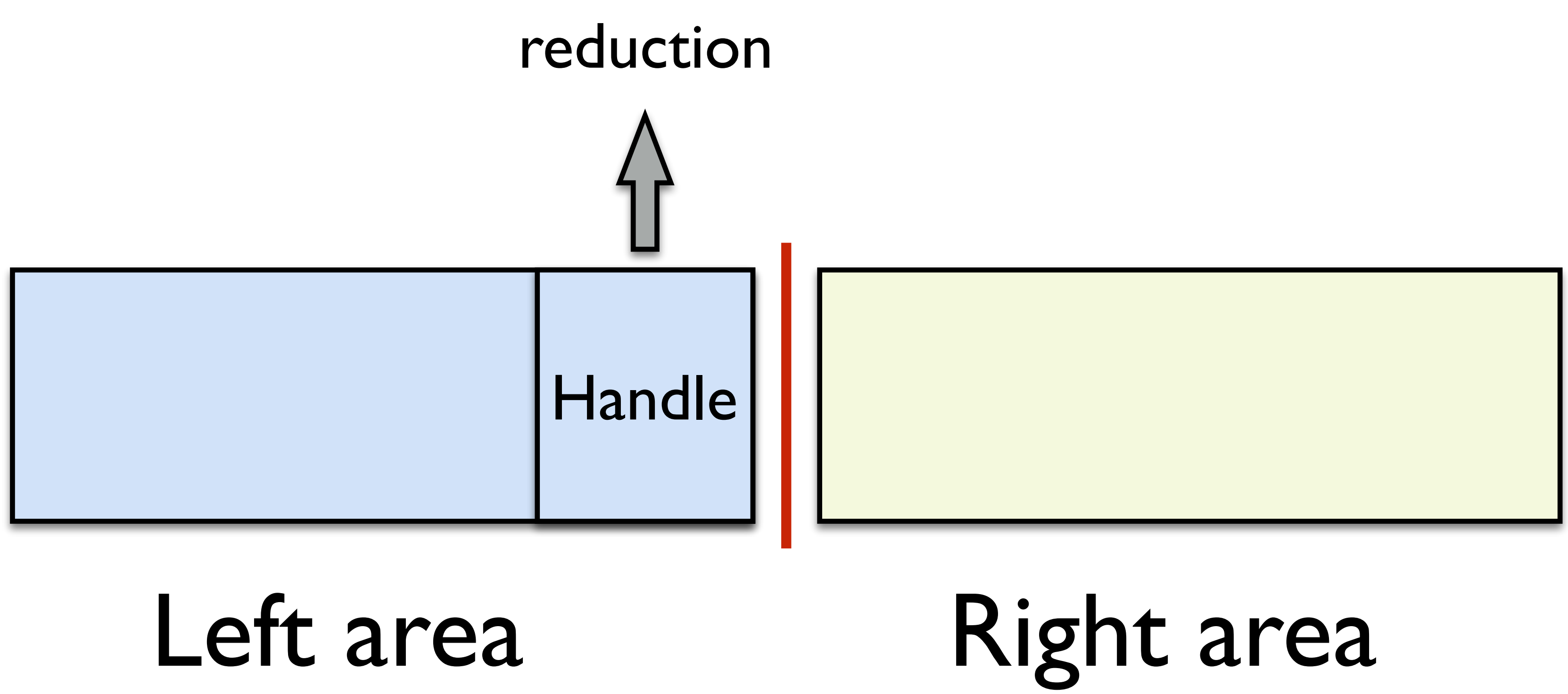
E → **F**
E → **E** **+** **F**
F → **F** ***** **T**
F → **T**
T → **int**
T → **(E)**



Simplifying our Terminology

- All activity in a shift/reduce parser is at the far right end of the left area.
- Idea: Represent 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.

- What’s the pattern of the “left area” in a shift/reduce parser?
- How to recognize it?



Finding Handles

- Where do we look for handles?
  - **At the top of the stack.**
- How do we search for handles?
  - What algorithm do we use to try to discover a handle?
- How do we recognize handles?
  - Once we've found a possible handle, how do we confirm that it's correct?

Question Two:

How do we search for handles?

Searching for Handles

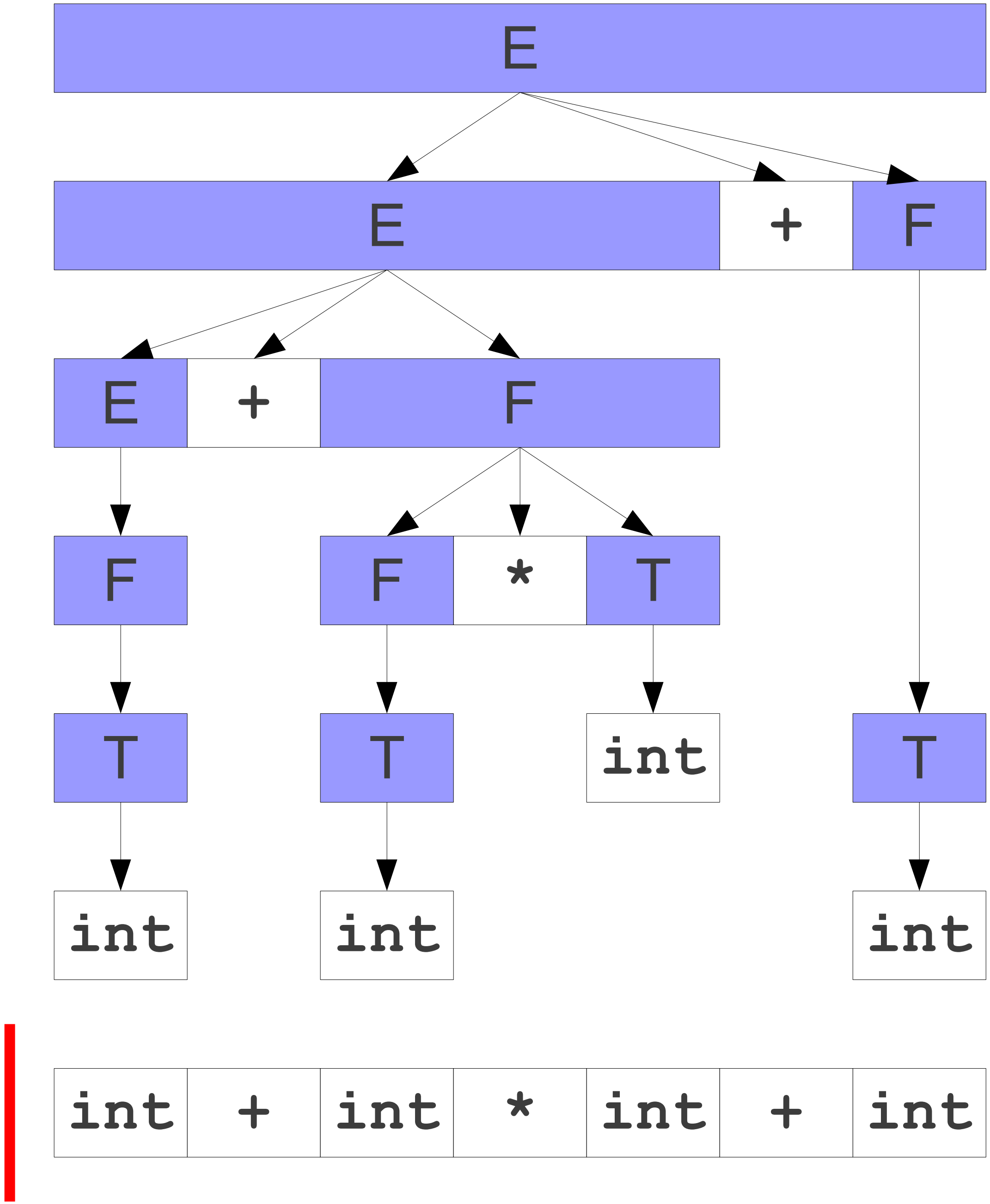
- When using a shift/reduce parser, we must decide whether to shift or reduce at each point.
- We only want to reduce when we know we have a handle.
- **Question:** How can we tell that we might be looking at a handle?

Exploring the Left Side

- The handle will always appear at the end of string in the left side of the parser.
- Can *any* string appear on the left side of the parser, or are there restrictions on what sorts of strings can appear there?
- If we can find a pattern to the strings that can appear on the left side, we might be able to exploit it to detect handles.

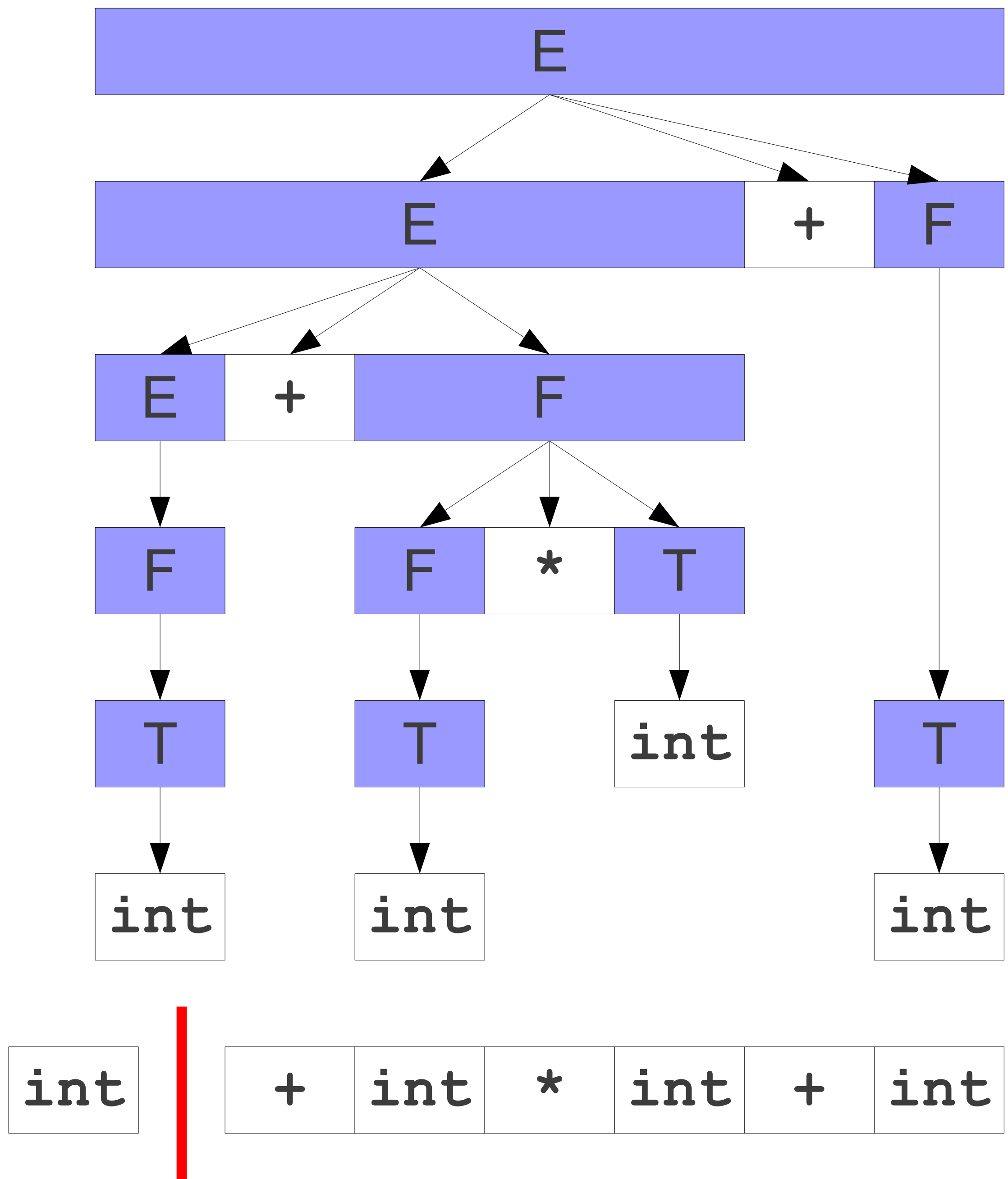
Another Look at Handles

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



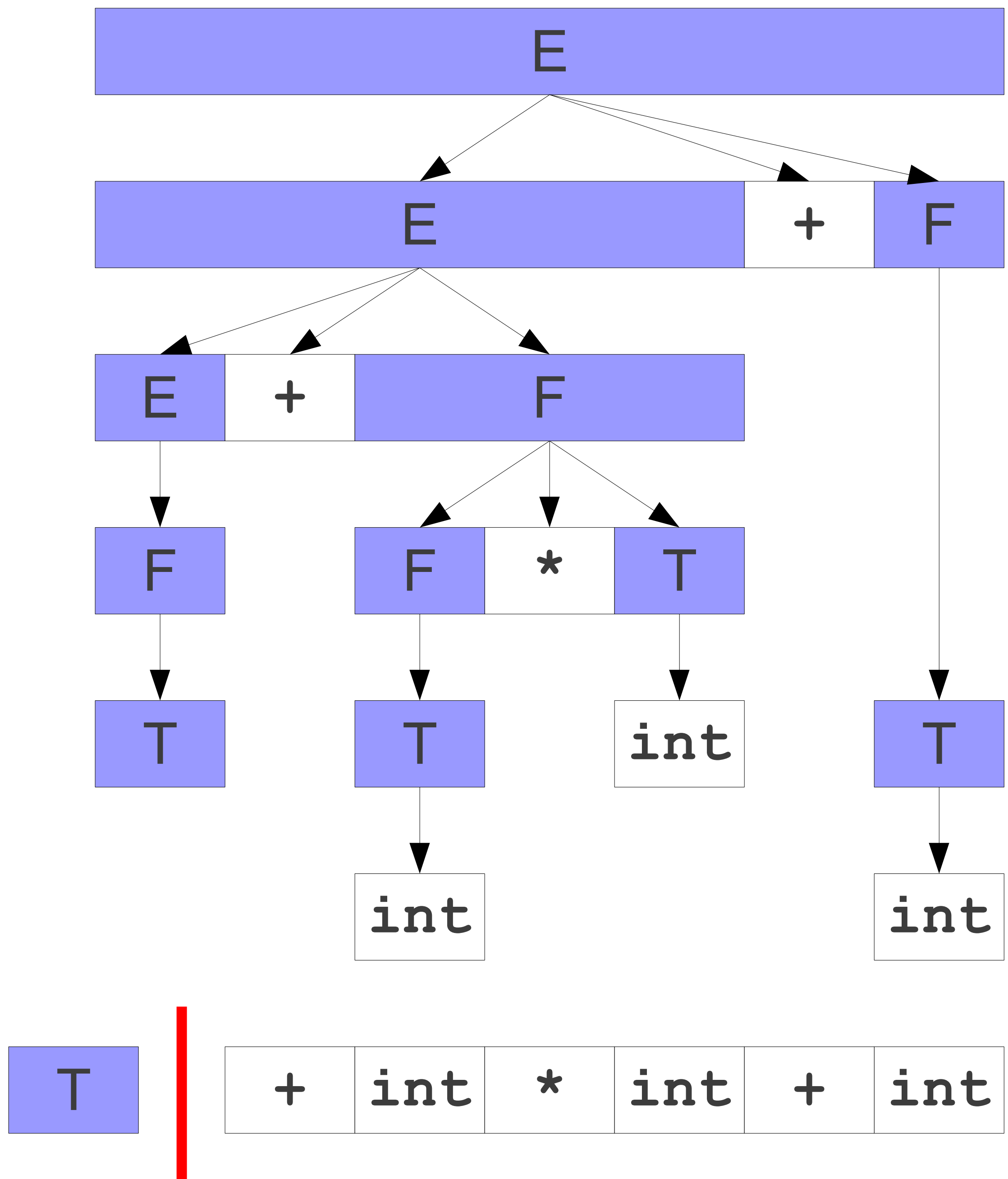
Another Look at Handles

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



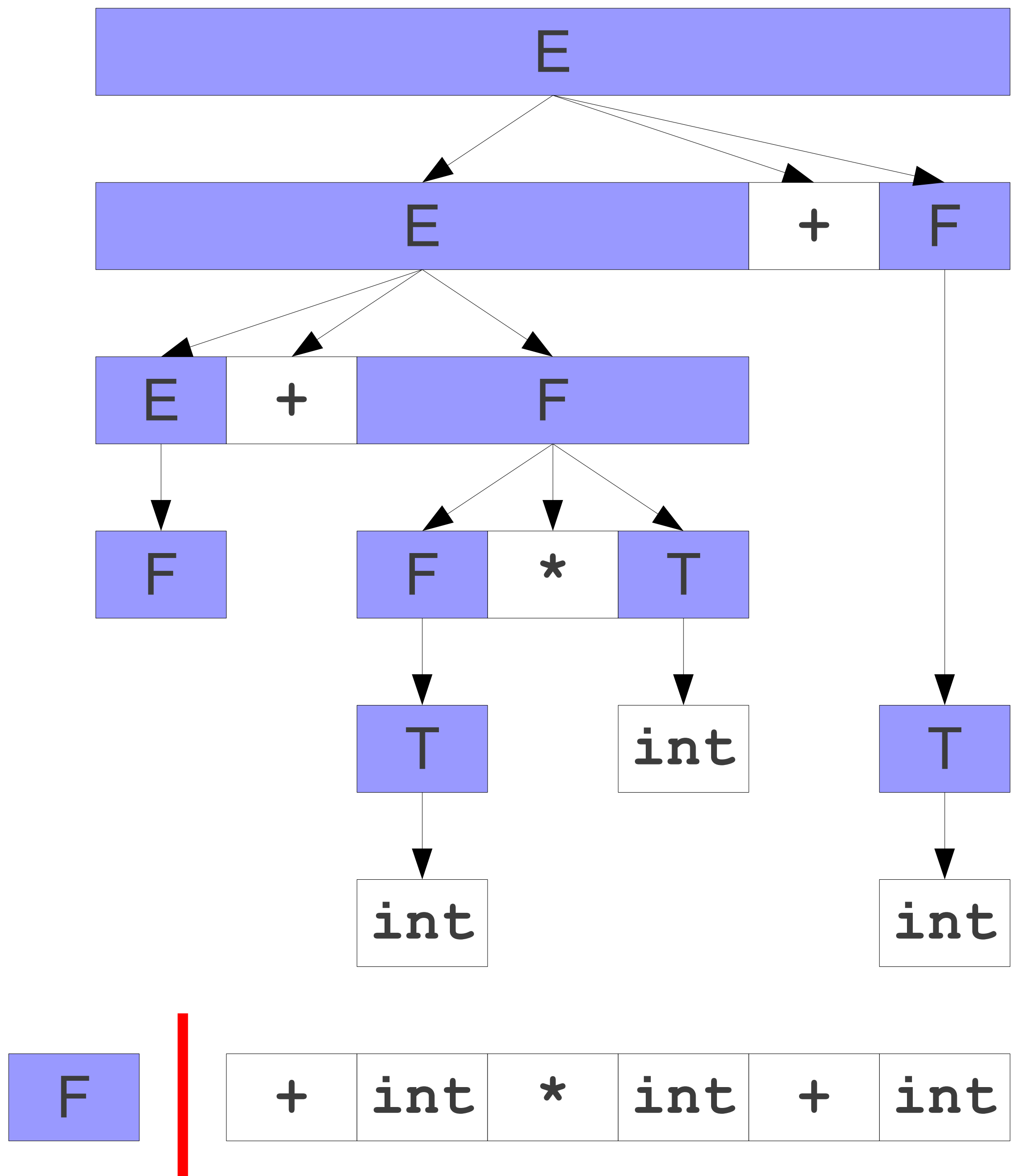
Another Look at Handles

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



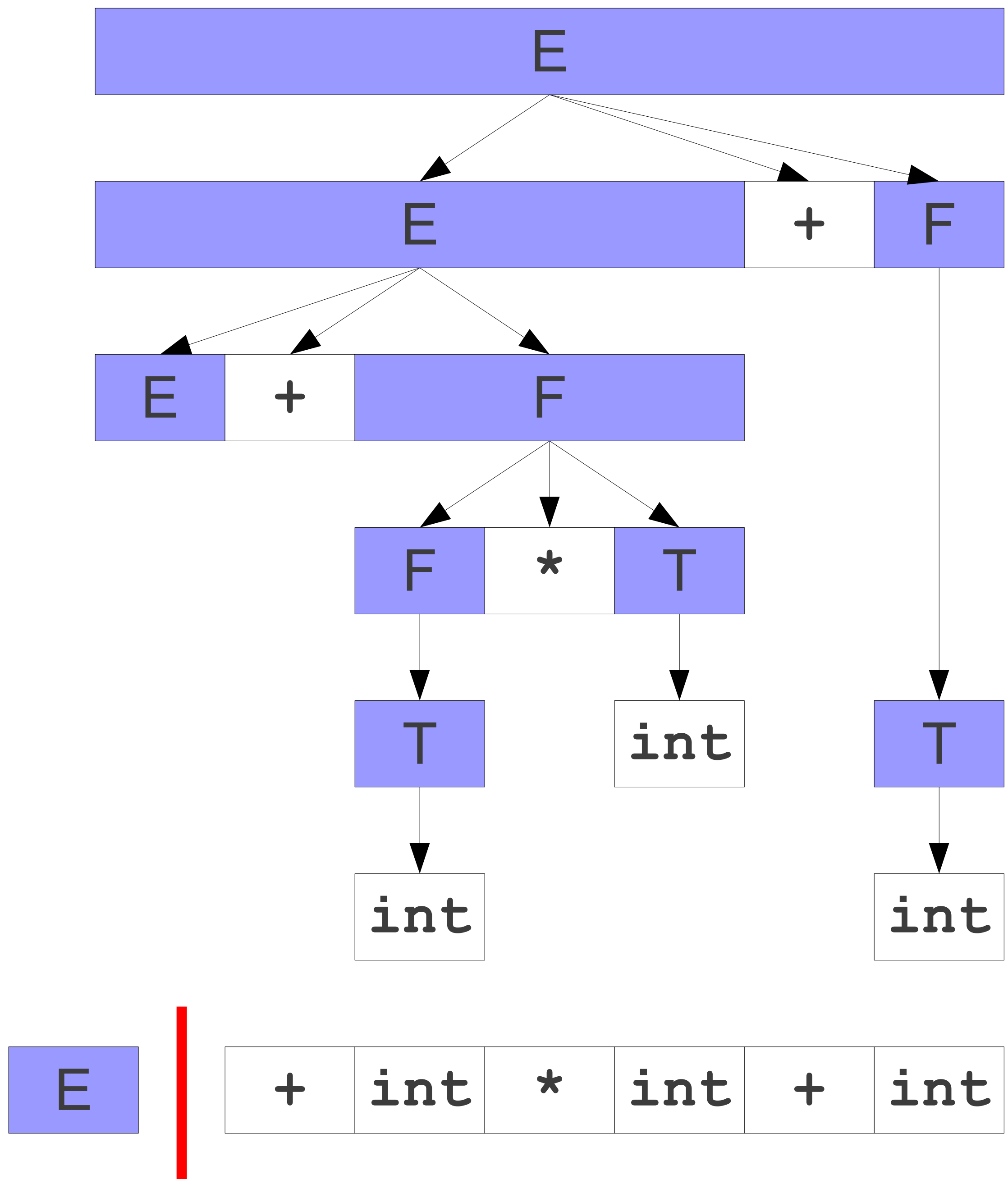
Another Look at Handles

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



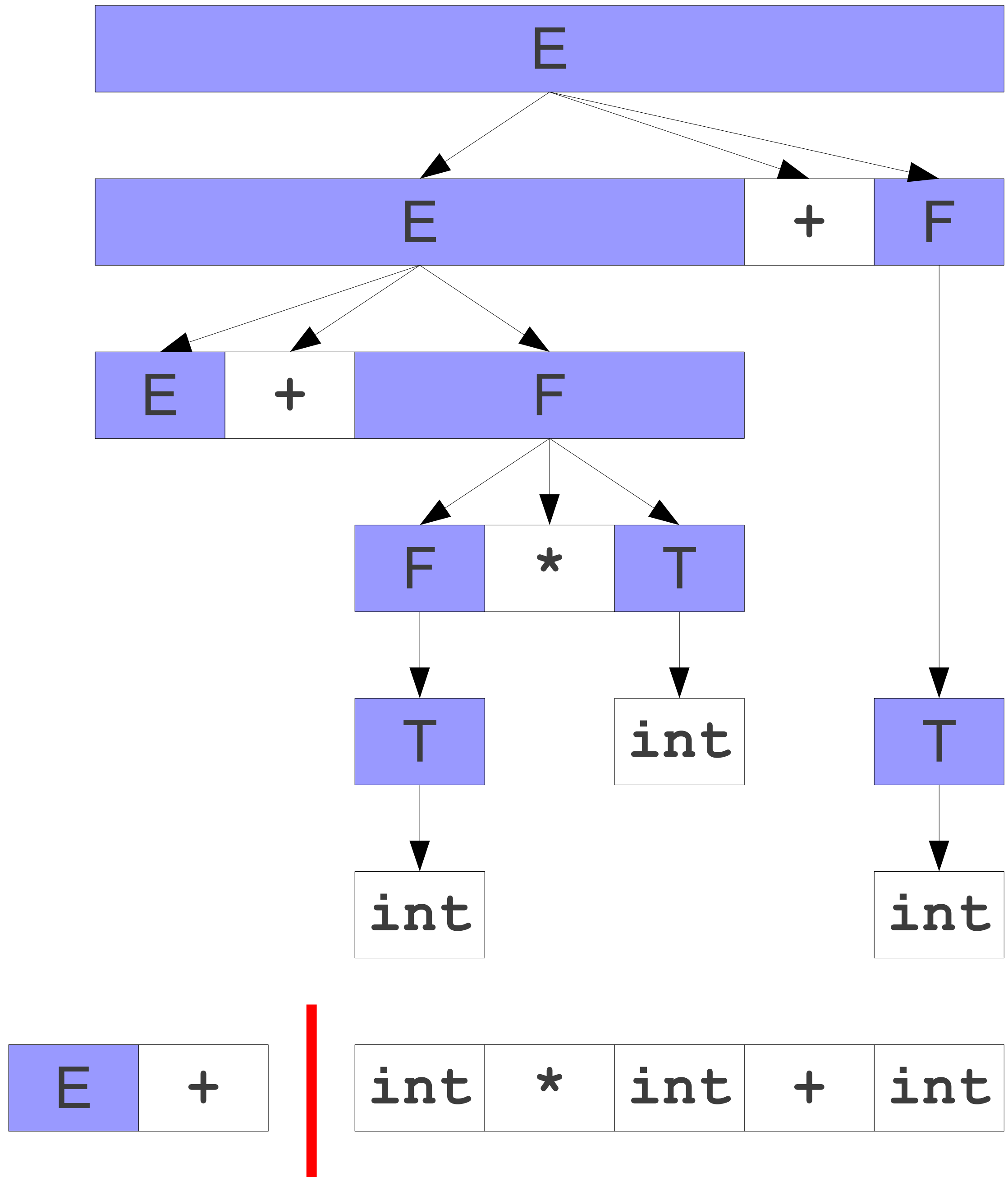
Another Look at Handles

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



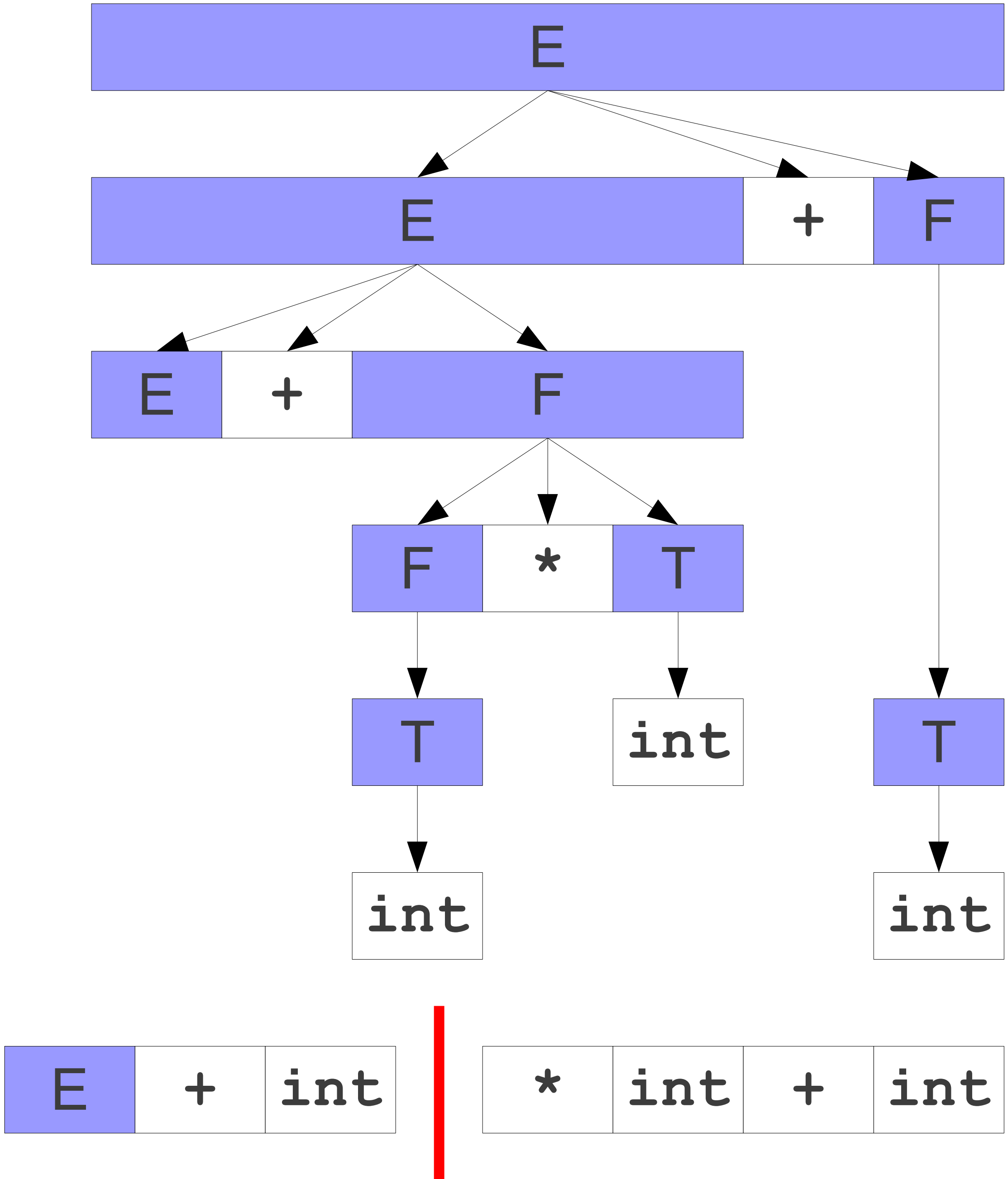
Another Look at Handles

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



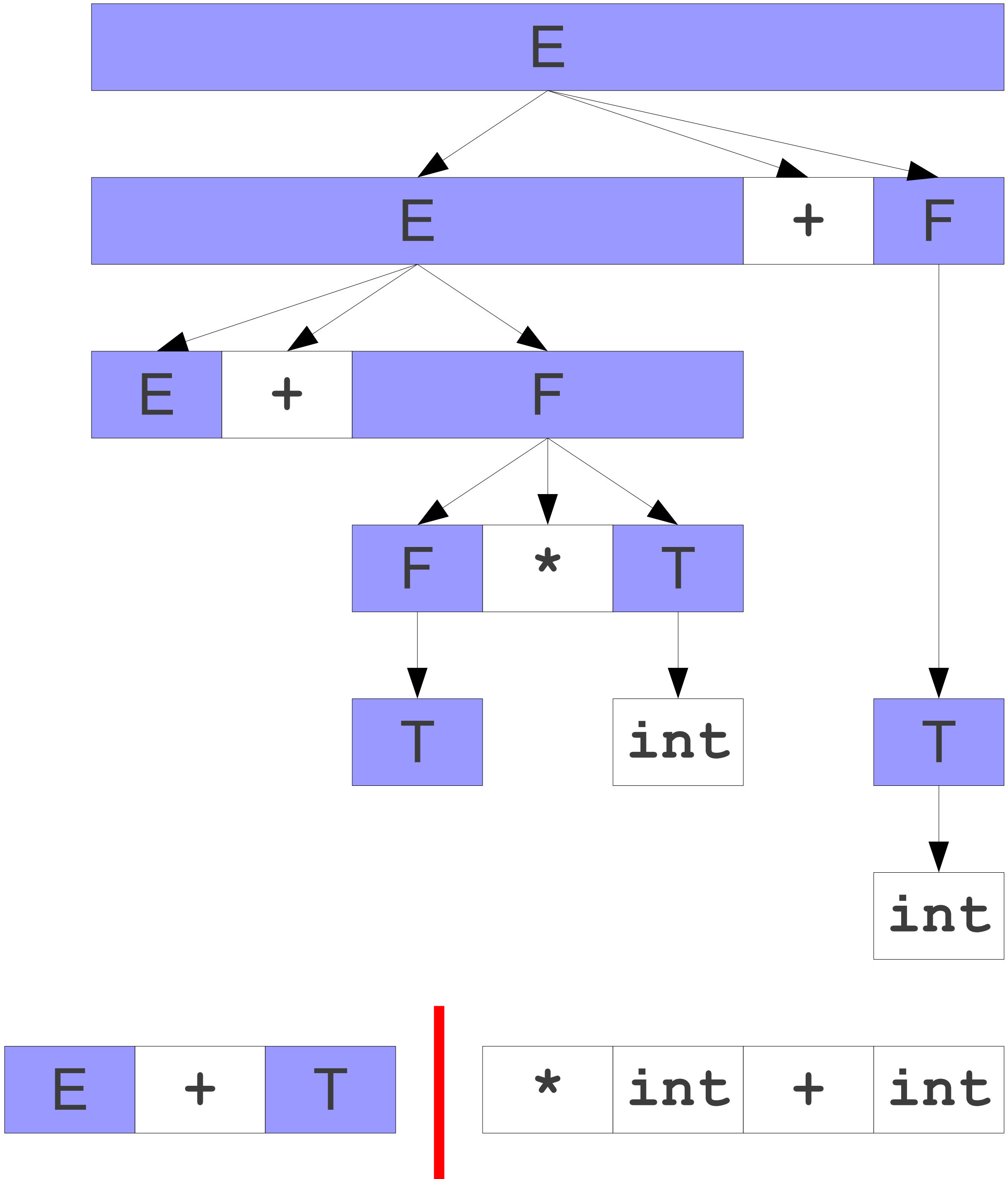
Another Look at Handles

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



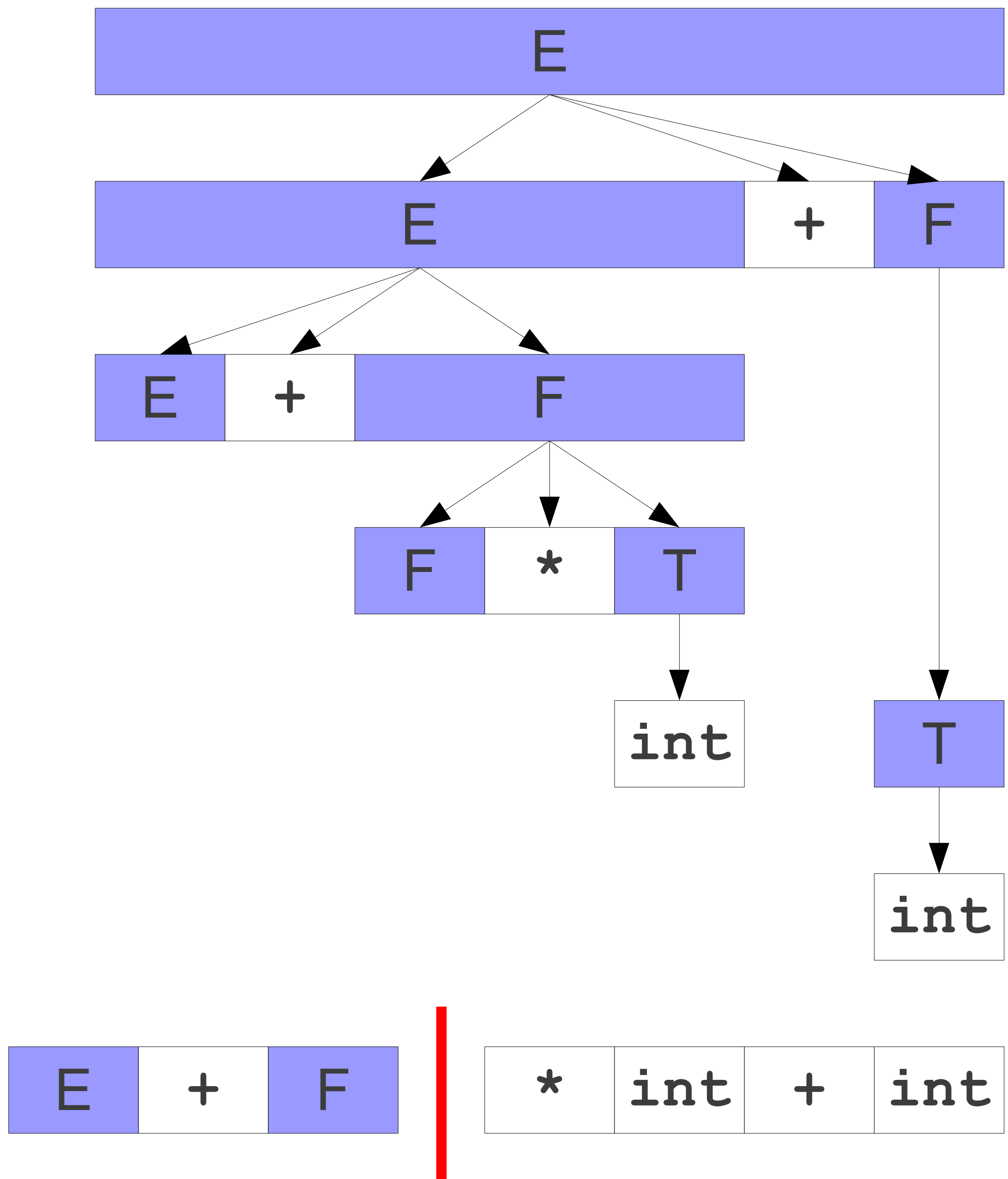
Another Look at Handles

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



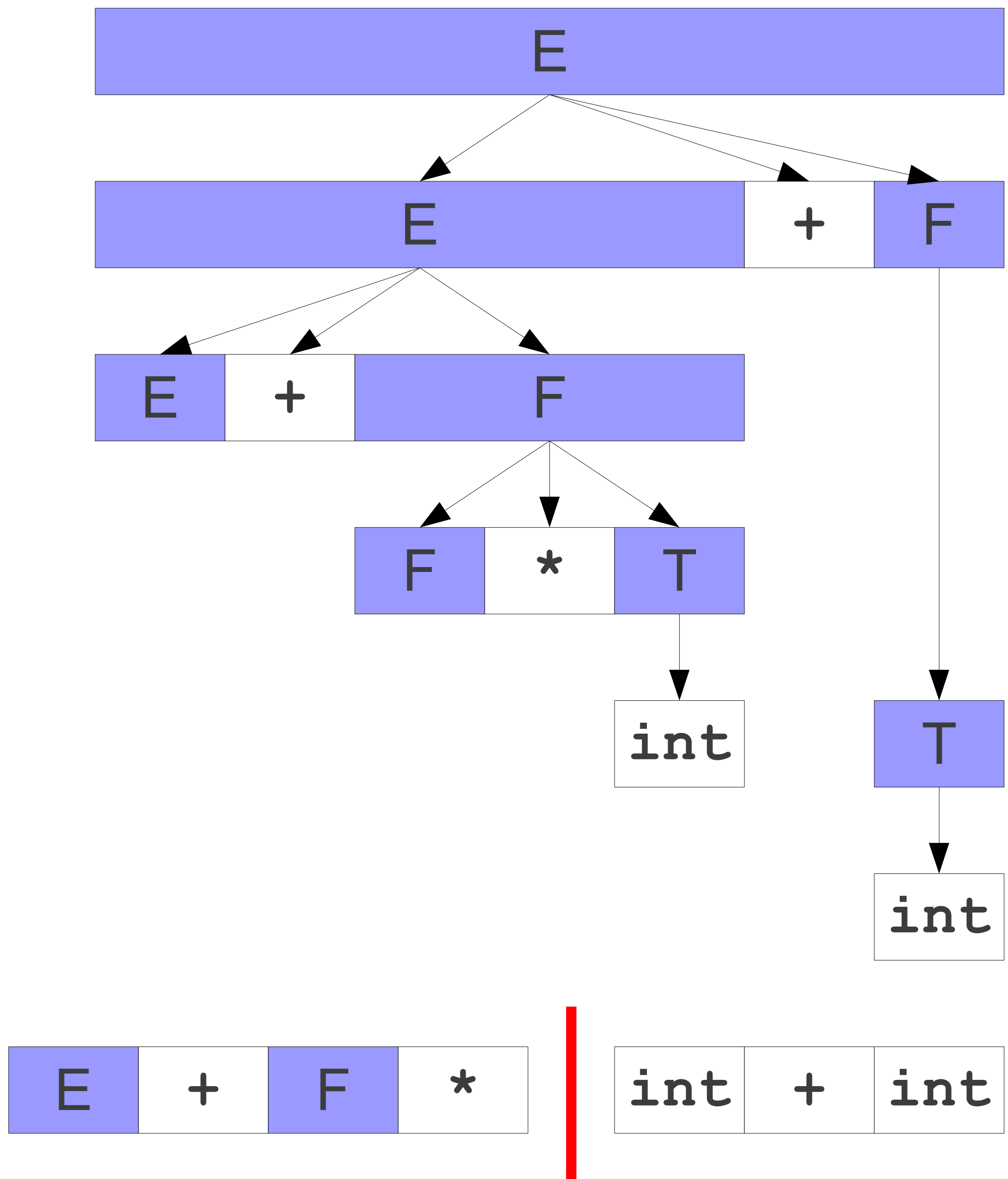
Another Look at Handles

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



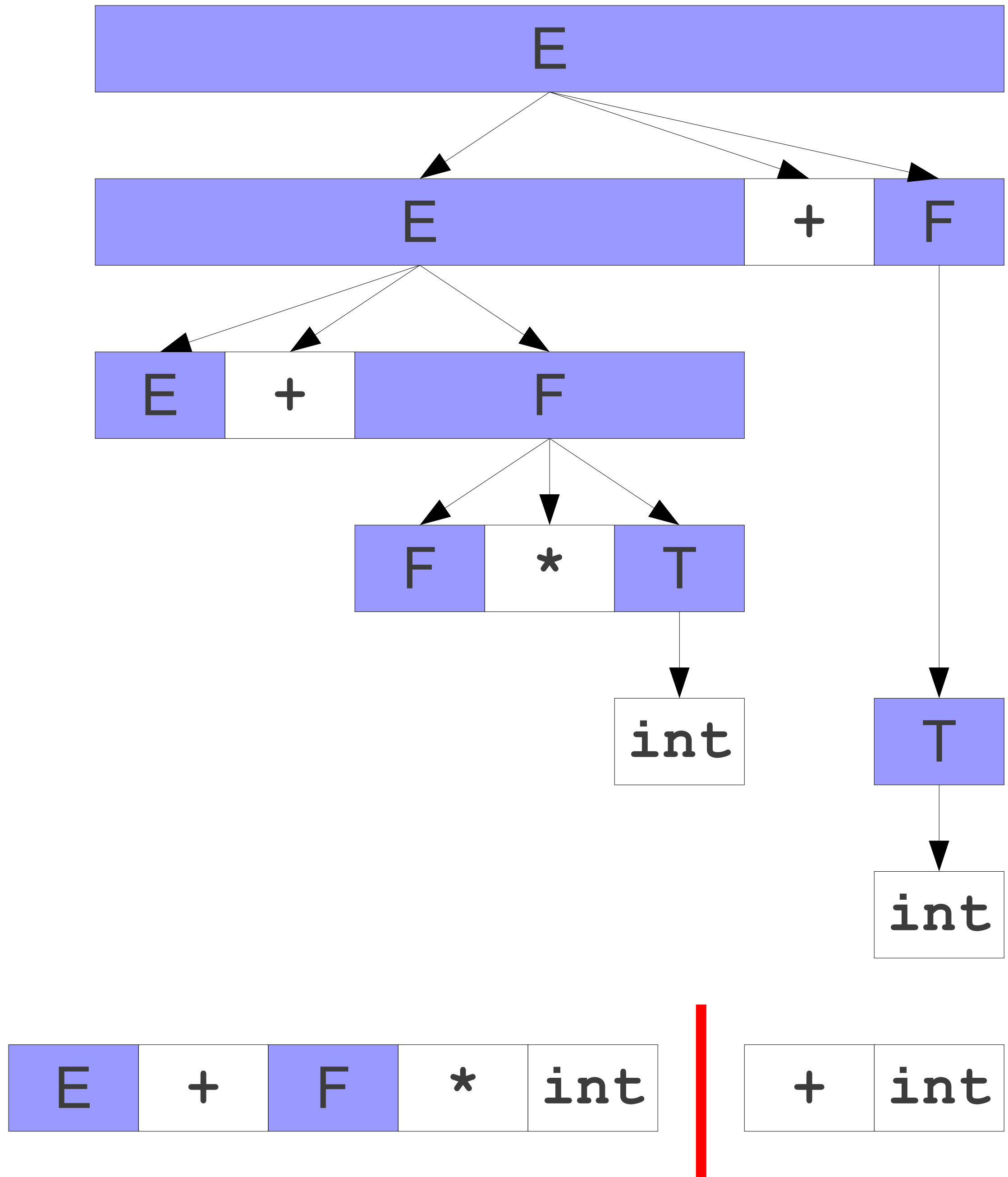
Another Look at Handles

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

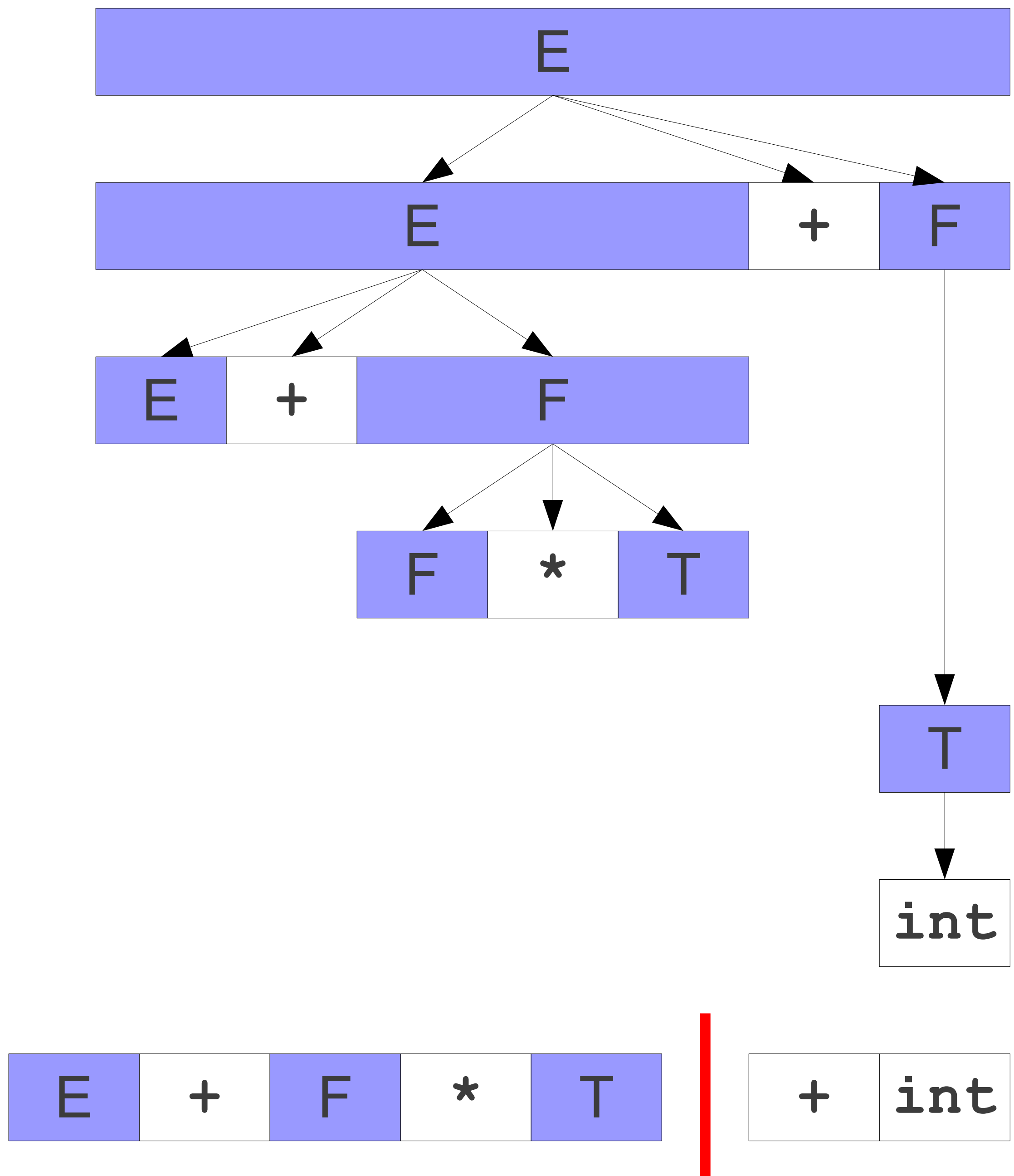


Another Look at Handles

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



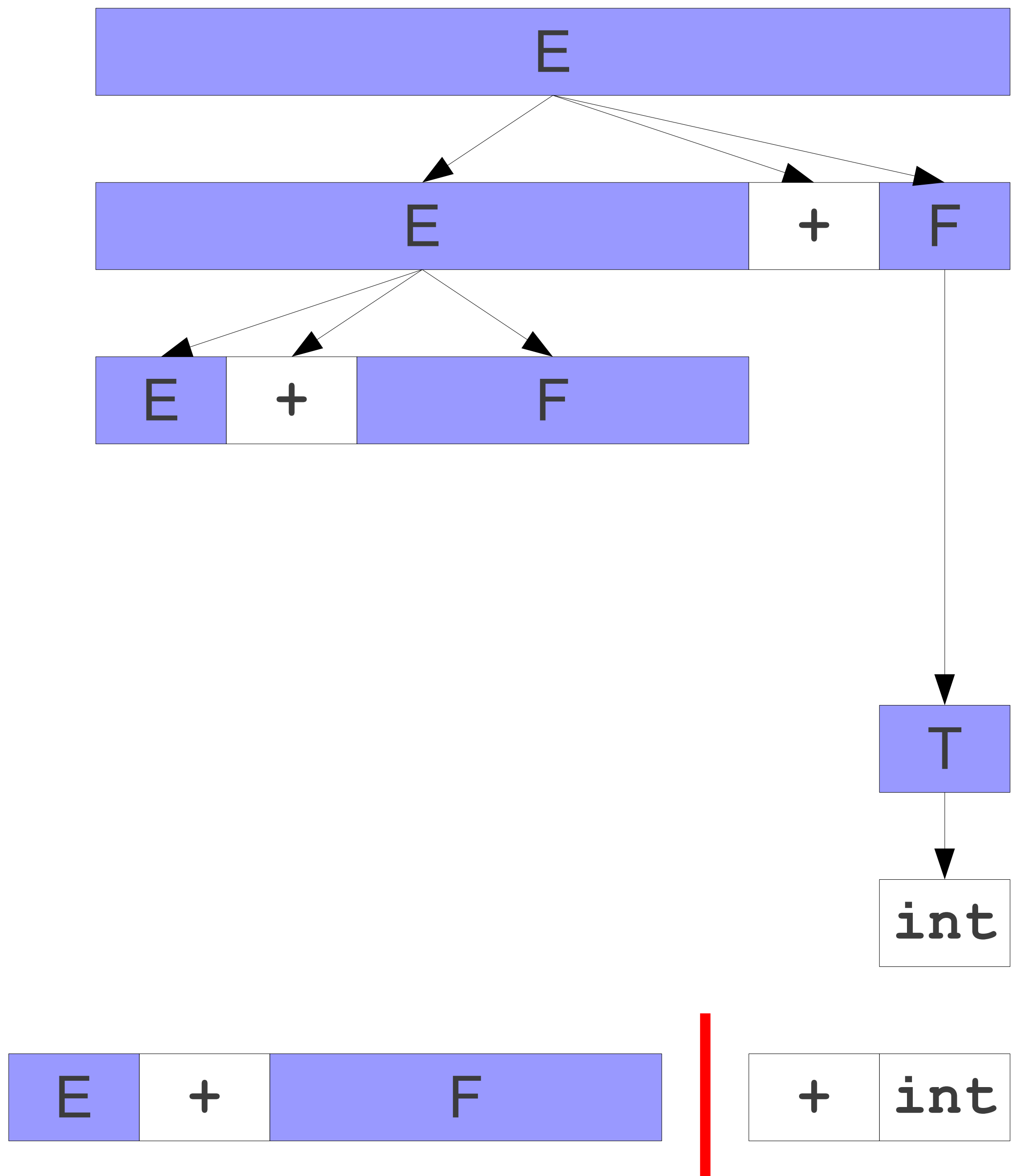
Another Look at Handles



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

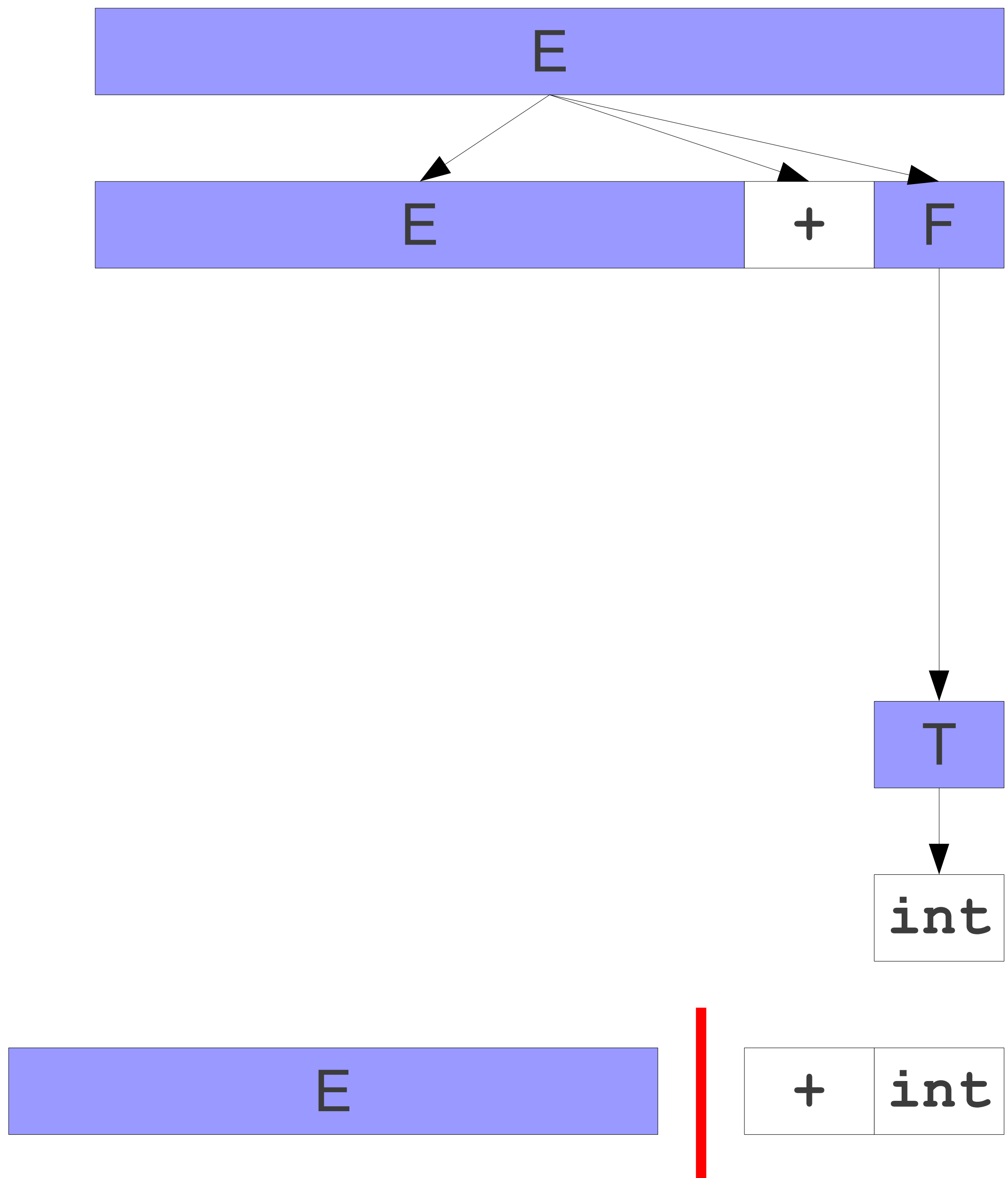
Another Look at Handles

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



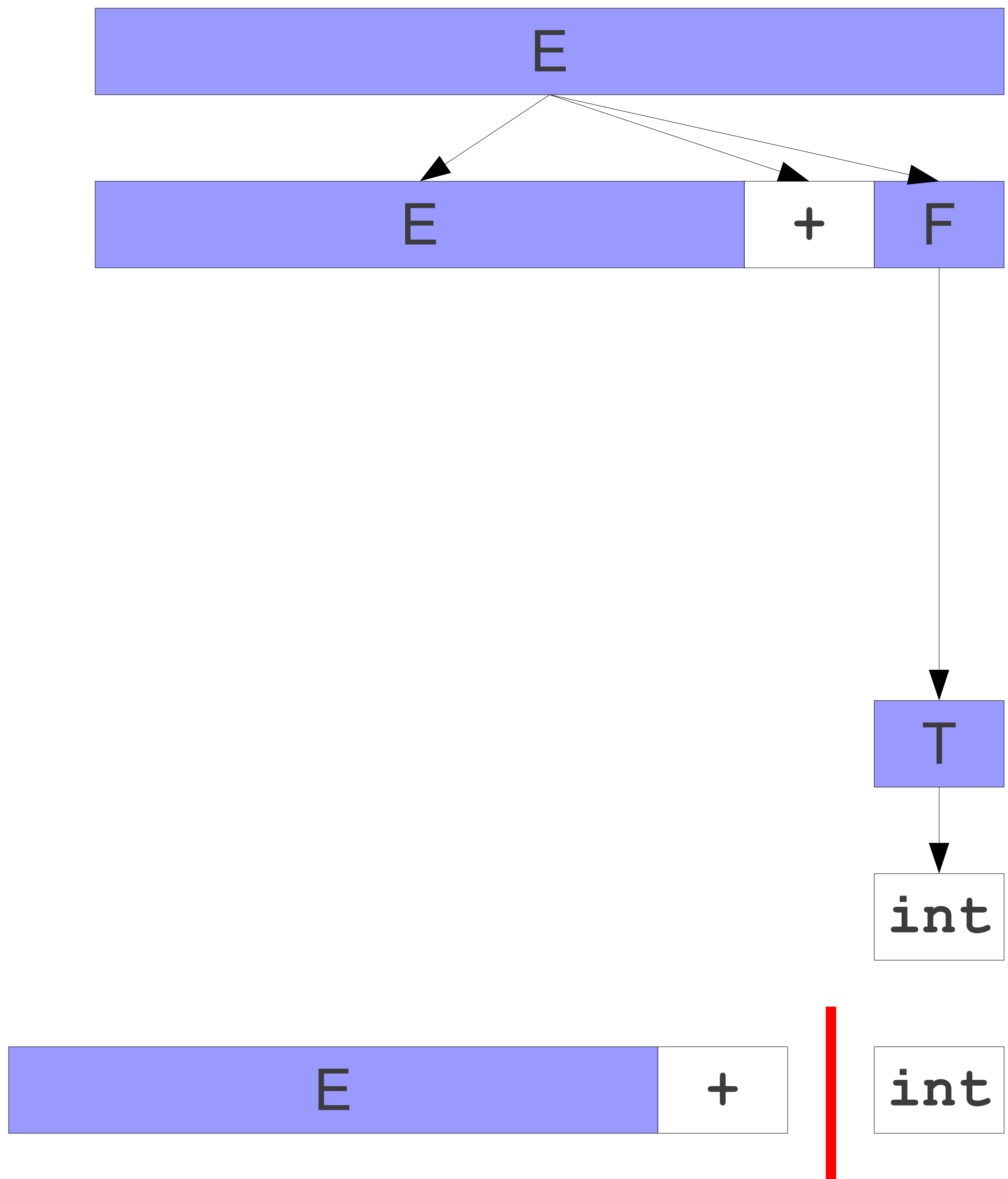
Another Look at Handles

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



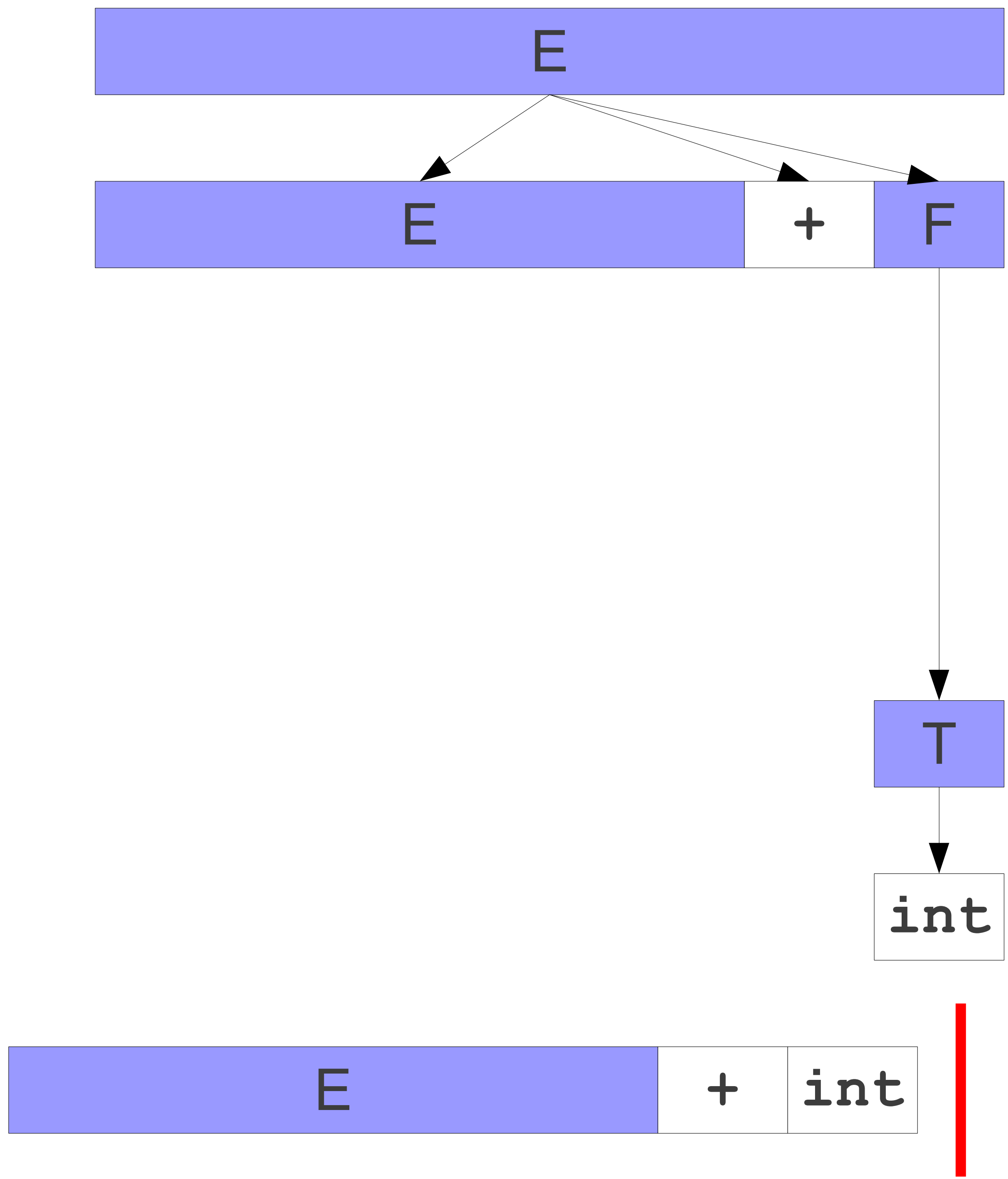
Another Look at Handles

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

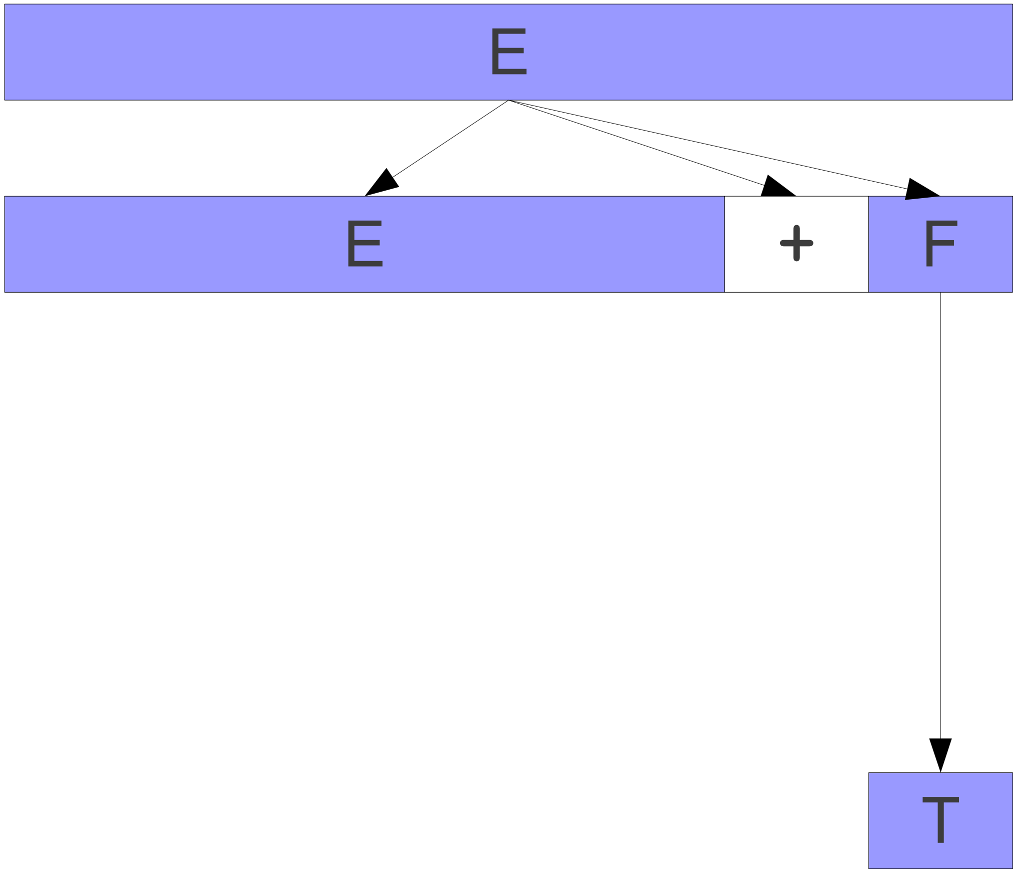


Another Look at Handles

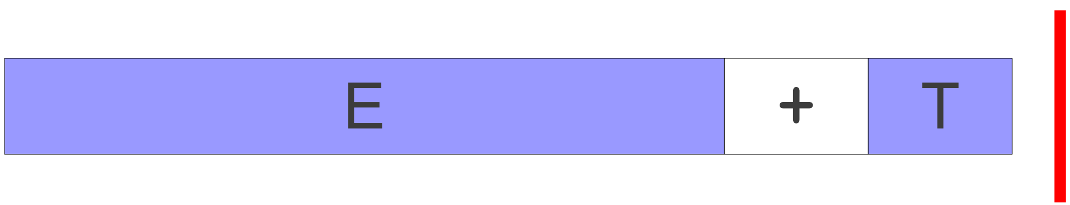
$E \rightarrow F$
$E \rightarrow E + F$
$F \rightarrow F * T$
$F \rightarrow T$
$T \rightarrow \texttt{int}$
$T \rightarrow (E)$



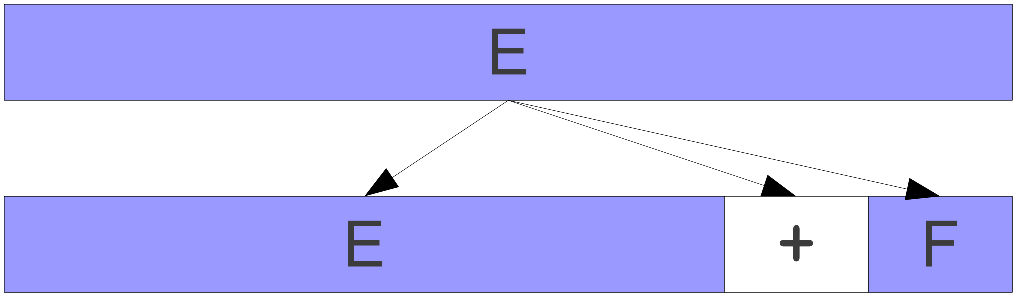
Another Look at Handles



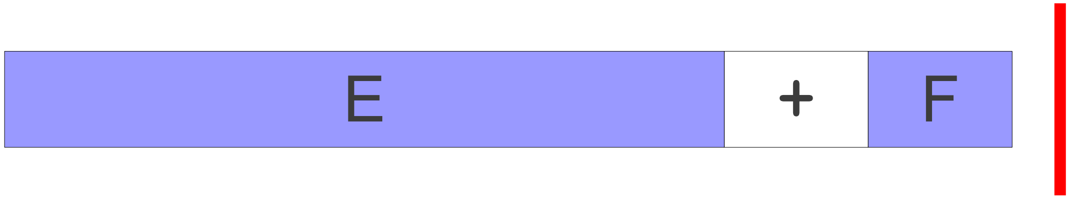
$\mathbf{E} \rightarrow \mathbf{F}$
$\mathbf{E} \rightarrow \mathbf{E} + \mathbf{F}$
$\mathbf{F} \rightarrow \mathbf{F} * \mathbf{T}$
$\mathbf{F} \rightarrow \mathbf{T}$
$\mathbf{T} \rightarrow$ `int`
$\mathbf{T} \rightarrow$ `(`$\mathbf{E}$`)`



Another Look at Handles



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



Another Look at Handles

E

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

E

Tracking Our Position

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

int	+	int	*	int	+	int

Tracking Our Position

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

`int`	`+`	`int`	`*`	`int`	`+`	`int`

Tracking Our Position

$S \rightarrow \cdot E$

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

int	+	int	*	int	+	int

Tracking Our Position

$S \rightarrow \cdot E$
$E \rightarrow \cdot E + F$

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

`int`	`+`	`int`	`*`	`int`	`+`	`int`

Tracking Our Position

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

S → · E
E → · E + F
E → · E + F

`int`	`+`	`int`	`*`	`int`	`+`	`int`

Tracking Our Position

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

$S \to \cdot E$
$E \to \cdot E + F$
$E \to \cdot E + F$
$E \to \cdot F$

`int`	`+`	`int`	`*`	`int`	`+`	`int`

Tracking Our Position

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

S → · E
E → · E + F
E → · E + F
E → · F
F → · T

`int`	`+`	`int`	`*`	`int`	`+`	`int`

Tracking Our Position

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

S → · E
E → · E + F
E → · E + F
E → · F
F → · T
T → · **int**

int	+	int	*	int	+	int

Tracking Our Position

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

$S \rightarrow \cdot E$
$E \rightarrow \cdot E + F$
$E \rightarrow \cdot E + F$
$E \rightarrow \cdot F$
$F \rightarrow \cdot T$
$T \rightarrow$ `int` $\cdot$

`int`	|	`+`	`int`	`*`	`int`	`+`	`int`

Tracking Our Position

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

$S \rightarrow \cdot E$
$E \rightarrow \cdot E + F$
$E \rightarrow \cdot E + F$
$E \rightarrow \cdot F$
$F \rightarrow \cdot T$

T	+	int	*	int	+	int

Tracking Our Position

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

$S \to \cdot E$
$E \to \cdot E + F$
$E \to \cdot E + F$
$E \to \cdot F$
$F \to T \cdot$

T	+	int	*	int	+	int

Tracking Our Position

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

$S \to \cdot E$
$E \to \cdot E + F$
$E \to \cdot E + F$
$E \to \cdot F$

F	|	+	int	*	int	+	int

Tracking Our Position

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

S → · E
E → · E + F
E → · E + F
E → F ·

F	|	`+`	`int`	`*`	`int`	`+`	`int`

Tracking Our Position

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

$S \to \cdot E$
$E \to \cdot E + F$
$E \to \cdot E + F$

E	|	+	int	*	int	+	int

Tracking Our Position

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

$S \to \cdot E$
$E \to \cdot E + F$
$E \to E \cdot + F$

E	|	+	int	*	int	+	int

Tracking Our Position

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

$S \rightarrow \cdot E$
$E \rightarrow \cdot E + F$
$E \rightarrow E + \cdot F$

E	+	|	int	*	int	+	int

Tracking Our Position

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

S → · E
E → · E + F
E → E + · F
F → · F * T

E	+	|	int	*	int	+	int

Tracking Our Position

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

S → · E
E → · E + F
E → E + · F
F → · F * T
F → · T

E	`+`	|	`int`	`*`	`int`	`+`	`int`

Tracking Our Position

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

S → · E
E → · E + F
E → E + · F
F → · F * T
F → · T
T → · `int`

E	+	|	`int`	*	`int`	+	`int`

Tracking Our Position

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

S → · E
E → · E + F
E → E + · F
F → · F * T
F → · T
T → int ·

E	+	int	|	*	int	+	int

Tracking Our Position

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

$S \rightarrow \cdot E$
$E \rightarrow \cdot E + F$
$E \rightarrow E + \cdot F$
$F \rightarrow \cdot F * T$
$F \rightarrow \cdot T$

E	+	T	|	*	int	+	int

Tracking Our Position

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

S → · E
E → · E + F
E → E + · F
F → · F * T
F → T ·

E	+	T	|	*	int	+	int

Tracking Our Position

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

S → · E
E → · E + F
E → E + · F
F → · F * T

E	+	F	|	*	int	+	int

Tracking Our Position

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

S → · E
E → · E + F
E → E + · F
F → F · * T

E	+	F	|	*	int	+	int

Tracking Our Position

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

S → · E
E → · E + F
E → E + · F
F → F * · T

E	+	F	*	|	int	+	int

Tracking Our Position

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

S → · E
E → · E + F
E → E + · F
F → F `*` · T
T → · `int`

E	`+`	F	`*`	|	`int`	`+`	`int`

Tracking Our Position

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

S → · E
E → · E + F
E → E + · F
F → F * · T
T → int ·

E	+	F	*	int	|	+	int

Tracking Our Position

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

S → · E
E → · E + F
E → E + · F
F → F * · T

E	+	F	*	T	|	+	int

Tracking Our Position

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

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

E	+	F	*	T	|	+	int

Tracking Our Position

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

S → · E
E → · E + F
E → E + · F

E	+	F	|	+	int

Tracking Our Position

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

$S \to \cdot E$
$E \to \cdot E + F$
$E \to E + F \cdot$

E	+	F	|	+	int

Tracking Our Position

$S \to \cdot E$
$E \to \cdot E + F$

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

E	`+`	`int`

Tracking Our Position

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

$S \rightarrow \cdot E$
$E \rightarrow E \cdot + F$

E	|	+	int

Tracking Our Position

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

$S \to \cdot E$
$E \to E + \cdot F$

E + | int

Tracking Our Position

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

S → · E
E → E + · F
F → · T

E	+	|	int

Tracking Our Position

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

$S \rightarrow \cdot E$
$E \rightarrow E + \cdot F$
$F \rightarrow \cdot T$
$T \rightarrow \cdot \texttt{int}$

E + | int

Tracking Our Position

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

$S \rightarrow \cdot\, E$
$E \rightarrow E + \cdot\, F$
$F \rightarrow \cdot\, T$
$T \rightarrow$ **int** $\cdot$

E	+	int

Tracking Our Position

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

S → · E
E → E + · F
F → · T

E	+	T

Tracking Our Position

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

$S \rightarrow \cdot E$
$E \rightarrow E + \cdot F$
$F \rightarrow T \cdot$

E	+	T

Tracking Our Position

S → · E
E → E + · F

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

E	+	F

Tracking Our Position

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

$S \rightarrow \cdot E$
$E \rightarrow E + F \cdot$

E	+	F

Tracking Our Position

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

$S \to \cdot E$

E

Tracking Our Position

$S \rightarrow E \cdot$

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

E

Generating Left-Hand Sides

- At any instant in time, the contents of the left side of the parser can be described using the following process:
  - Trace out, from the start symbol, the series of productions that have not yet been completed and where we are in each production.
  - For each production, in order, output all of the symbols up to the point where we change from one production to the next.

Recognizing Left-Hand Sides

- Given that we have a procedure for *generating* left-hand sides, can we build a procedure for *recognizing* those left-hand sides?
- Idea: At each point, track
  - Which production we are in, and
  - Where we are in that production.
- At each point, we can do one of two things:
  - Match the next symbol of the candidate left-hand side with the next symbol in the current production, or
  - If the next symbol of the candidate left-hand side is a nonterminal, nondeterministically guess which production to try next.

Recognizing Left-Hand Sides

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

E	+	F	*	int		+	int

Recognizing Left-Hand Sides

$S \to \cdot E$

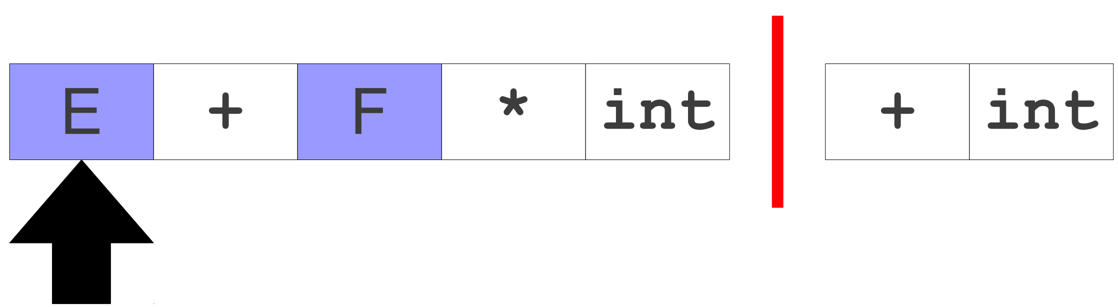
S → **E**
E → **F**
E → **E** `+` **F**
F → **F** `*` **T**
F → **T**
T → `int`
T → `(`**E**`)`

E	+	F	*	int	|	+	int

Recognizing Left-Hand Sides

$S \rightarrow \cdot\, E$

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



Recognizing Left-Hand Sides

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

$S \rightarrow \cdot\ E$
$E \rightarrow \cdot\ E + F$

E	+	F	*	int	|	+	int

Recognizing Left-Hand Sides

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

$S \rightarrow \cdot E$
$E \rightarrow \cdot E + F$
$E \rightarrow \cdot E + F$

E	+	F	*	int			+	int

Recognizing Left-Hand Sides

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

$S \to \cdot E$
$E \to \cdot E + F$
$E \to E \cdot + F$

E	+	F	*	int	|	+	int

Recognizing Left-Hand Sides

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

$S \rightarrow \cdot E$
$E \rightarrow \cdot E + F$
$E \rightarrow E + \cdot F$

E	+	F	*	int	|	+	int

Recognizing Left-Hand Sides

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

S → · E
E → · E + F
E → E + · F
F → · F * T

E	+	F	*	`int`	|	+	`int`

Recognizing Left-Hand Sides

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

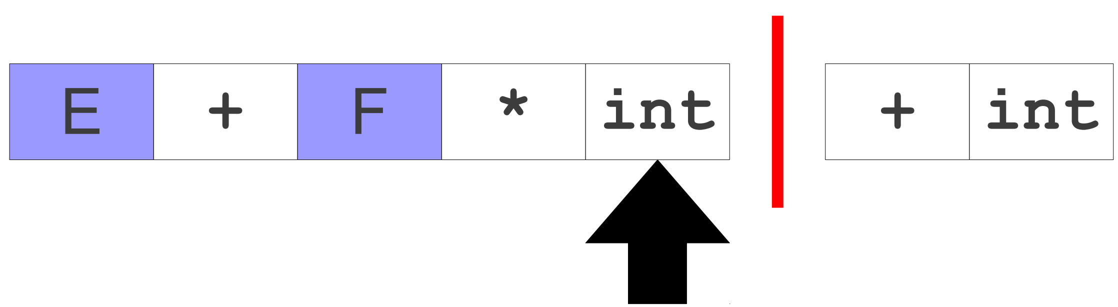
S → · E
E → · E + F
E → E + · F
F → F · * T

E	+	F	*	int	|	+	int

Recognizing Left-Hand Sides

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

S → · E
E → · E + F
E → E + · F
F → F * · T



Recognizing Left-Hand Sides

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

S → · E
E → · E + F
E → E + · F
F → F * · T
T → · `int`

E	+	F	*	int	+	int

Recognizing Left-Hand Sides

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

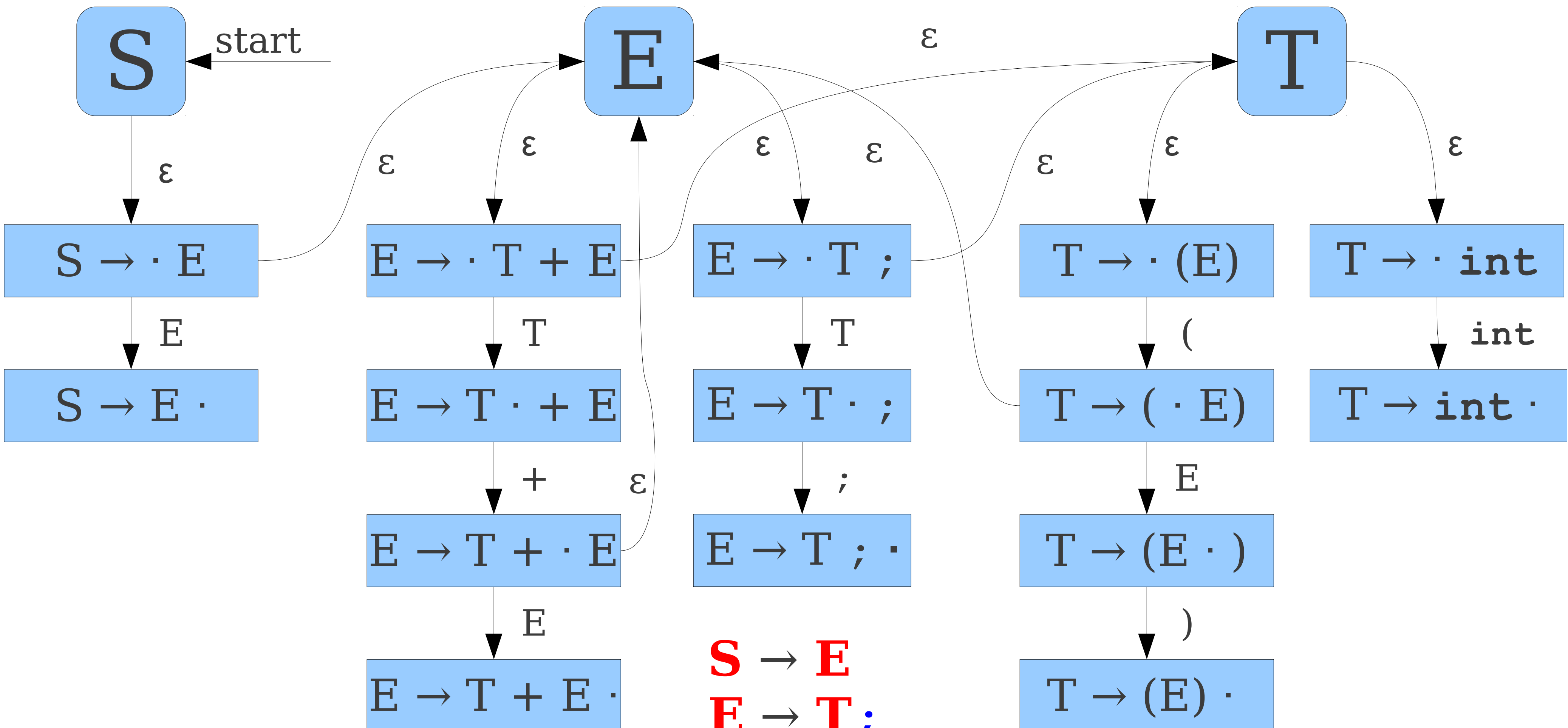
S → · E
E → · E + F
E → E + · F
F → F * · T
T → `int` ·

E	+	F	*	int	|	+	int

An Important Result

- There are only finitely many productions, and within those productions only finitely many positions.
- At any point in time, we only need to track where we are in one production.
- There are only finitely many options we can take at any one point.
- **We can use a finite automaton as our recognizer.**

An Automaton for Left Areas



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

Constructing the Automaton

- Create a state for each nonterminal.
- For each production $\mathbf{A} \to \boldsymbol{\gamma}$:
  - Construct states $\mathbf{A} \to \boldsymbol{\alpha} \cdot \boldsymbol{\omega}$ for each possible way of splitting $\boldsymbol{\gamma}$ into two substrings $\boldsymbol{\alpha}$ and $\boldsymbol{\omega}$.
  - Add transitions on $\boldsymbol{x}$ between $\mathbf{A} \to \boldsymbol{\alpha} \cdot \boldsymbol{x\omega}$ and $\mathbf{A} \to \boldsymbol{\alpha x} \cdot \boldsymbol{\omega}$.
- For each state $\mathbf{A} \to \boldsymbol{\alpha} \cdot \mathbf{B}\boldsymbol{\omega}$ for nonterminal $\mathbf{B}$, add an ε-transition from $\mathbf{A} \to \boldsymbol{\alpha} \cdot \mathbf{B}\boldsymbol{\omega}$ to $\mathbf{B}$.

Why This Matters

- Our initial goal was to find handles.
- When running this automaton, if we ever end up in a state with a rule of the form

$$\mathbf{A} \rightarrow \boldsymbol{\omega} \cdot$$

- Then we might be looking at a handle.
- This automaton can be used to discover possible handle locations!

Adding Determinism

- Typically, this handle-finding automaton is implemented deterministically.
- We could construct a deterministic parsing automaton by constructing the nondeterministic automaton and applying the subset construction, but there is a more direct approach.

A Deterministic Automaton

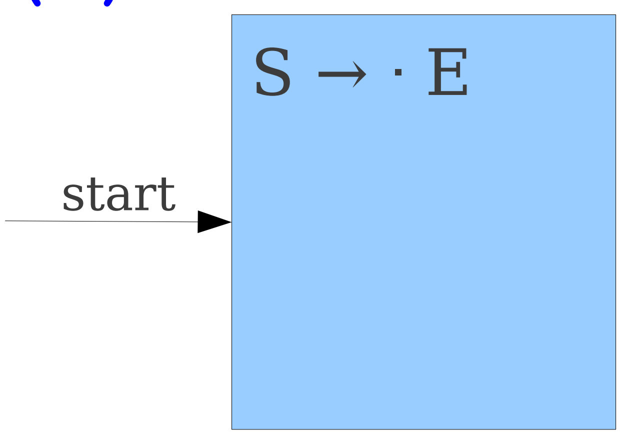
$S \rightarrow E$

$E \rightarrow T$ `;`

$E \rightarrow T$ `+` E

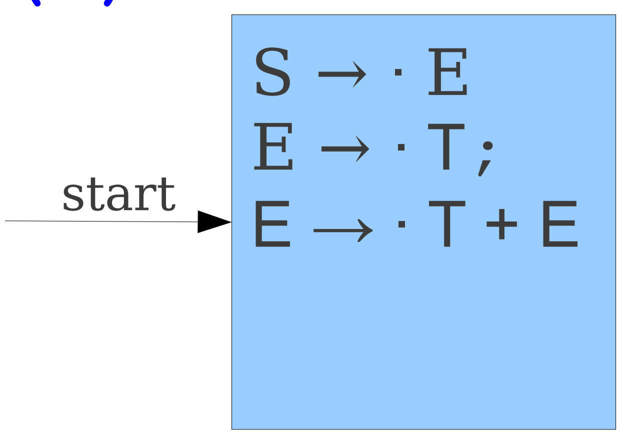
$T \rightarrow$ `int`

$T \rightarrow$ `(`E`)`



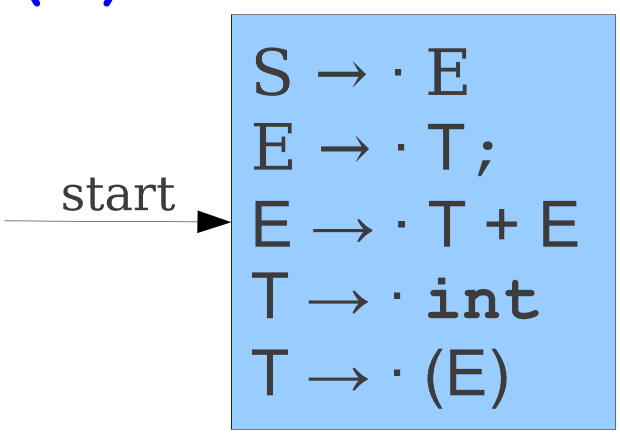
A Deterministic Automaton

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



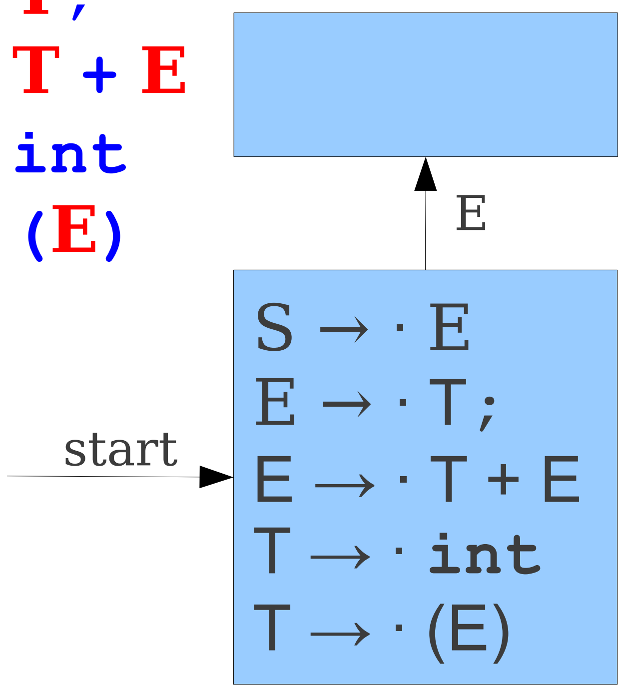
A Deterministic Automaton

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



A Deterministic Automaton

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



A Deterministic Automaton

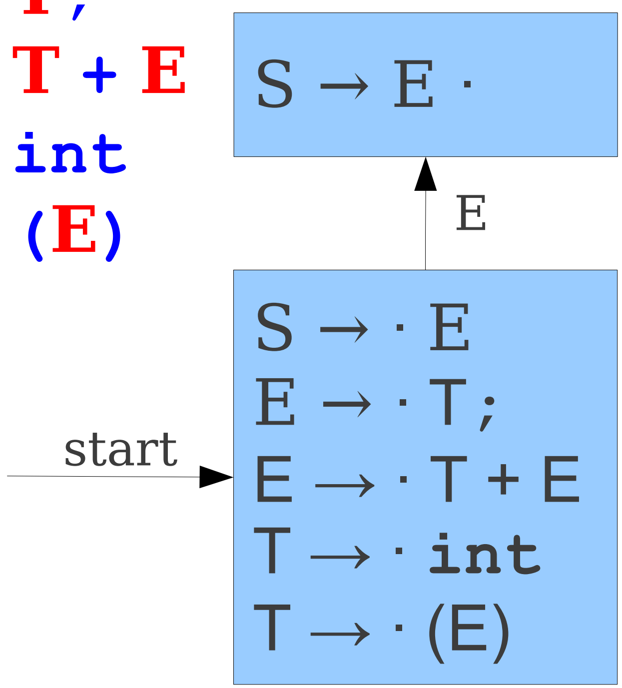
$\mathbf{S} \rightarrow \mathbf{E}$

$\mathbf{E} \rightarrow \mathbf{T}$ `;`

$\mathbf{E} \rightarrow \mathbf{T}$ `+` $\mathbf{E}$

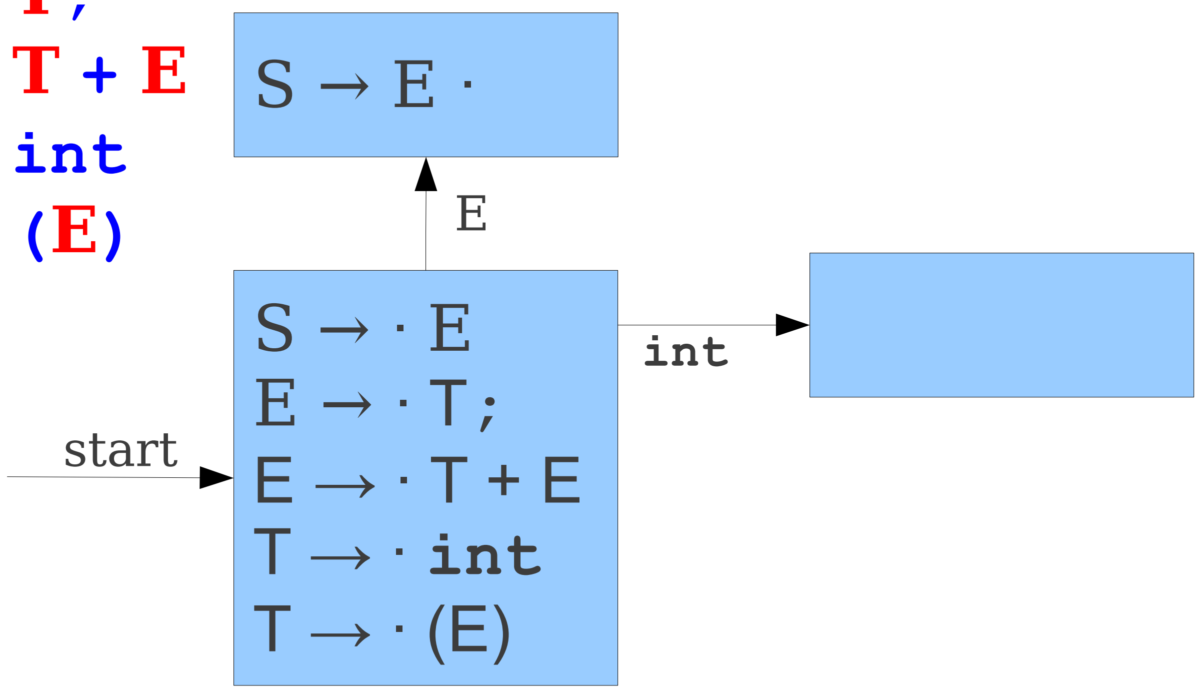
$\mathbf{T} \rightarrow$ `int`

$\mathbf{T} \rightarrow$ `(`$\mathbf{E}$`)`



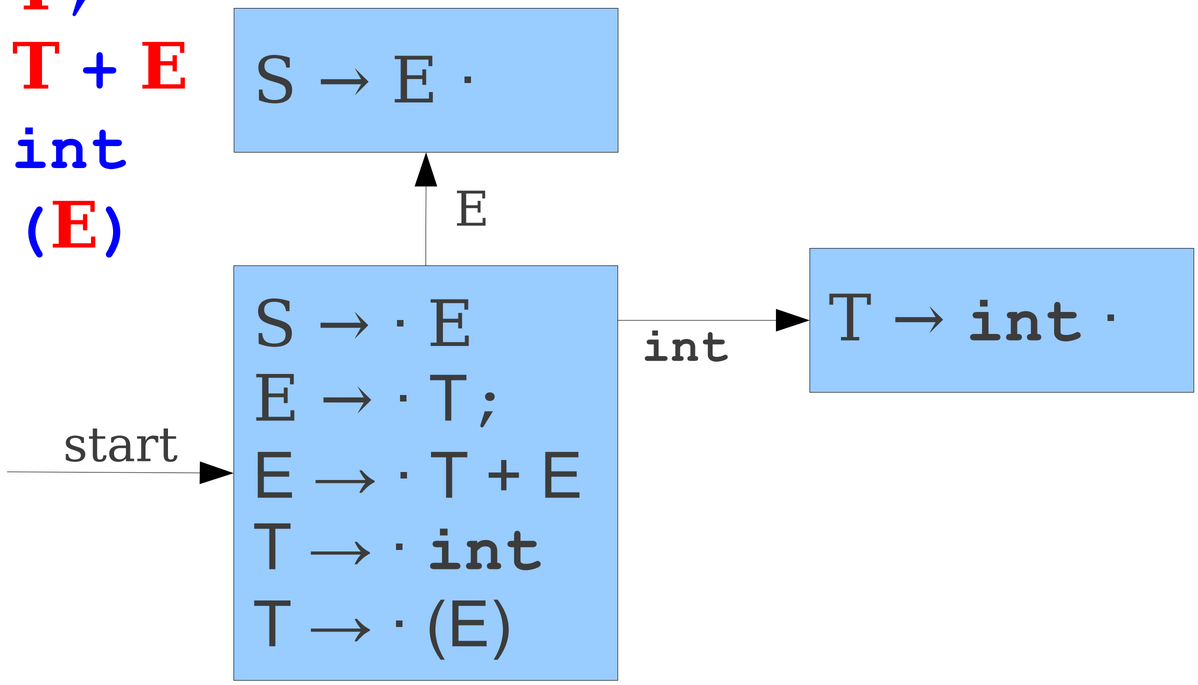
A Deterministic Automaton

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



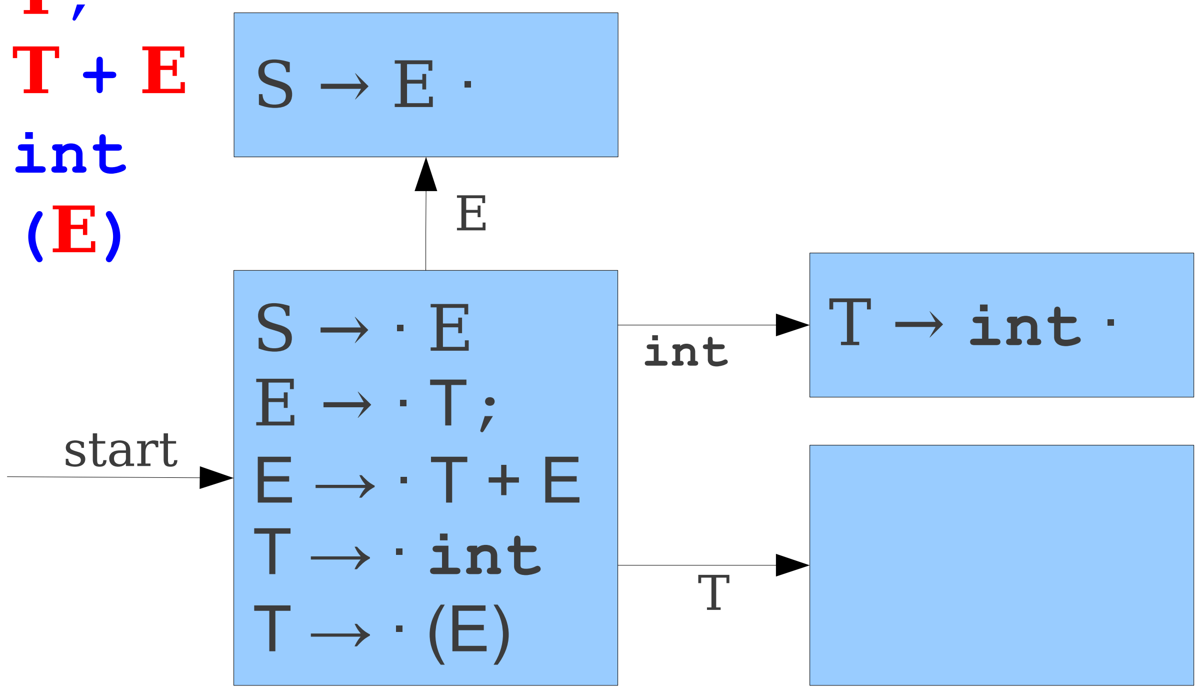
A Deterministic Automaton

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



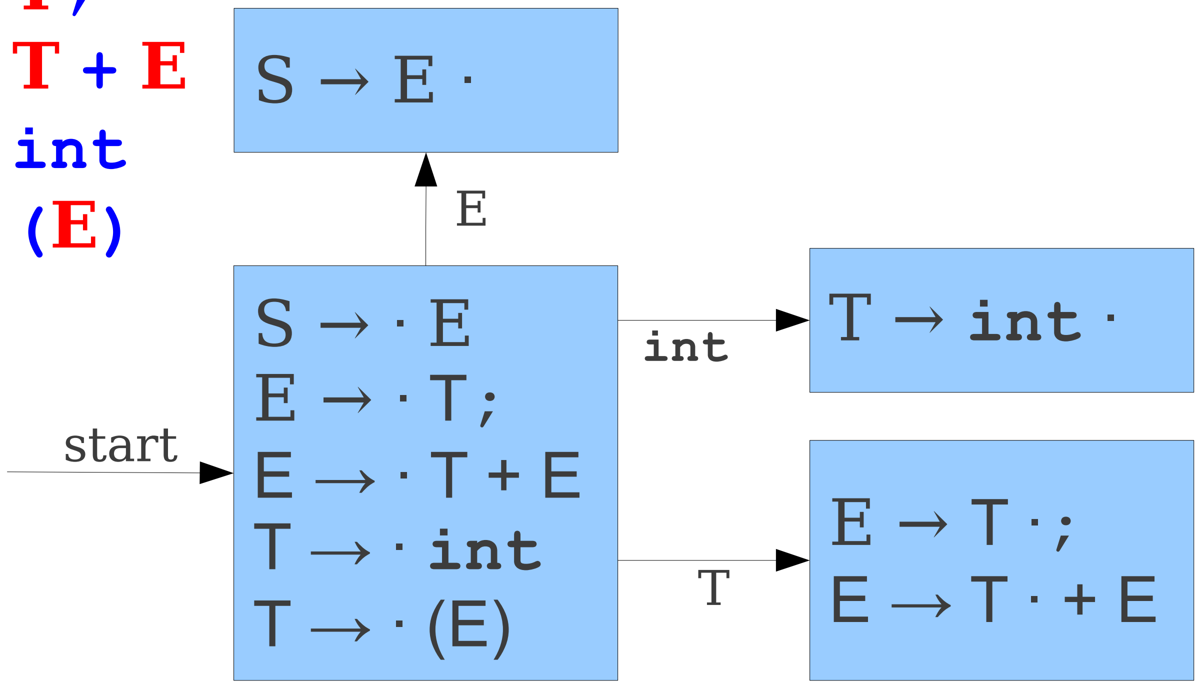
A Deterministic Automaton

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



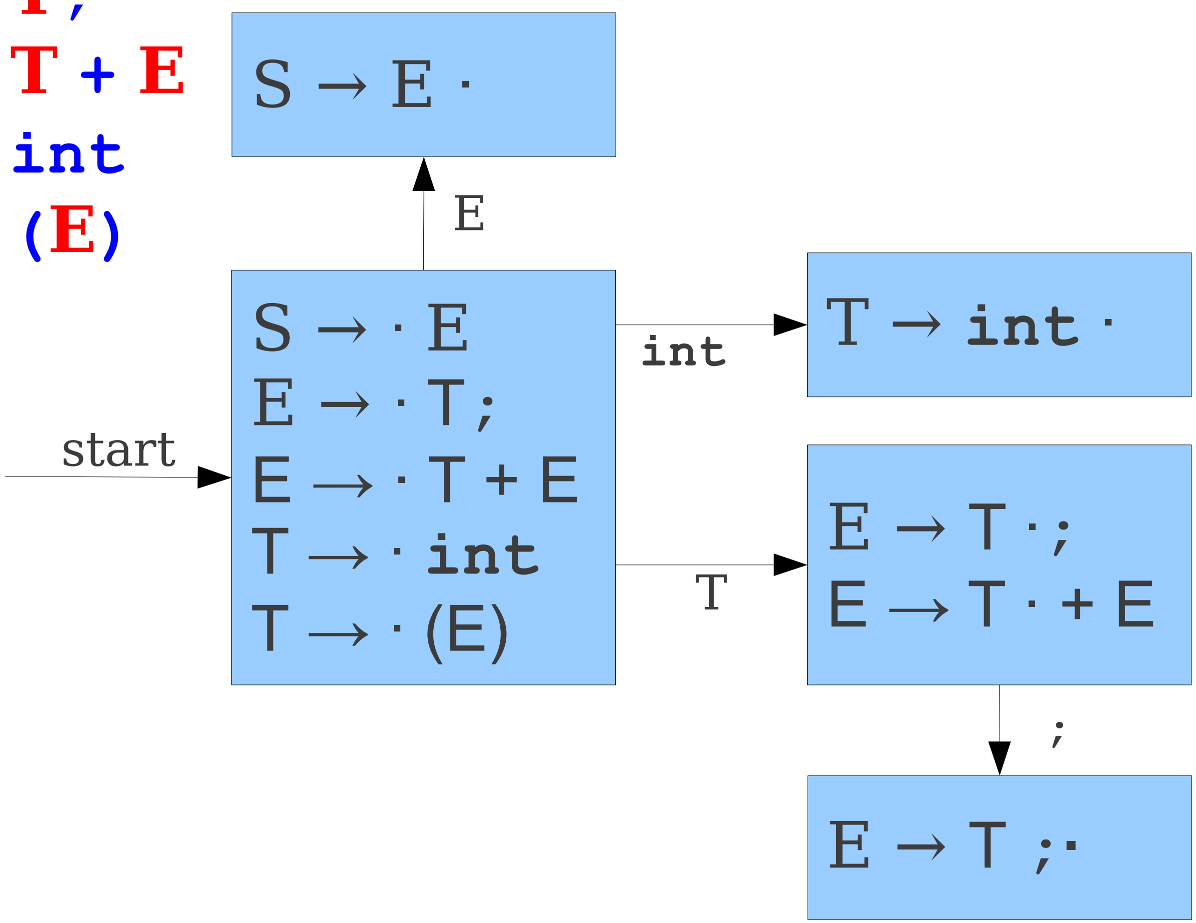
A Deterministic Automaton

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



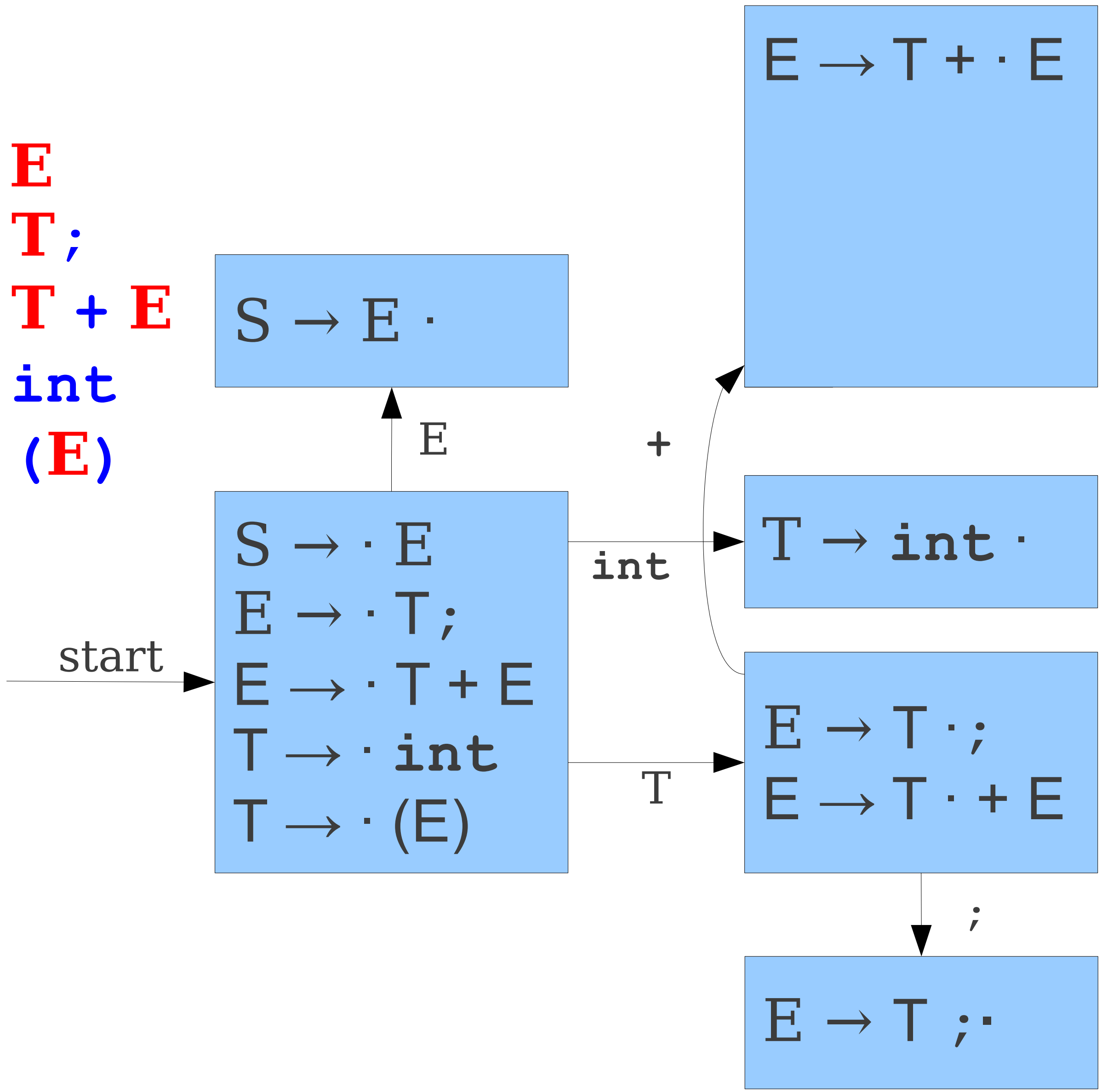
A Deterministic Automaton

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



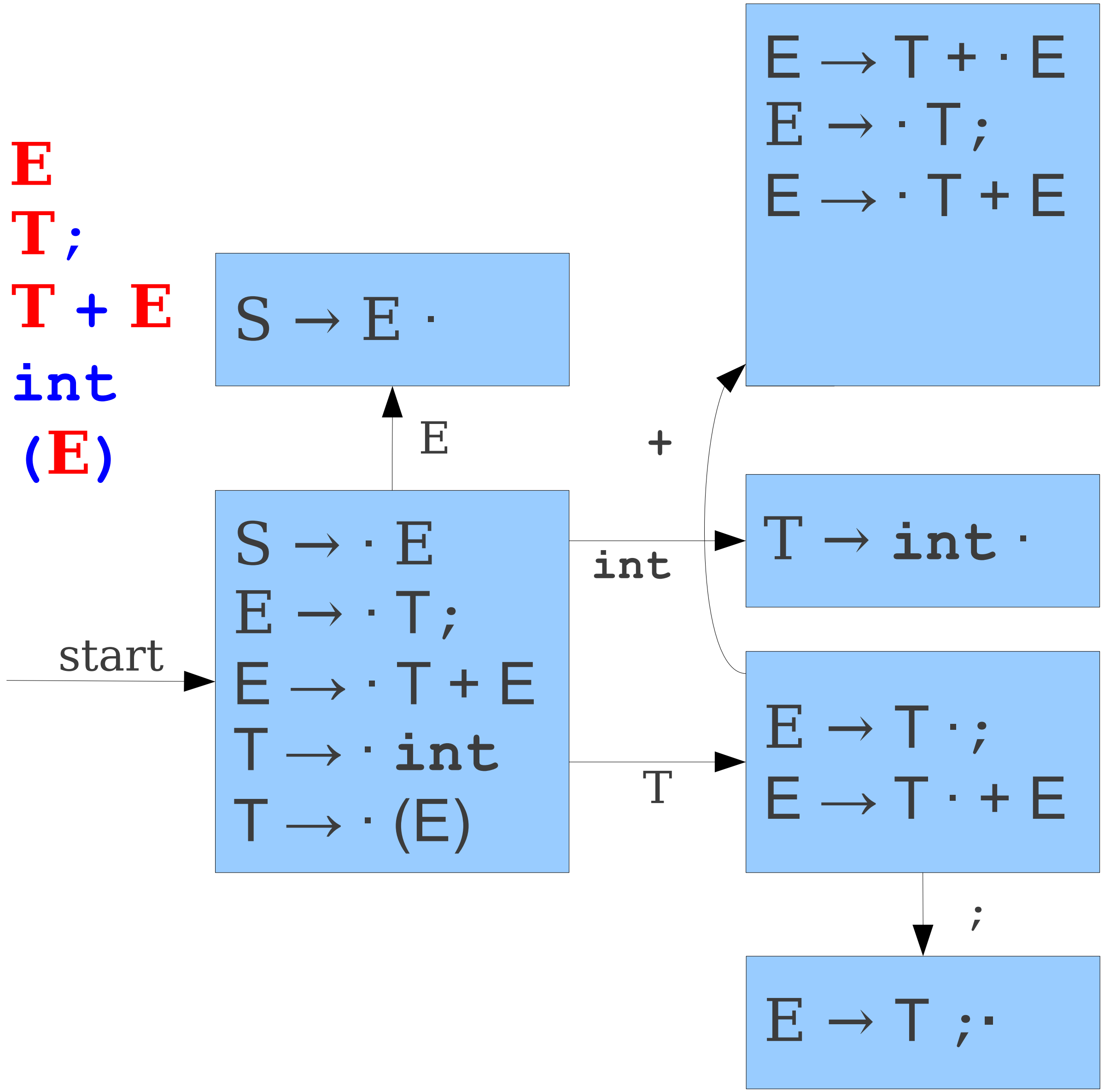
A Deterministic Automaton

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



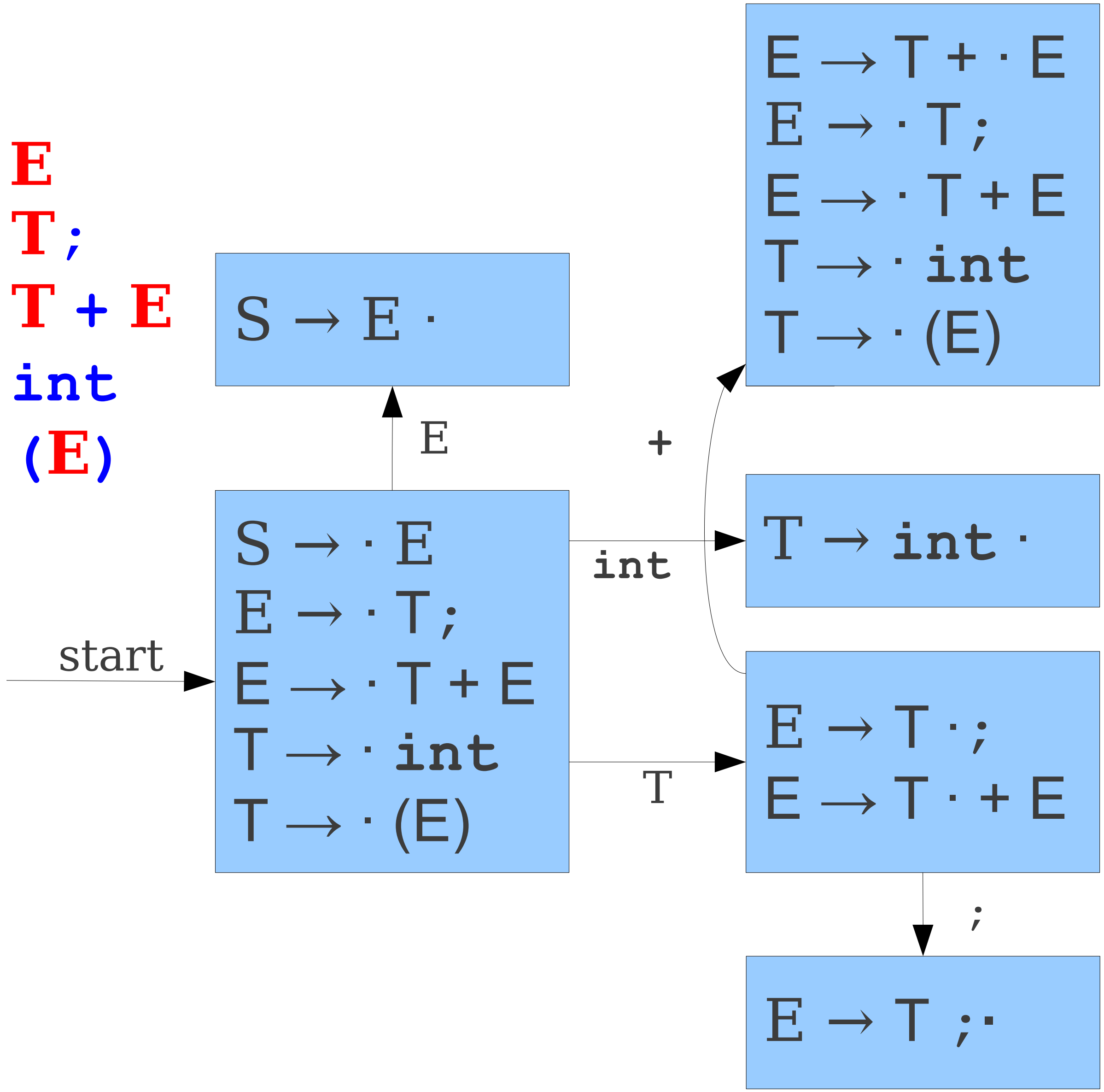
A Deterministic Automaton

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



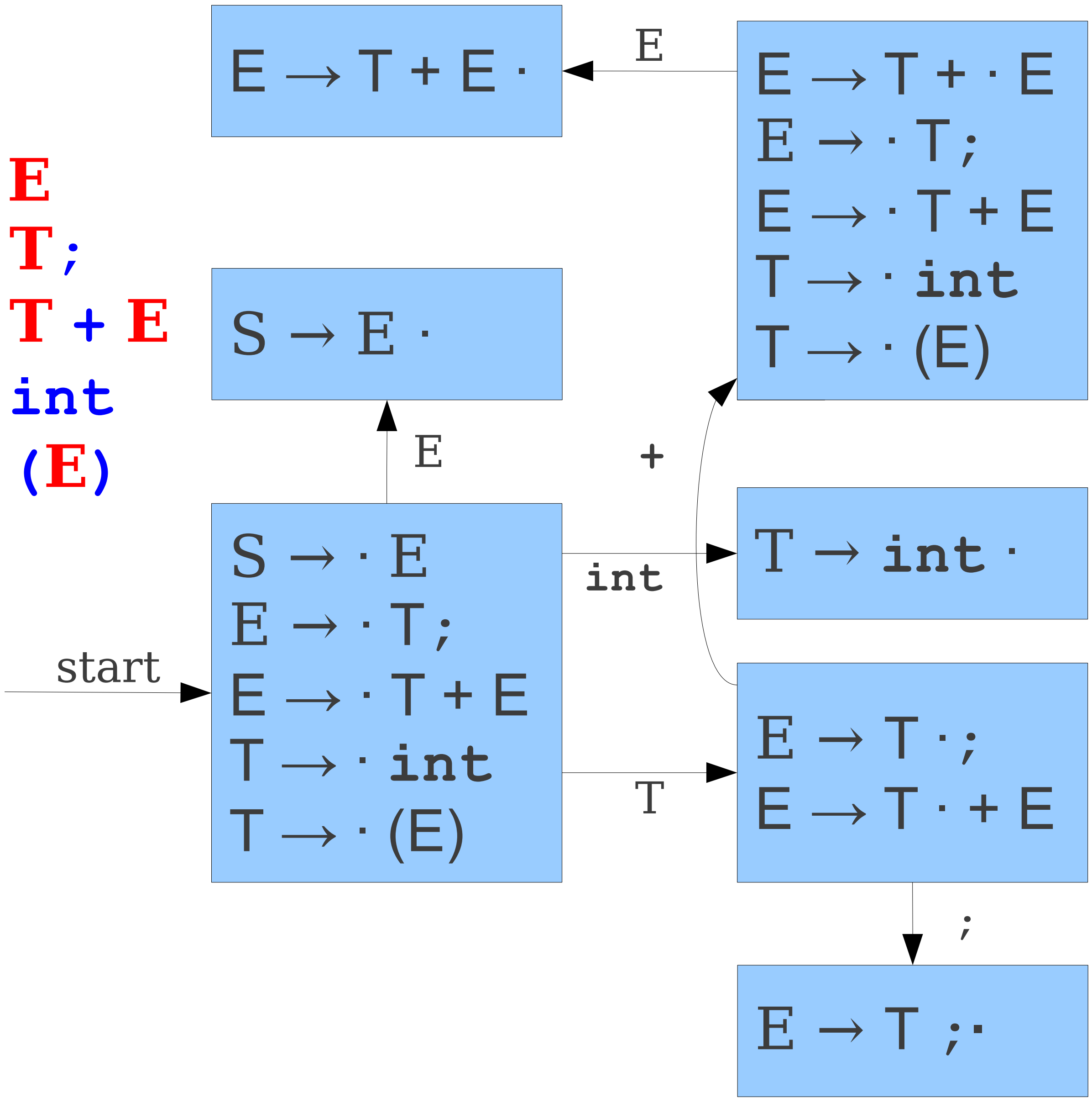
A Deterministic Automaton

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



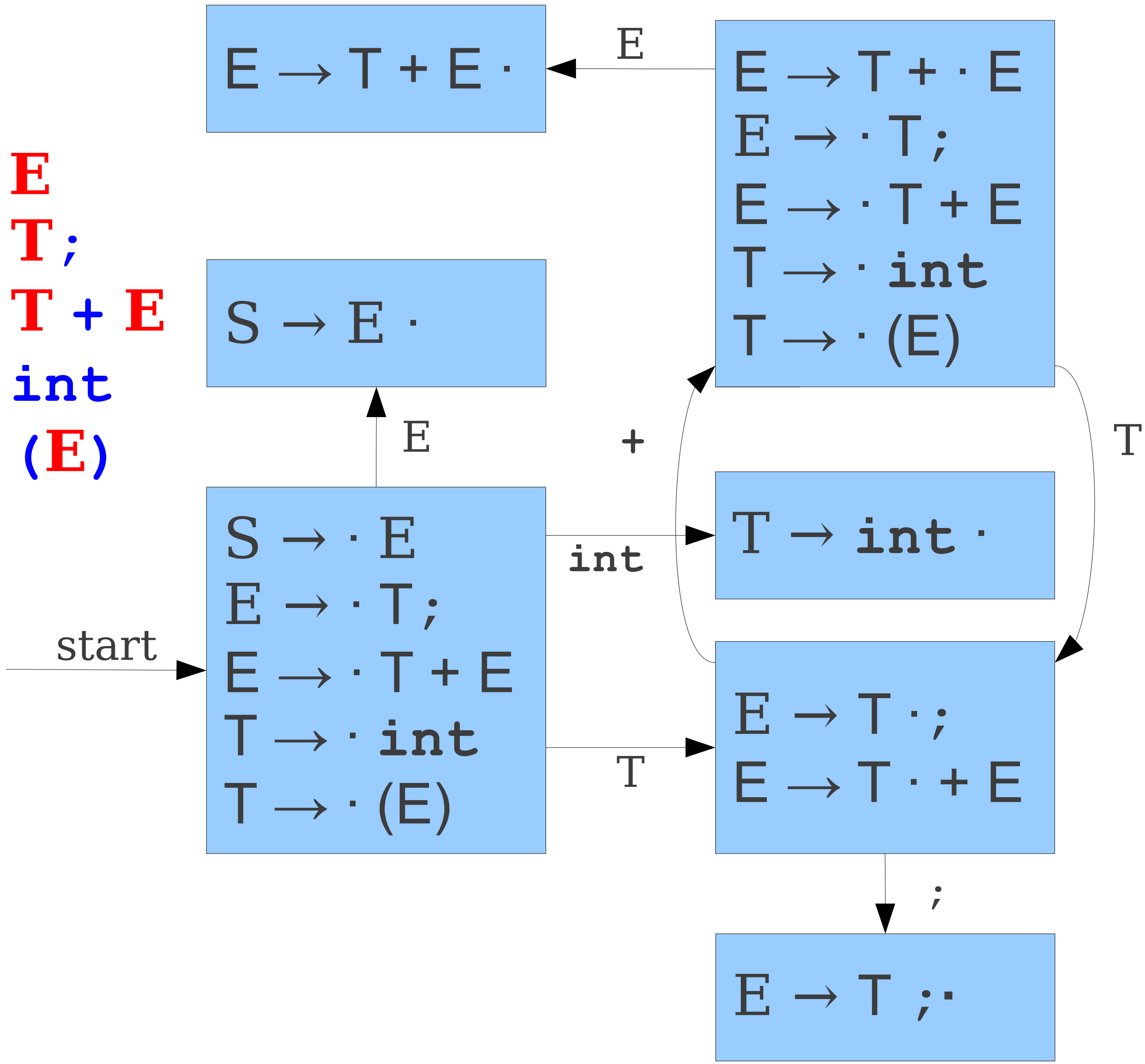
A Deterministic Automaton

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



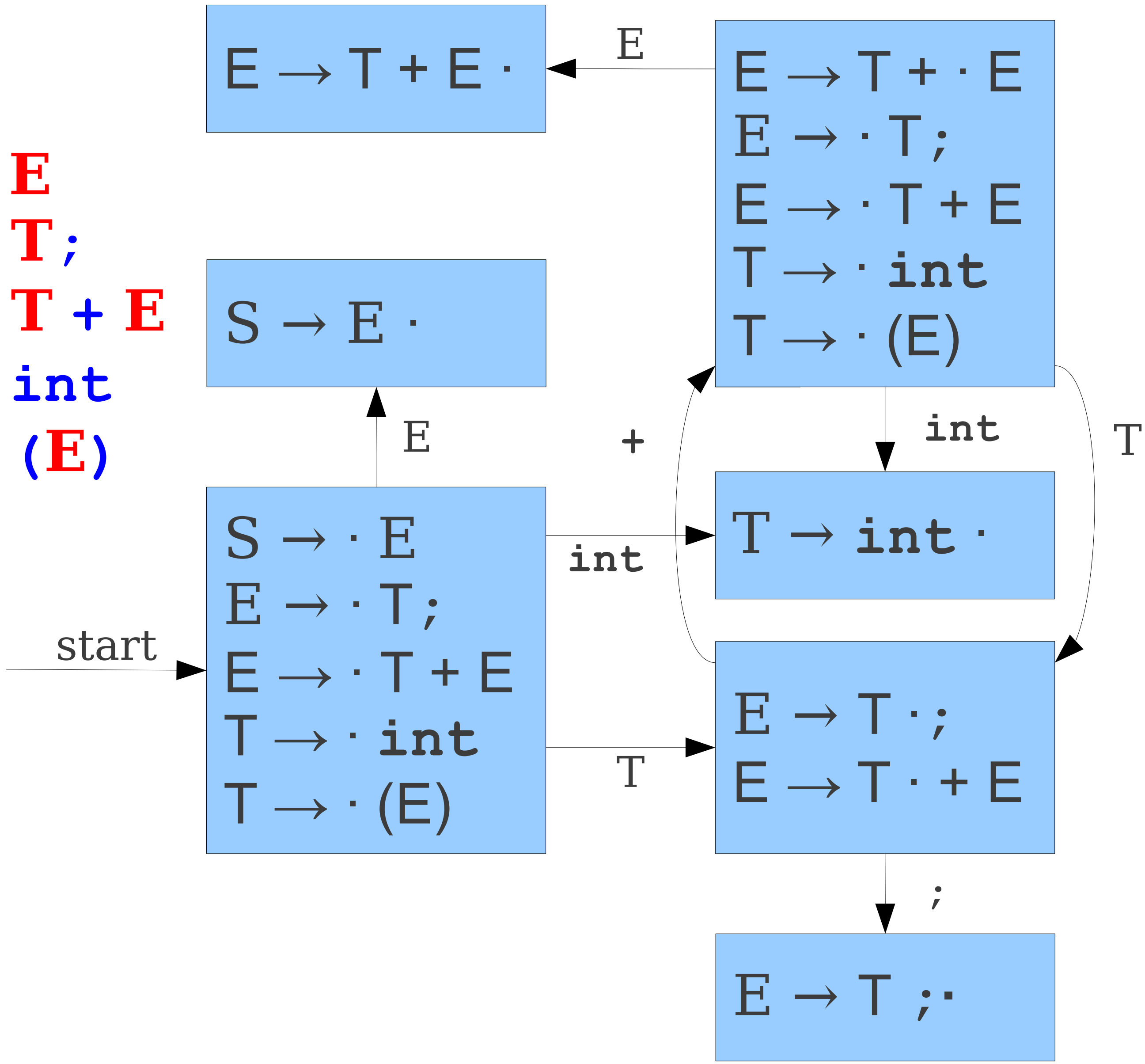
A Deterministic Automaton

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



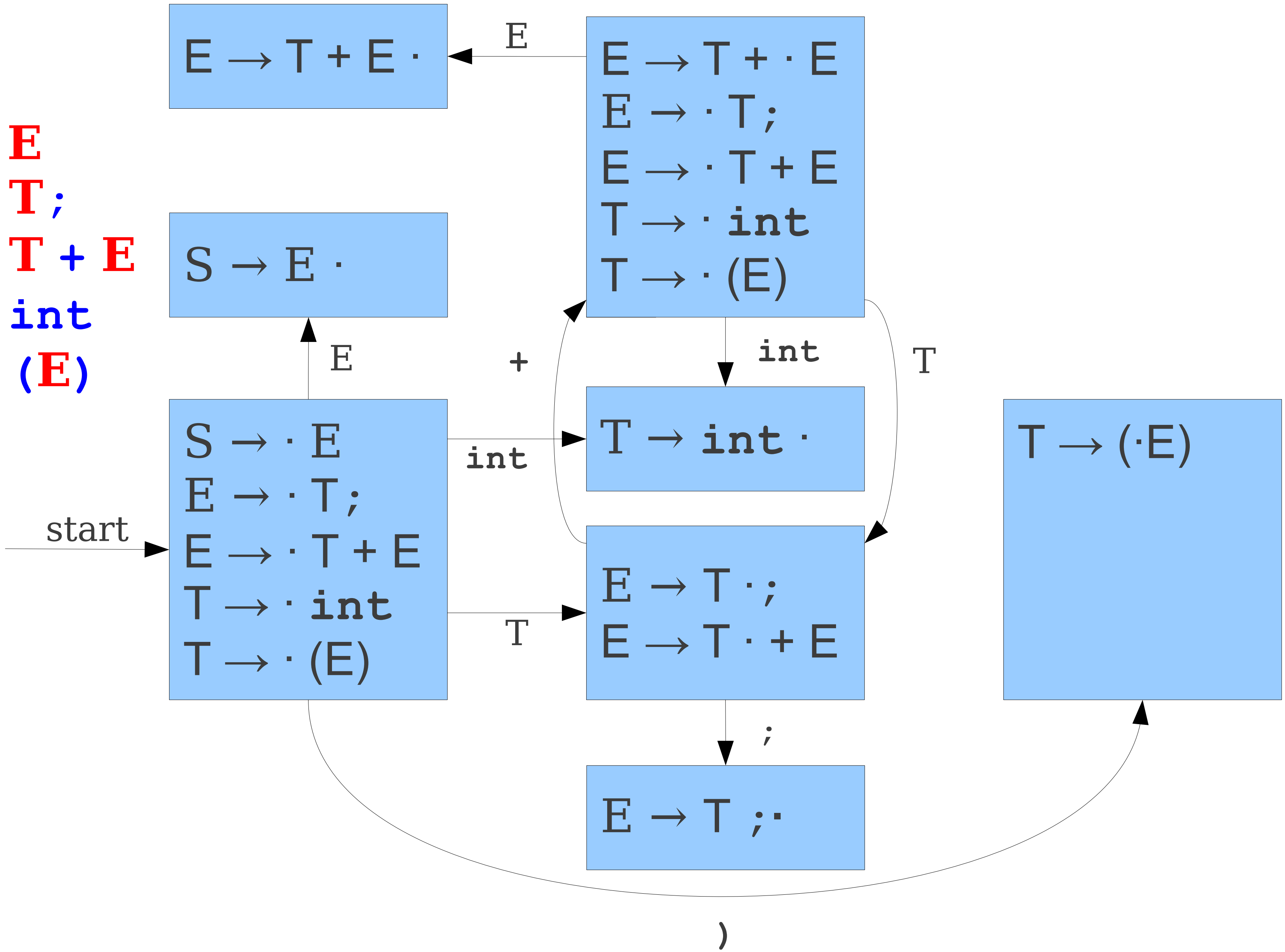
A Deterministic Automaton

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



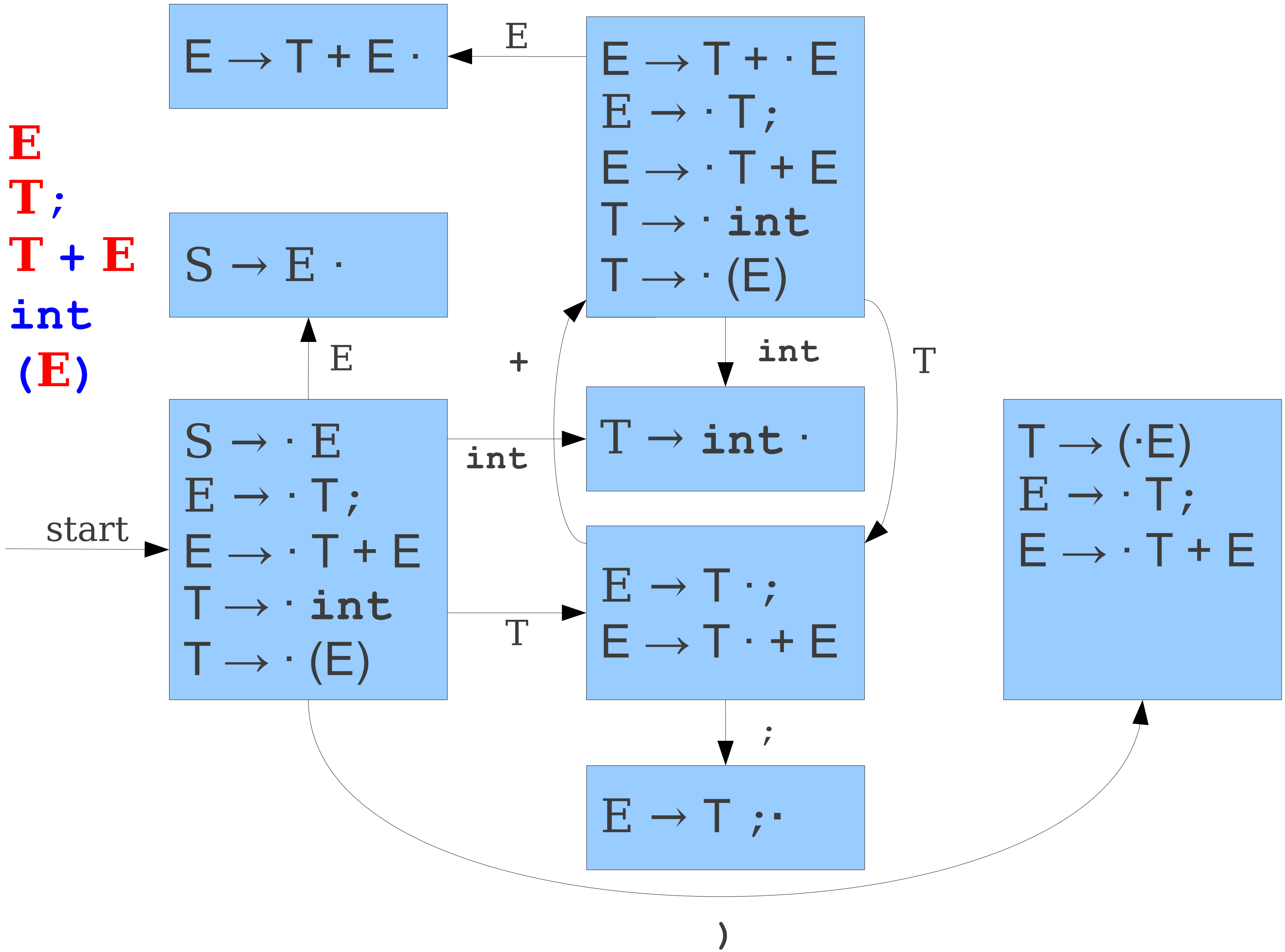
A Deterministic Automaton

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



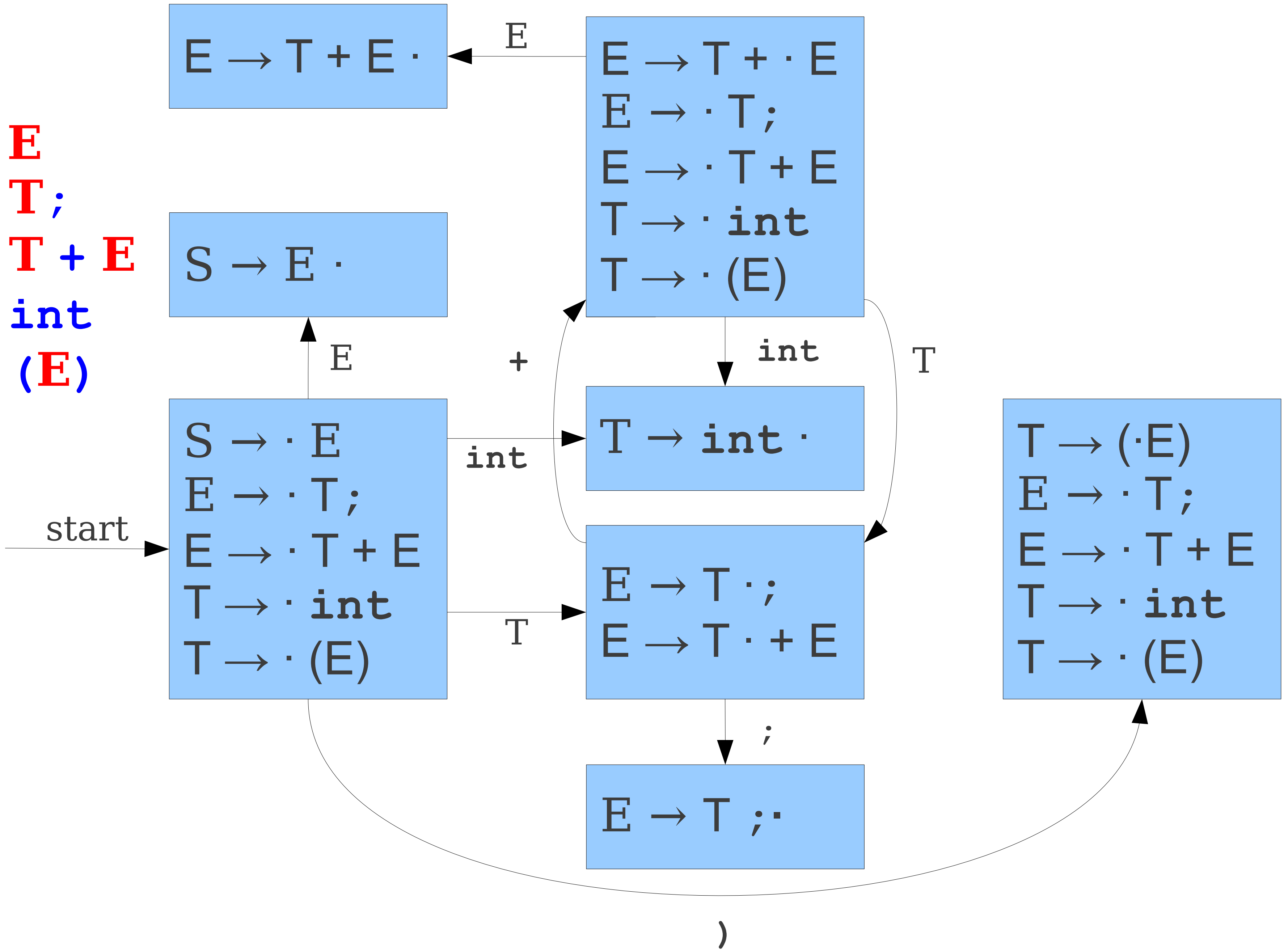
A Deterministic Automaton

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



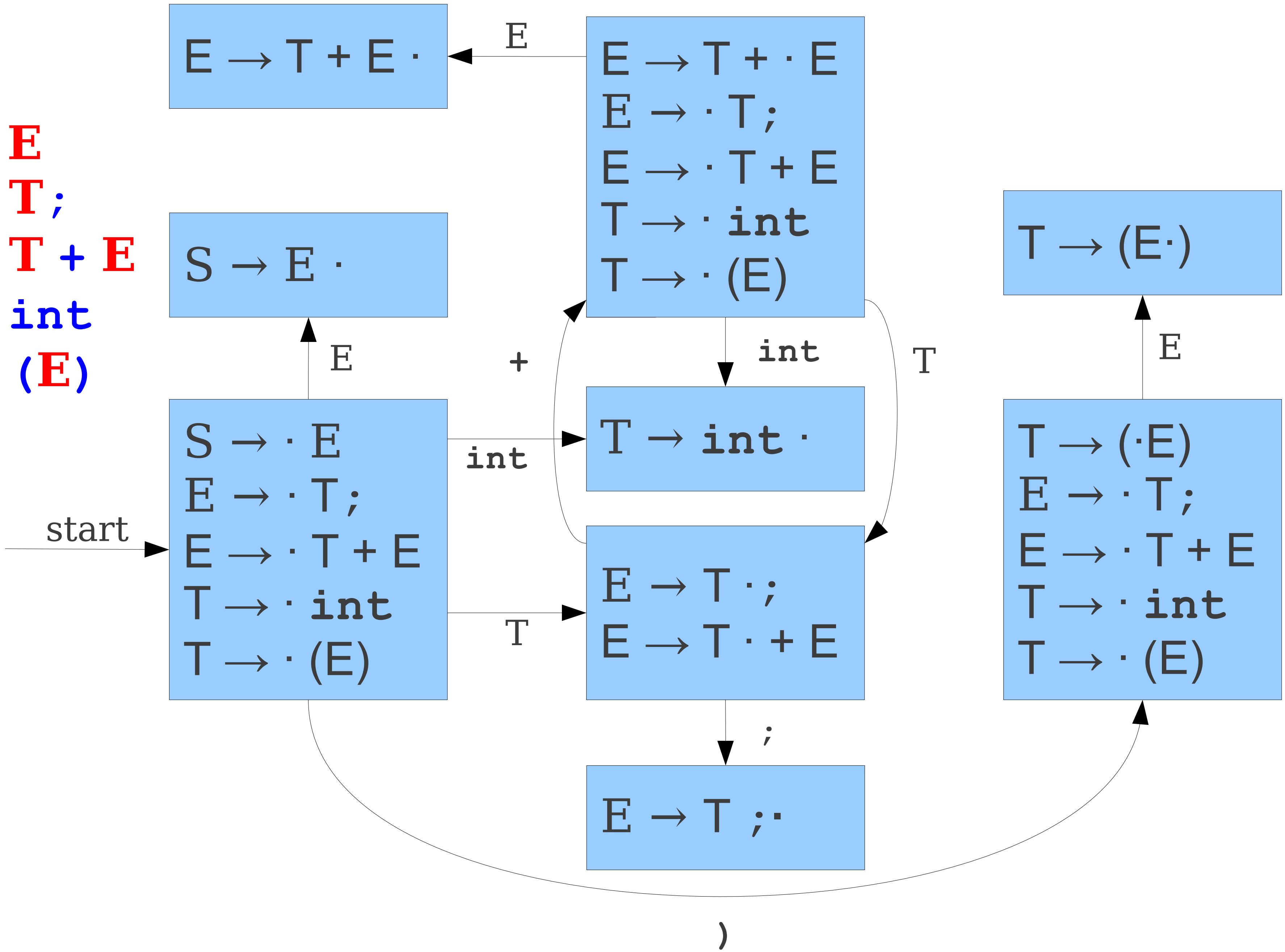
A Deterministic Automaton

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



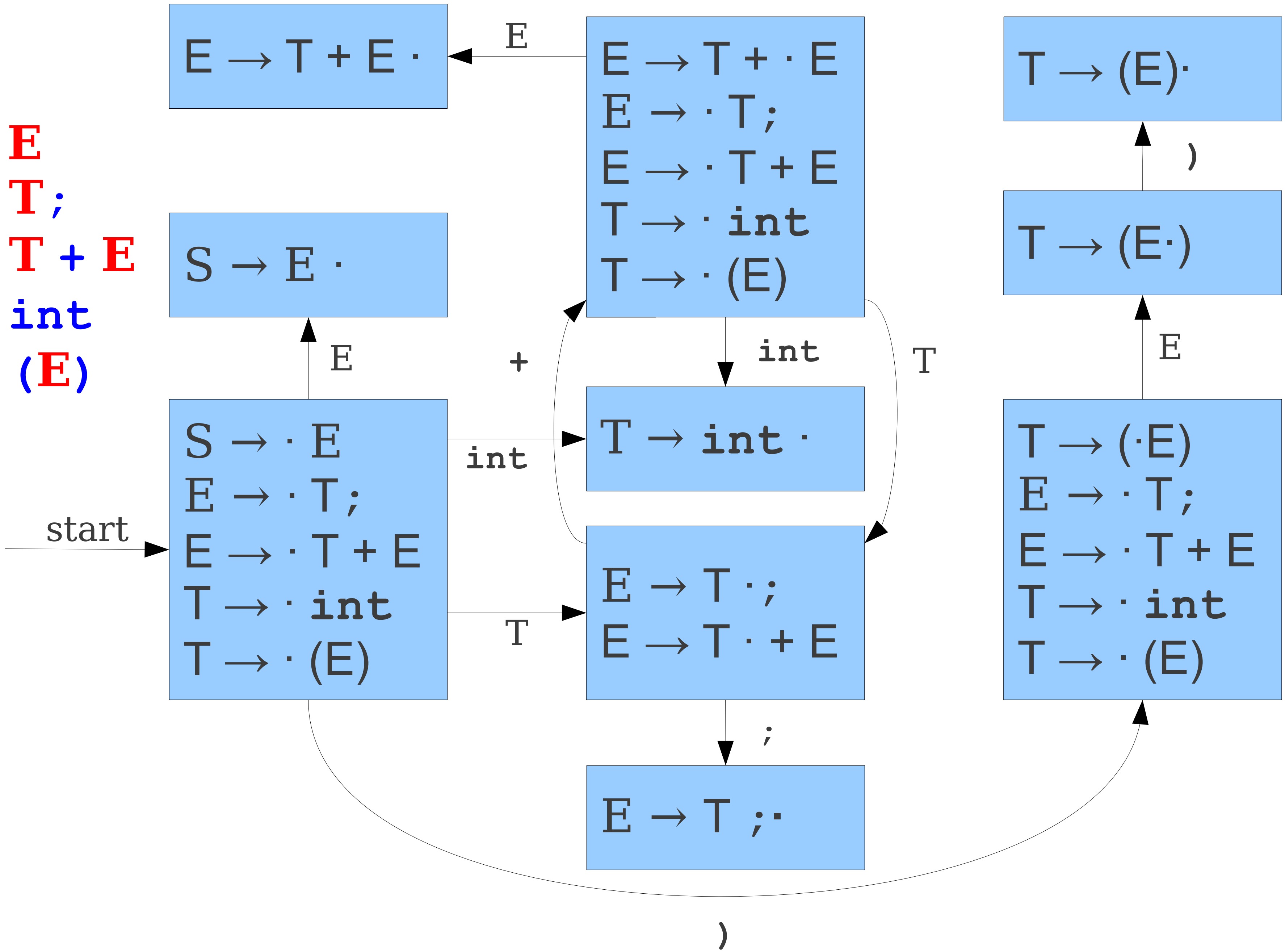
A Deterministic Automaton

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



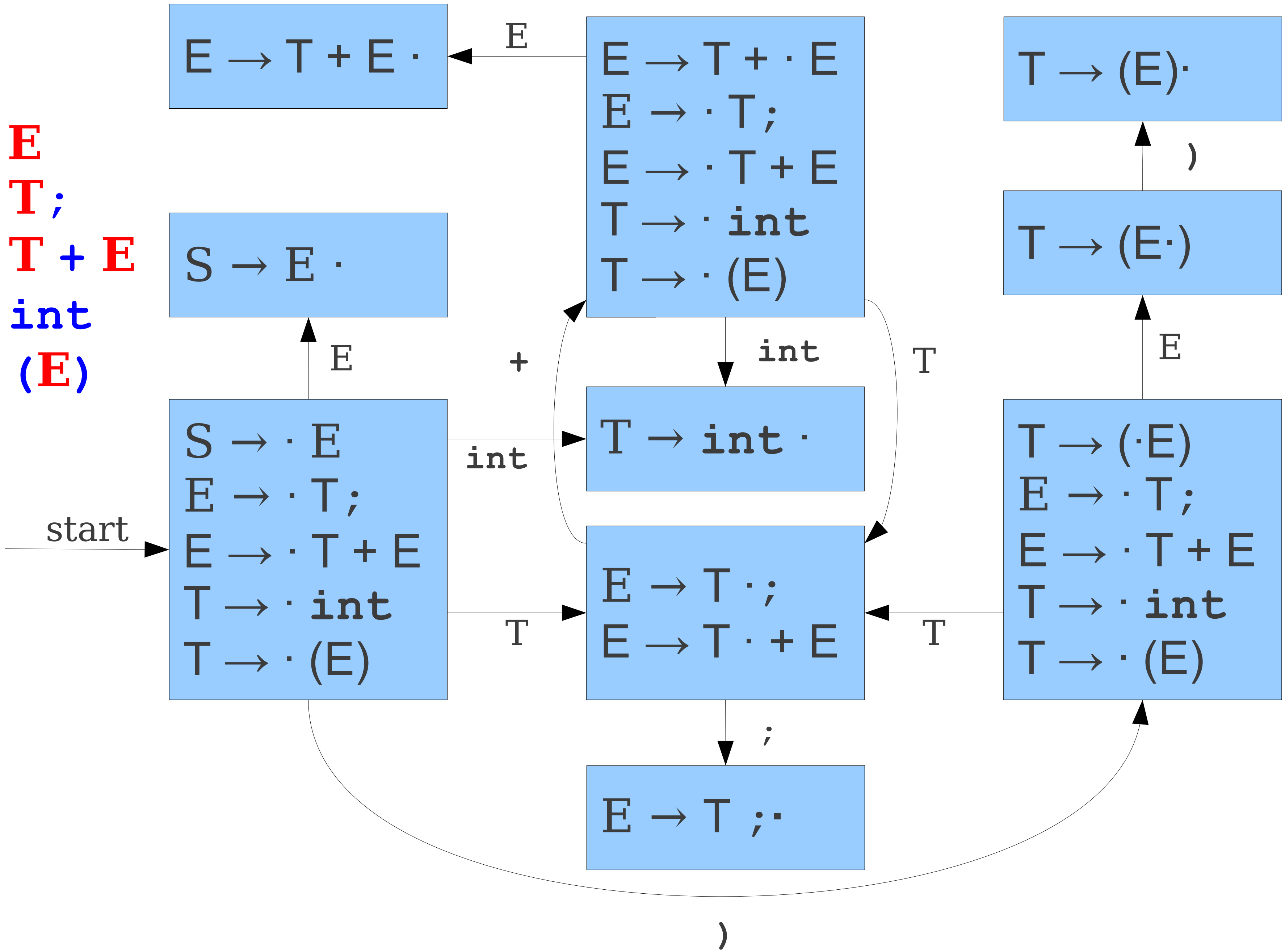
A Deterministic Automaton

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



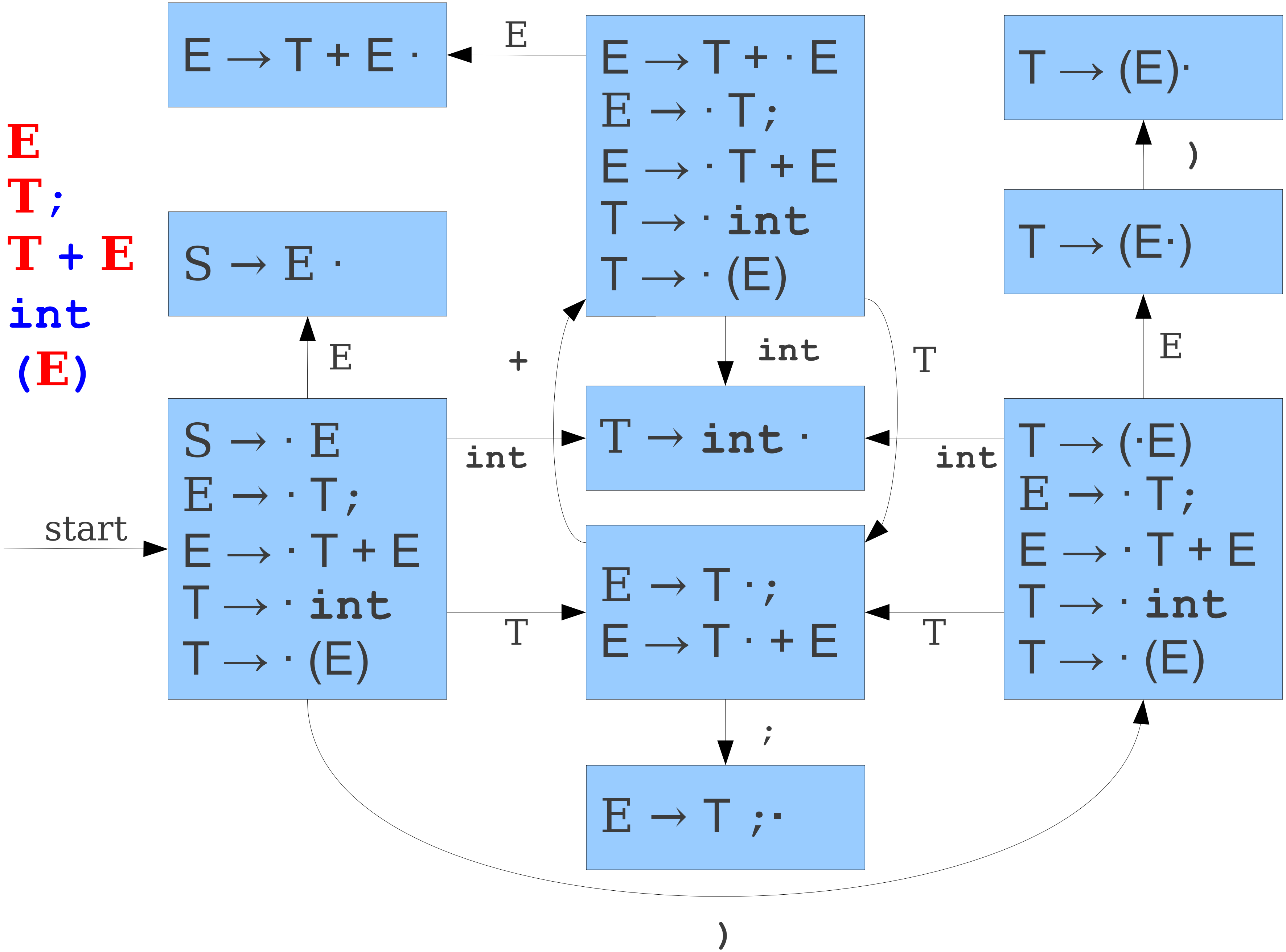
A Deterministic Automaton

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



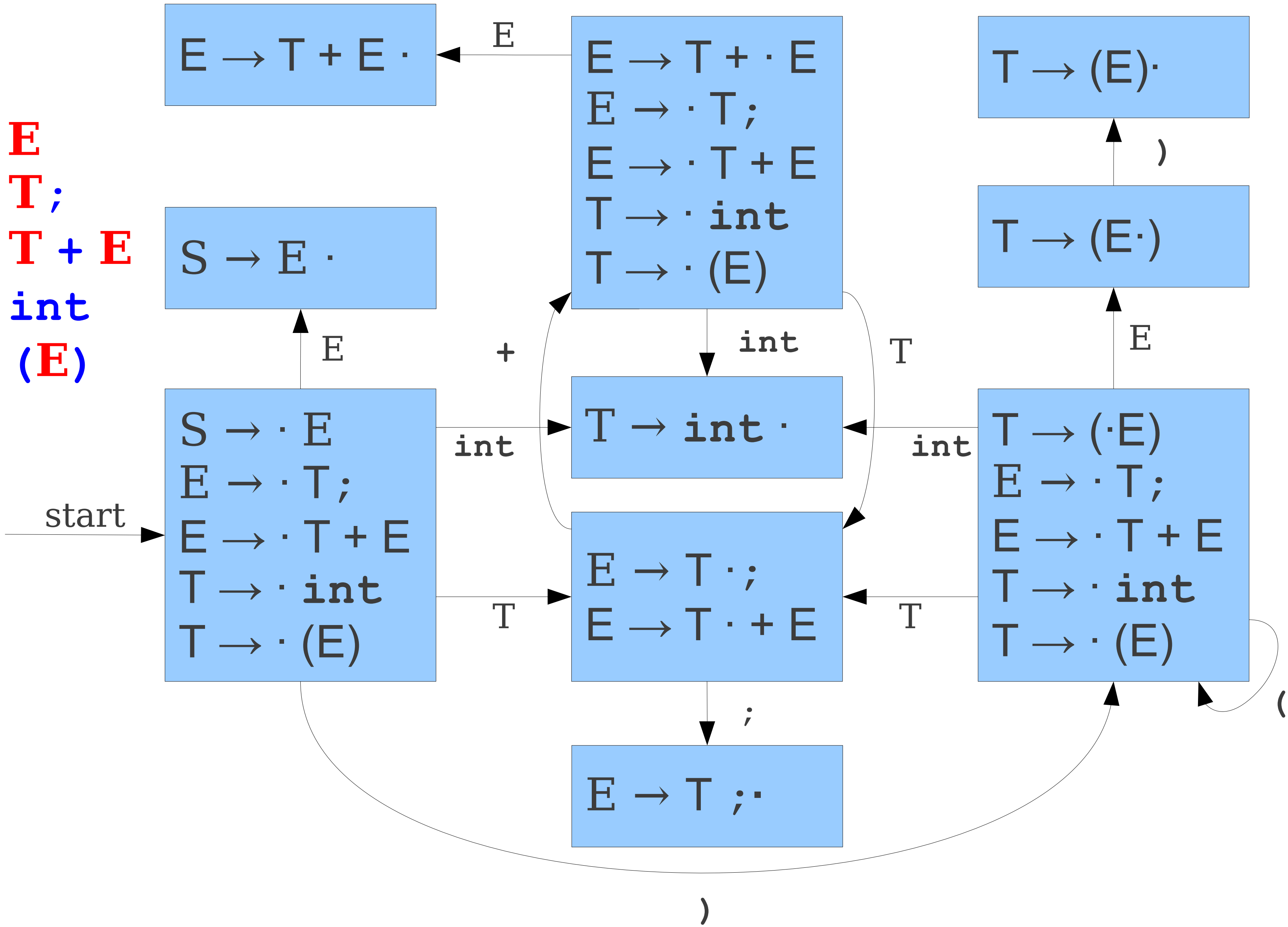
A Deterministic Automaton

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



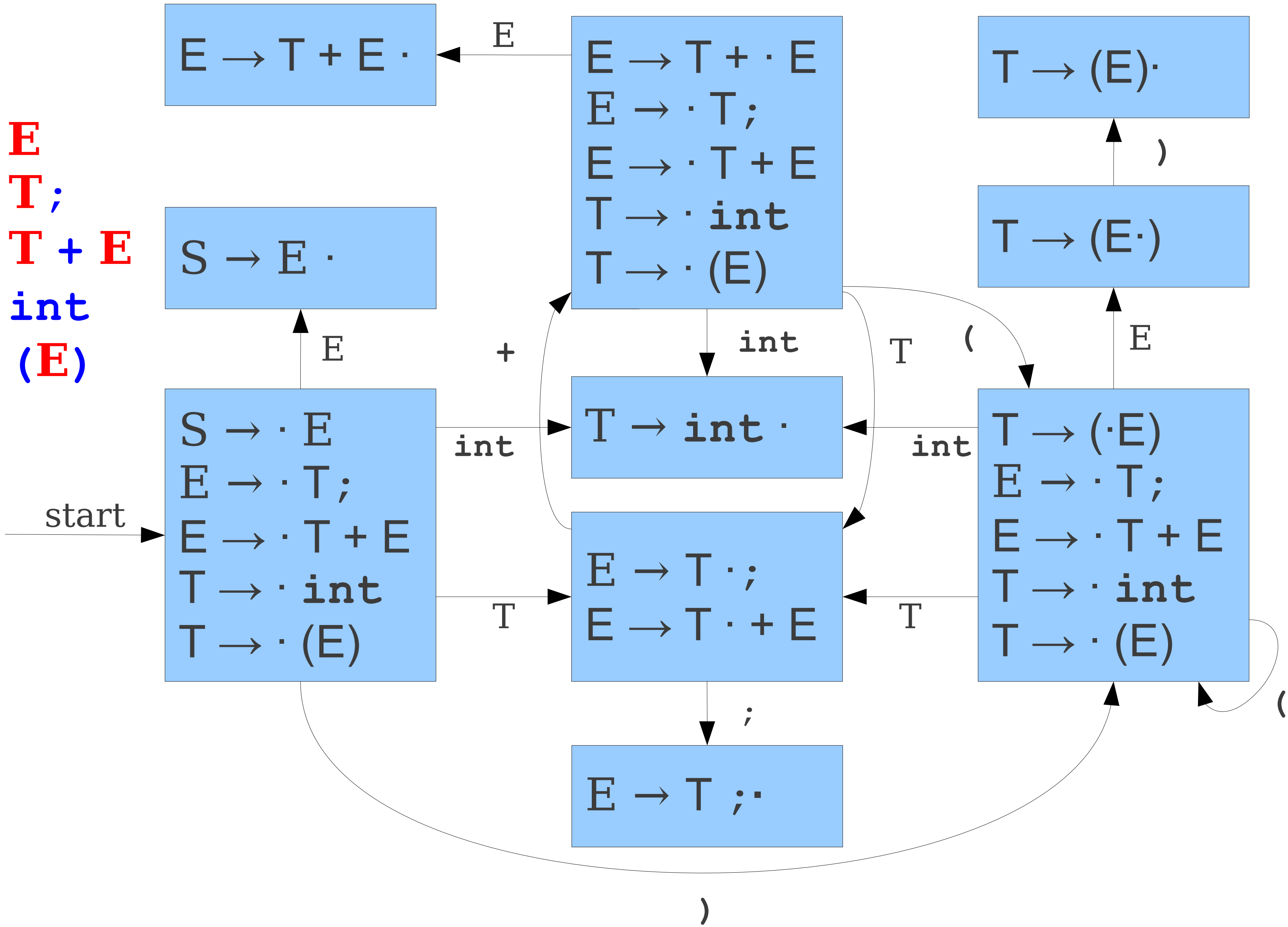
A Deterministic Automaton

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



A Deterministic Automaton

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



Constructing the Automaton II

- Begin in a state containing $\mathbf{S} \rightarrow \cdot\ \mathbf{A}$, where $\mathbf{S}$ is the augmented start symbol.
- Compute the **closure** of the state:
  - If $\mathbf{A} \rightarrow \boldsymbol{\alpha} \cdot \mathbf{B}\boldsymbol{\omega}$ is in the state, add $\mathbf{B} \rightarrow \cdot\ \boldsymbol{\gamma}$ to the state for each production $\mathbf{B} \rightarrow \boldsymbol{\gamma}$.
  - Yet another fixed-point iteration!
- Repeat until no new states are added:
  - If a state contains a production $\mathbf{A} \rightarrow \boldsymbol{\alpha} \cdot \boldsymbol{x}\boldsymbol{\omega}$ for symbol $\boldsymbol{x}$, add a transition on $\boldsymbol{x}$ from that state to the state containing the closure of $\mathbf{A} \rightarrow \boldsymbol{\alpha}\boldsymbol{x} \cdot \boldsymbol{\omega}$
- This is equivalent to a subset construction on the NFA.

Handle-Finding Automata

- Handling-finding automata can be very large.
- NFA has states proportional to the size of the grammar, so DFA can have size exponential in the size of the grammar.
  - There are grammars that can exhibit this worst-case.
- Automata are almost always generated by tools like **`bison`**.

Finding Handles

- Where do we look for handles?
  - **At the top of the stack.**
- How do we search for handles?
  - **Build a handle-finding automaton.**
- How do we recognize handles?
  - Once we've found a possible handle, how do we confirm that it's correct?

Question Three:

How do we recognize handles?

Handle Recognition

- Our automaton will tell us all places where a handle might be.
- However, if the automaton says that there might be a handle at a given point, we need a way to confirm this.
- We'll thus use **predictive bottom-up parsing**:
  - Have a deterministic procedure for guessing where handles are.
- There are many predictive algorithms, each of which recognize different grammars.

Our First Algorithm: **LR(0)**

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

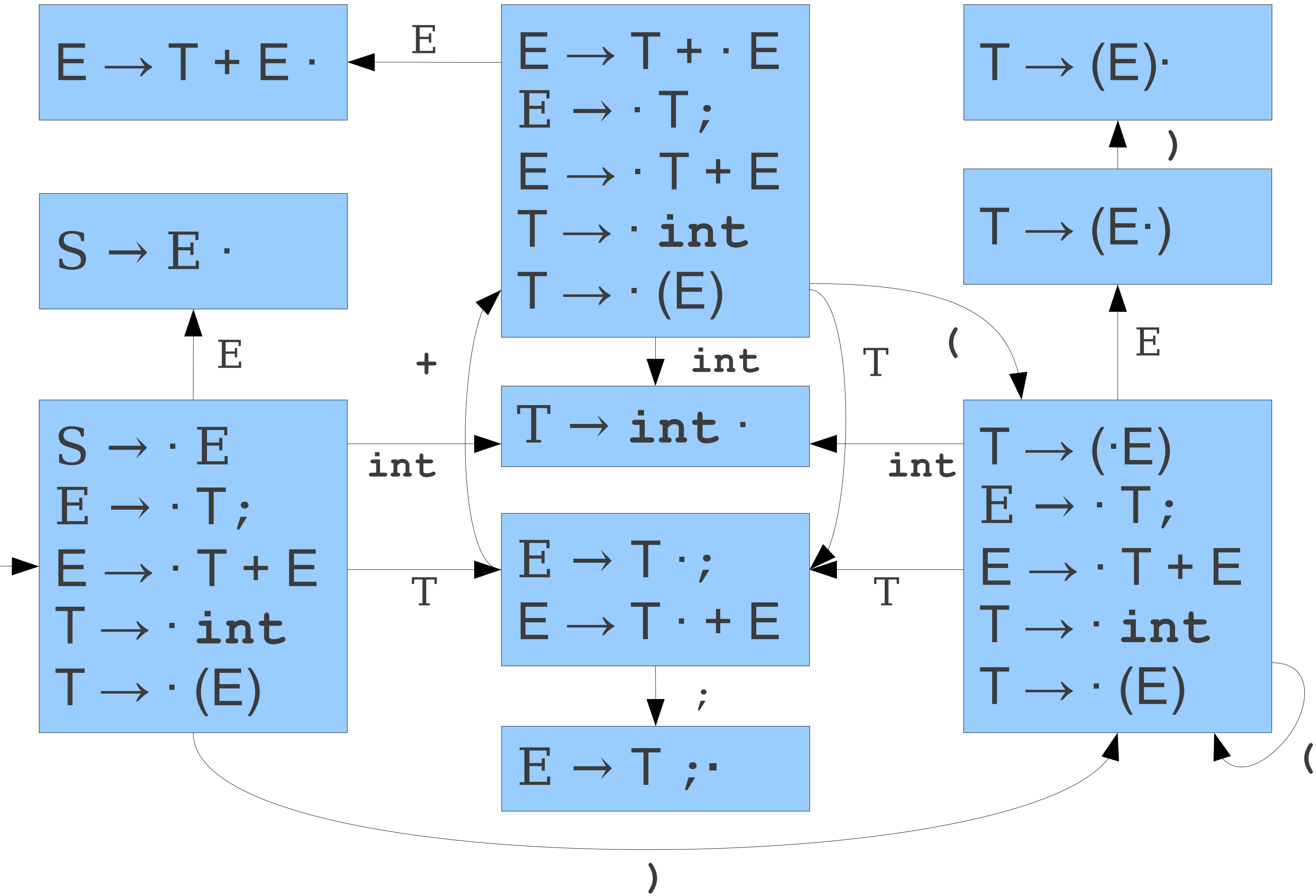
LR(0) Parsing

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

int	+	(	int	+	int	;	)	;

LR(0) Parsing

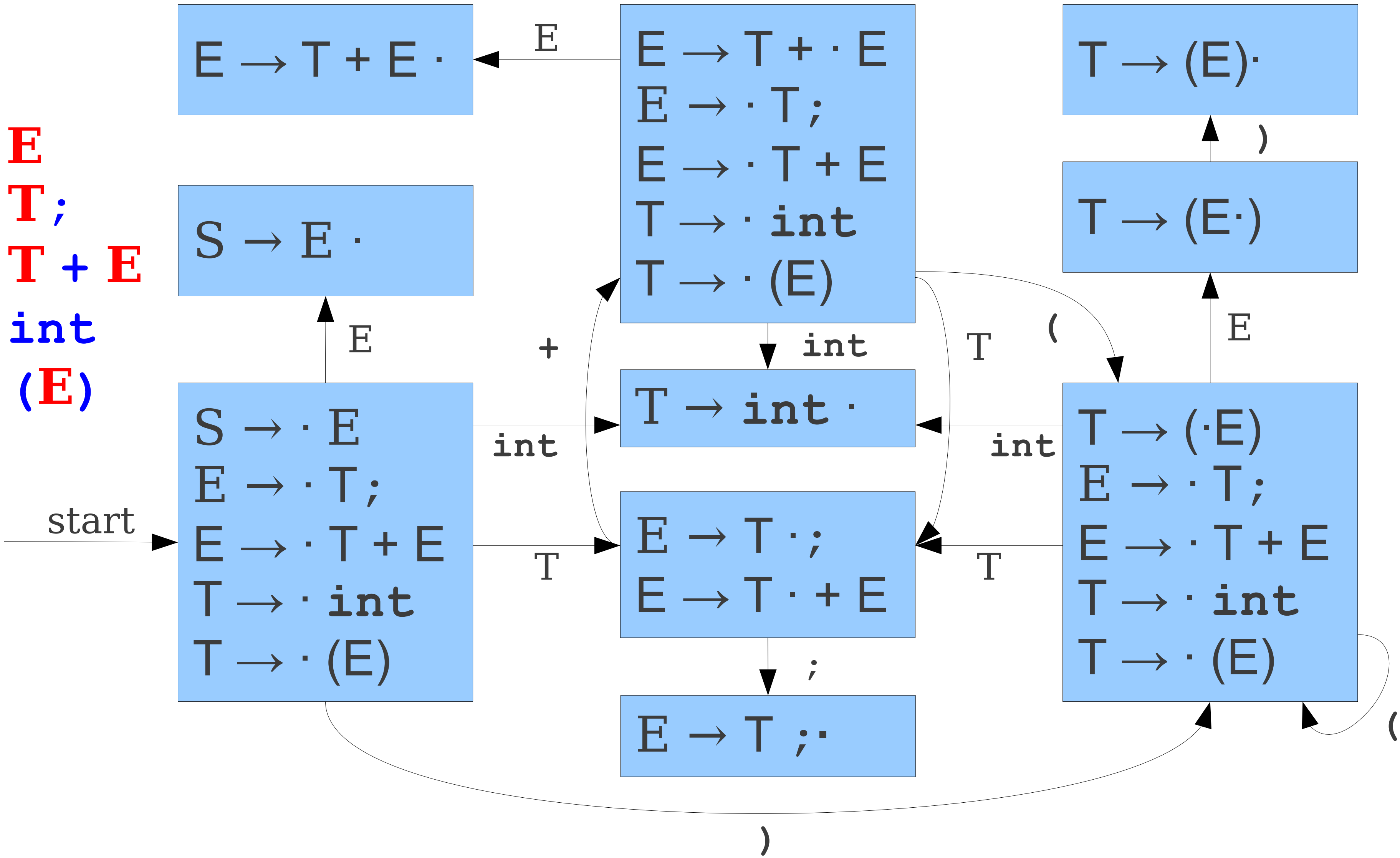
S → **E**
E → **T** `;`
E → **T** `+` **E**
T → `int`
T → `(` **E** `)`



int	+	(	int	+	int	;	)	;

LR(0) Parsing

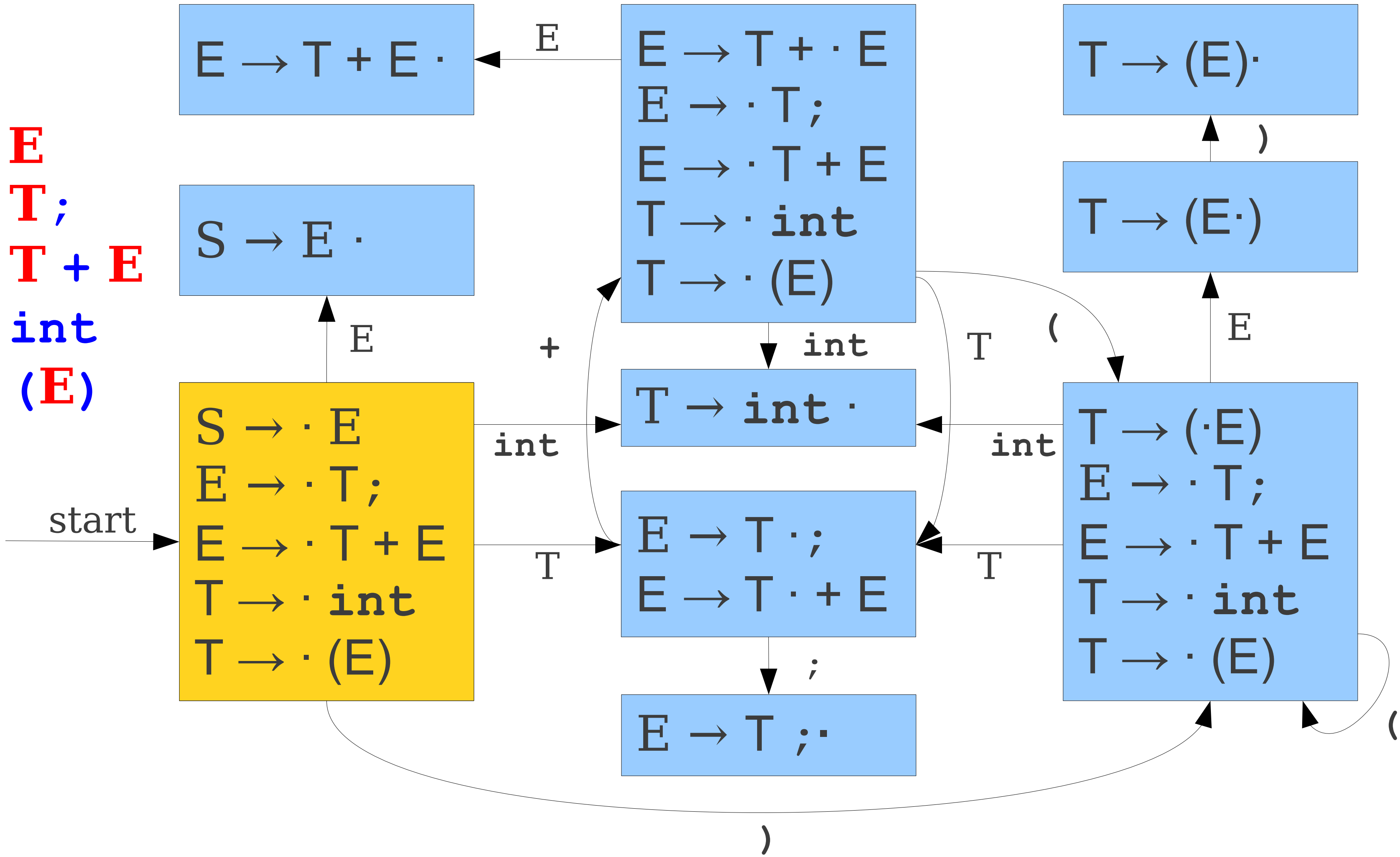
S → E
E → T;
E → T + E
T → int
T → (E)



int	+	(	int	+	int	;	)	;

LR(0) Parsing

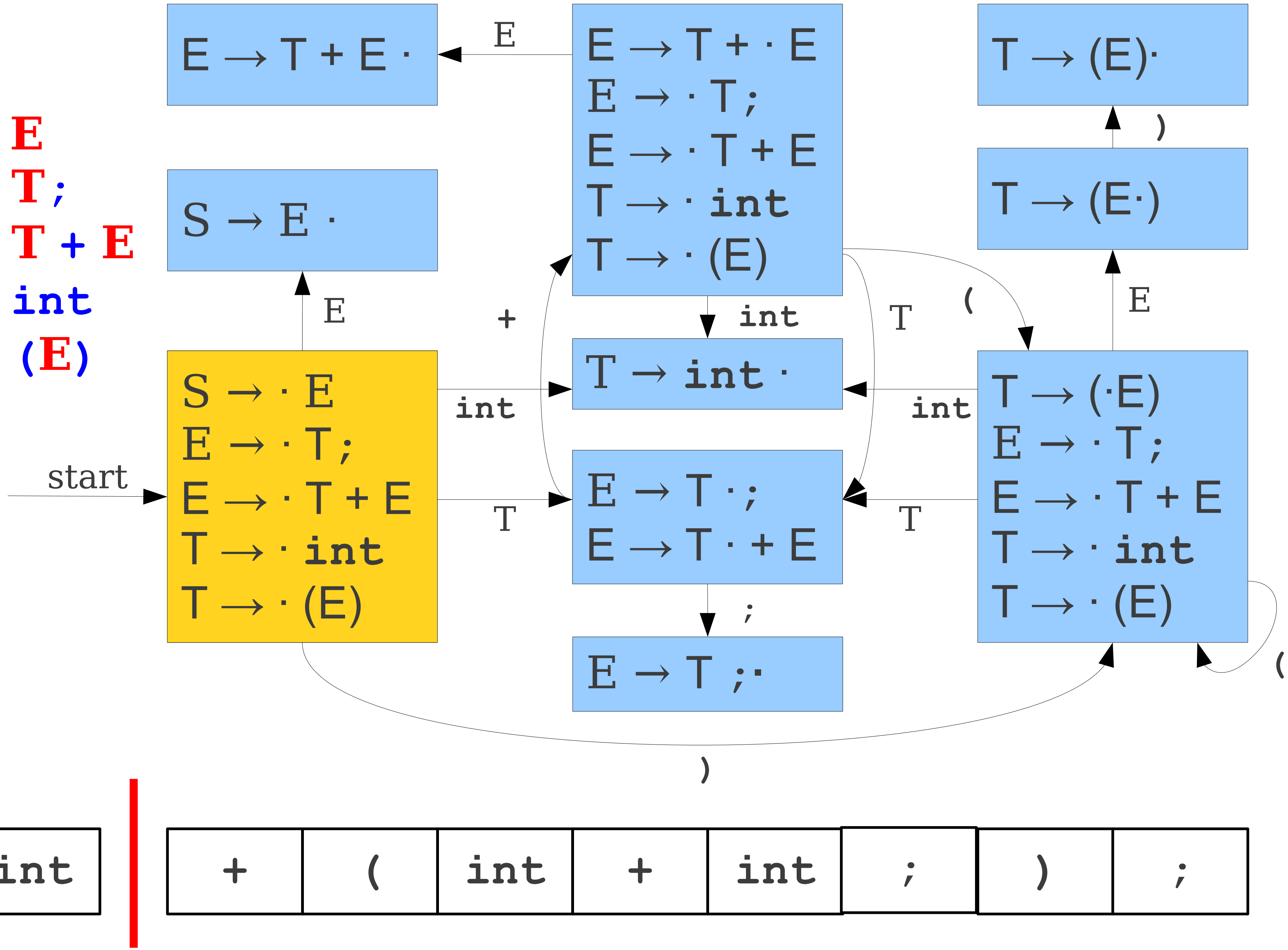
S → **E**
E → **T** `;`
E → **T** + **E**
T → `int`
T → `(`**E**`)`



`int`	`+`	`(`	`int`	`+`	`int`	`;`	`)`	`;`

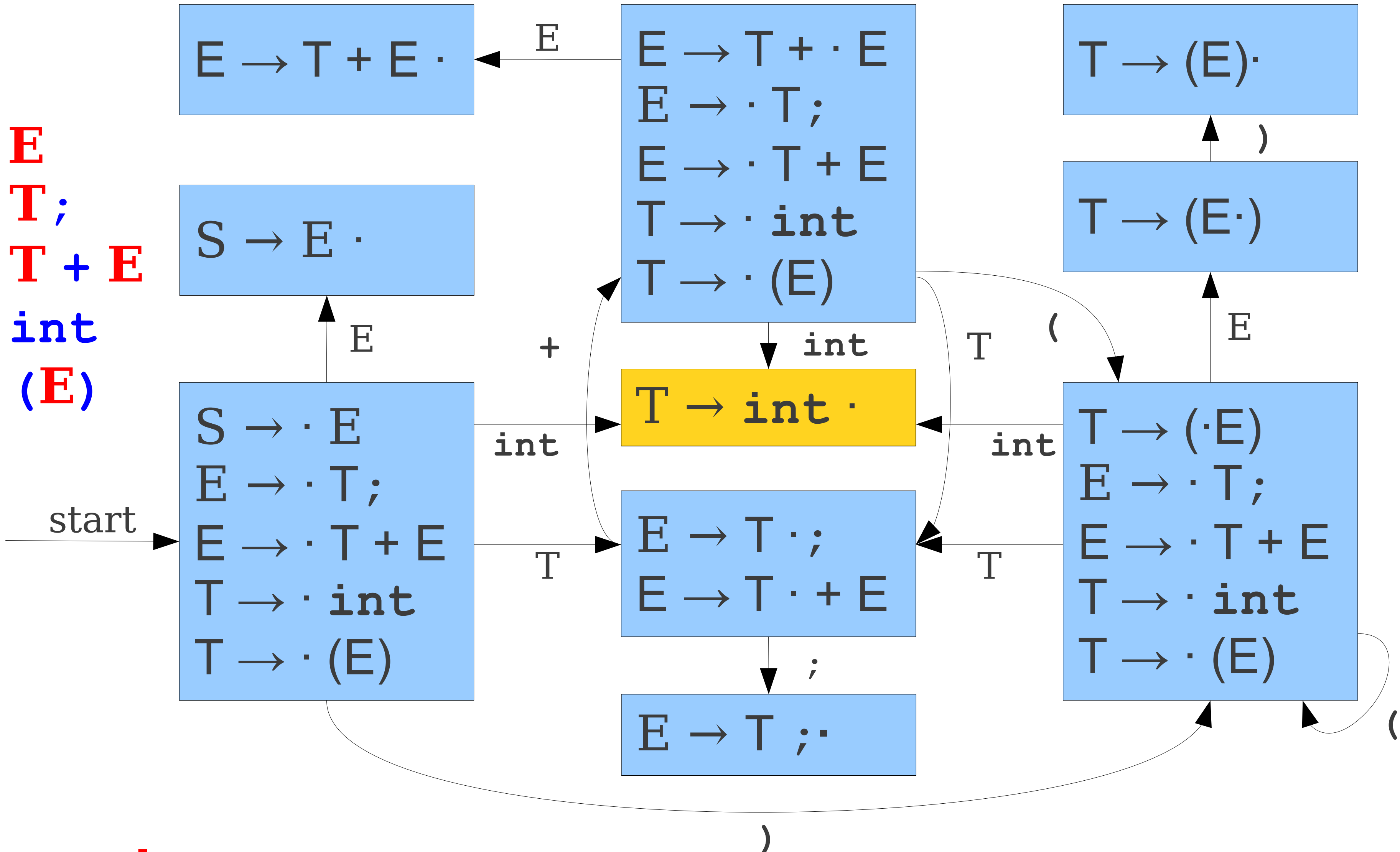
LR(0) Parsing

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



LR(0) Parsing

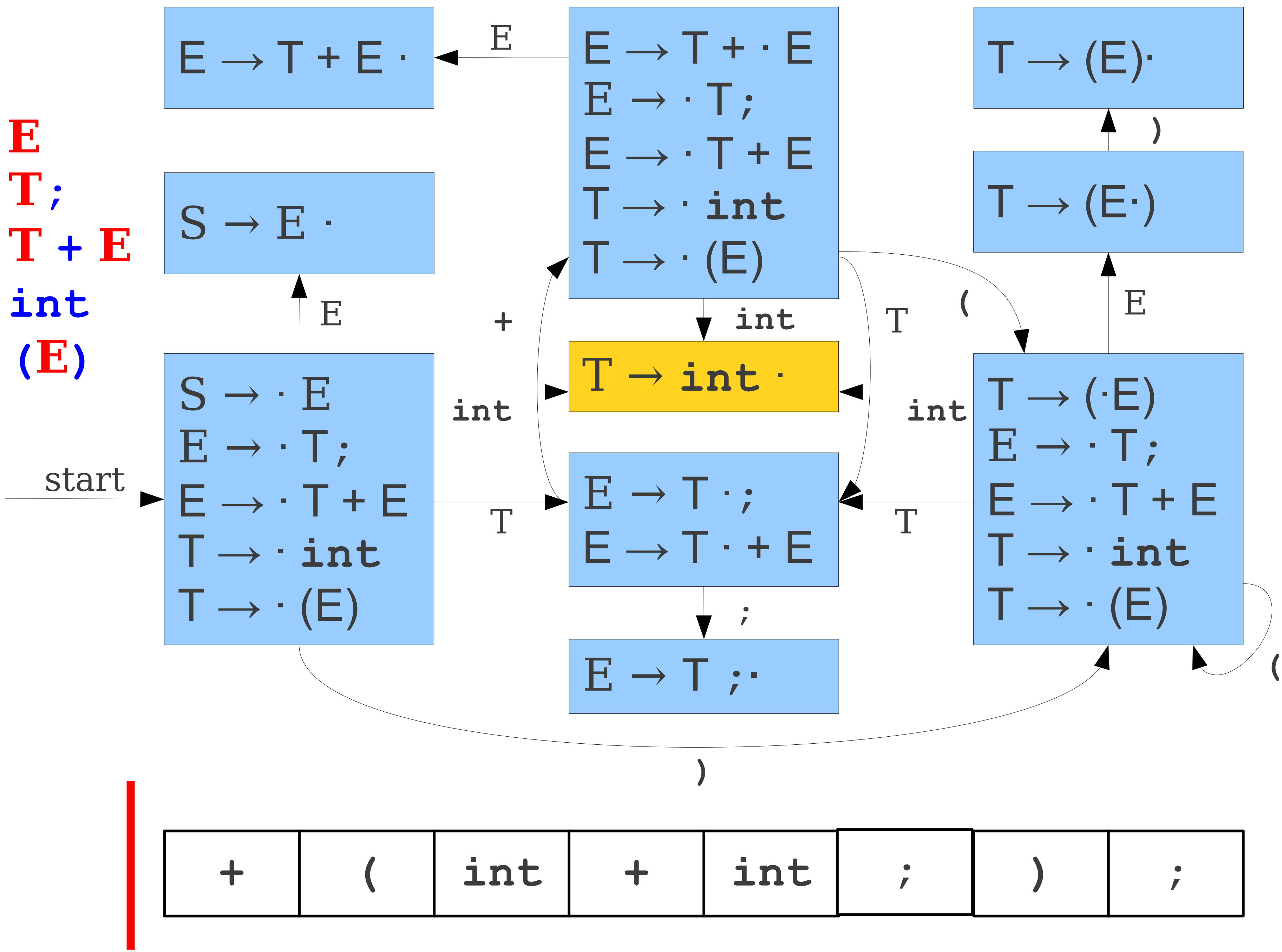
S → **E**
E → **T** `;`
E → **T** `+` **E**
T → `int`
T → `(`**E**`)`



int	|	+	(	int	+	int	;	)	;

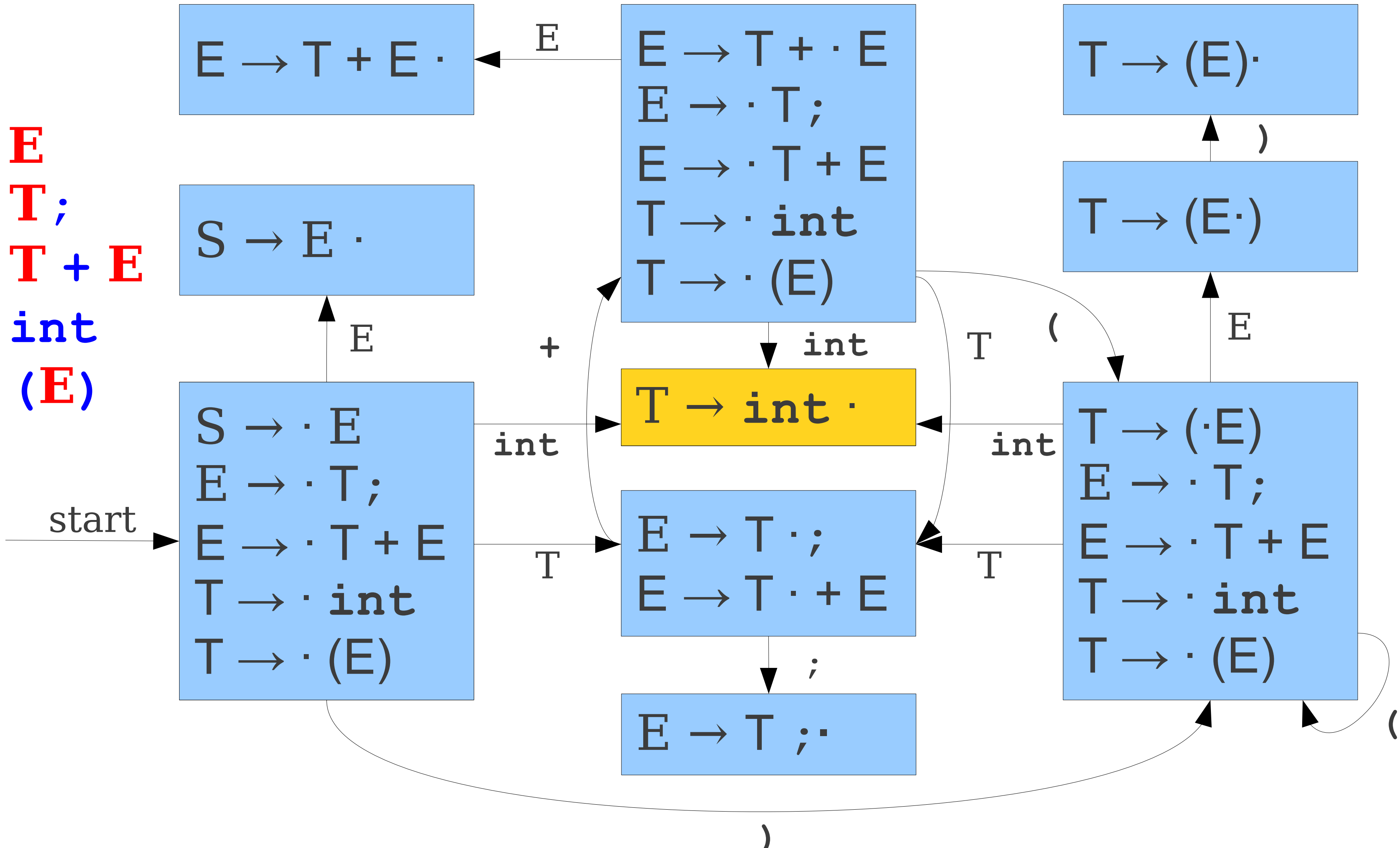
LR(0) Parsing

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



LR(0) Parsing

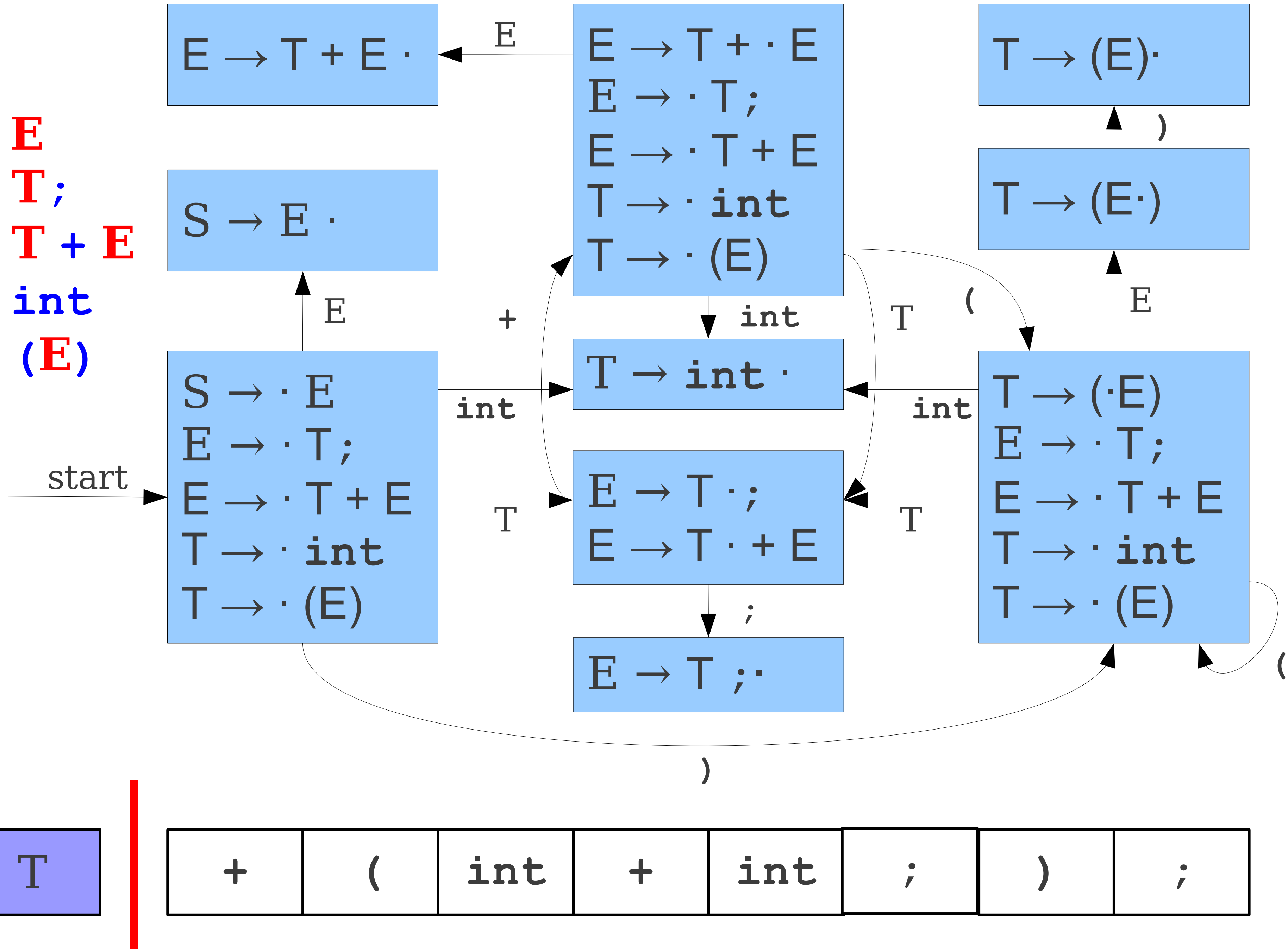
S → **E**
E → **T** `;`
E → **T** + **E**
T → `int`
T → `(`**E**`)`



T | + (int + int ;) ;

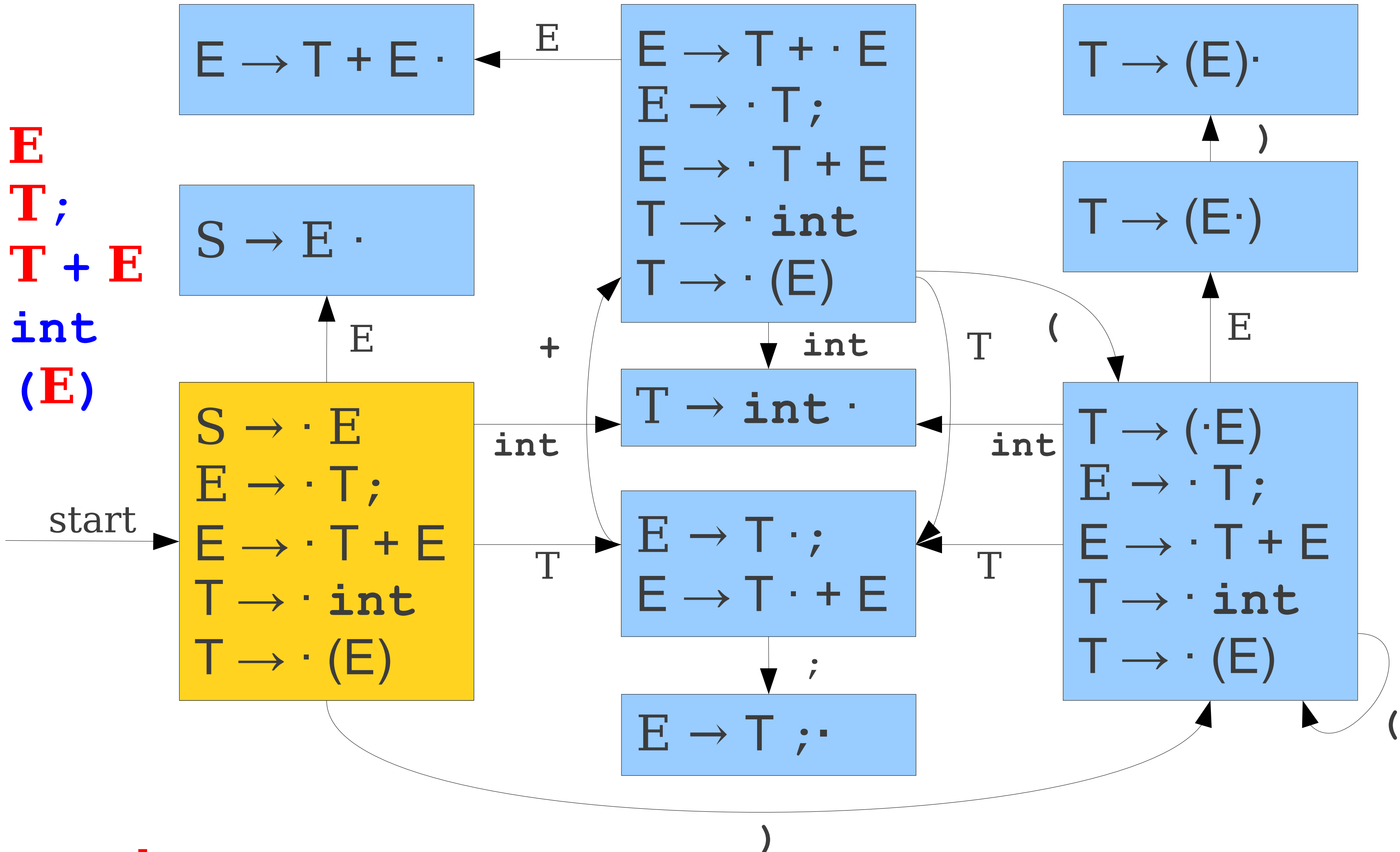
LR(0) Parsing

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



LR(0) Parsing

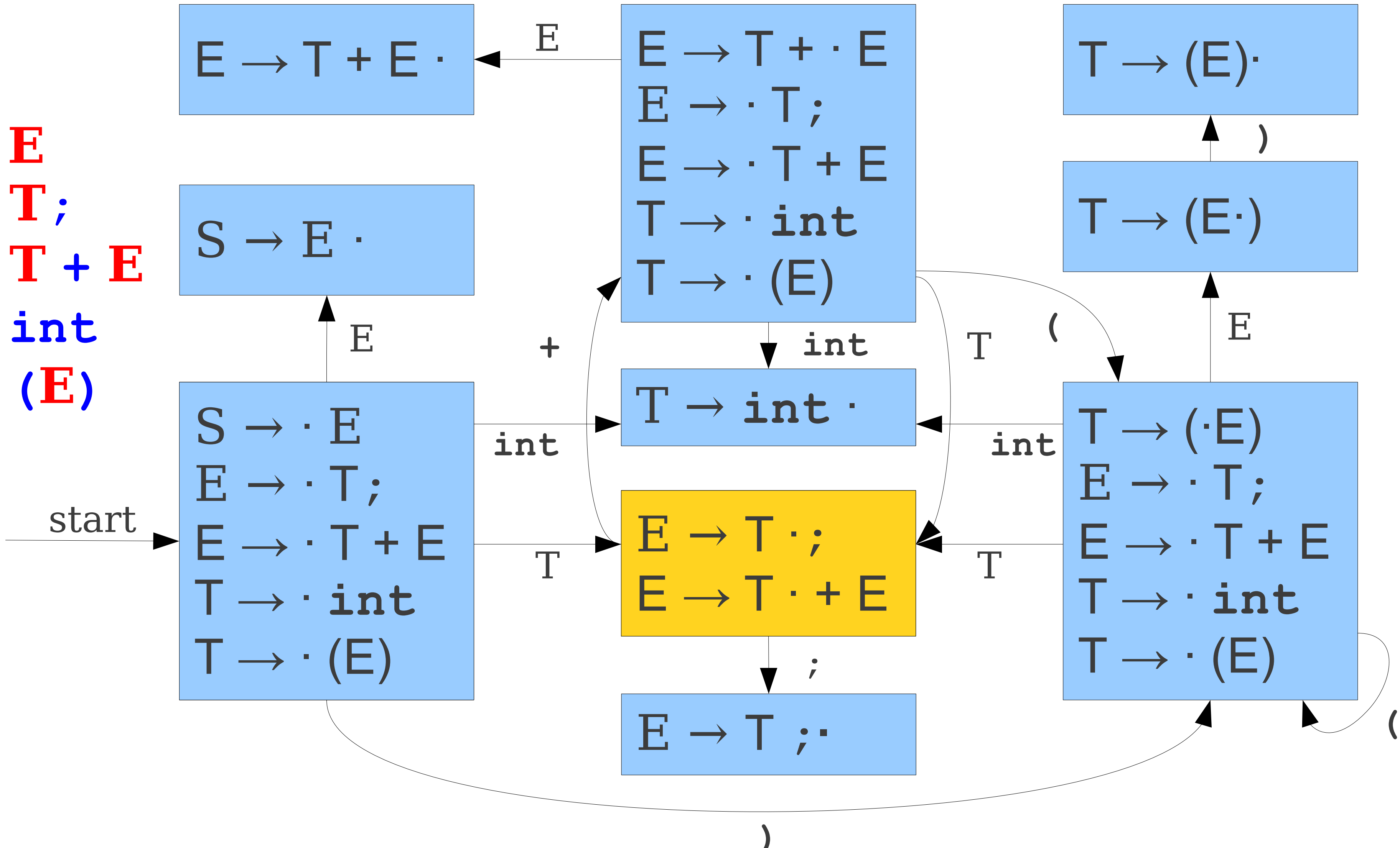
S → **E**
E → **T** ;
E → **T** + **E**
T → int
T → (**E**)

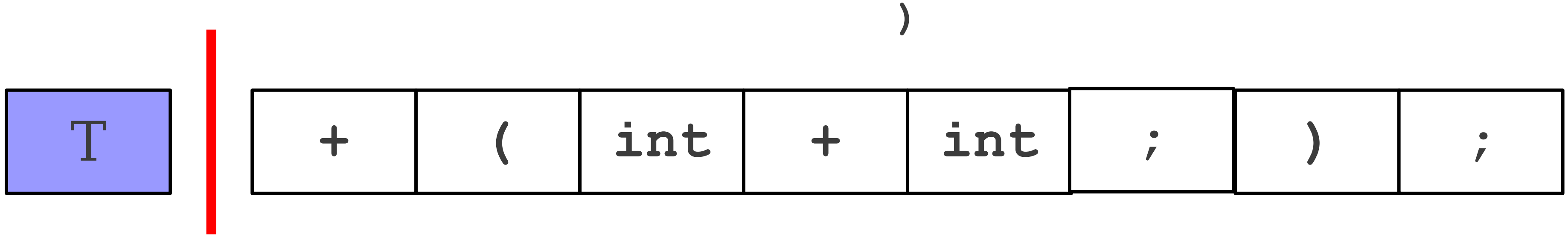


T		+	(	int	+	int	;	)	;

LR(0) Parsing

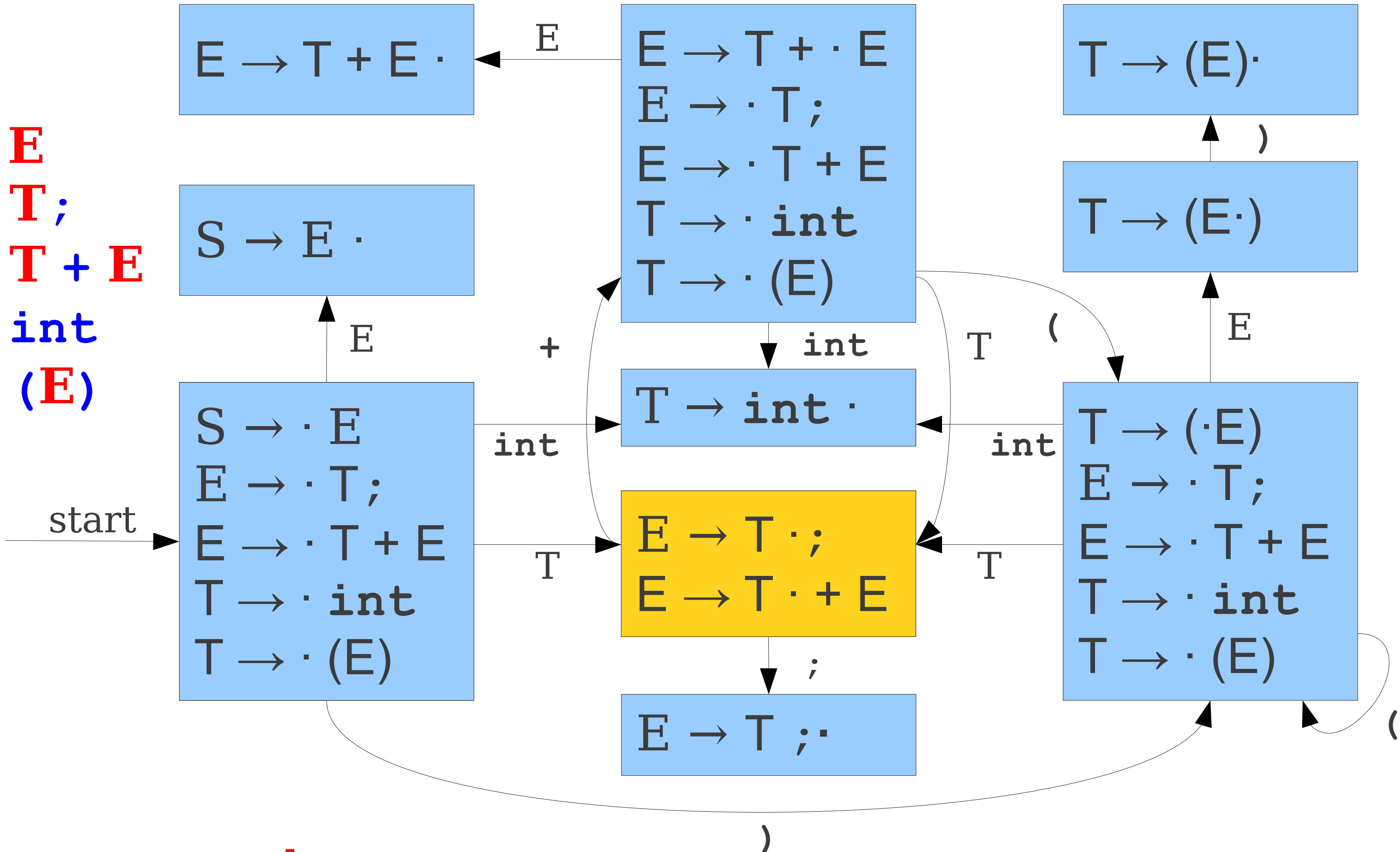
S → **E**
E → **T** `;`
E → **T** `+` **E**
T → `int`
T → `(`**E**`)`





LR(0) Parsing

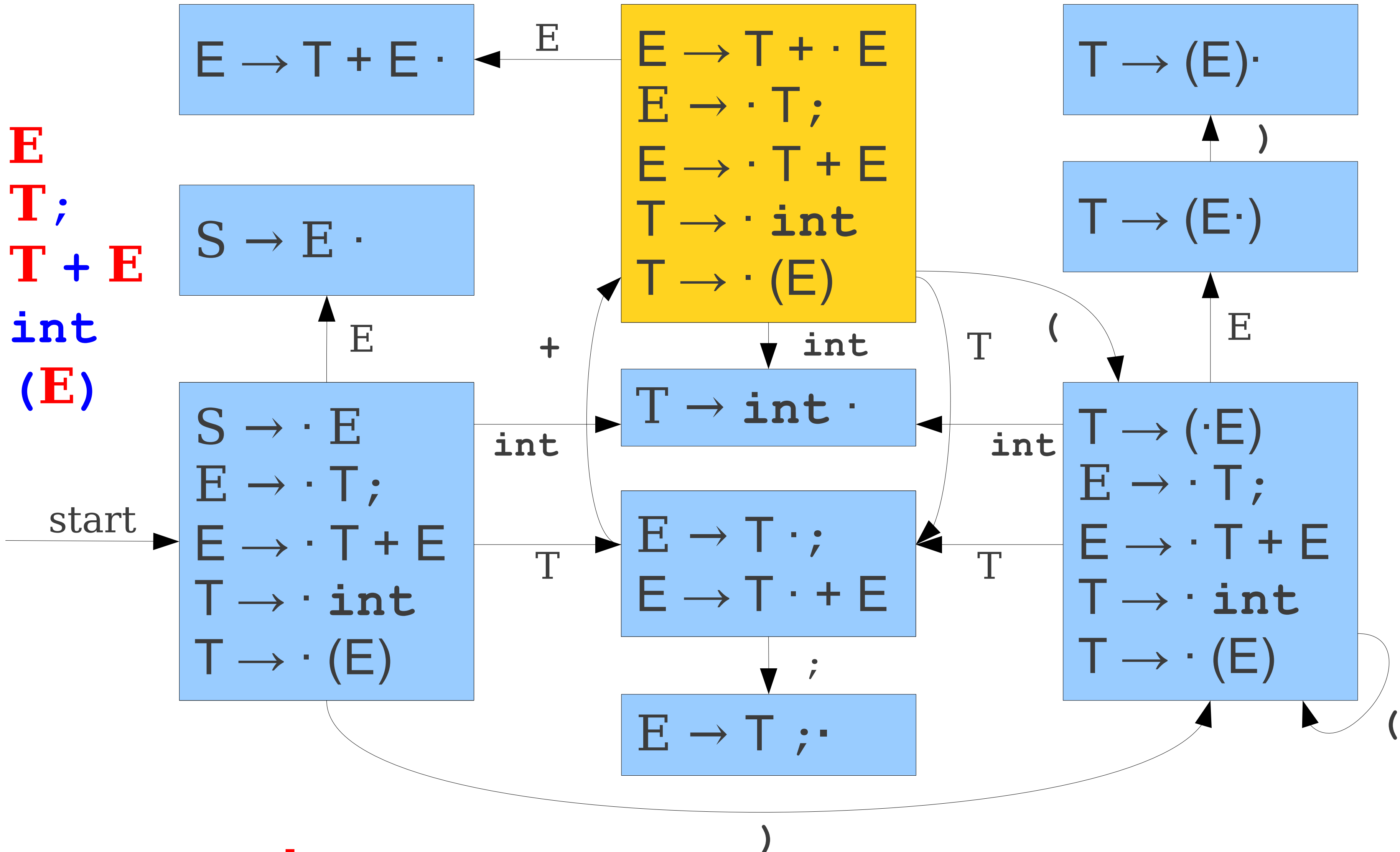
S → **E**
E → **T** `;`
E → **T** `+` **E**
T → `int`
T → `(`**E**`)`

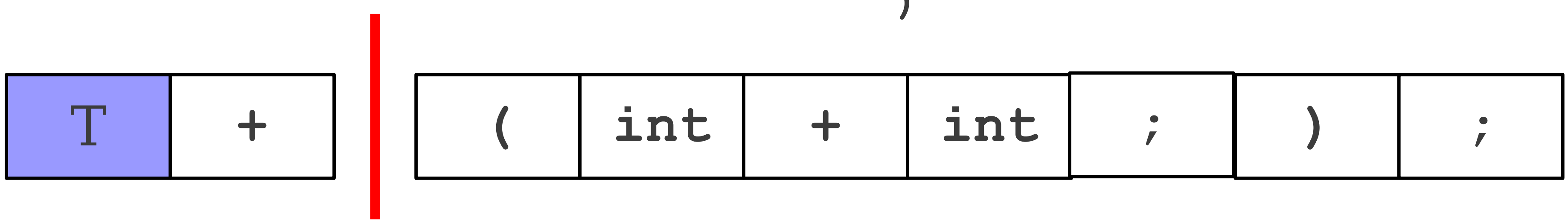


T	+		(	int	+	int	;	)	;

LR(0) Parsing

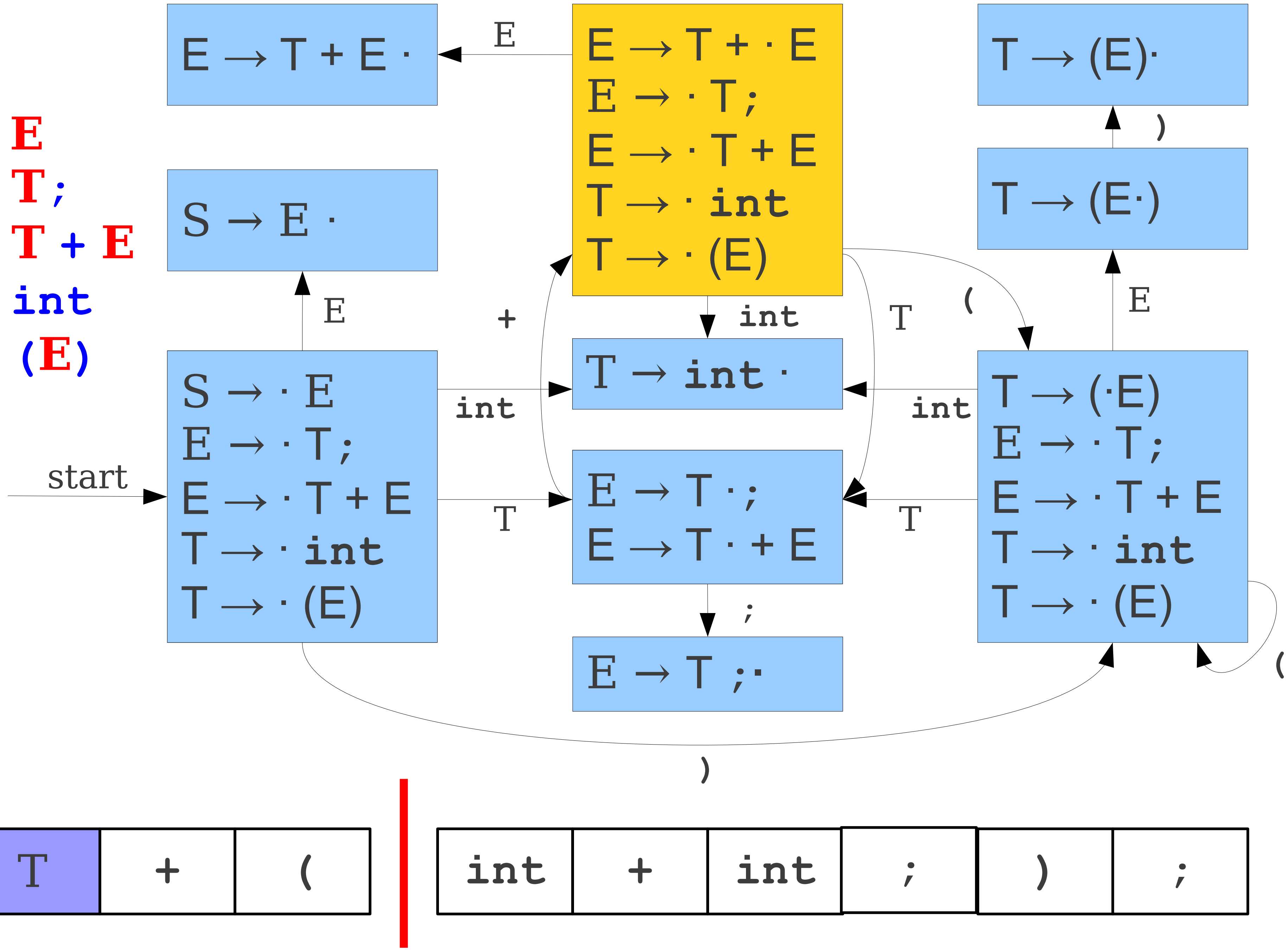
S → **E**
E → **T**;
E → **T** + **E**
T → **int**
T → (**E**)





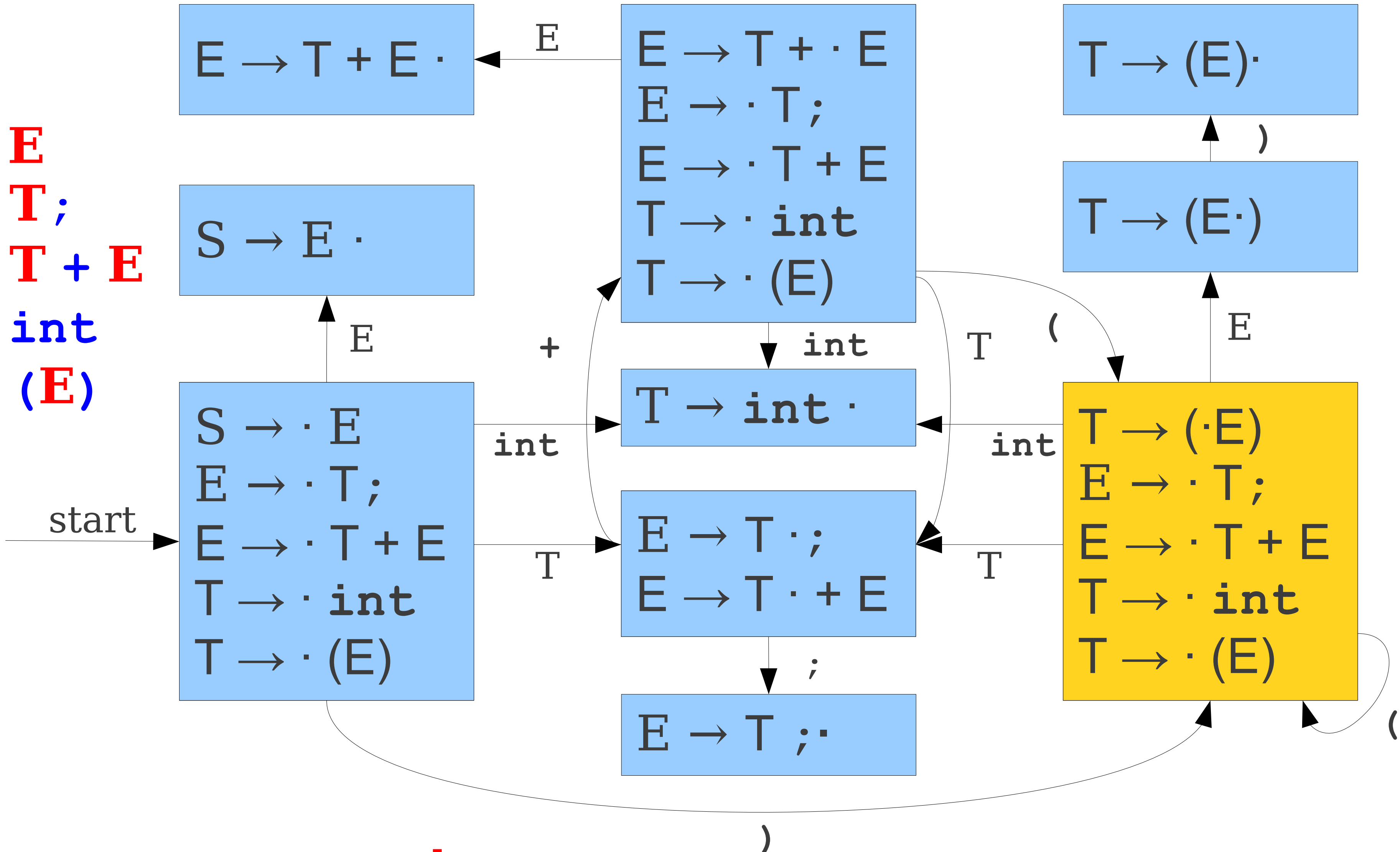
LR(0) Parsing

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



LR(0) Parsing

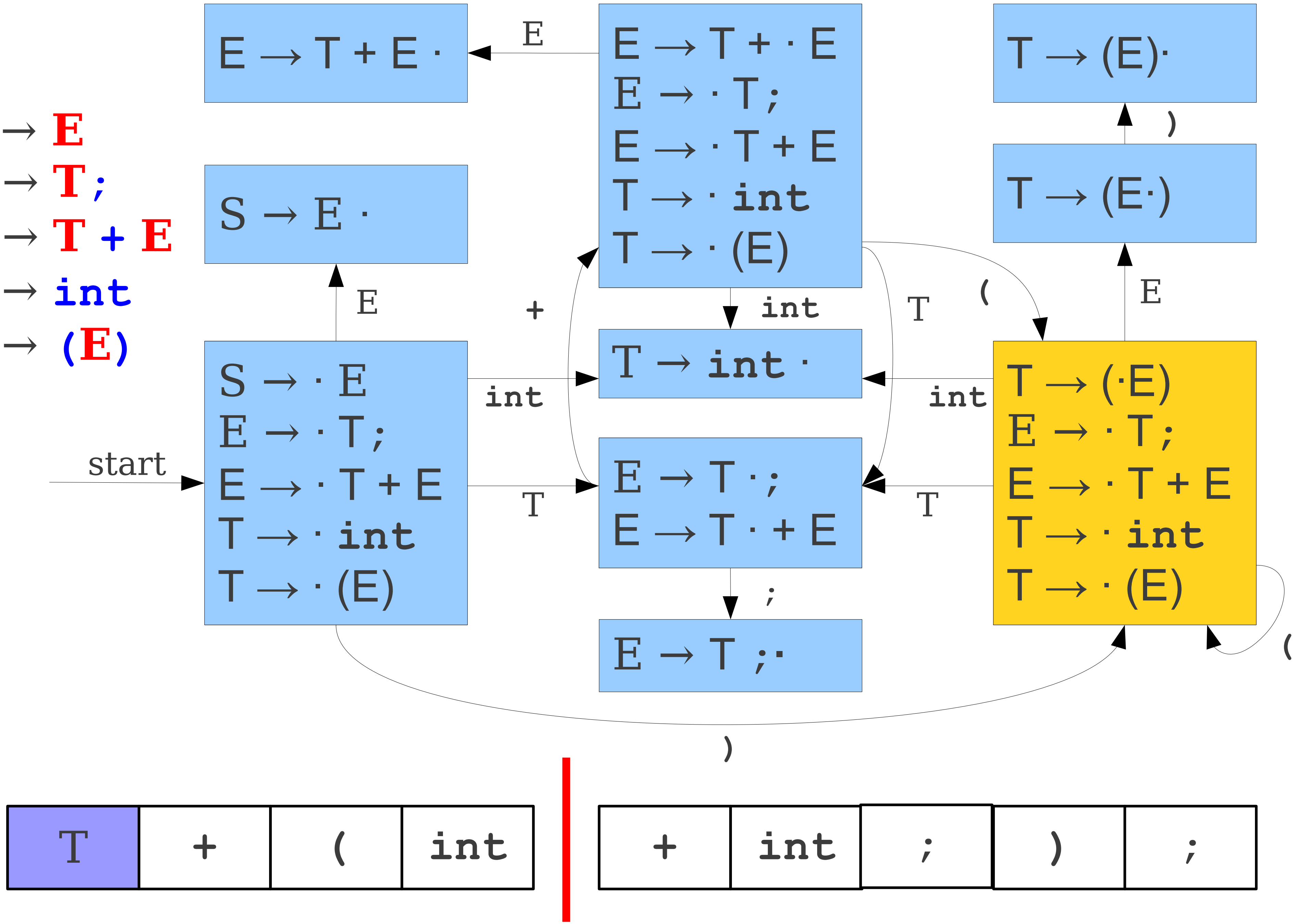
S → **E**
E → **T** `;`
E → **T** `+` **E**
T → `int`
T → `(`**E**`)`



T	+	(	|	int	+	int	;	)	;

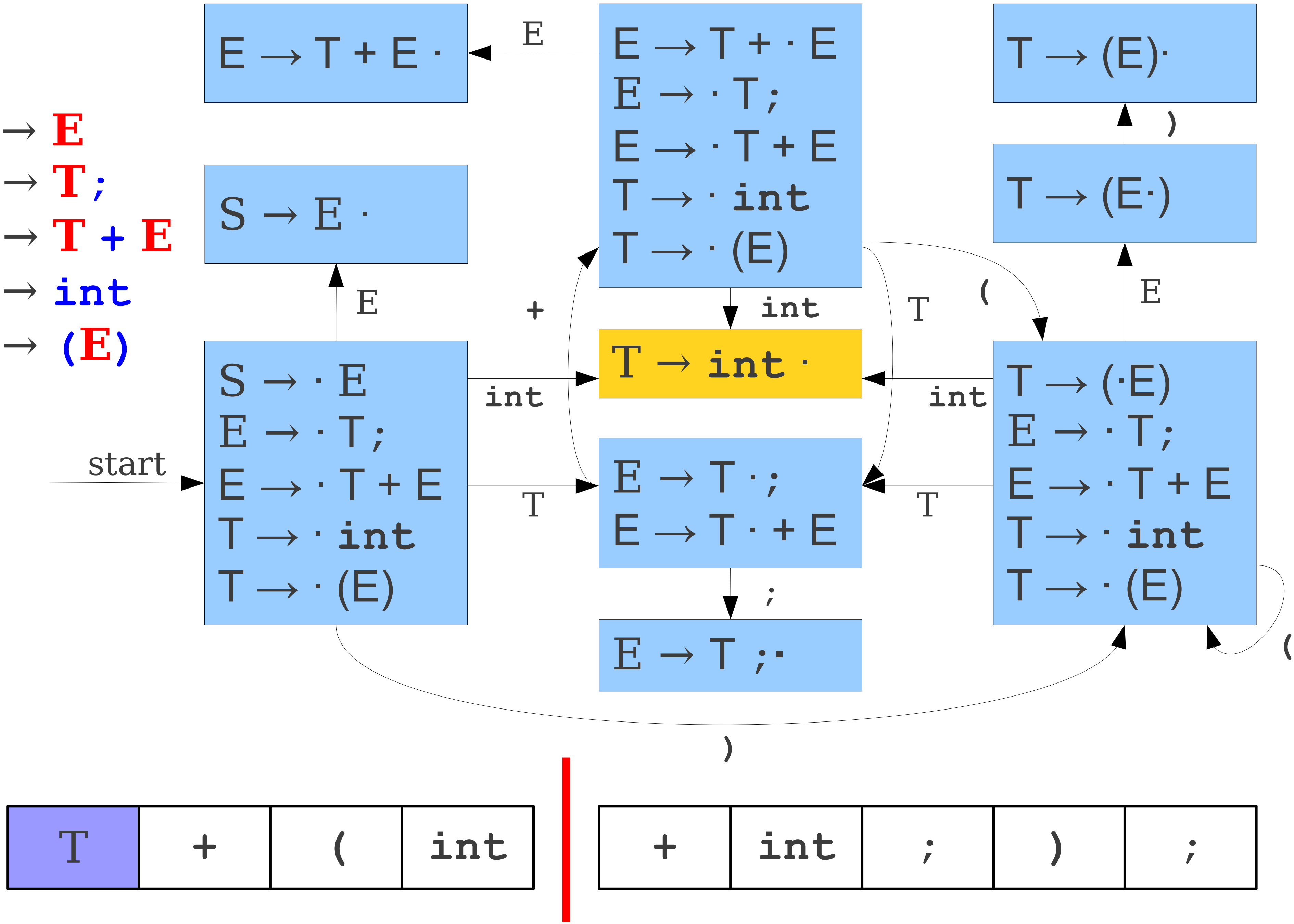
LR(0) Parsing

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



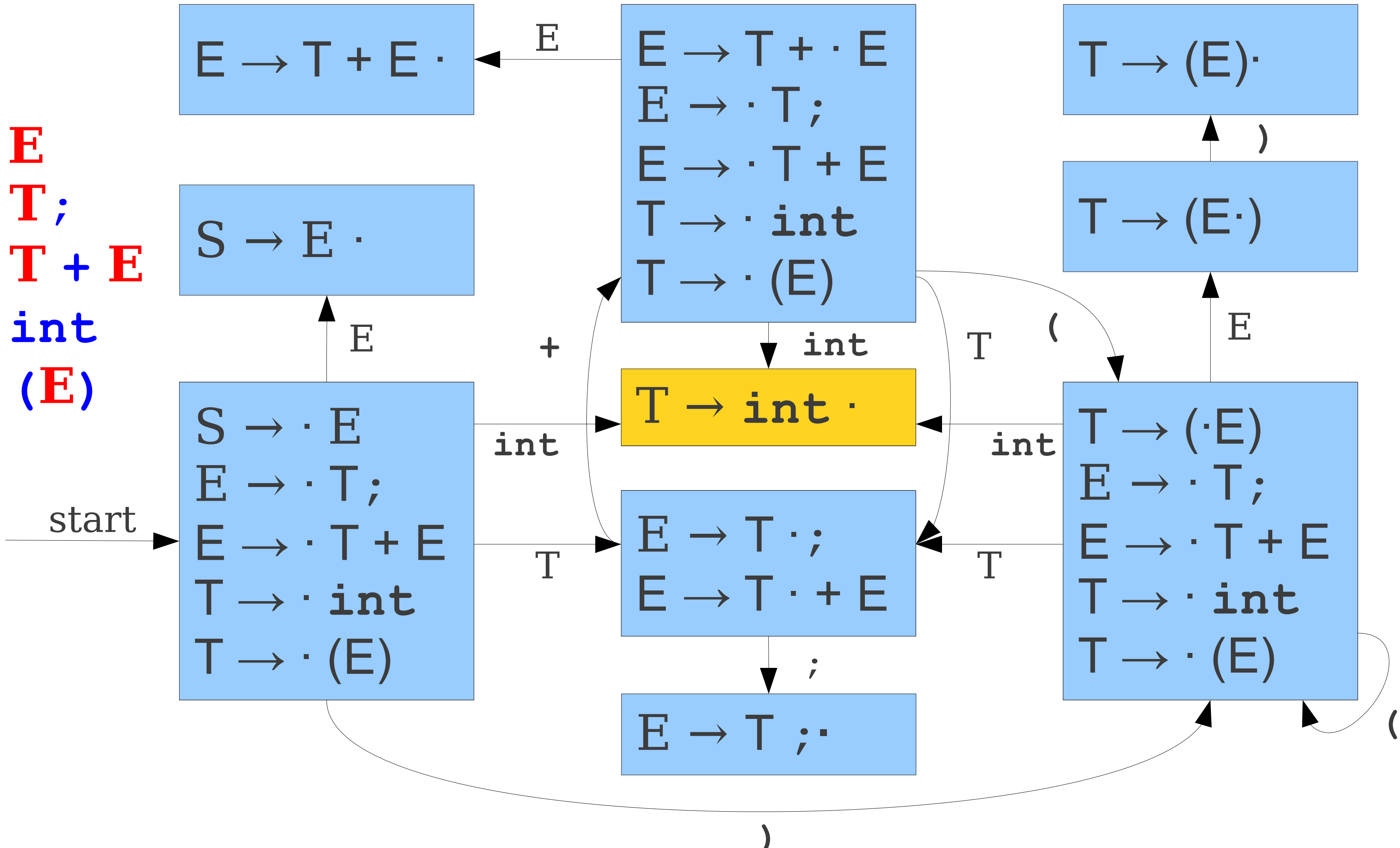
LR(0) Parsing

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



LR(0) Parsing

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

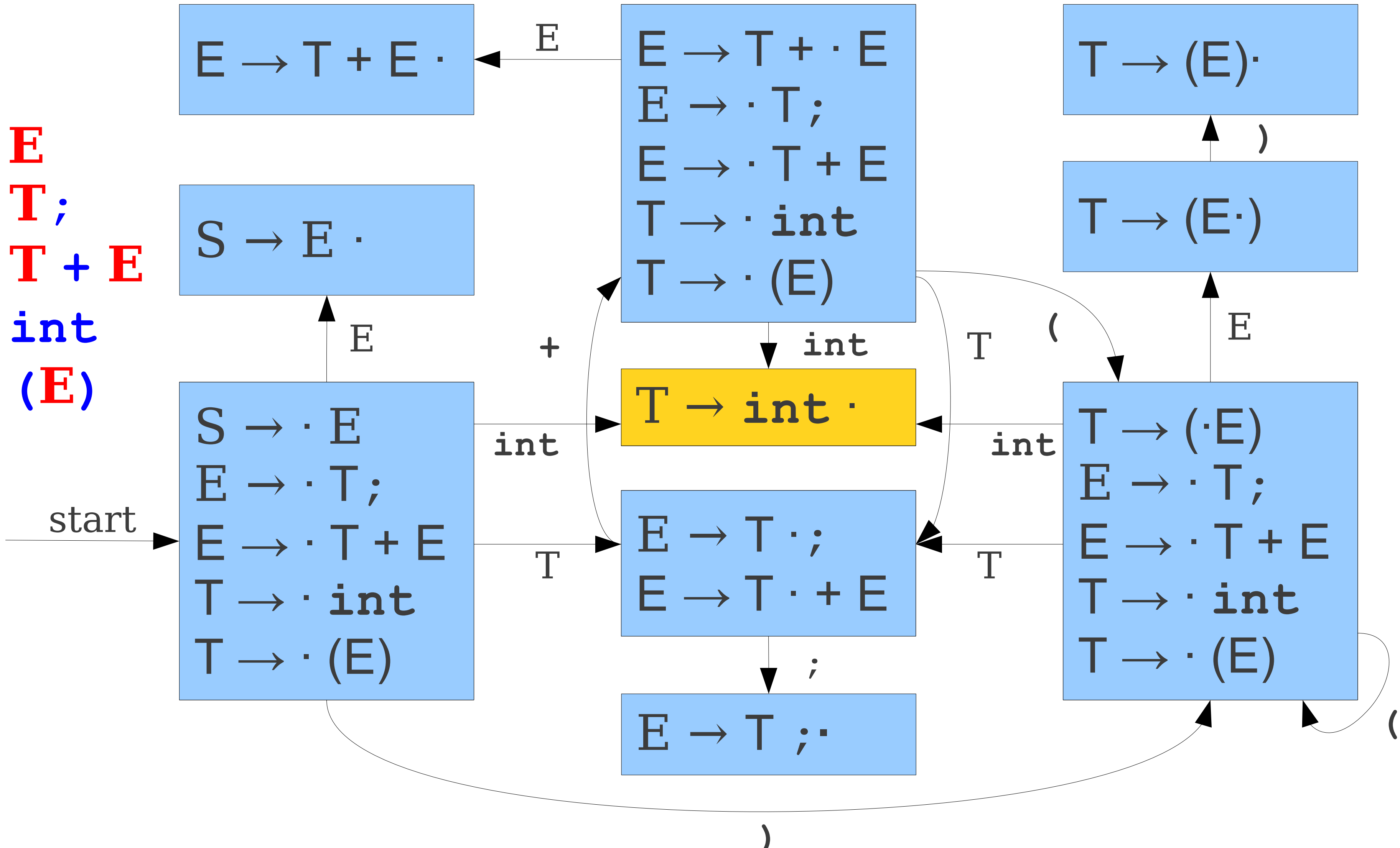


T	+	(

+	int	;	)	;

LR(0) Parsing

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

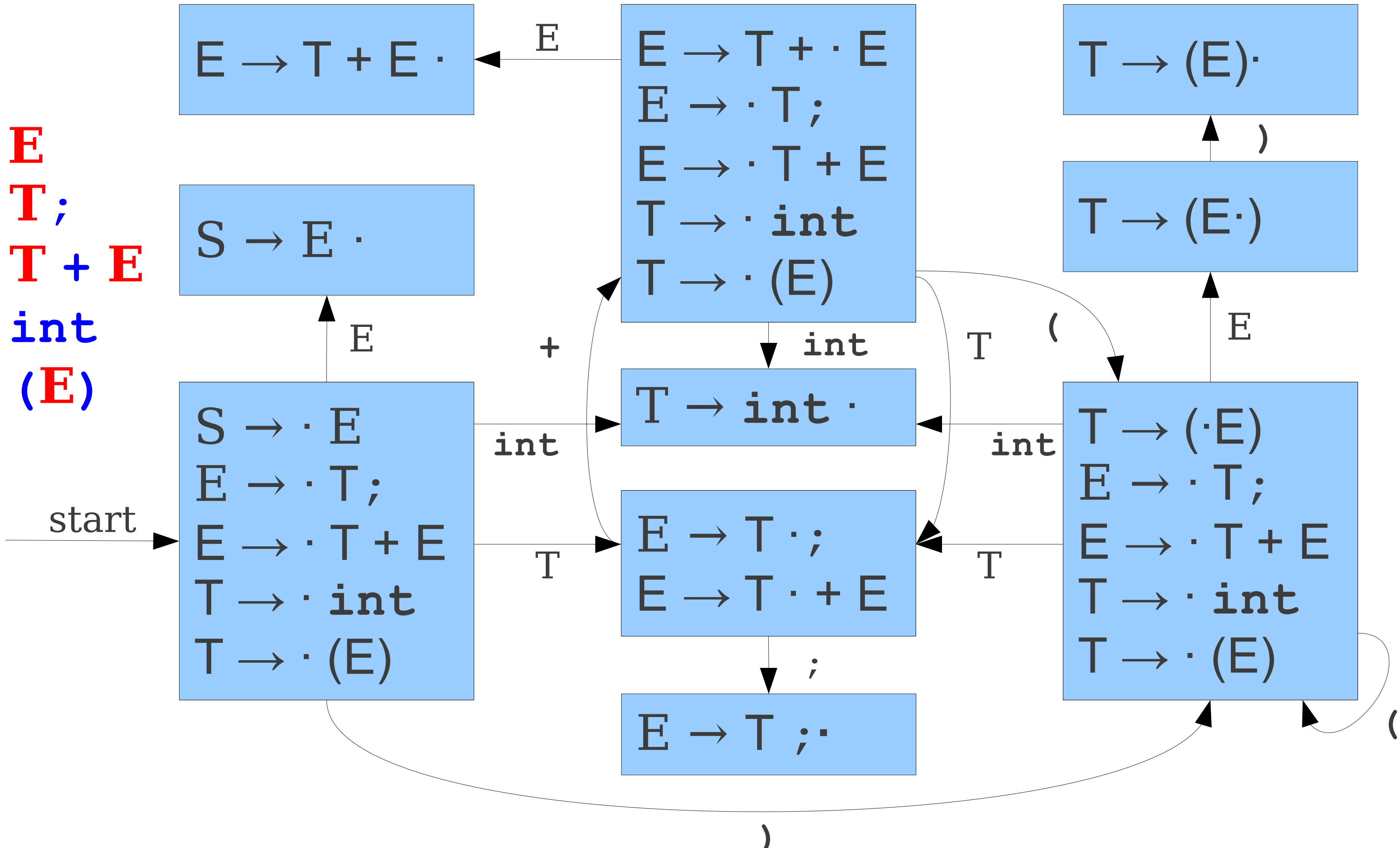


T	+	(	T

+	int	;	)	;

LR(0) Parsing

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

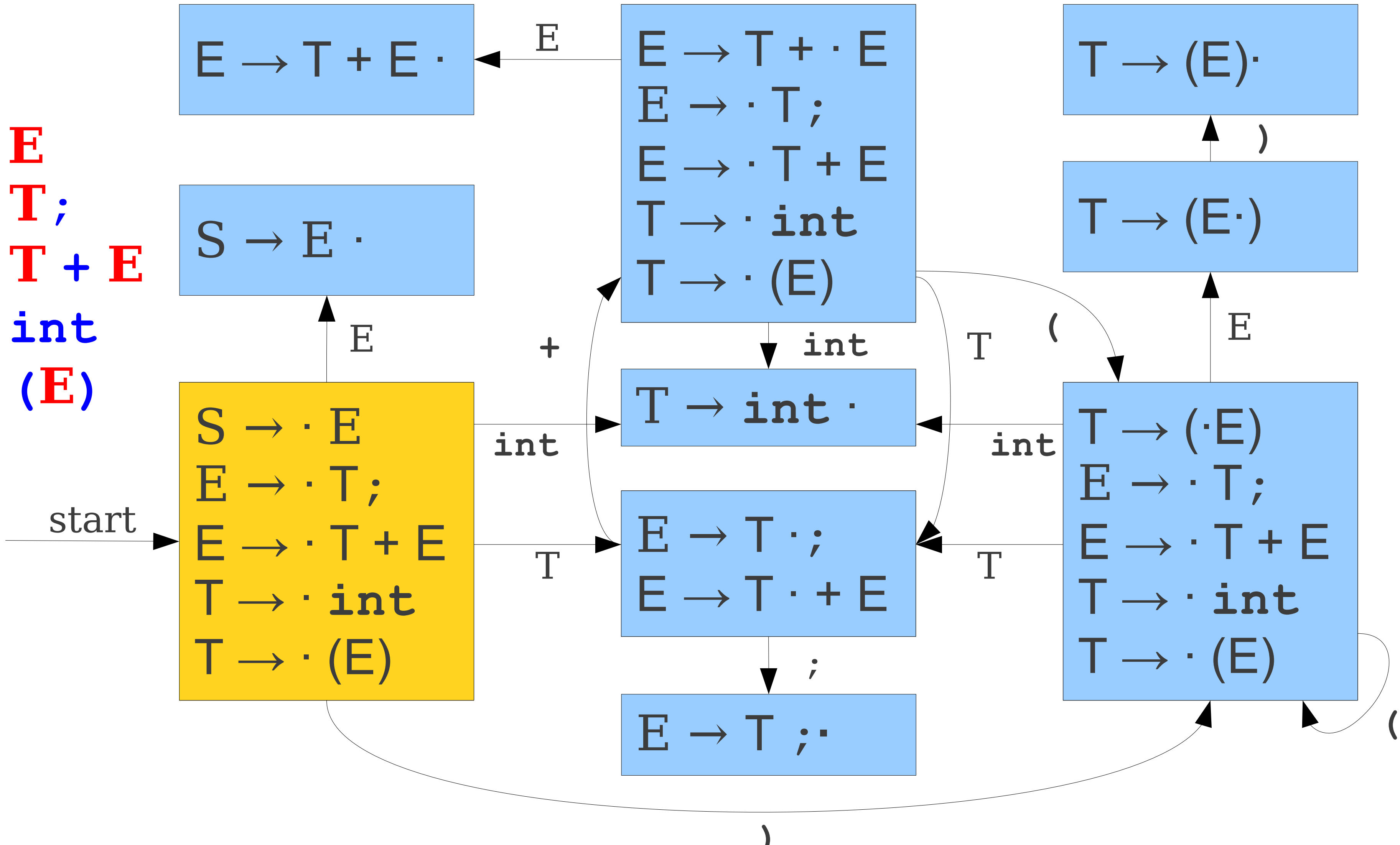


T	+	(	T

+	int	;	)	;

LR(0) Parsing

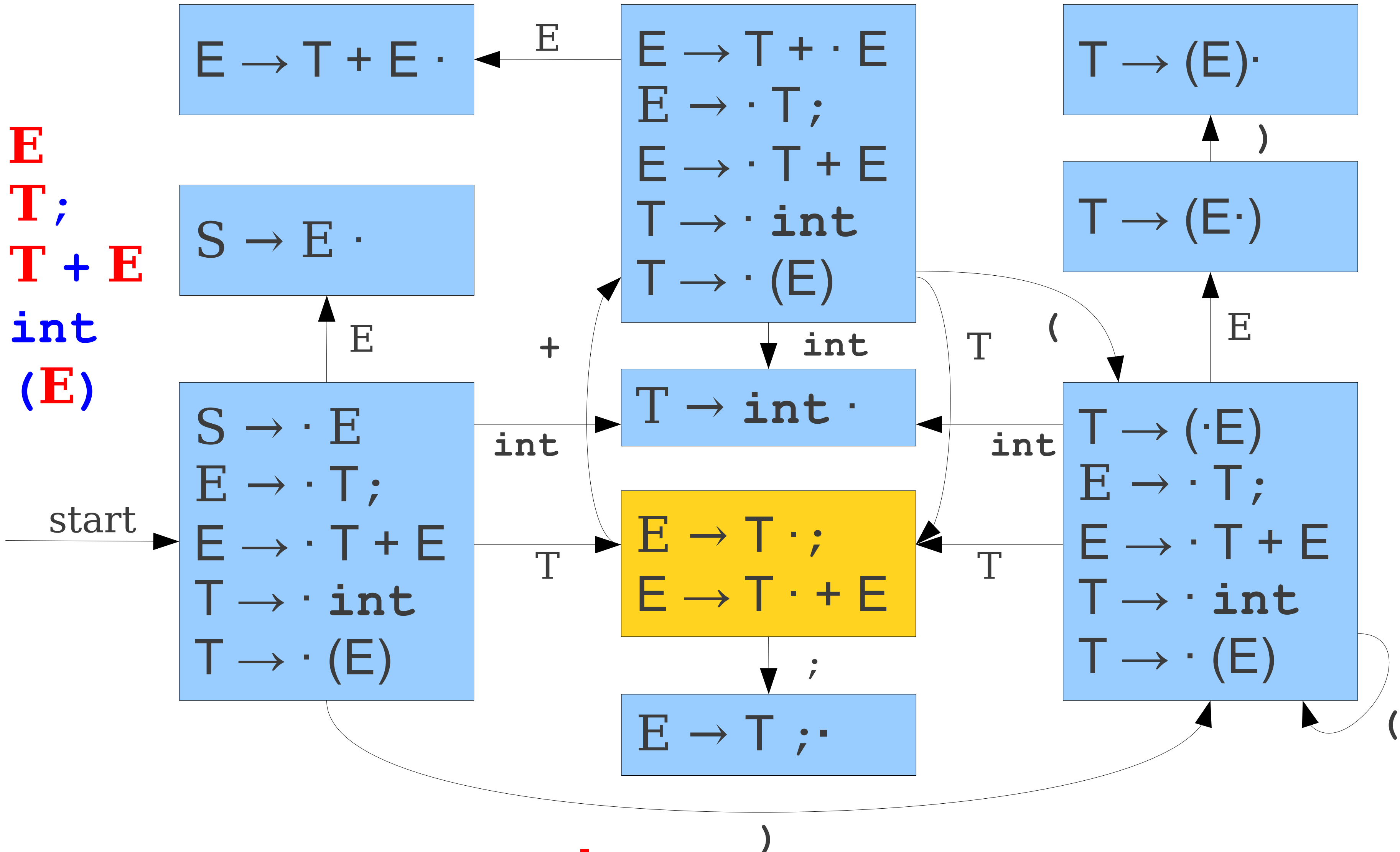
$S \rightarrow E$
$E \rightarrow T;$
$E \rightarrow T + E$
$T \rightarrow$ **int**
$T \rightarrow (E)$



T	+	(	T		+	int	;	)	;

LR(0) Parsing

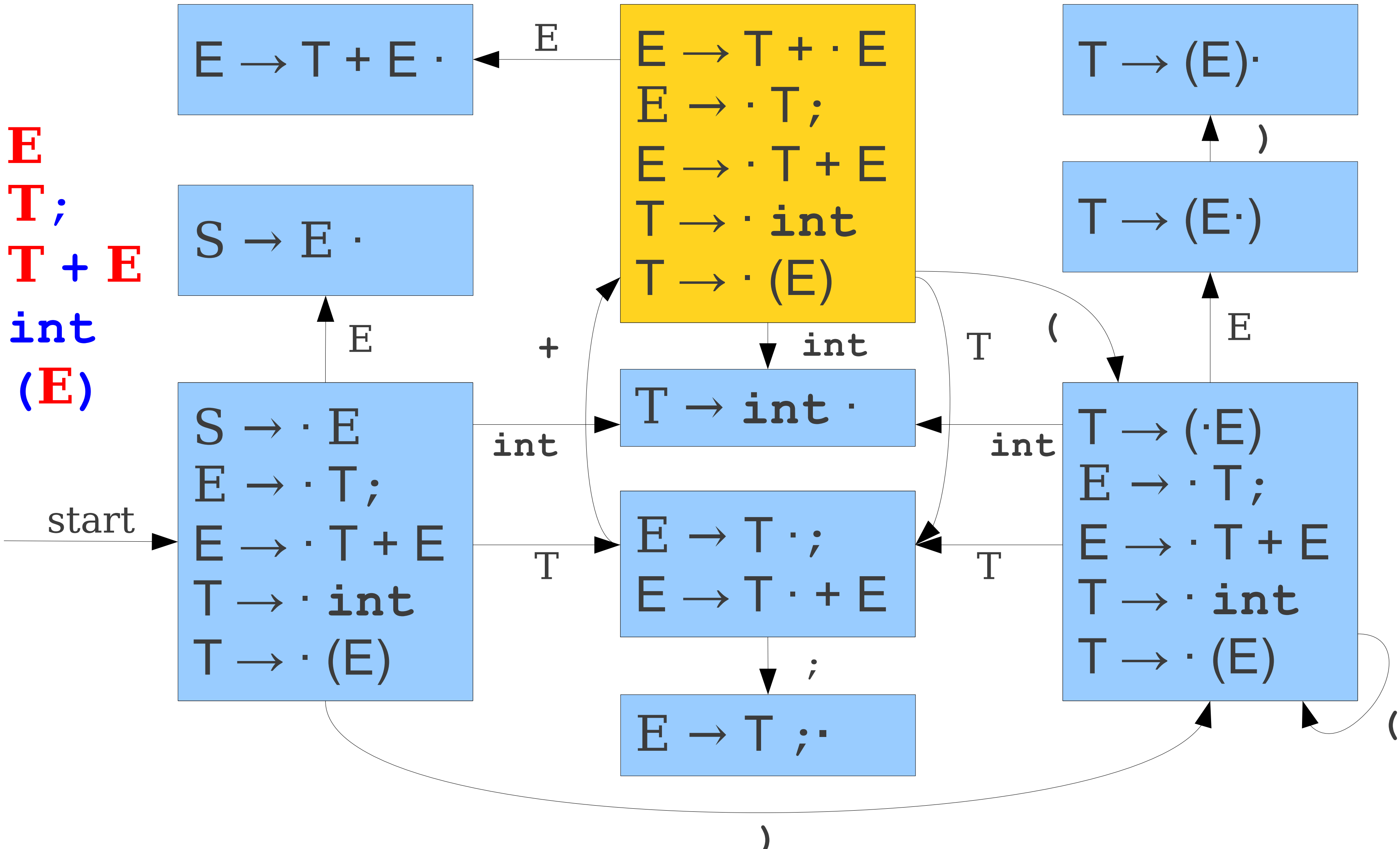
S → **E**
E → **T** `;`
E → **T** `+` **E**
T → `int`
T → `(`**E**`)`

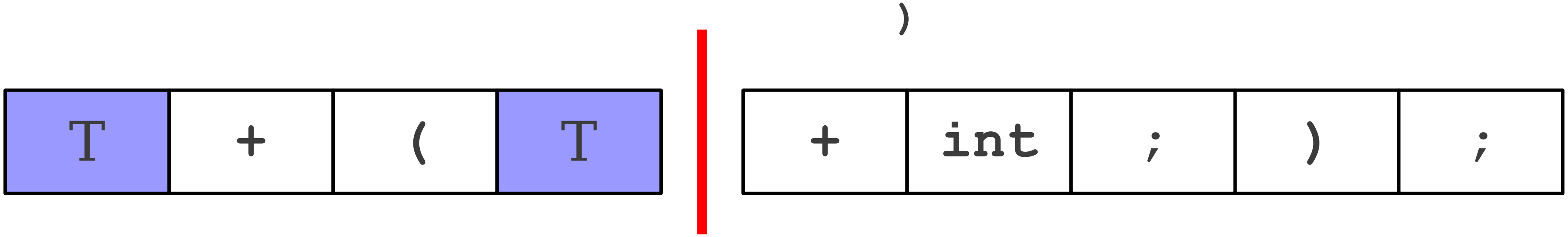


T	+	(	T	+	int	;	)	;

LR(0) Parsing

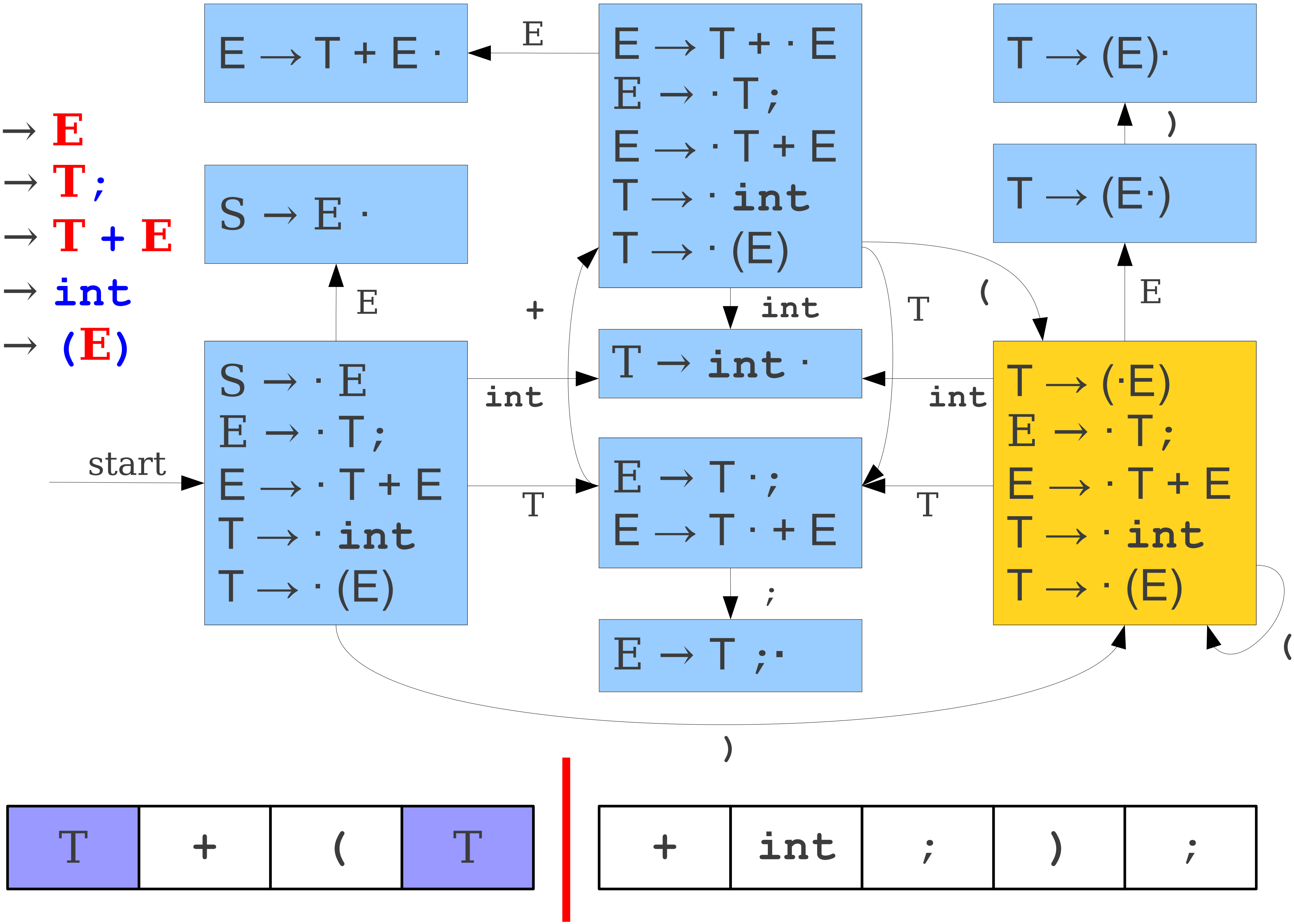
$S \rightarrow E$
$E \rightarrow T;$
$E \rightarrow T + E$
$T \rightarrow$ **int**
$T \rightarrow (E)$





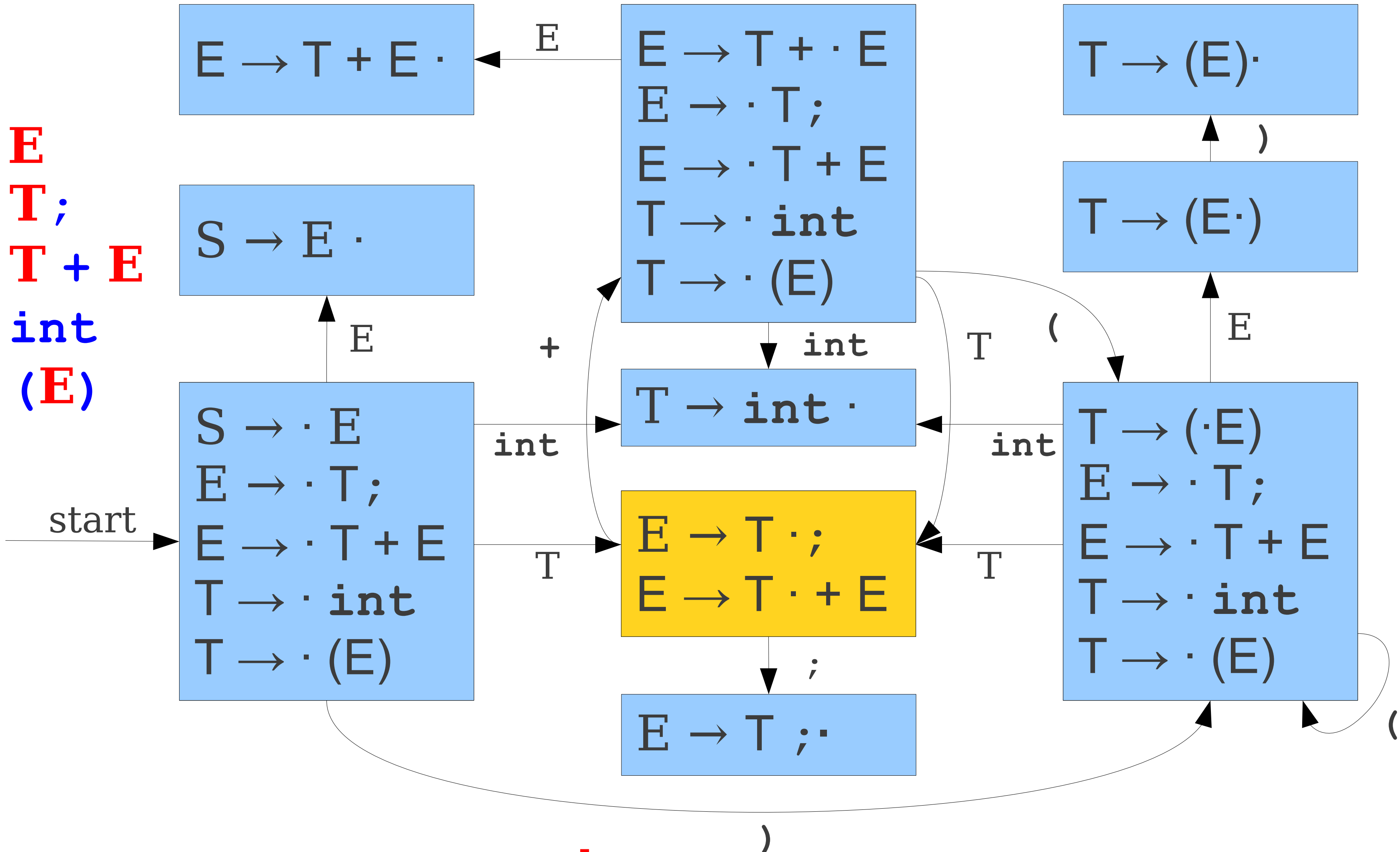
LR(0) Parsing

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



LR(0) Parsing

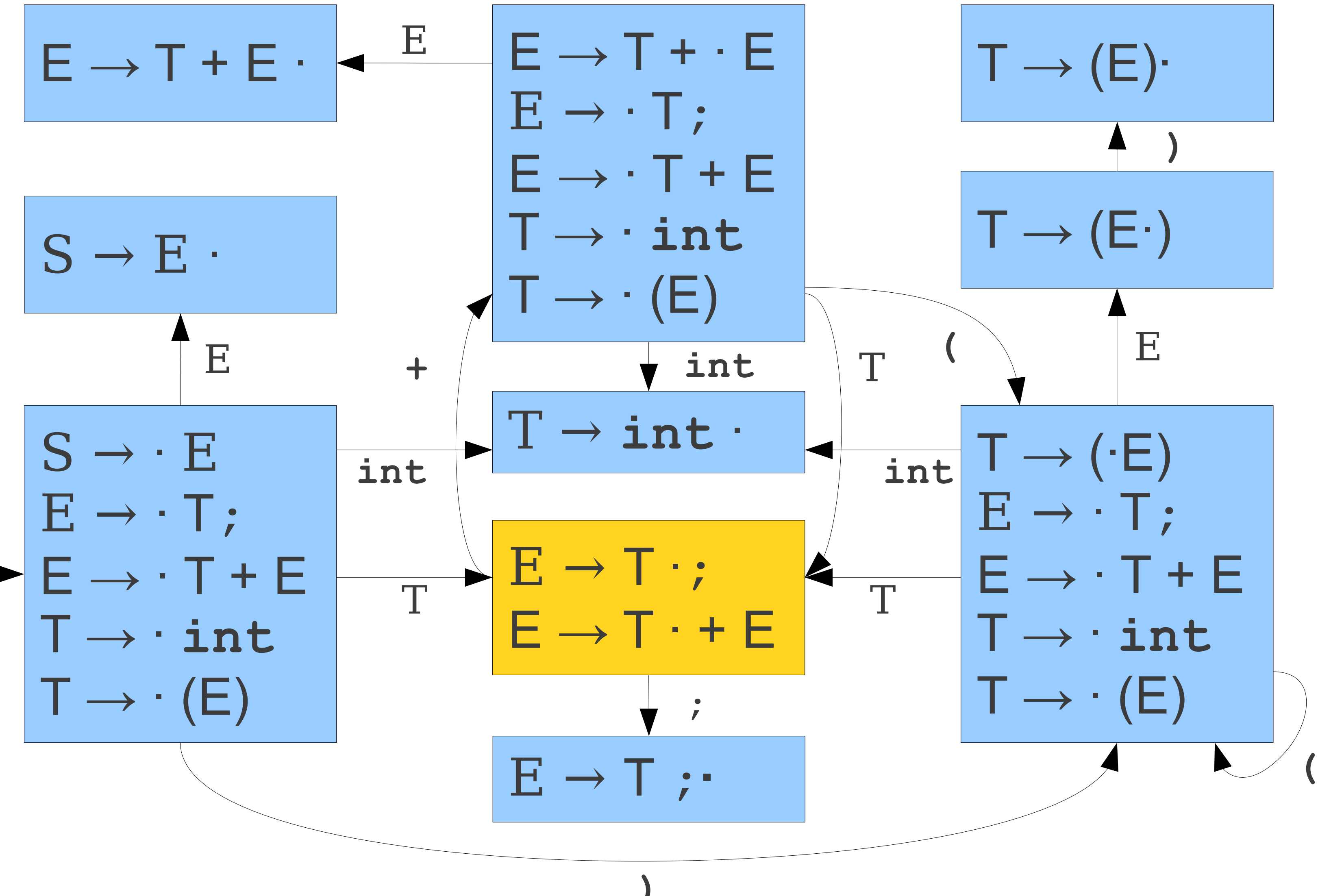
S → **E**
E → **T** ;
E → **T** + **E**
T → `int`
T → (**E**)

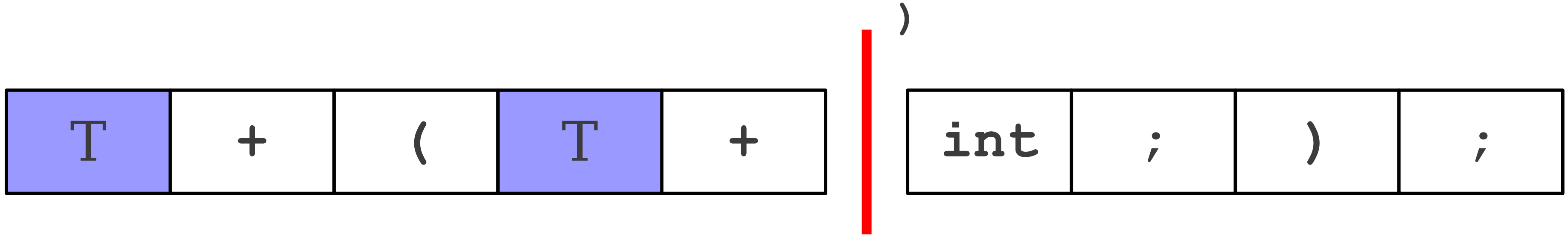


T	+	(	T	|	+	int	;	)	;

LR(0) Parsing

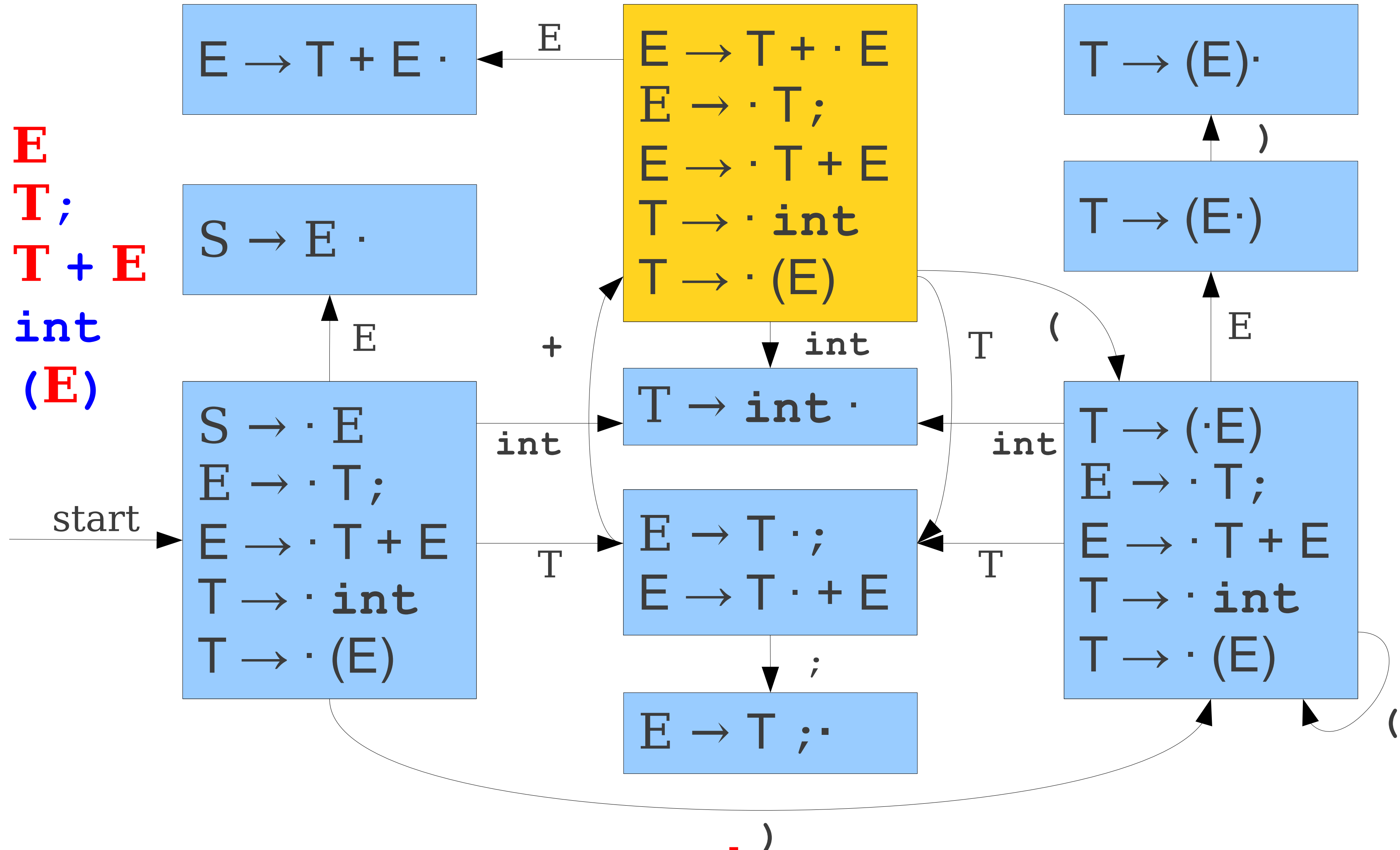
S → **E**
E → **T** `;`
E → **T** `+` **E**
T → `int`
T → `(`**E**`)`





LR(0) Parsing

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

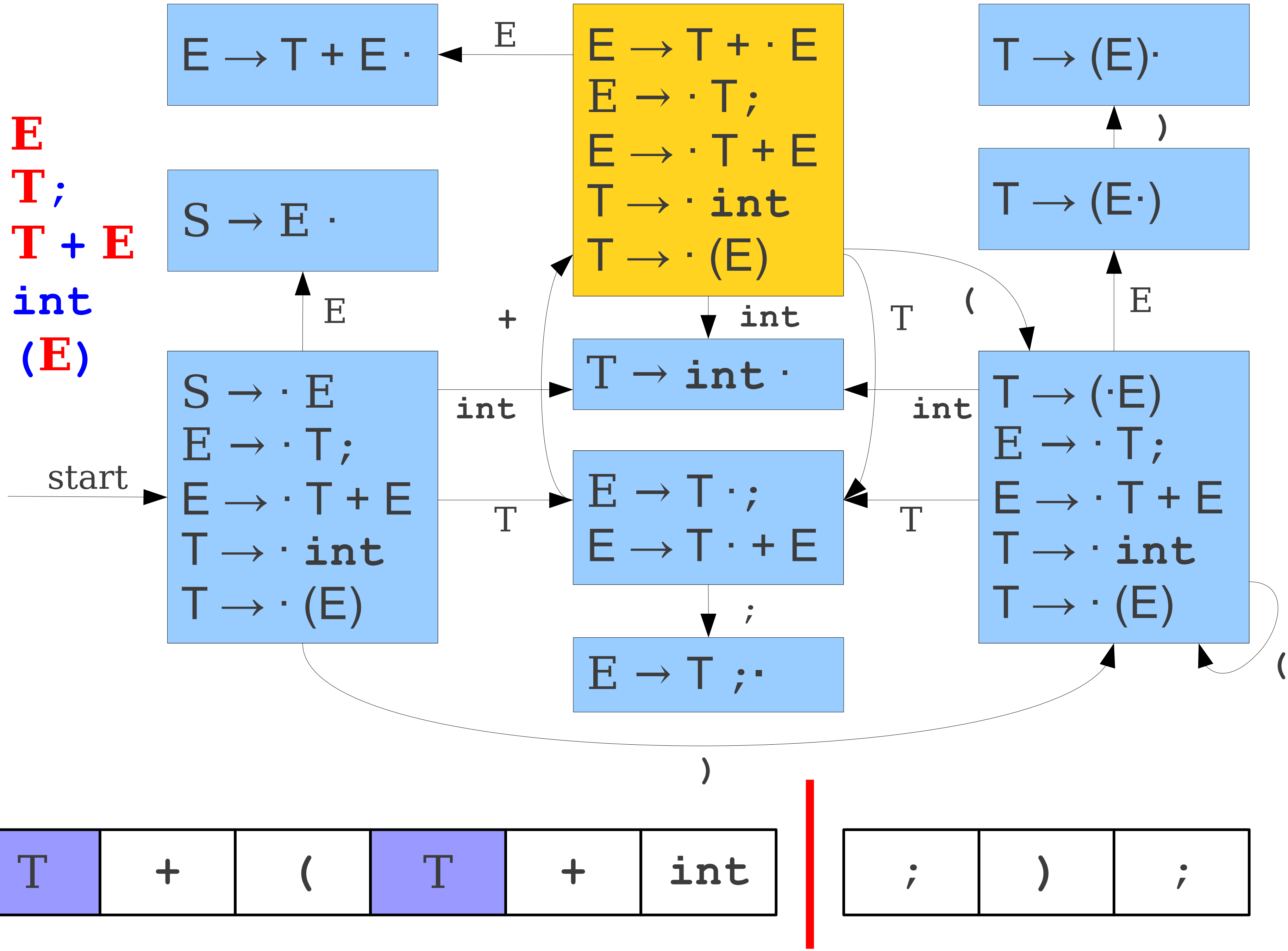


T	+	(	T	+

int	;	)	;

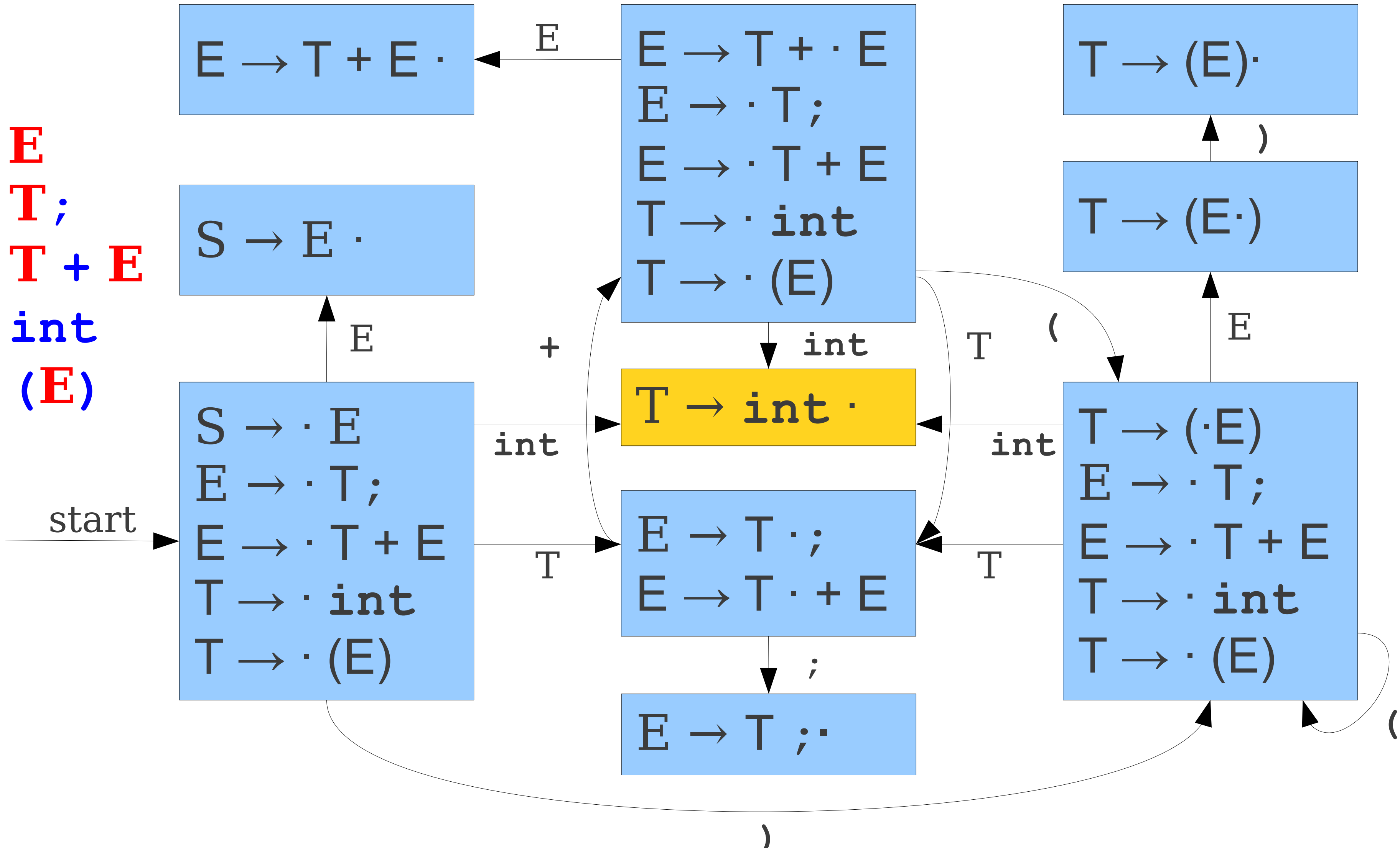
LR(0) Parsing

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



LR(0) Parsing

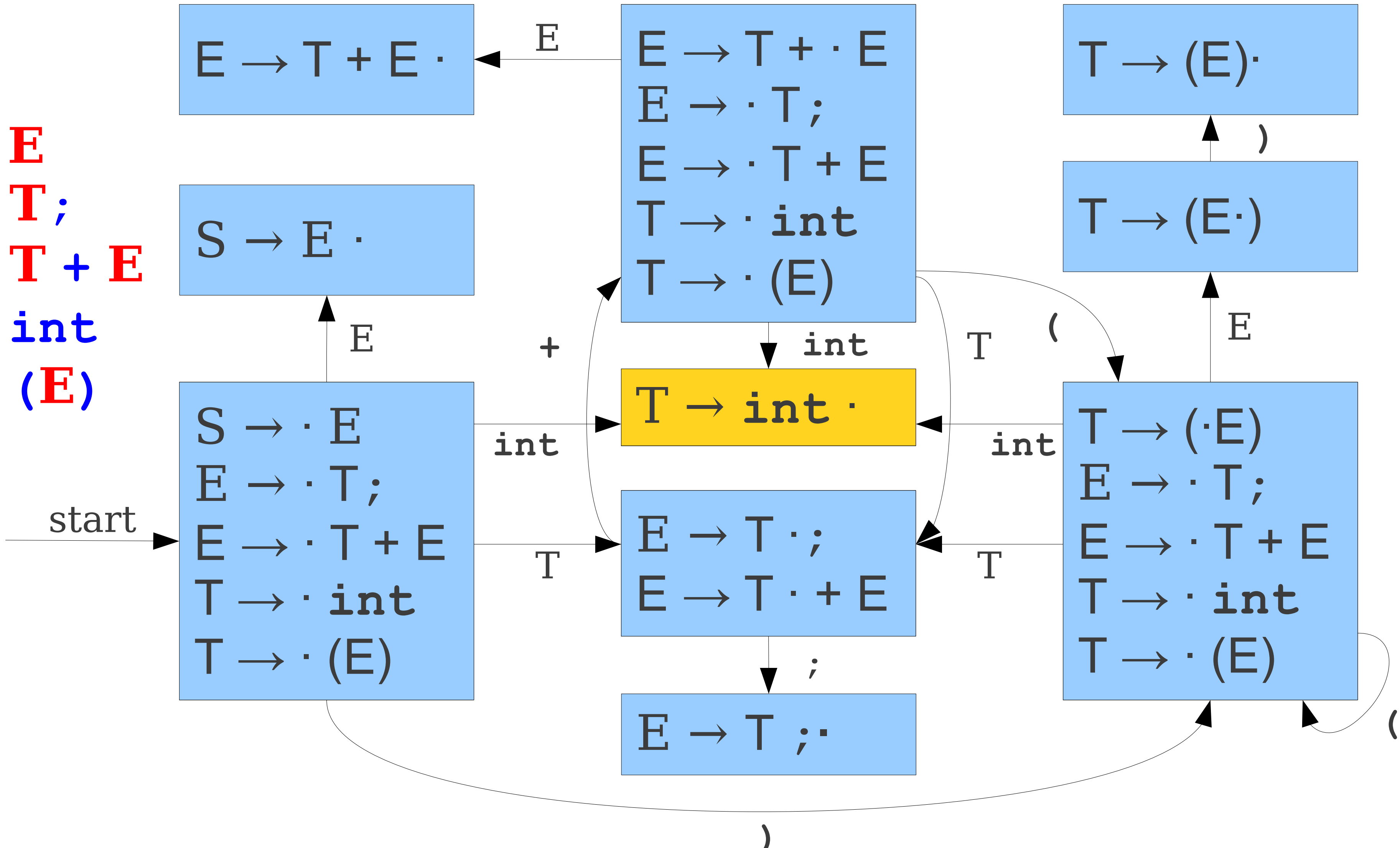
S → **E**
E → **T** ;
E → **T** + **E**
T → int
T → (**E**)

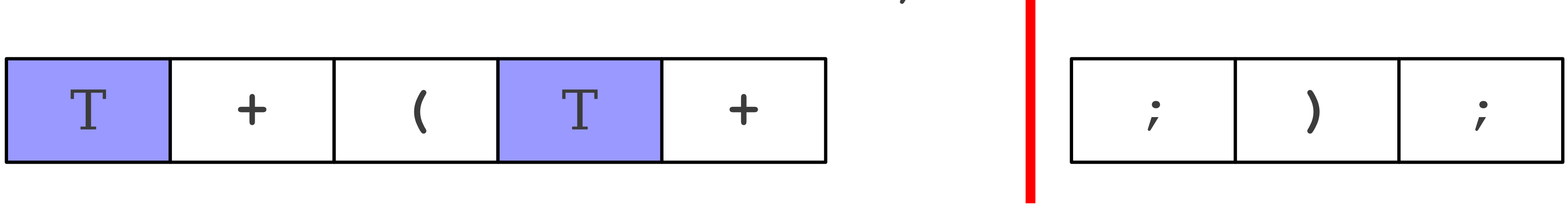


T	+	(	T	+	int	;	)	;

LR(0) Parsing

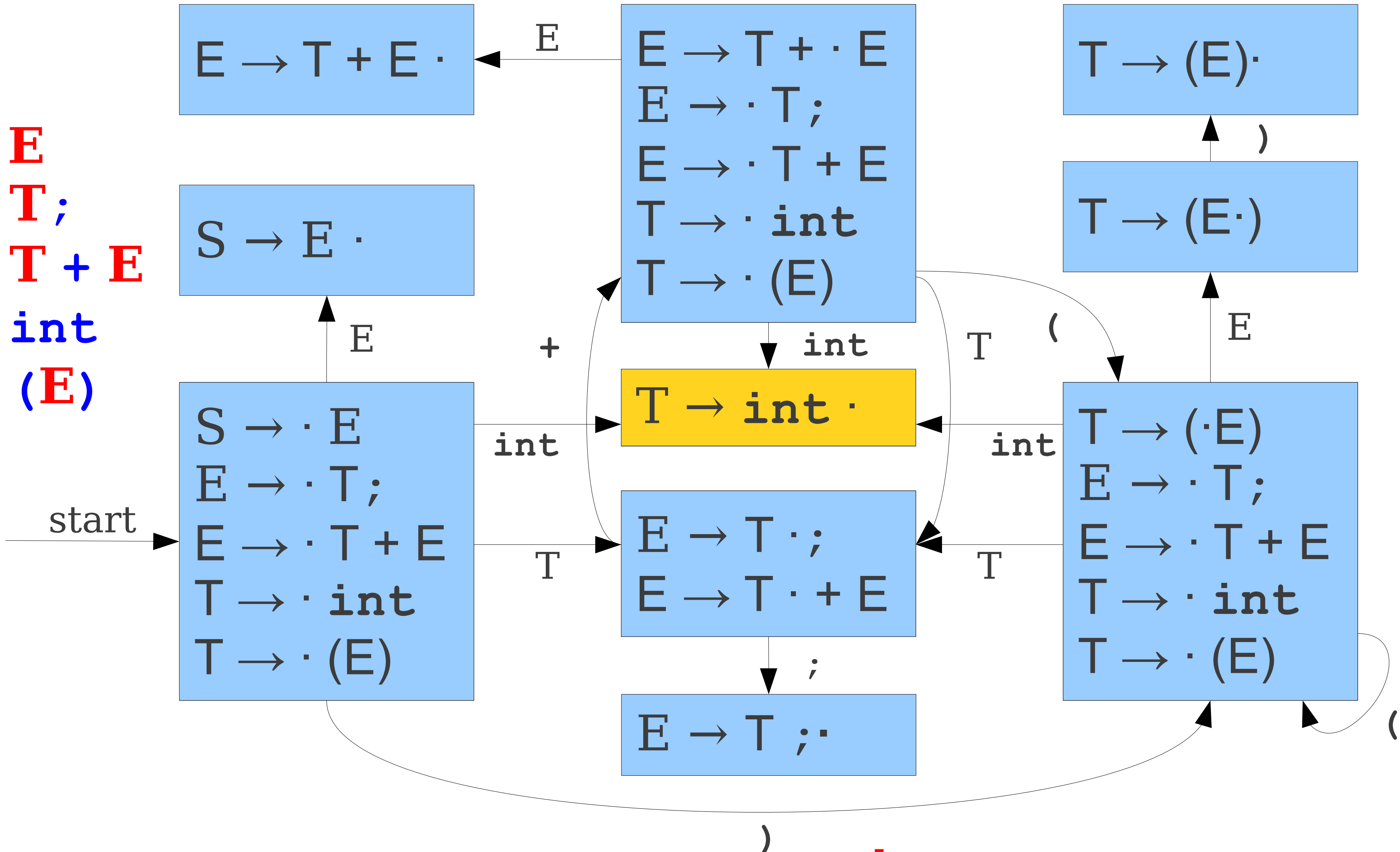
$S \to E$
$E \to T;$
$E \to T + E$
$T \to$ **int**
$T \to (E)$





LR(0) Parsing

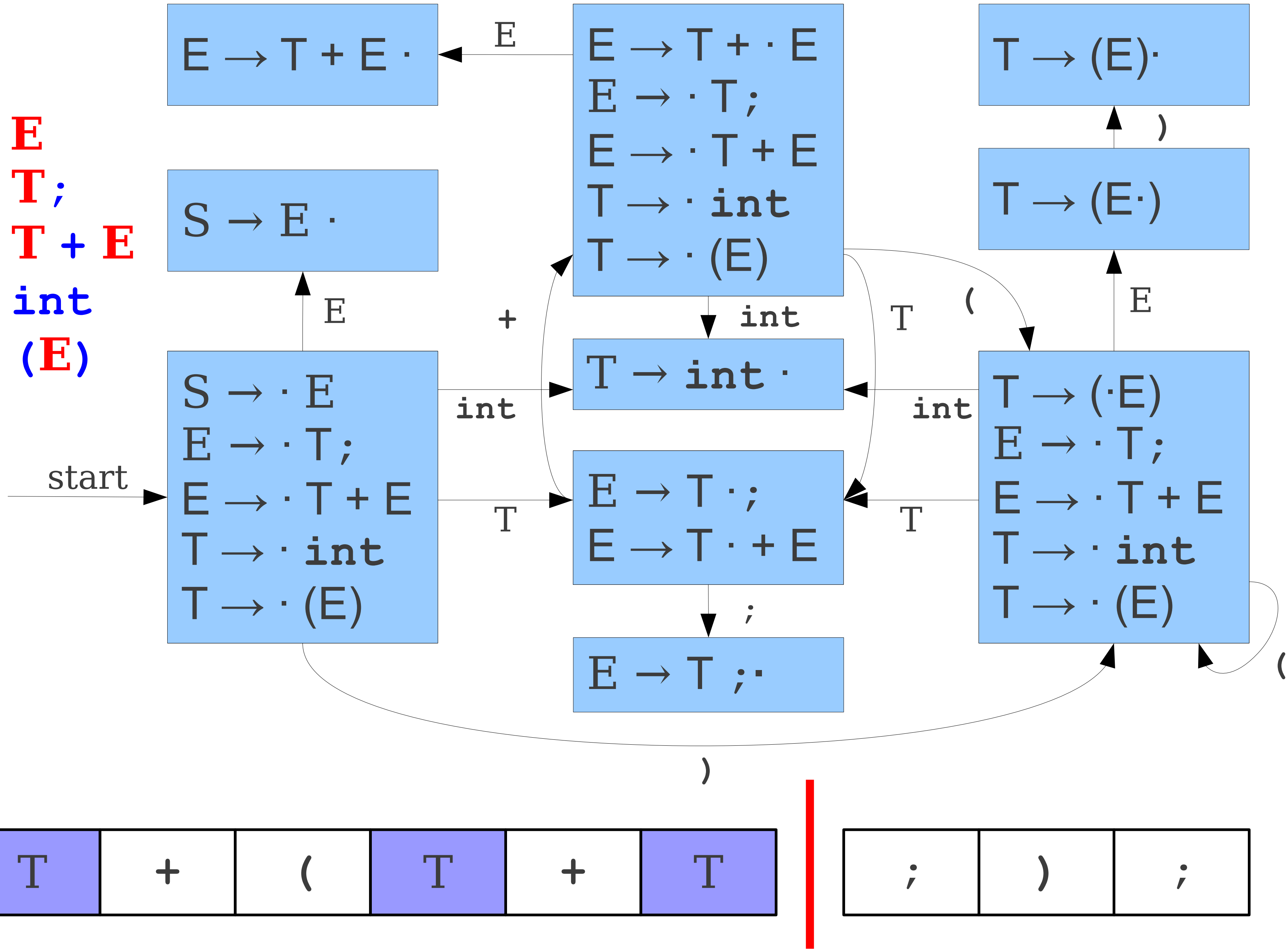
S → **E**
E → **T** ;
E → **T** + **E**
T → int
T → (**E**)



T	+	(	T	+	T	|	;	)	;

LR(0) Parsing

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

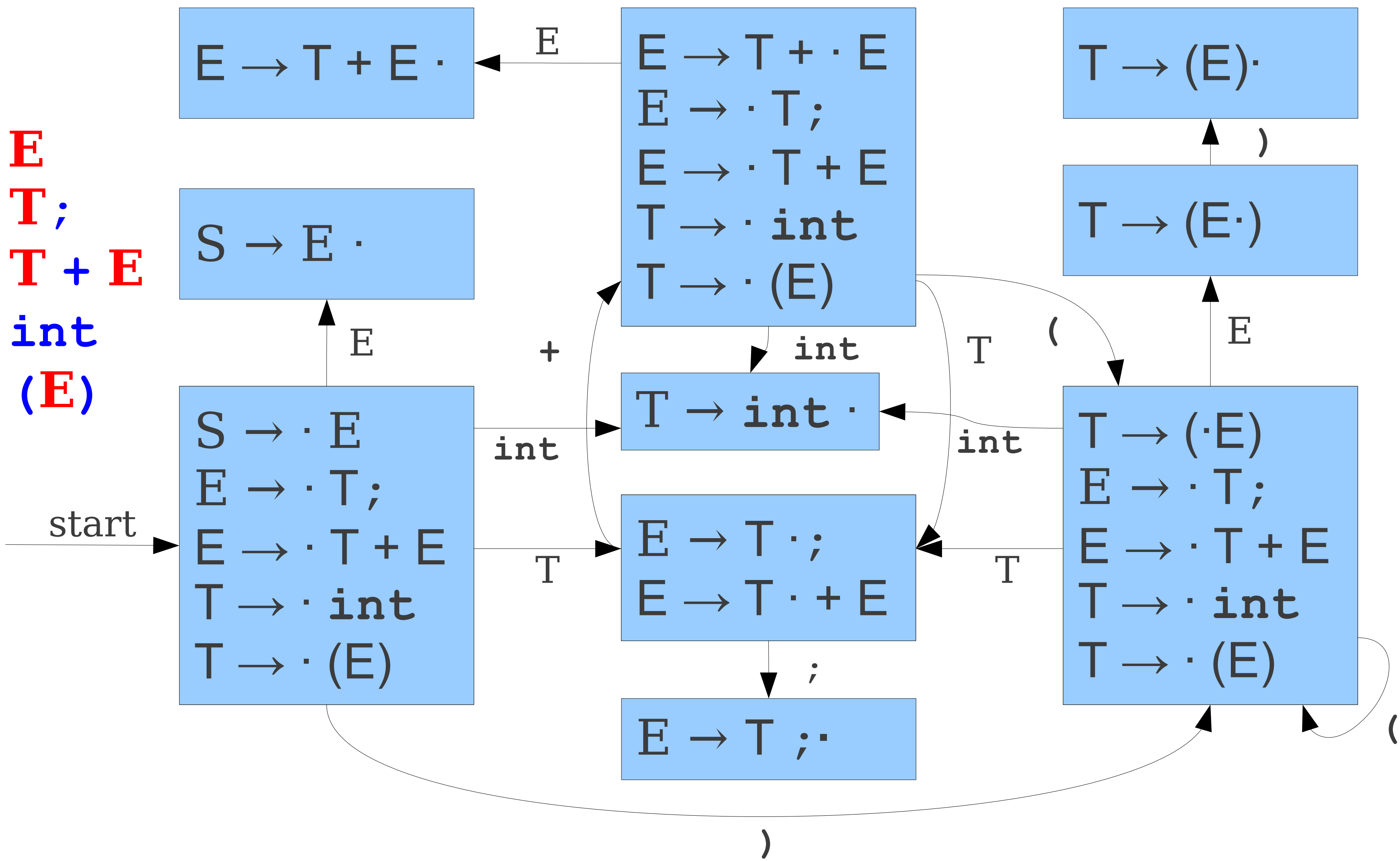


An Optimization

- 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

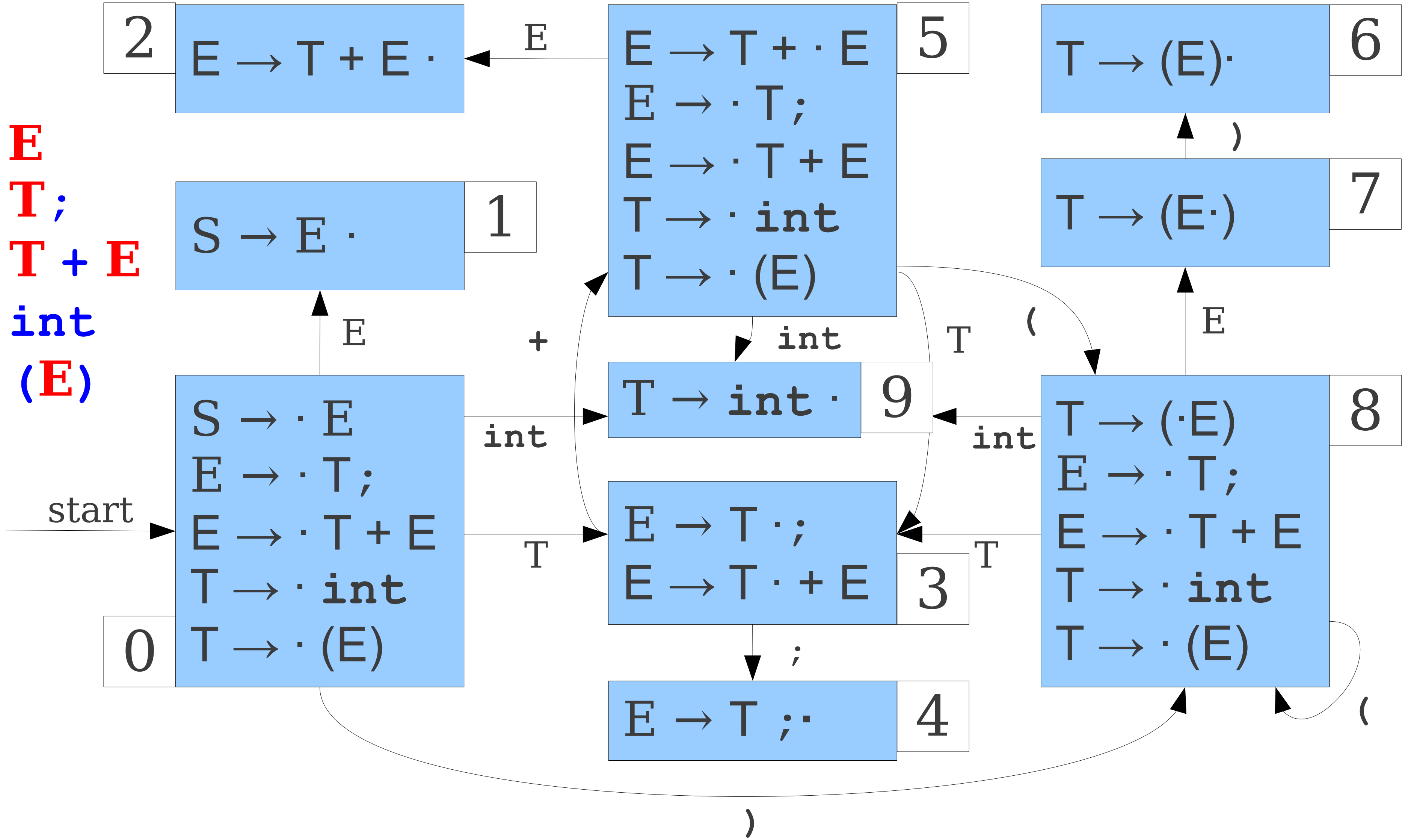
S → **E**
E → **T** ;
E → **T** + **E**
T → int
T → (**E**)



int	+	(	int	+	int	;	)	;

LR(0) Parsing

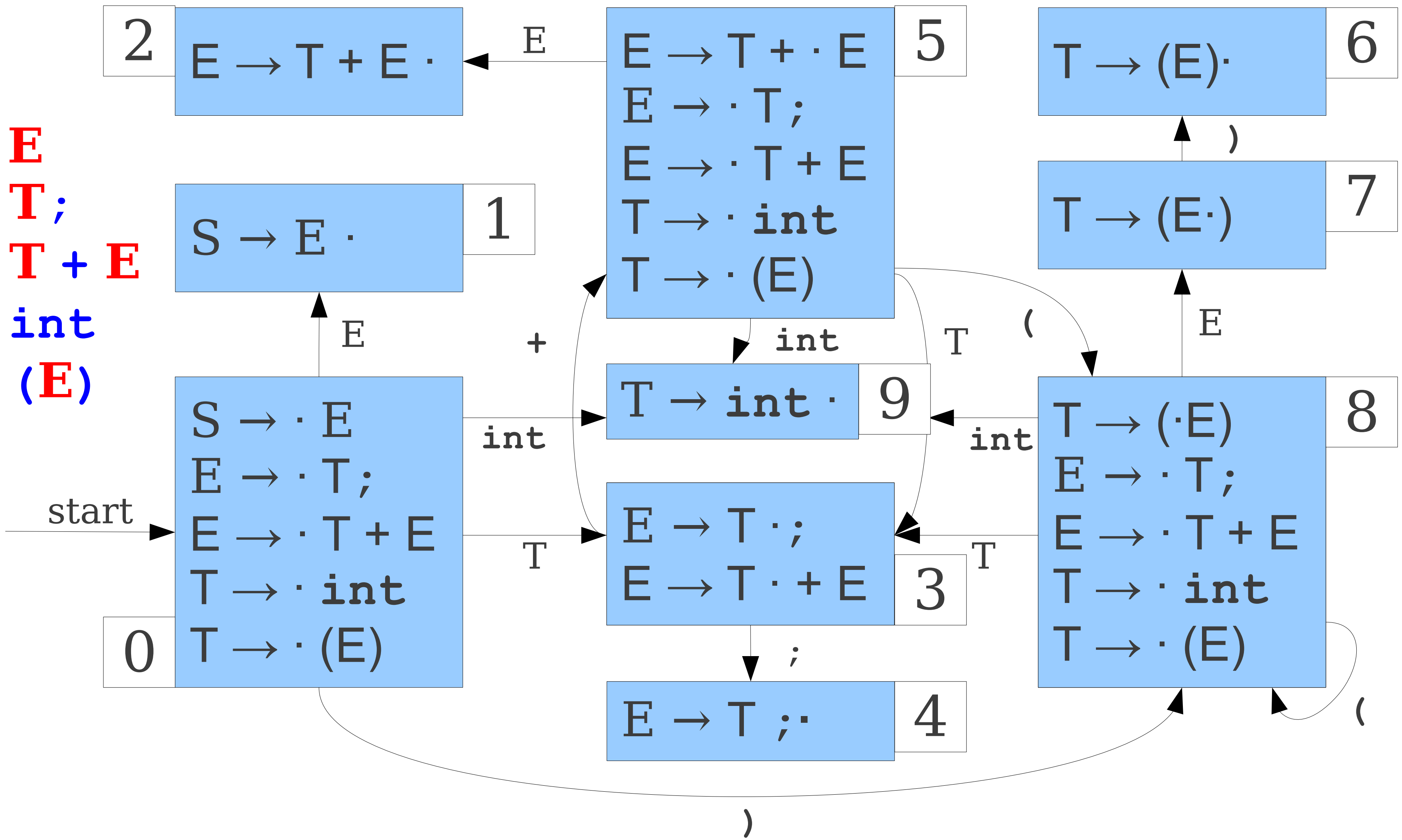
S → E
E → T;
E → T + E
T → int
T → (E)

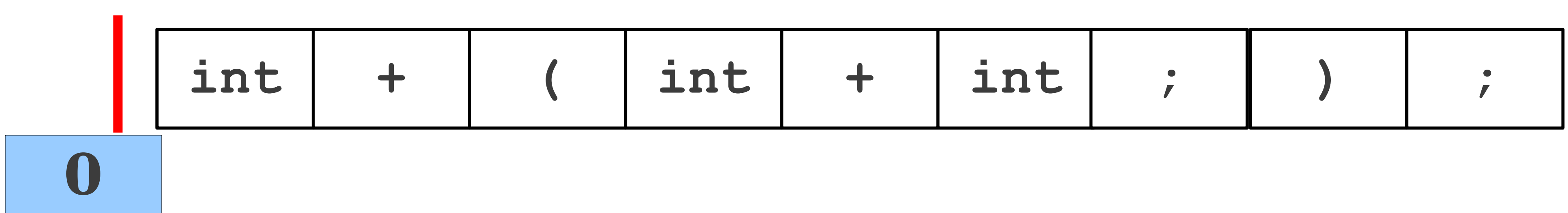


int	+	(	int	+	int	;	)	;

LR(0) Parsing

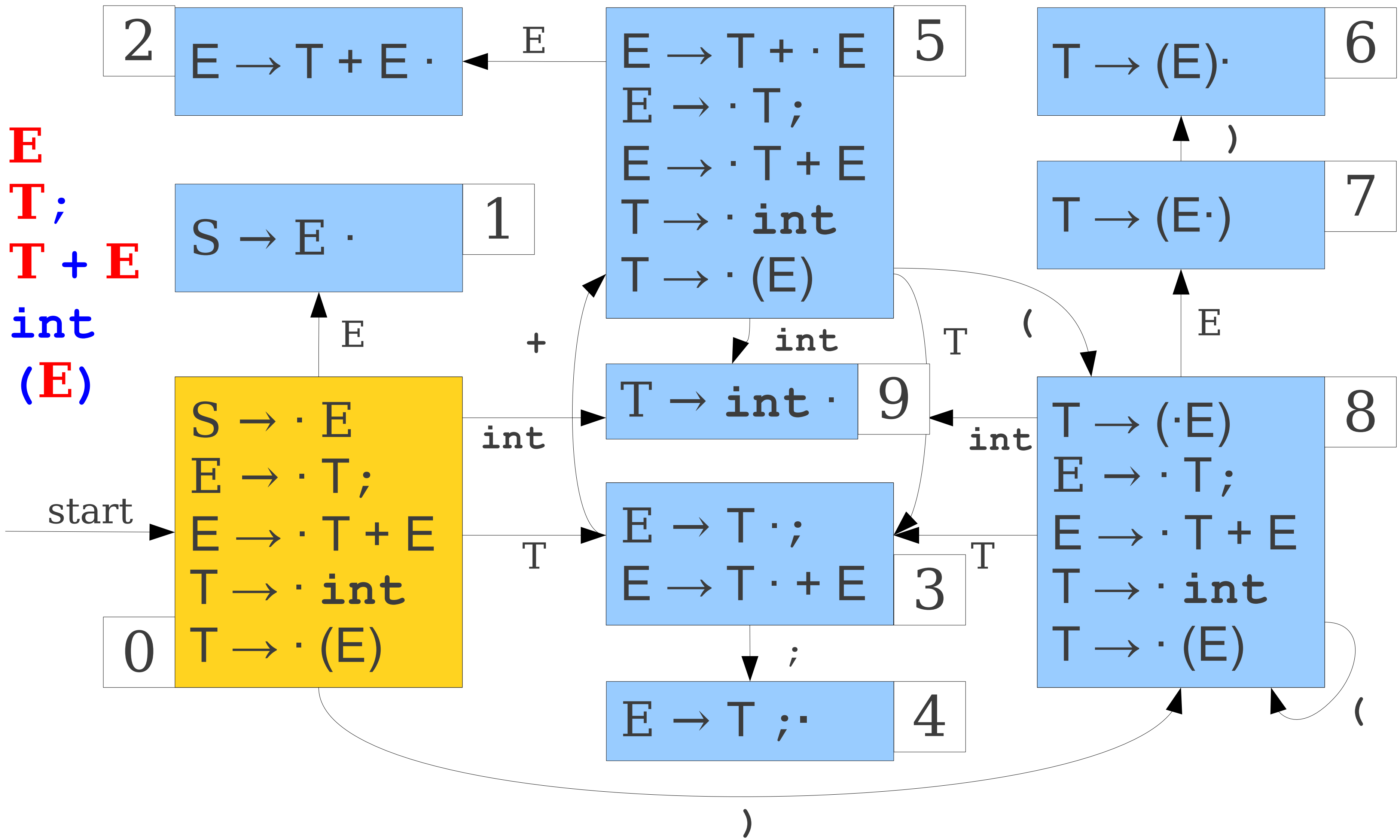
S → E
E → T;
E → T + E
T → int
T → (E)





LR(0) Parsing

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

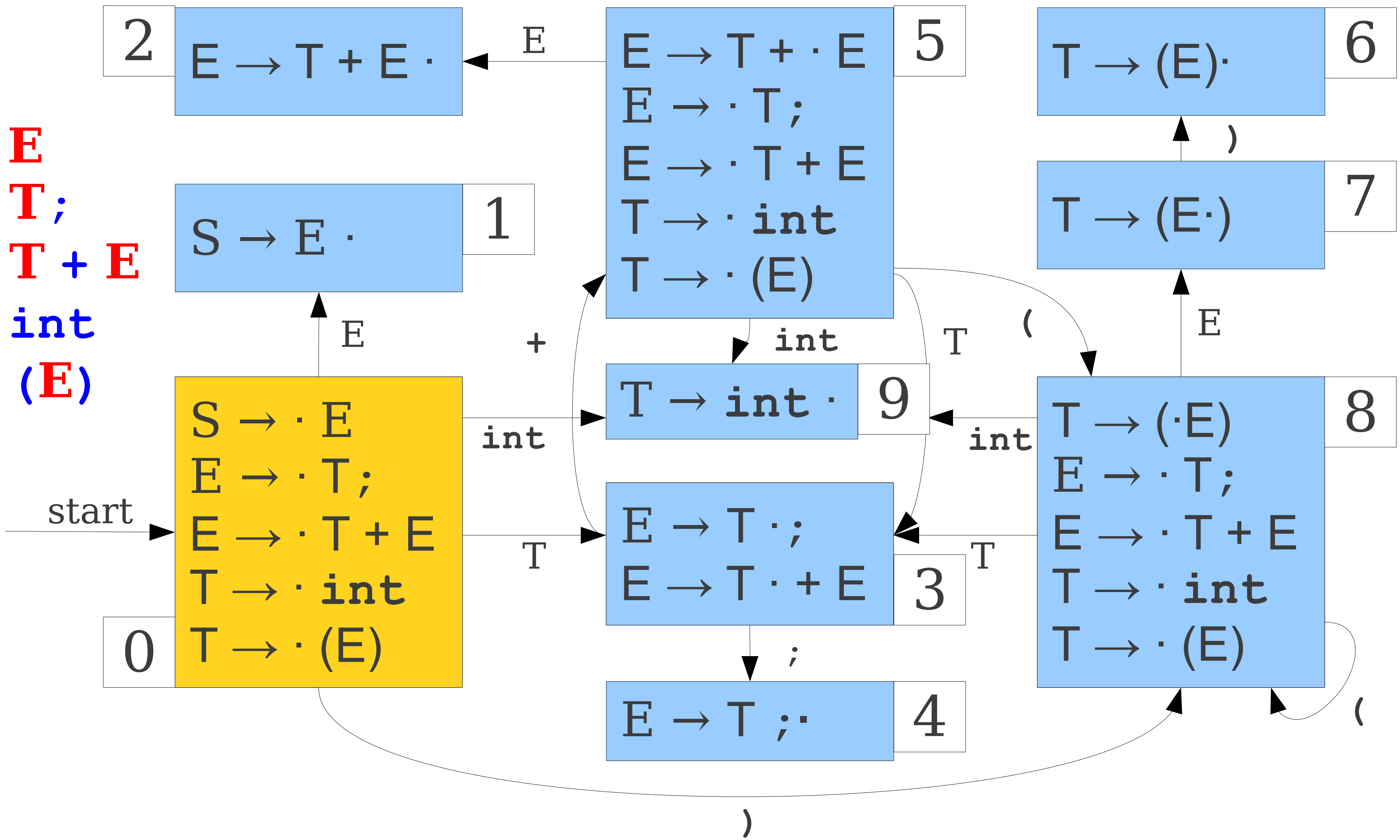


int	+	(	int	+	int	;	)	;

0

LR(0) Parsing

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

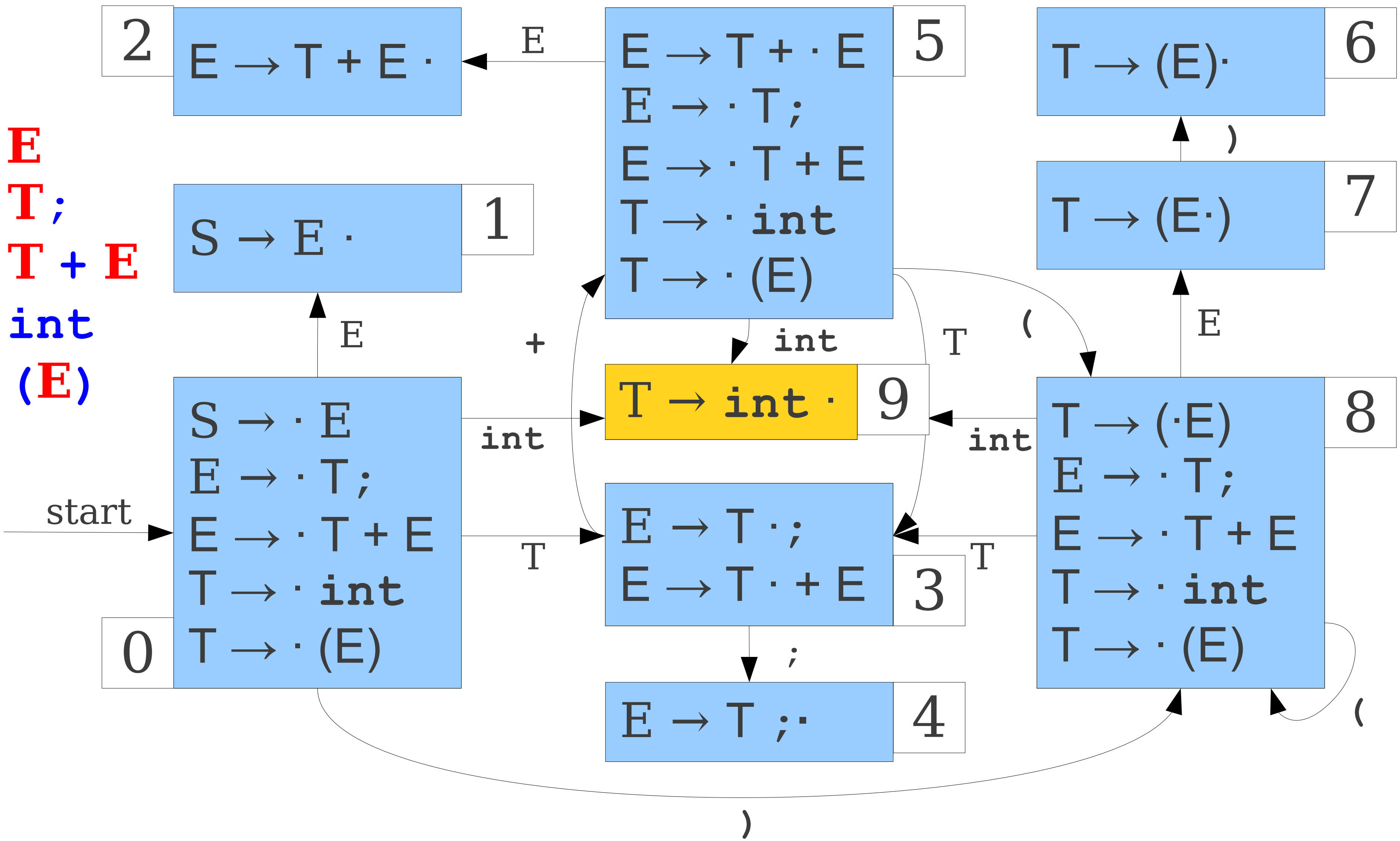


int	|	+	(	int	+	int	;	)	;

0

LR(0) Parsing

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

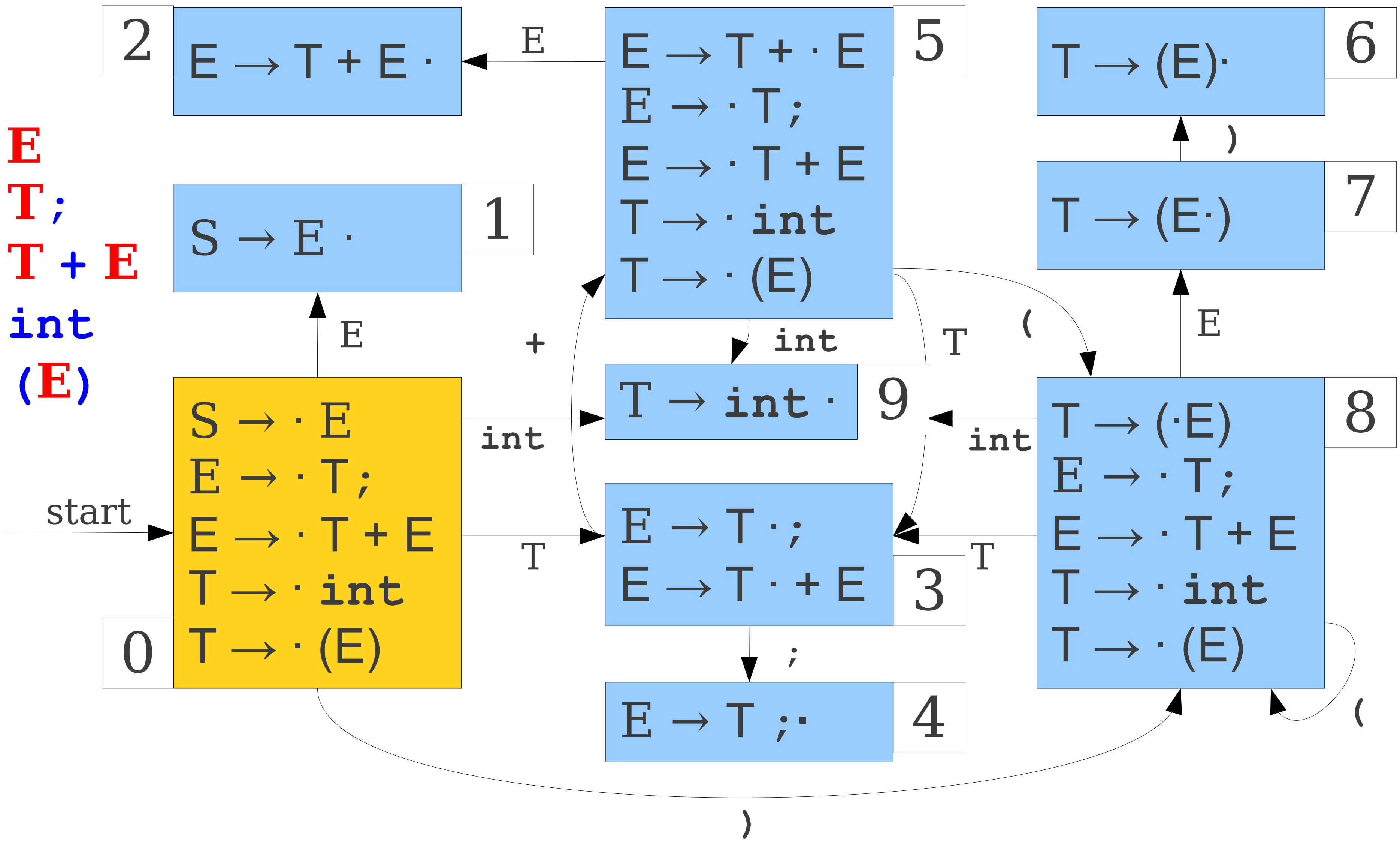


int	‖	+	(	int	+	int	;	)	;

0

LR(0) Parsing

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

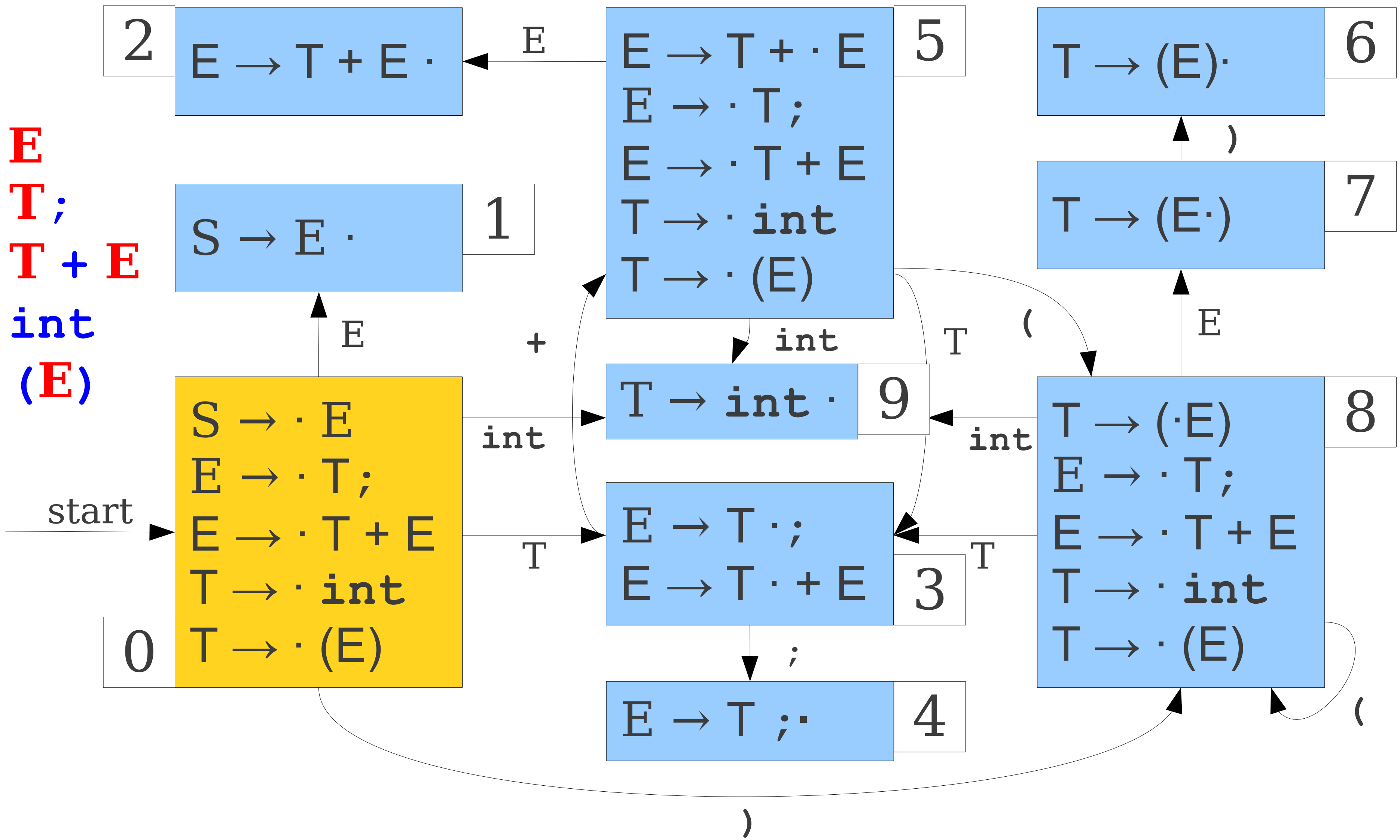


+	(	int	+	int	;	)	;

0

LR(0) Parsing

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

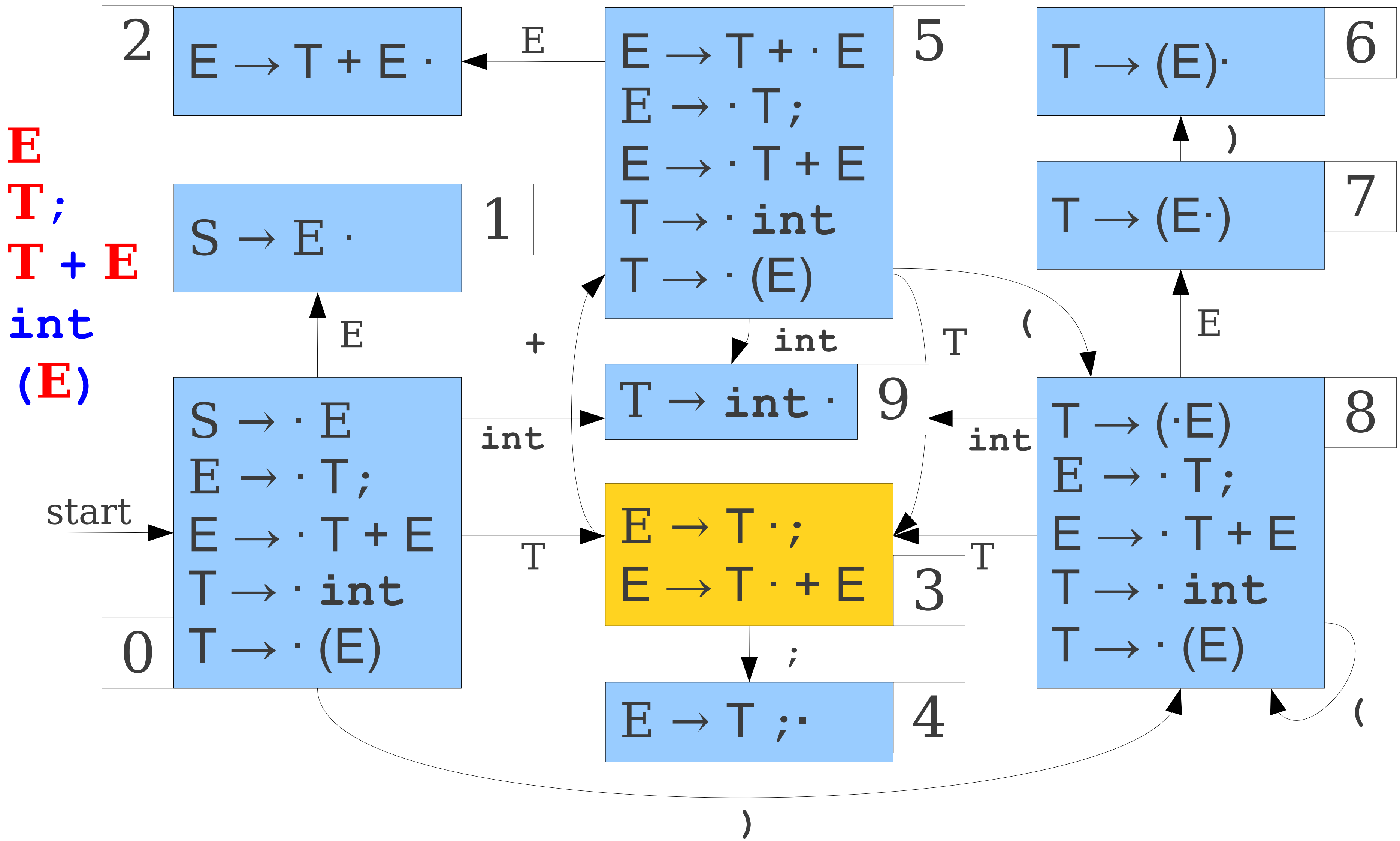


T
0

+	(	int	+	int	;	)	;

LR(0) Parsing

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

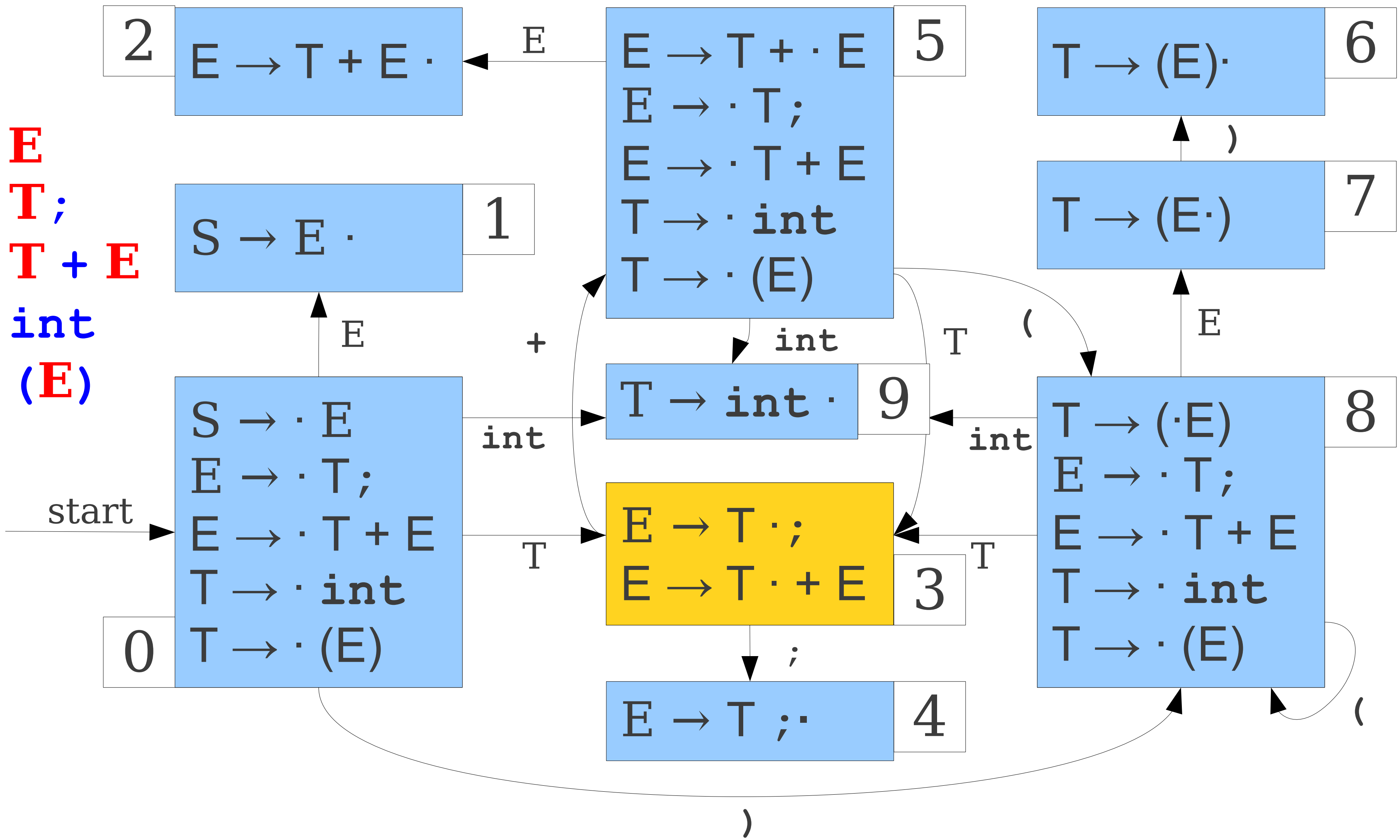


T	+	(	int	+	int	;	)	;

0

LR(0) Parsing

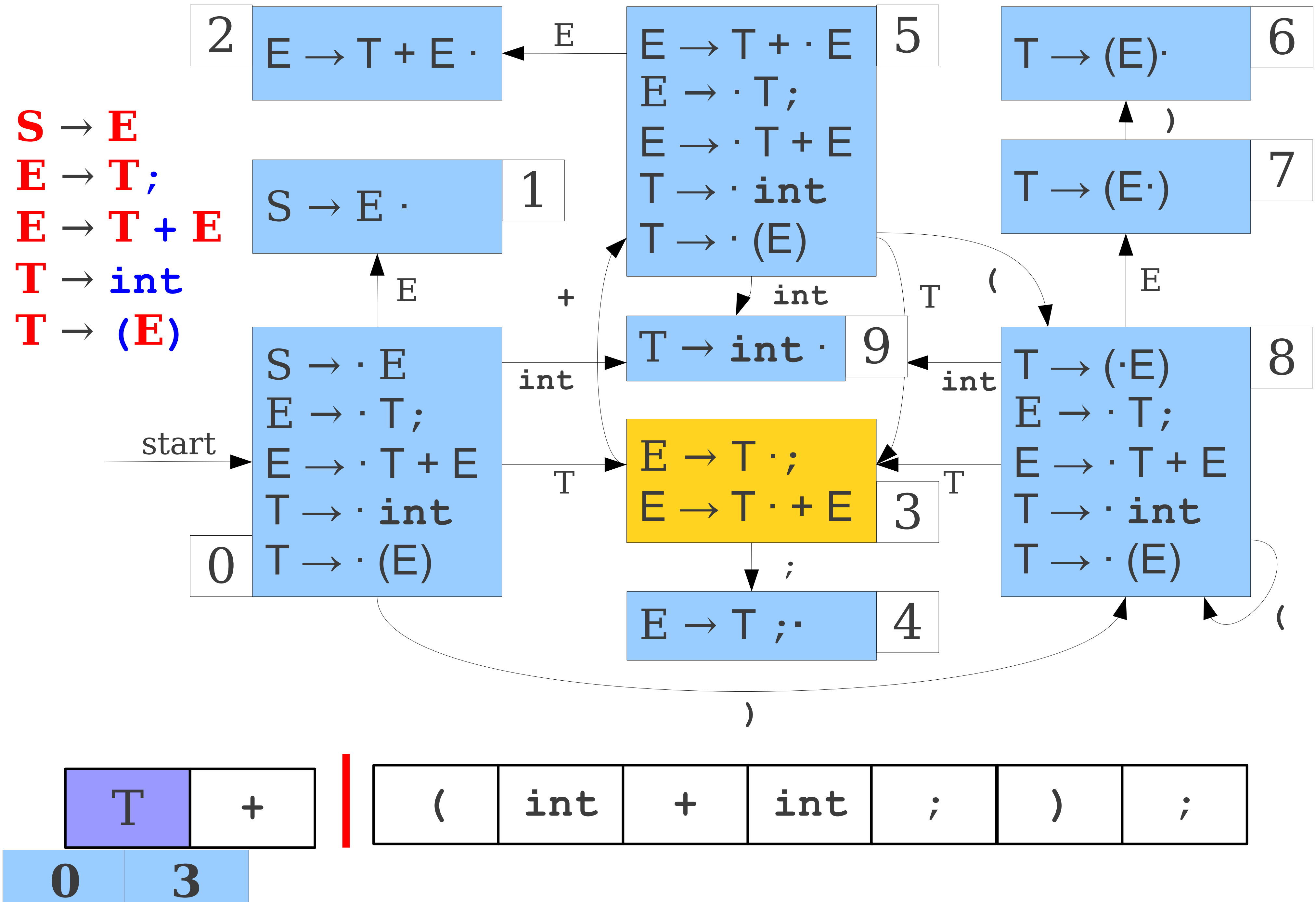
S → **E**
E → **T** ;
E → **T** + **E**
T → **int**
T → **(E)**



T		+	(	int	+	int	;	)	;
0	3								

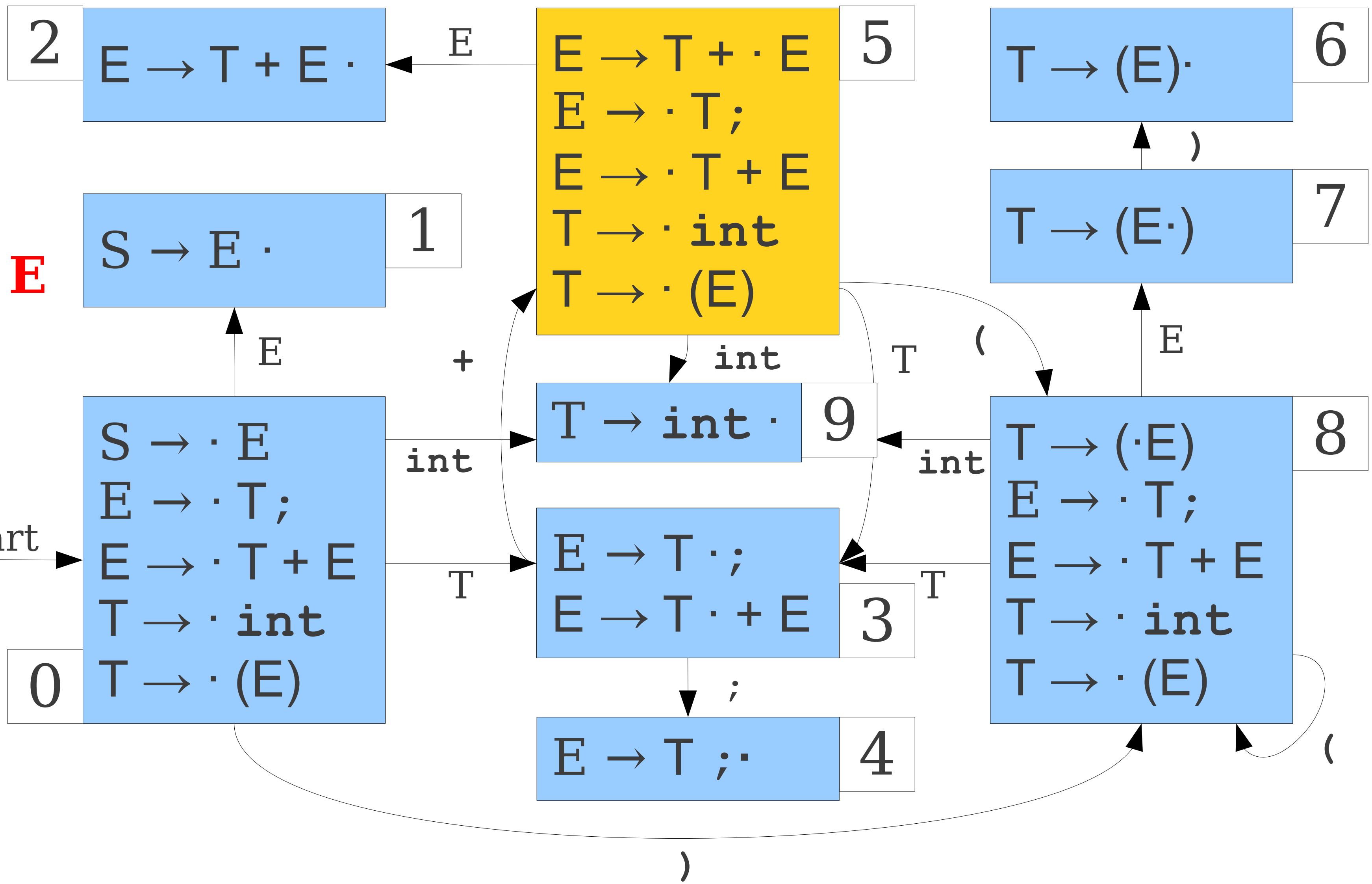
LR(0) Parsing

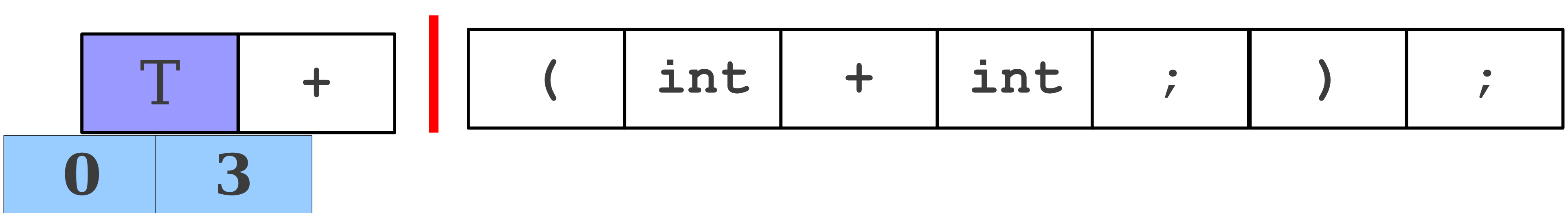
S → **E**
E → **T** `;`
E → **T** + **E**
T → `int`
T → `(E)`



LR(0) Parsing

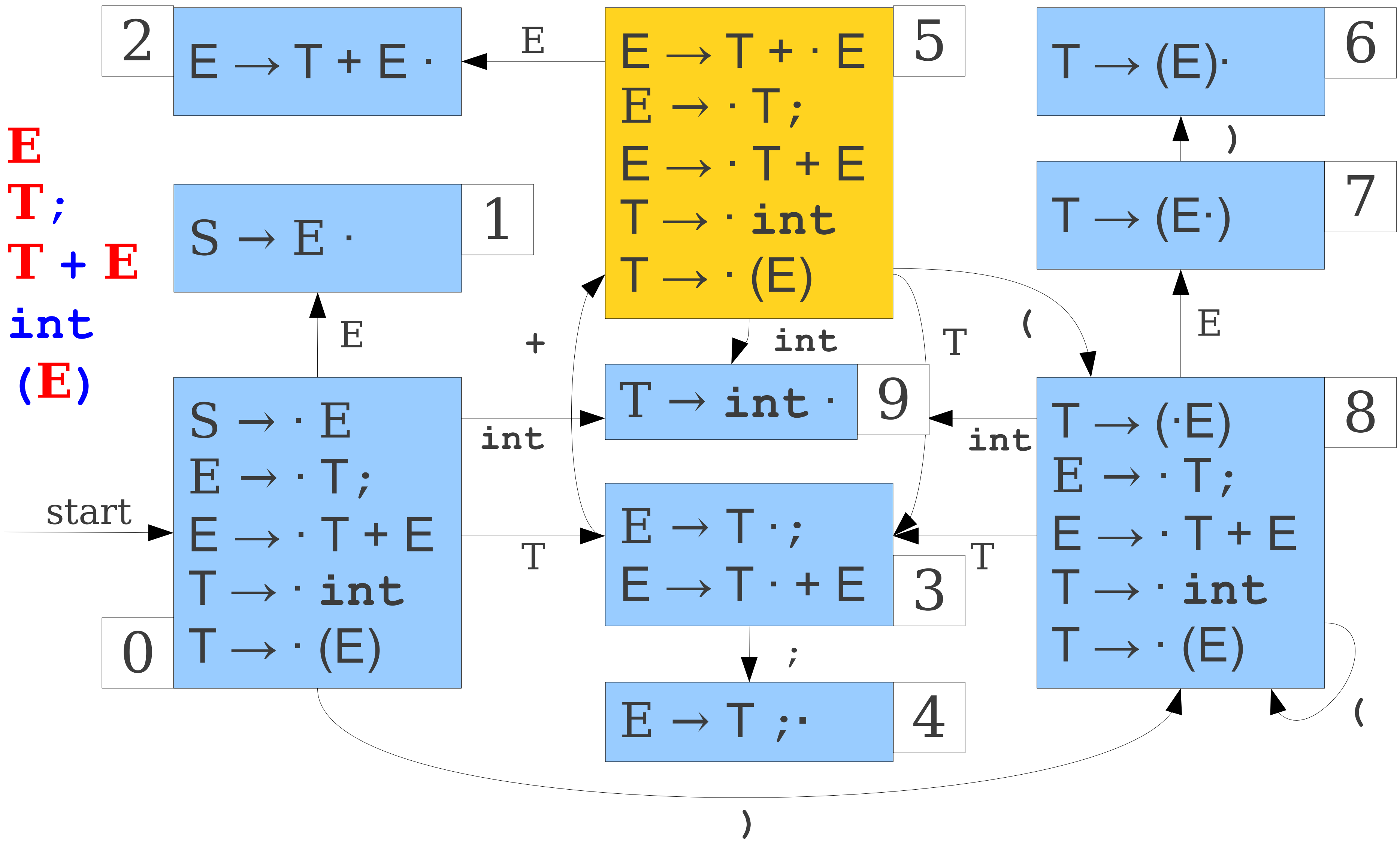
S → **E**
E → **T** ;
E → **T** + **E**
T → `int`
T → (**E**)

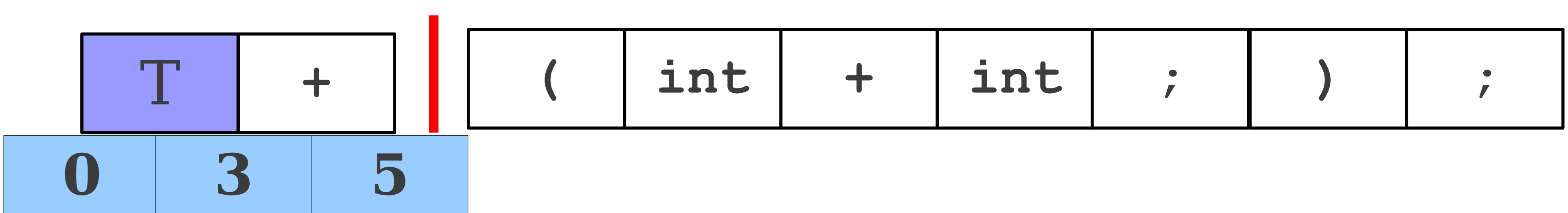




LR(0) Parsing

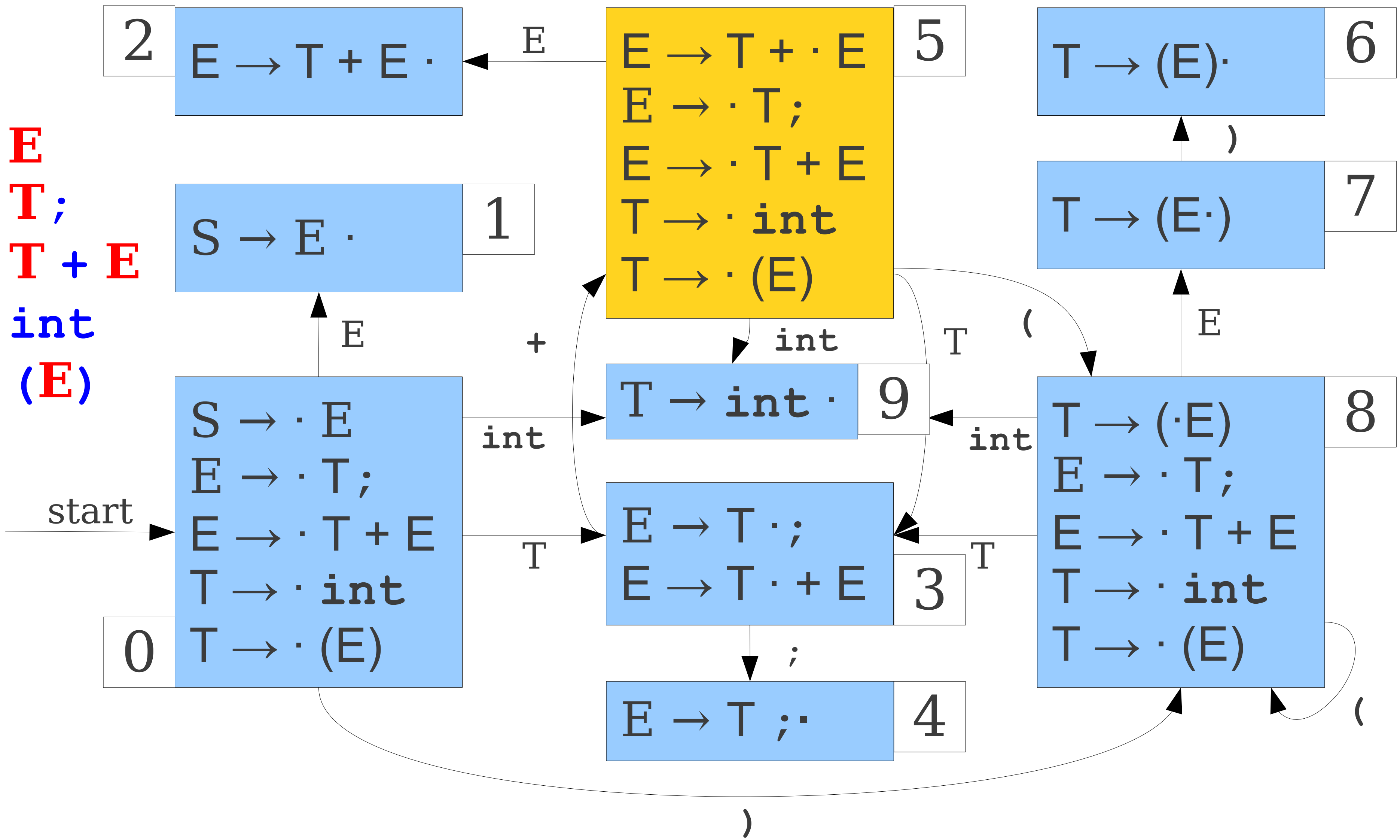
S → **E**
E → **T** **;**
E → **T** **+** **E**
T → **int**
T → **(E)**





LR(0) Parsing

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



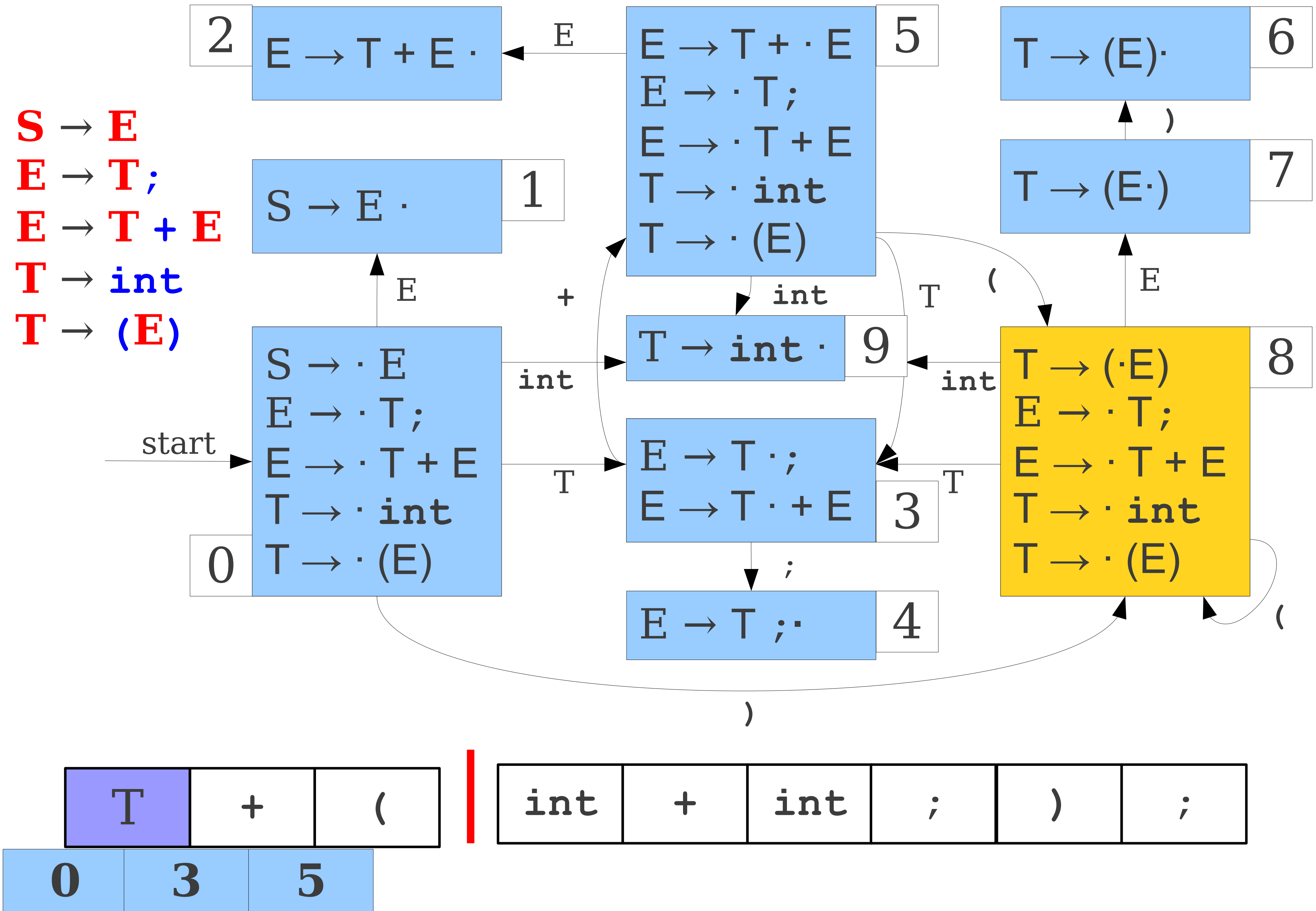
T	+	(	|	int	+	int	;	)

int	+	int	;	)	;

0	3	5

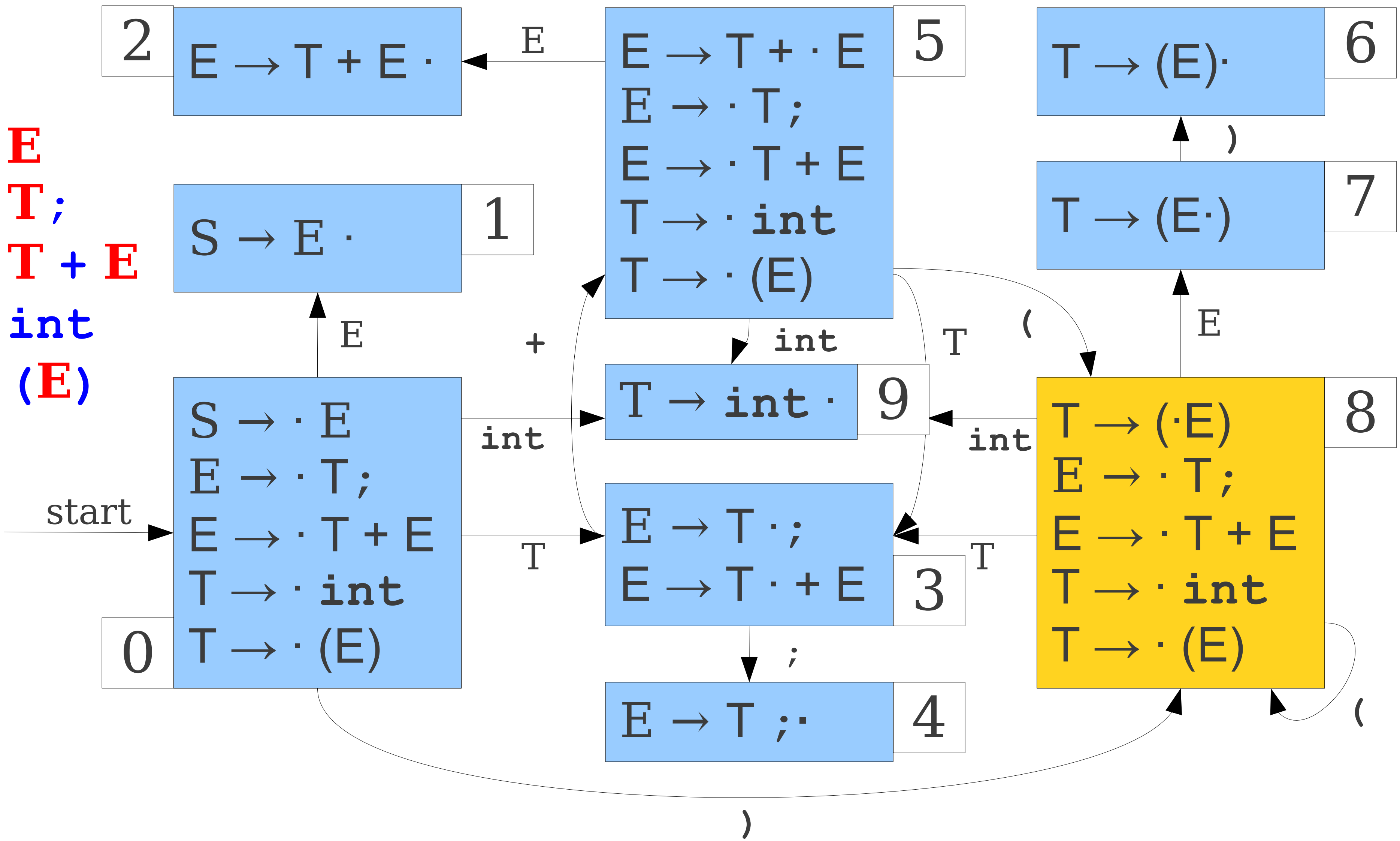
LR(0) Parsing

$$\begin{aligned} S &\to E \\ E &\to T; \\ E &\to T + E \\ T &\to \texttt{int} \\ T &\to (E) \end{aligned}$$



LR(0) Parsing

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

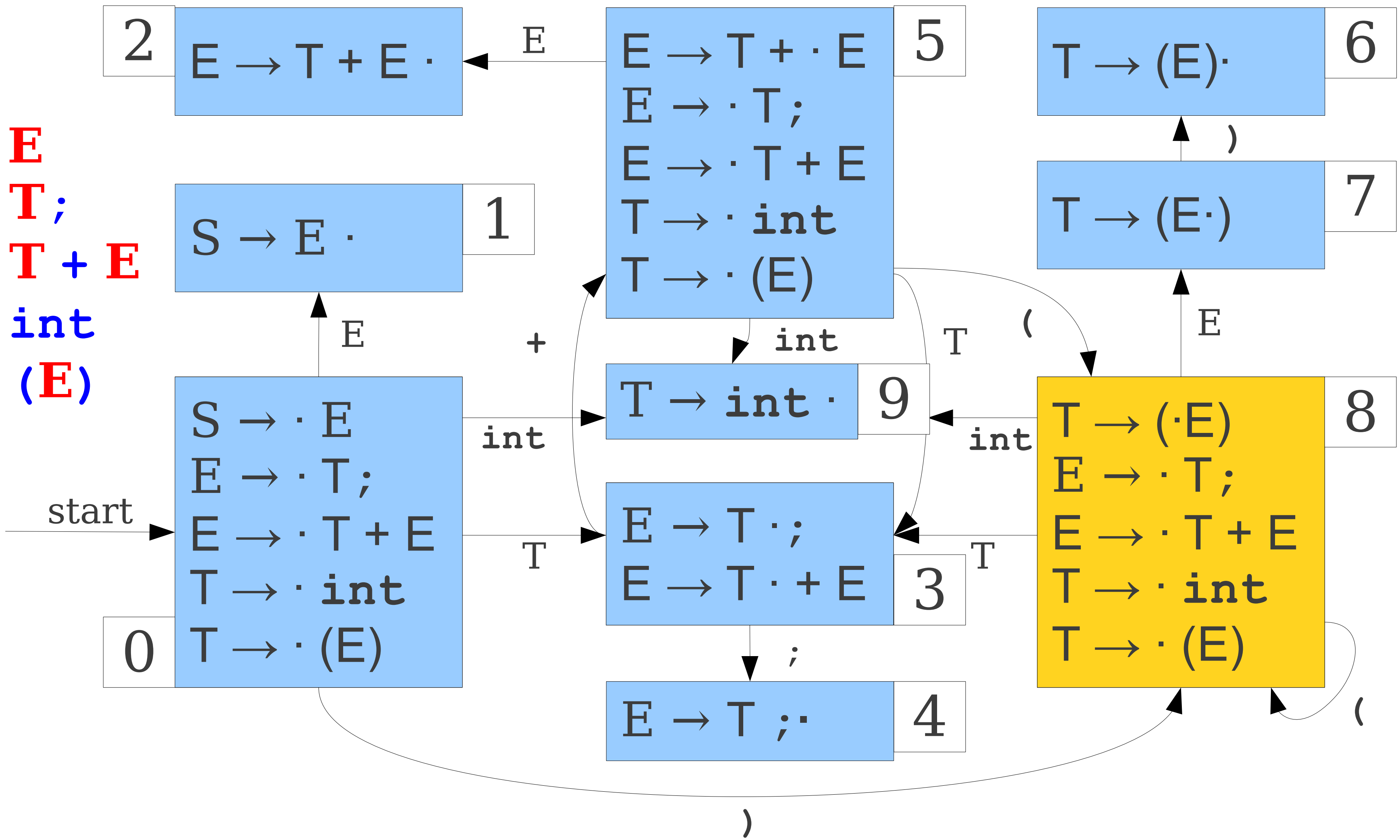


T	+	(
0	3	5	8

int	+	int	;	)	;

LR(0) Parsing

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

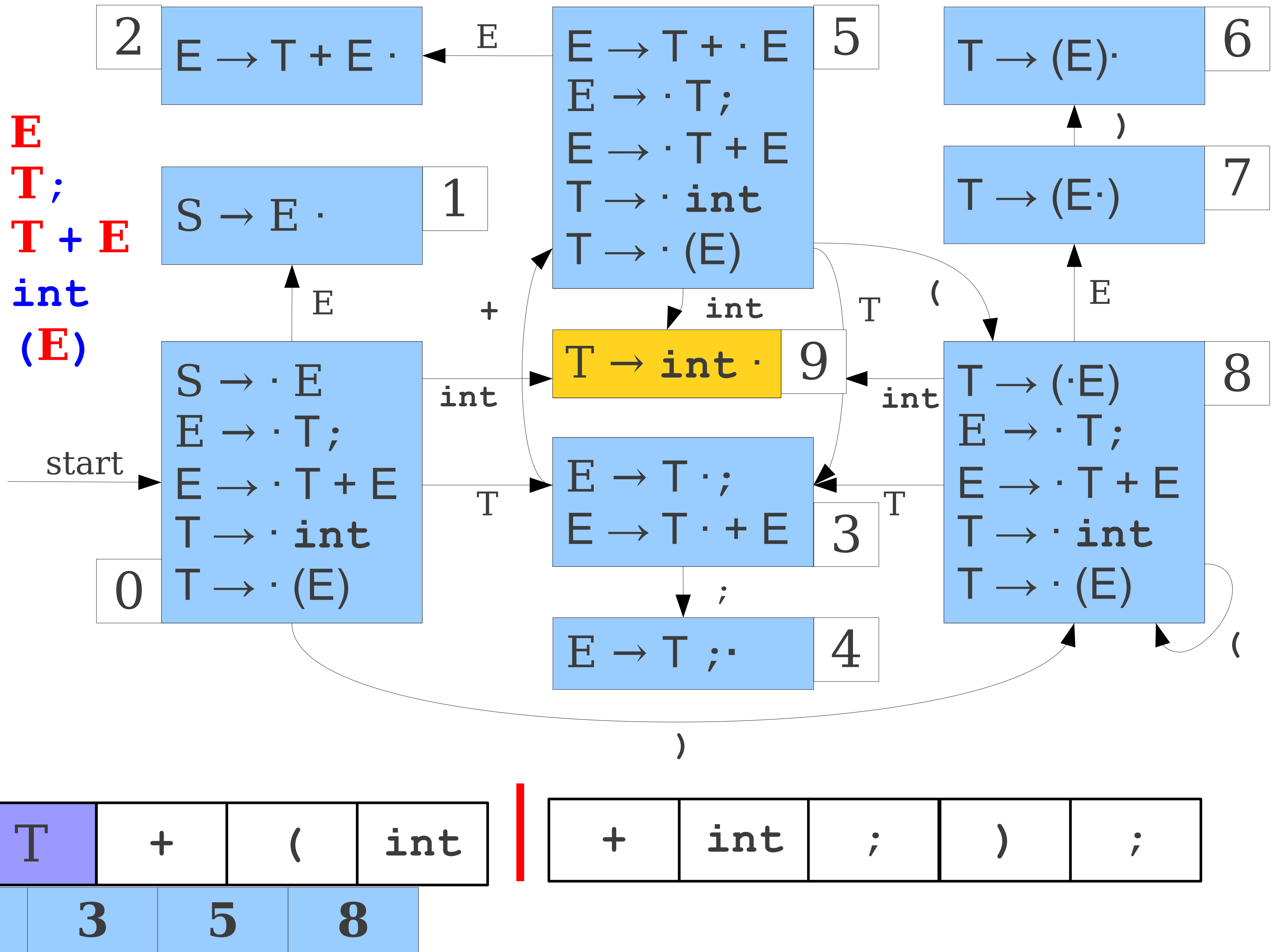


T	+	(	int
0	3	5	8

+	int	;	)	;

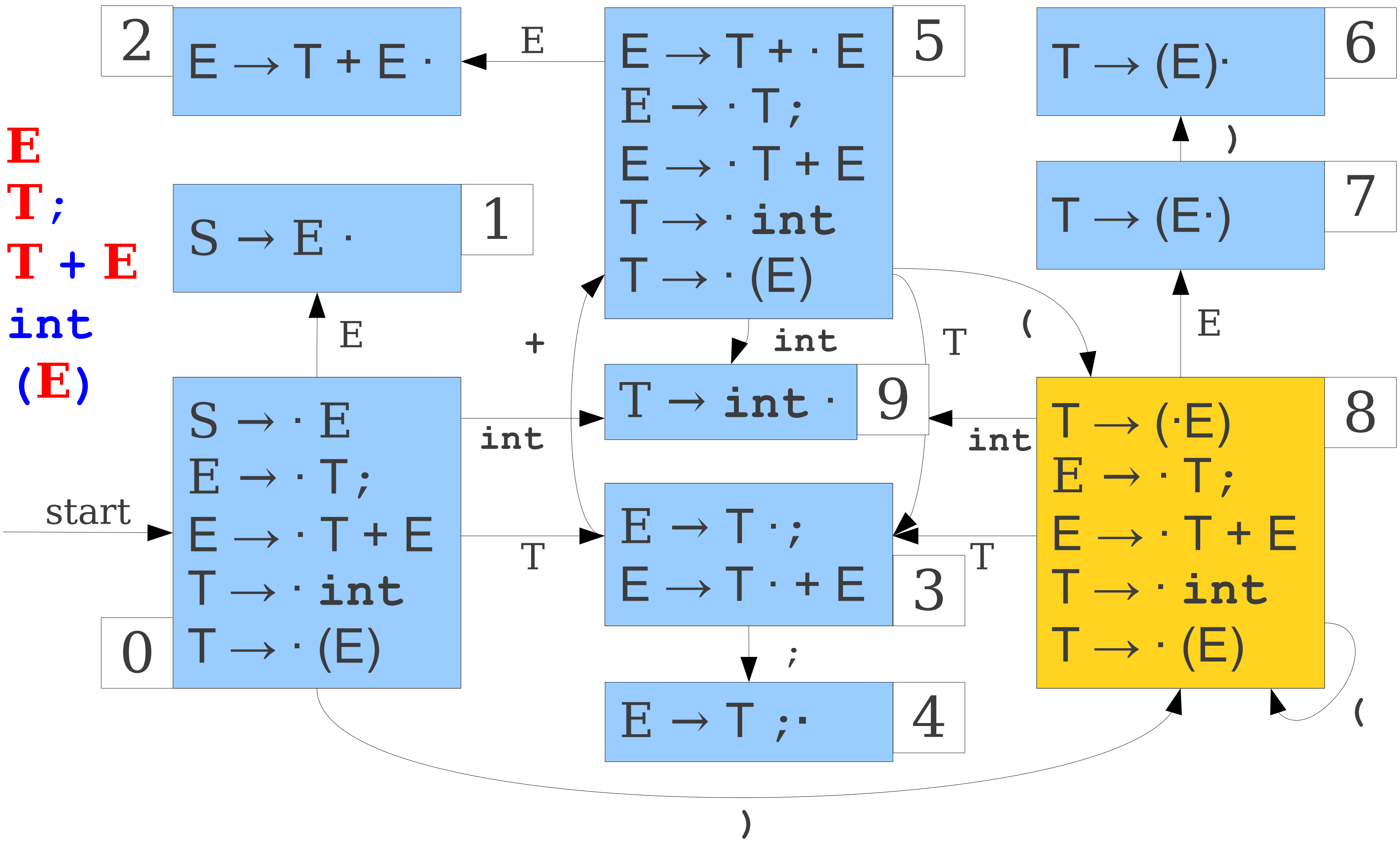
LR(0) Parsing

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**)



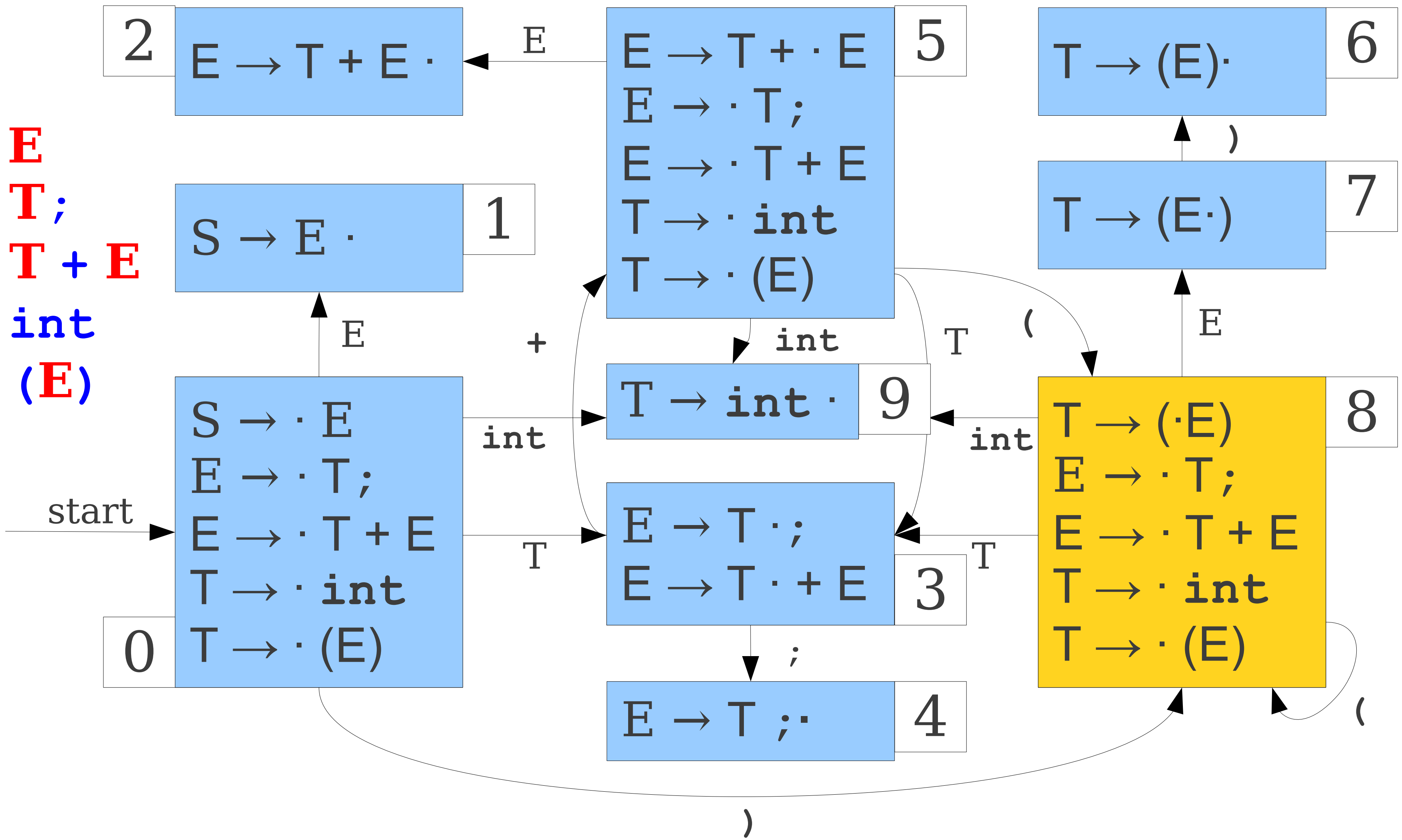
T	+	(

0	3	5	8

+	int	;	)	;

LR(0) Parsing

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

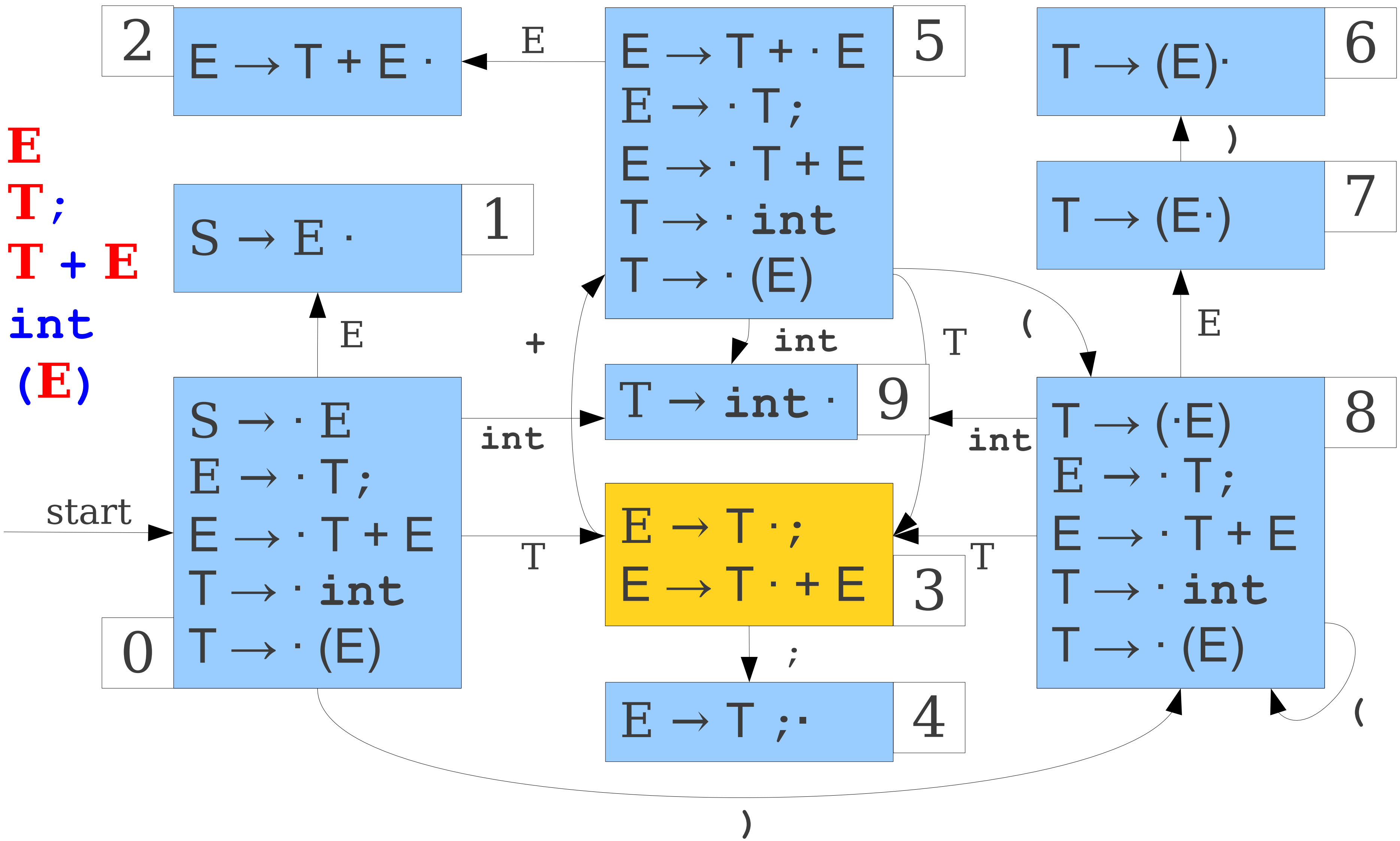


T	+	(	T
0	3	5	8

+	int	;	)	;

LR(0) Parsing

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



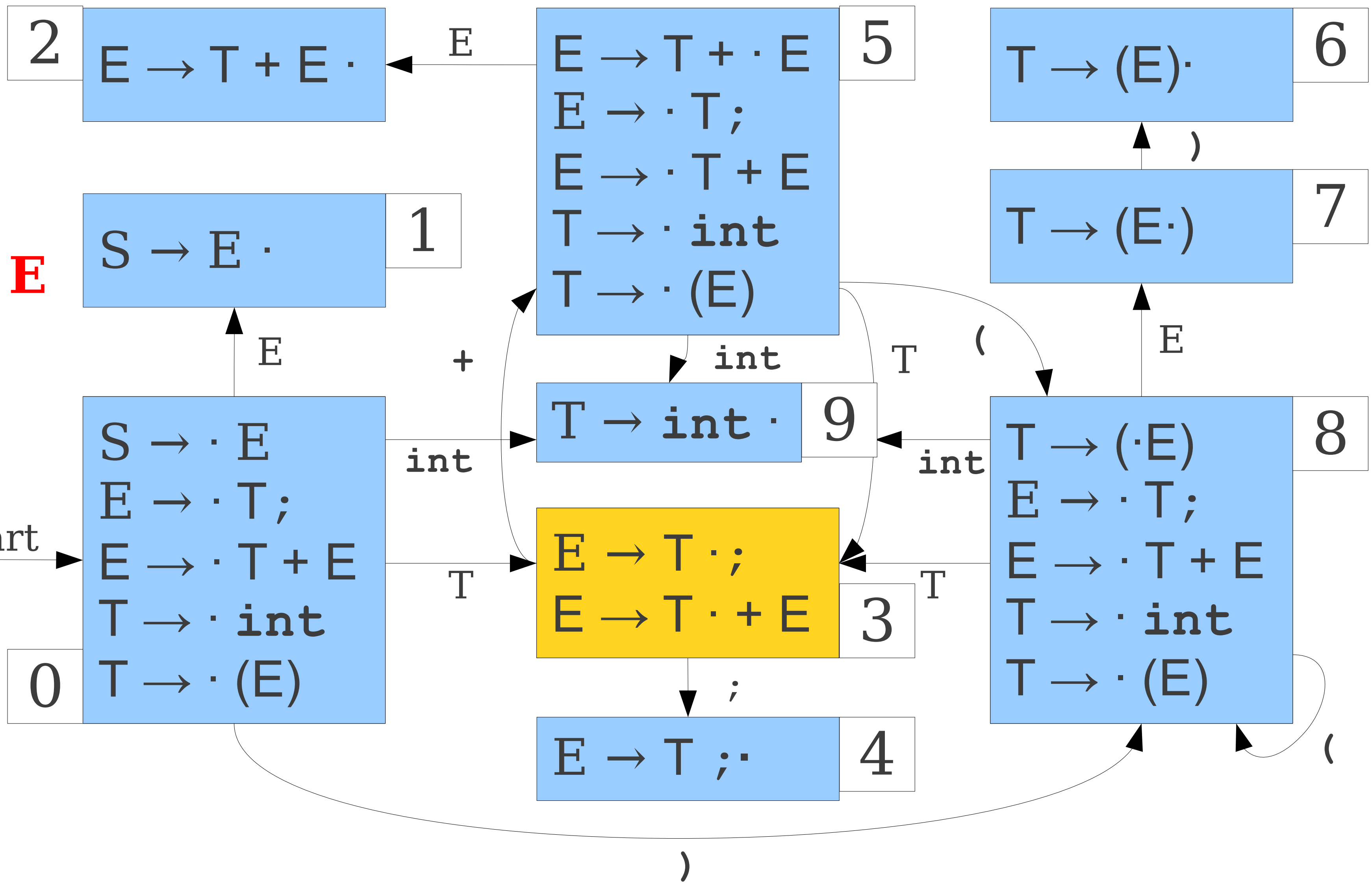
T	+	(	T

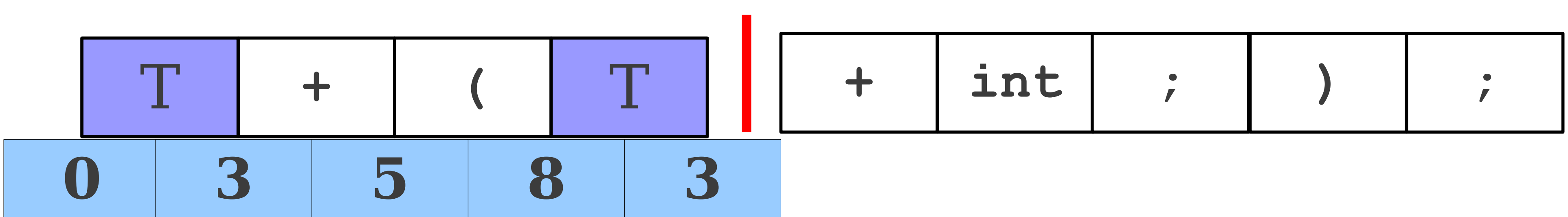
0	3	5	8

+	int	;	)	;

LR(0) Parsing

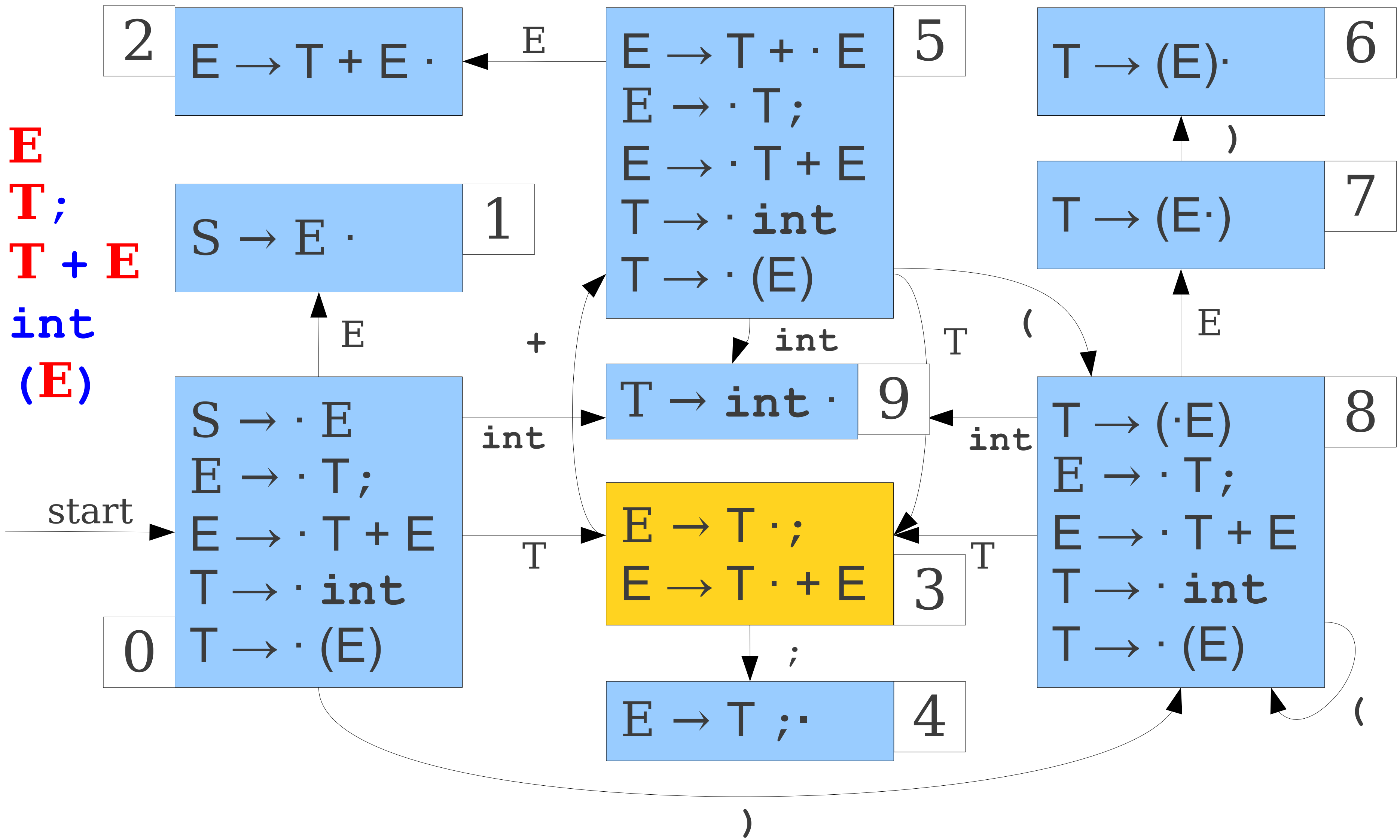
S → **E**
E → **T** ;
E → **T** + **E**
T → `int`
T → (**E**)





LR(0) Parsing

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

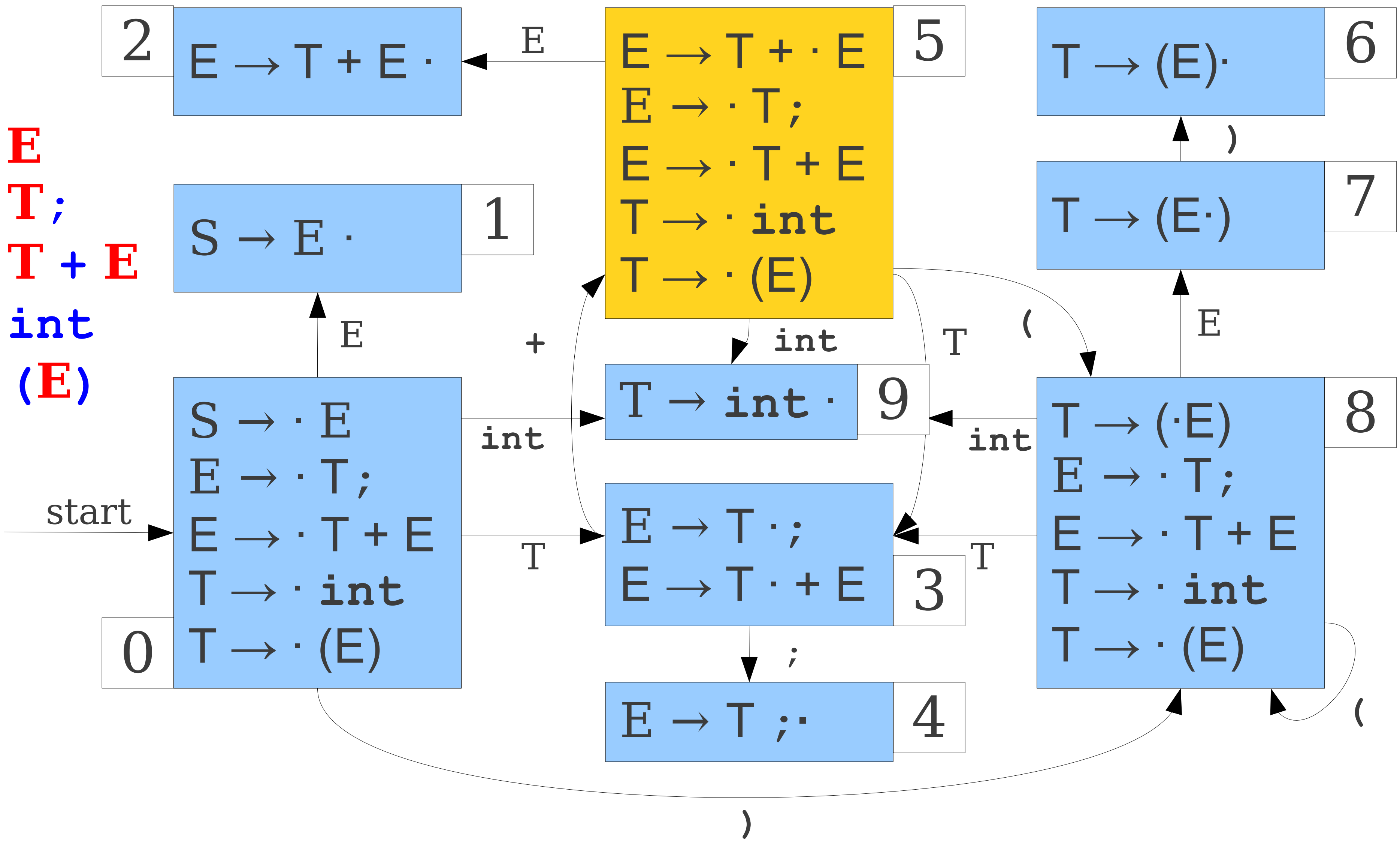


T	+	(	T	+
0	**3**	**5**	**8**	**3**

`int`	;	)	;

LR(0) Parsing

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

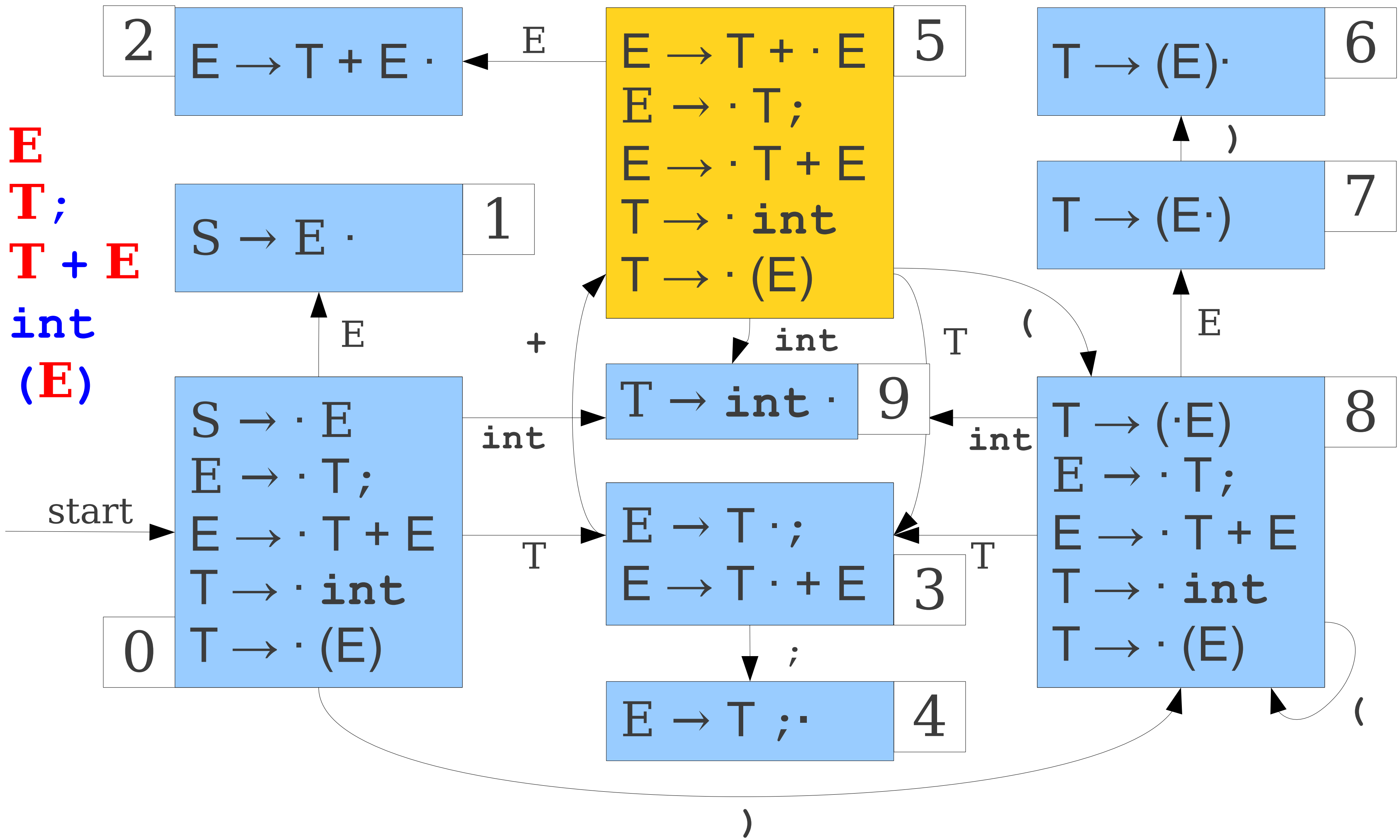


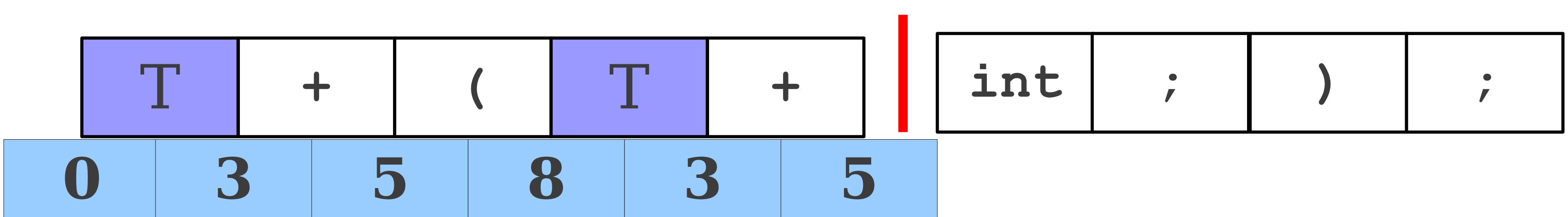
T	+	(	T	+	|	int	;	)	;

0	**3**	**5**	**8**	**3**

LR(0) Parsing

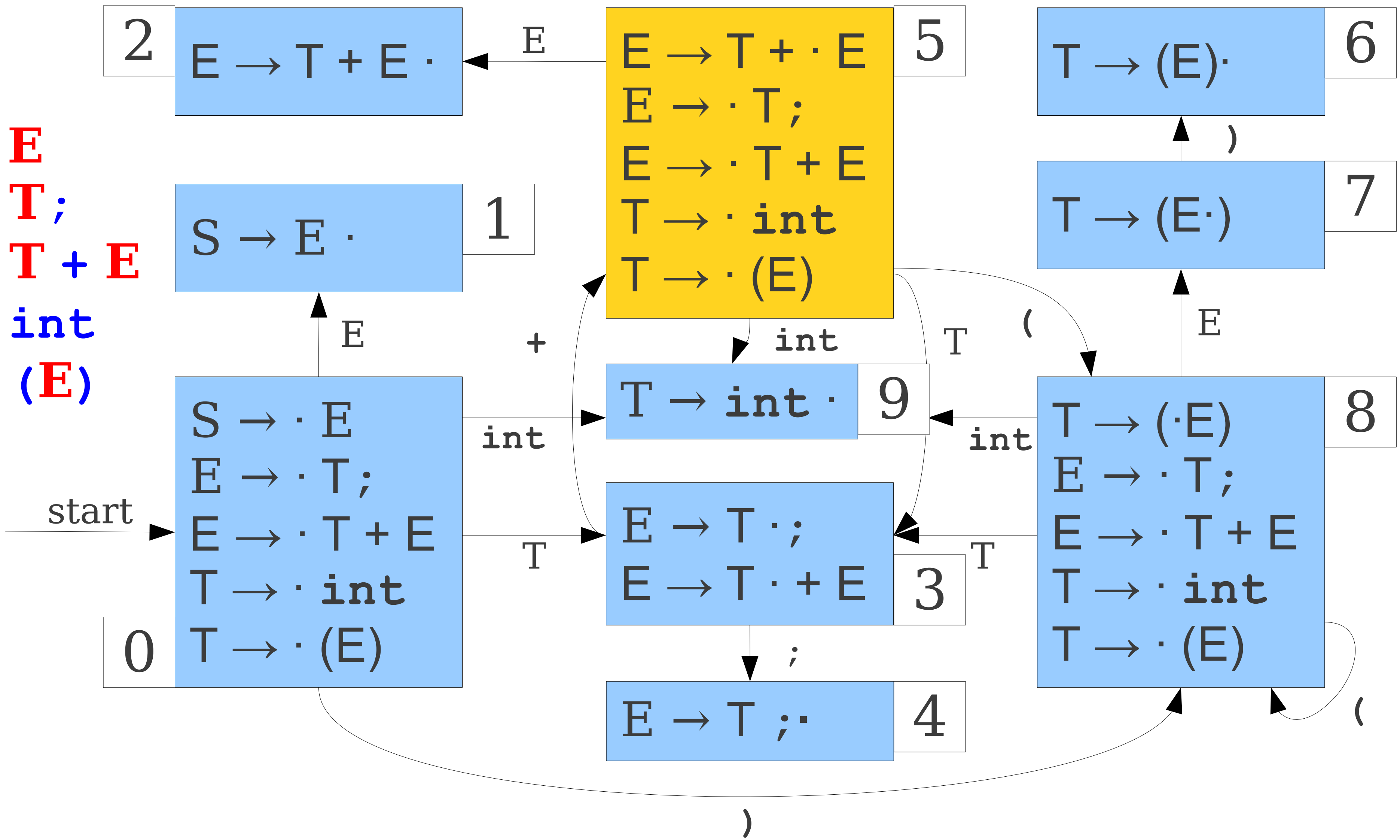
S → **E**
E → **T** ;
E → **T** + **E**
T → `int`
T → (**E**)

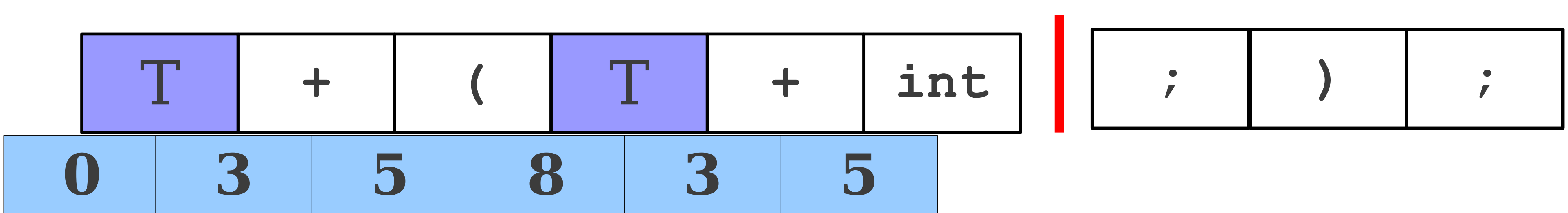




LR(0) Parsing

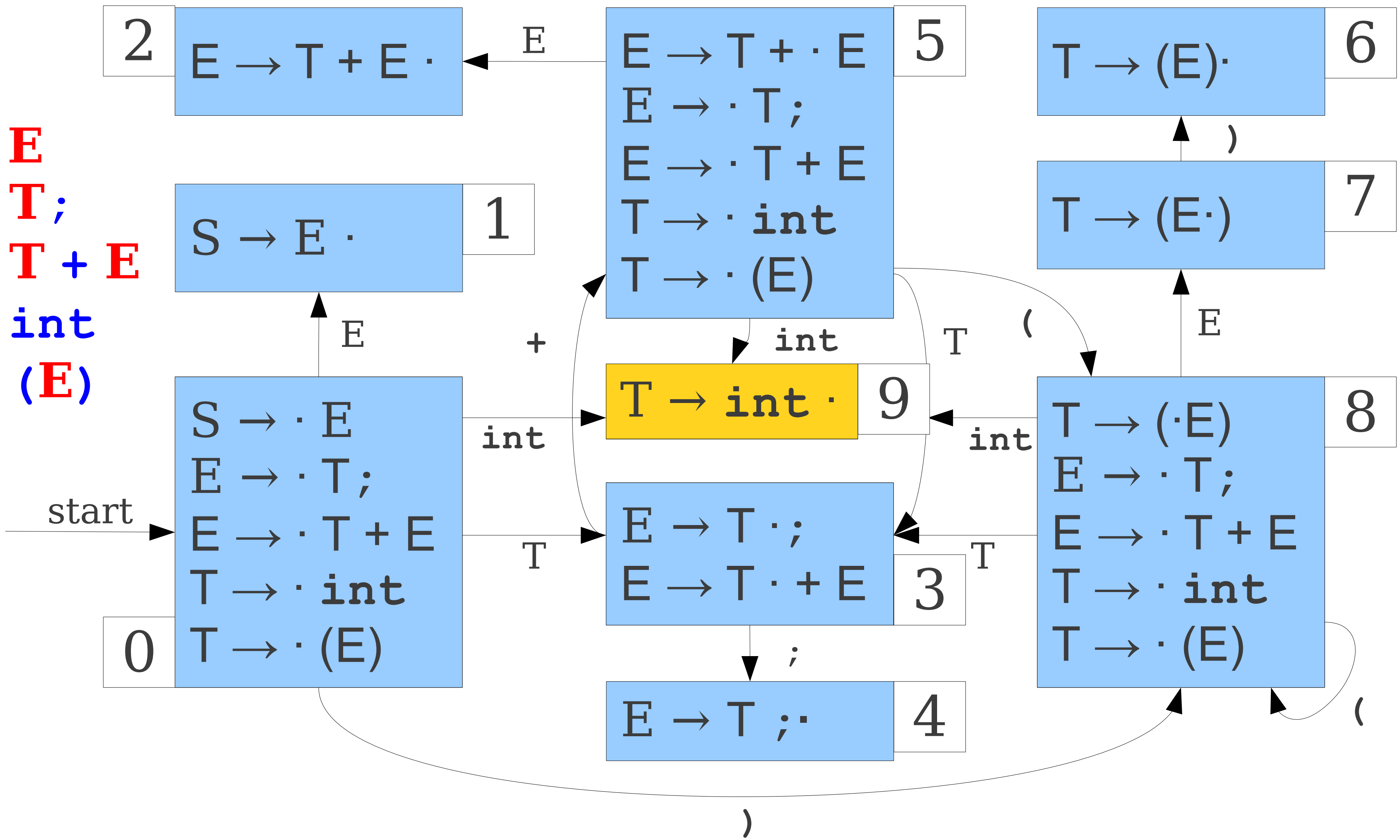
S → **E**
E → **T** ;
E → **T** + **E**
T → int
T → (**E**)





LR(0) Parsing

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



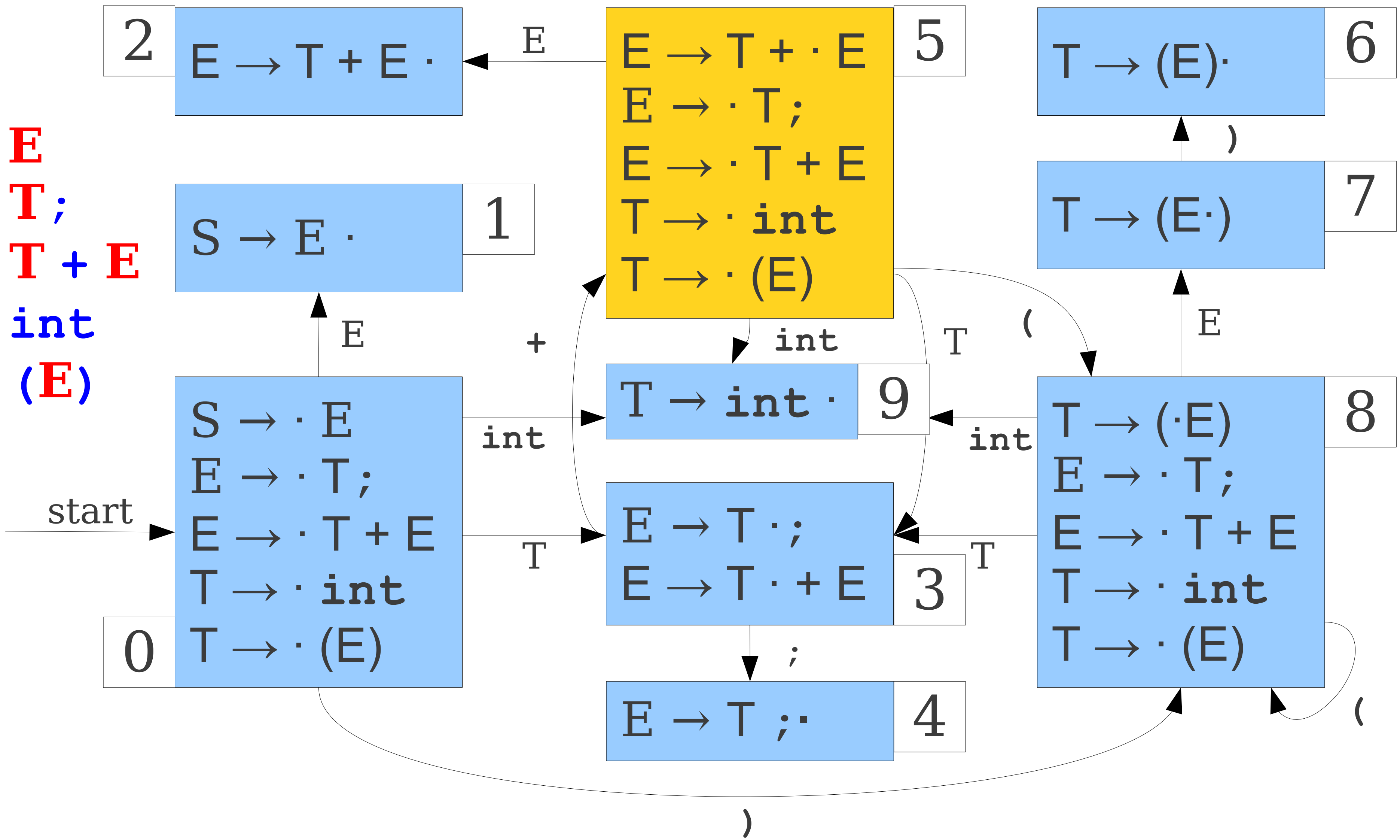
T	+	(	T	+	int

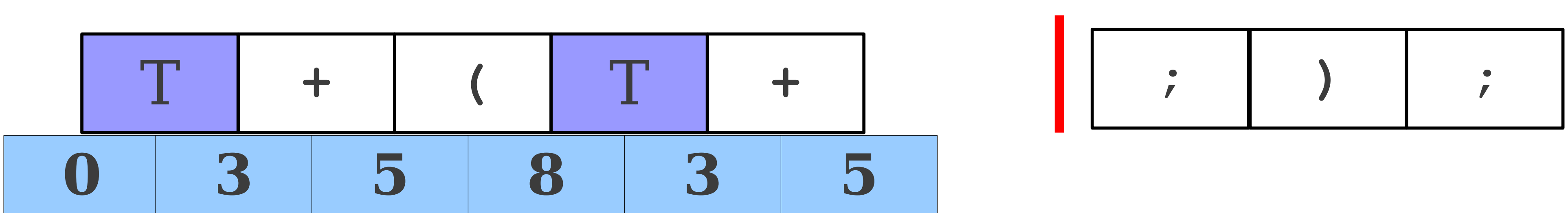
;	)	;

0	3	5	8	3	5

LR(0) Parsing

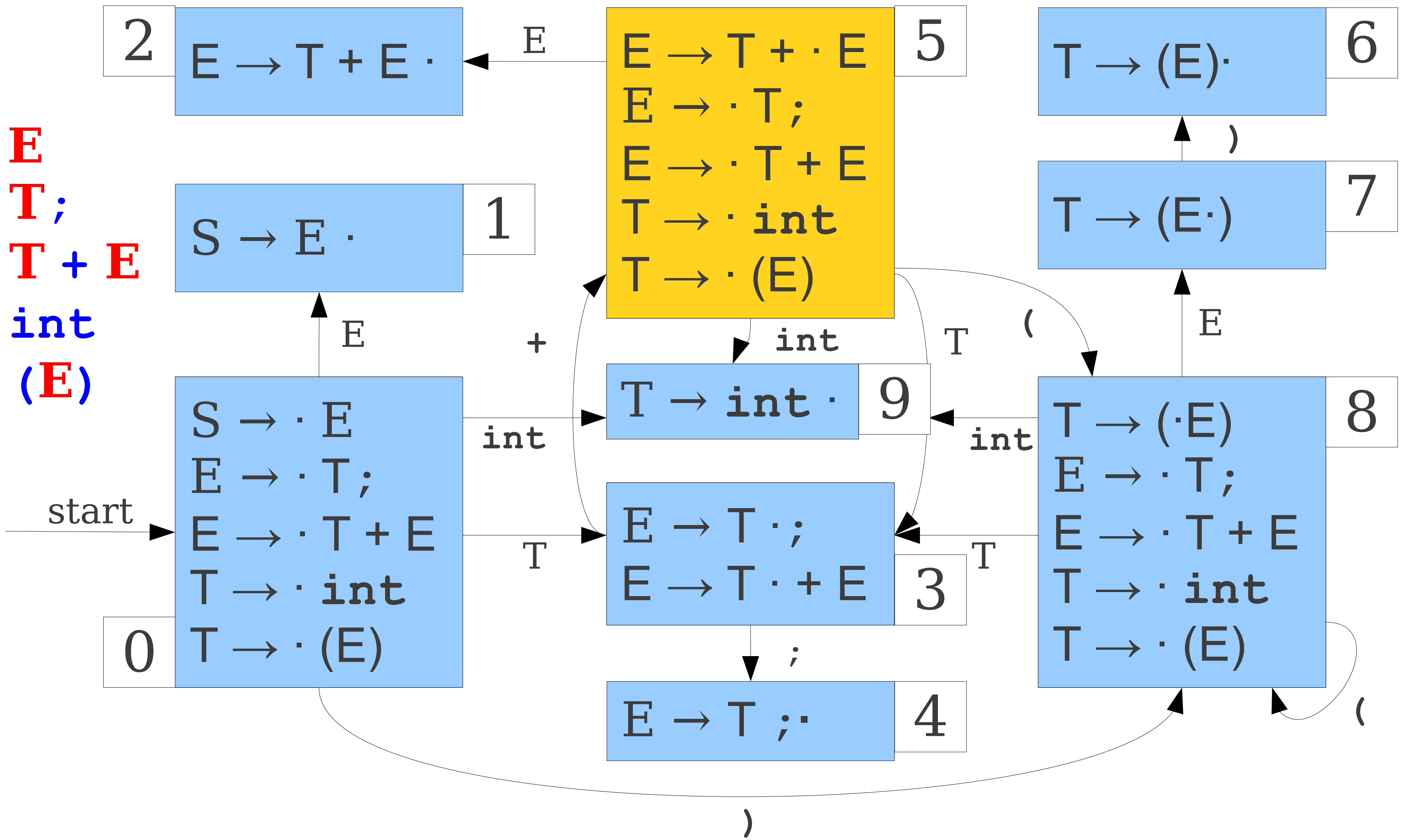
S → **E**
E → **T** ;
E → **T** + **E**
T → **int**
T → (**E**)





LR(0) Parsing

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



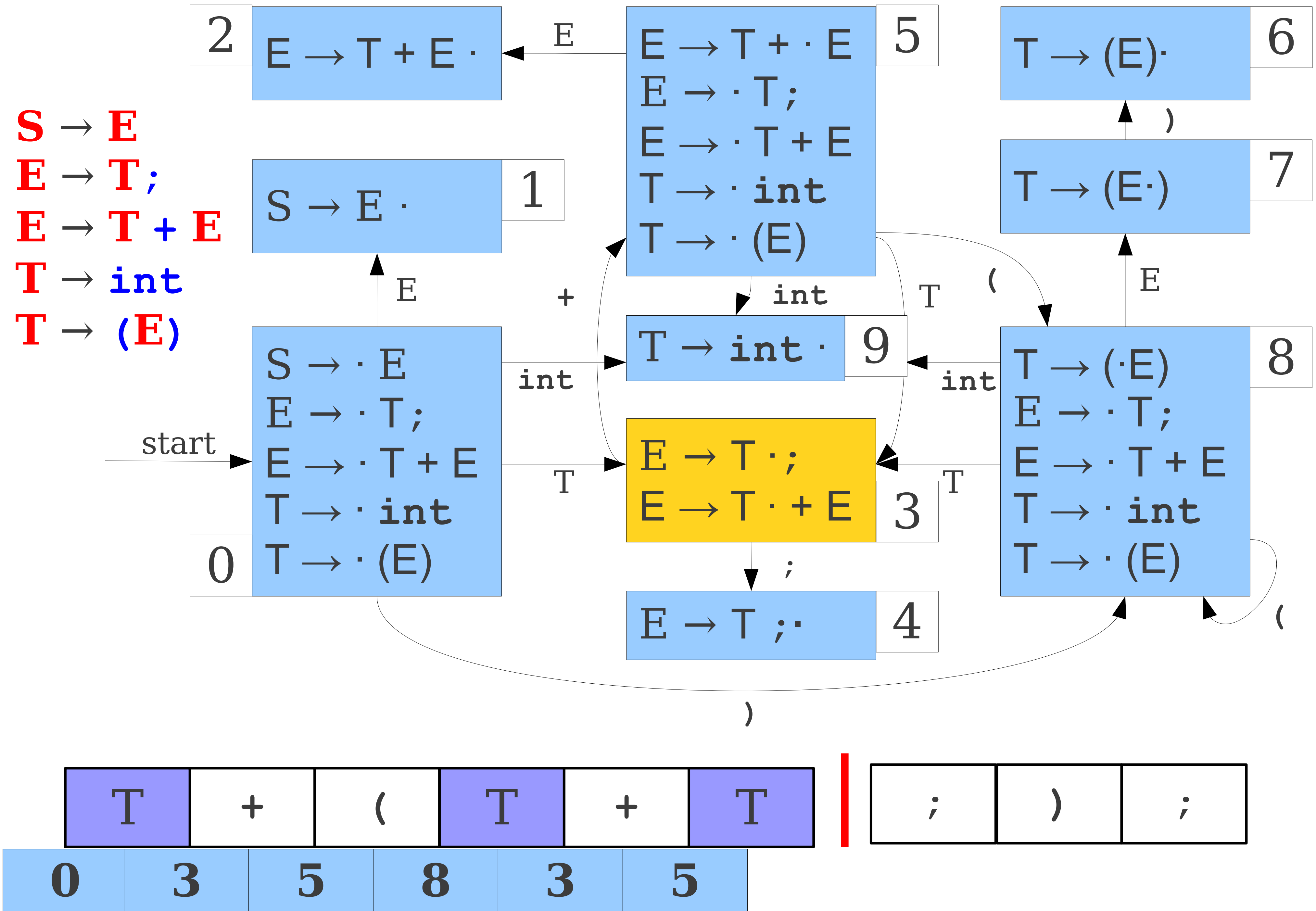
T	+	(	T	+	T

;	)	;

0	3	5	8	3	5

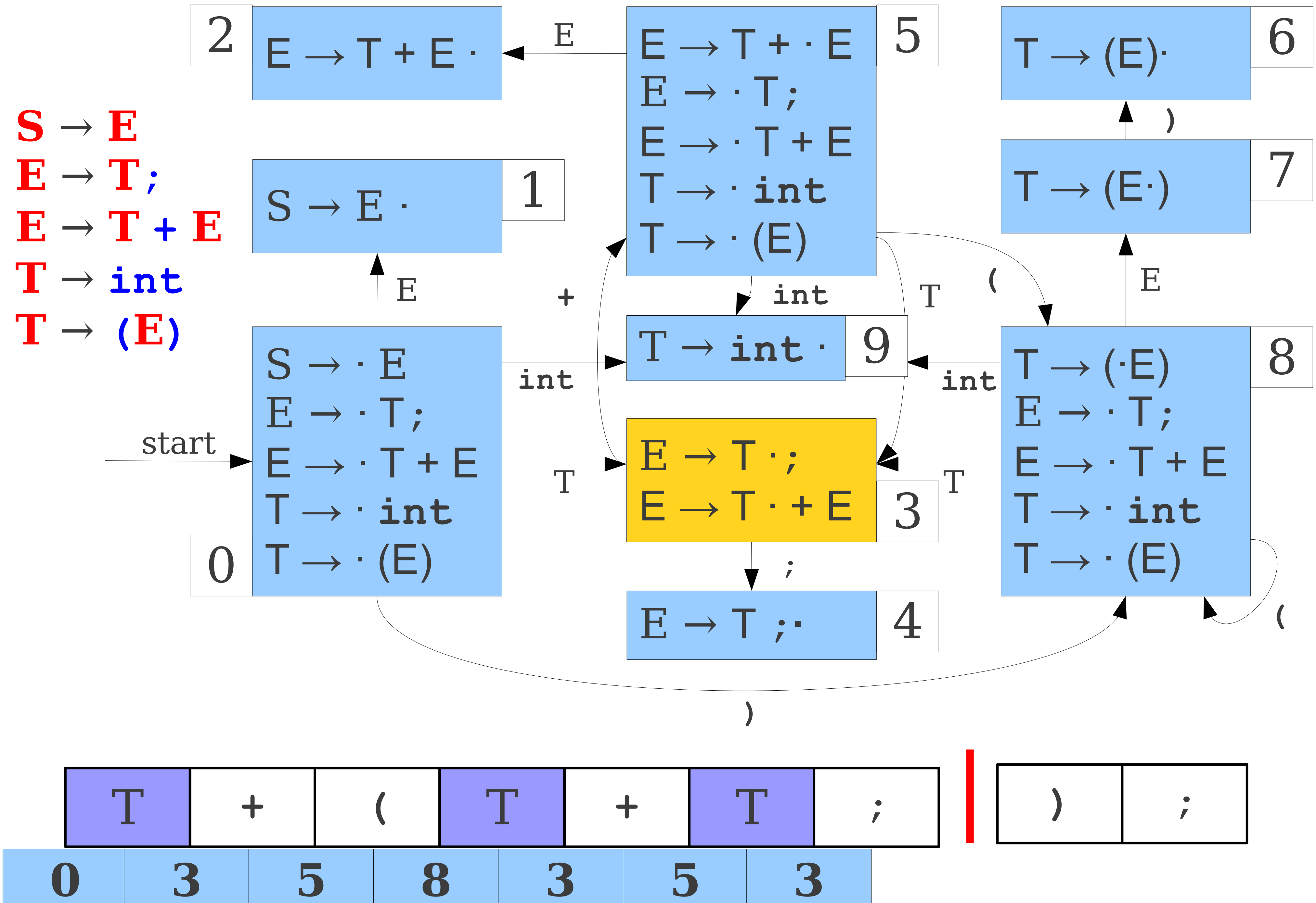
LR(0) Parsing

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



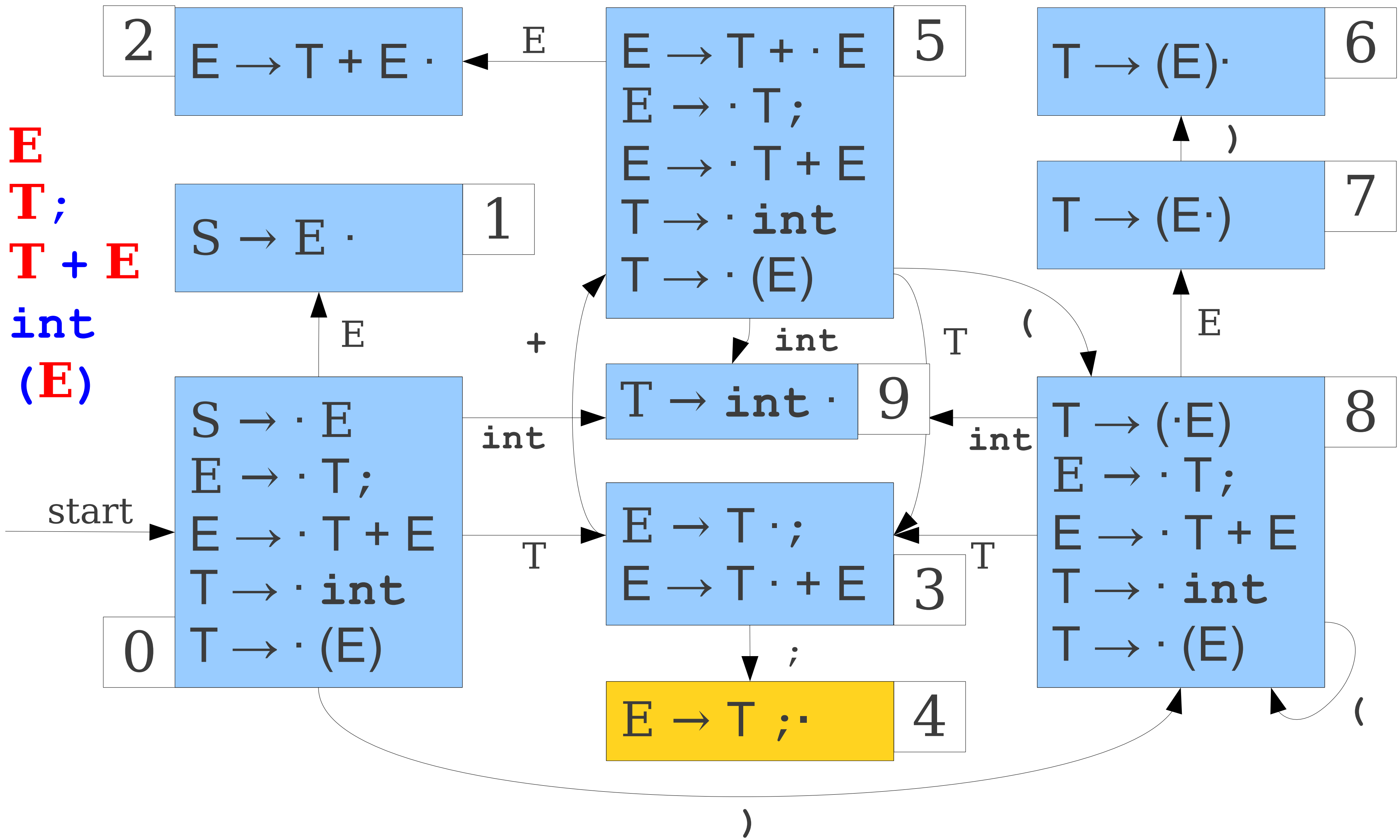
LR(0) Parsing

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**)



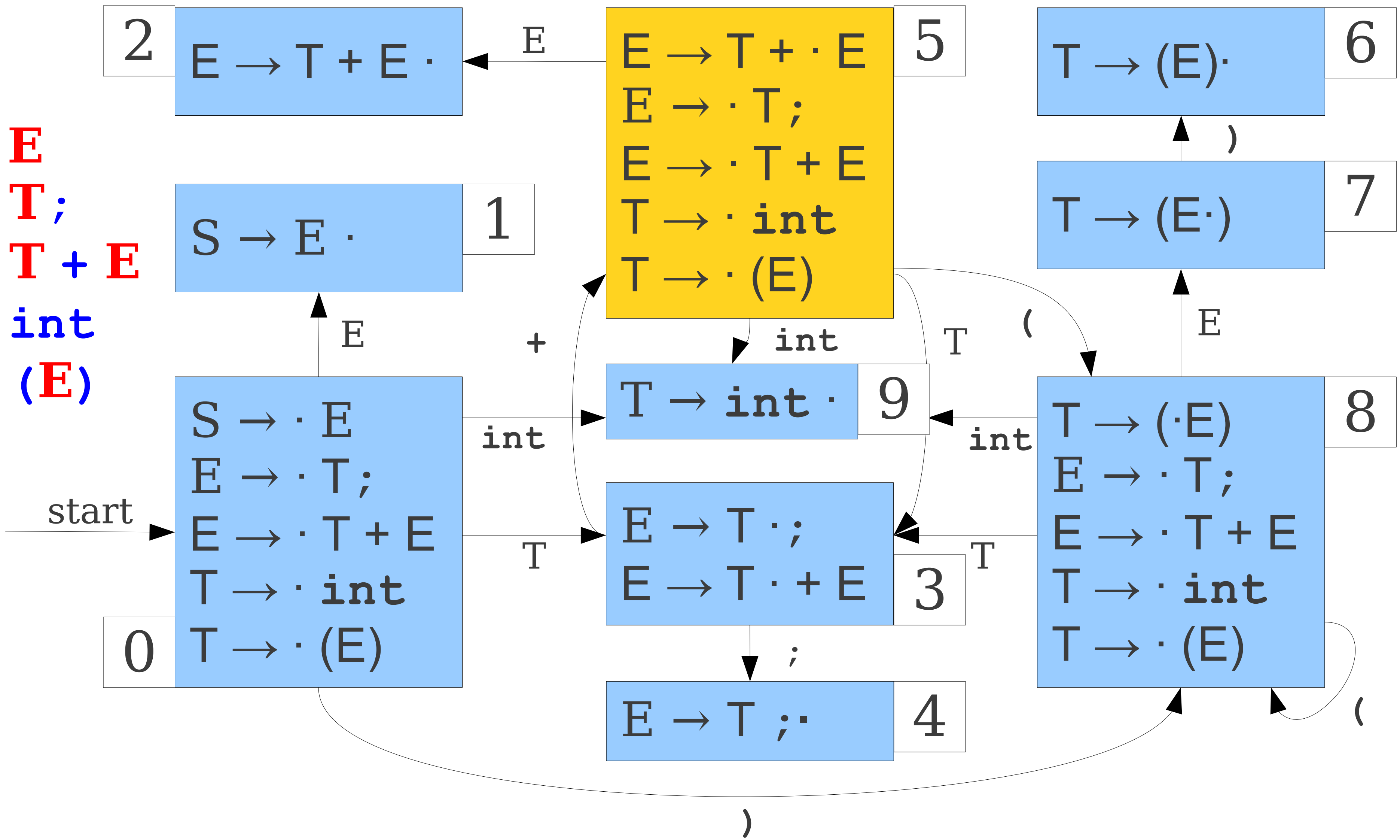
T	+	(	T	+	T	;

)	;

0	3	5	8	3	5	3

LR(0) Parsing

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



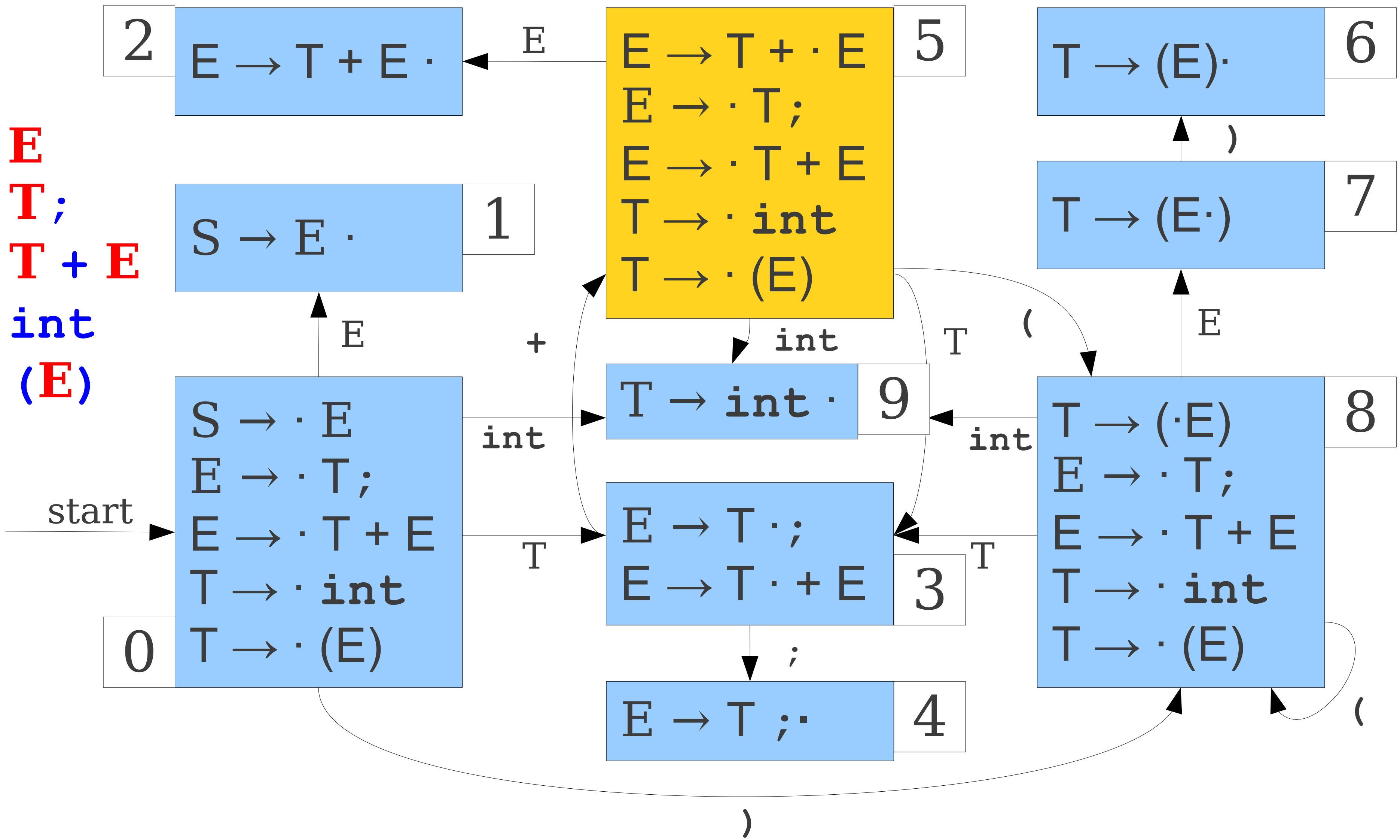
T	+	(	T	+

0	3	5	8	3	5

)	;

LR(0) Parsing

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



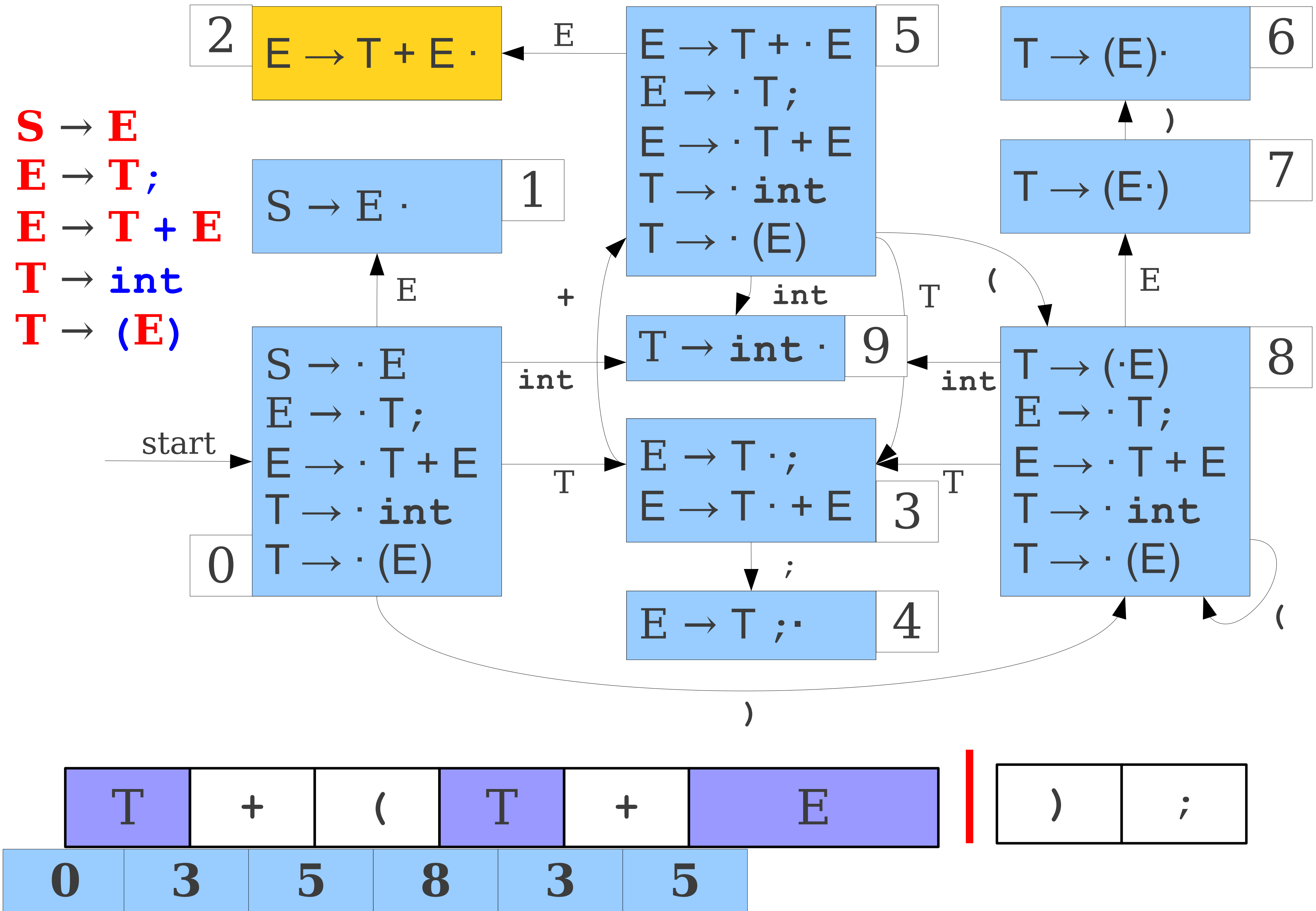
T	+	(	T	+	E

)	;

0	3	5	8	3	5

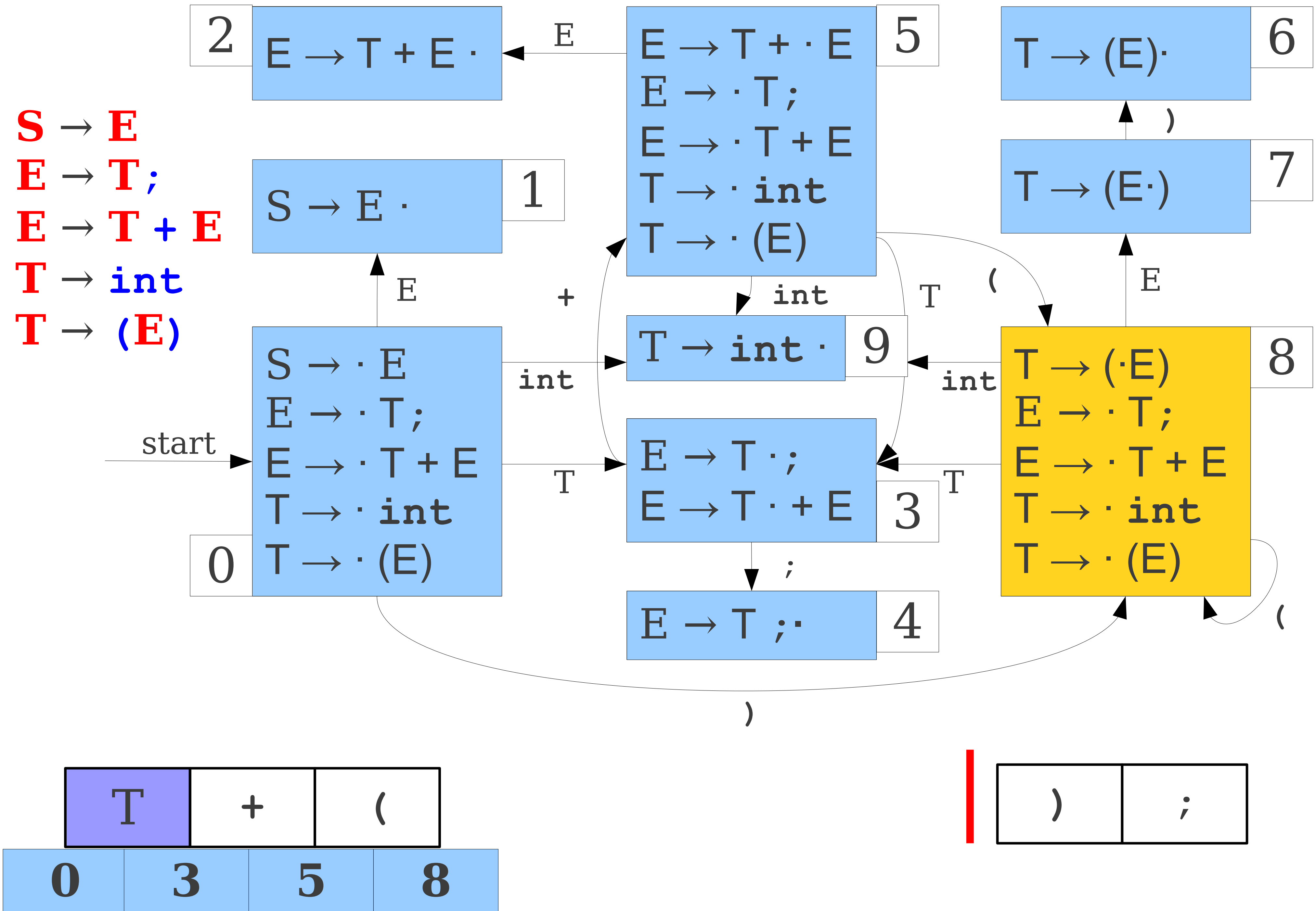
LR(0) Parsing

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



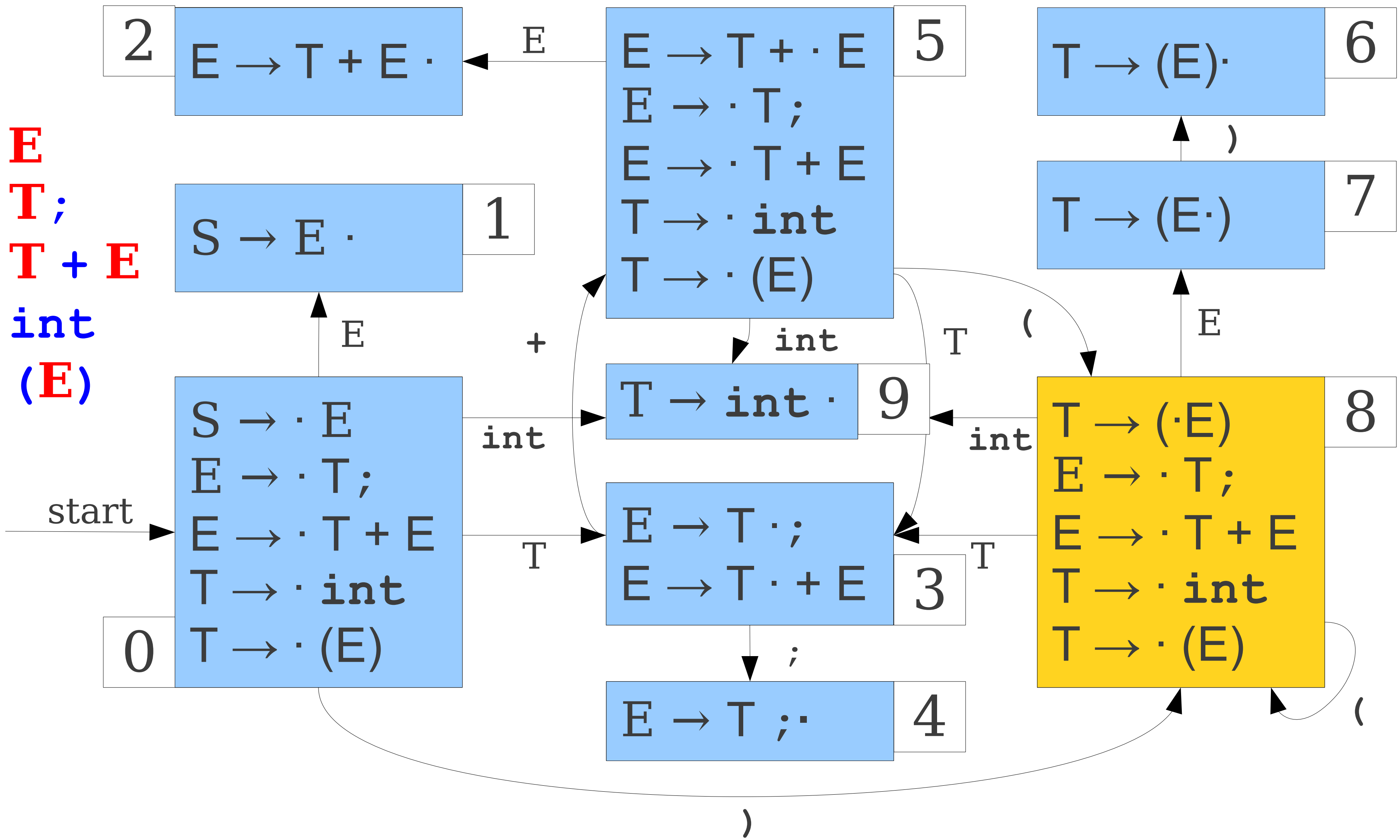
LR(0) Parsing

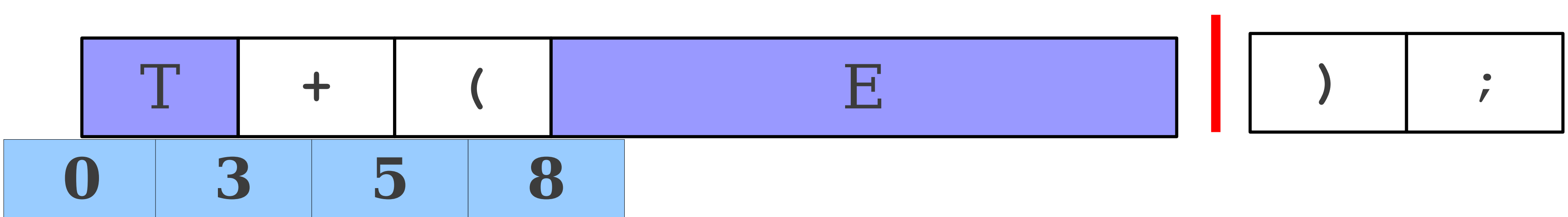
S → **E**
E → **T** ;
E → **T** + **E**
T → `int`
T → (**E**)



LR(0) Parsing

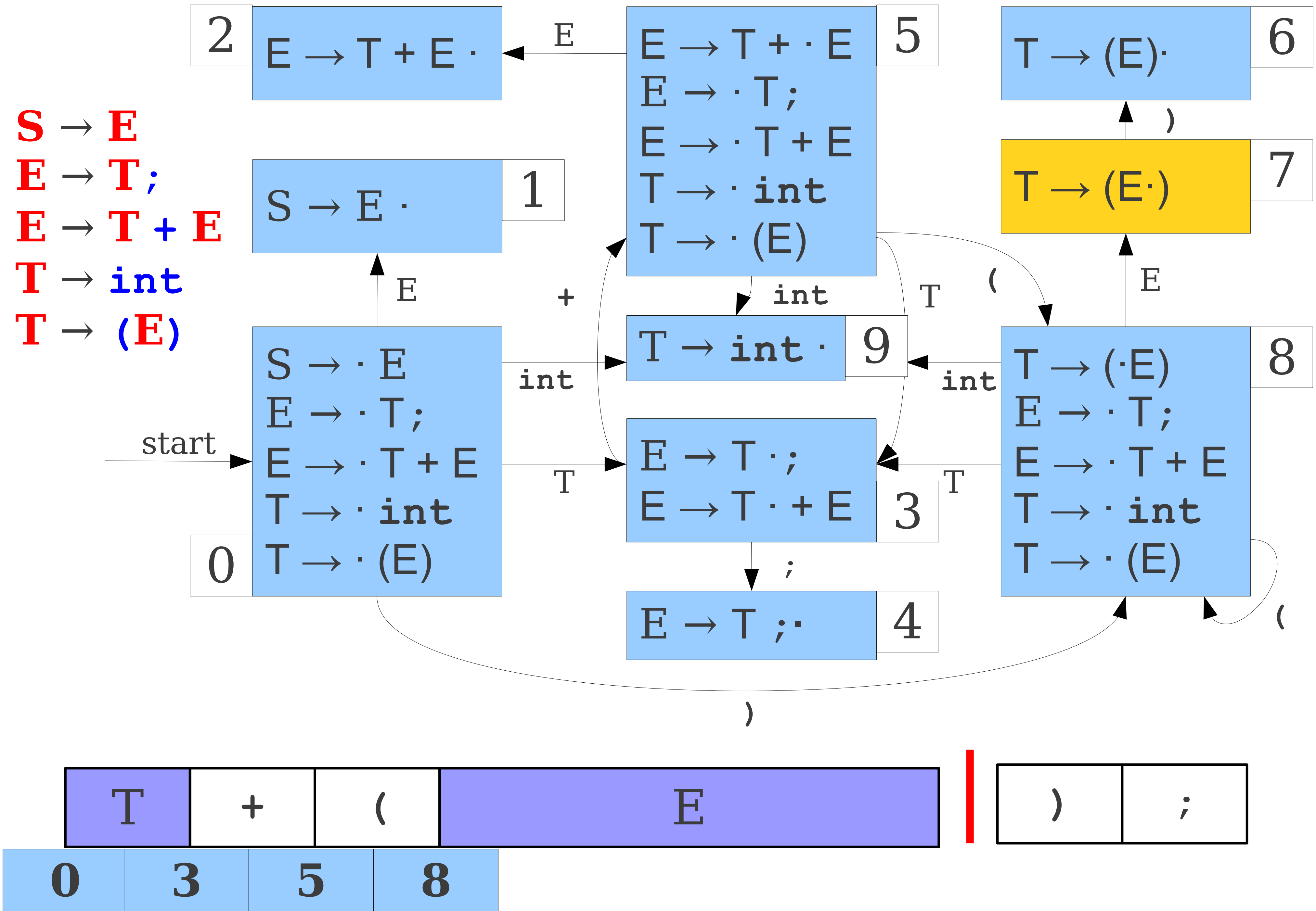
S → **E**
E → **T** **;**
E → **T** **+** **E**
T → **int**
T → **(E)**





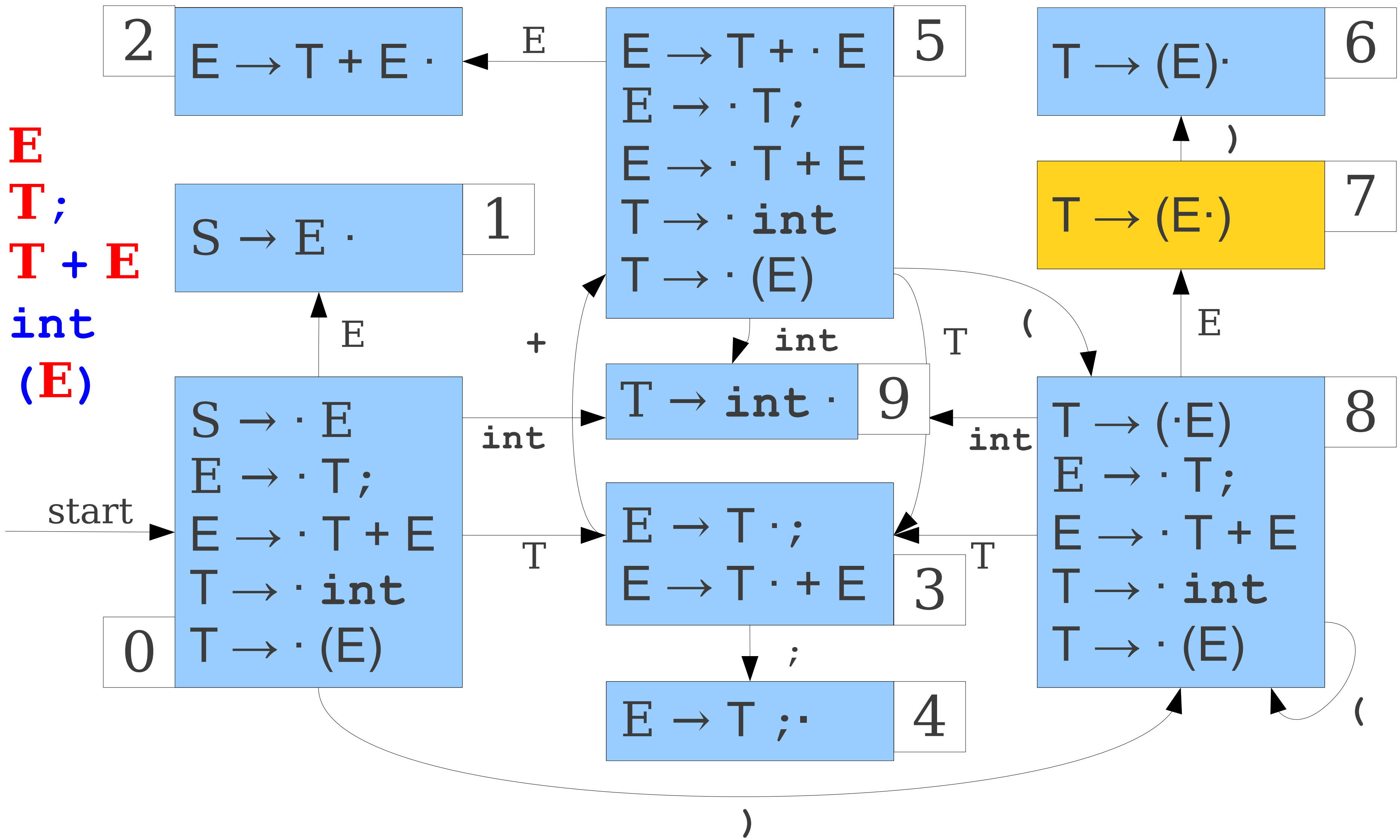
LR(0) Parsing

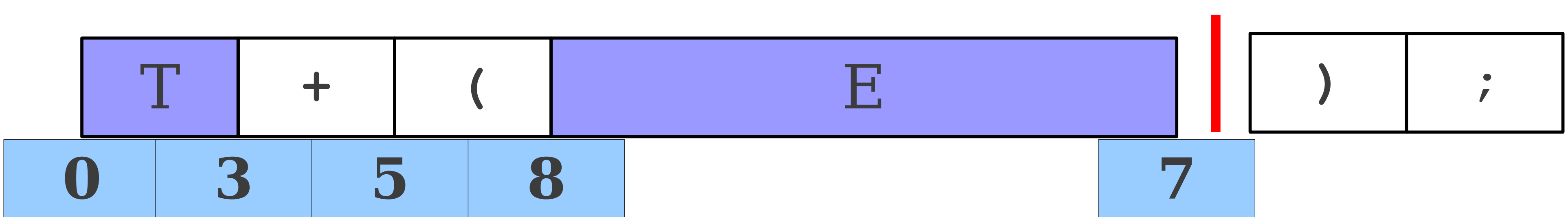
$S \to E$
$E \to T;$
$E \to T + E$
$T \to$ int
$T \to (E)$



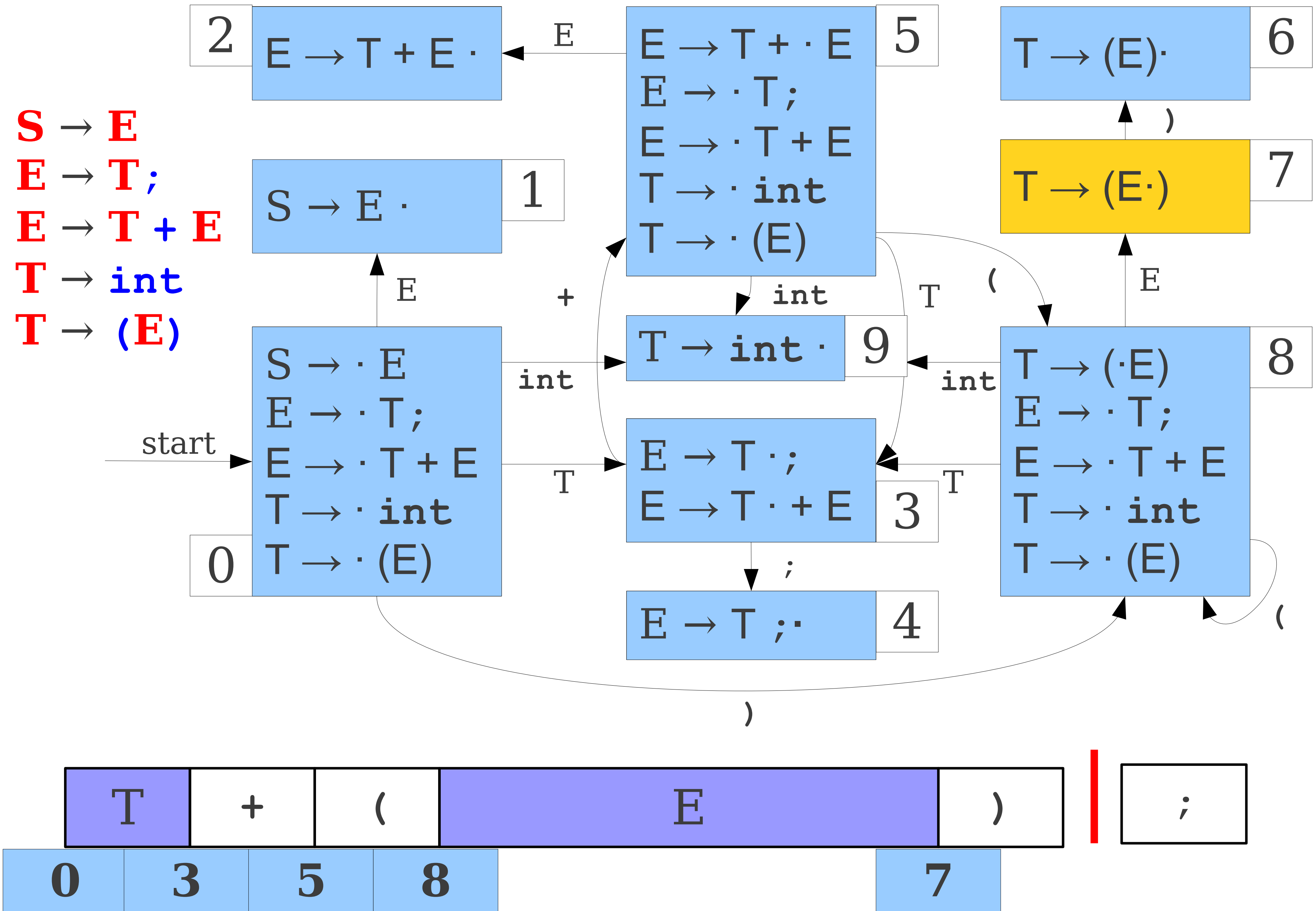
LR(0) Parsing

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



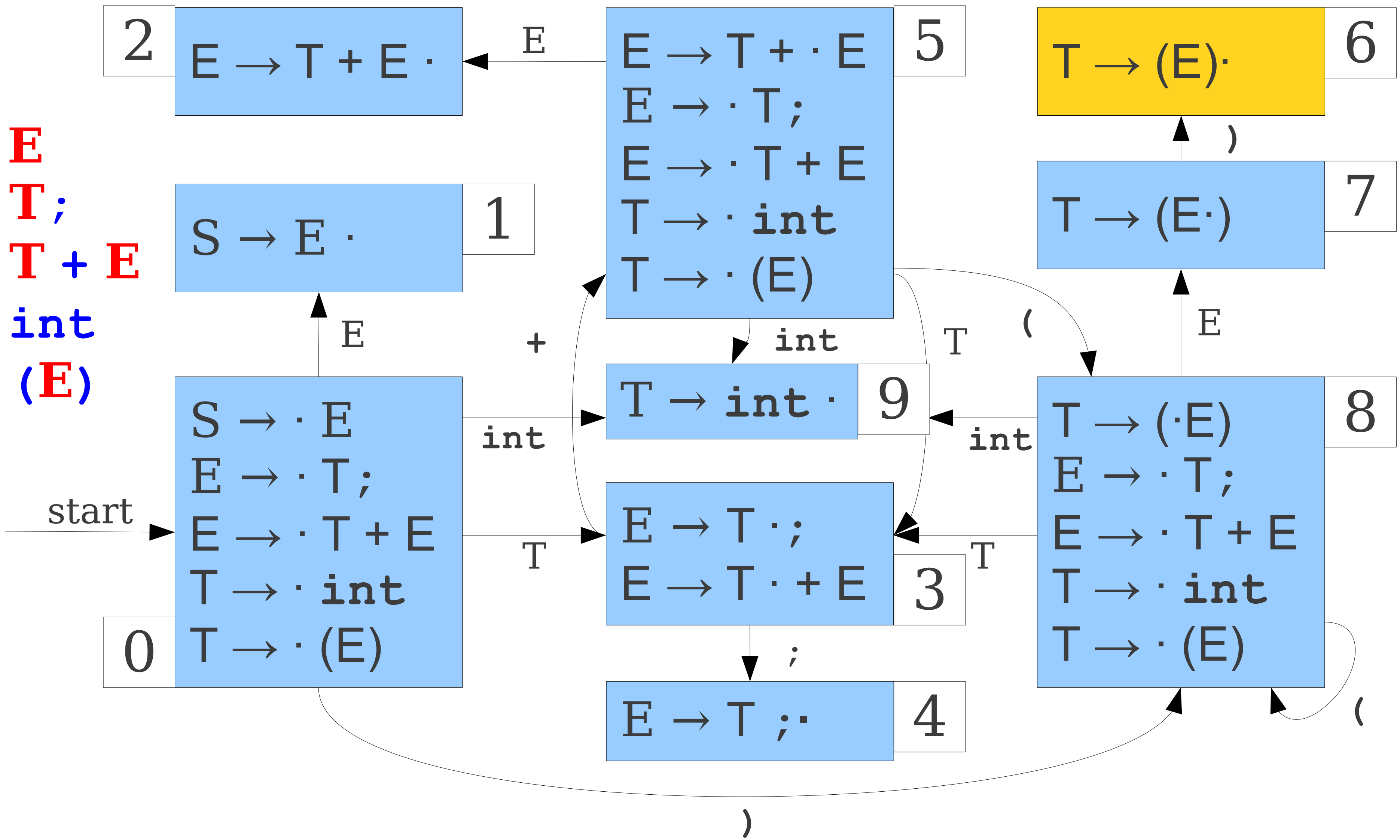


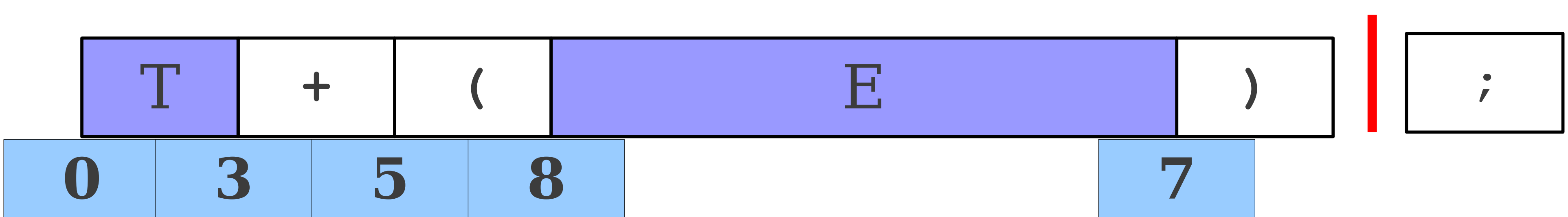
LR(0) Parsing


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

LR(0) Parsing

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





LR(0) Parsing

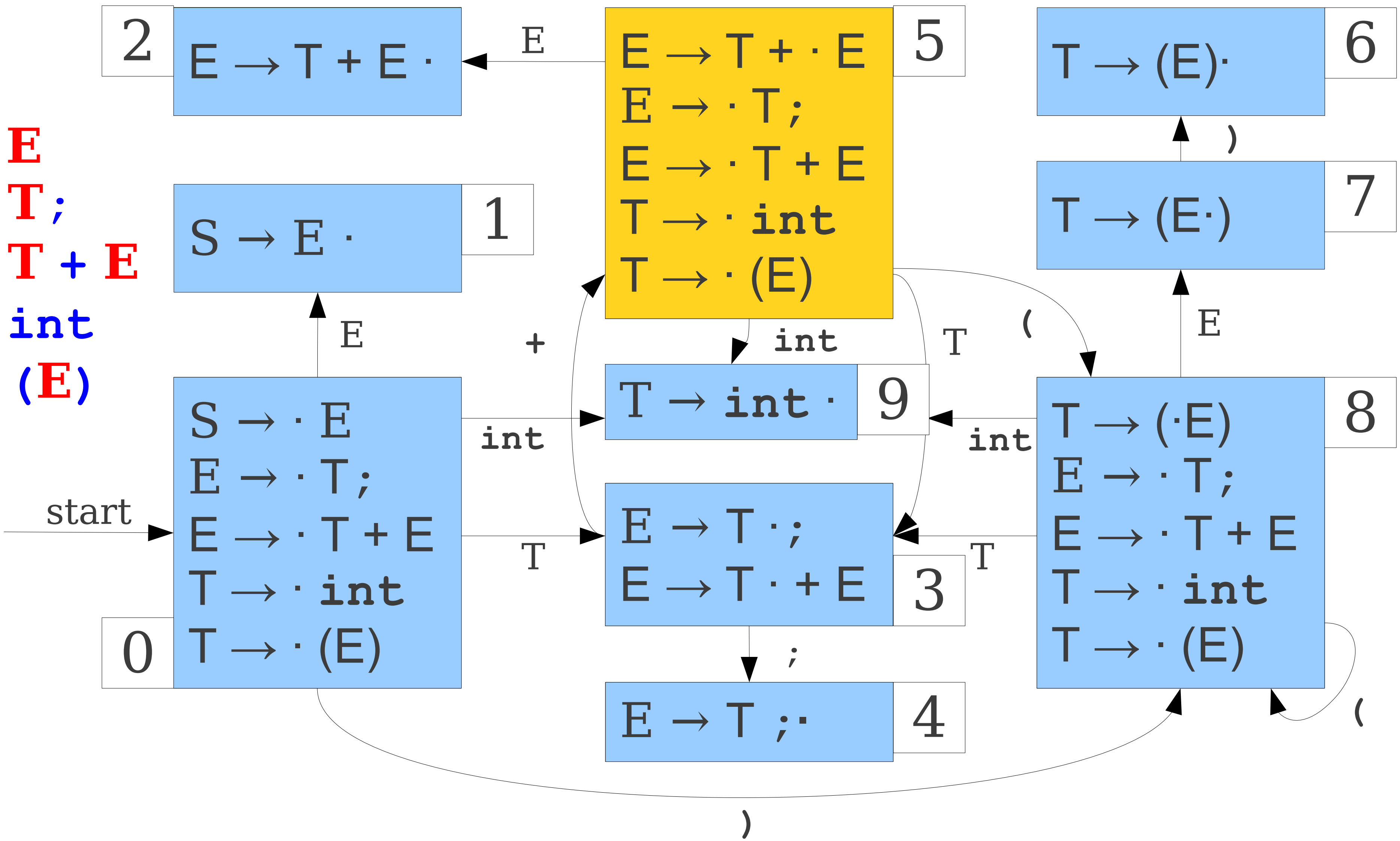
$\mathbf{S} \rightarrow \mathbf{E}$

$\mathbf{E} \rightarrow \mathbf{T}$ `;`

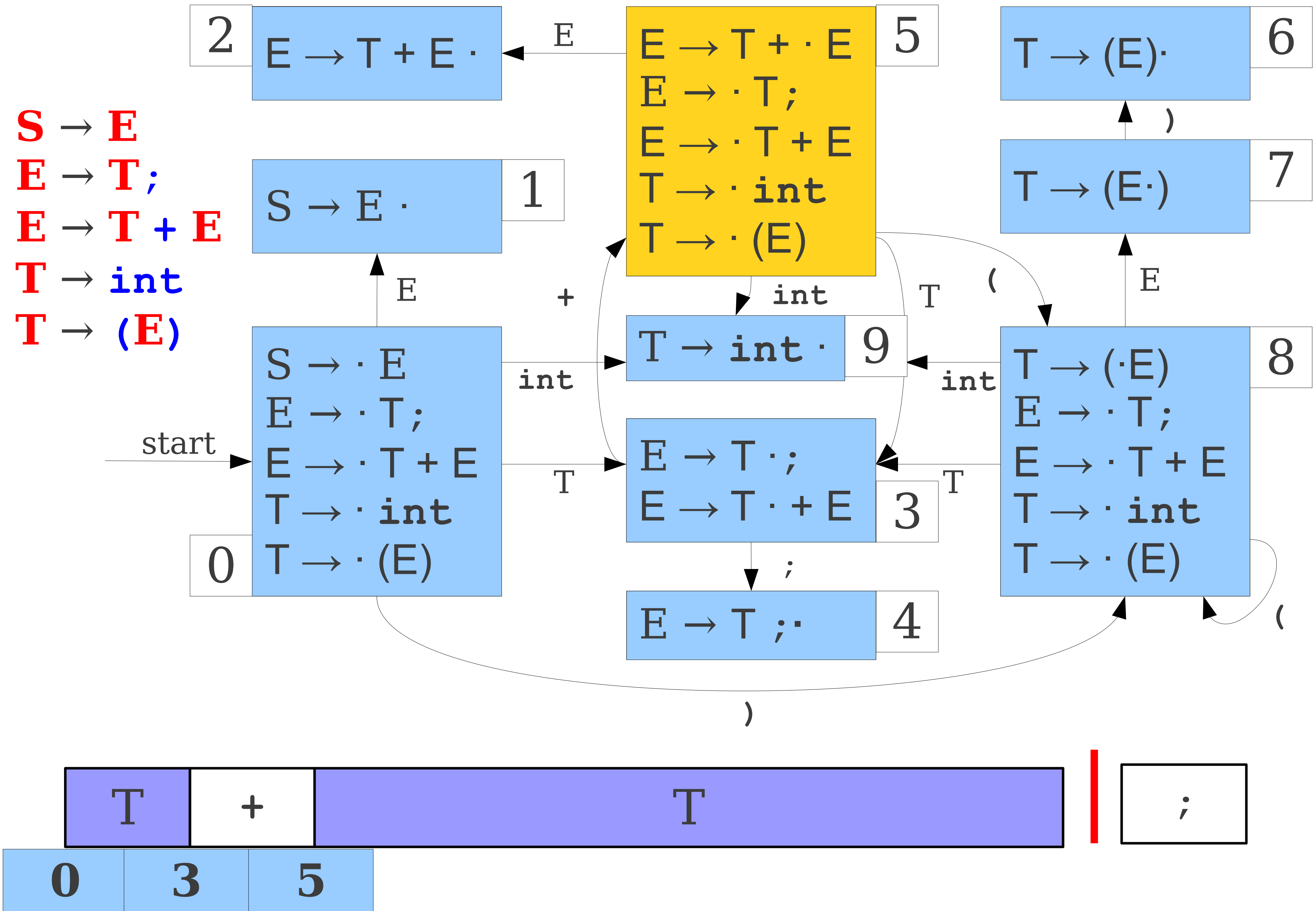
$\mathbf{E} \rightarrow \mathbf{T} + \mathbf{E}$

$\mathbf{T} \rightarrow$ `int`

$\mathbf{T} \rightarrow$ `(`$\mathbf{E}$`)`



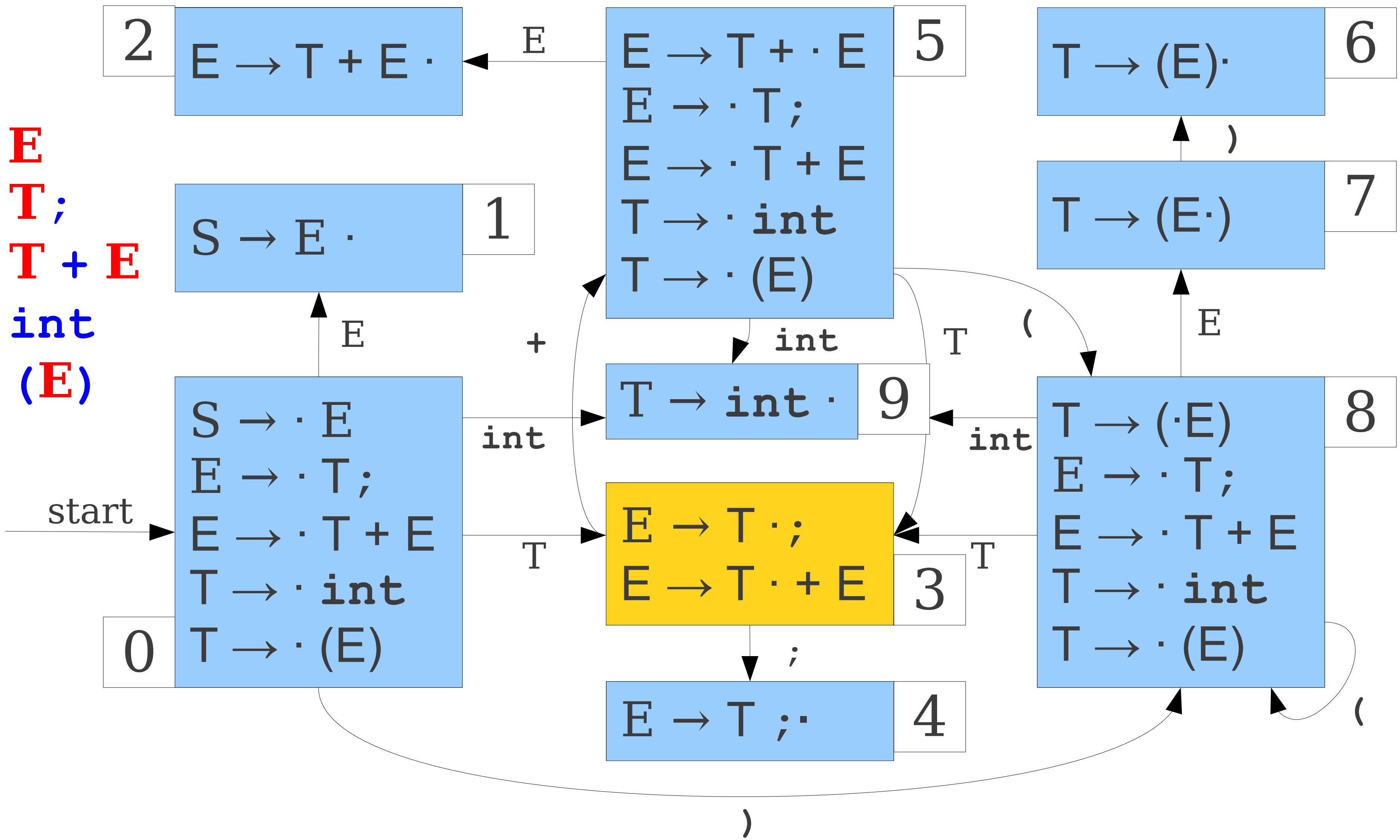
LR(0) Parsing

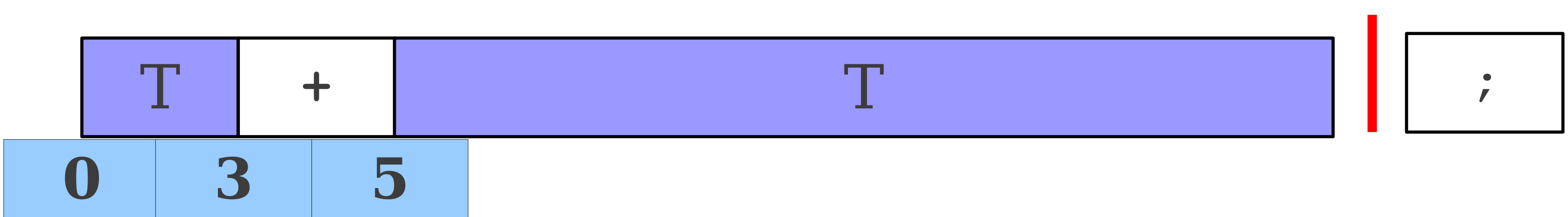


S → E
E → T;
E → T + E
T → int
T → (E)
2
E → T + E ·
E
5
E → T + · E
E → · T;
E → · T + E
T → · int
T → · (E)
6
T → (E)·
)
7
T → (E·)
1
S → E ·
E
+
int
T
(
E
9
T → int ·
int
int
start
0
S → · E
E → · T;
E → · T + E
T → · int
T → · (E)
T
3
E → T ·;
E → T · + E
T
8
T → (·E)
E → · T;
E → · T + E
T → · int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

LR(0) Parsing

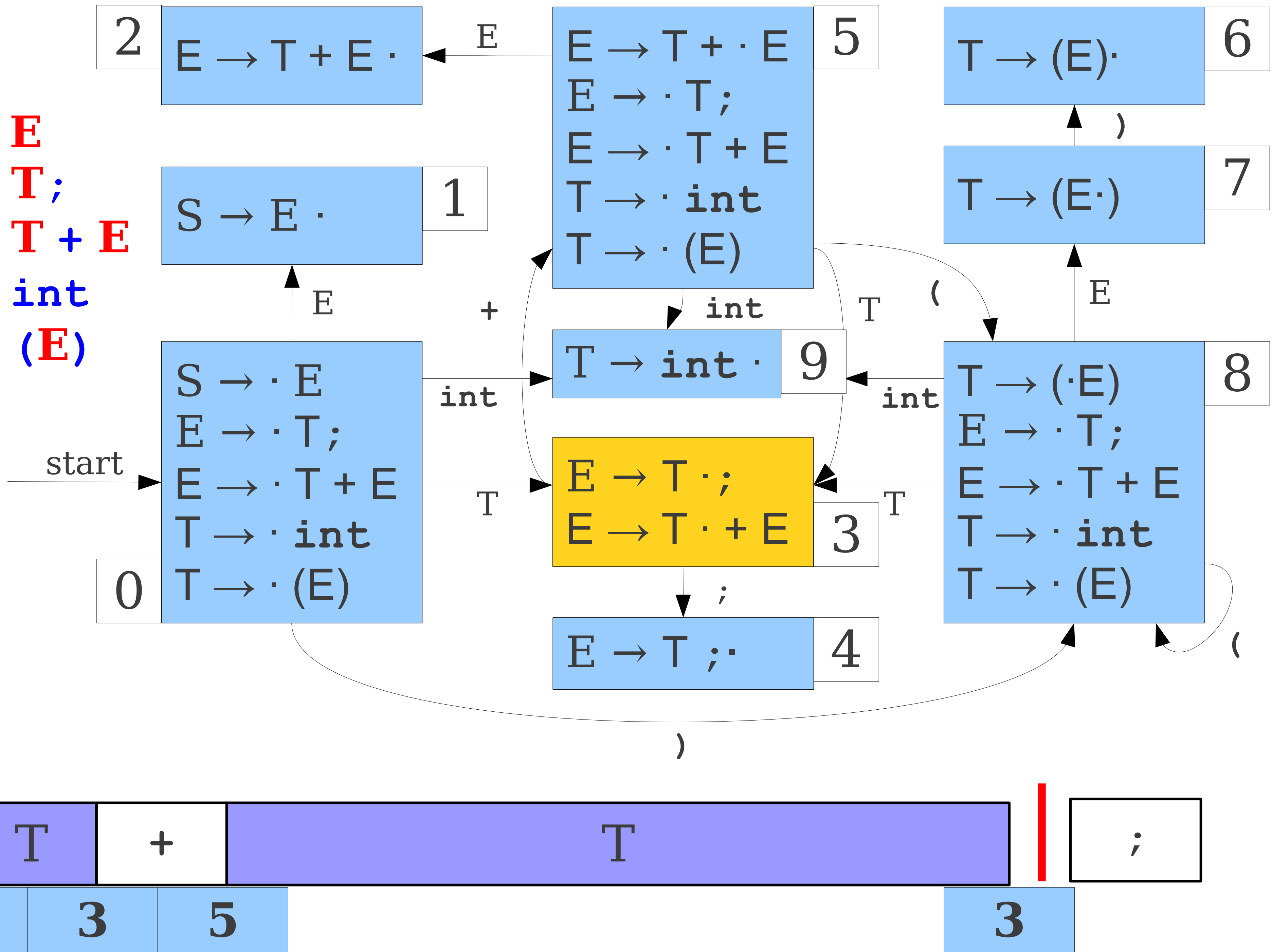
S → **E**
E → **T** ;
E → **T** + **E**
T → **int**
T → **(E)**



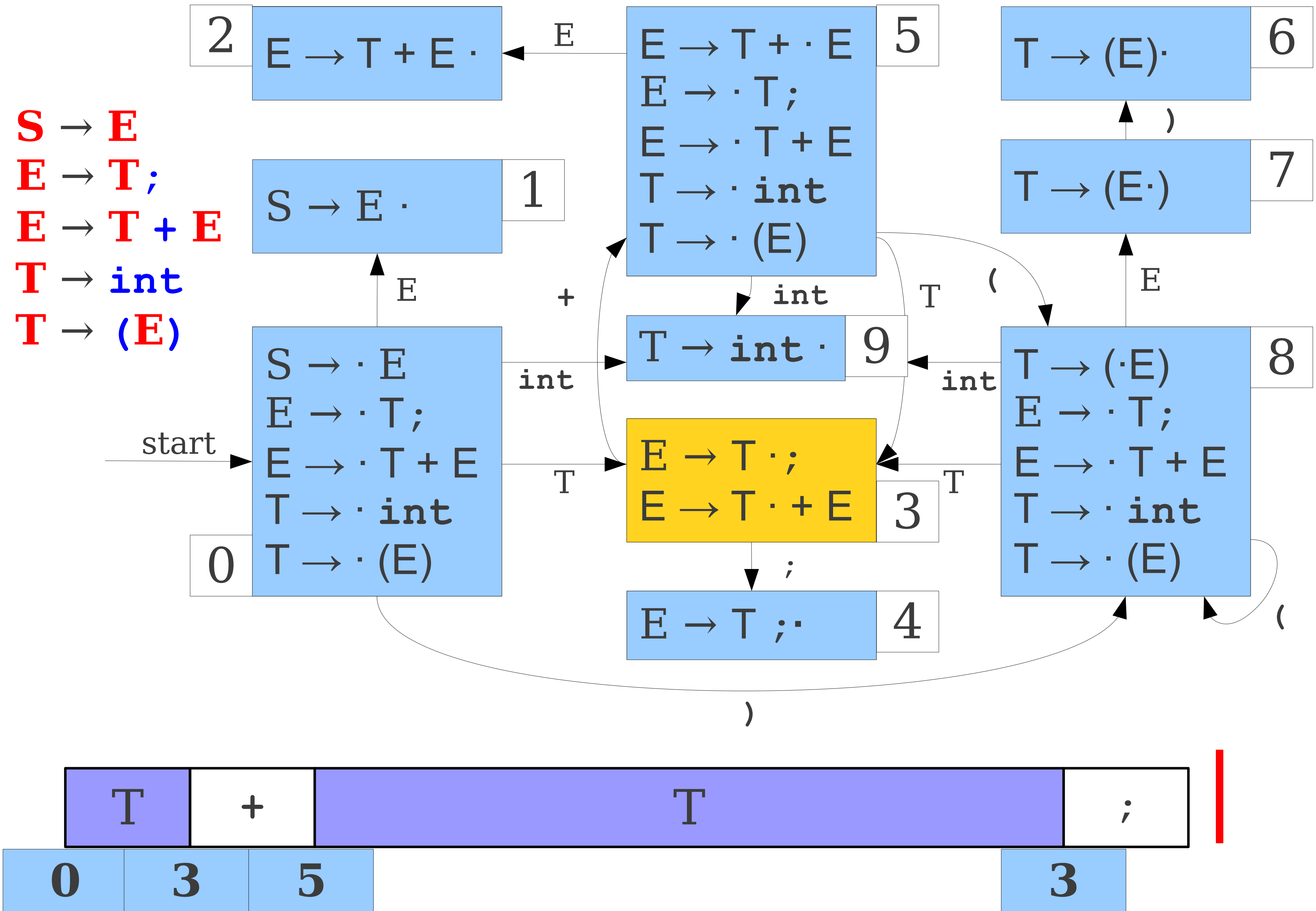


LR(0) Parsing

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



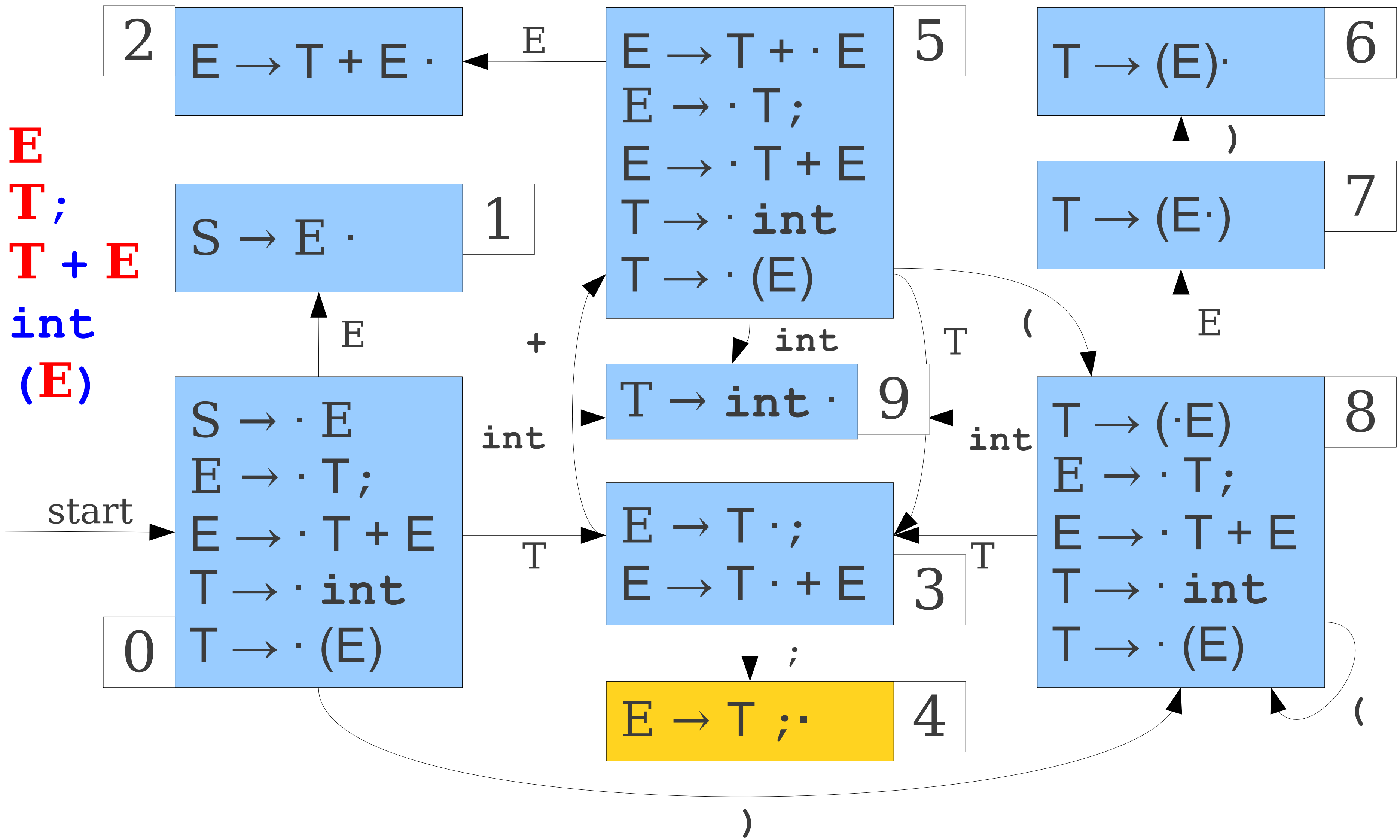
LR(0) Parsing

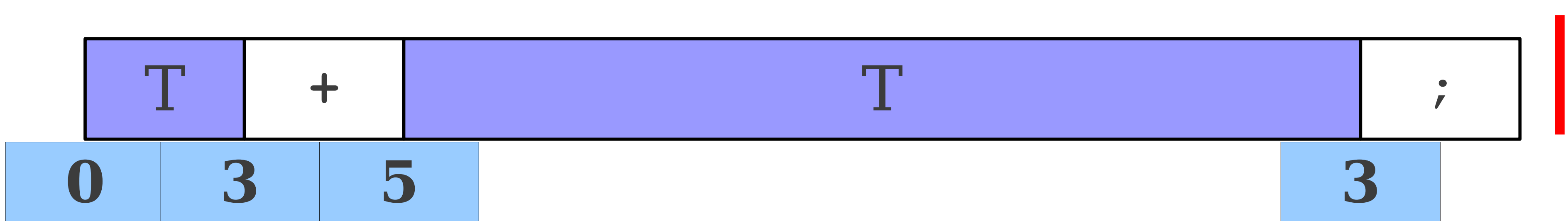


S → E
E → T;
E → T + E
T → int
T → (E)
2
E → T + E ·
E
5
E → T + · E
E → · T;
E → · T + E
T → · int
T → · (E)
6
T → (E)·
)
7
T → (E·)
1
S → E ·
E
+
int
T
(
E
9
T → int ·
int
int
start
0
S → · E
E → · T;
E → · T + E
T → · int
T → · (E)
T
3
E → T ·;
E → T · + E
T
8
T → (·E)
E → · T;
E → · T + E
T → · int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

LR(0) Parsing

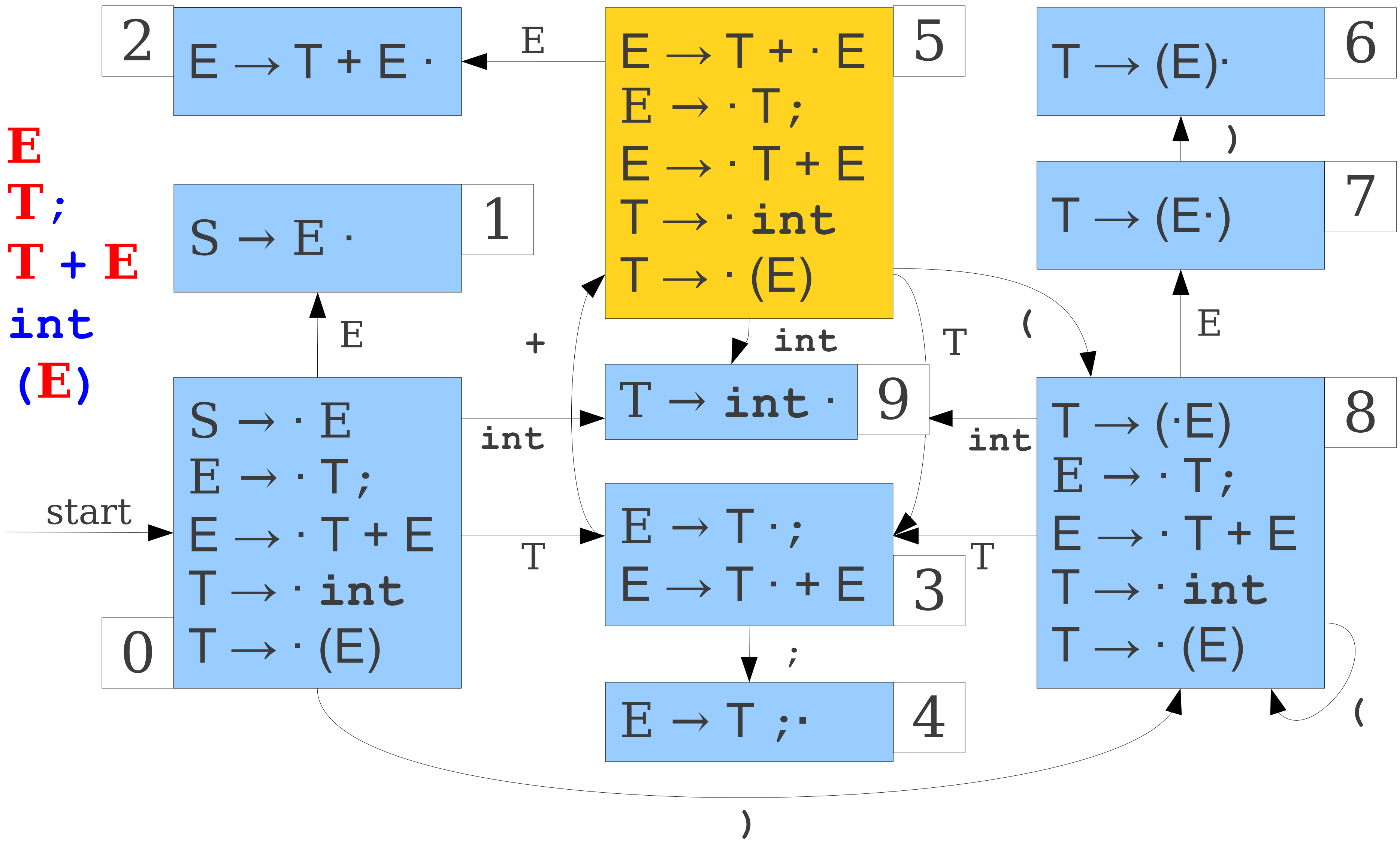
S → **E**
E → **T** ;
E → **T** + **E**
T → int
T → (**E**)



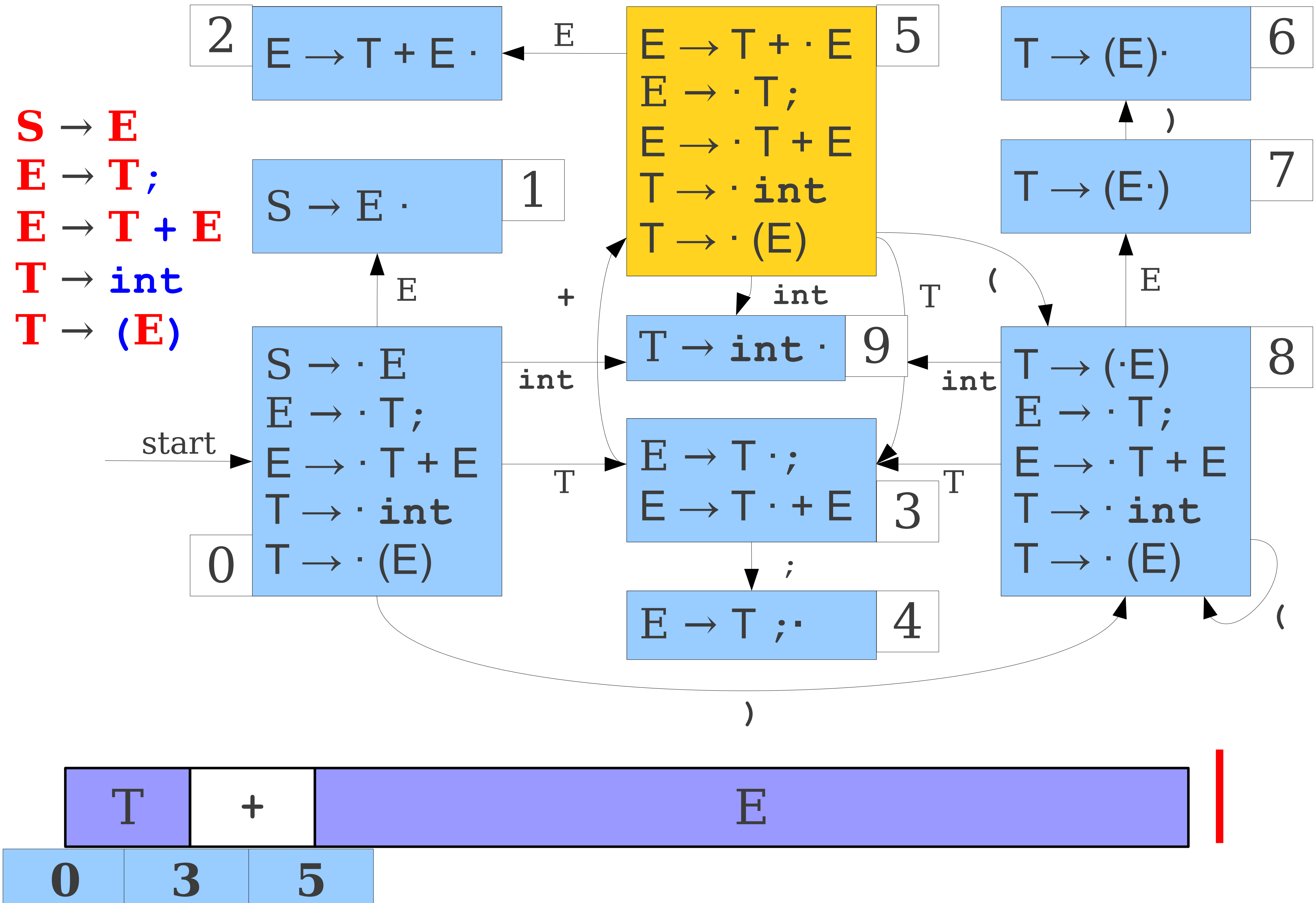


LR(0) Parsing

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

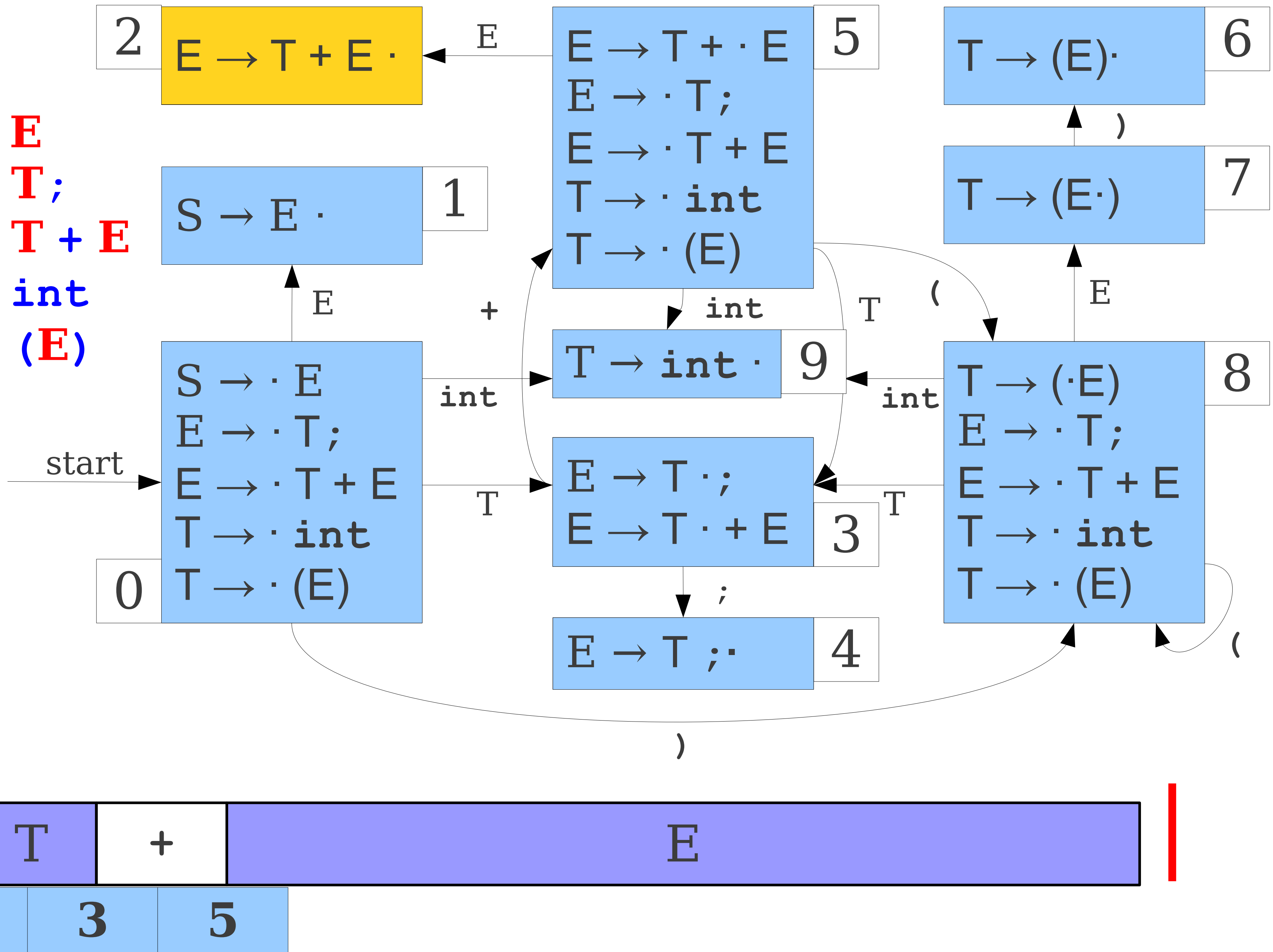


LR(0) Parsing


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

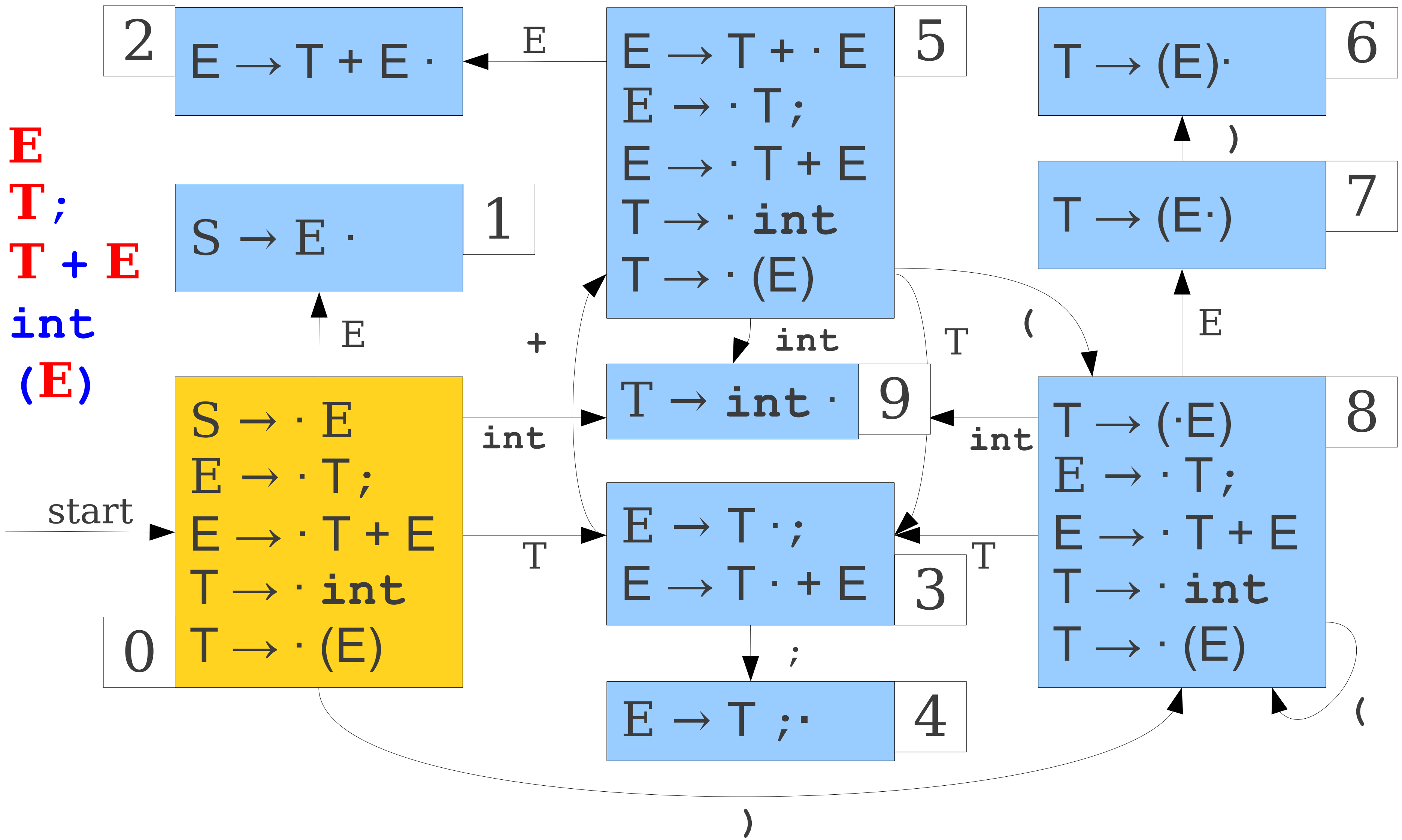
LR(0) Parsing

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



LR(0) Parsing

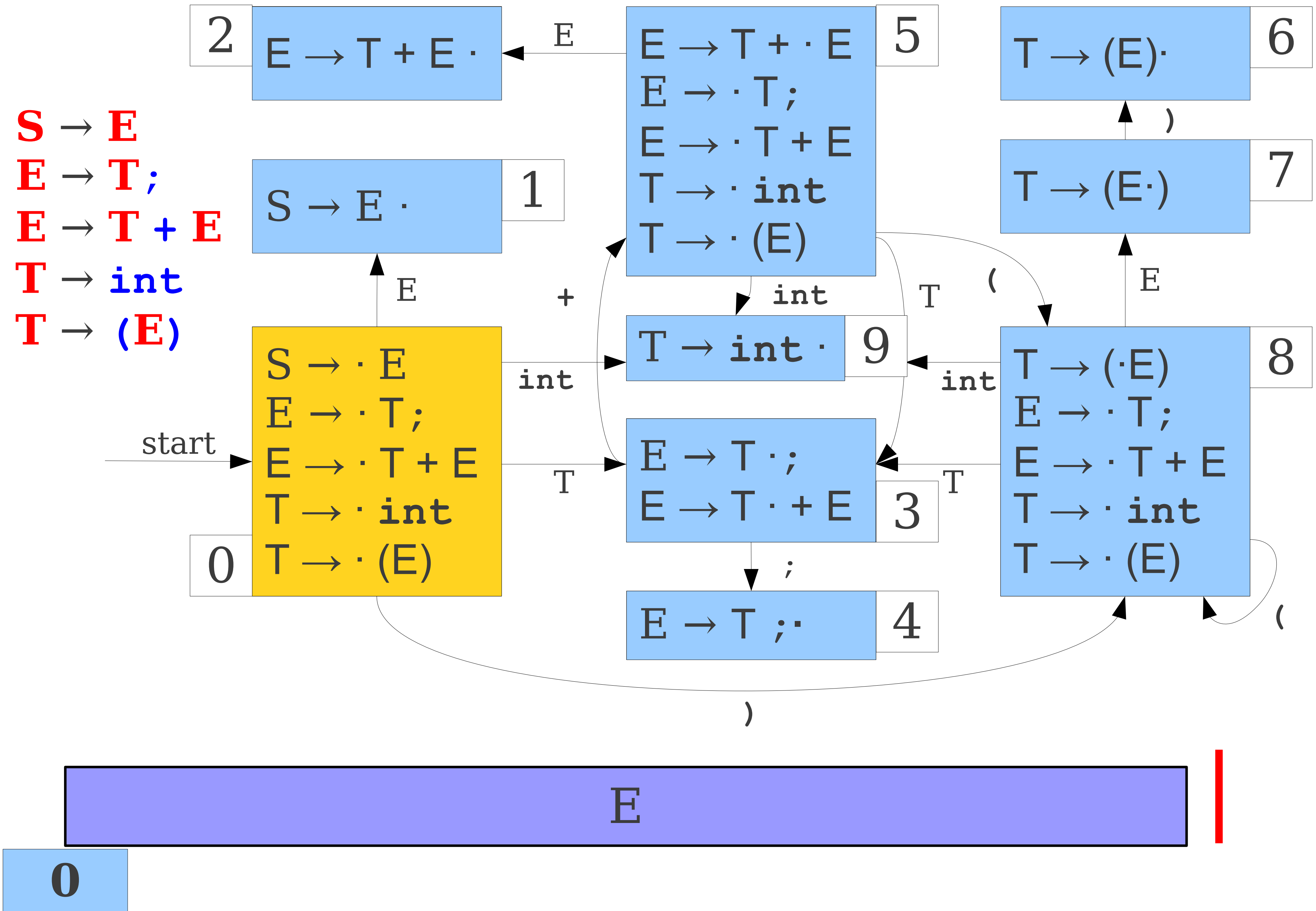
S → E
E → T;
E → T + E
T → int
T → (E)



0

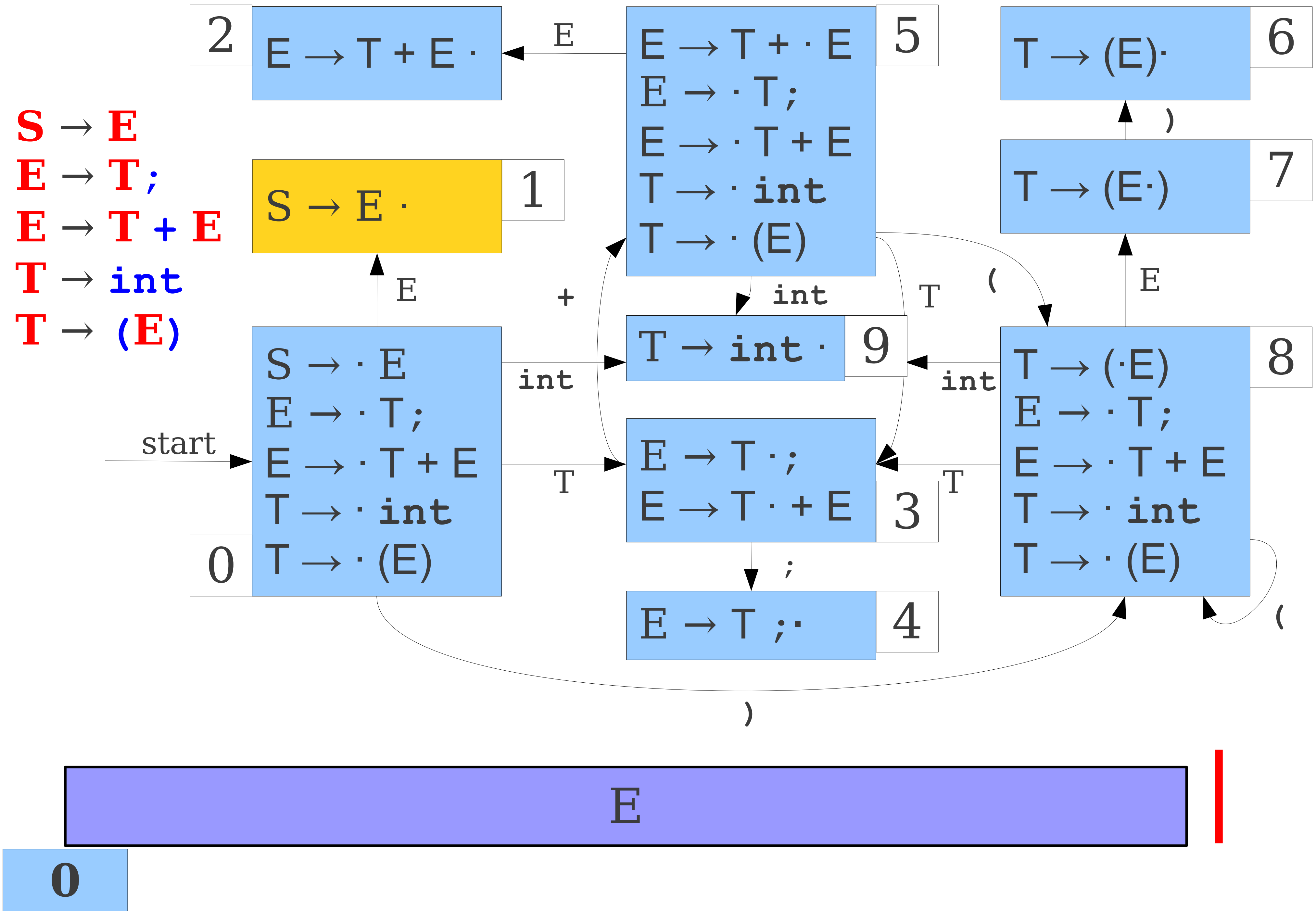
LR(0) Parsing

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



LR(0) Parsing

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

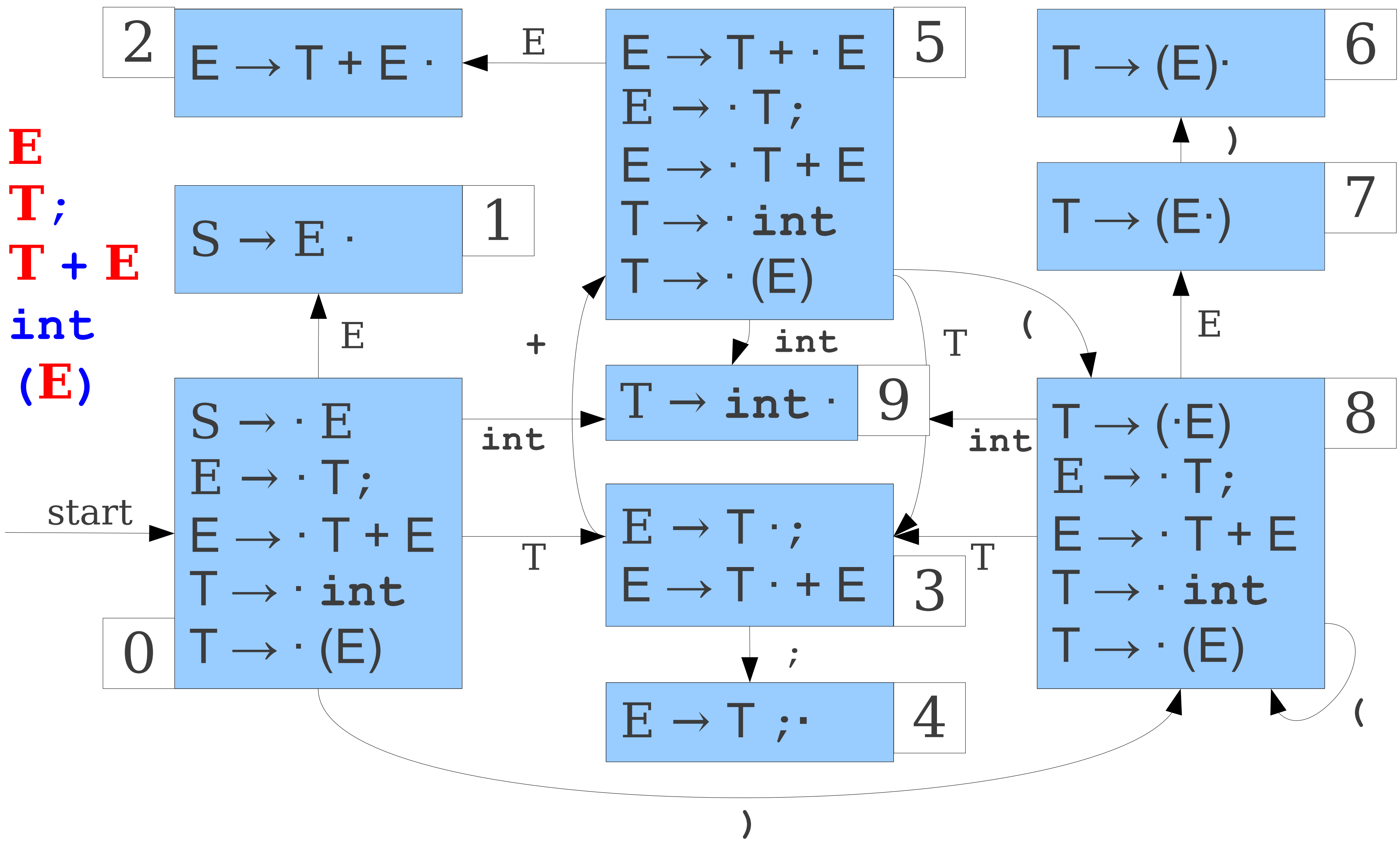


Representing the Automaton

- LR(0) parsers are usually represented via two tables: an **action** table and a **goto** table.
- The **action** table maps each state to an action:
  - **shift**, which shifts the next terminal, and
  - **reduce $\mathbf{A} \rightarrow \boldsymbol{\omega}$**, which performs reduction $\mathbf{A} \rightarrow \boldsymbol{\omega}$.
  - Any state of the form $\mathbf{A} \rightarrow \boldsymbol{\omega} \cdot$ does that reduction; everything else shifts.
- The **goto** table maps state/symbol pairs to a next state.
  - This is just the transition table for the automaton.

Building LR(0) Tables

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



LR(0) Tables

	int	+	;	(	)	E	T	Action
0	9			8		1	3	Shift
1								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**

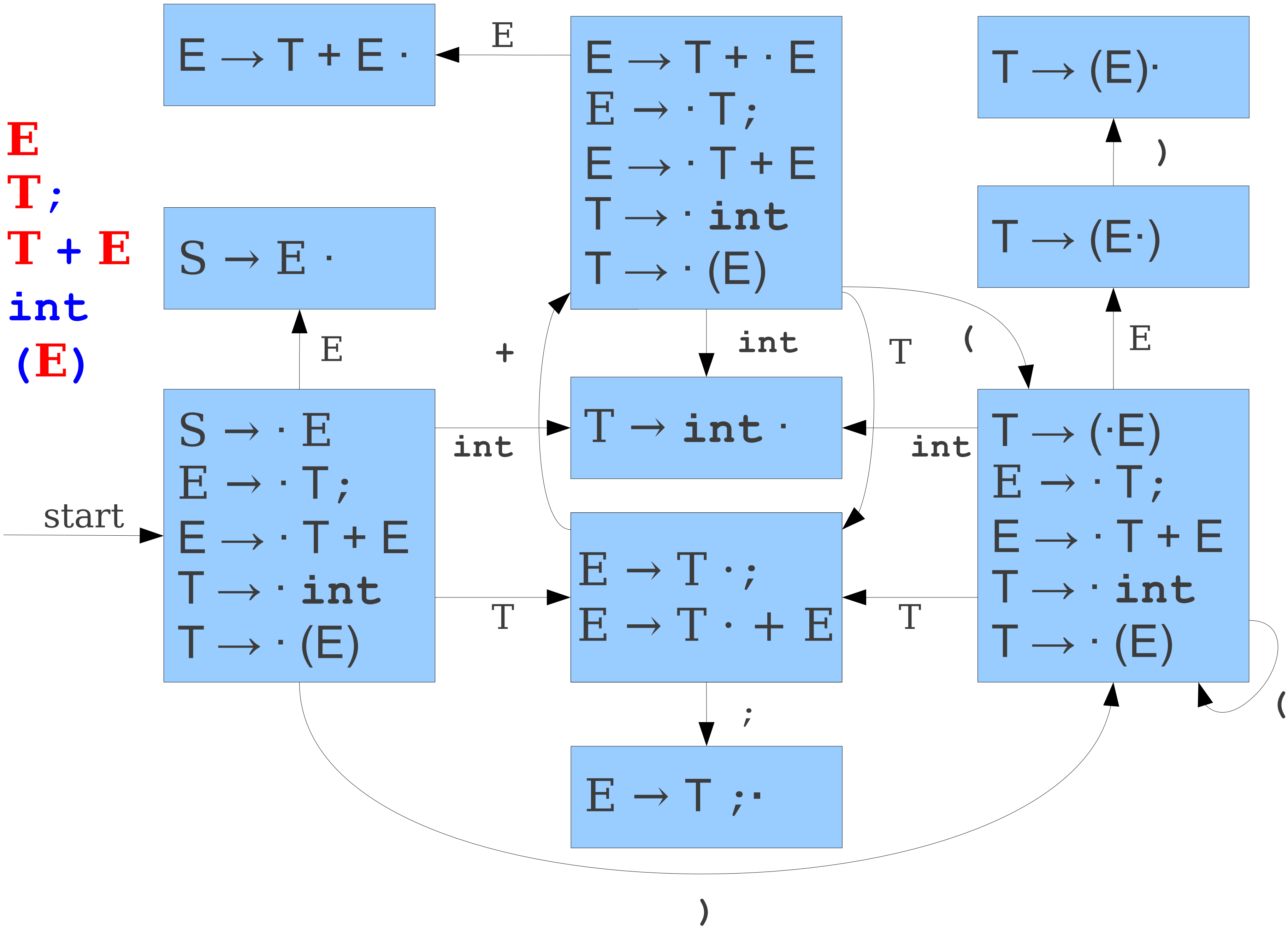
The LR(0) Algorithm

- Maintain a stack of (symbol, state) pairs, which is initially (`?`, 1) for some dummy symbol `?`.
- While the stack is not empty:
  - Let **state** be the top state.
  - If **action[state]** is **shift**:
    - Let `t` be the next symbol in the input.
    - Push (`t`, **goto[state, `t`]**) atop the stack.
  - If **action[state]** is **reduce A** → $\boldsymbol{\omega}$:
    - Remove $|\boldsymbol{\omega}|$ symbols from the top of the stack.
    - Let **top-state** be the state on top of the stack.
    - Push (**A**, **goto[top-state, A]**) atop the stack.
  - Otherwise, report an error.

The Limits of LR(0)

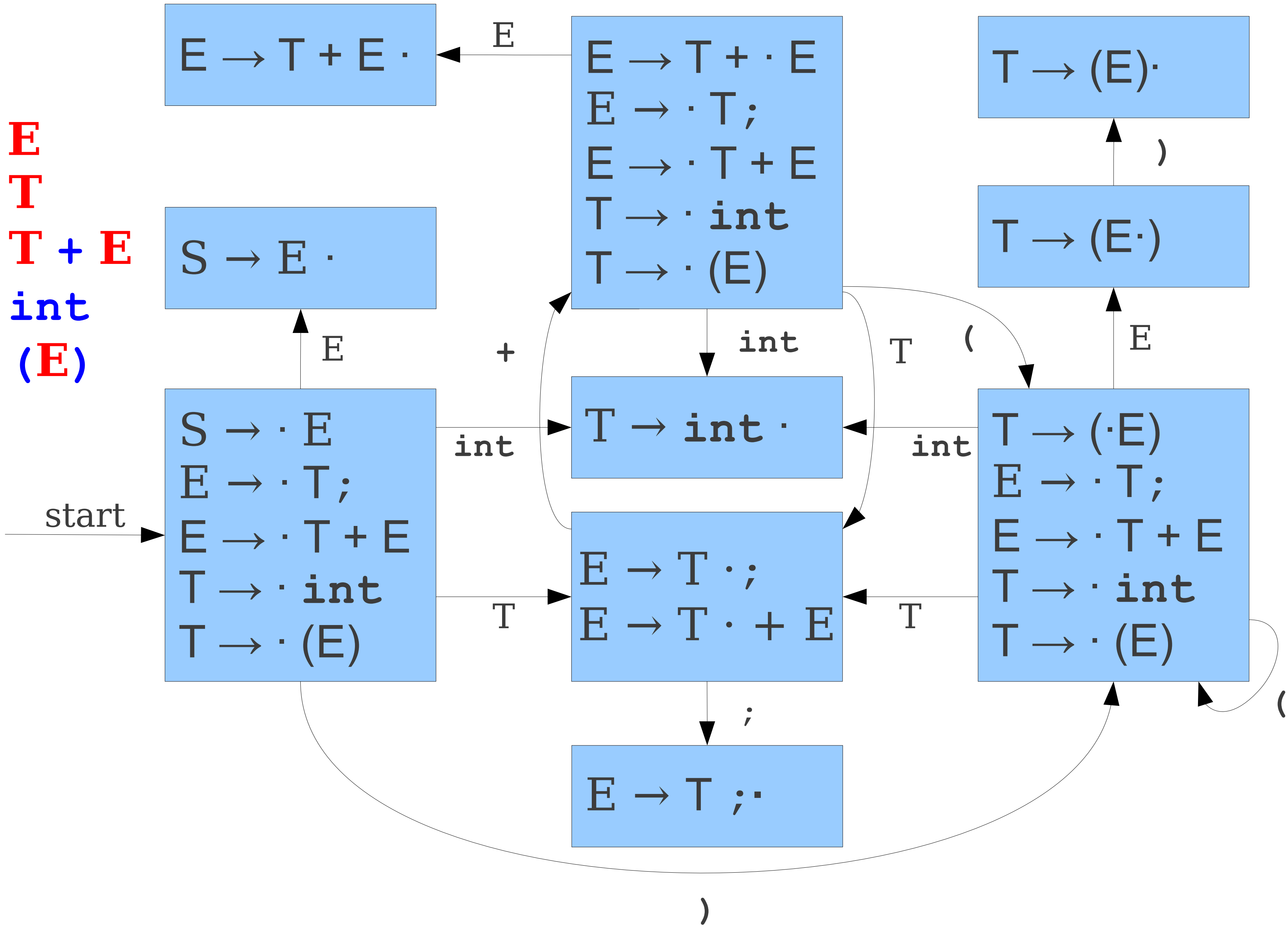
A Non-LR(0) Grammar

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



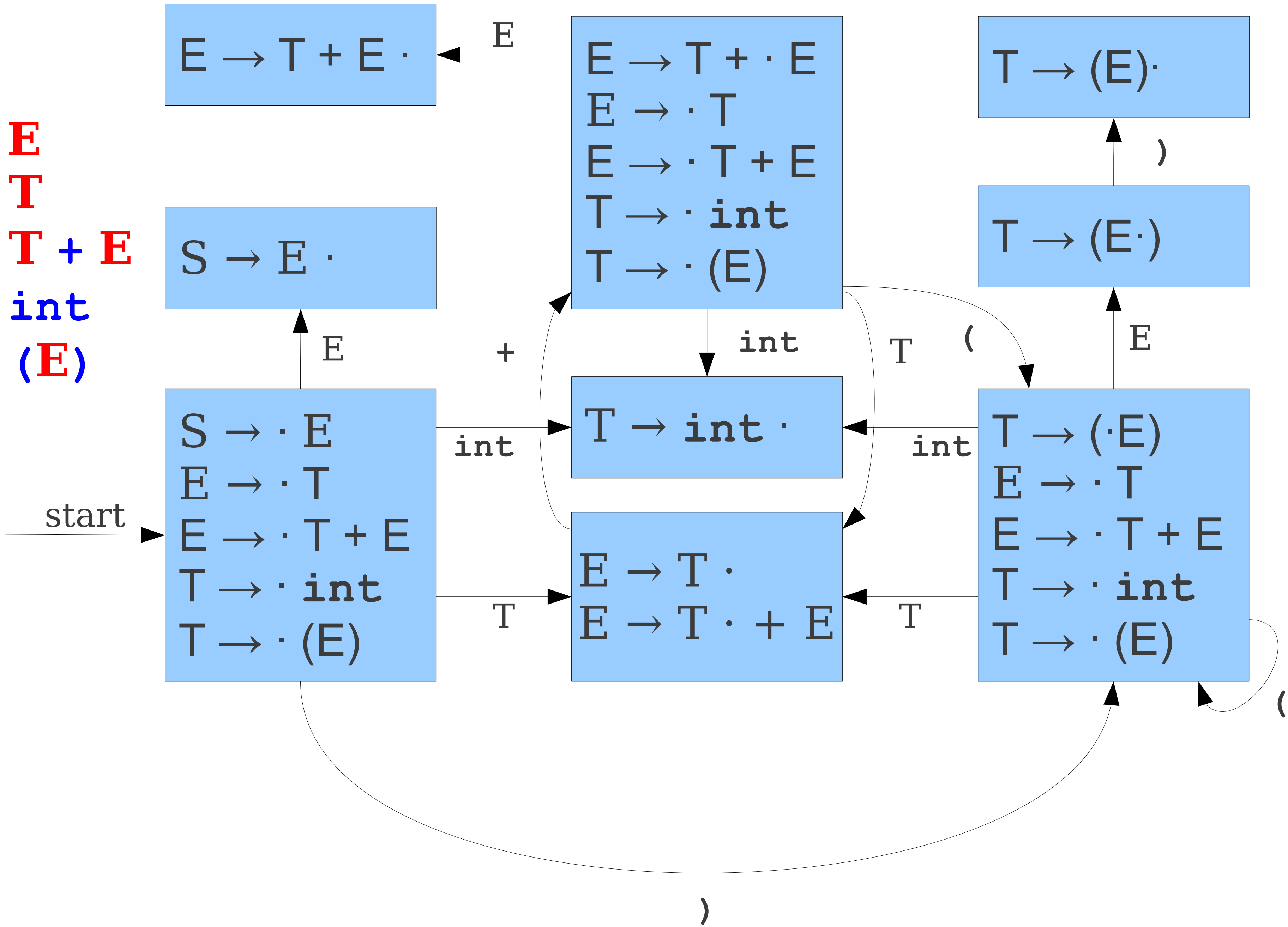
A Non-LR(0) Grammar

S → E
E → T
E → T + E
T → int
T → (E)



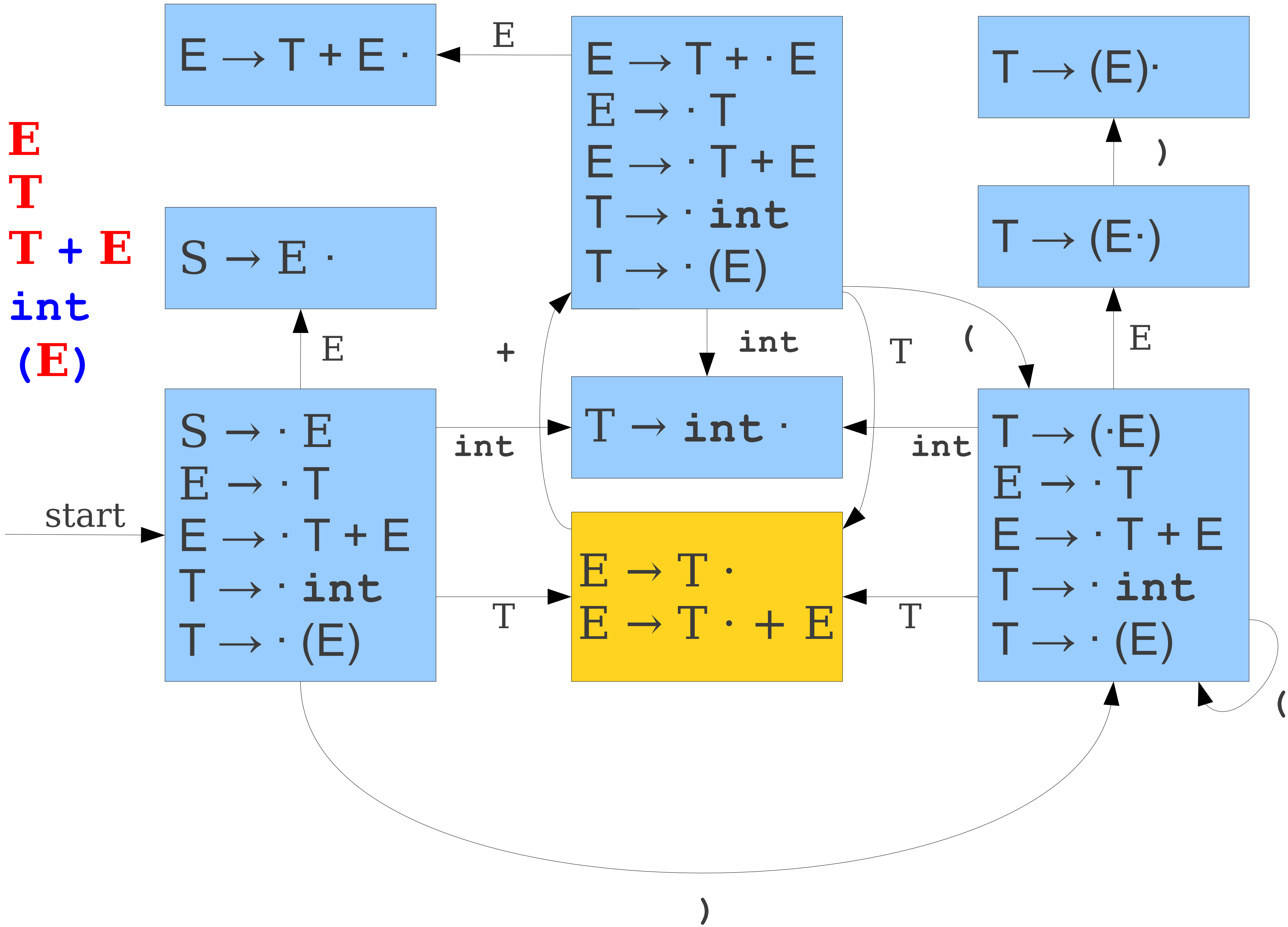
A Non-LR(0) Grammar

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



A Non-LR(0) Grammar

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

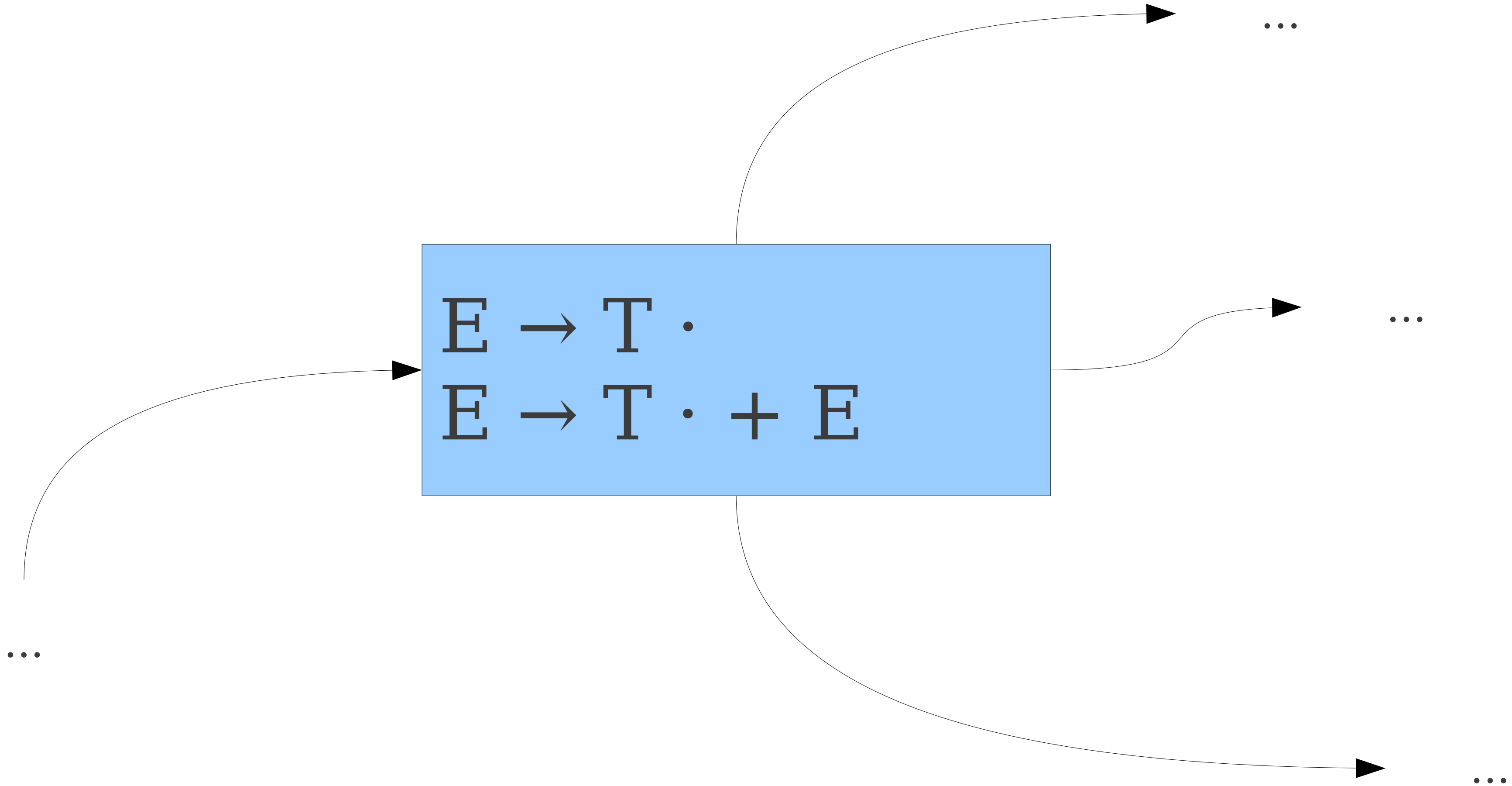


LR Conflicts

- A **shift/reduce conflict** is an error where a shift/reduce parser cannot tell whether to shift a token or perform a reduction.
  - Often happens when two productions overlap.
- A **reduce/reduce conflict** is an error where a shift/reduce parser cannot tell which of many reductions to perform.
  - Often the result of ambiguous grammars.
- A grammar whose handle-finding automaton contains a shift/reduce conflict or a reduce/reduce conflict is not LR(0).
- Can you have a shift/shift conflict?

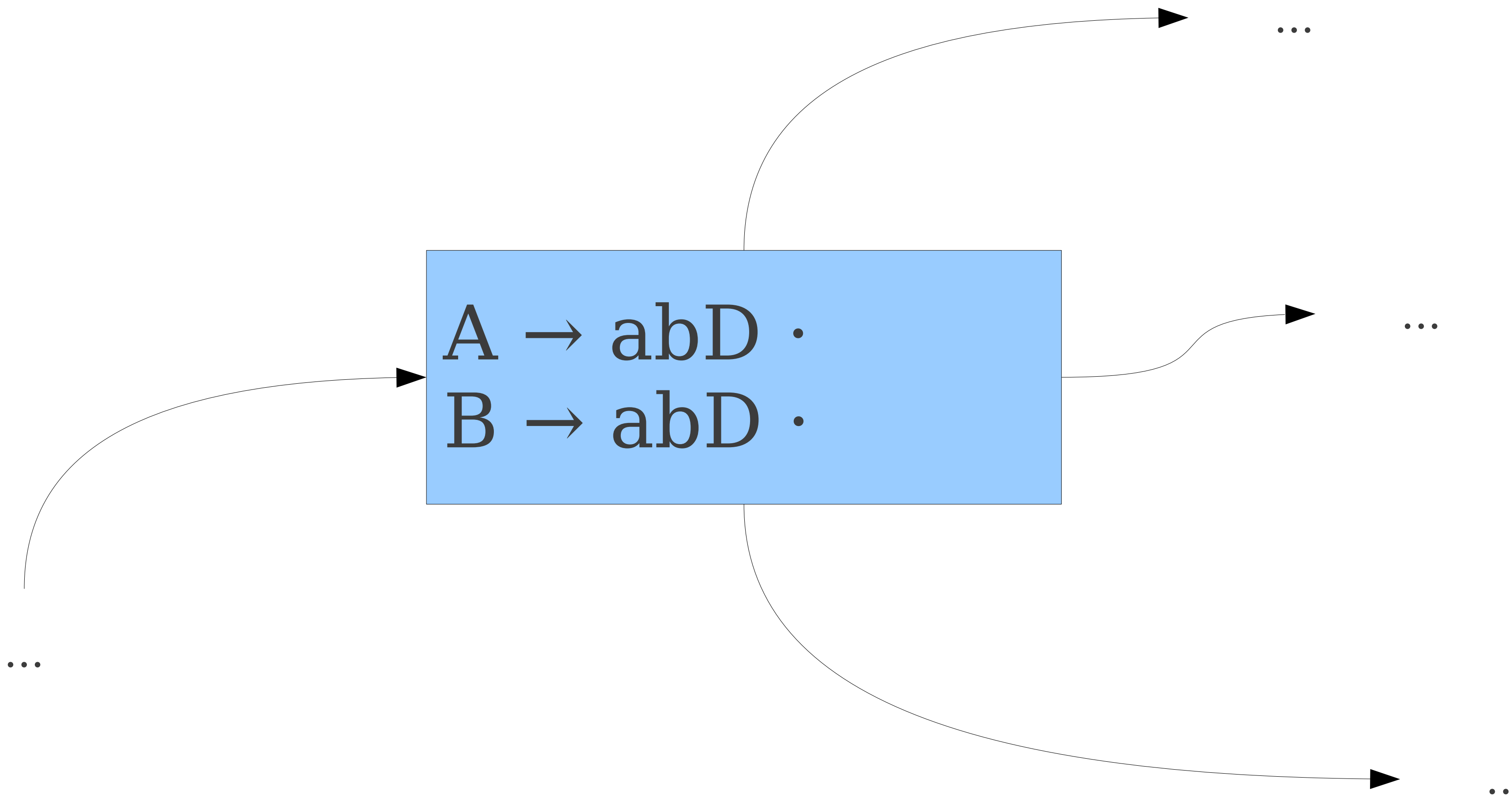
What error is this?

What error is this?



What about this?

What about this?



What do these conflicts mean?

- Recall: our automaton was constructed by looking for viable prefixes.
- Each accepting state represents a point where the handle might occur.
- A **shift/reduce** conflict is a state where the handle might occur, but we might actually need to keep searching.
- A **reduce/reduce** conflict is a state where we know we have found the handle, but can't tell which reduction to apply.

Next Time

- Add Lookahead to parse a more powerful subset of CFGs than LR(0).