Bottom-Up Parsing, Part II

Announcements

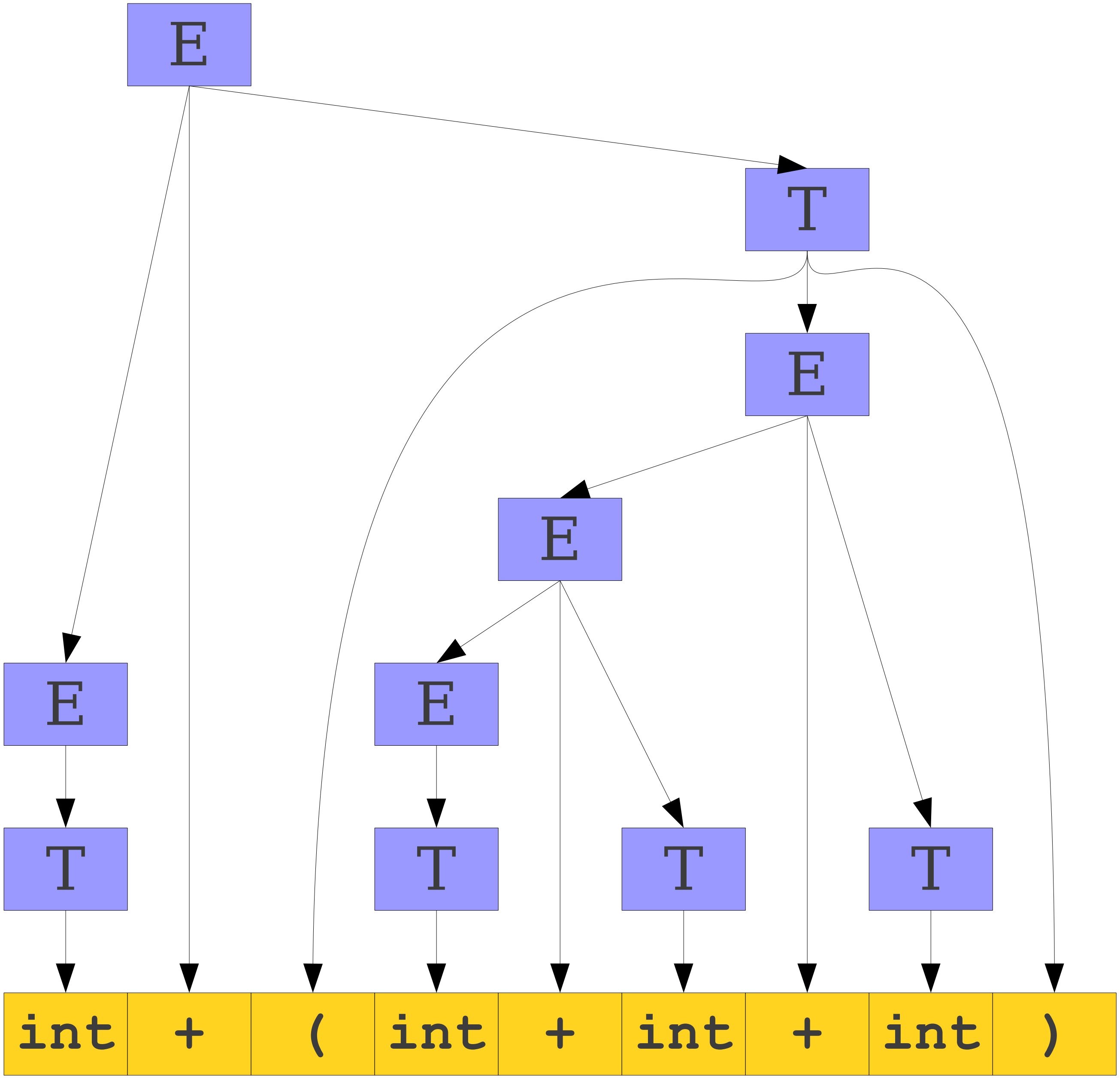
- Programming Assignment 1 due tonight at 11:59PM.
- Programming Assignment 2 (parsing) out, due Friday, July 20th at 11:59PM.
  - Play around with the **`bison`** parser generator!
  - See how real parsers are written!

Announcements

- C++ review session tonight in Gates B12 from 7:00PM – 8:30PM.
  - Covers classes and inheritance.
  - Extremely valuable for the second programming assignment, especially if you have not seen C++ inheritance before.

One View of a Bottom-Up Parse

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



A Second View of a Bottom-Up Parse

$\mathbf{E} \rightarrow \mathbf{T}$
$\mathbf{E} \rightarrow \mathbf{E} + \mathbf{T}$
$\mathbf{T} \rightarrow$ `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 Second View of a Bottom-Up Parse

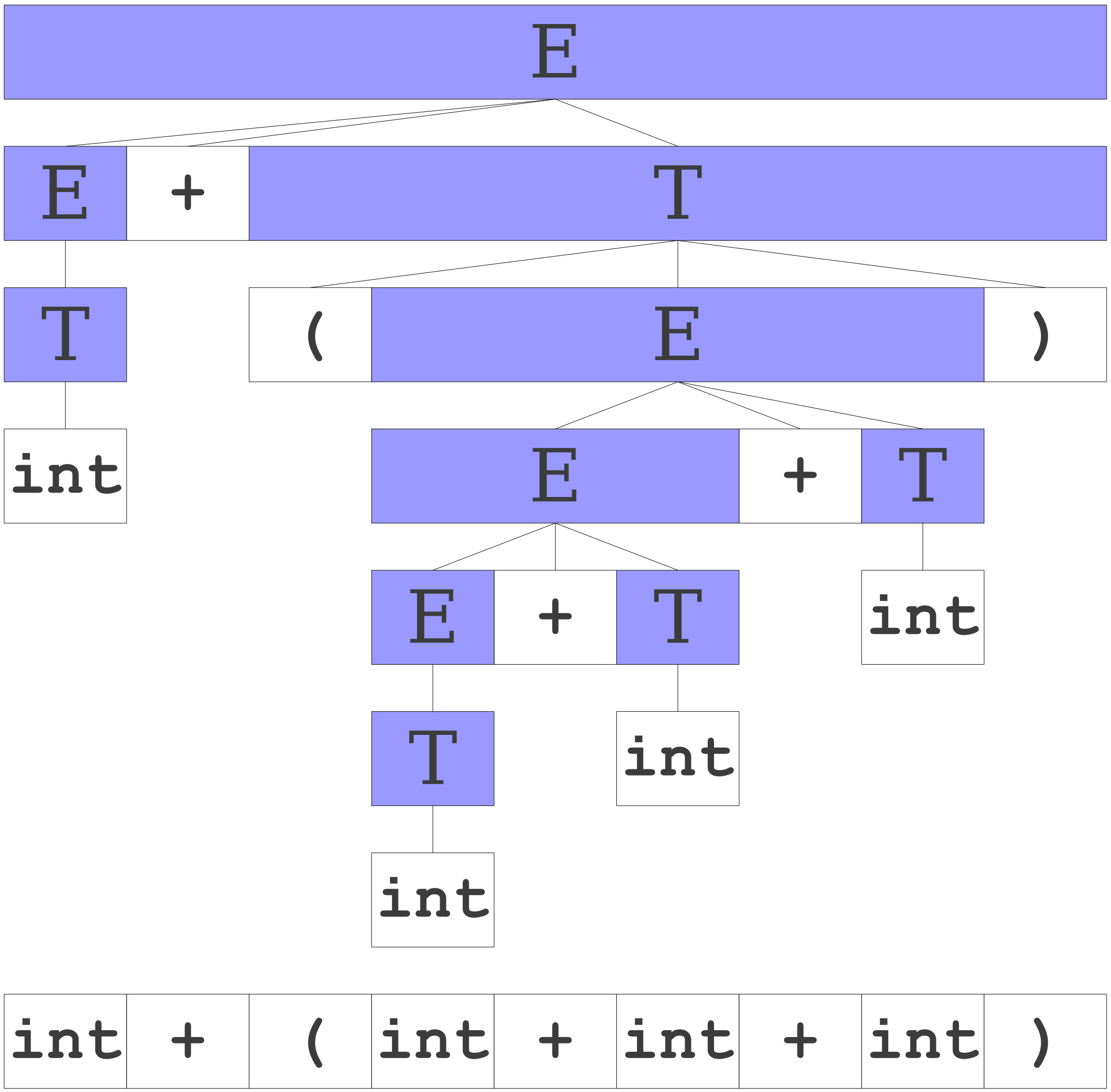
$E \rightarrow T$
$E \rightarrow E$ **+** T
$T \rightarrow$ **int**
$T \rightarrow$ **(**E**)**

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

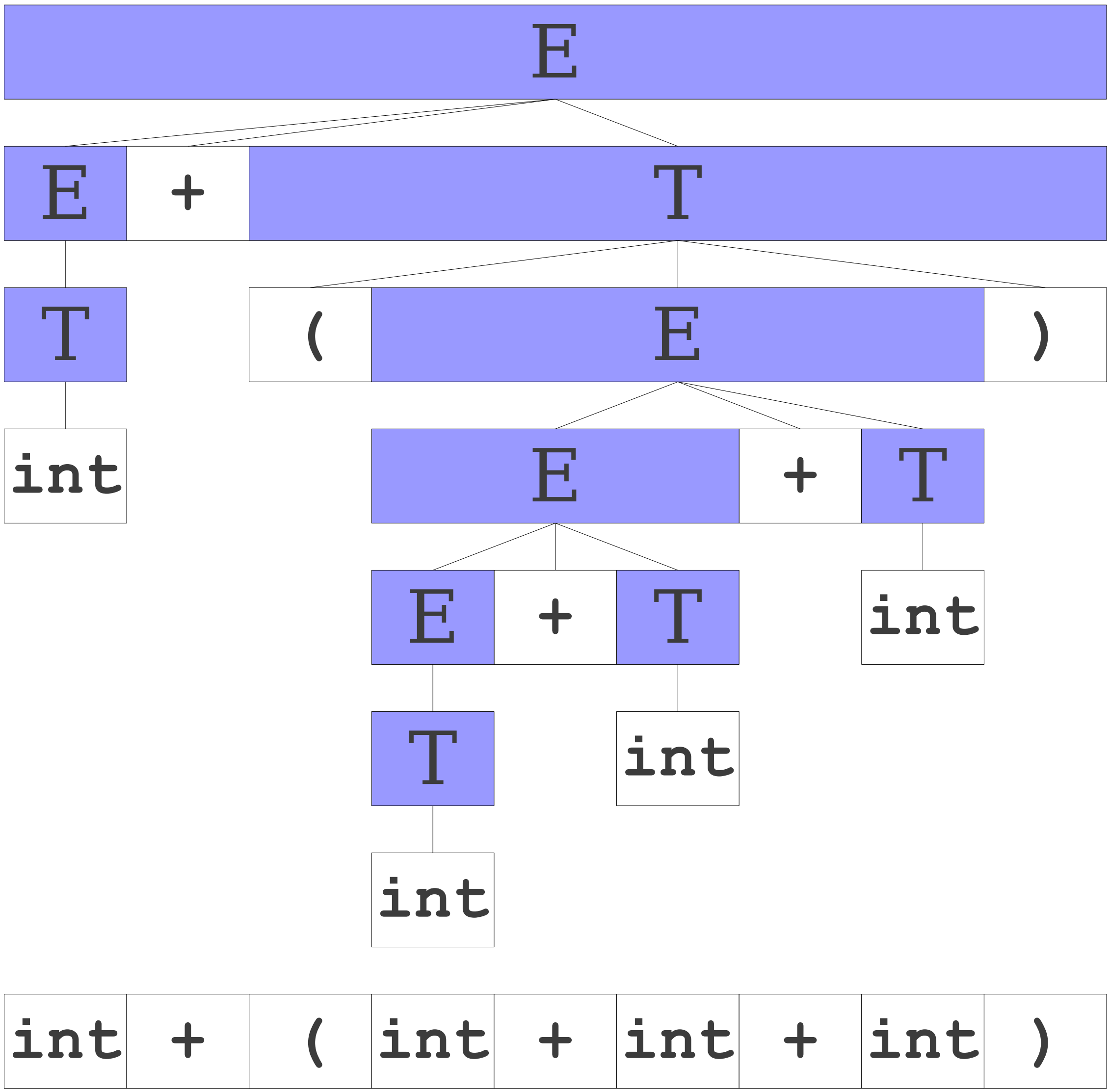
A Third View of a Bottom-Up Parse

```
  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
```



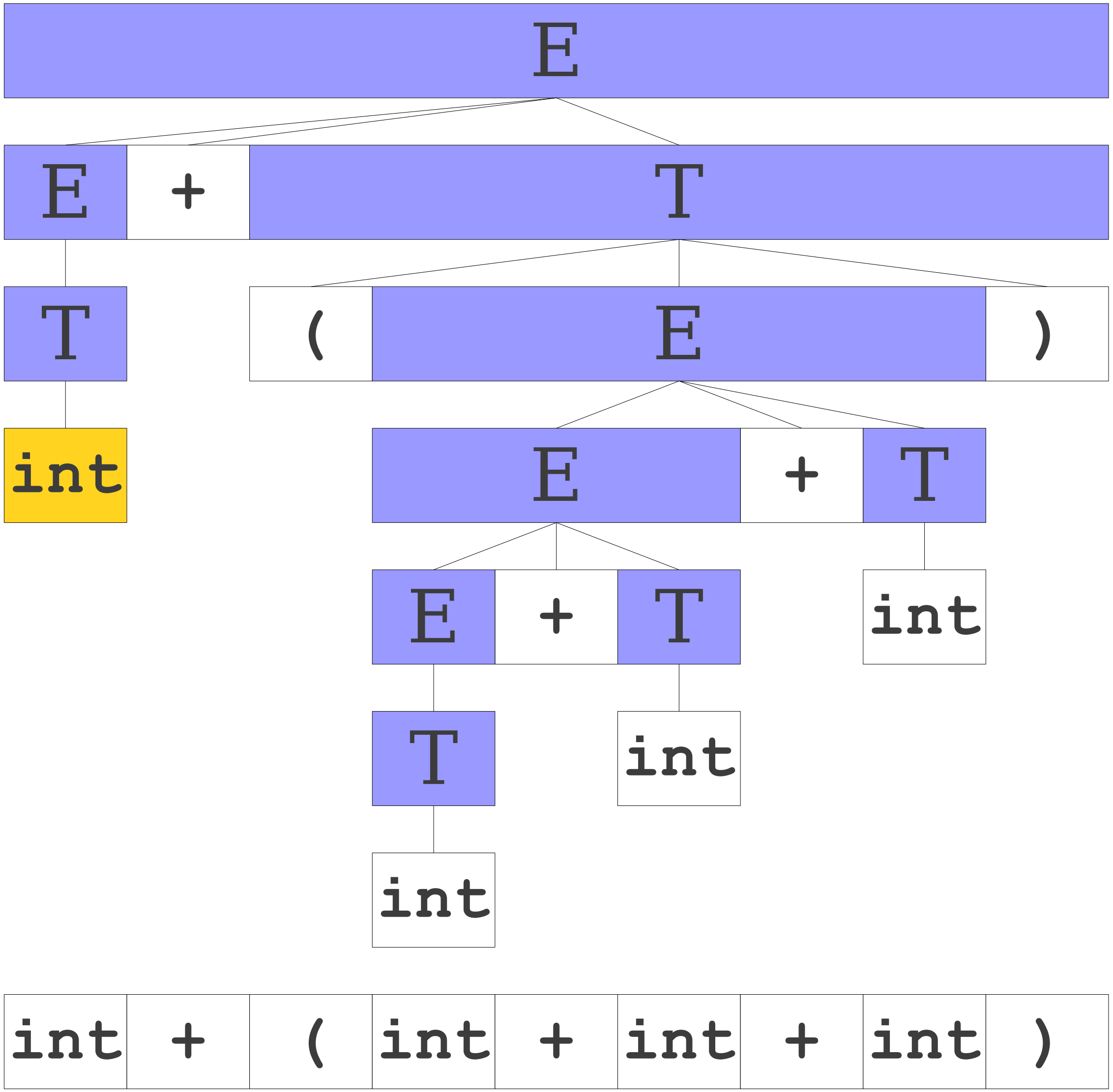
A Third View of a Bottom-Up Parse

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



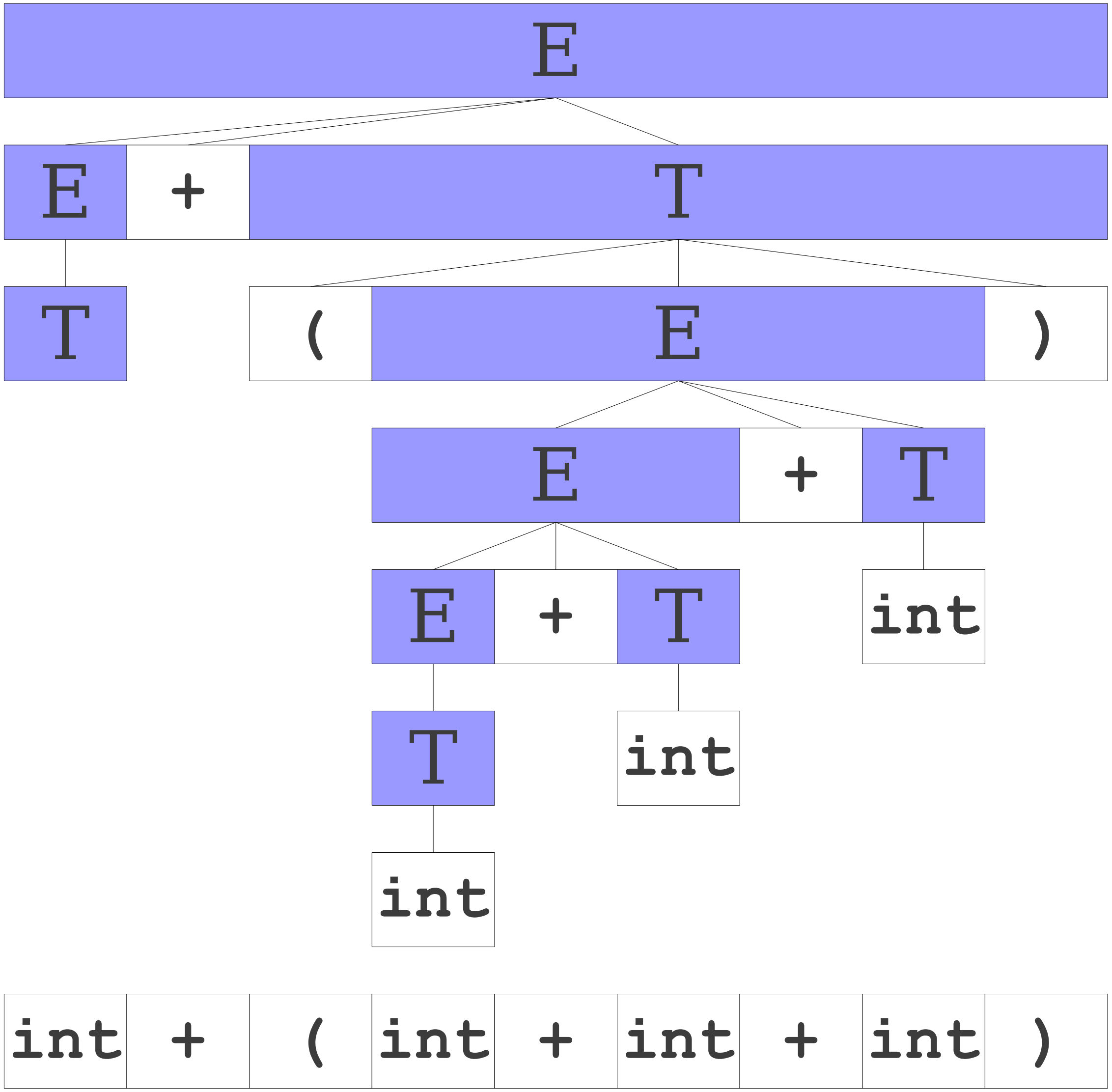
A Third View of a Bottom-Up Parse

```
  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
```



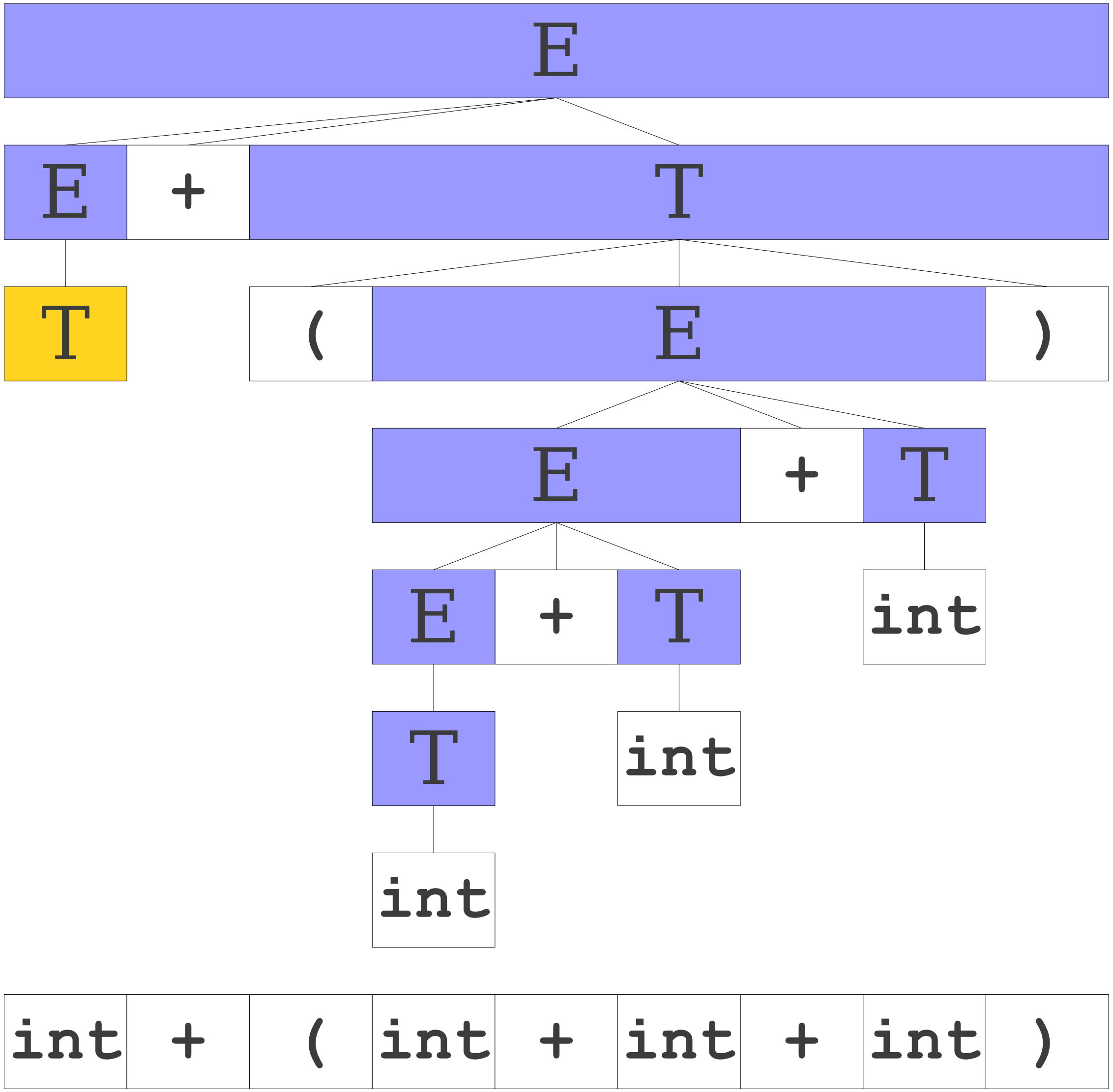
A Third View of a Bottom-Up Parse

⇒ **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**



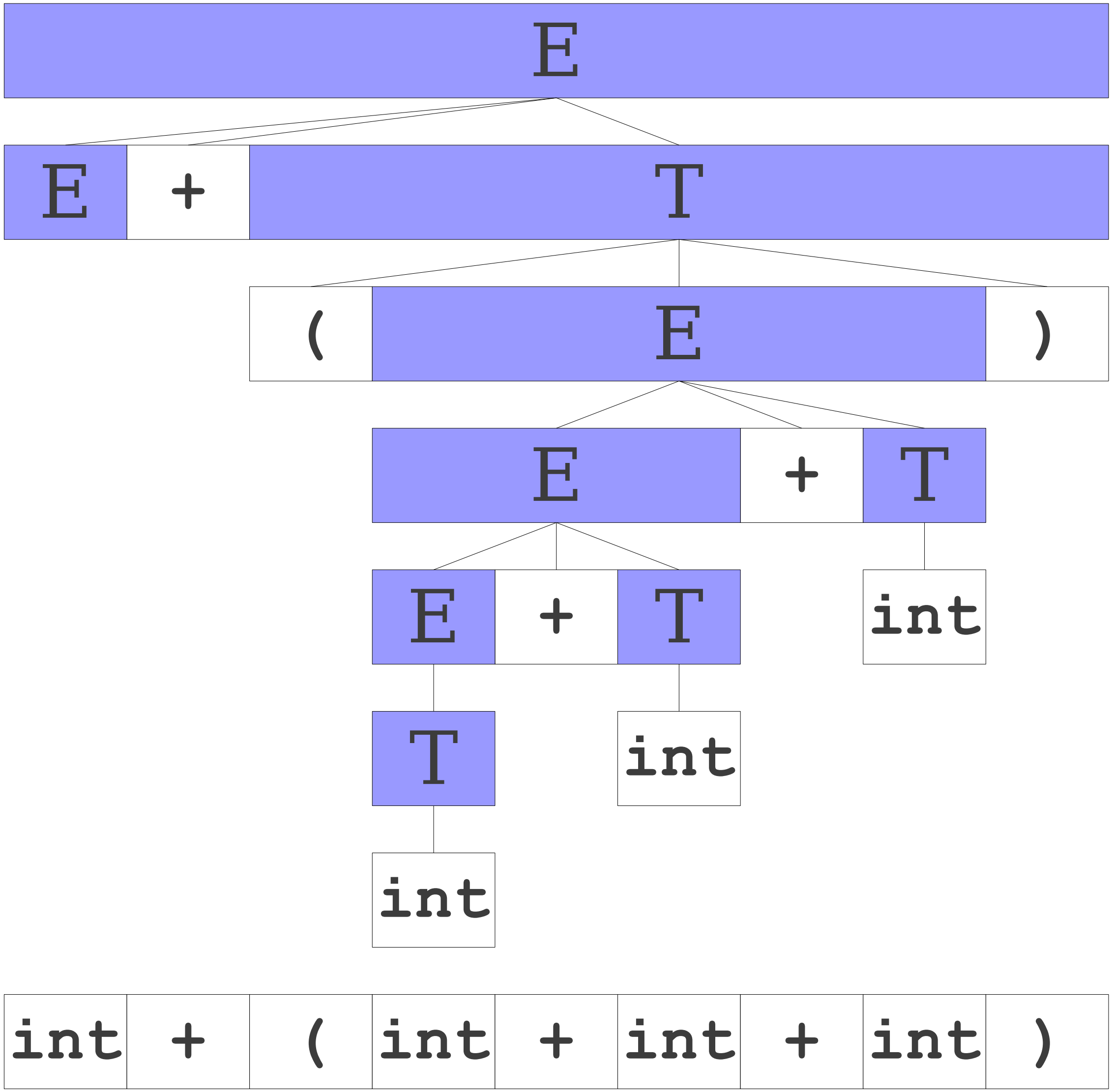
A Third View of a Bottom-Up Parse

⇒ **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**



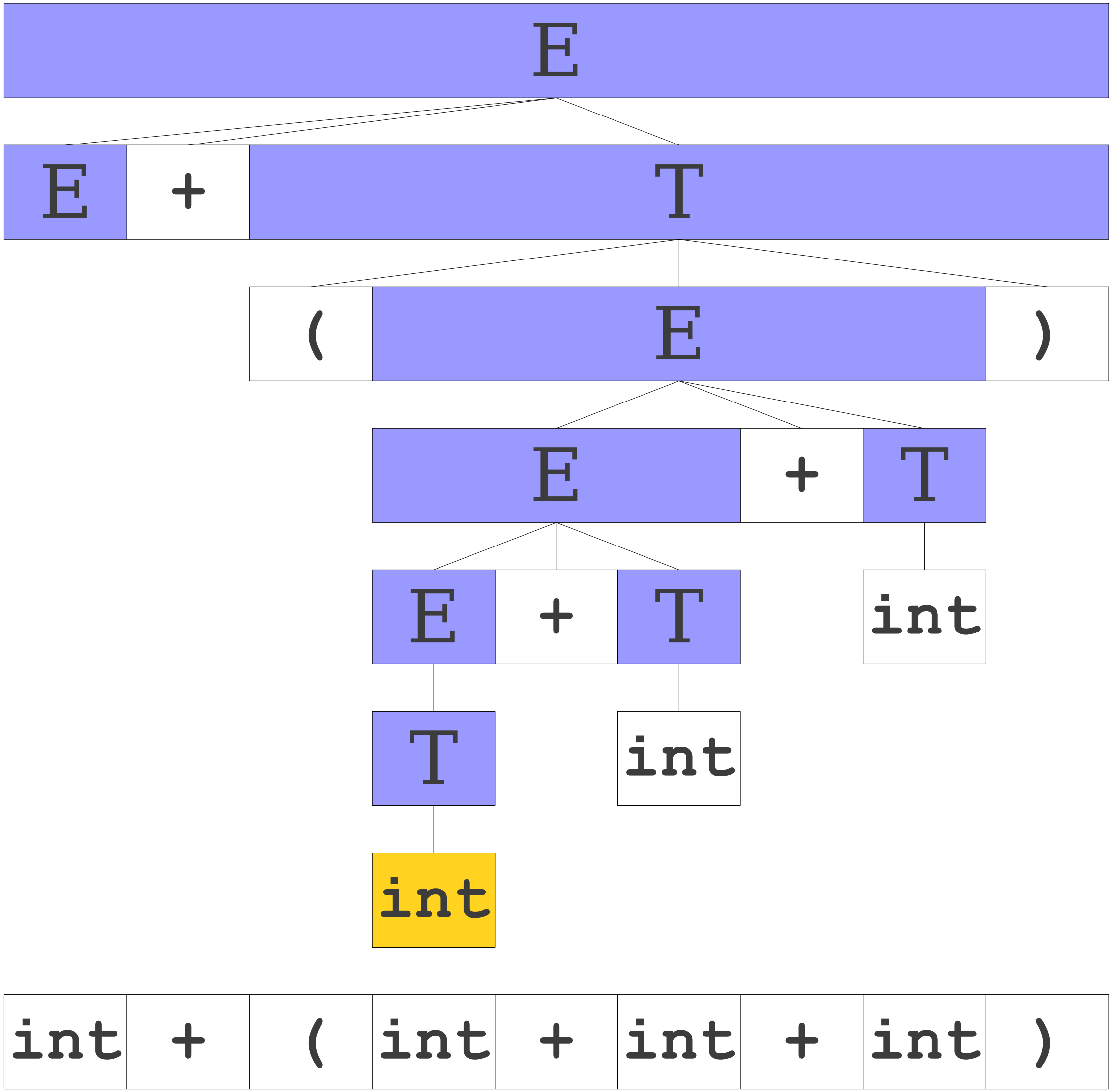
A Third View of a Bottom-Up Parse

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



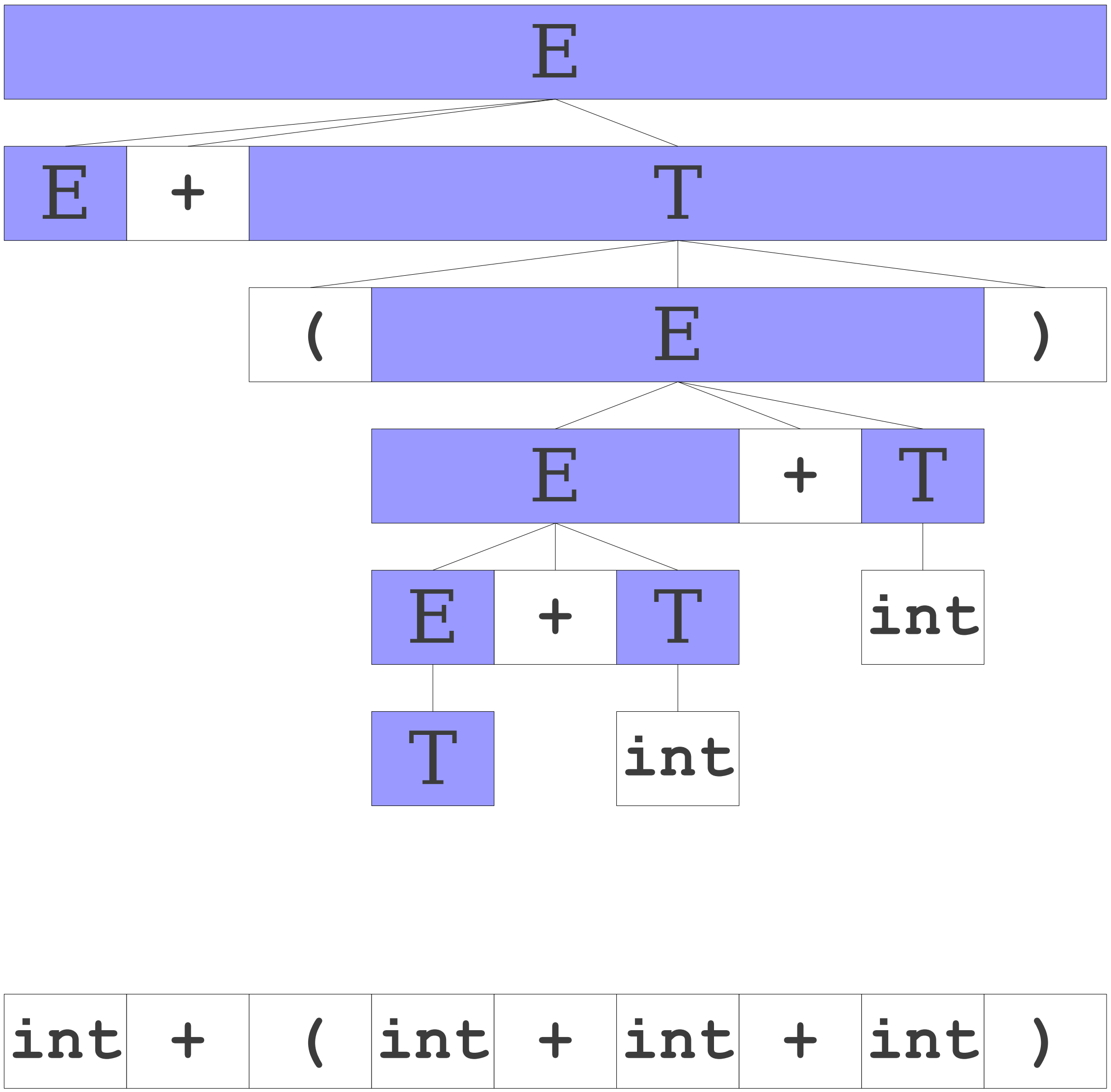
A Third View of a Bottom-Up Parse

⇒ **E** + (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**



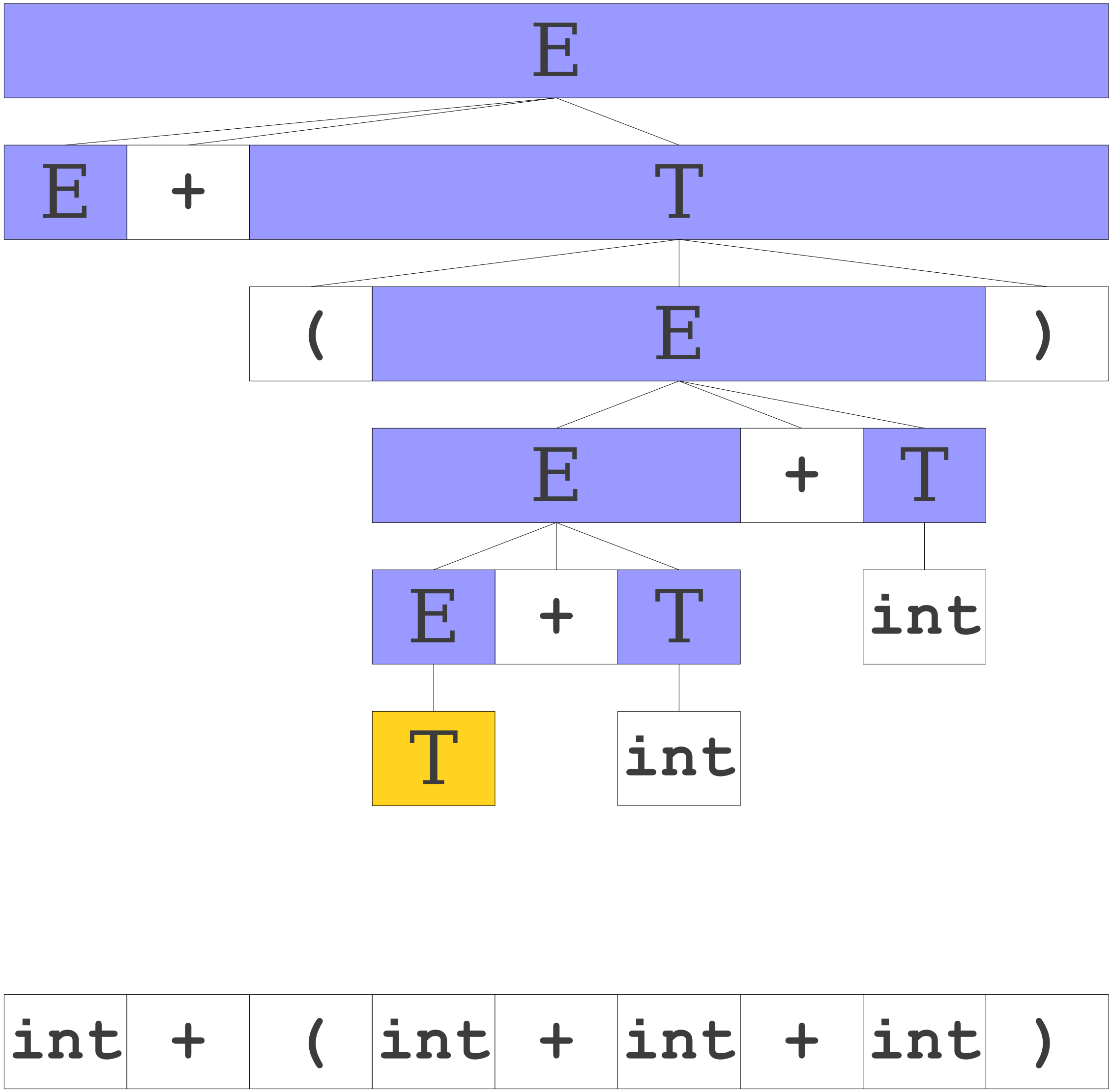
A Third View of a Bottom-Up Parse

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



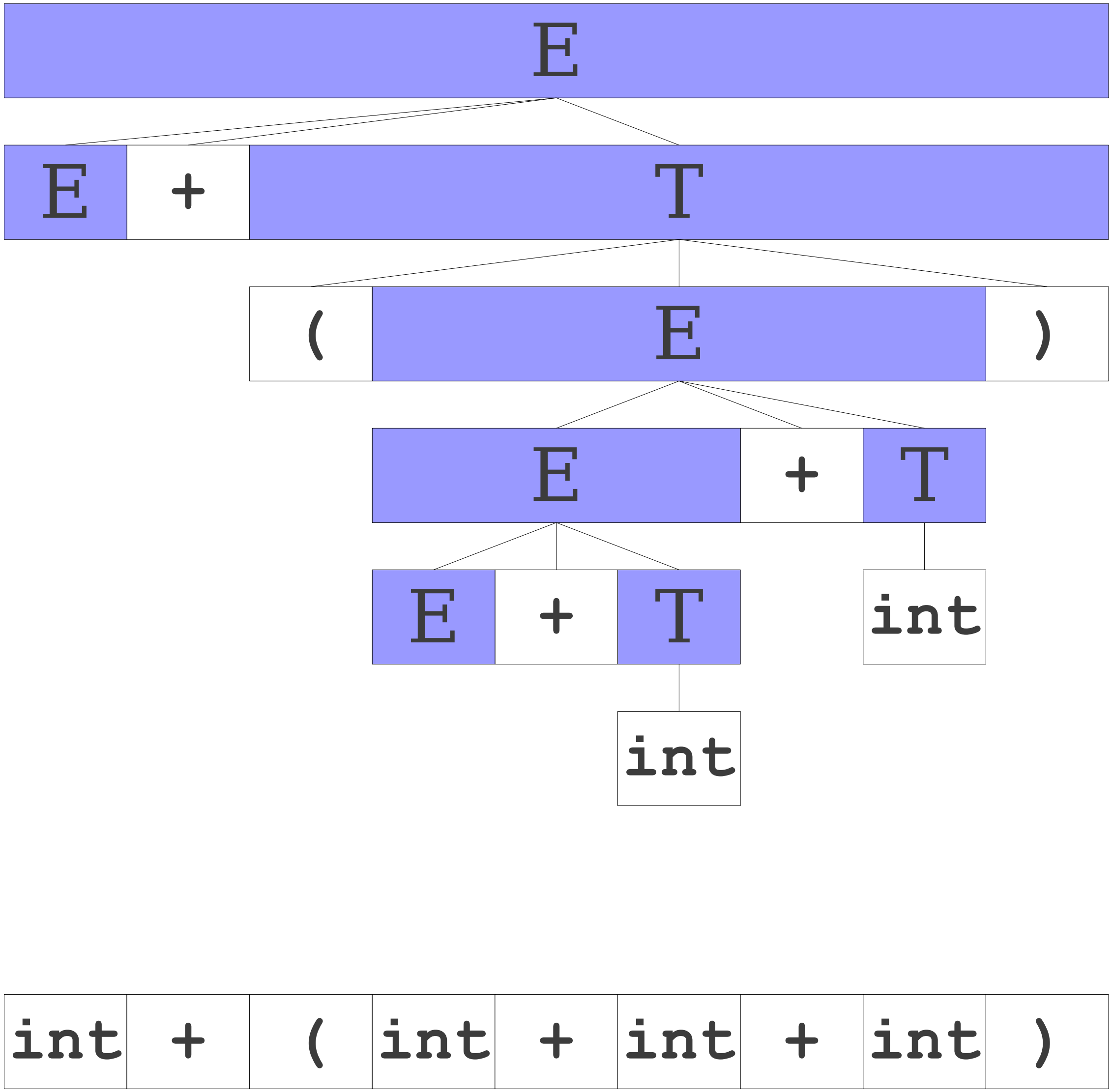
A Third View of a Bottom-Up Parse

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



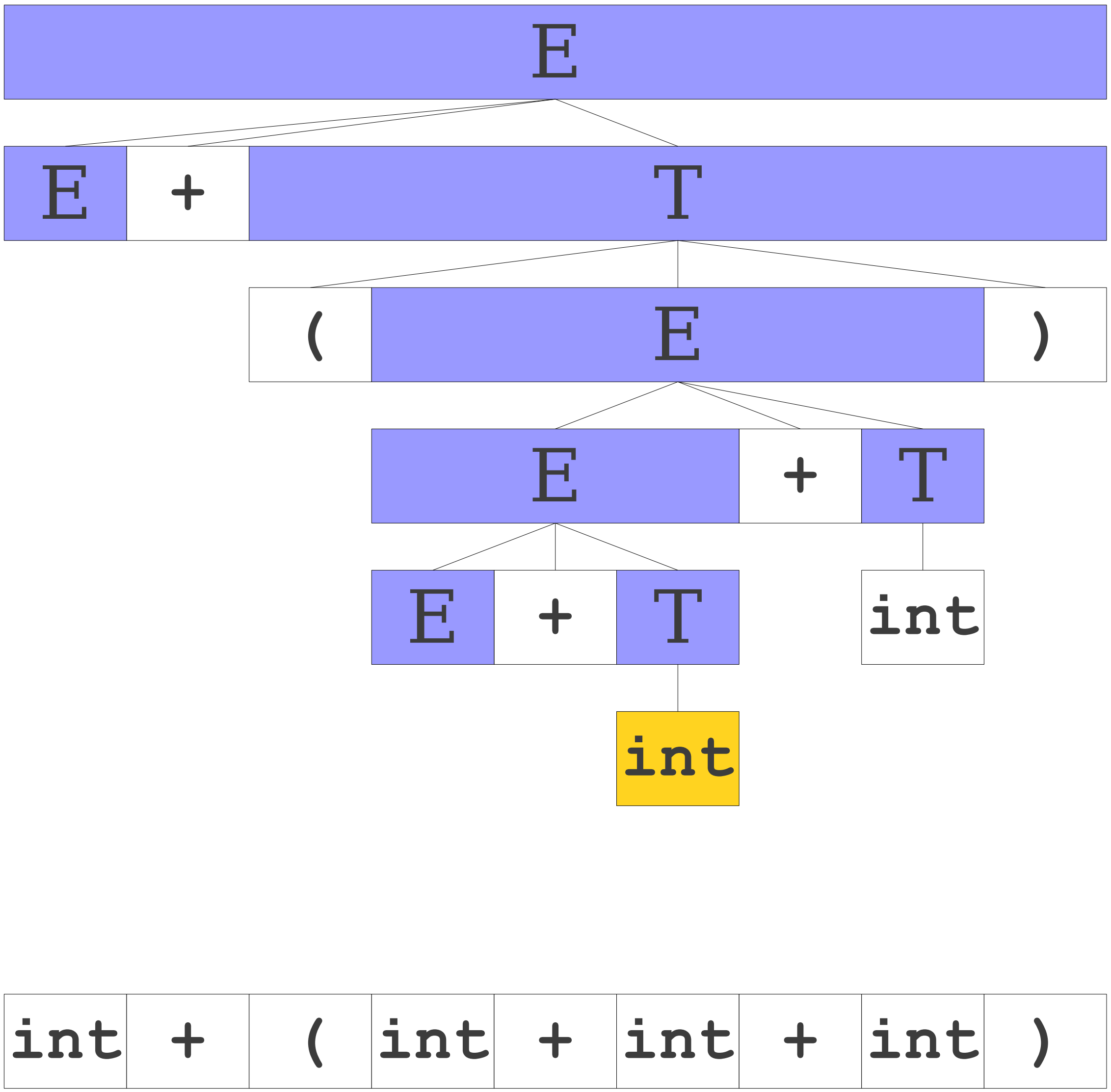
A Third View of a Bottom-Up Parse

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

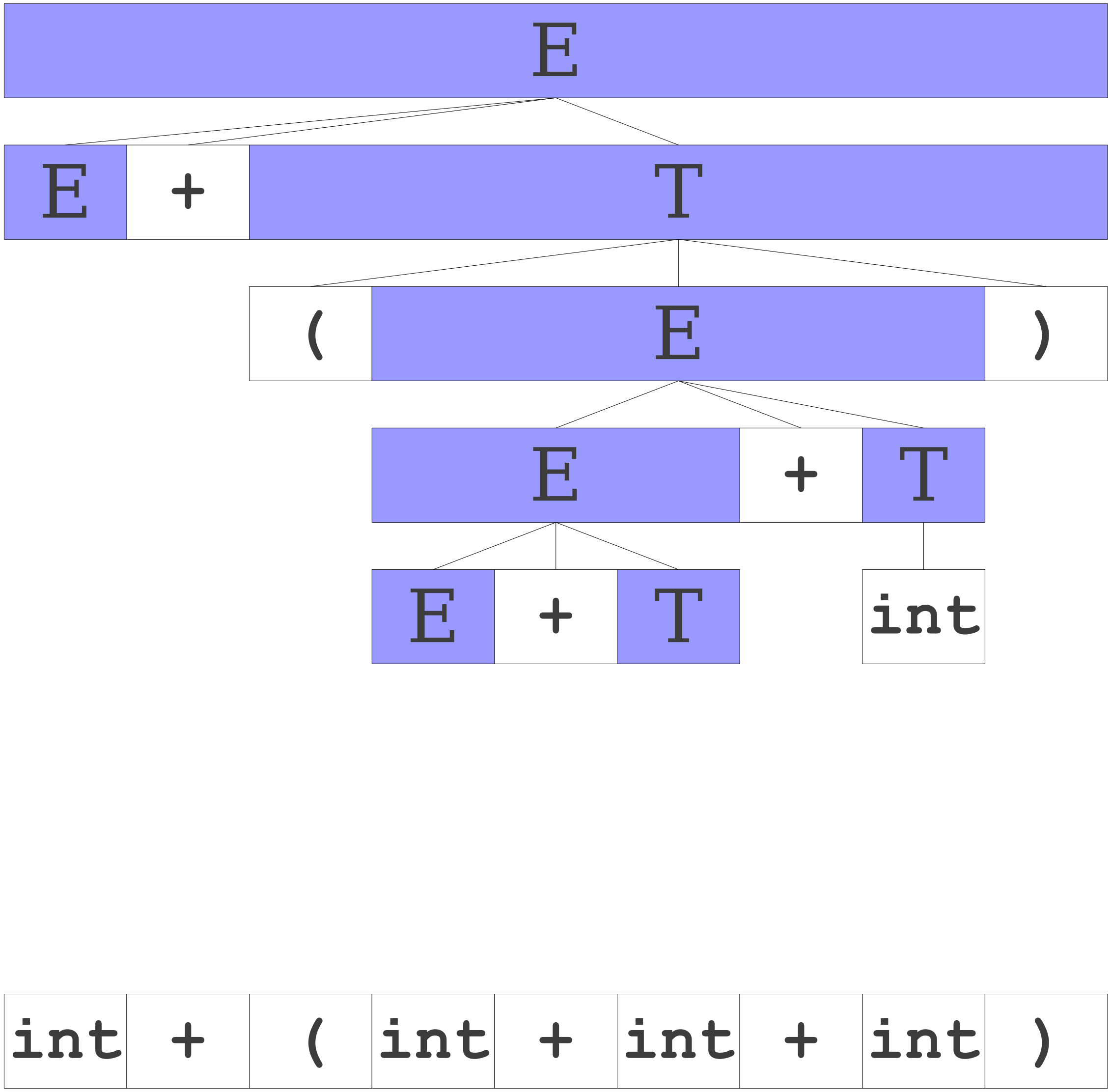


A Third View of a Bottom-Up Parse

⇒ 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



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

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

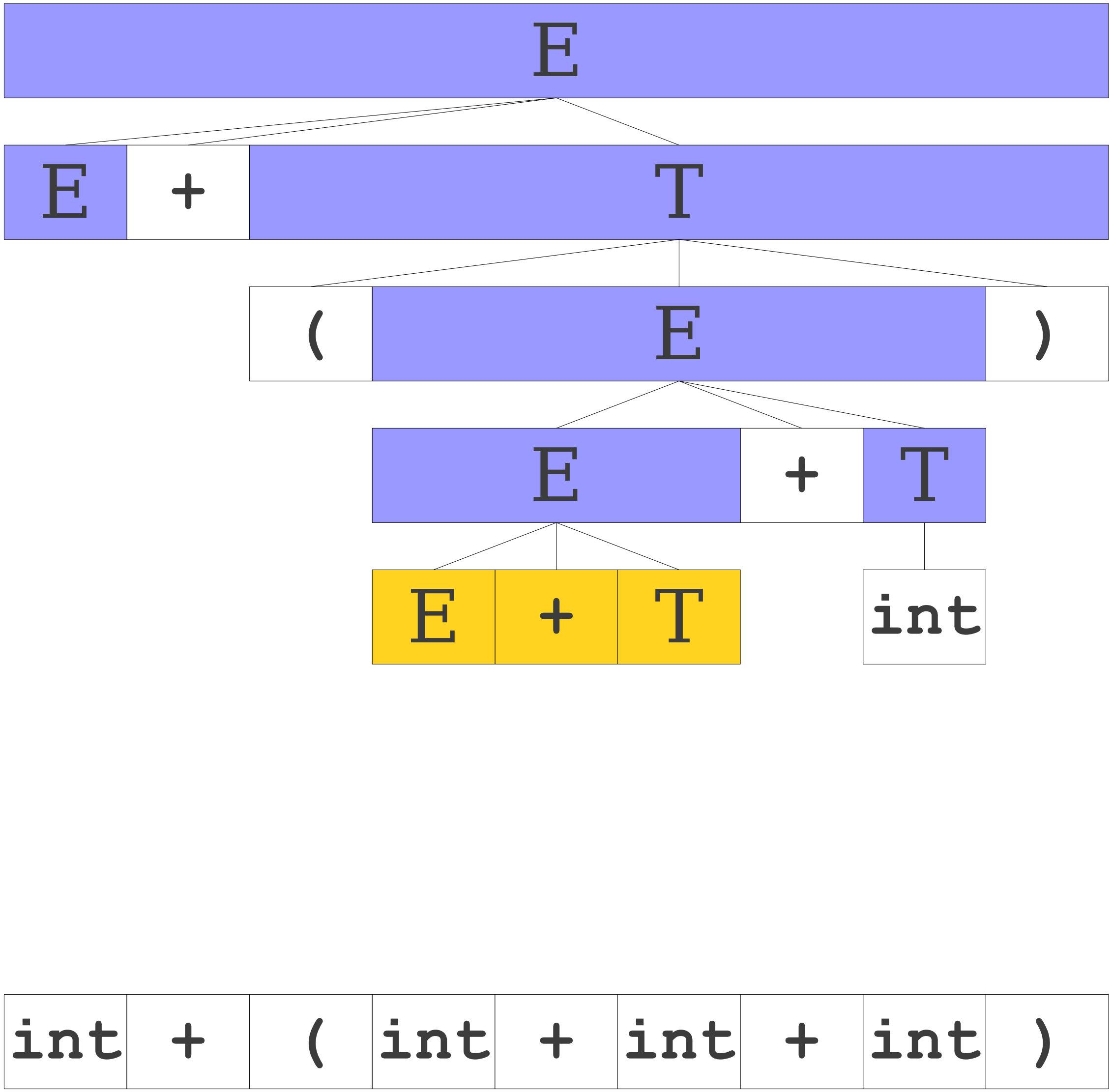
$\Rightarrow$ **E** + (**E** + **T**)

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

$\Rightarrow$ **E** + **T**

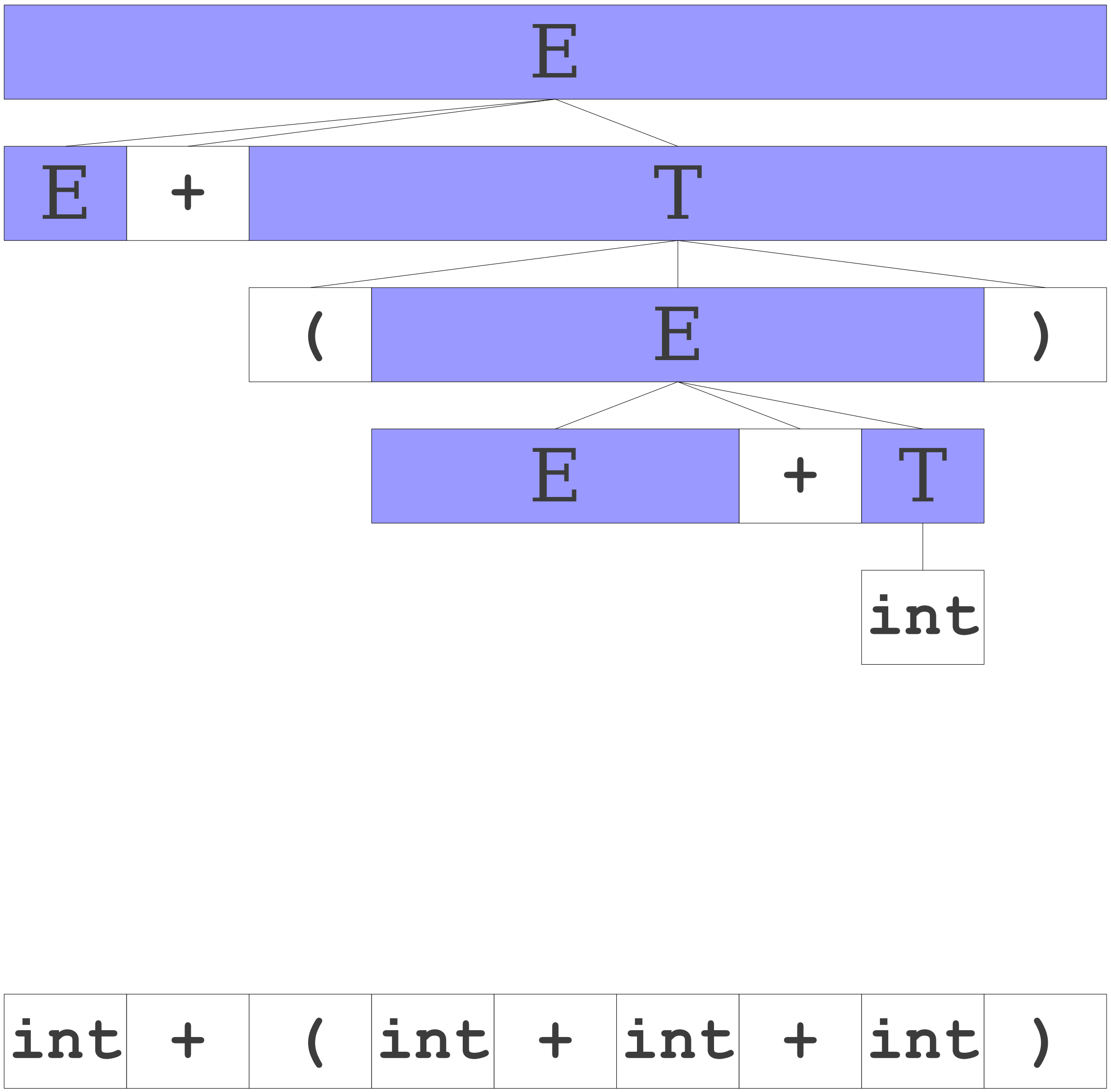
$\Rightarrow$ **E**

A Third View of a Bottom-Up Parse



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

A Third View of a Bottom-Up Parse



⇒ **E** + (**E** + int)

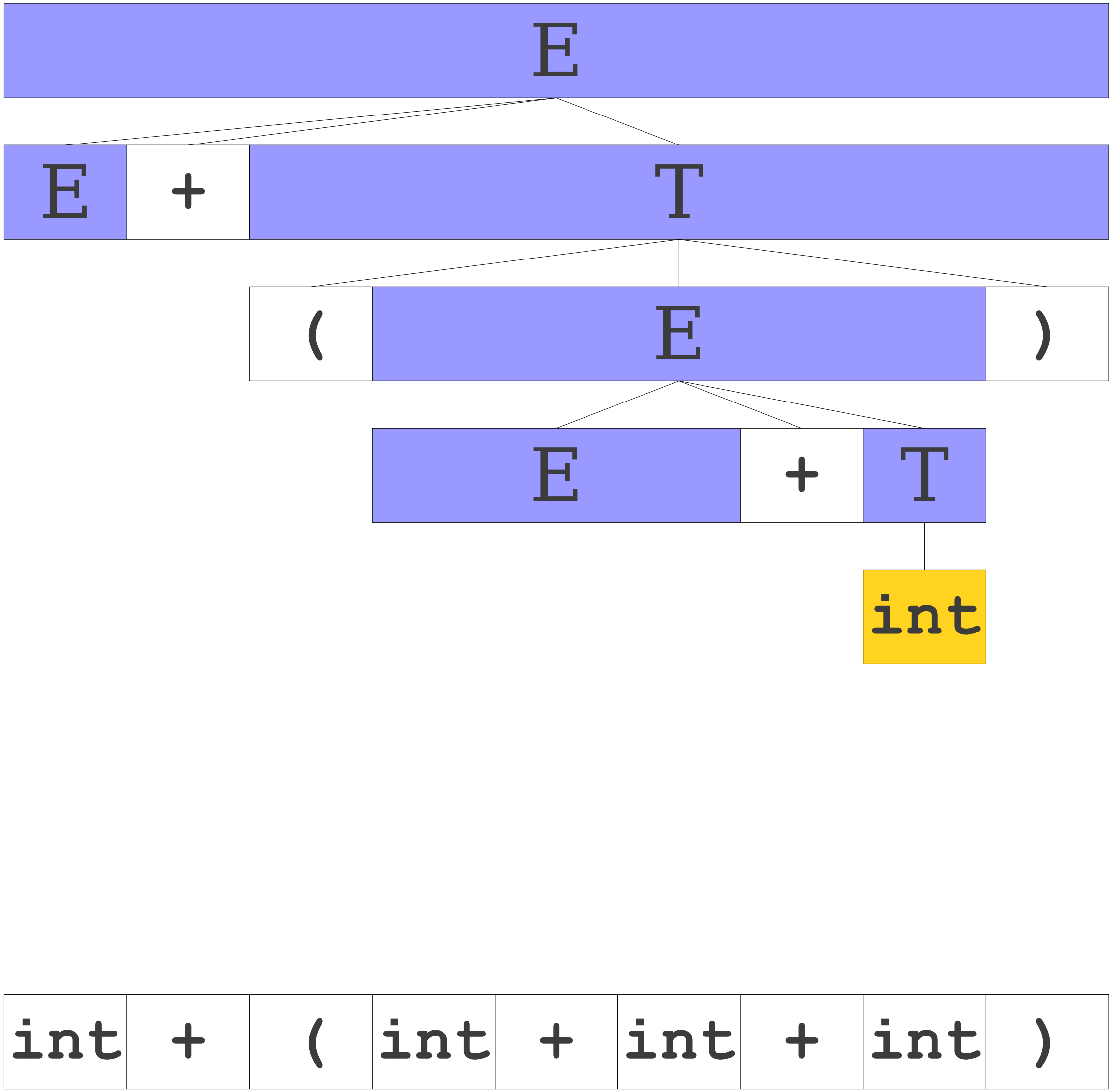
⇒ **E** + (**E** + **T**)

⇒ **E** + (**E**)

⇒ **E** + **T**

⇒ **E**

A Third View of a Bottom-Up Parse



⇒ **E + (E + int)**

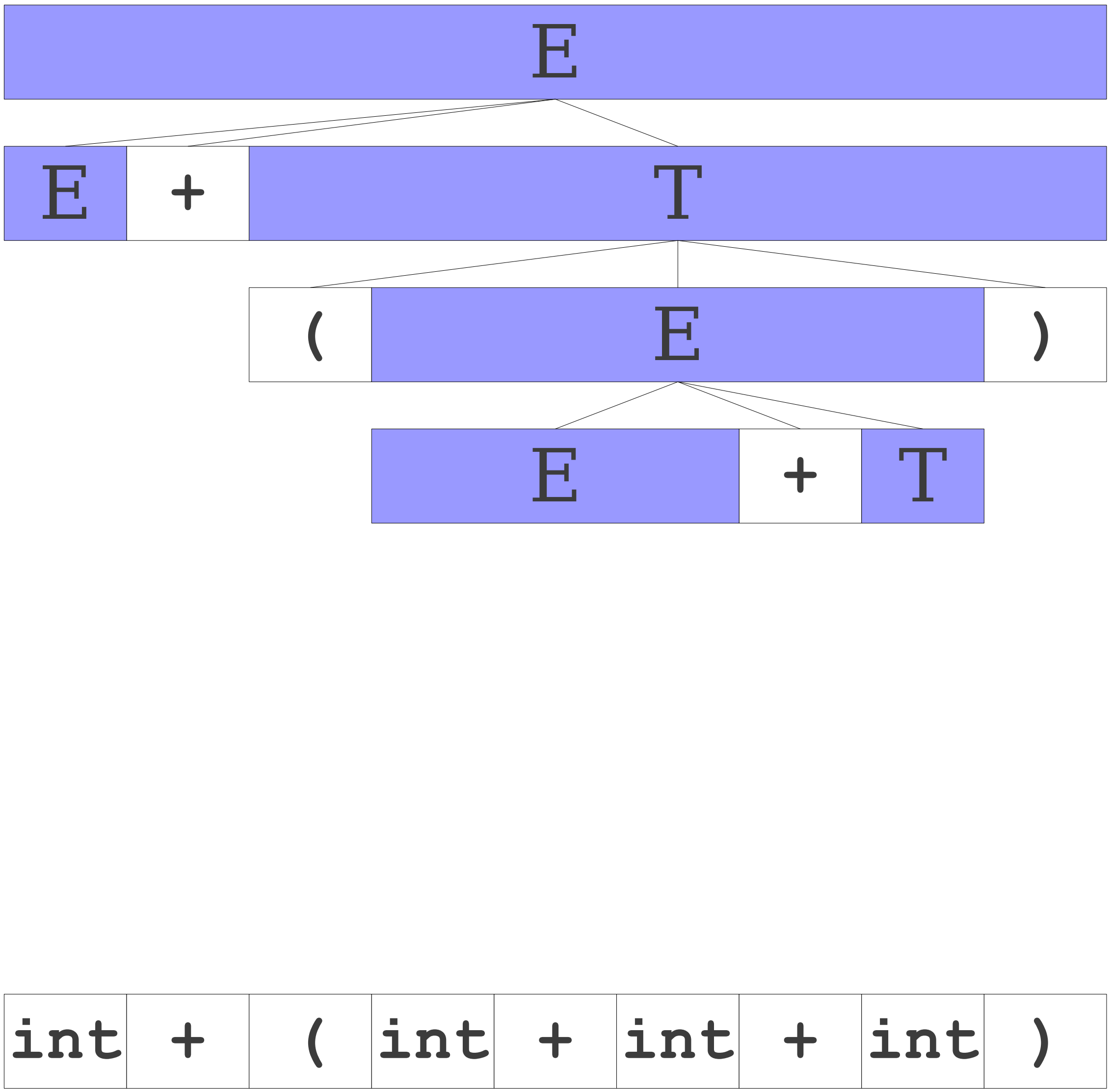
⇒ **E + (E + T)**

⇒ **E + (E)**

⇒ **E + T**

⇒ **E**

A Third View of a Bottom-Up Parse



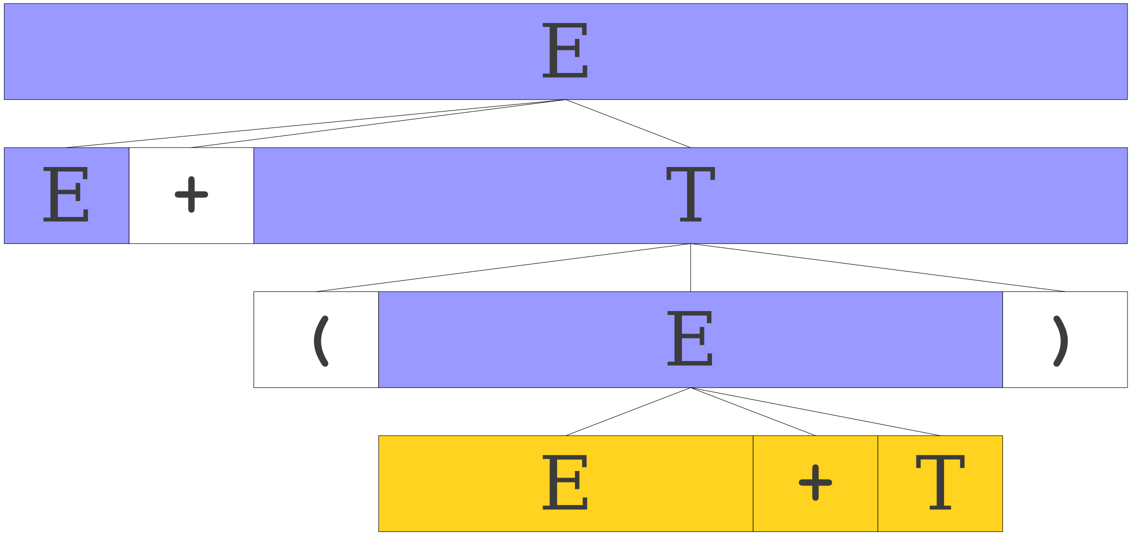
$\Rightarrow$ **E + (E + T)**

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

$\Rightarrow$ **E + T**

$\Rightarrow$ **E**

A Third View of a Bottom-Up Parse

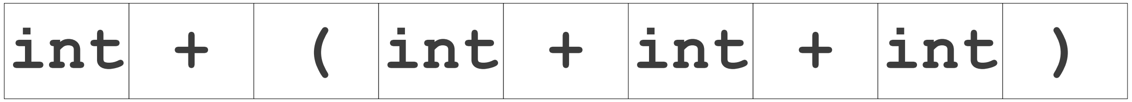


$\Rightarrow$ **E + (E + T)**

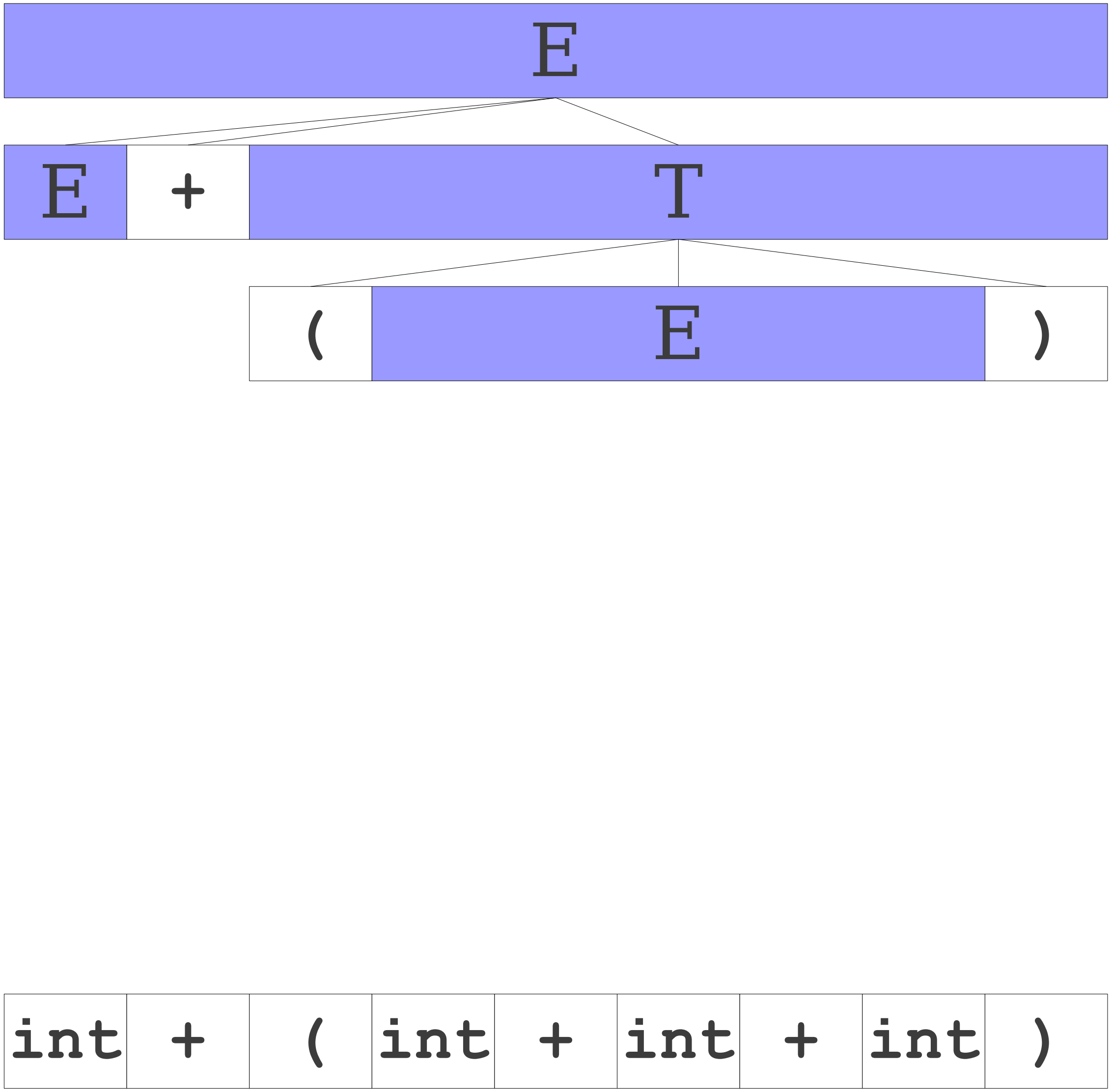
$\Rightarrow$ **E + (E)**

$\Rightarrow$ **E + T**

$\Rightarrow$ **E**



A Third View of a Bottom-Up Parse

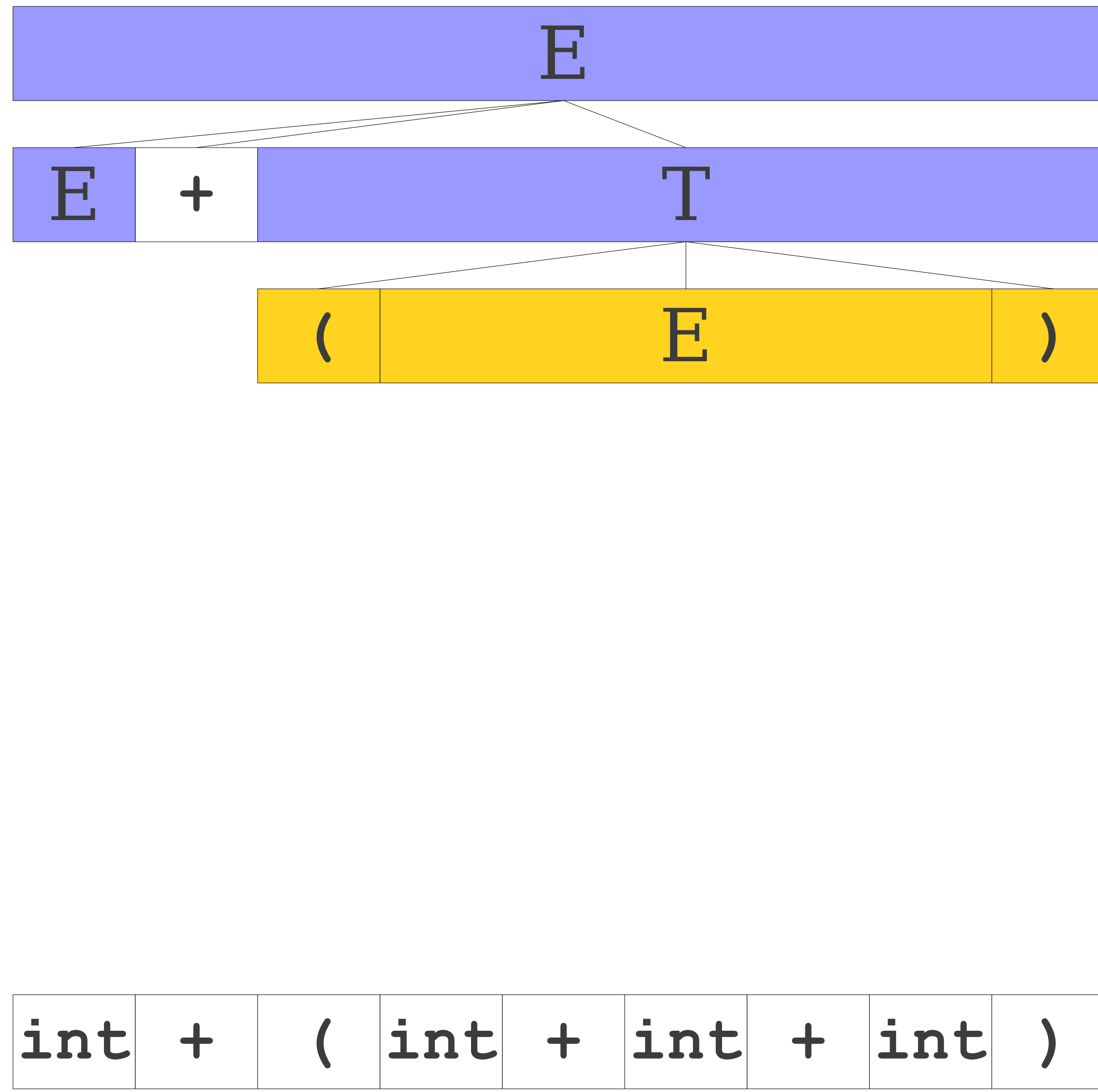


$\Rightarrow$ **E + (E)**

$\Rightarrow$ **E + T**

$\Rightarrow$ **E**

A Third View of a Bottom-Up Parse

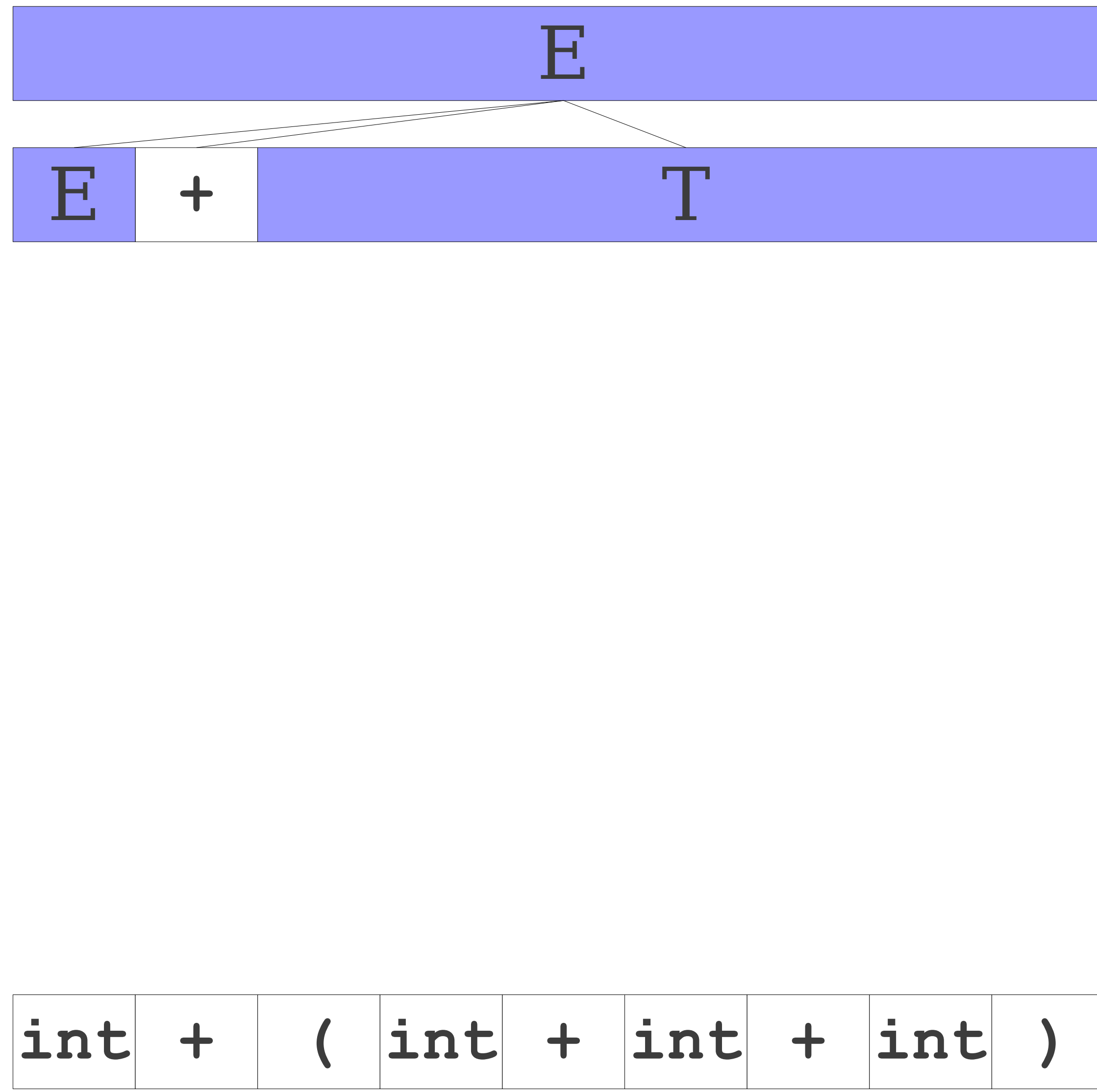


$\Rightarrow$ **E + (E)**

$\Rightarrow$ **E + T**

$\Rightarrow$ **E**

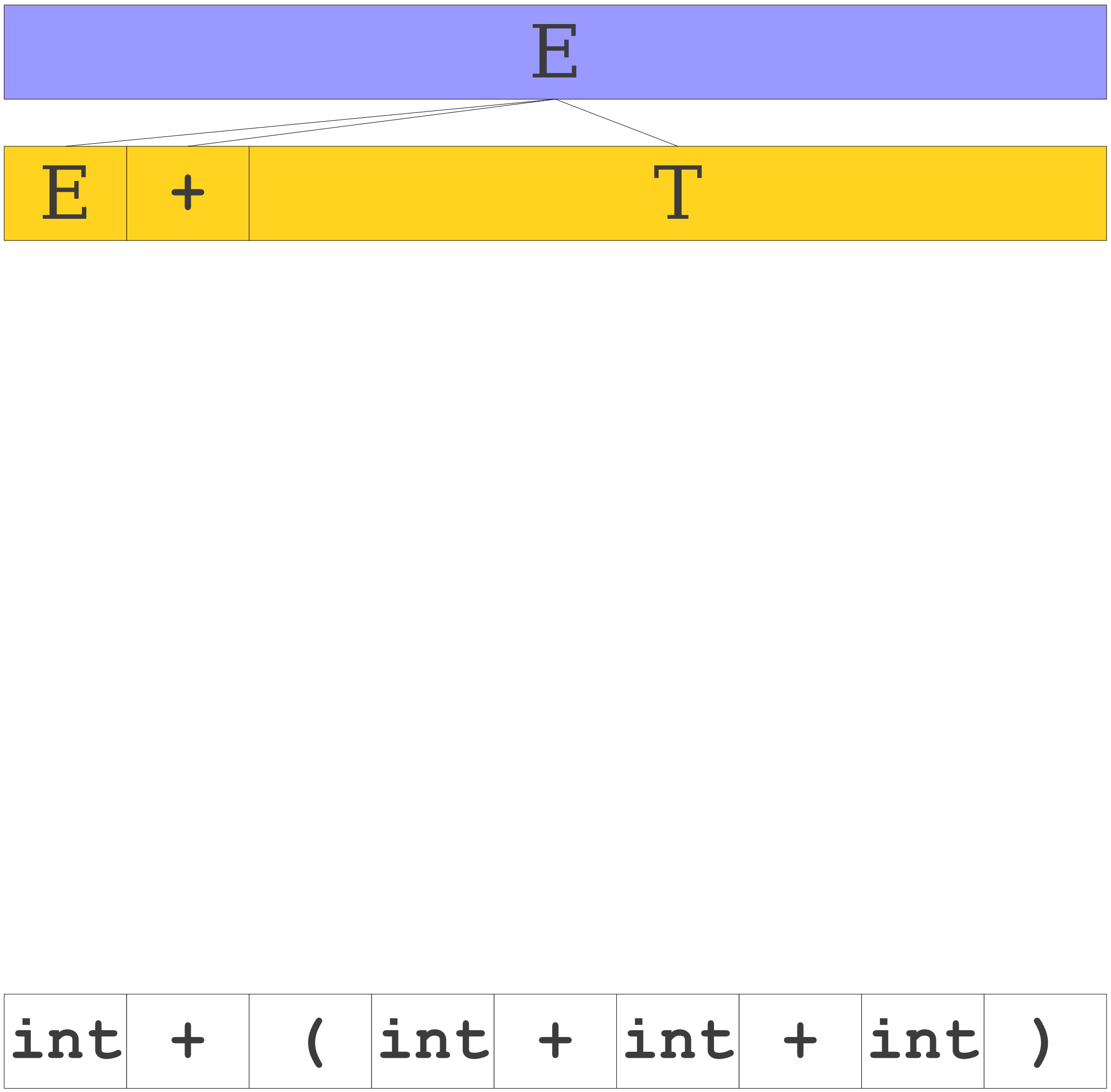
A Third View of a Bottom-Up Parse



$\Rightarrow$ **E + T**

$\Rightarrow$ **E**

A Third View of a Bottom-Up Parse



$\Rightarrow$ **E + T**

$\Rightarrow$ **E**

A Third View of a Bottom-Up Parse

E								

$\Rightarrow$ **E**

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

Handles

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

Question One:

Where are handles?

A Sample Shift/Reduce Parse

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

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

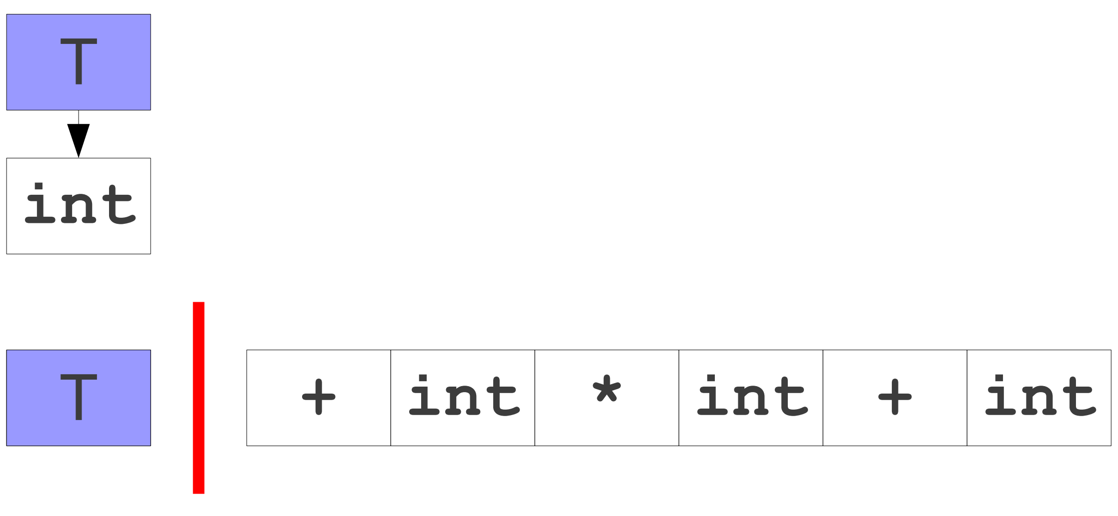
A Sample Shift/Reduce Parse

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

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

A Sample Shift/Reduce Parse

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



A Sample Shift/Reduce Parse

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

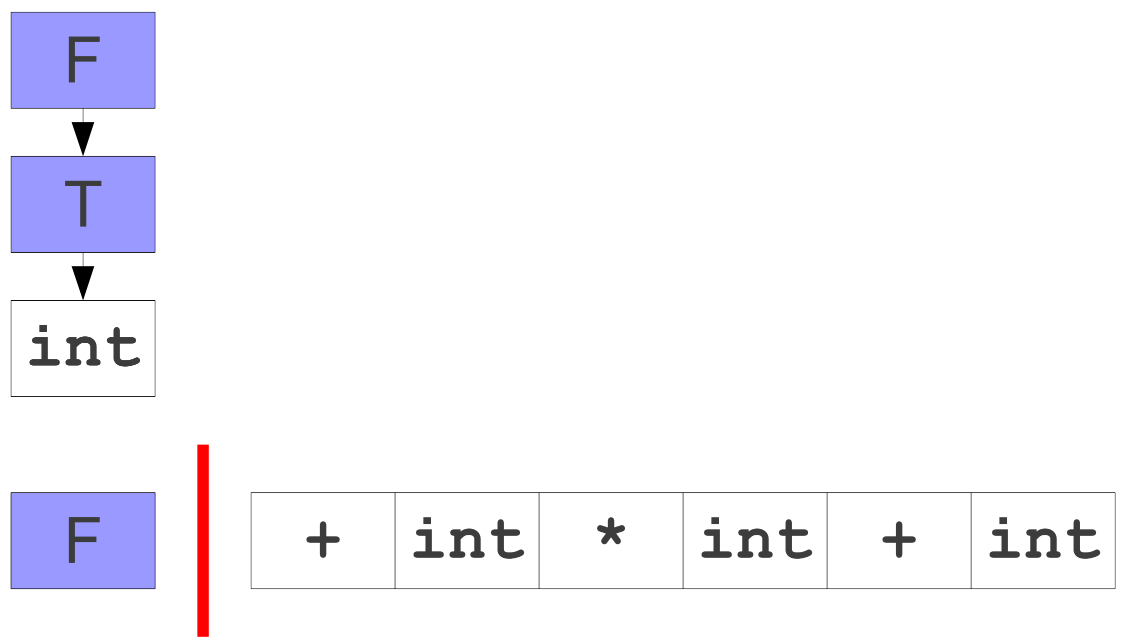
$\mathbf{E} \rightarrow \mathbf{E}$ **+** $\mathbf{F}$

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

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

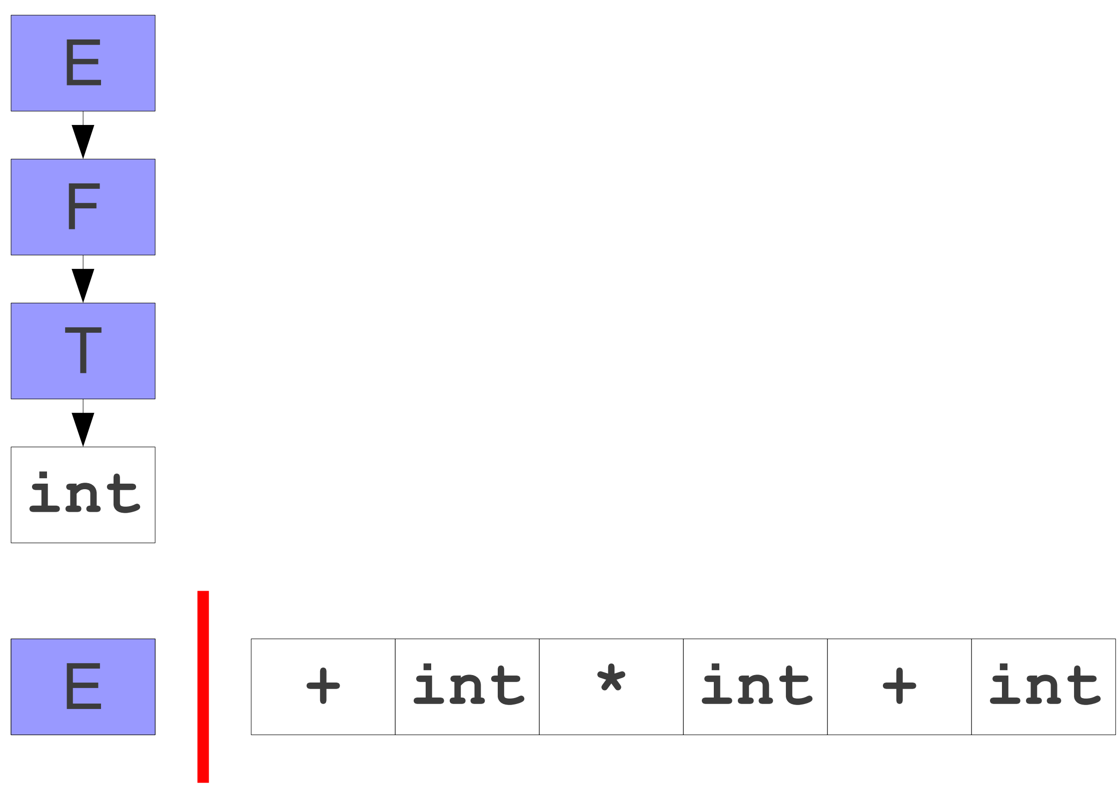
$\mathbf{T} \rightarrow$ **int**

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



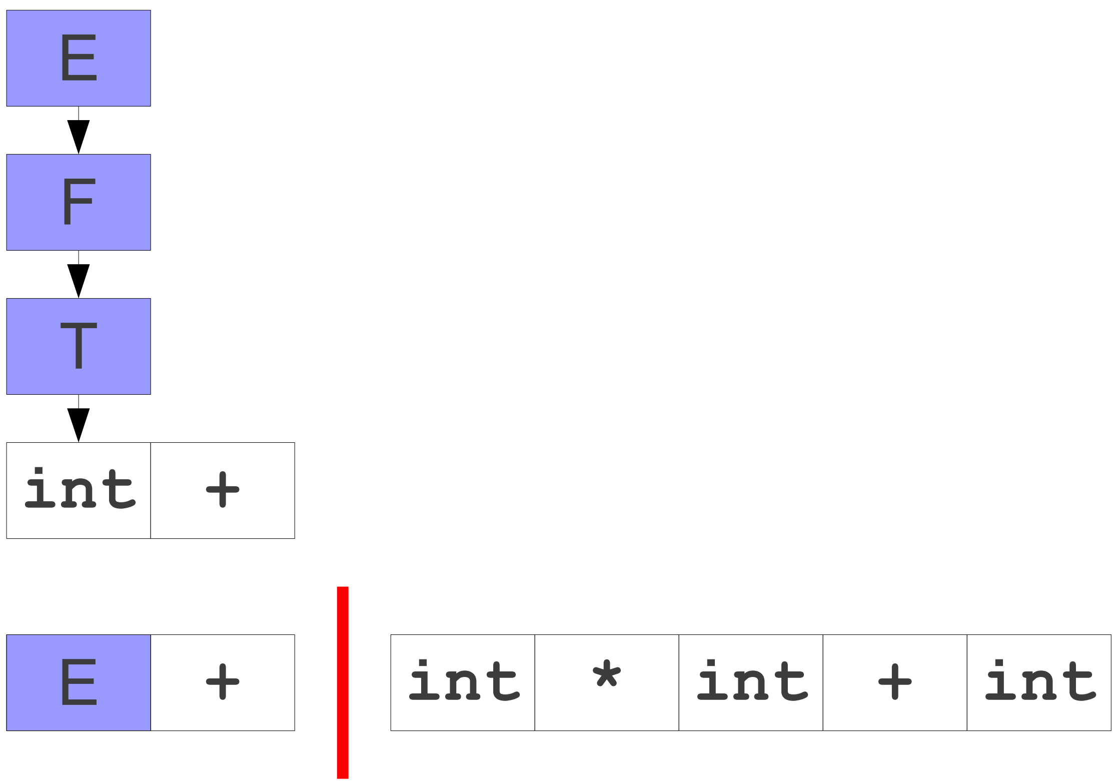
A Sample Shift/Reduce Parse

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



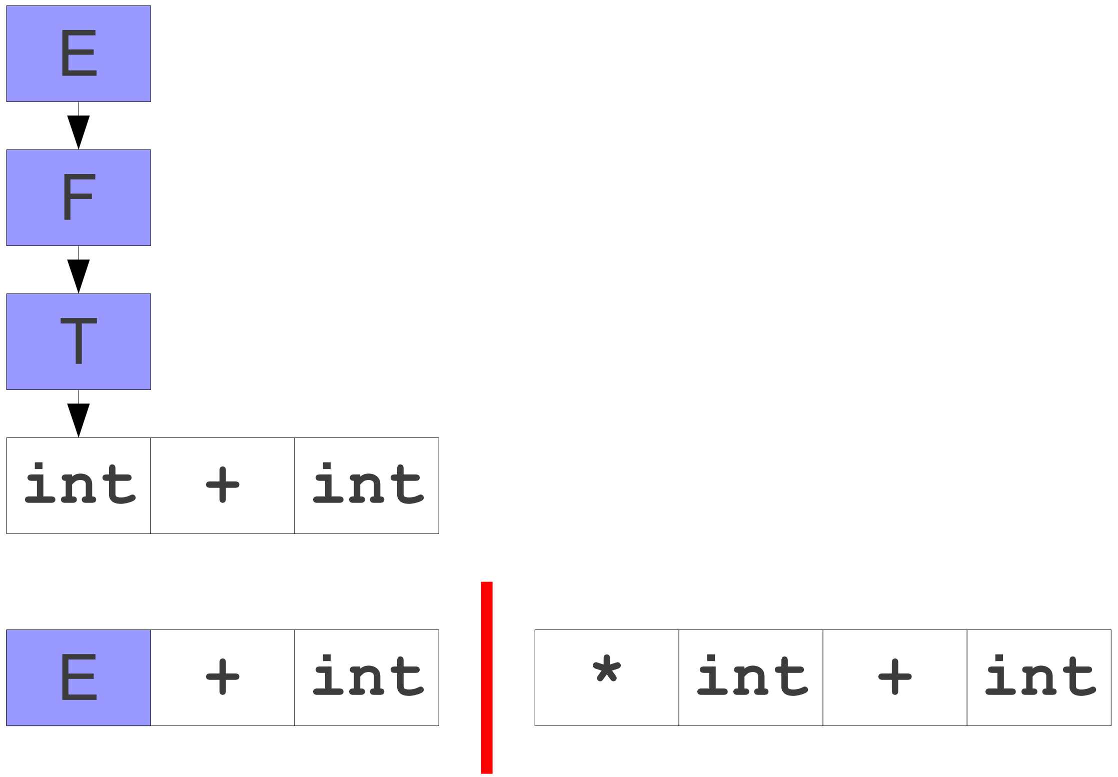
A Sample Shift/Reduce Parse

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



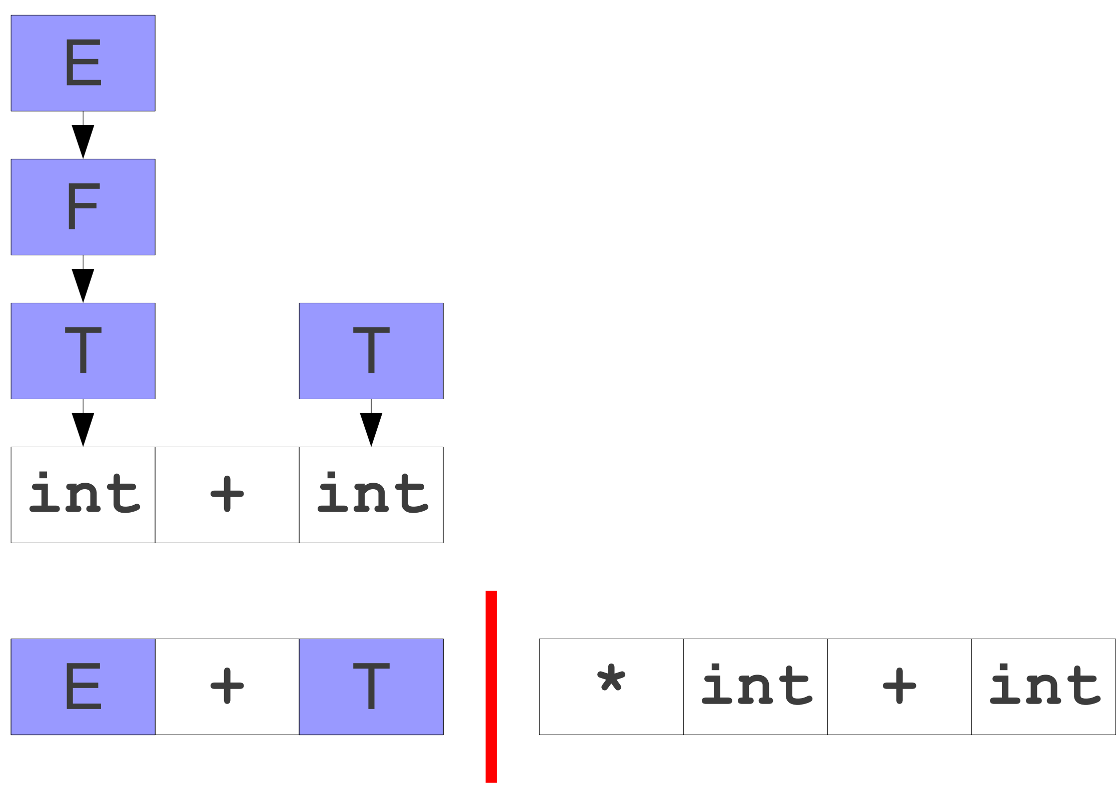
A Sample Shift/Reduce Parse

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



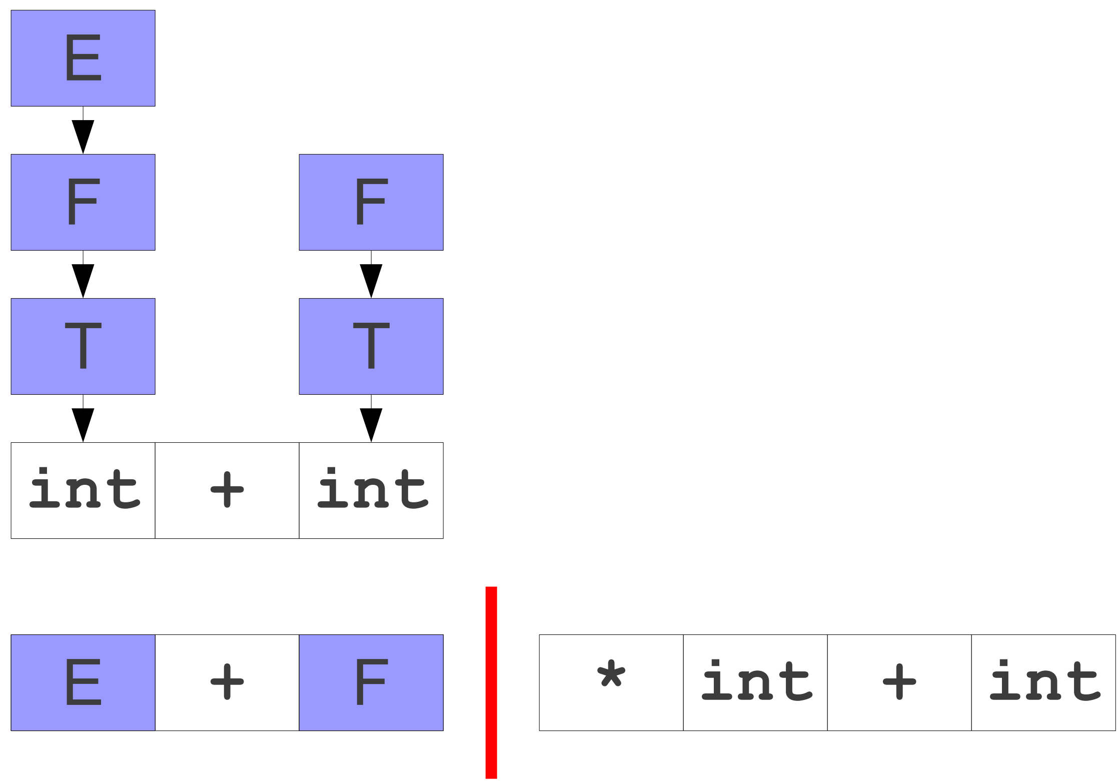
A Sample Shift/Reduce Parse

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



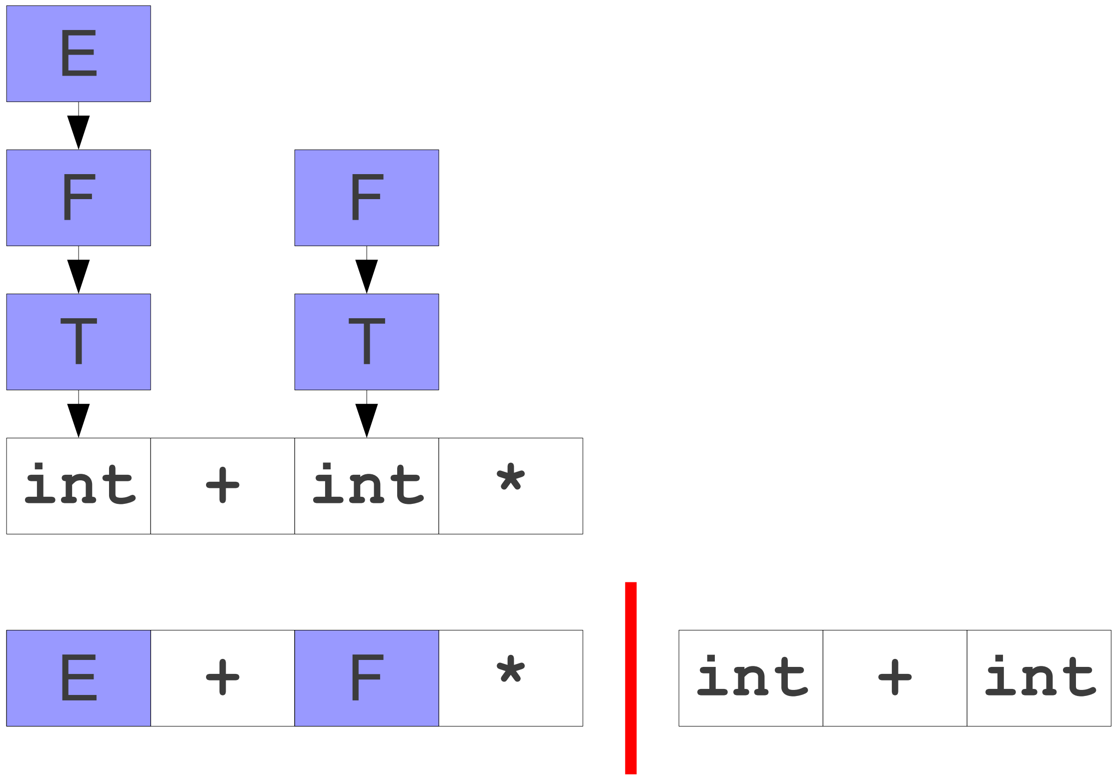
A Sample Shift/Reduce Parse

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



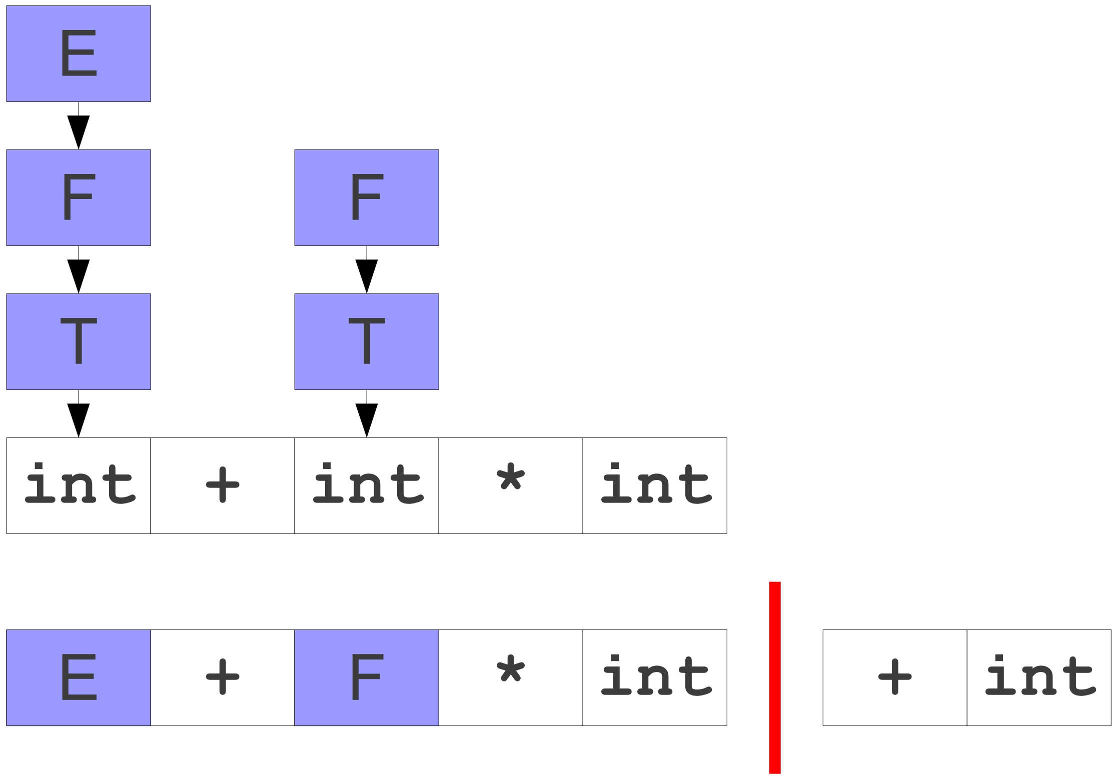
A Sample Shift/Reduce Parse

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



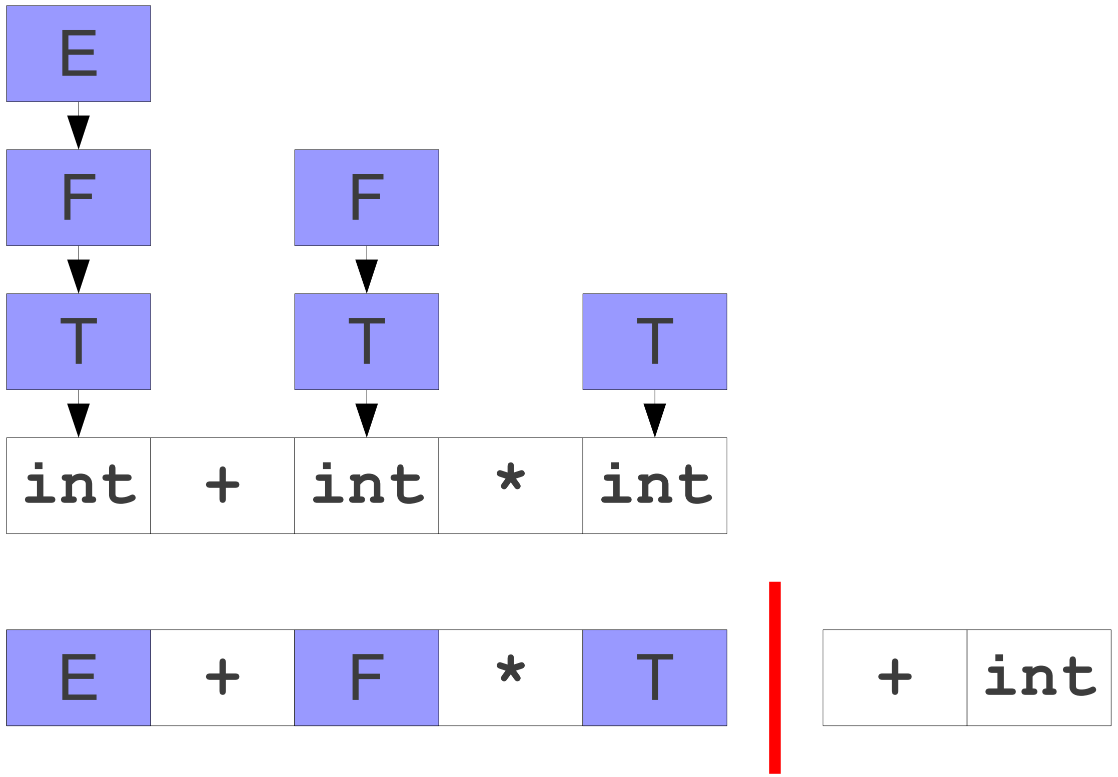
A Sample Shift/Reduce Parse

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



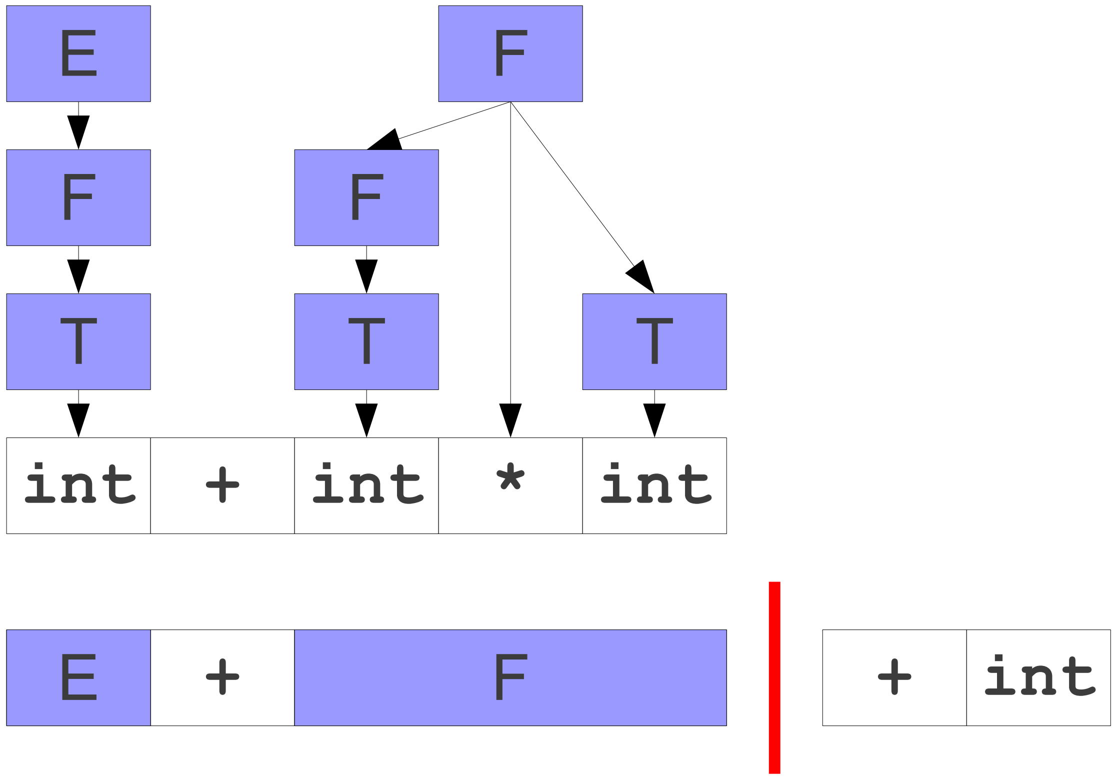
A Sample Shift/Reduce Parse

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



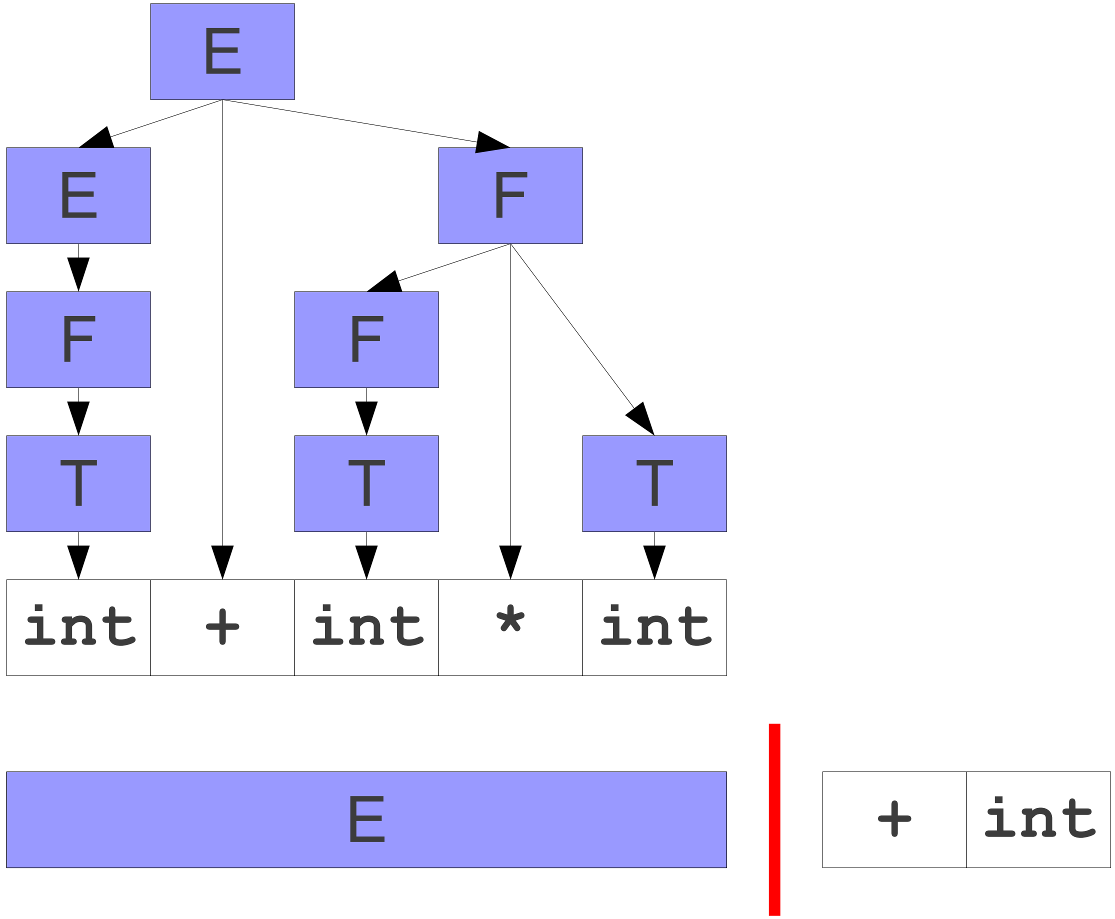
A Sample Shift/Reduce Parse

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



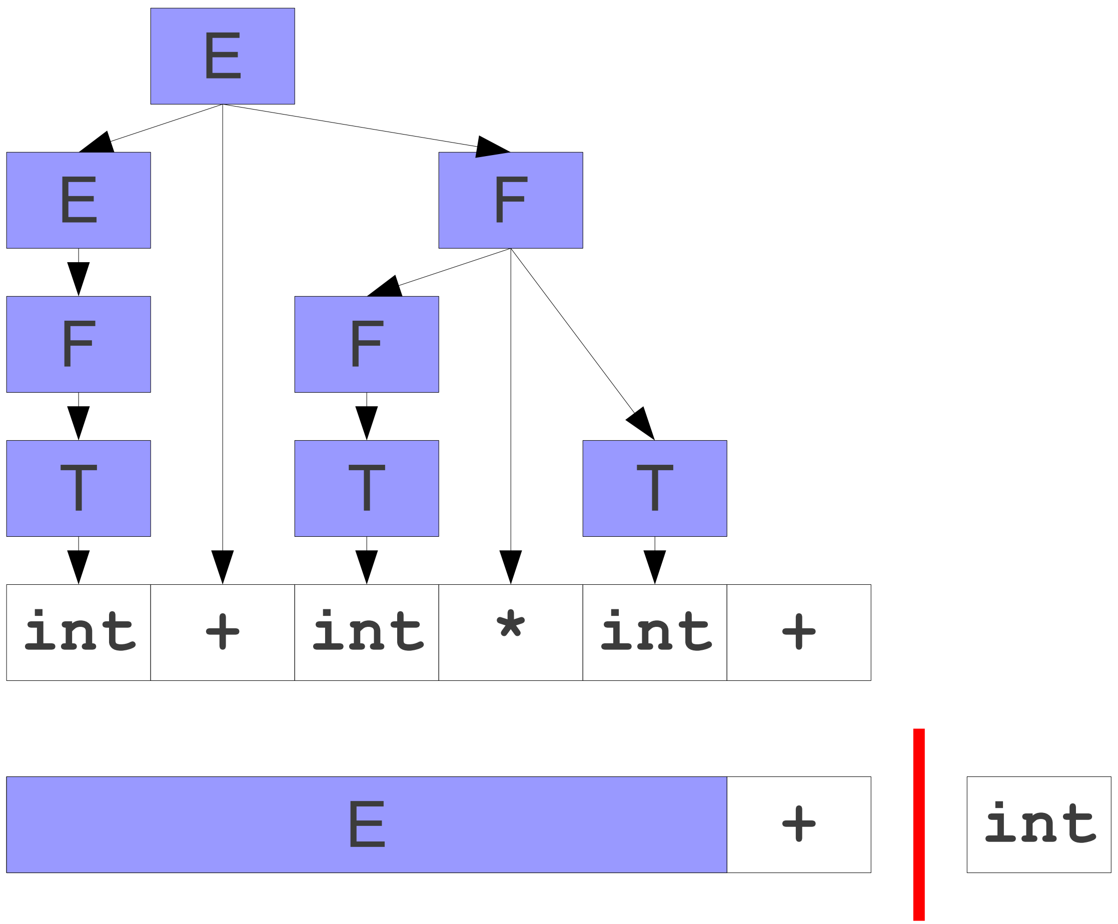
A Sample Shift/Reduce Parse

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



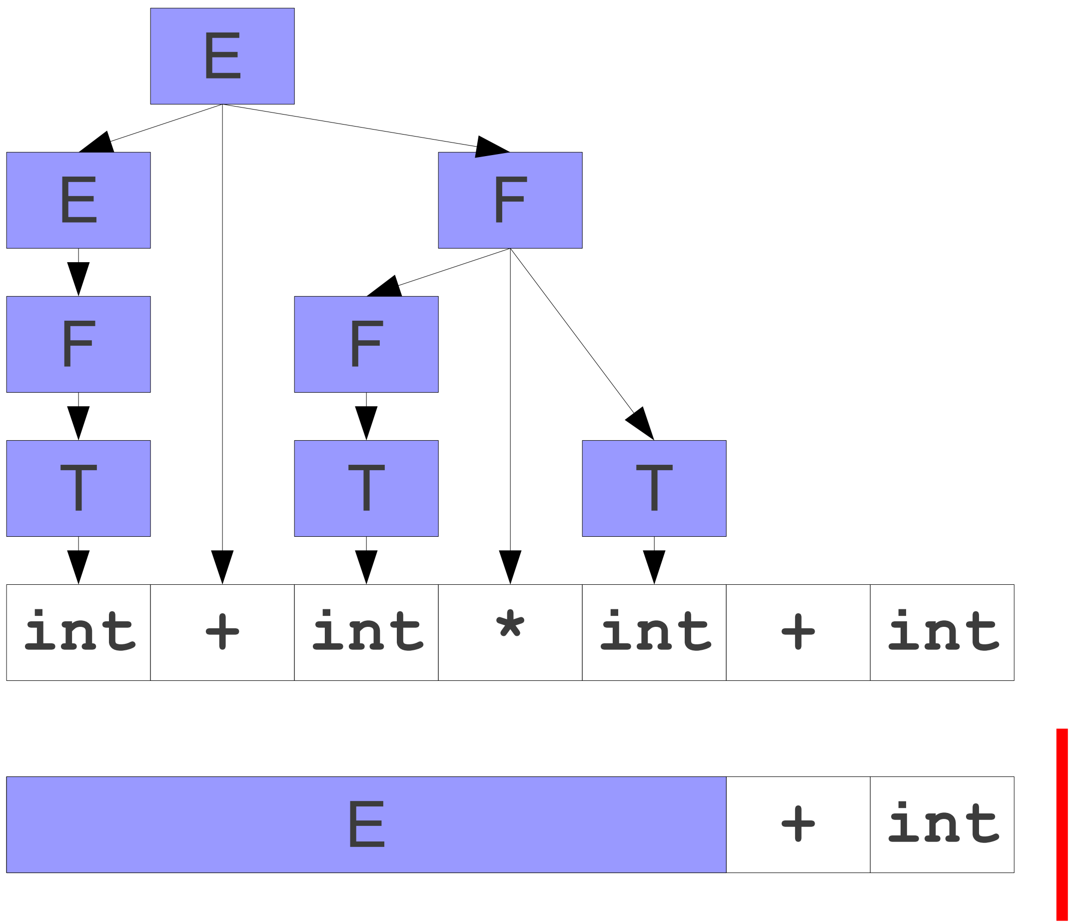
A Sample Shift/Reduce Parse

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



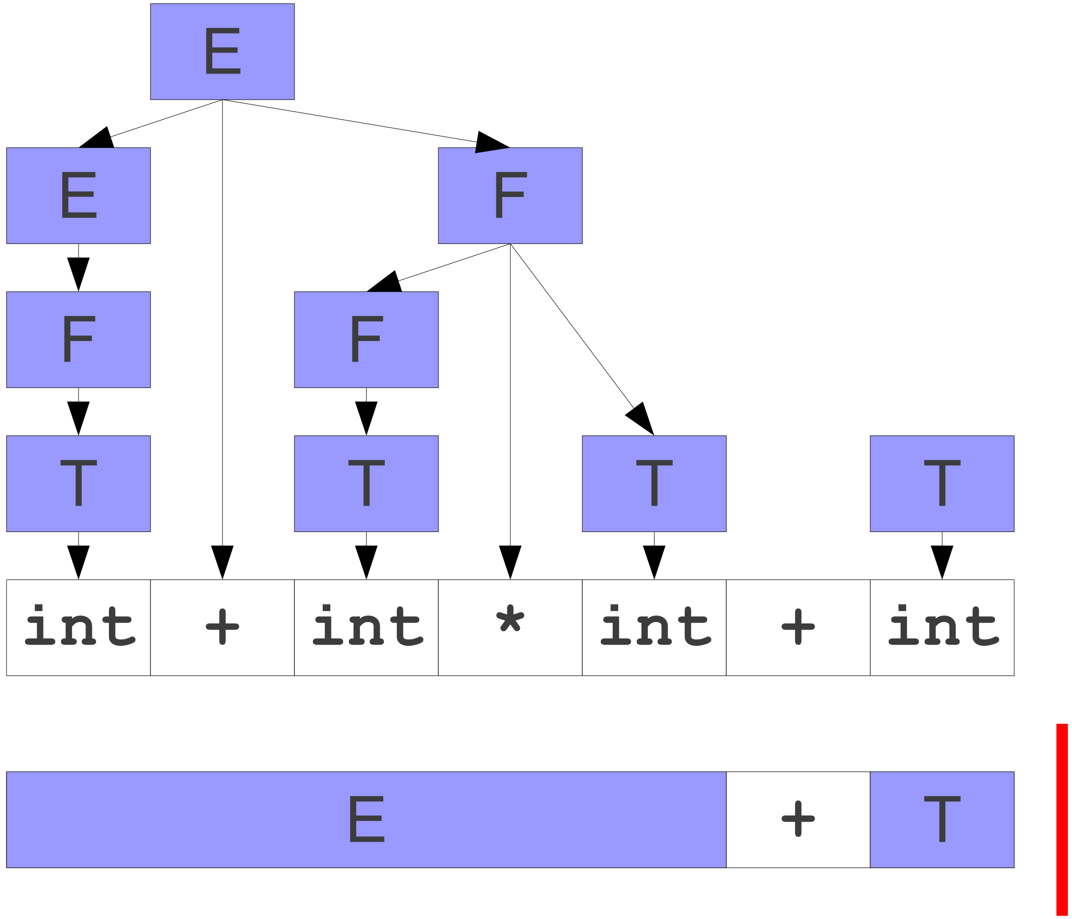
A Sample Shift/Reduce Parse

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



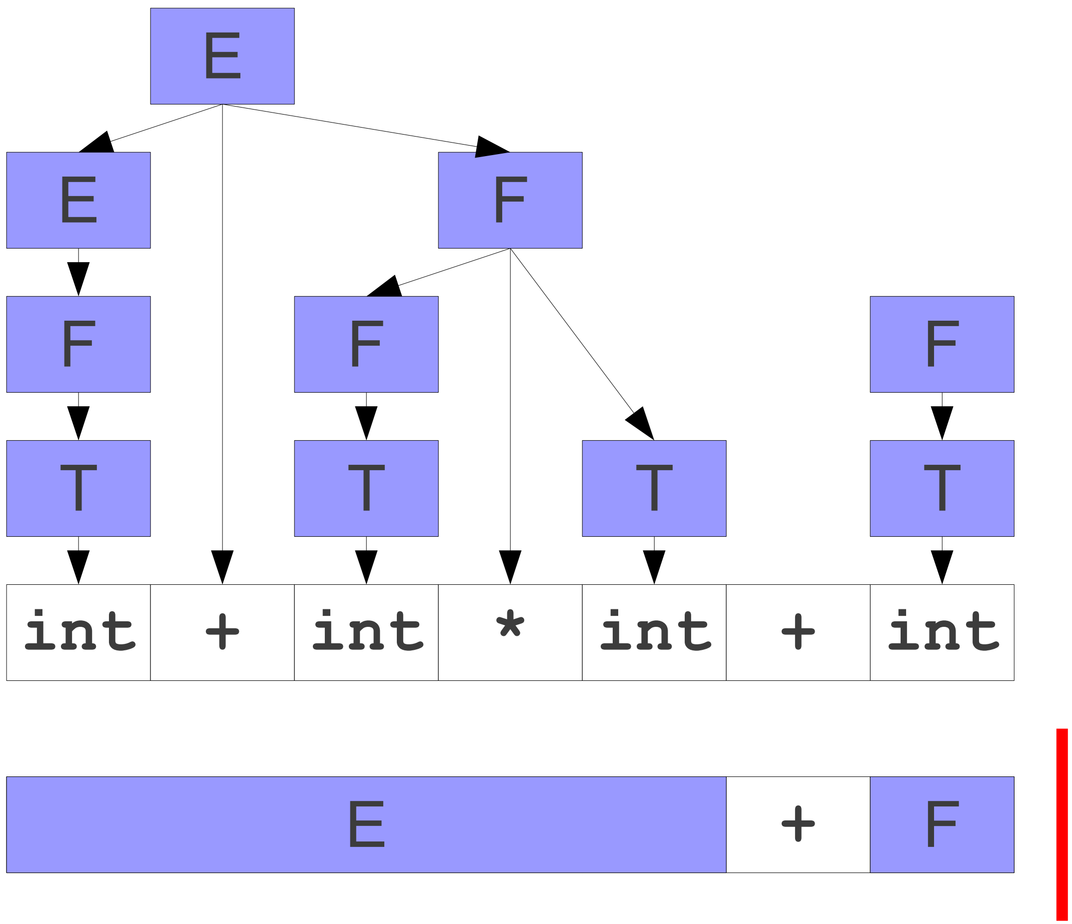
A Sample Shift/Reduce Parse

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



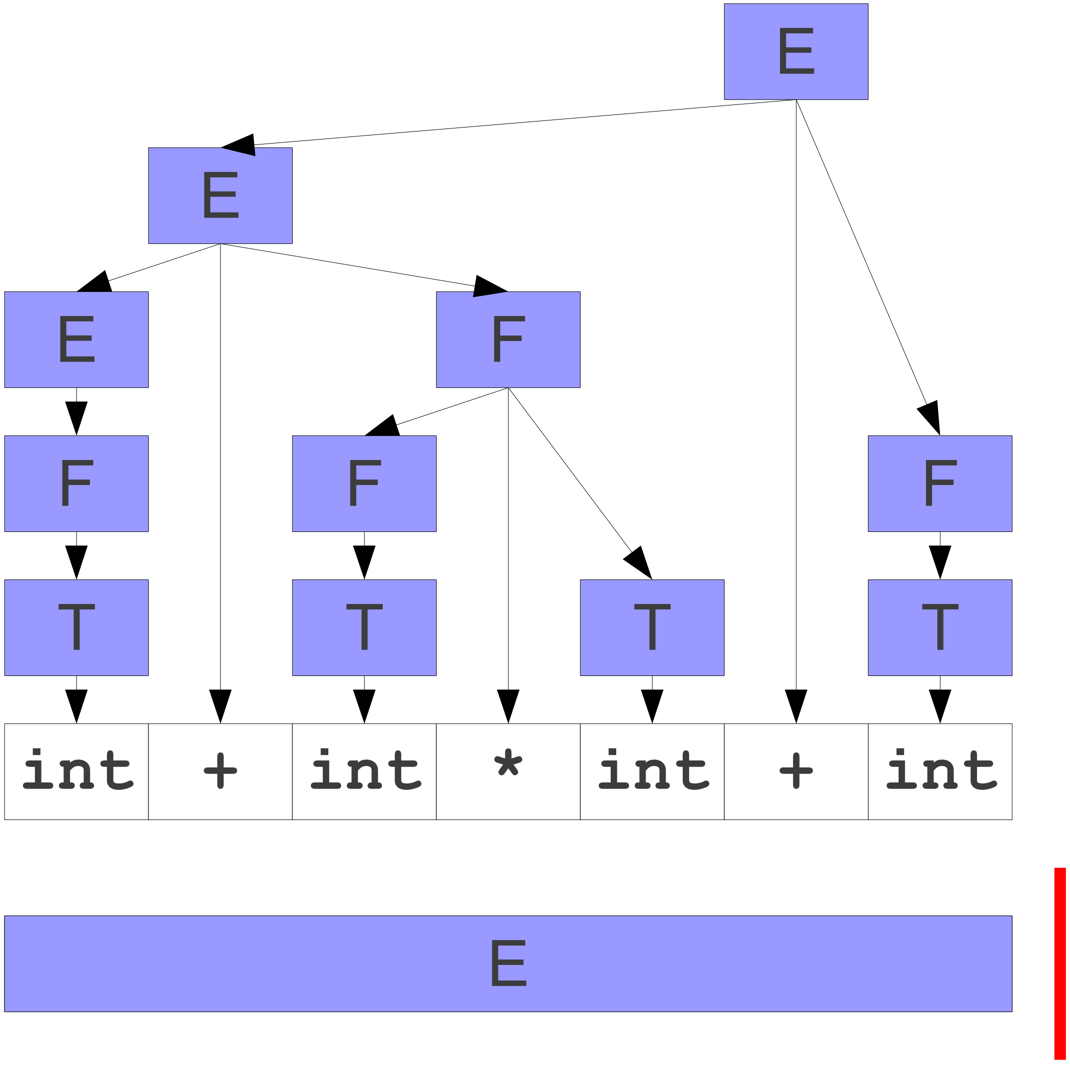
A Sample Shift/Reduce Parse

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



An Important Corollary

- Since reductions are always at the right side of the left area, we never need to shift from the left to the right.
- No need to “uncover” something to do a reduction.
- Consequently, shift/reduce parsing means
  - **Shift**: Move a terminal from the right to the left area.
  - **Reduce**: Replace some number of symbols at the right side of the left area.

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

$\mathbf{E} \to \mathbf{F}$

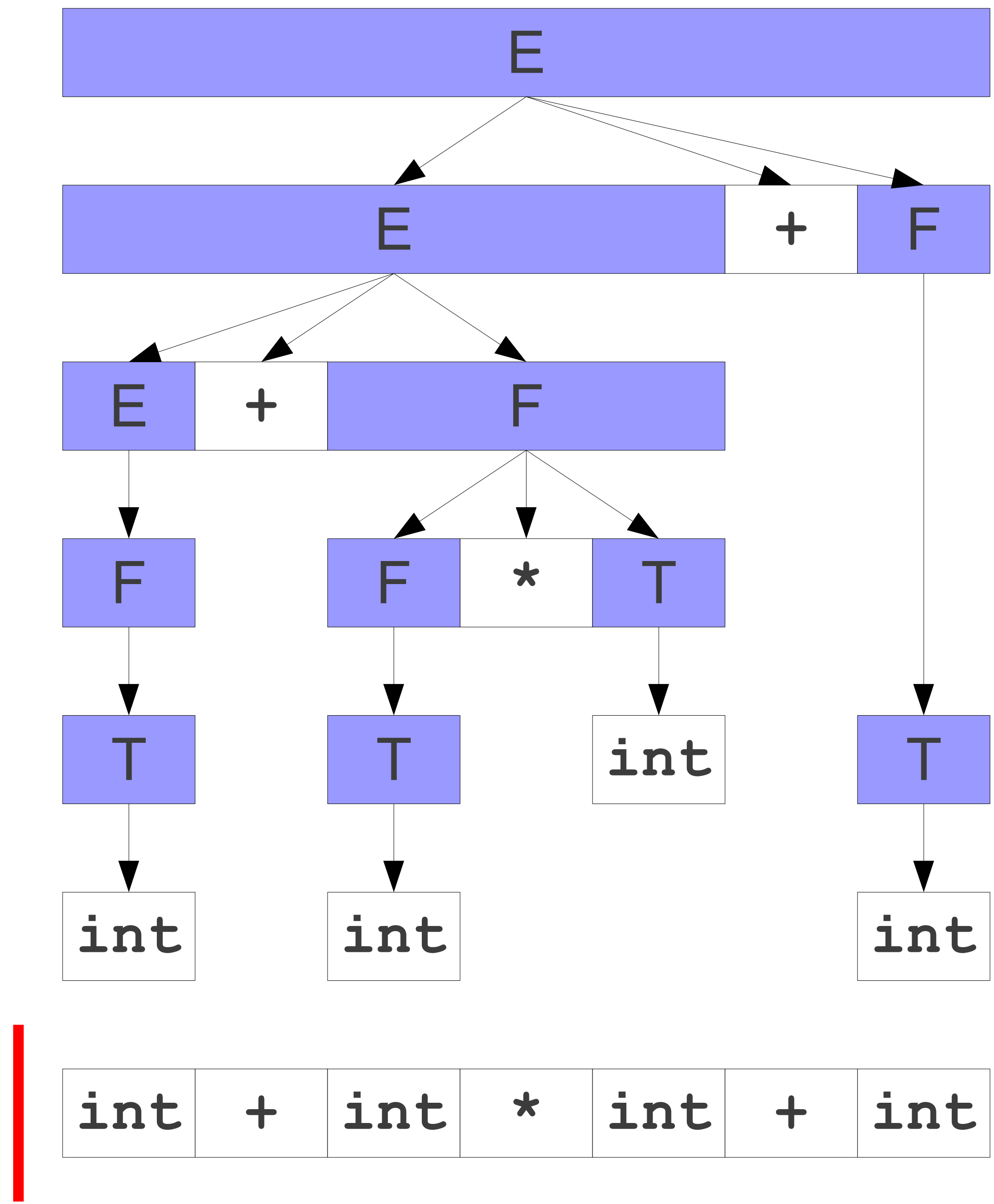
$\mathbf{E} \to \mathbf{E} + \mathbf{F}$

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

$\mathbf{F} \to \mathbf{T}$

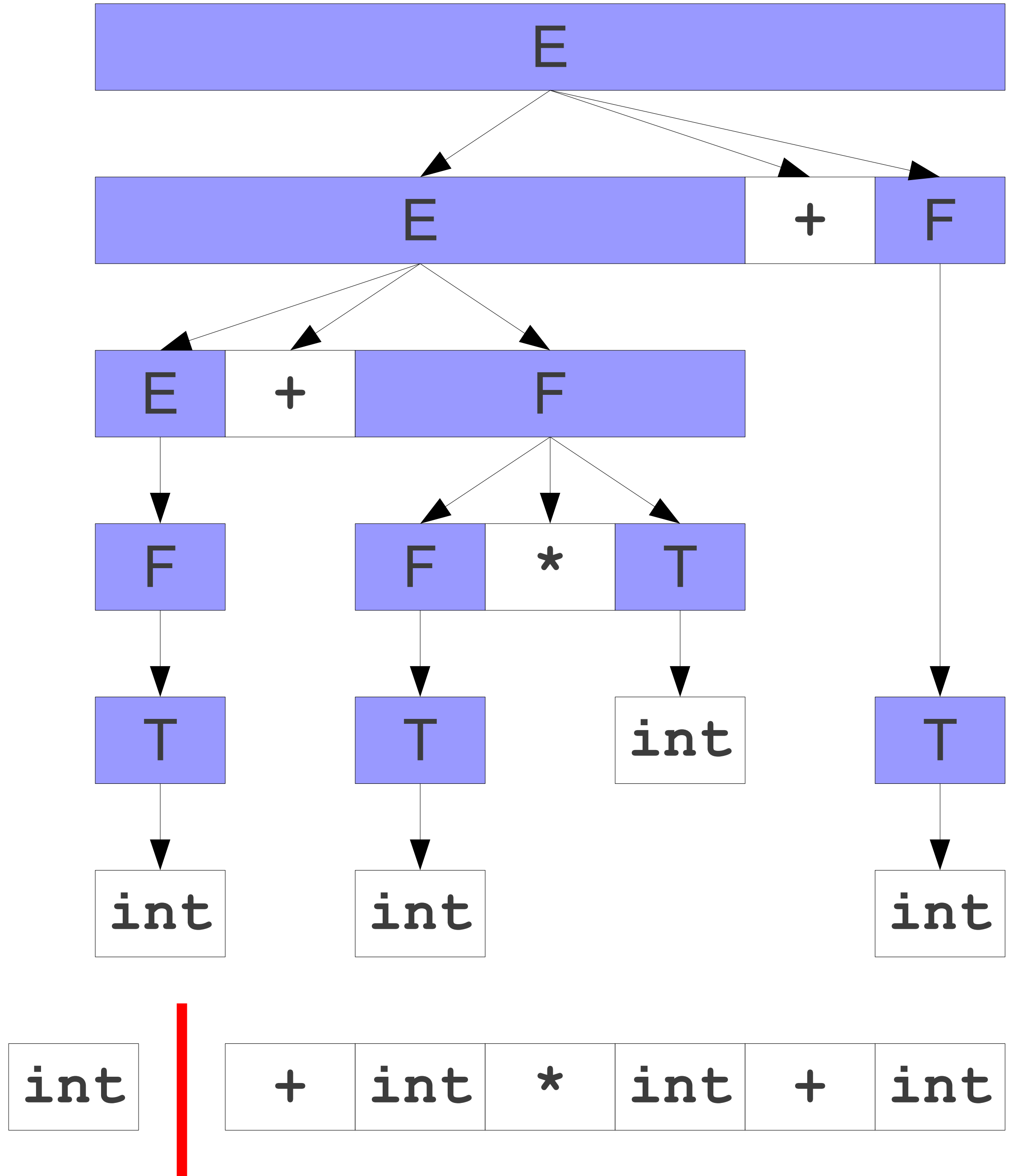
$\mathbf{T} \to$ `int`

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



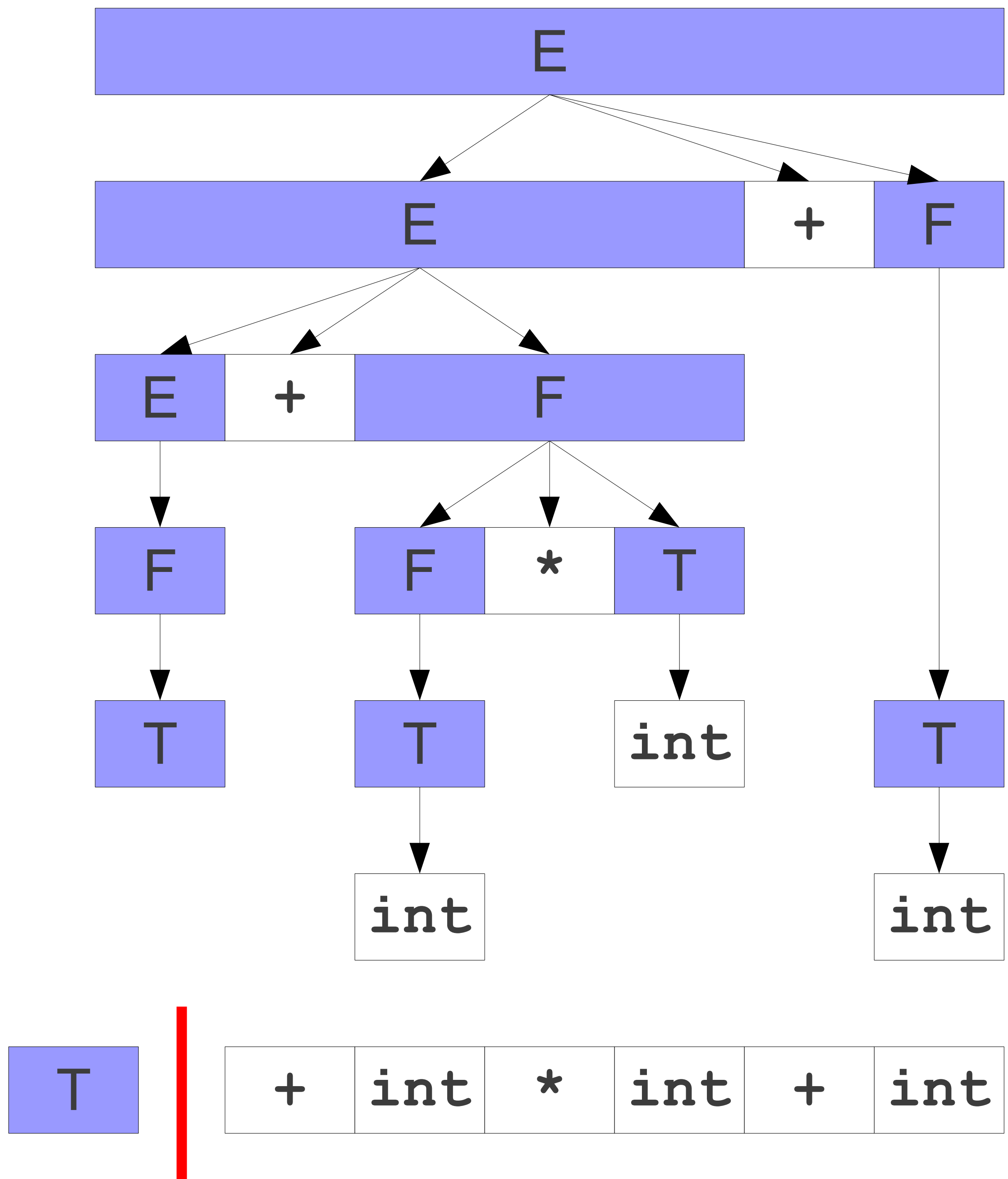
Another Look at Handles

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



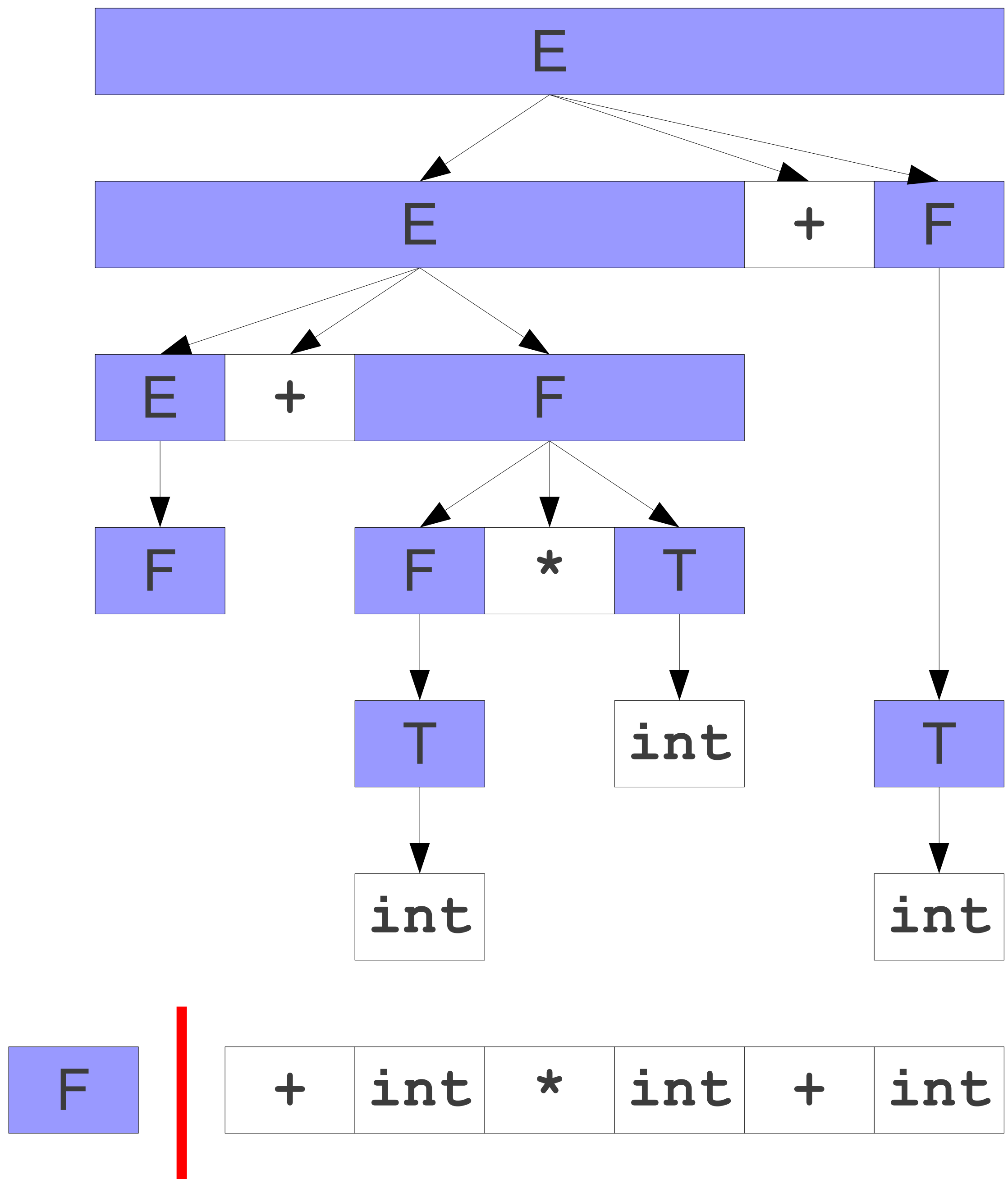
Another Look at Handles

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



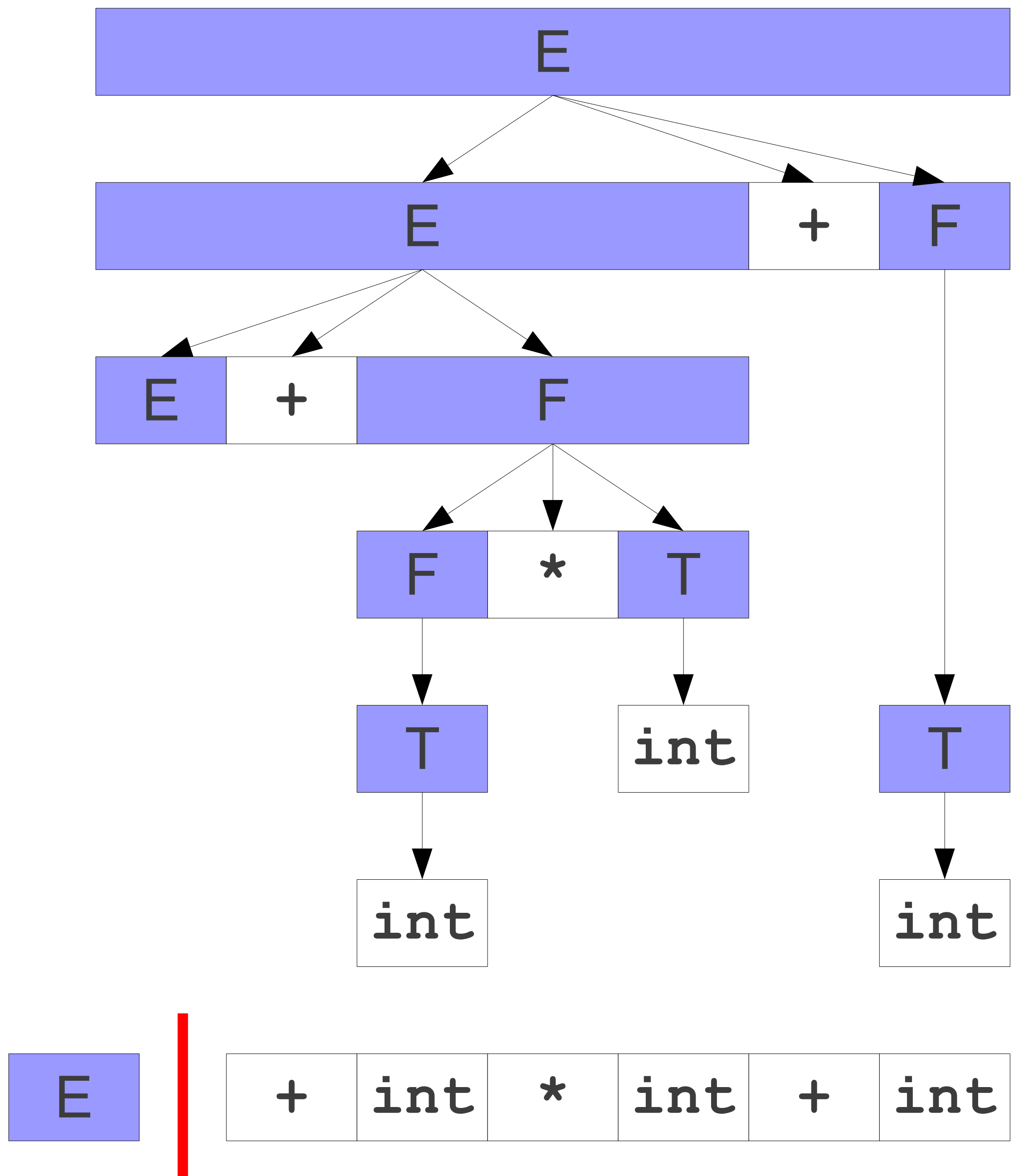
Another Look at Handles

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



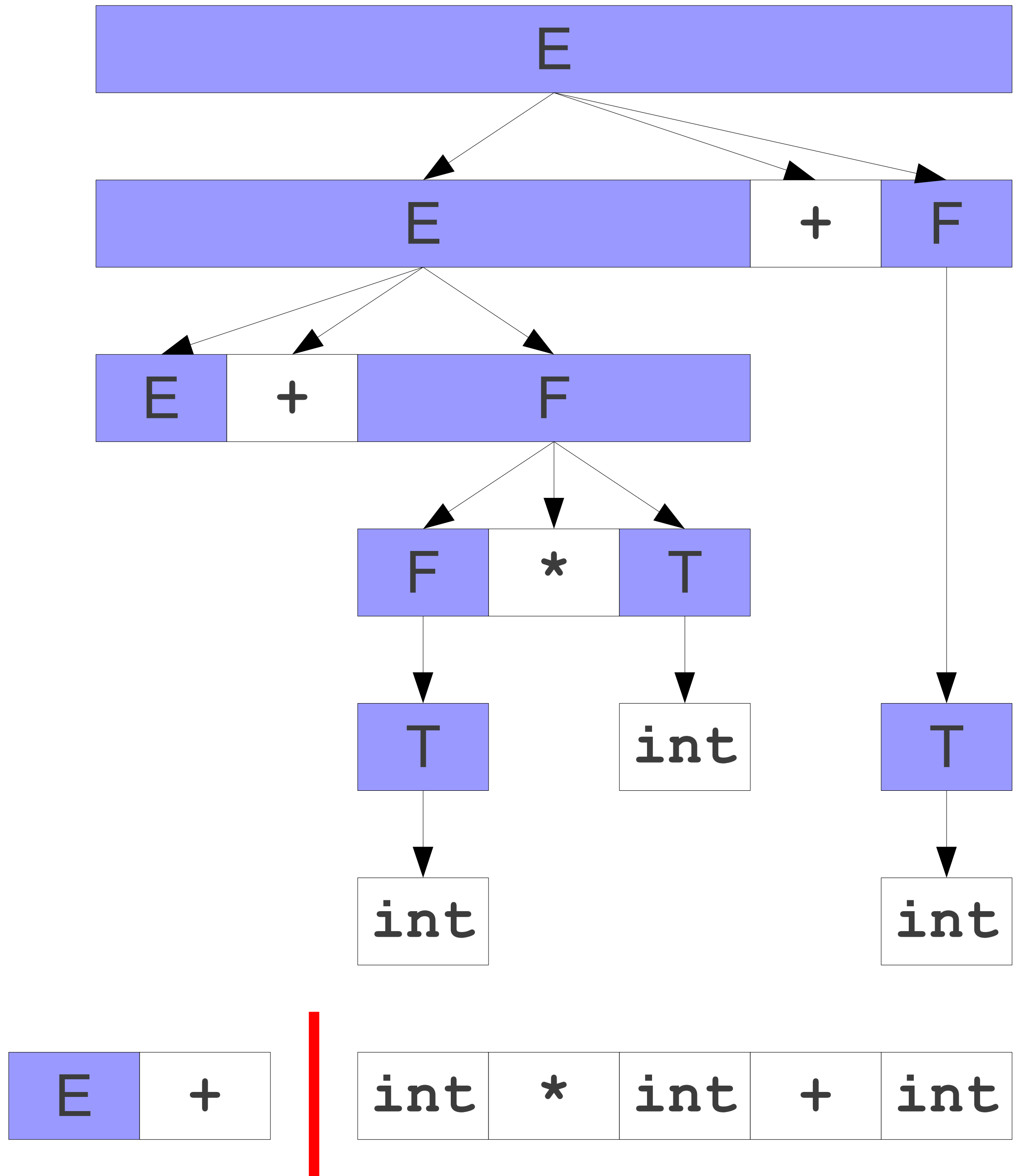
Another Look at Handles

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



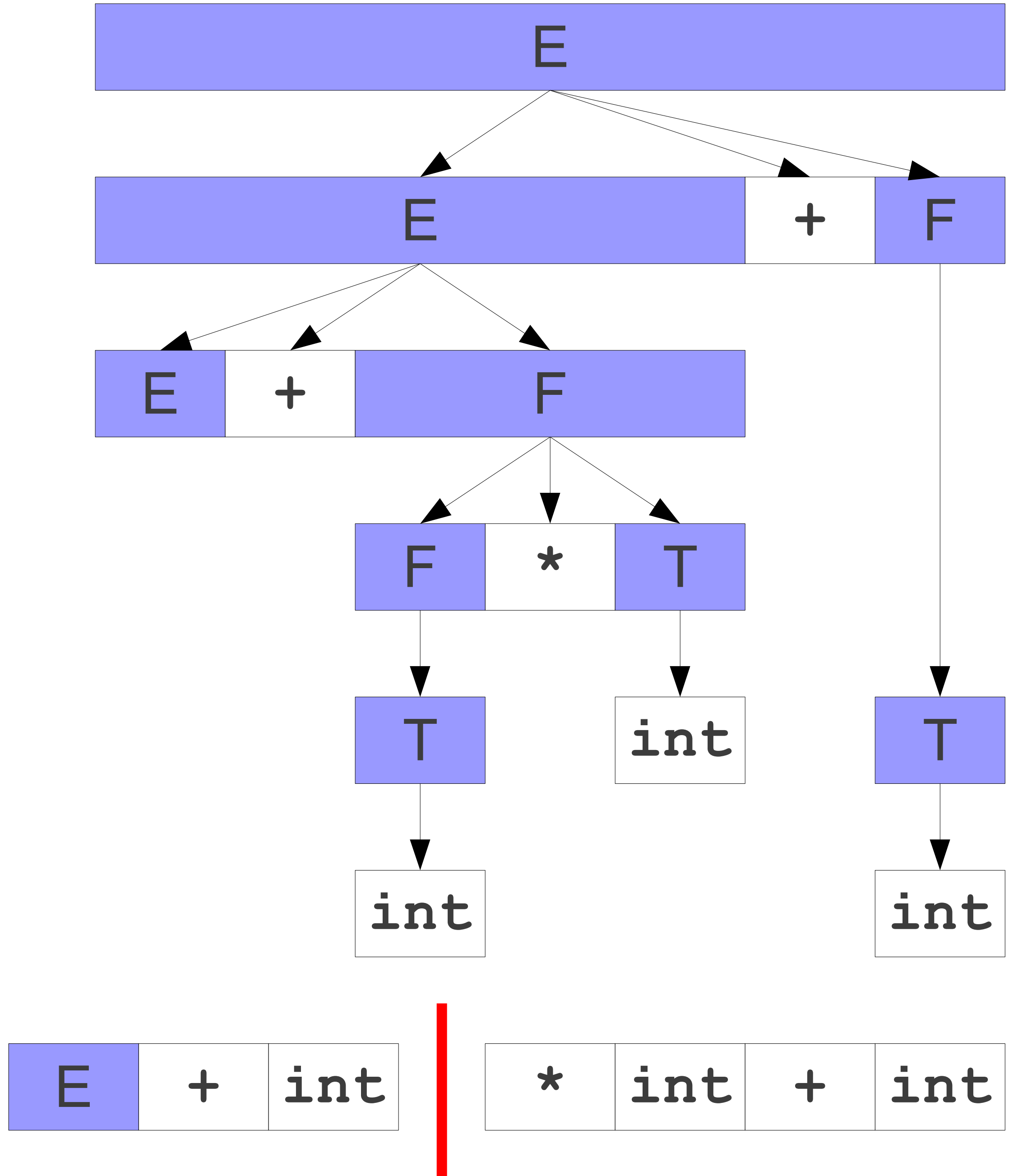
Another Look at Handles

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



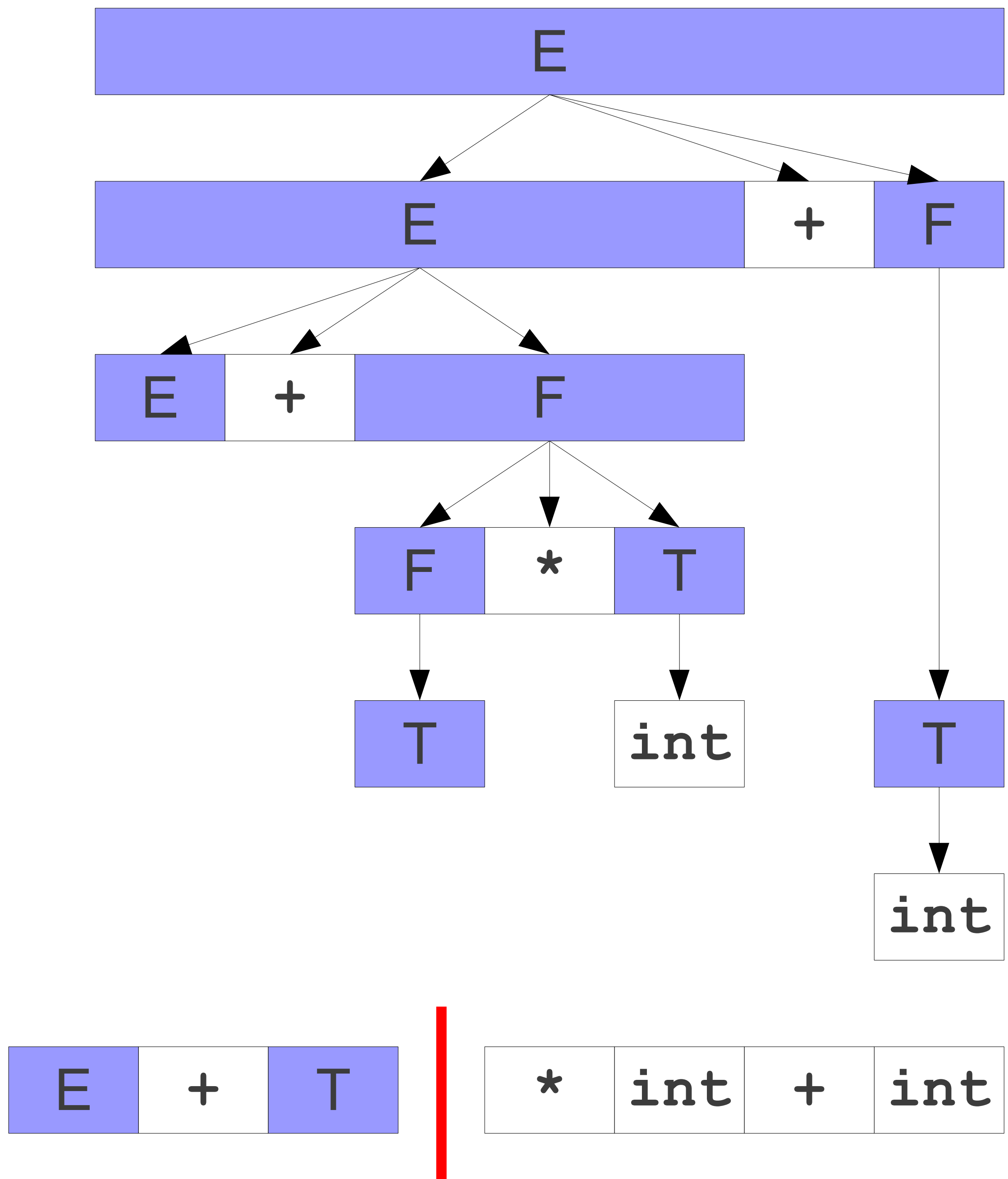
Another Look at Handles

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



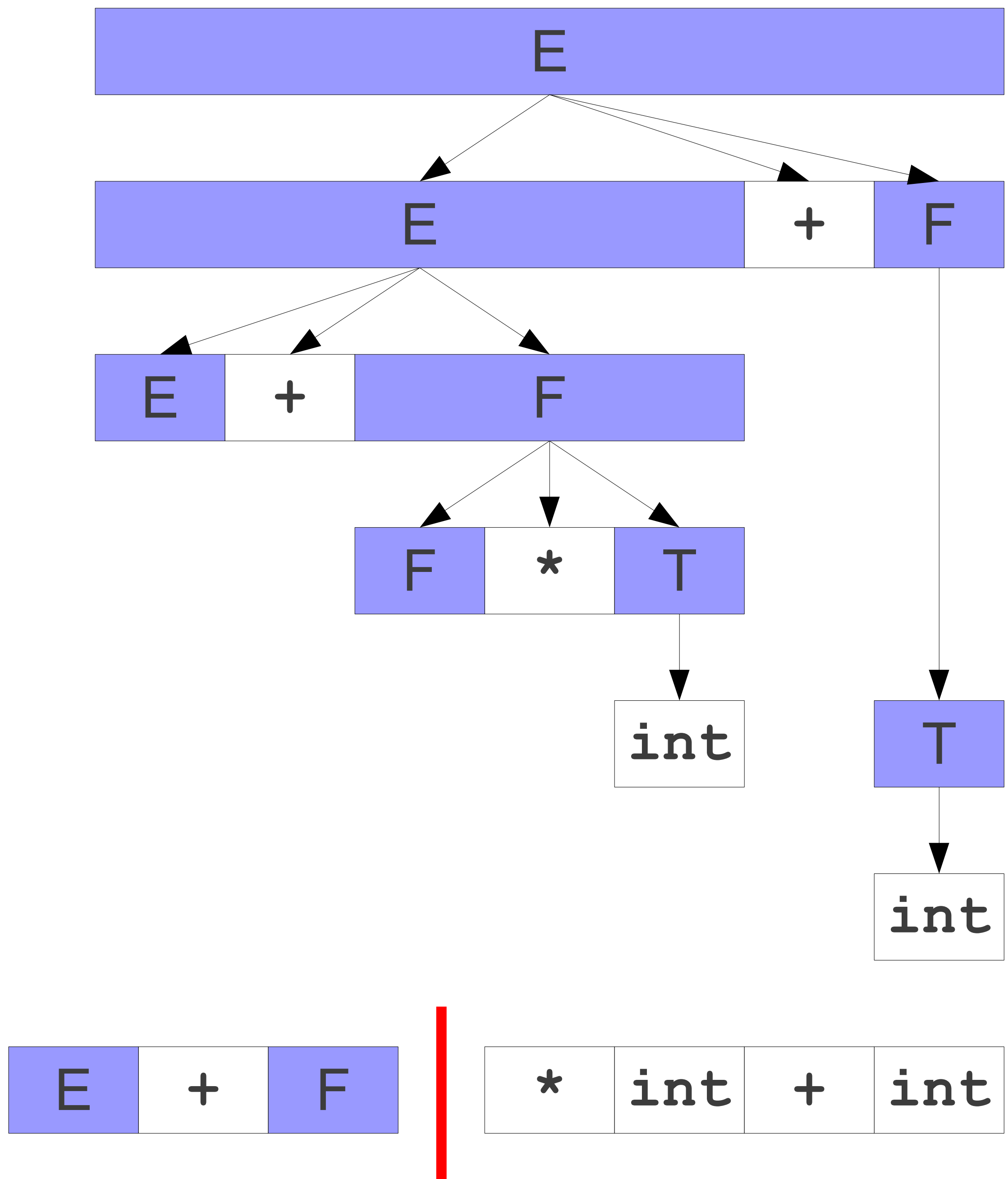
Another Look at Handles

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

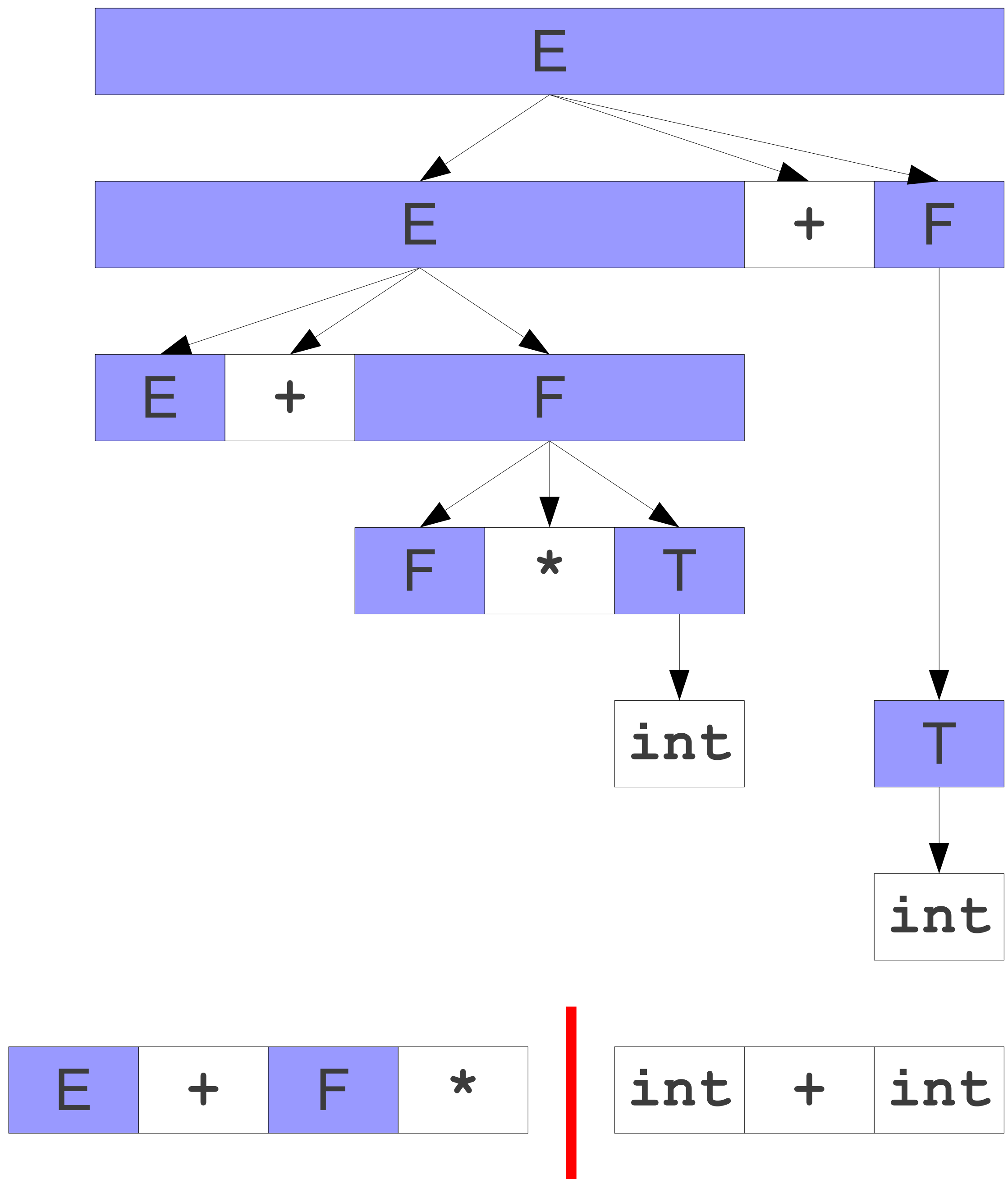


Another Look at Handles

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



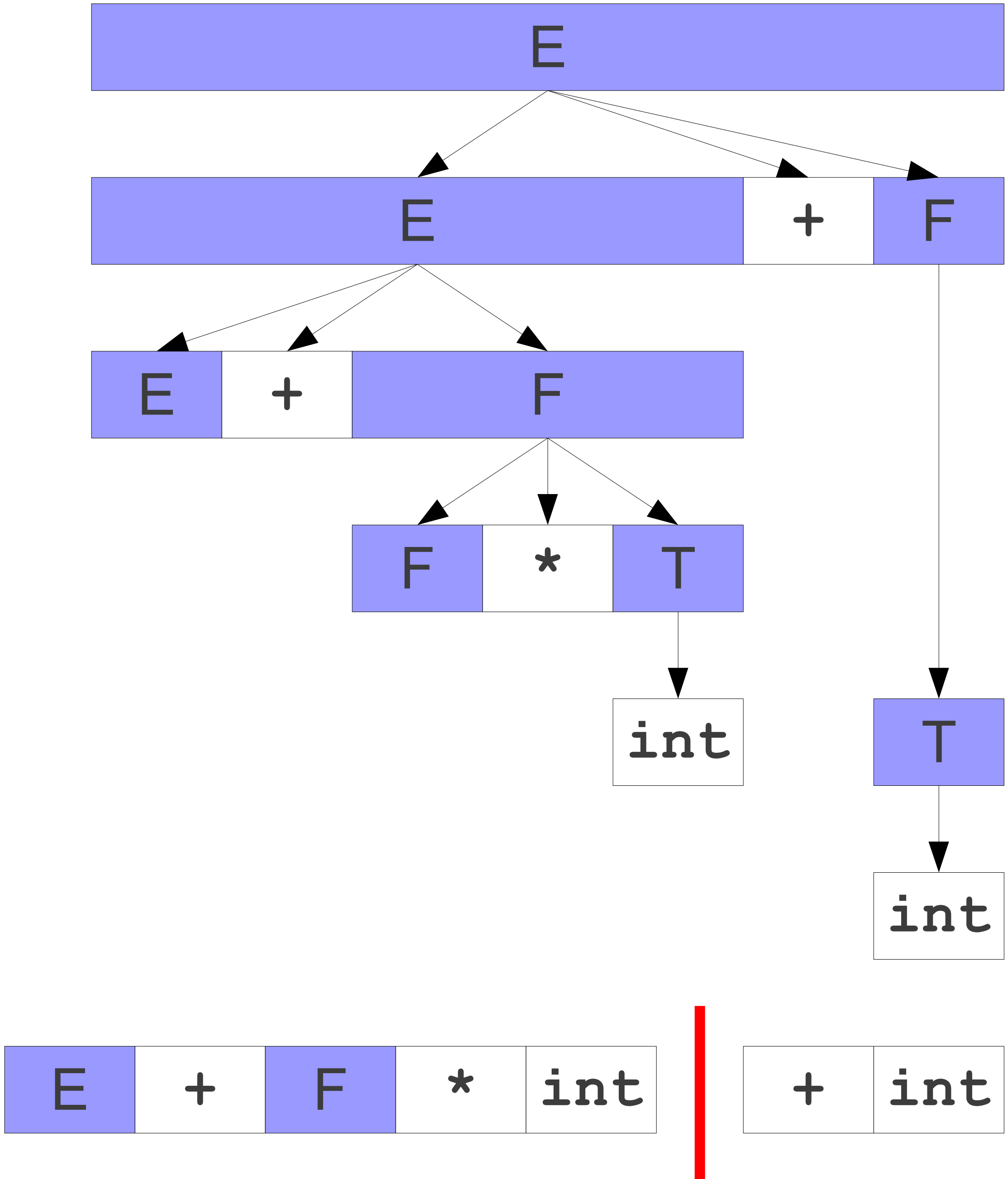
Another Look at Handles



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

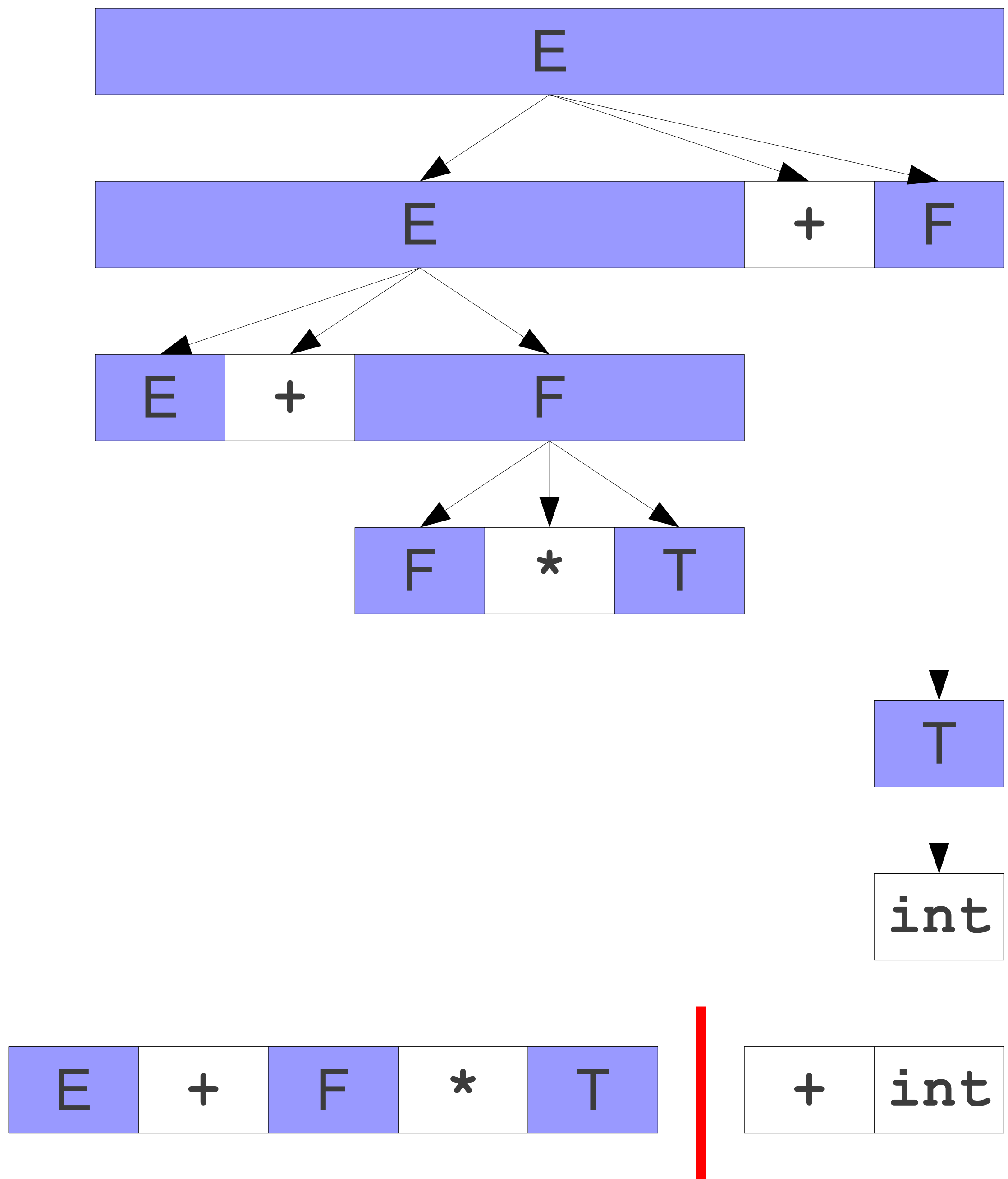
Another Look at Handles

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



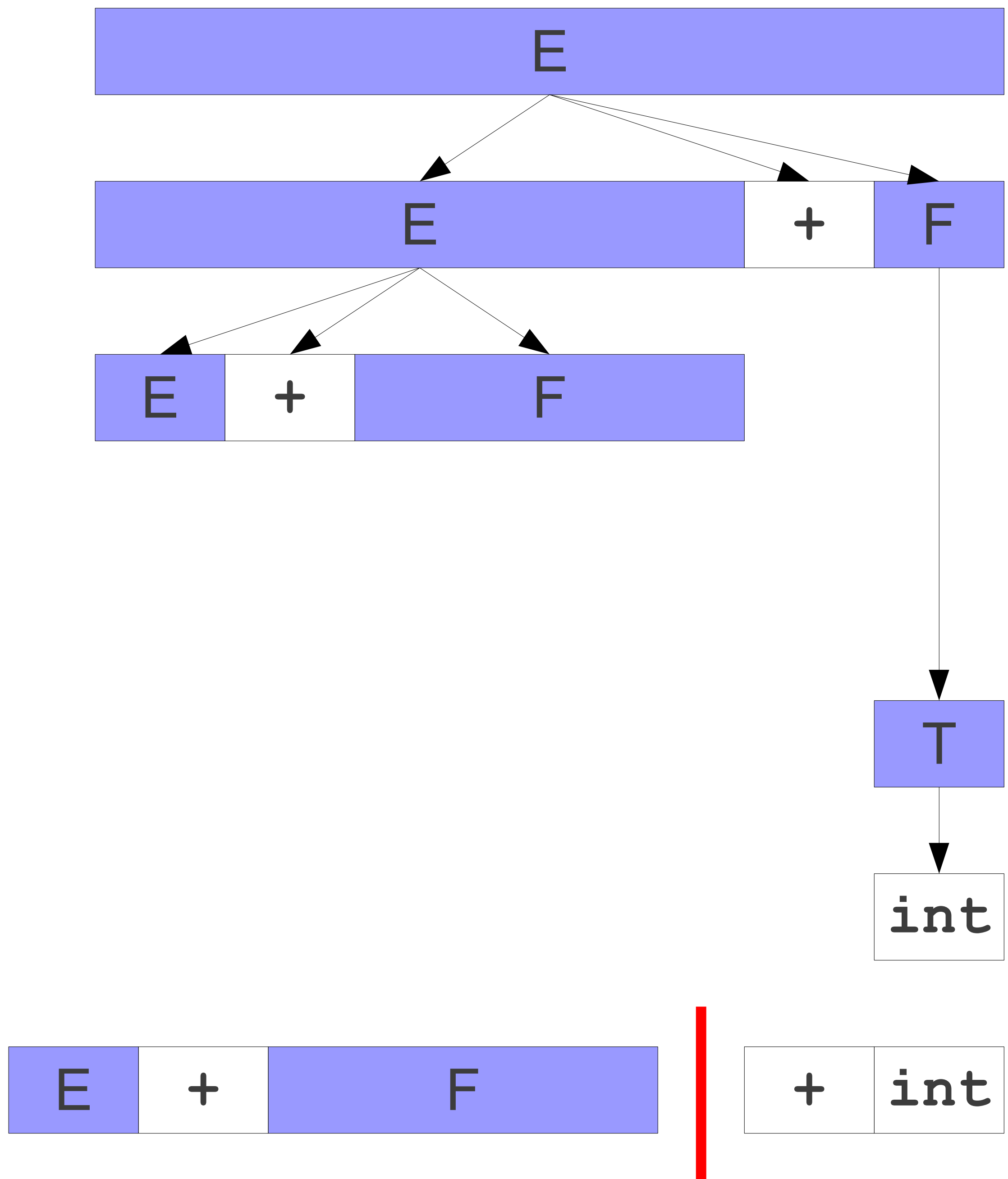
Another Look at Handles

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



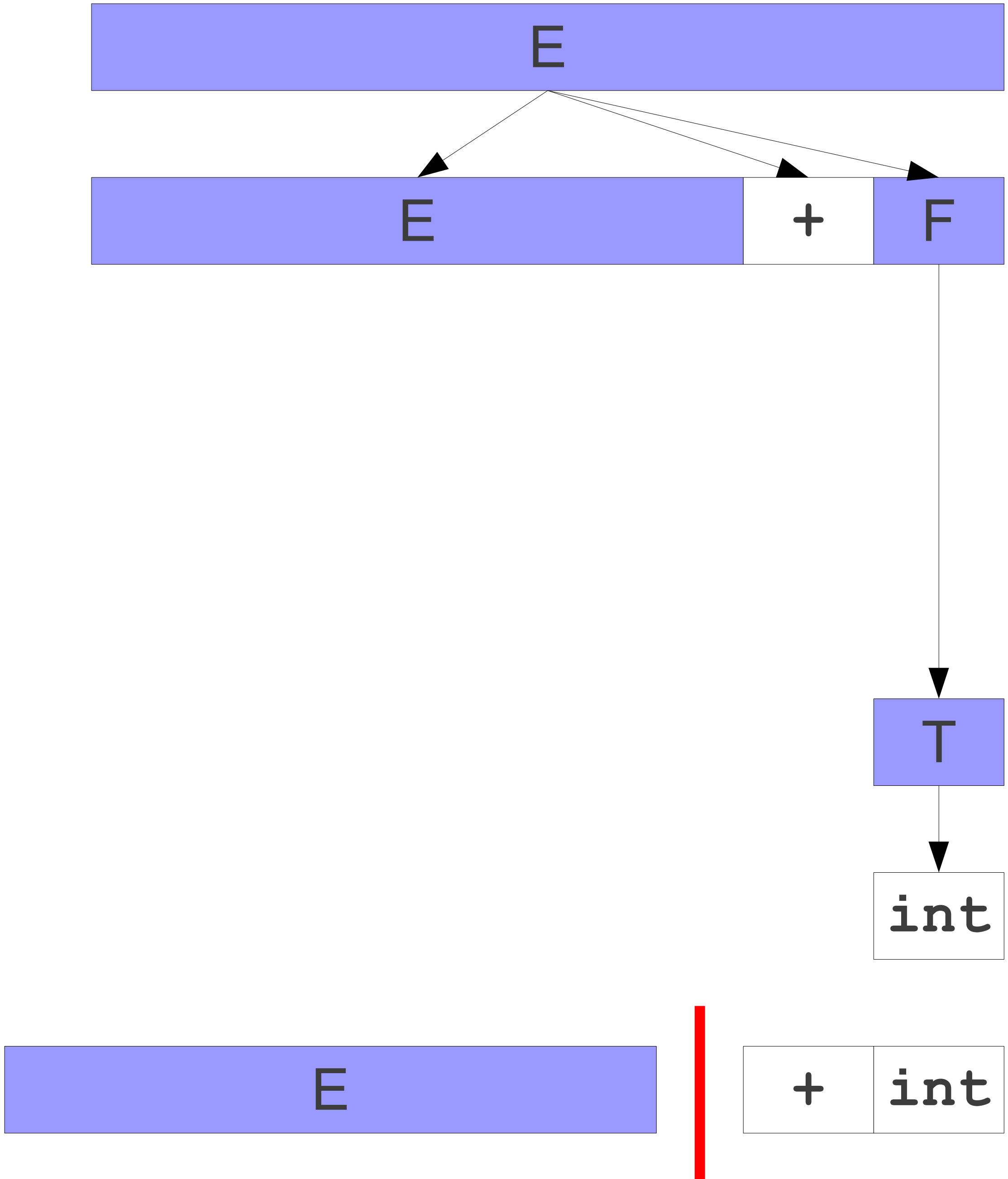
Another Look at Handles

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



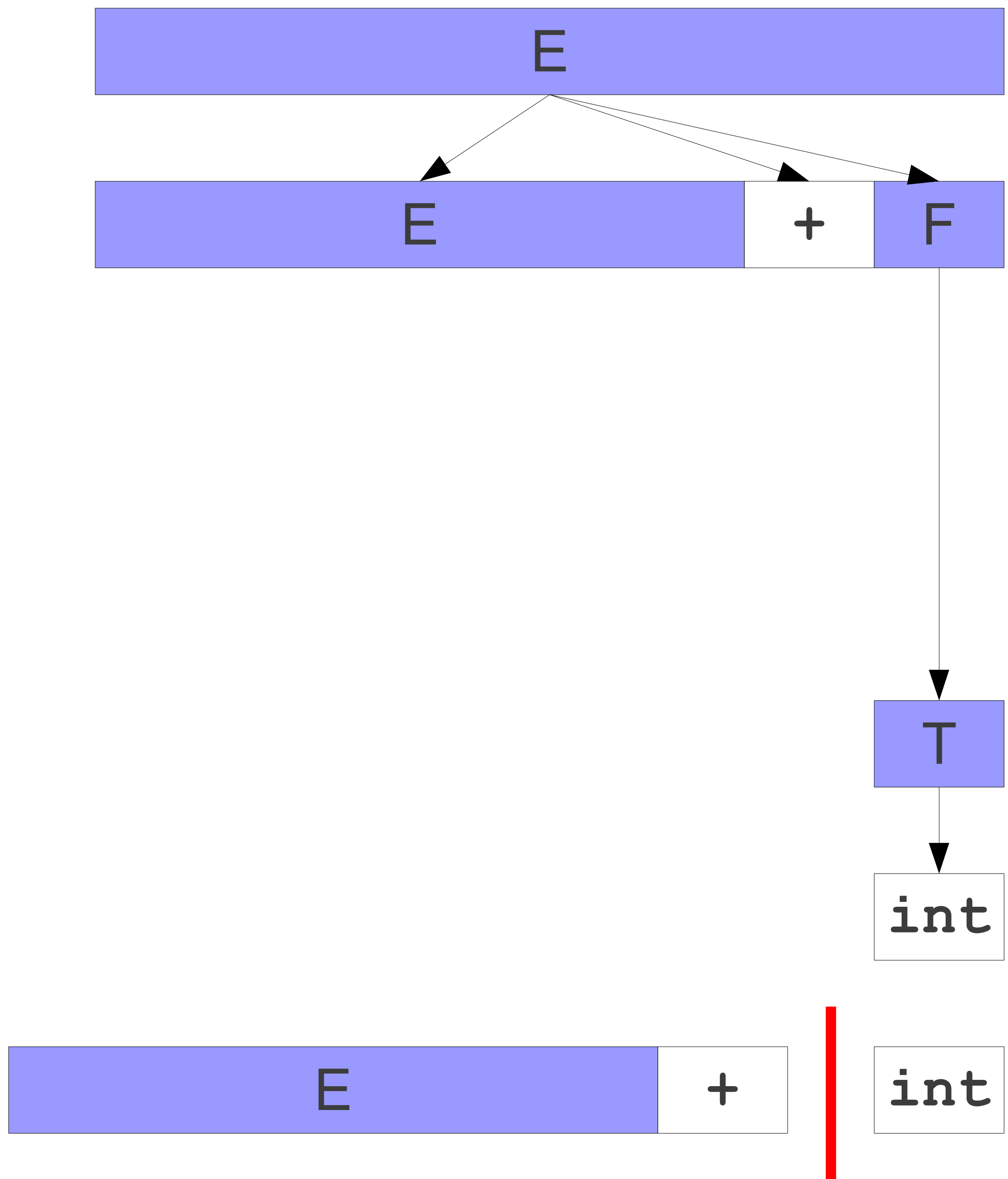
Another Look at Handles

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



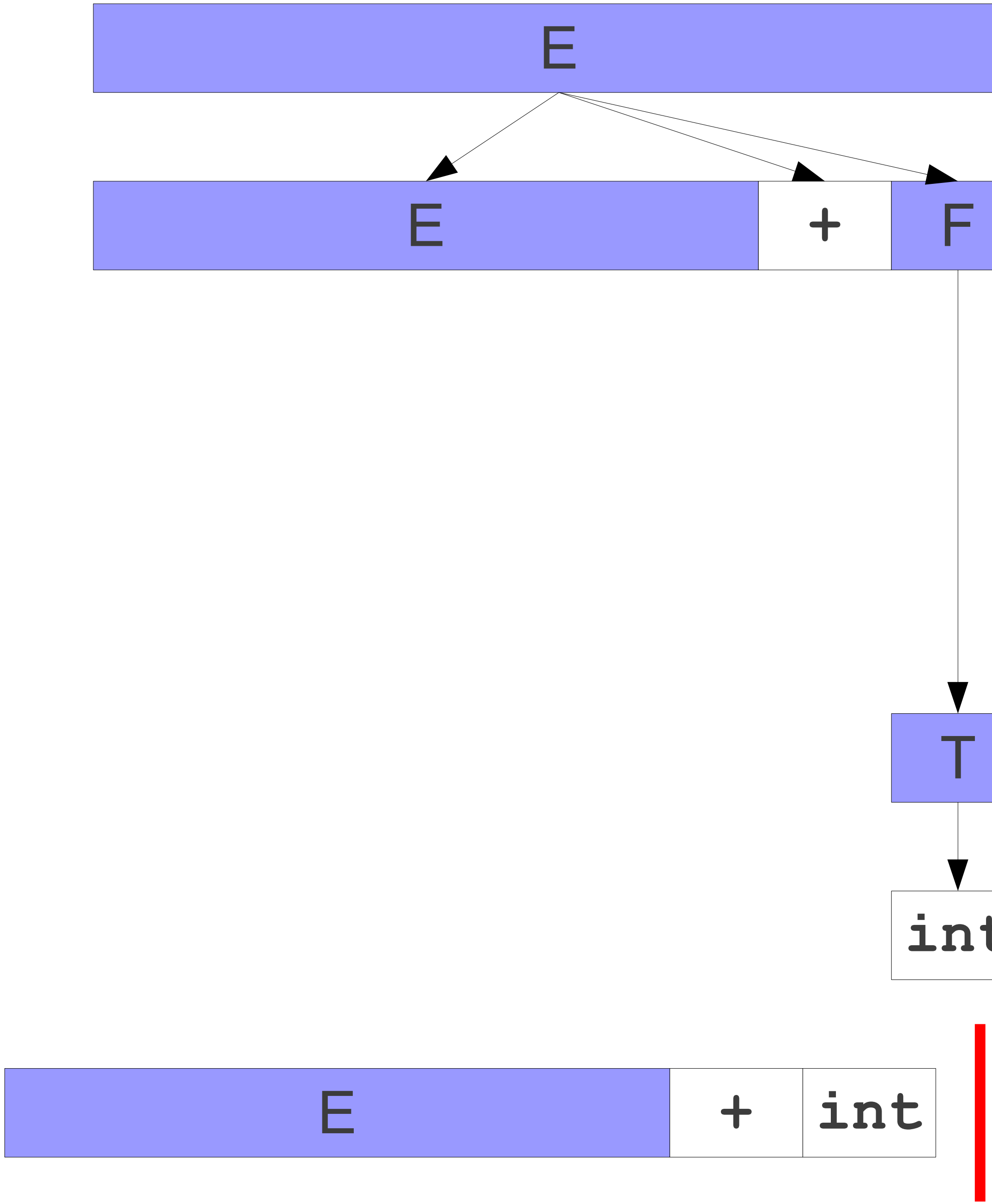
Another Look at Handles

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

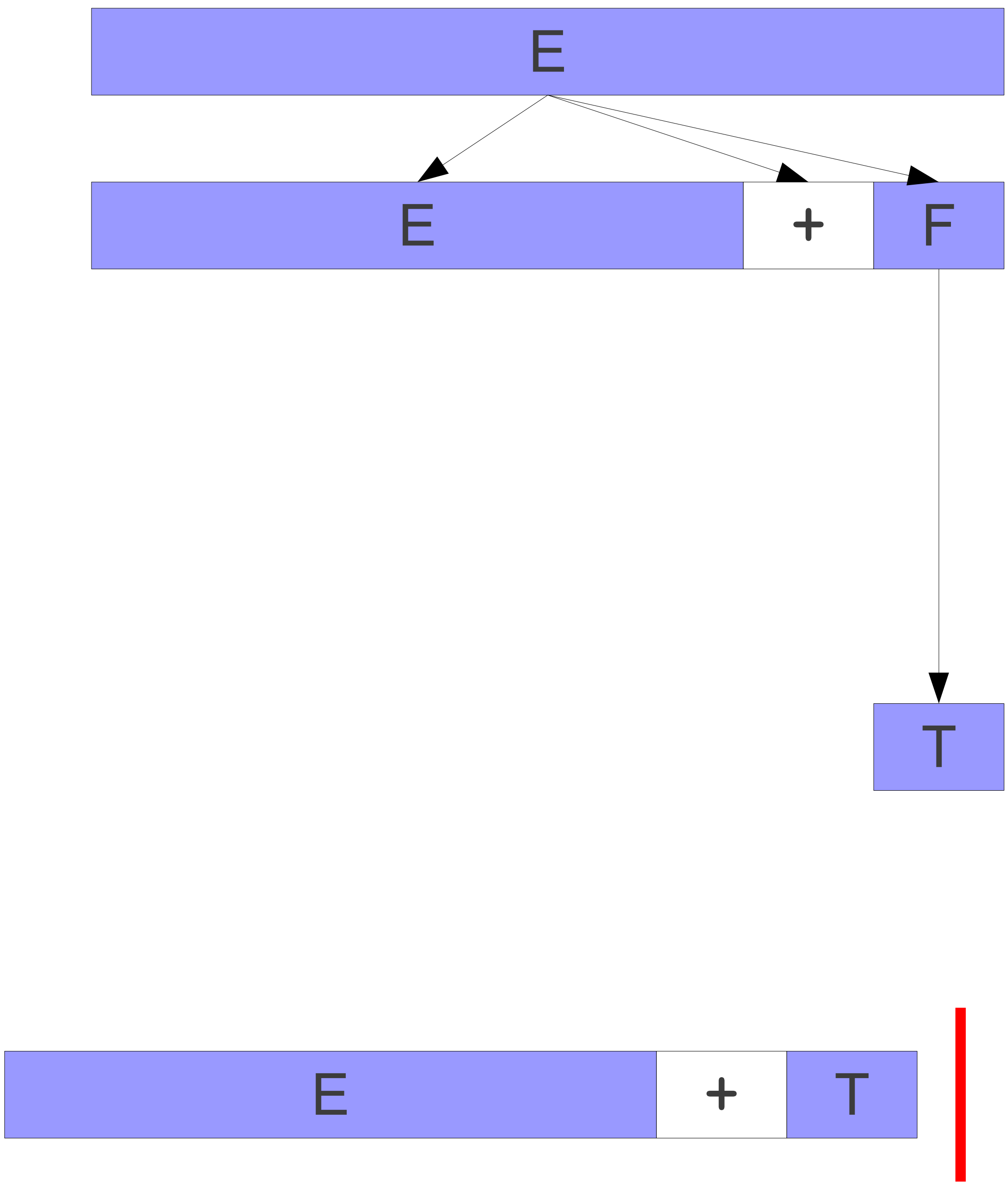


Another Look at Handles

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

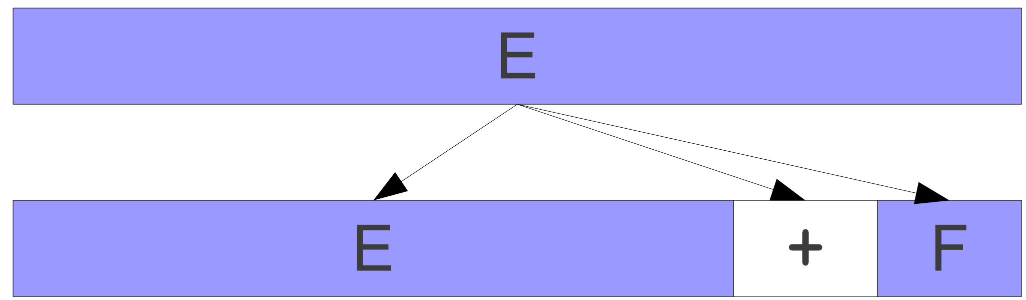


Another Look at Handles

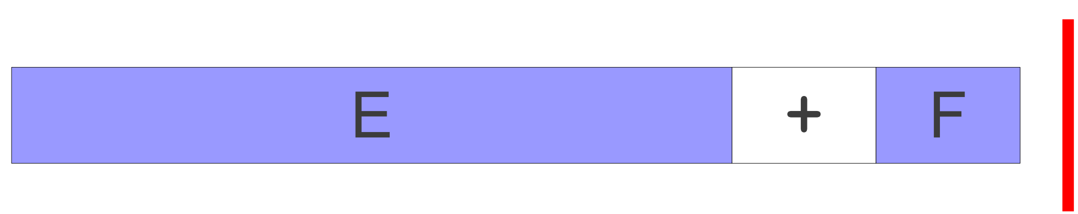


$$
\begin{aligned}
E &\to F \\
E &\to E \; \mathtt{+} \; F \\
F &\to F \; \mathtt{*} \; T \\
F &\to T \\
T &\to \mathtt{int} \\
T &\to \mathtt{(}E\mathtt{)}
\end{aligned}
$$

Another Look at Handles



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



Another Look at Handles

E

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

E

Tracking Our Position

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

int	+	int	*	int	+	int

Tracking Our Position

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

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

Tracking Our Position

$S \rightarrow \cdot E$

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

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**)**

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

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

Tracking Our Position

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

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

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`)`

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

`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
T → · `int`

`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
T → `int` ·

`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**)**

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

T	+	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**)**

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

T	|	+	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`)`

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

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

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

E	+	|	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**)**

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

E	+	|	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**)**

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

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

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

E	+	T	*	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)

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

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

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

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

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 **)**

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

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 → **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

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

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

E	+	F	*	T	|	+	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**)**

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

E	+	F	*	T	|	+	int

Tracking Our Position

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

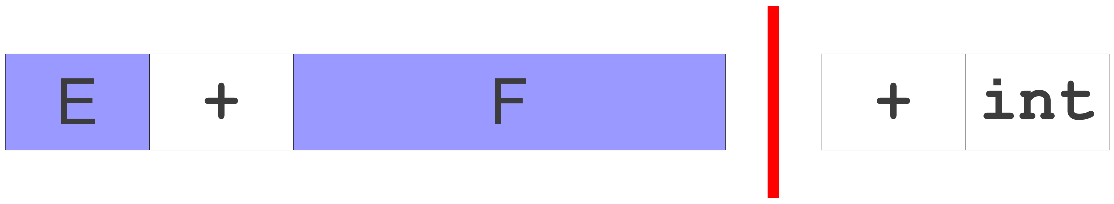
S → · E
E → · E + F
E → E + · F

E	+	F	|	+	int

Tracking Our Position

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

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



Tracking Our Position

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

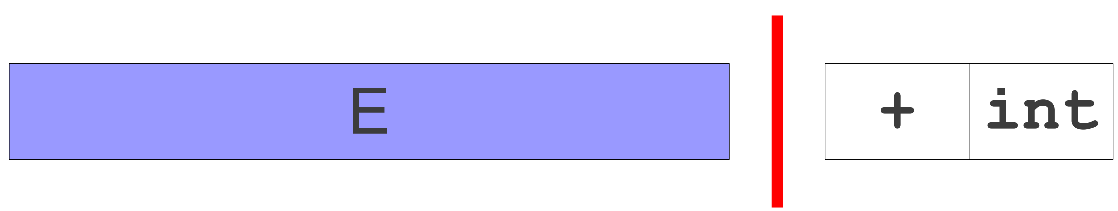
S → · E
E → · E + F

E	|	+	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



Tracking Our Position

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

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

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 **)**

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

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**)**

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

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
T → `int` ·

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

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

E	+	T

Tracking Our Position

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

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

E	+	F

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**)**

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

E	+	F

Tracking Our Position

$S \rightarrow \cdot E$

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

E

Tracking Our Position

S → E ·

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

$S \rightarrow \cdot E$

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

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	+	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**)**

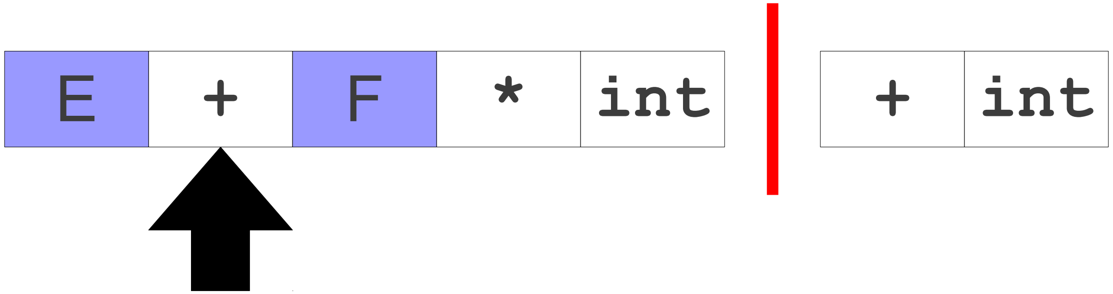
$S \to \cdot E$
$E \to \cdot E + F$
$E \to \cdot E + F$

E	+	F	*	int	|	+	int

Recognizing Left-Hand Sides

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

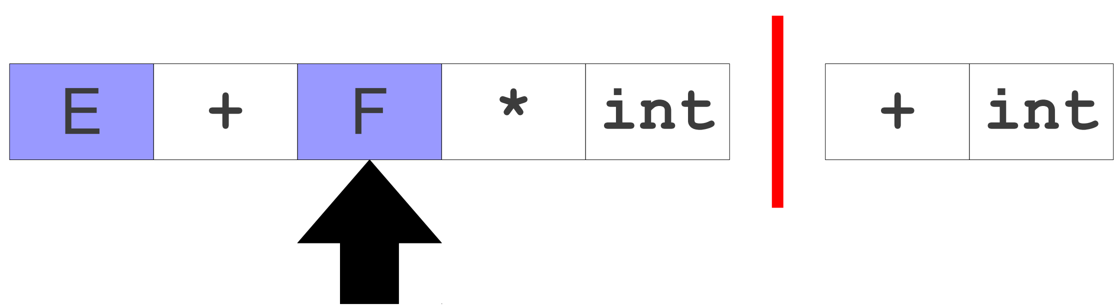
S → · E
E → · E + F
E → E · + F



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)

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



Recognizing Left-Hand Sides

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

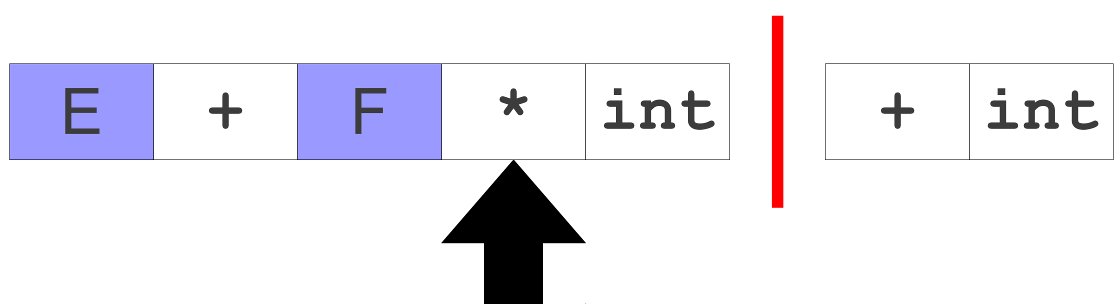
S → · E
E → · E + F
E → E + · F
F → · F * T

E	+	F	*	int	|	+	int

Recognizing Left-Hand Sides

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

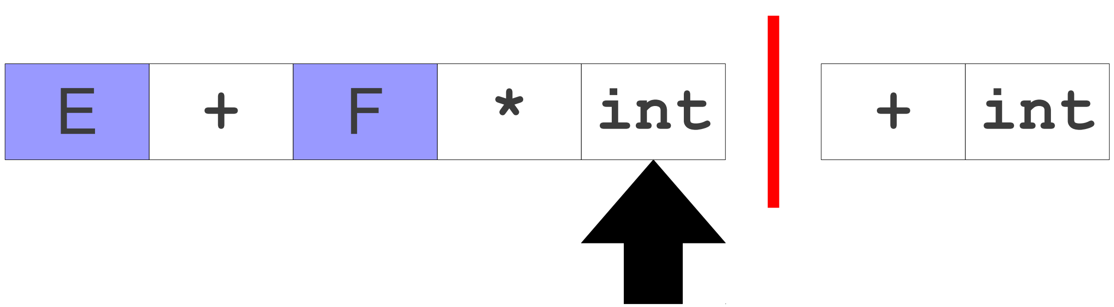
S → · E
E → · E + F
E → E + · F
F → F · `*` T



Recognizing Left-Hand Sides

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

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



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**)

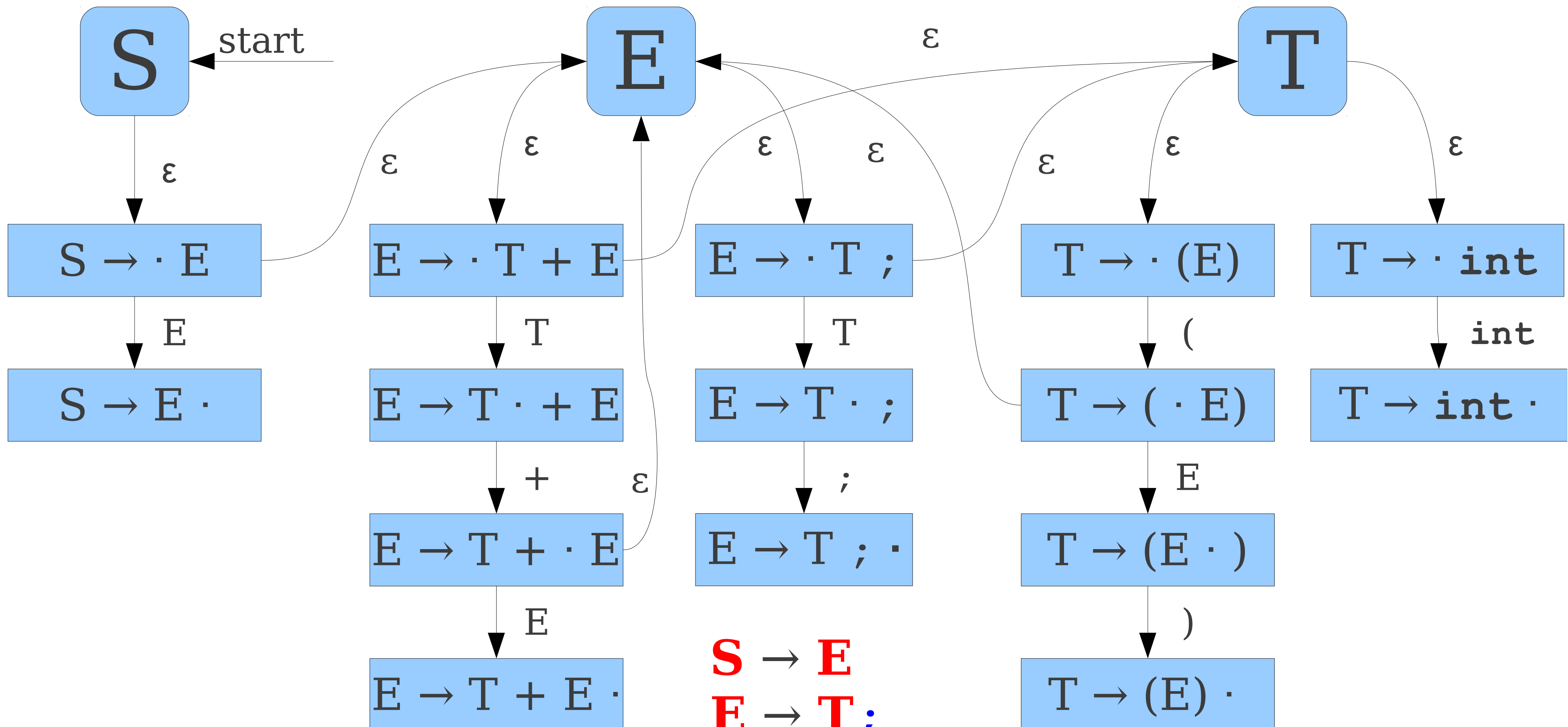
S → · E
E → · E + F
E → E + · F
F → F * · T
T → int ·

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

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

- 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

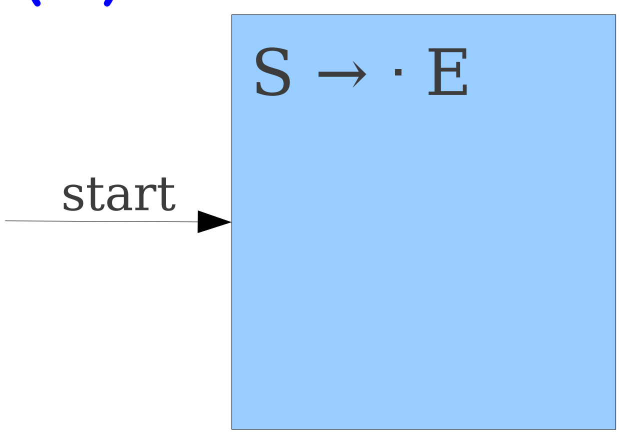
S → **E**

E → **T** `;`

E → **T** `+` **E**

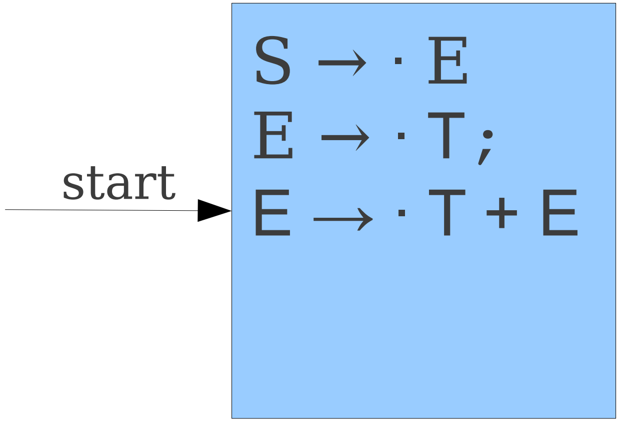
T → `int`

T → `(`**E**`)`



A Deterministic Automaton

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



A Deterministic Automaton

$\mathbf{S} \to \mathbf{E}$

$\mathbf{E} \to \mathbf{T}\texttt{;}$

$\mathbf{E} \to \mathbf{T}\ \texttt{+}\ \mathbf{E}$

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

$\mathbf{T} \to \texttt{(}\mathbf{E}\texttt{)}$

start

$S \to \cdot\, E$

$E \to \cdot\, T\texttt{;}$

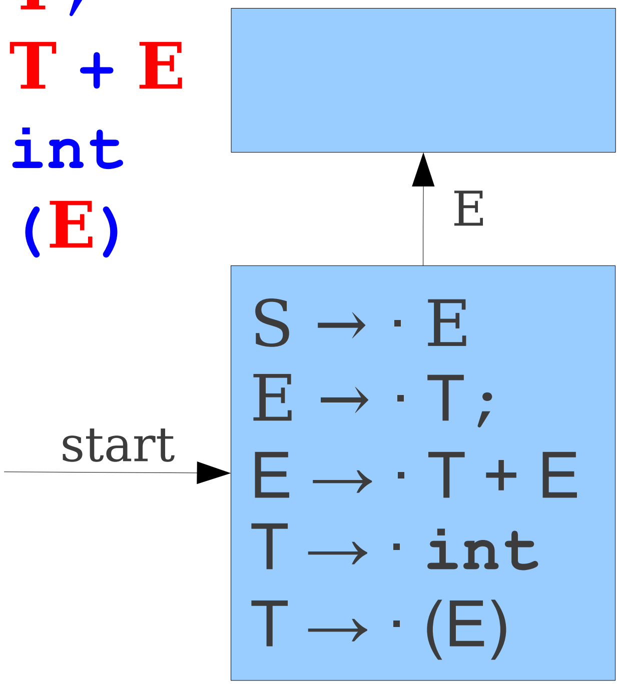
$E \to \cdot\, T + E$

$T \to \cdot\, \texttt{int}$

$T \to \cdot\, (E)$

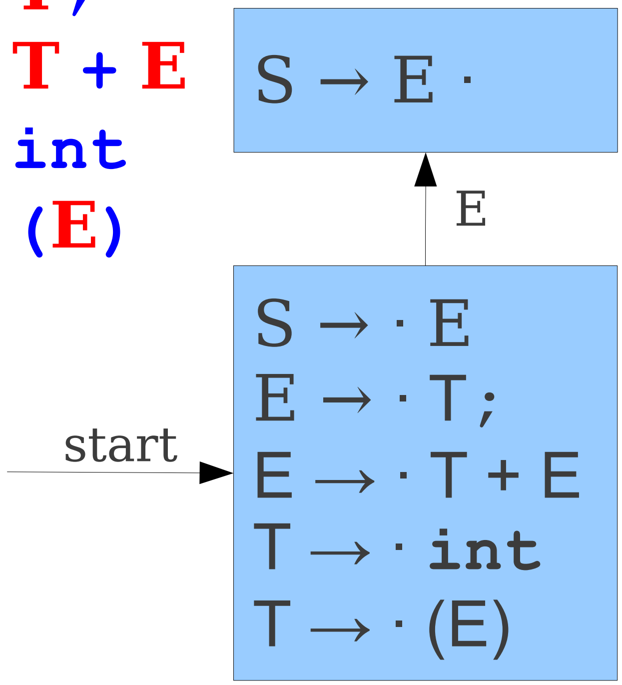
A Deterministic Automaton

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



A Deterministic Automaton

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



A Deterministic Automaton

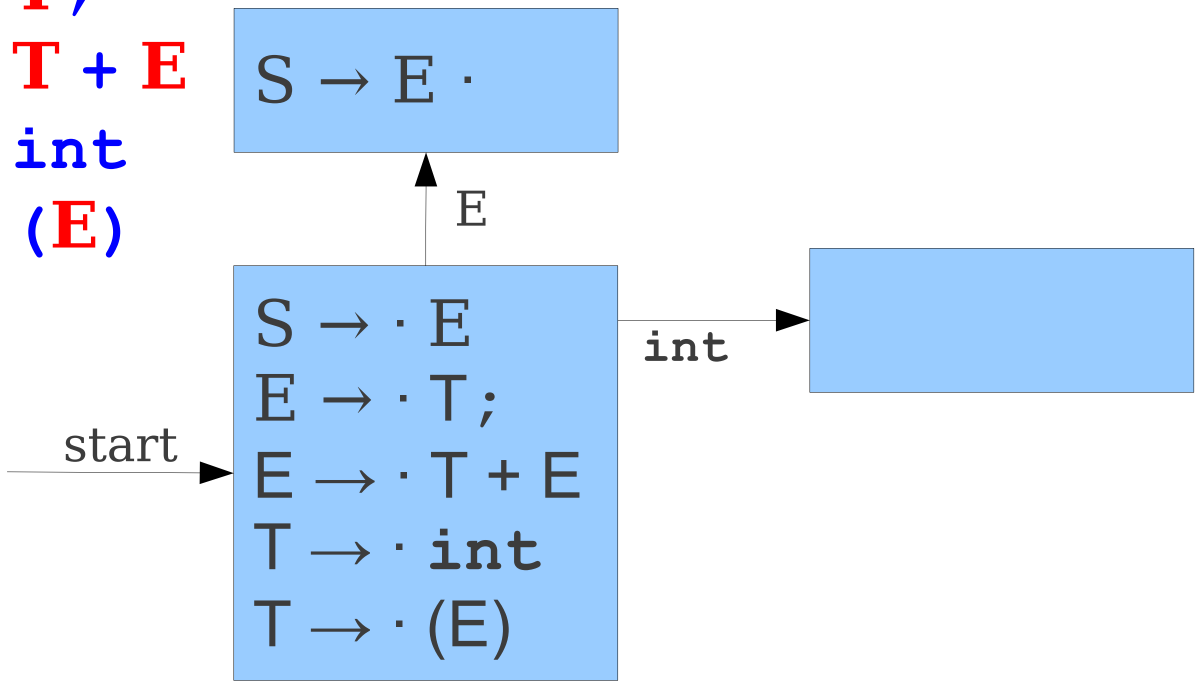
$\mathbf{S} \rightarrow \mathbf{E}$

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

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

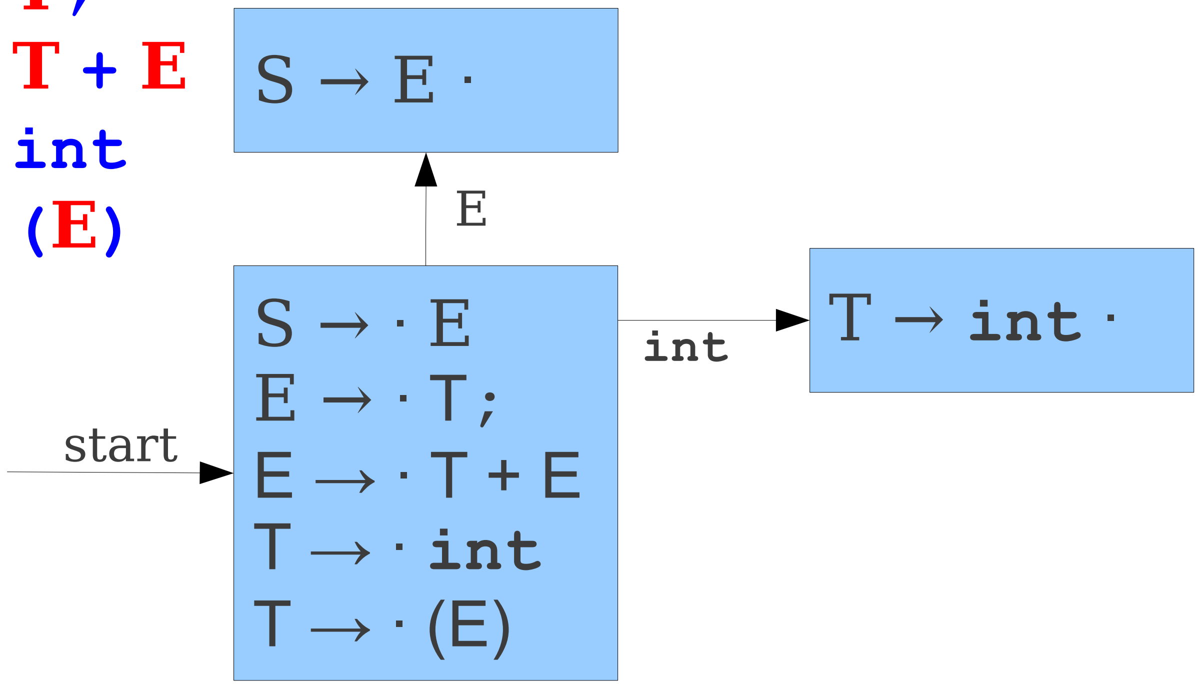
$\mathbf{T} \rightarrow$ **`int`**

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



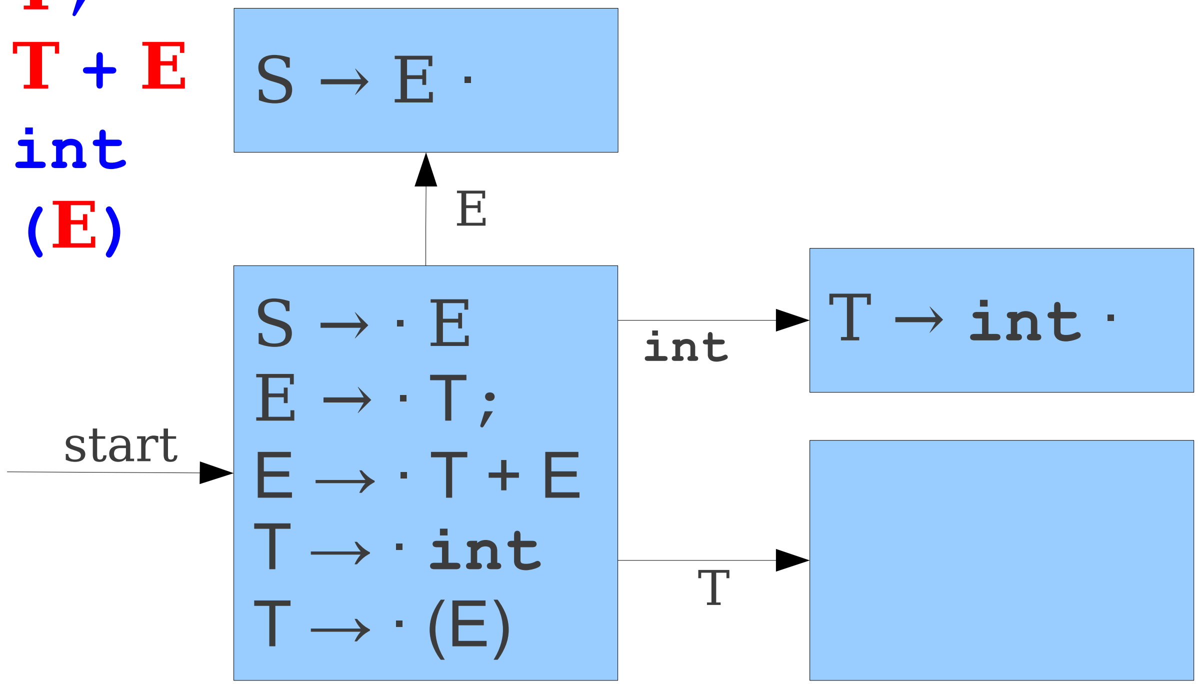
A Deterministic Automaton

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



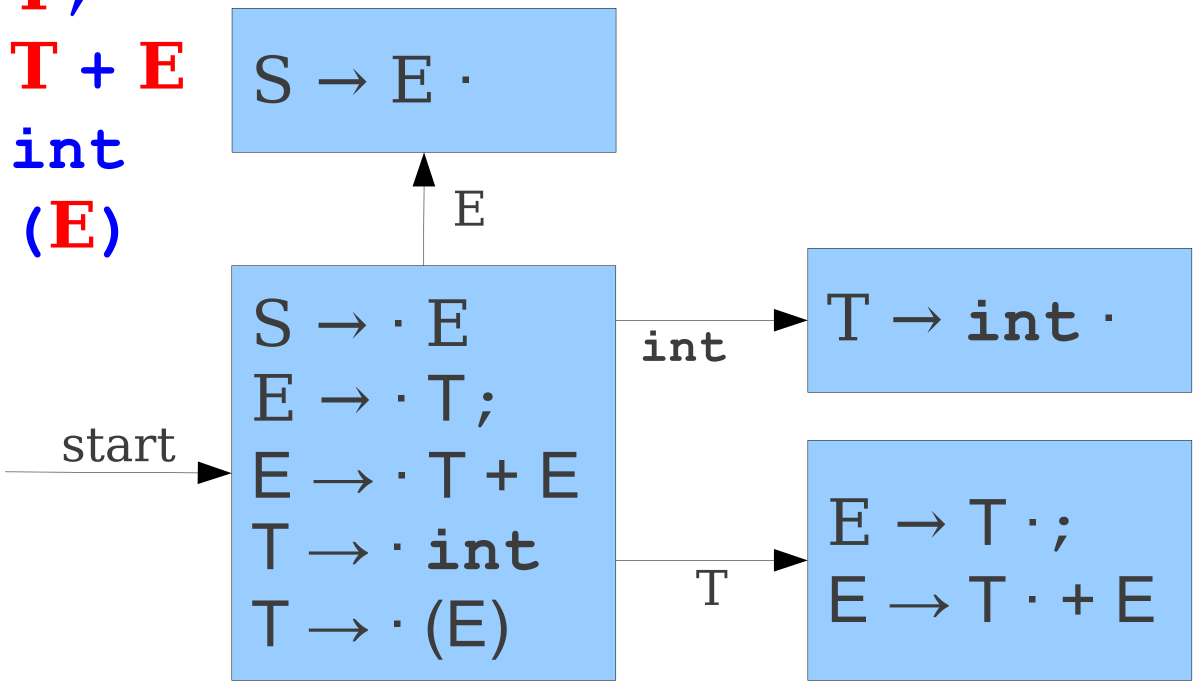
A Deterministic Automaton

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



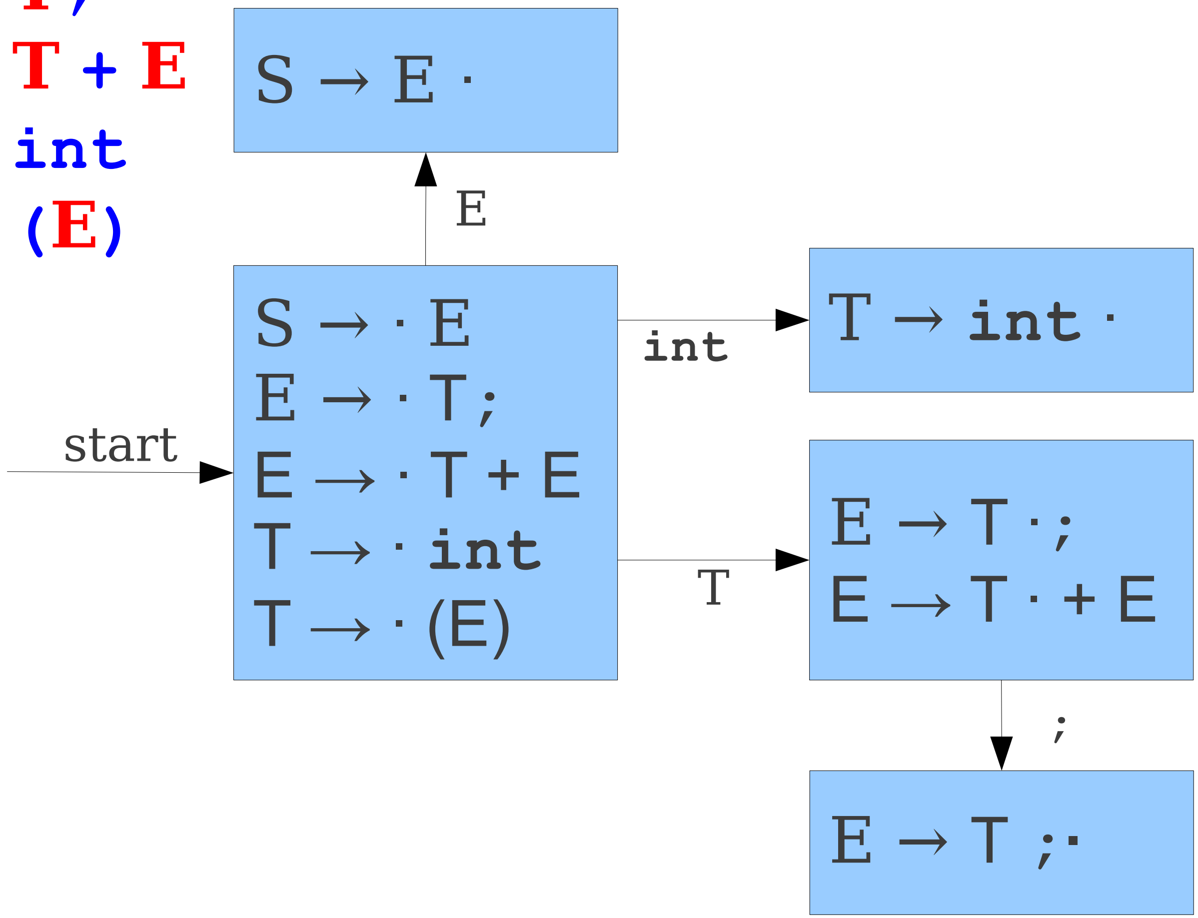
A Deterministic Automaton

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



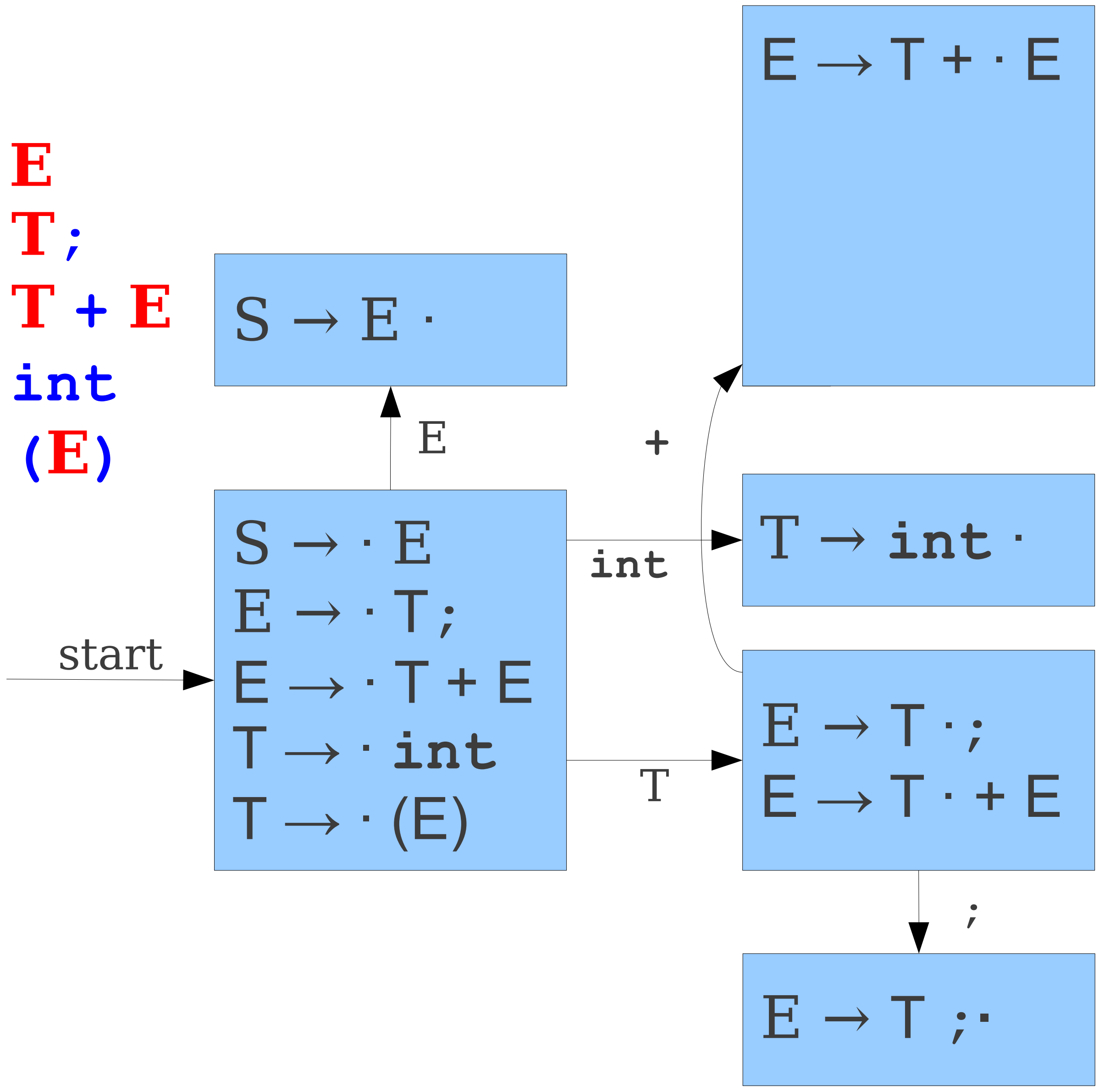
A Deterministic Automaton

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



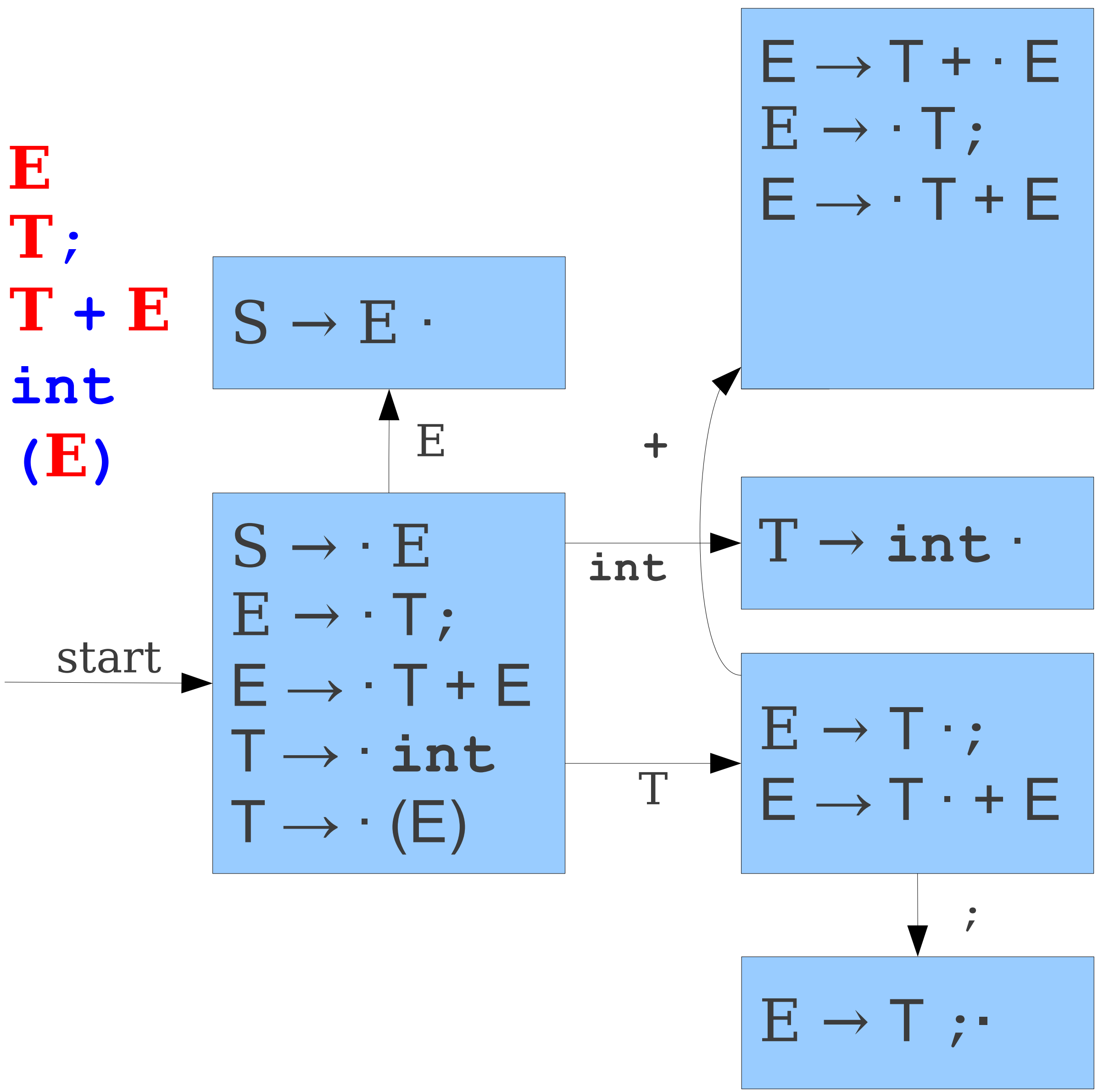
A Deterministic Automaton

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



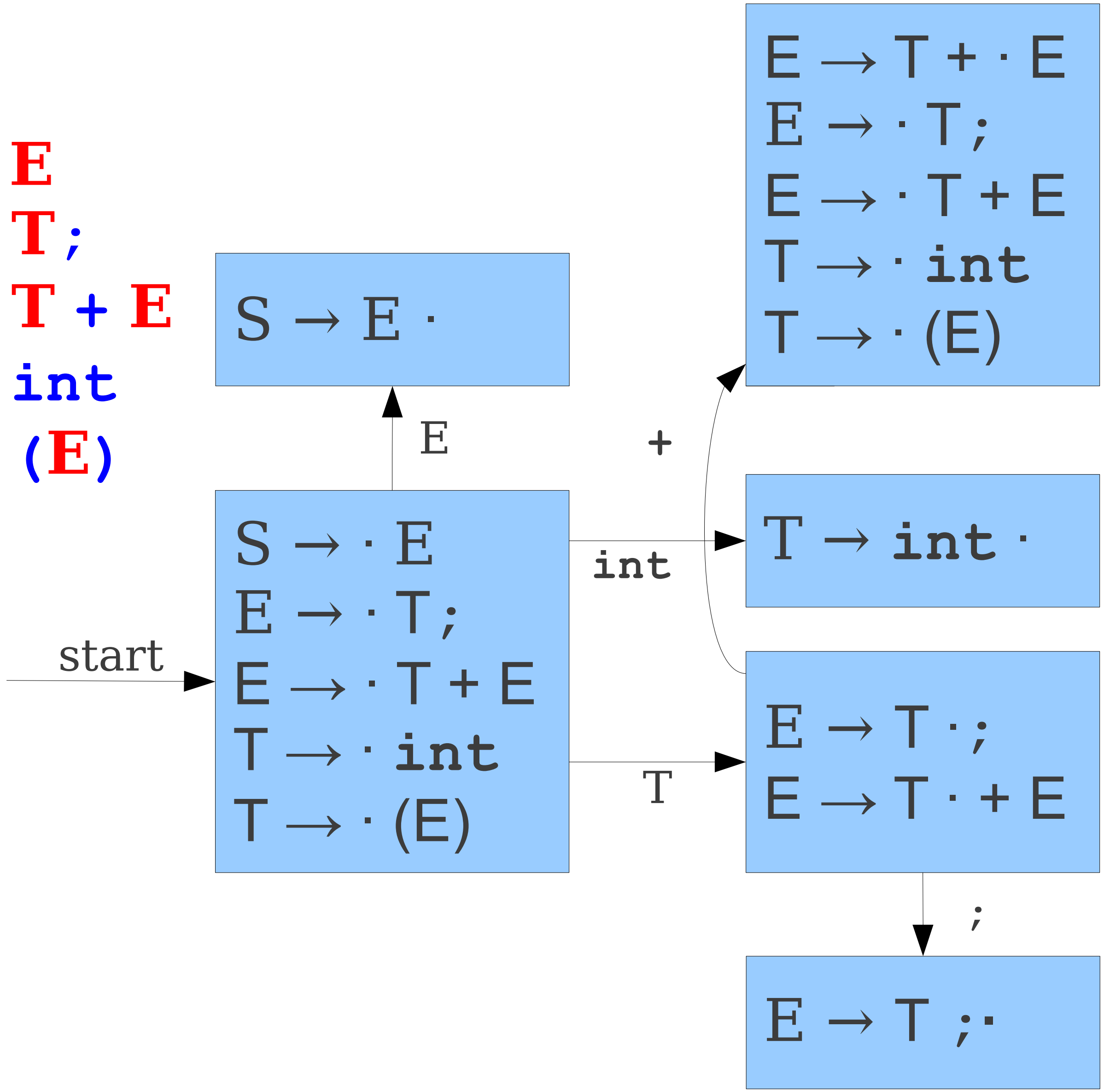
A Deterministic Automaton

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



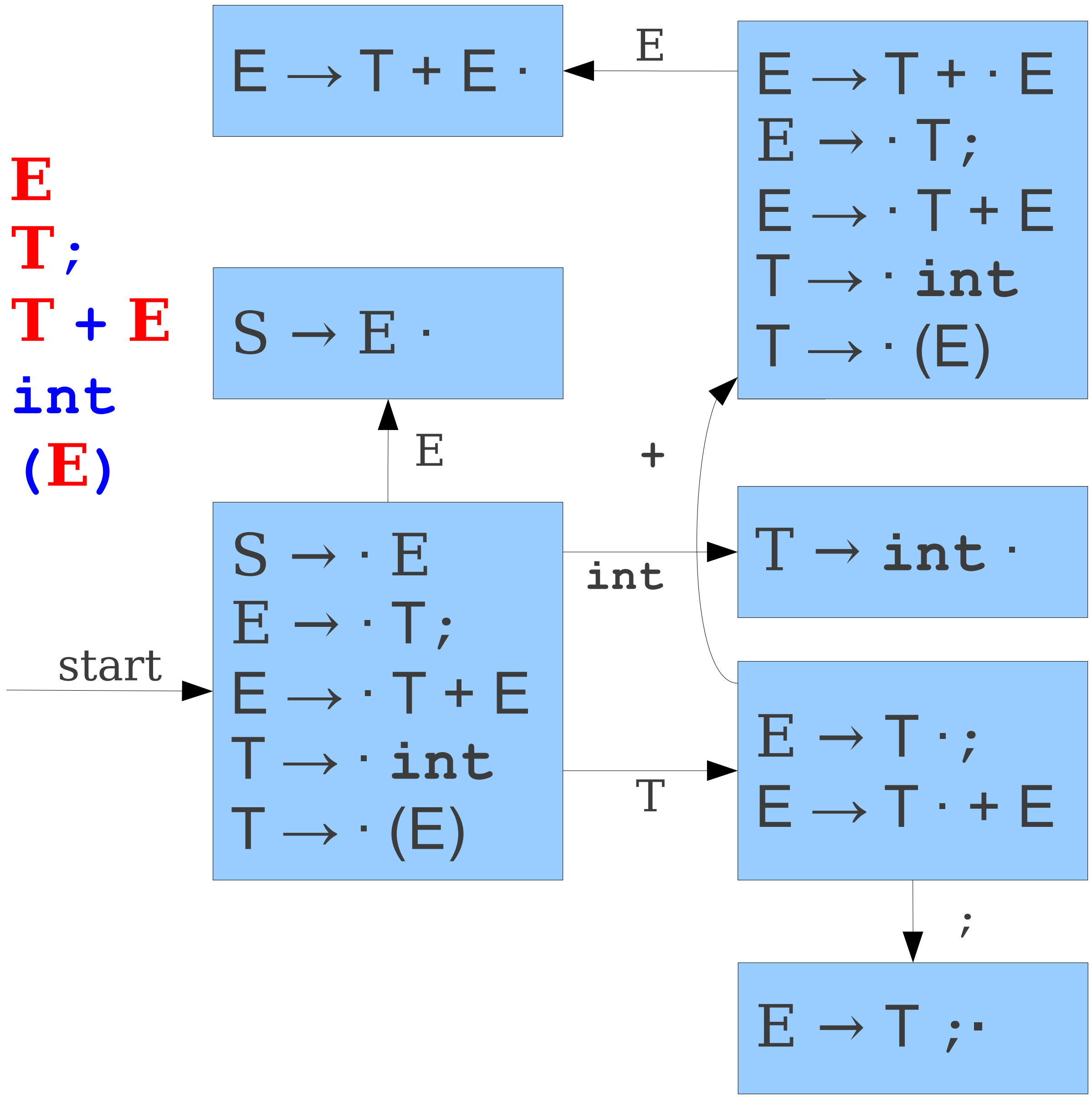
A Deterministic Automaton

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



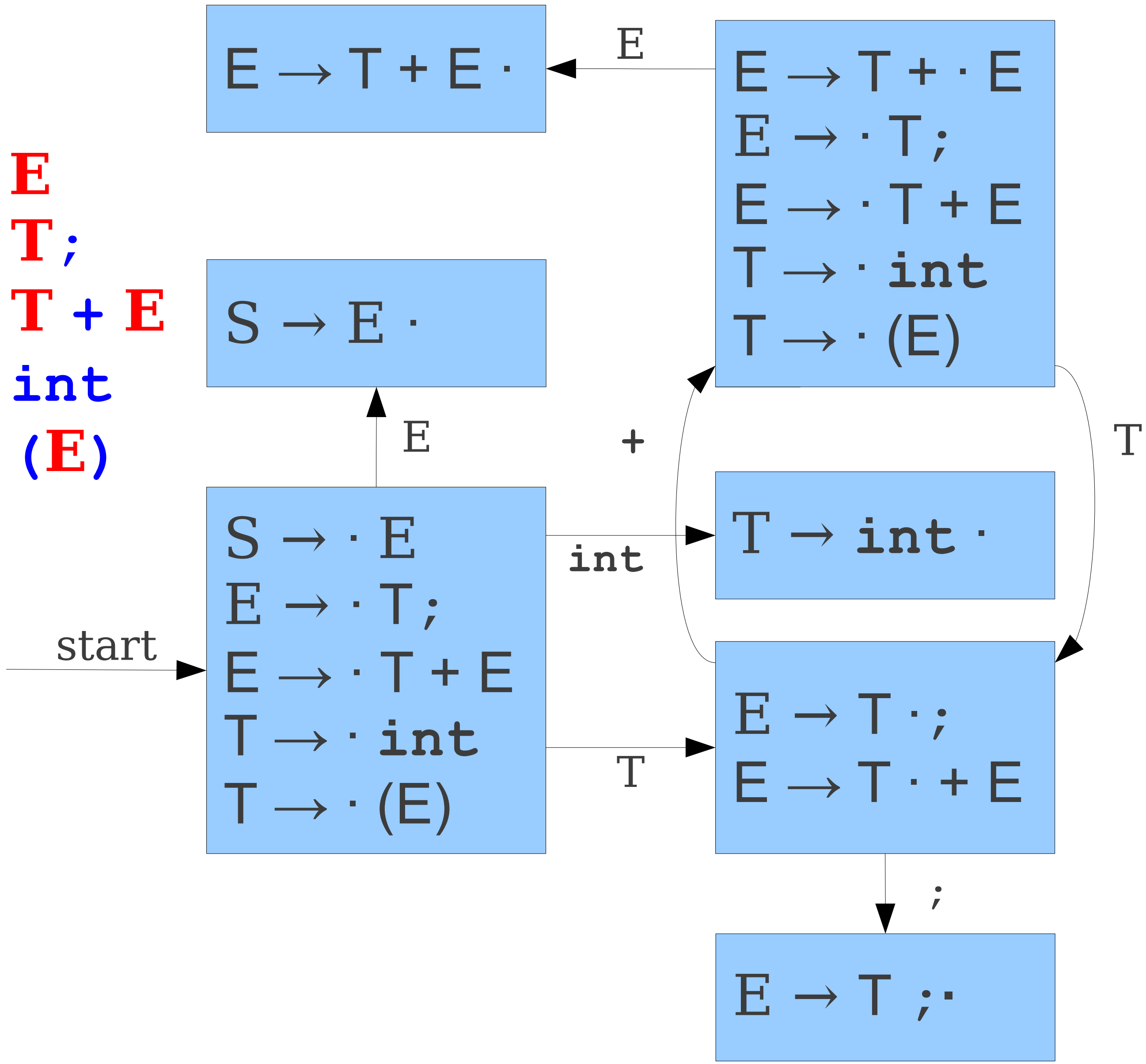
A Deterministic Automaton

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



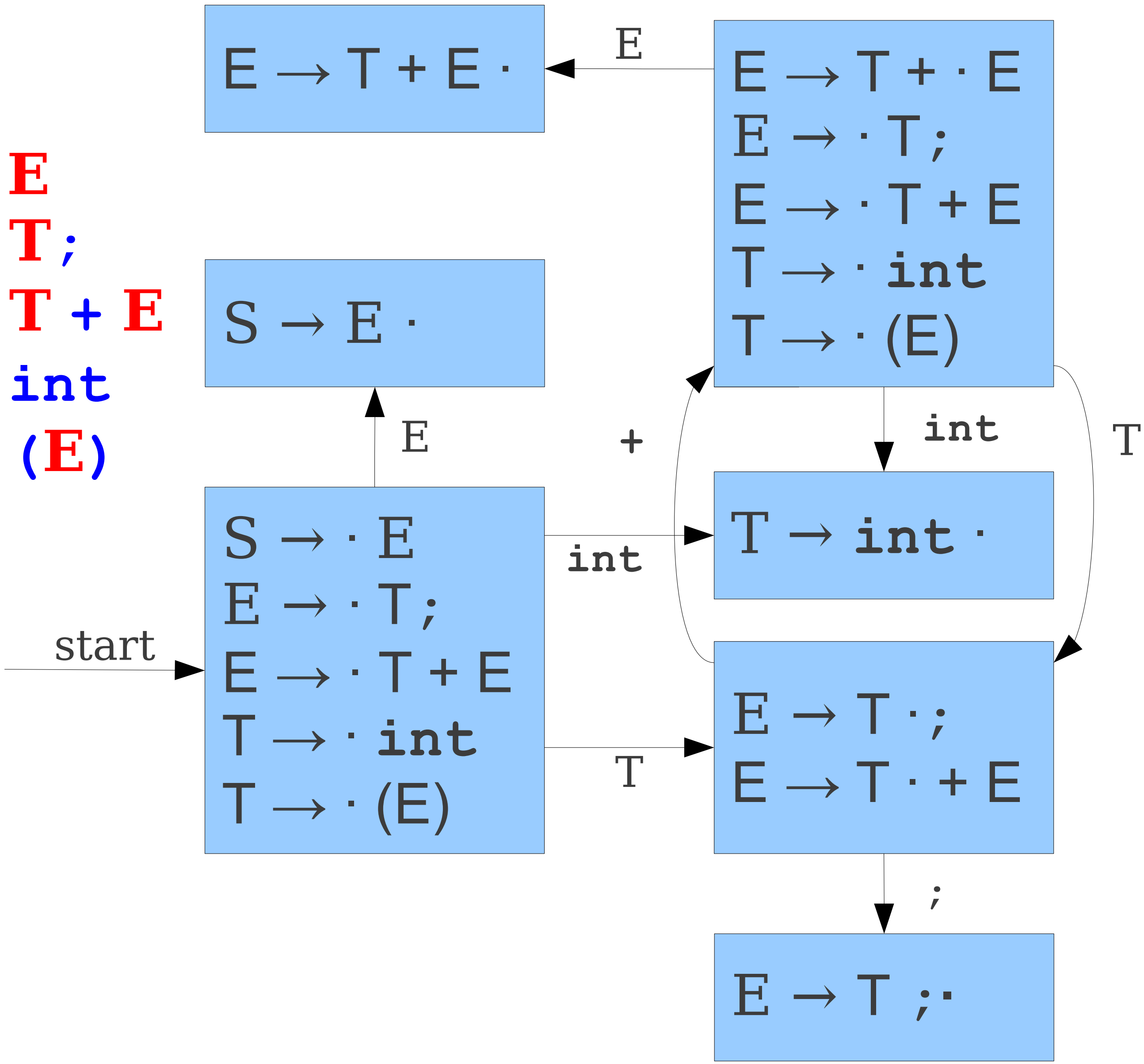
A Deterministic Automaton

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

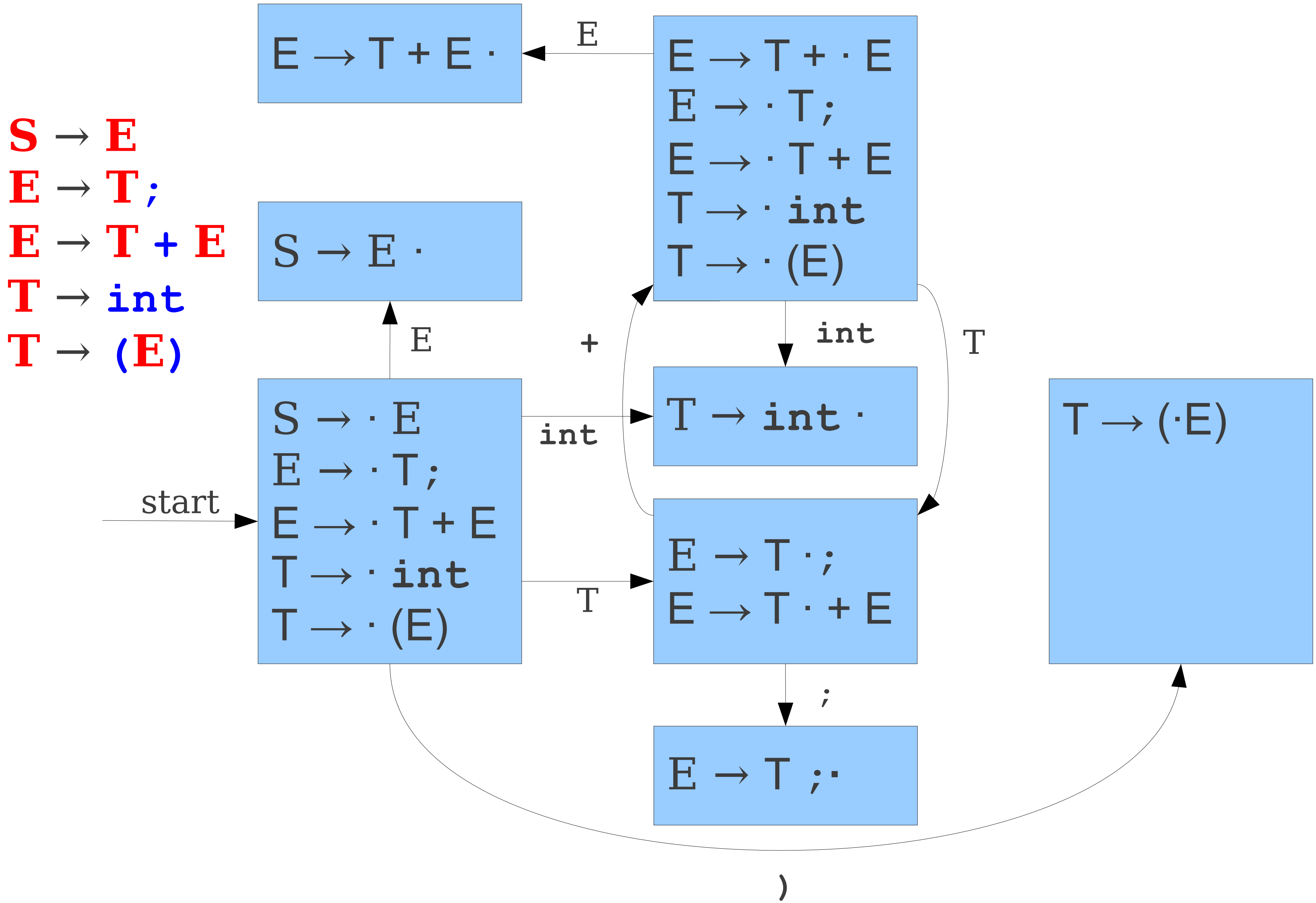


A Deterministic Automaton

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

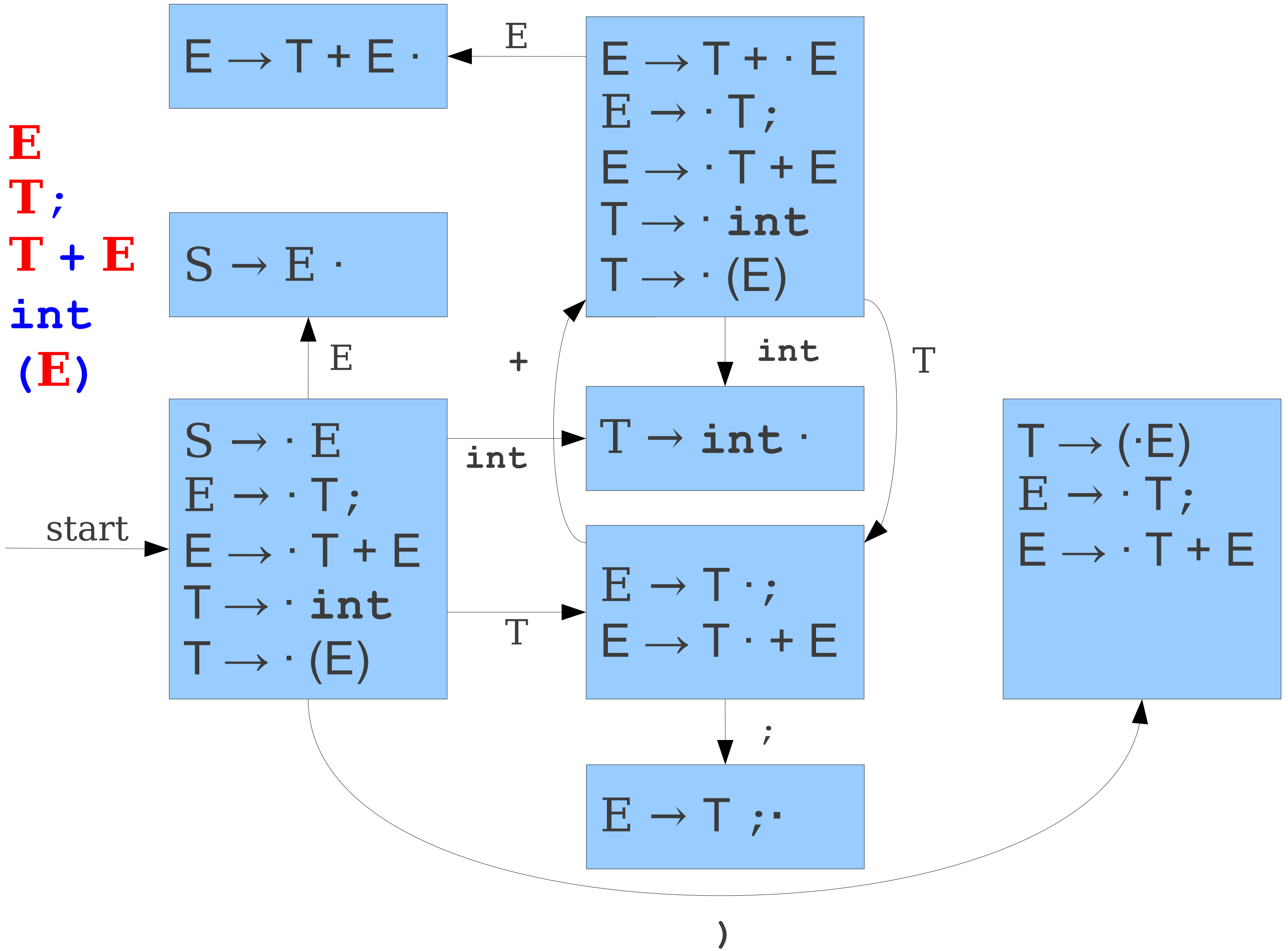


A Deterministic Automaton



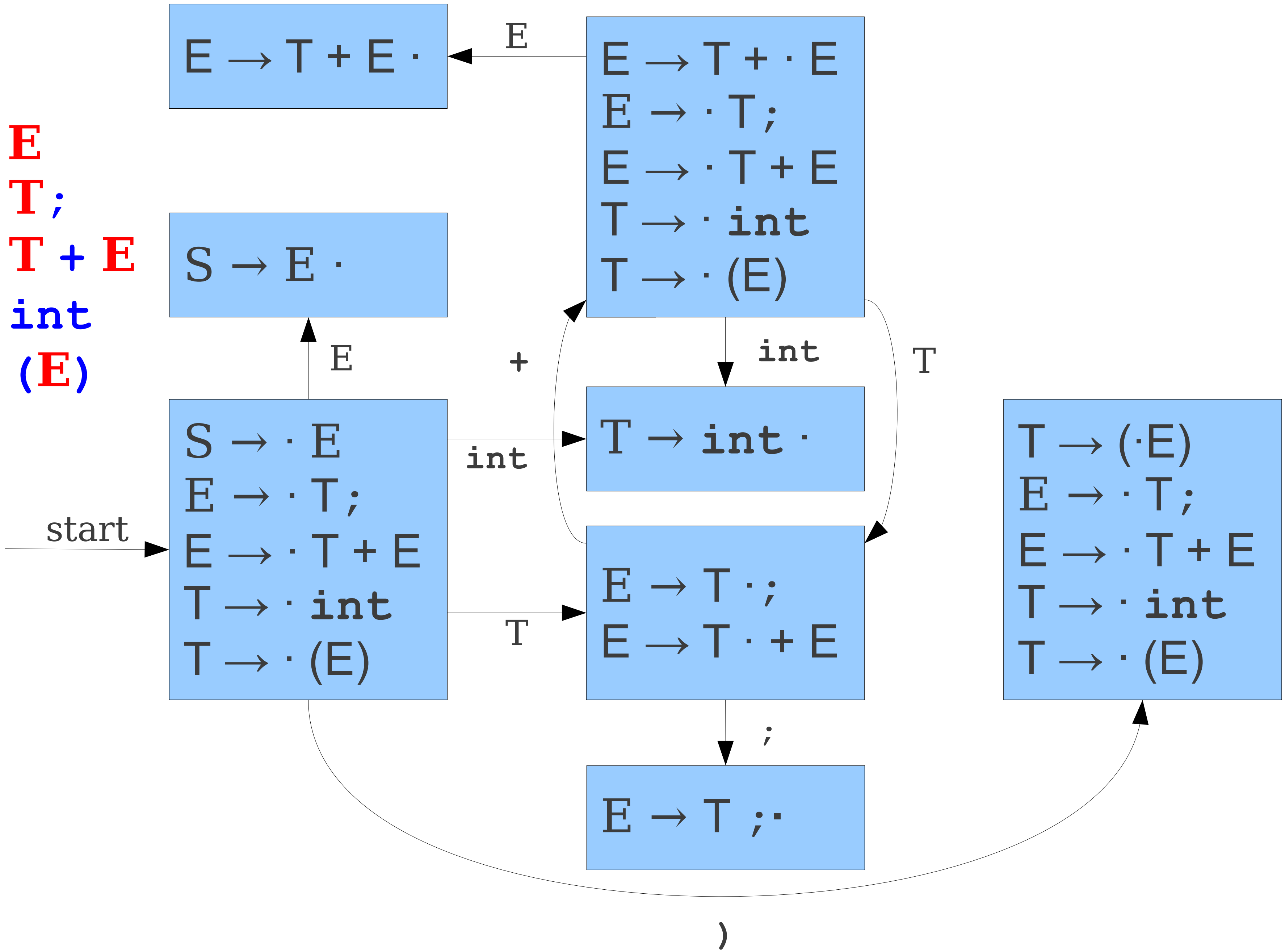
A Deterministic Automaton

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



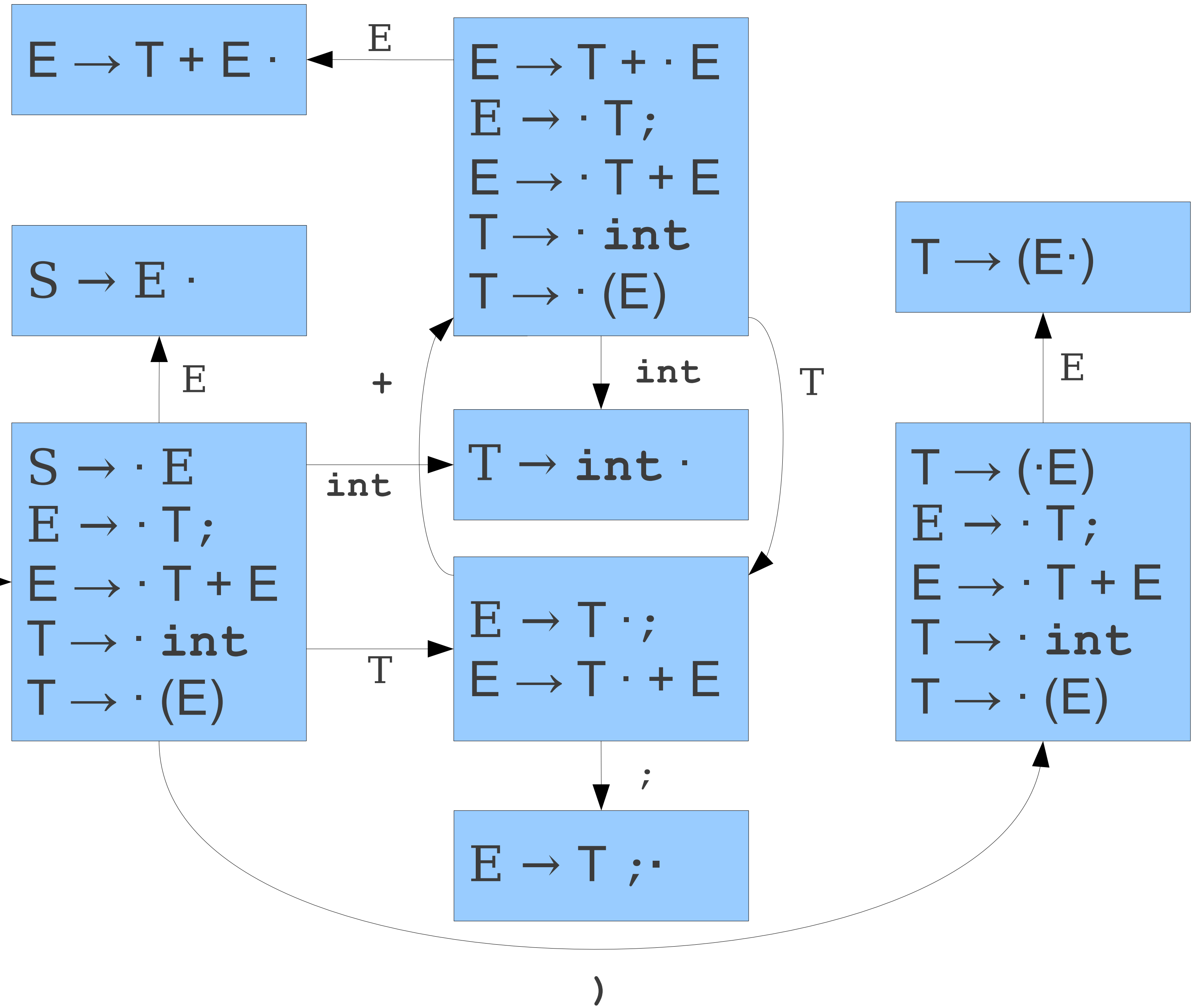
A Deterministic Automaton

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



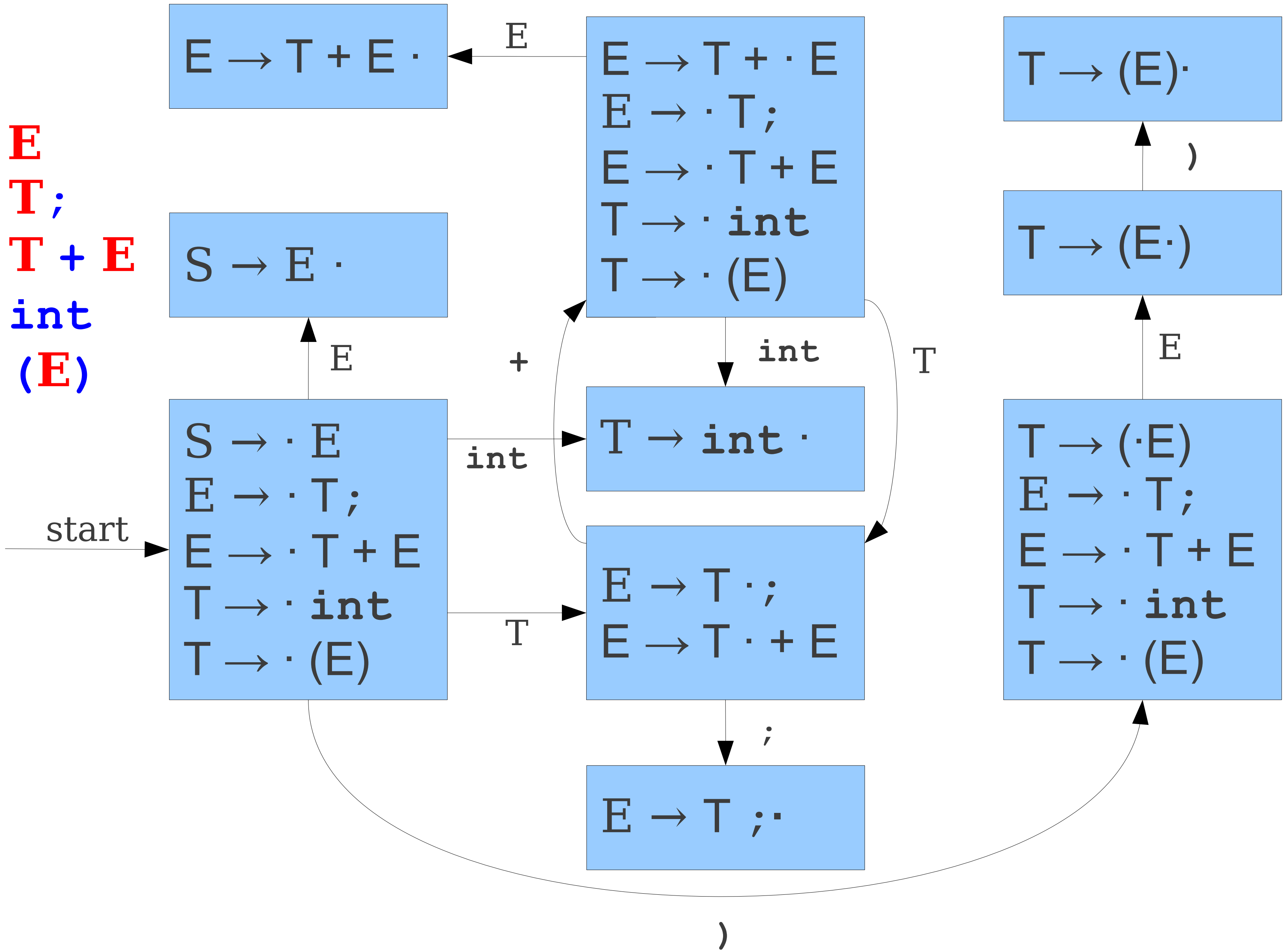
A Deterministic Automaton

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



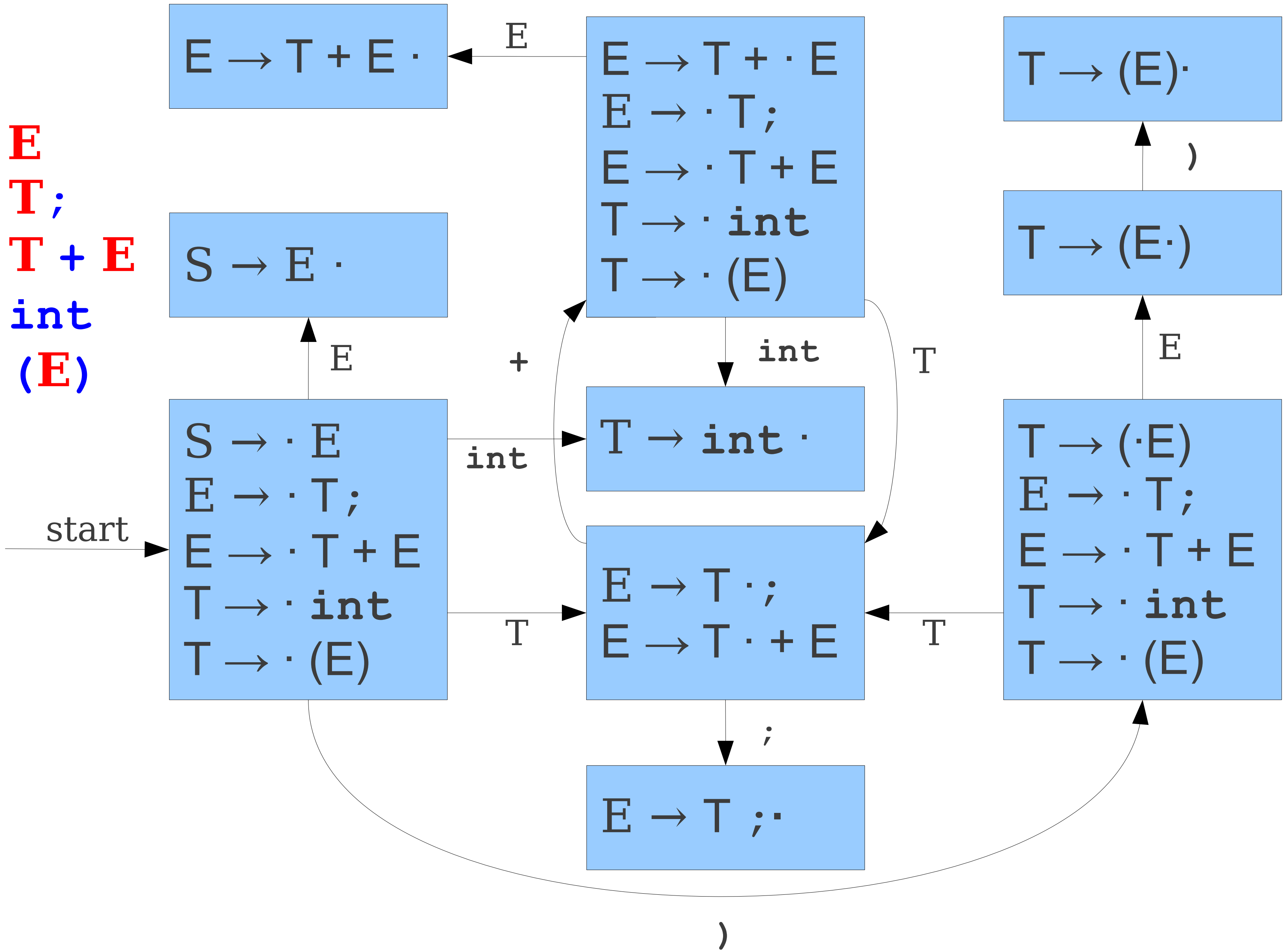
A Deterministic Automaton

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



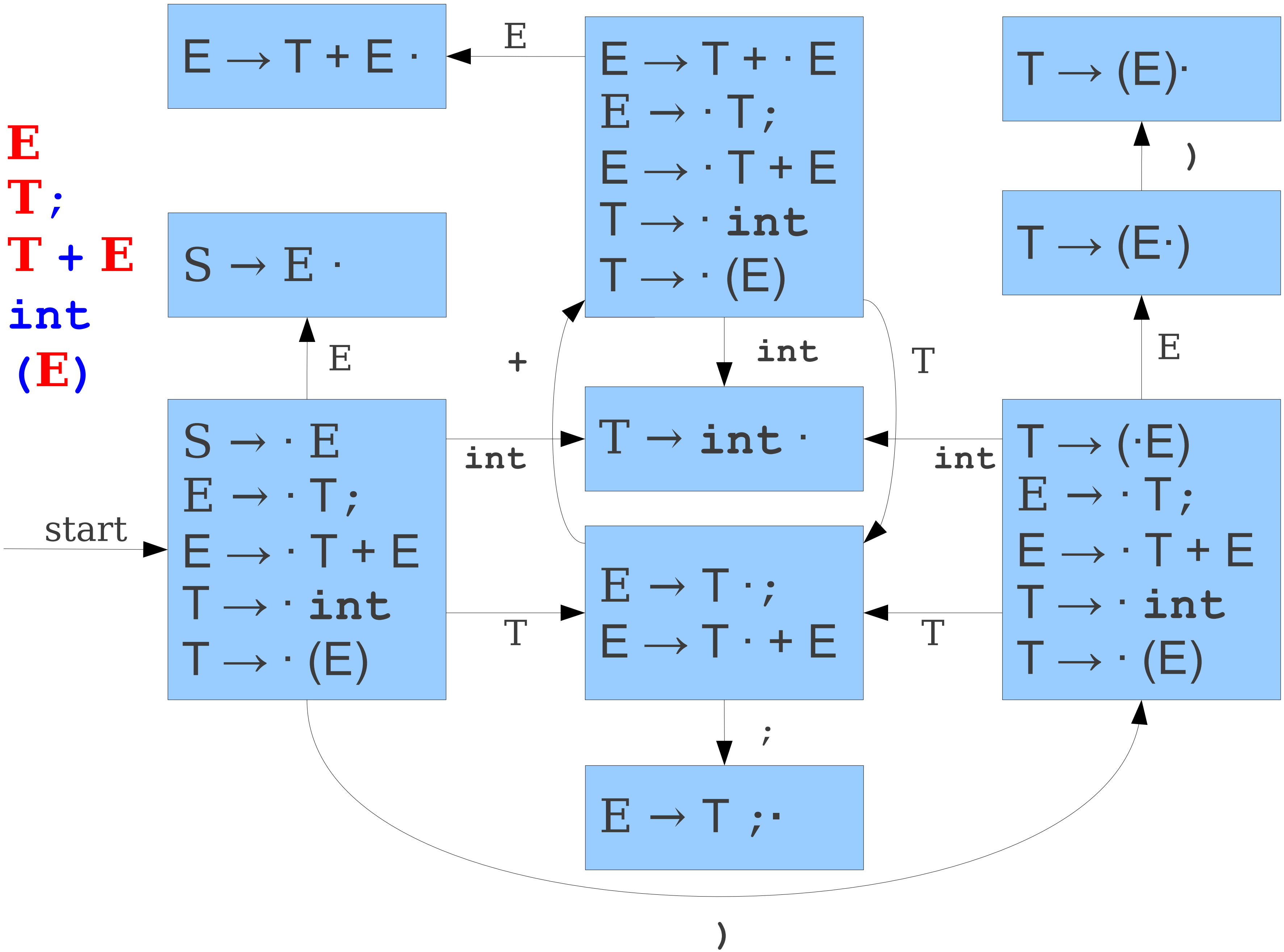
A Deterministic Automaton

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



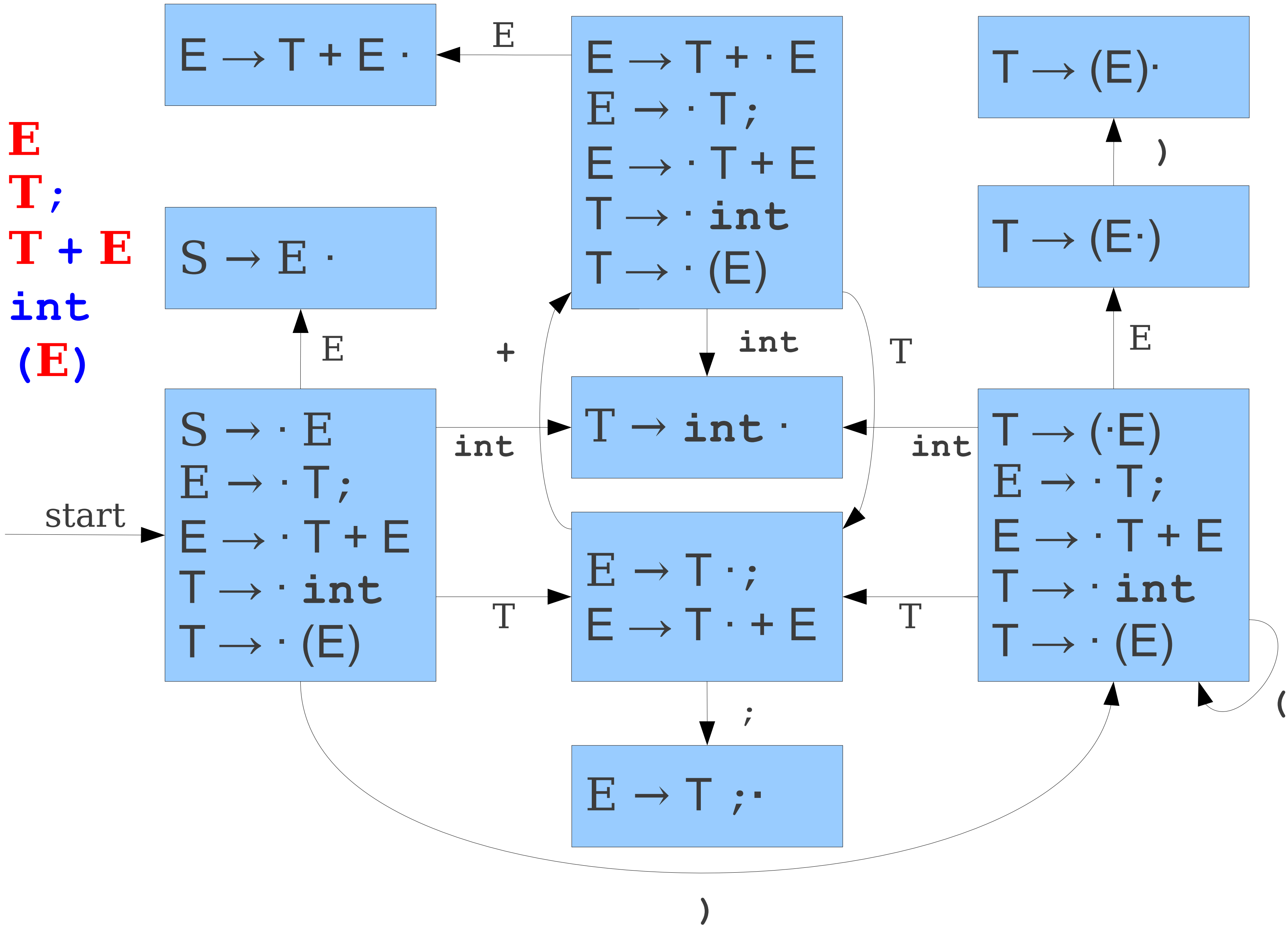
A Deterministic Automaton

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



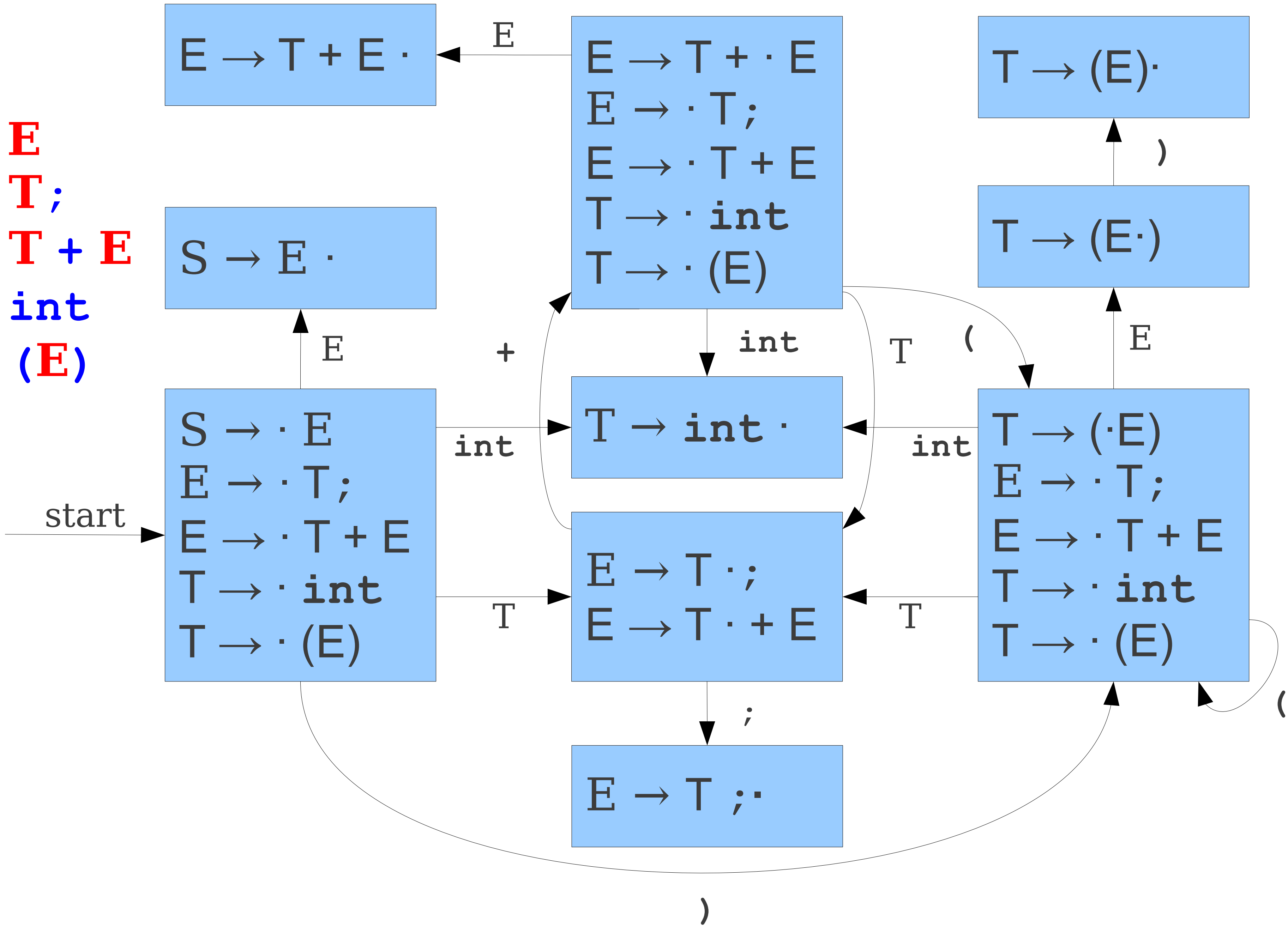
A Deterministic Automaton

S → **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

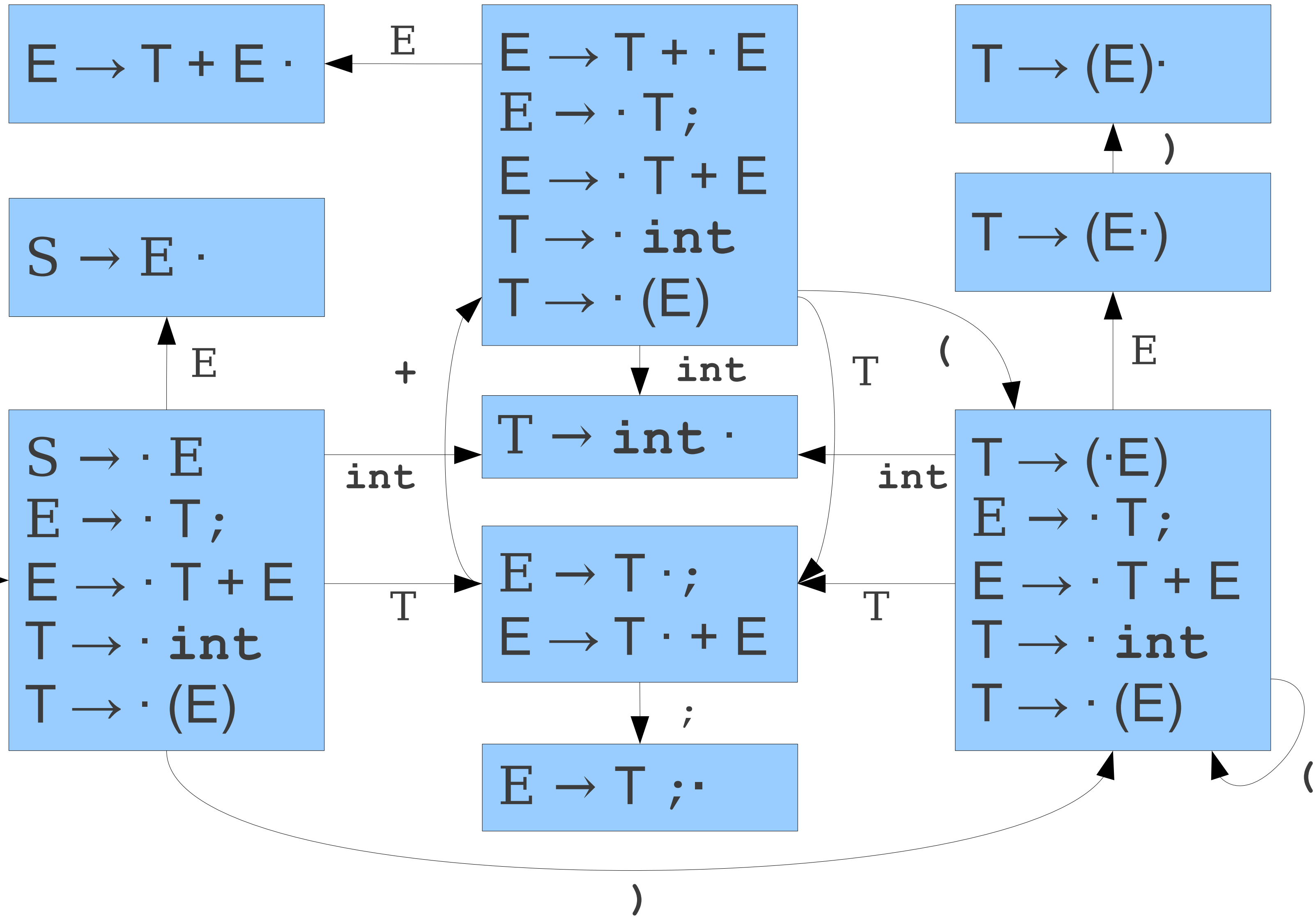
$S \rightarrow E$
$E \rightarrow T$ `;`
$E \rightarrow T$ `+` E
$T \rightarrow$ `int`
$T \rightarrow$ `(`E`)`

int	+	(	int	+	int	;	)	;

LR(0) Parsing

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

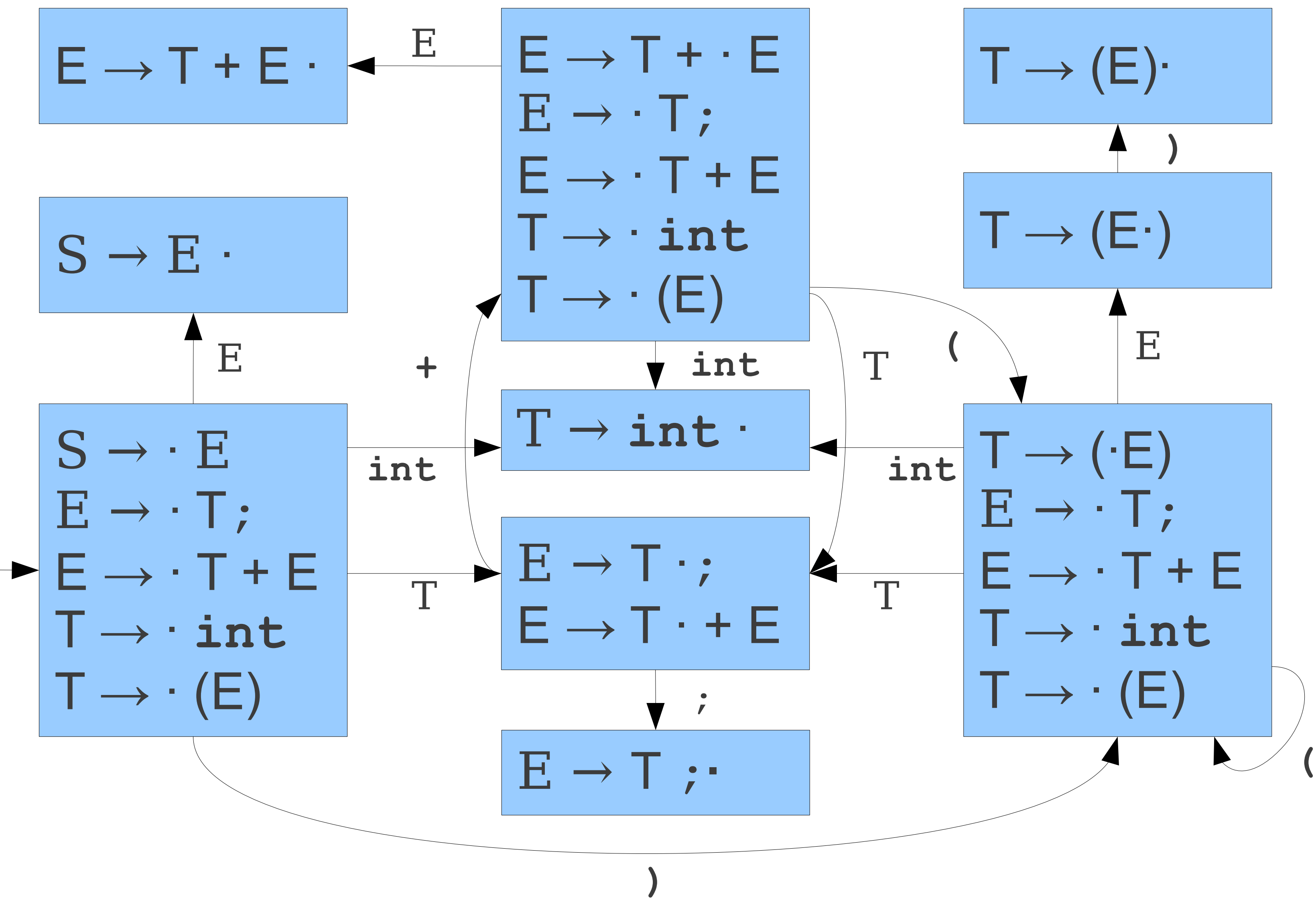
start



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

LR(0) Parsing

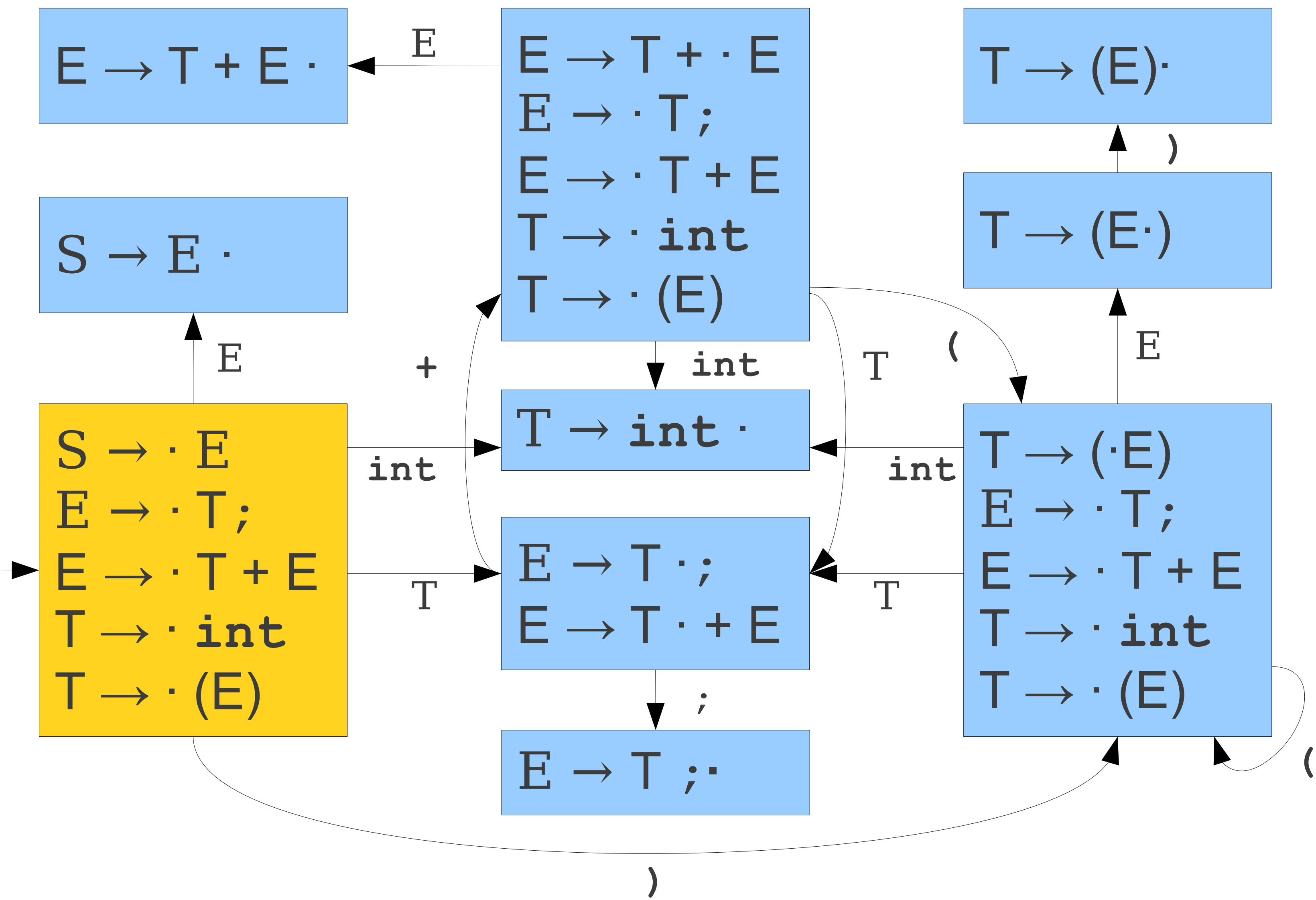
$S \rightarrow E$
$E \rightarrow T;$
$E \rightarrow T + E$
$T \rightarrow \texttt{int}$
$T \rightarrow (E)$



int	+	(	int	+	int	;	)	;

LR(0) Parsing

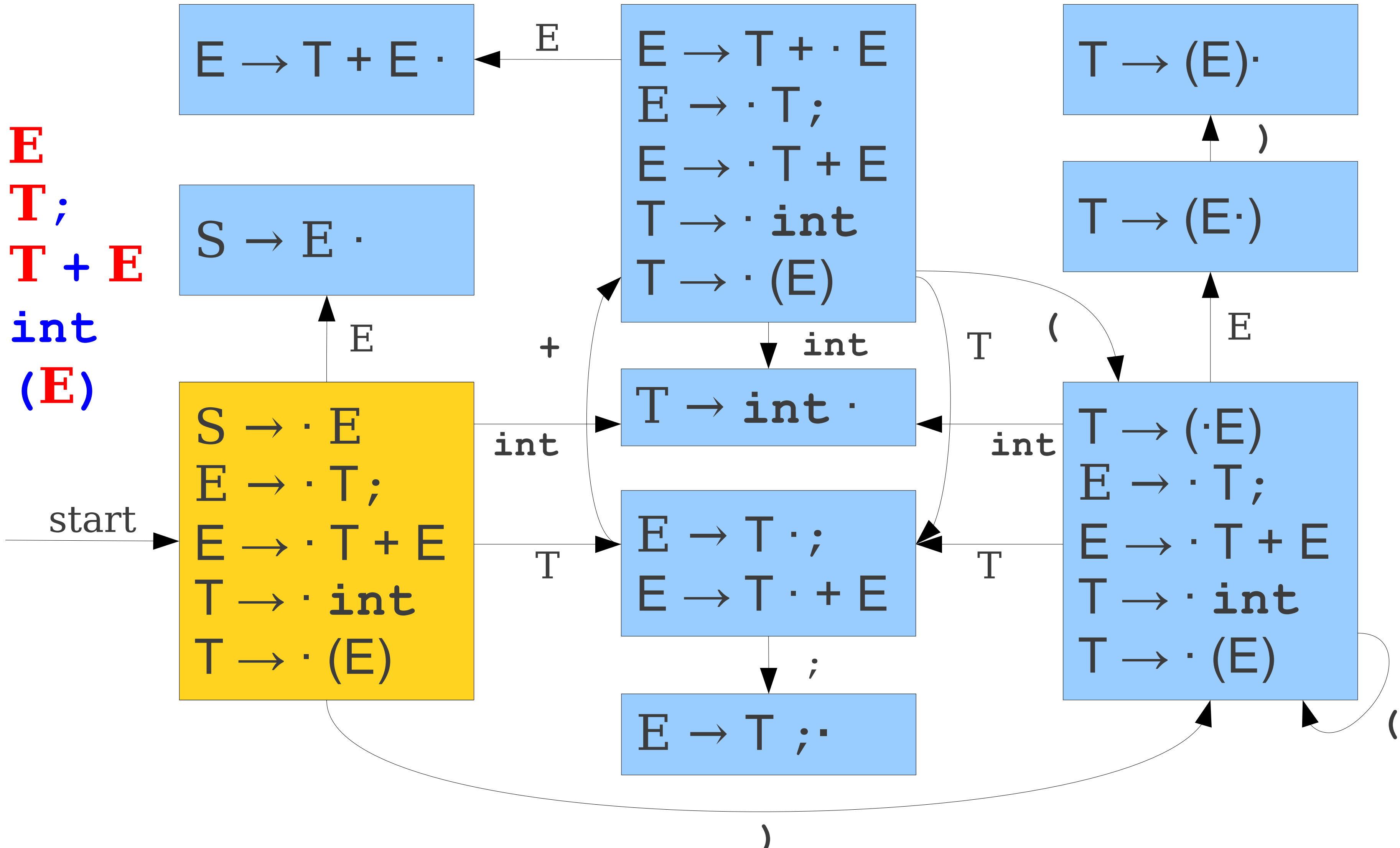
$S \rightarrow E$
$E \rightarrow T;$
$E \rightarrow T + E$
$T \rightarrow$ `int`
$T \rightarrow (E)$

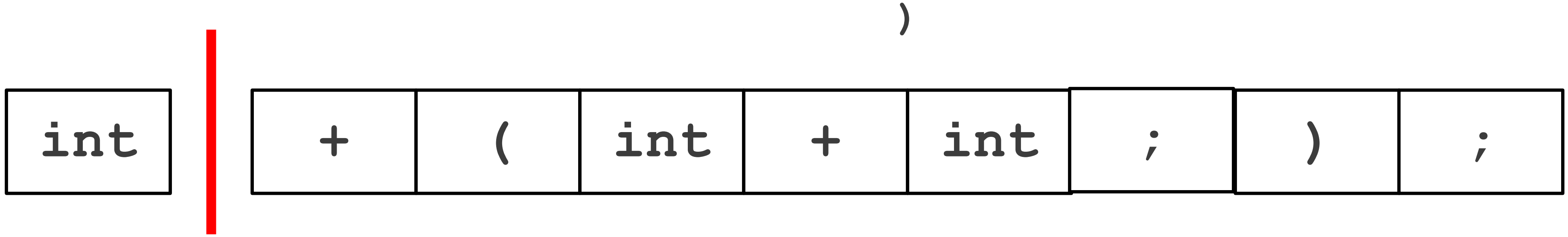


int	+	(	int	+	int	;	)	;

LR(0) Parsing

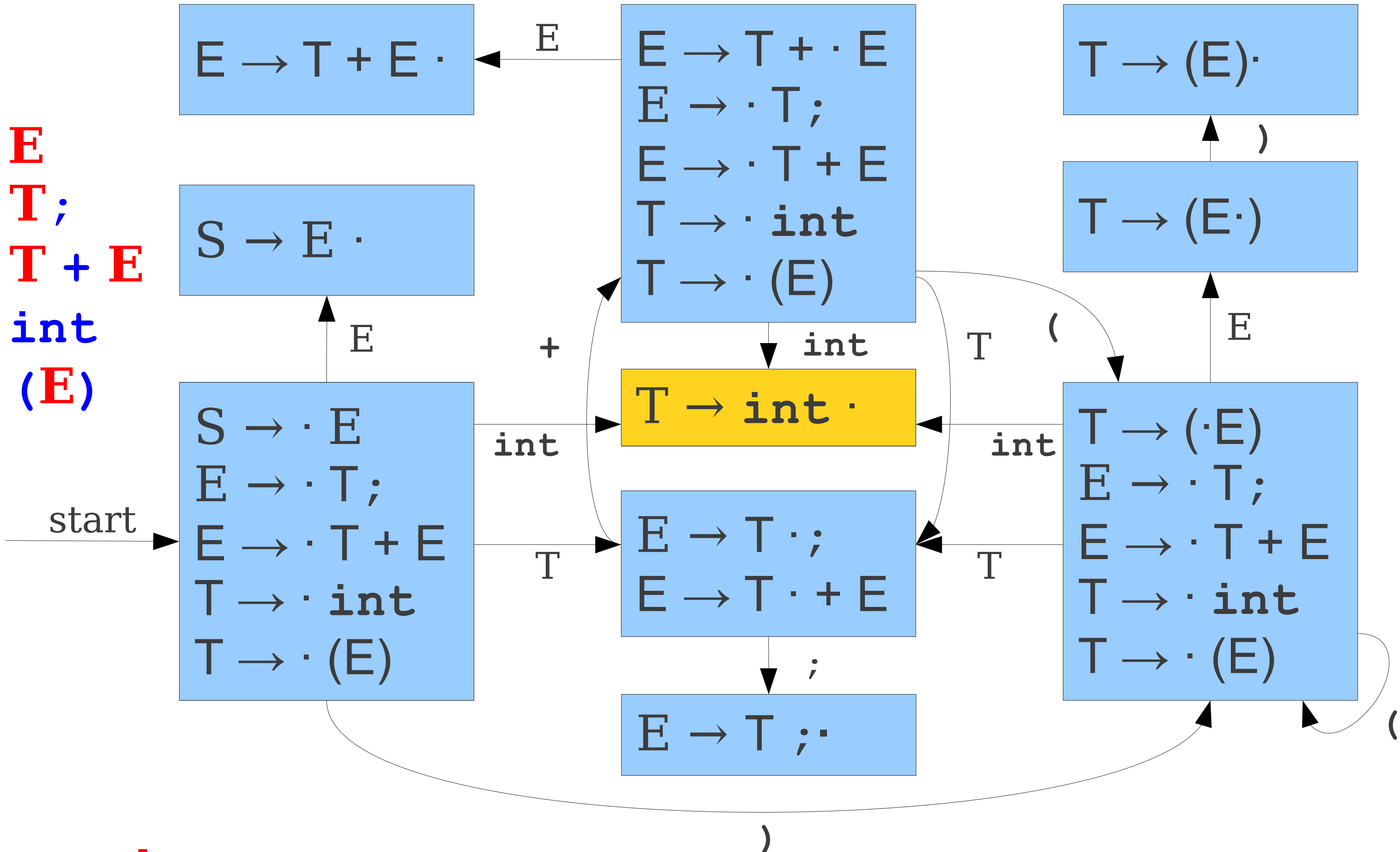
S → **E**
E → **T** **;**
E → **T** **+** **E**
T → **int**
T → **(E)**





LR(0) Parsing

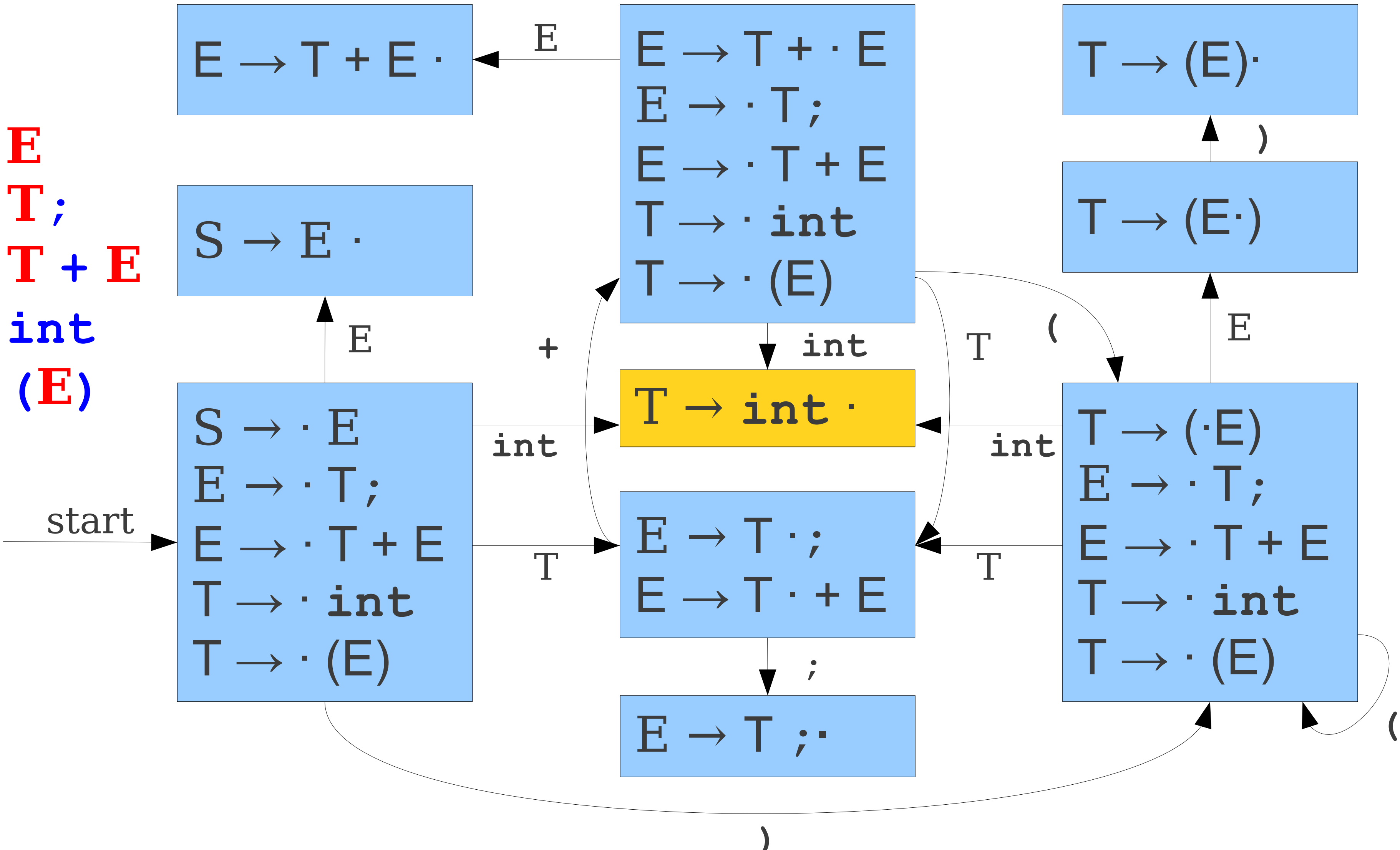
S → **E**
E → **T ;**
E → **T + E**
T → **int**
T → **(E)**



int		+	(	int	+	int	;	)	;

LR(0) Parsing

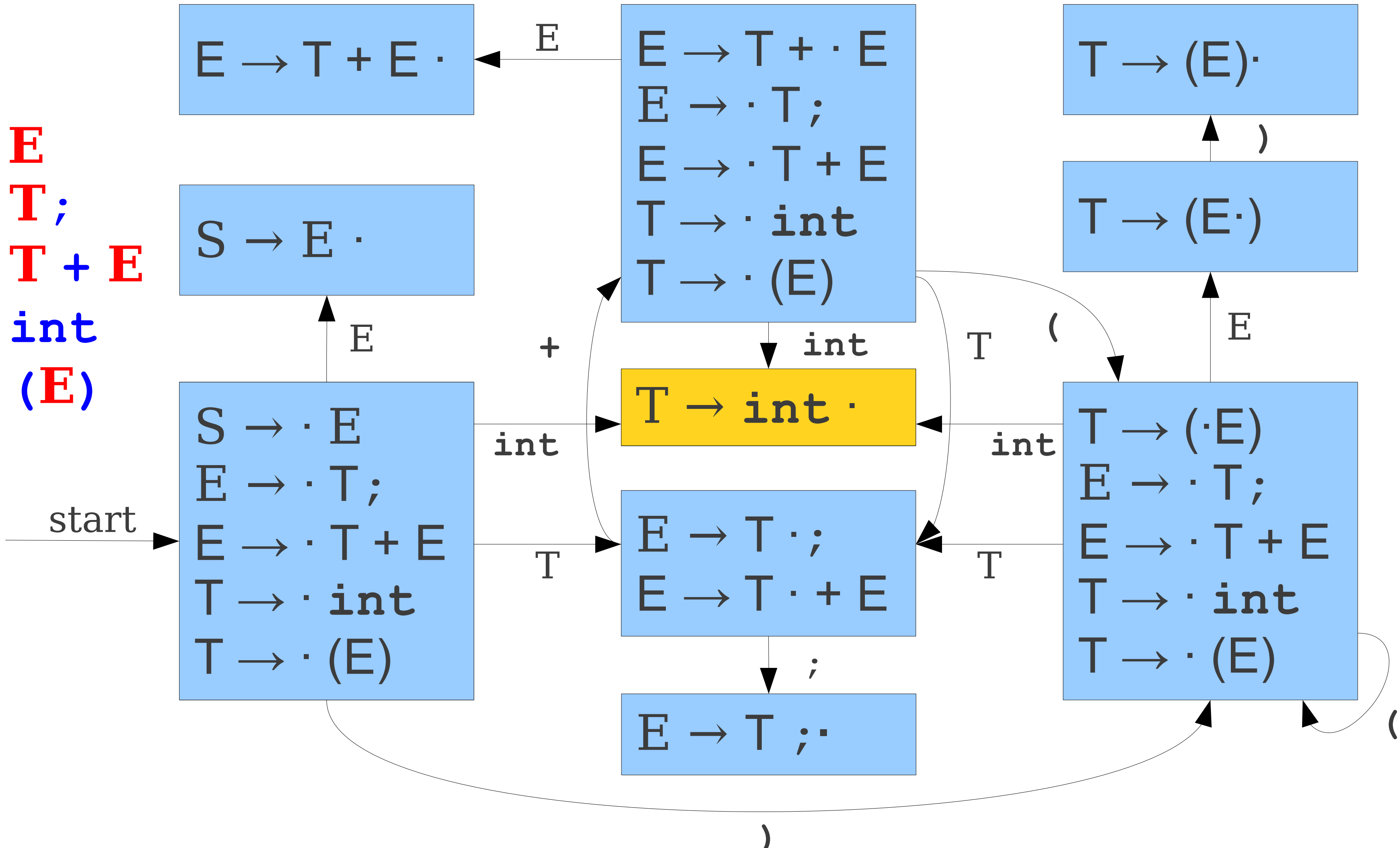
S → E
E → T;
E → T + E
T → int
T → (E)

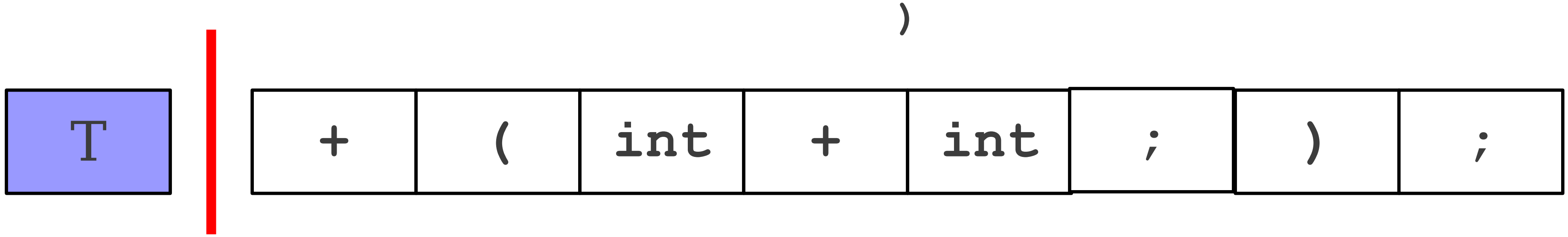


+	(	int	+	int	;	)	;

LR(0) Parsing

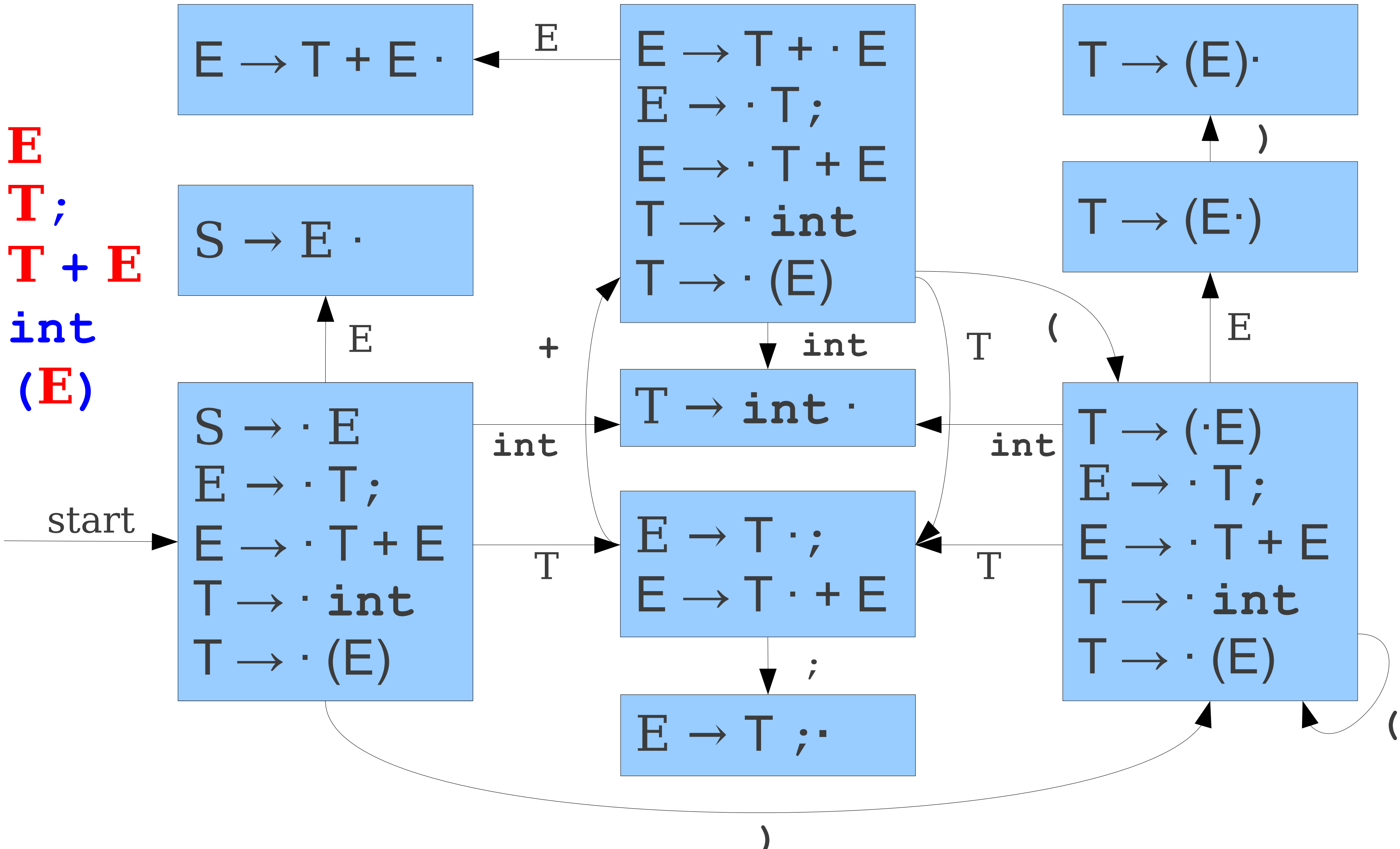
S → E
E → T;
E → T + E
T → int
T → (E)





LR(0) Parsing

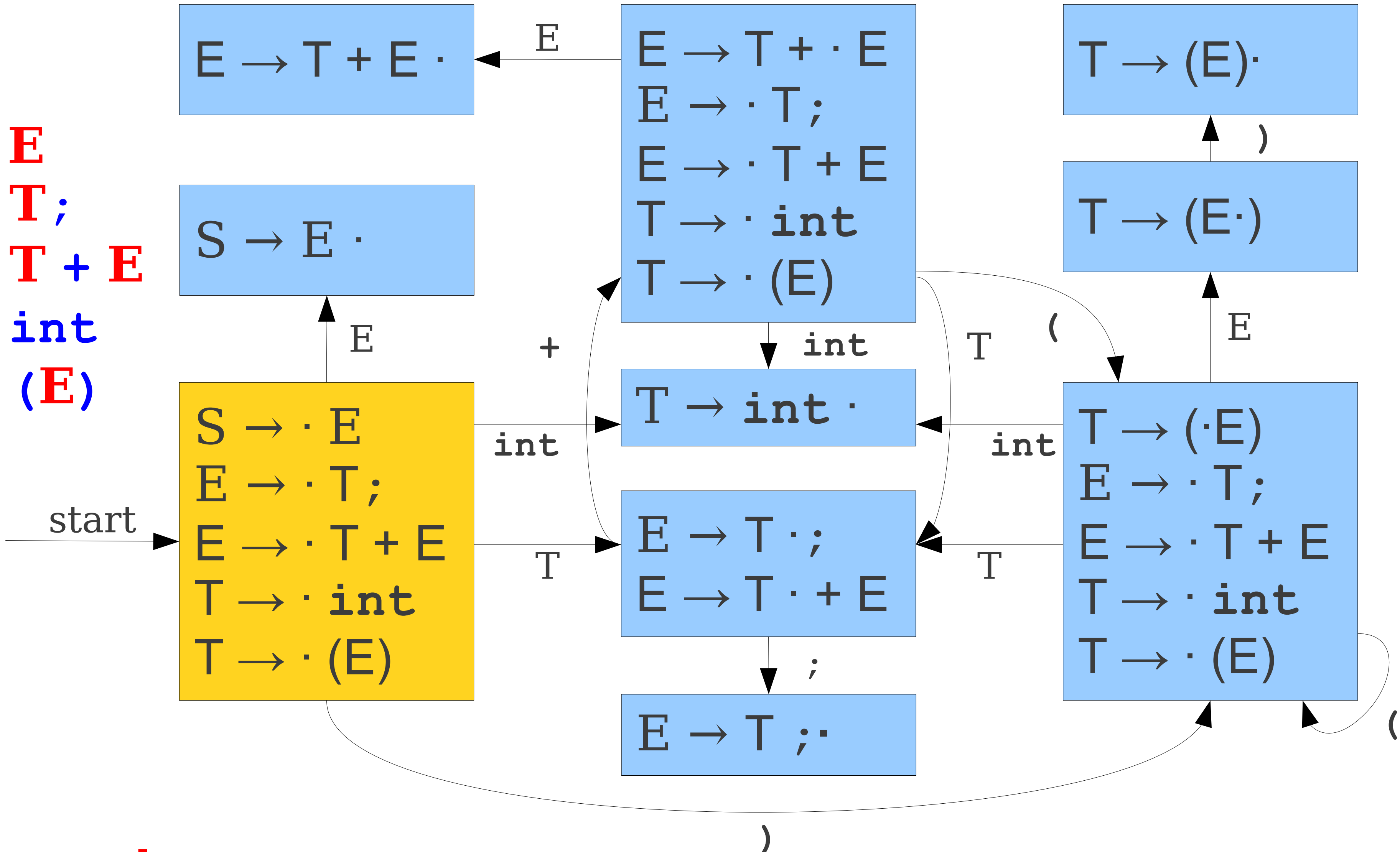
S → **E**
E → **T ;**
E → **T + E**
T → **int**
T → **(E)**



T		+	(	int	+	int	;	)	;

LR(0) Parsing

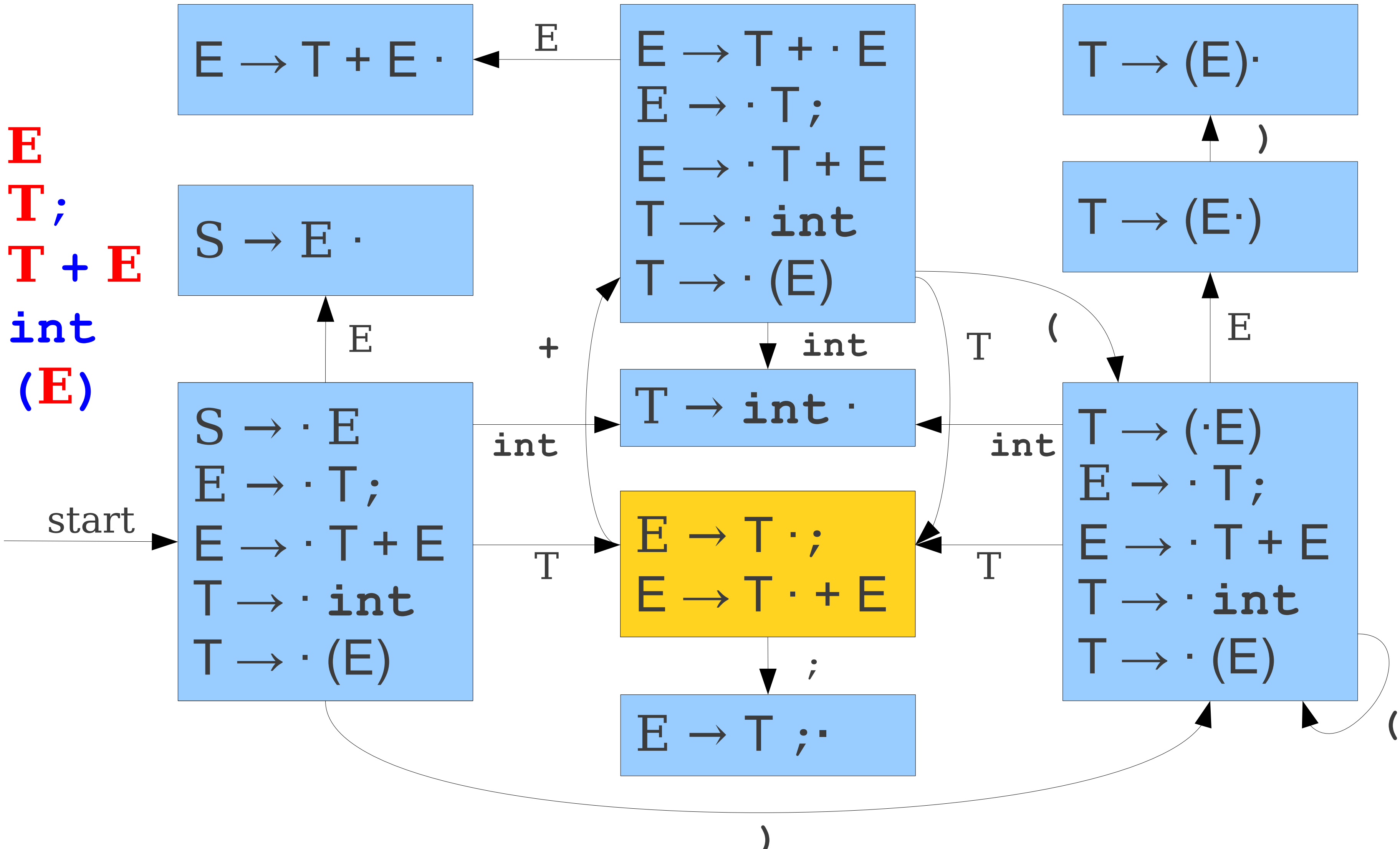
$S \rightarrow E$
$E \rightarrow T;$
$E \rightarrow T + E$
$T \rightarrow \texttt{int}$
$T \rightarrow (E)$

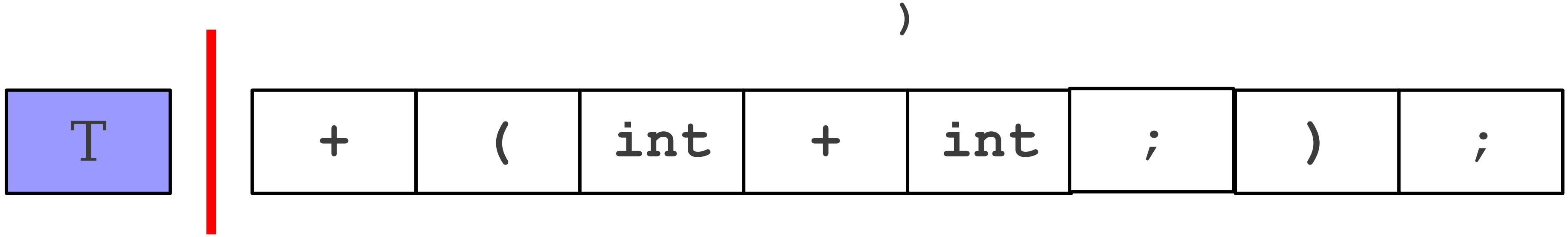


T	+	(	int	+	int	;	)	;

LR(0) Parsing

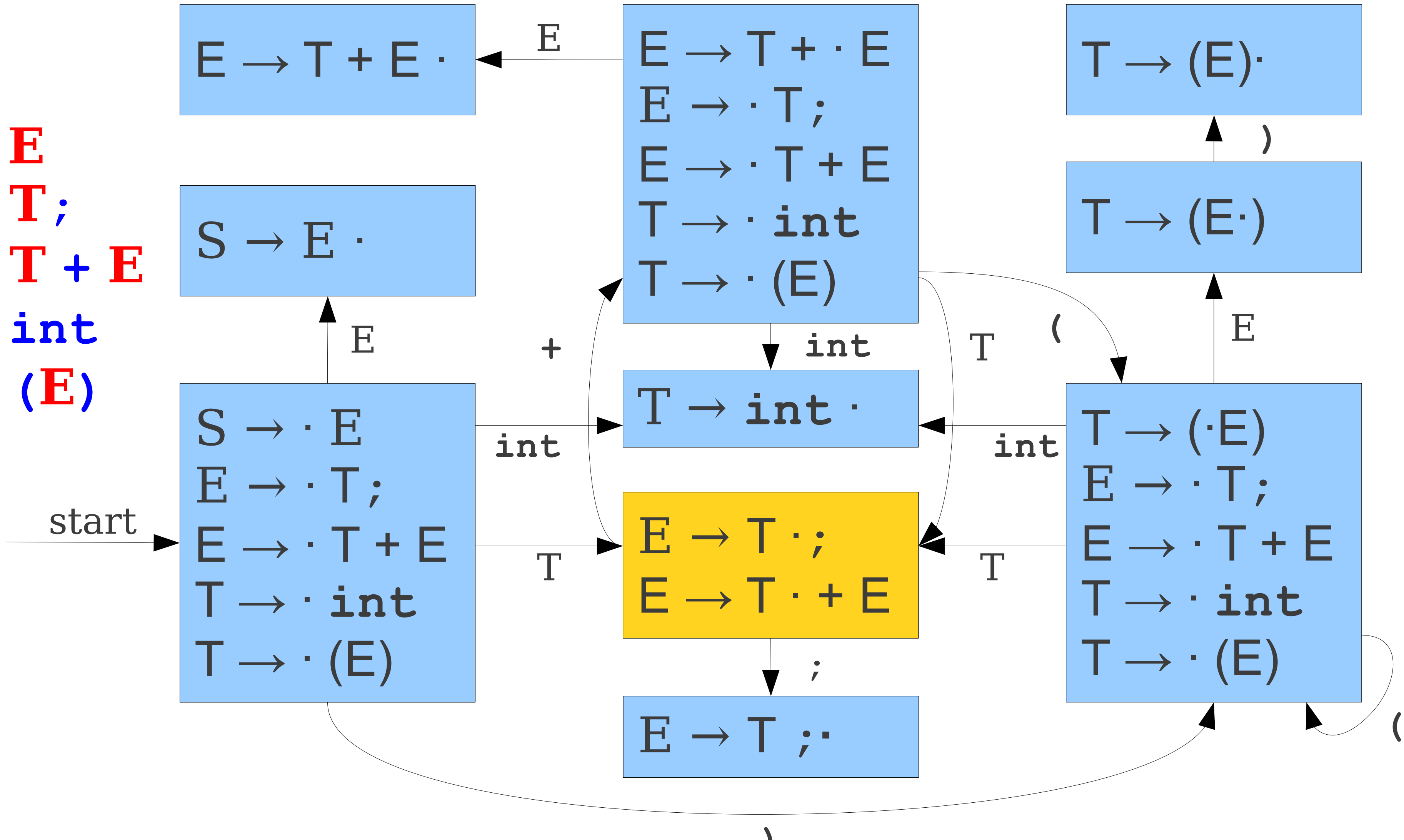
$S \to E$
$E \to T;$
$E \to T + E$
$T \to \texttt{int}$
$T \to (E)$





LR(0) Parsing

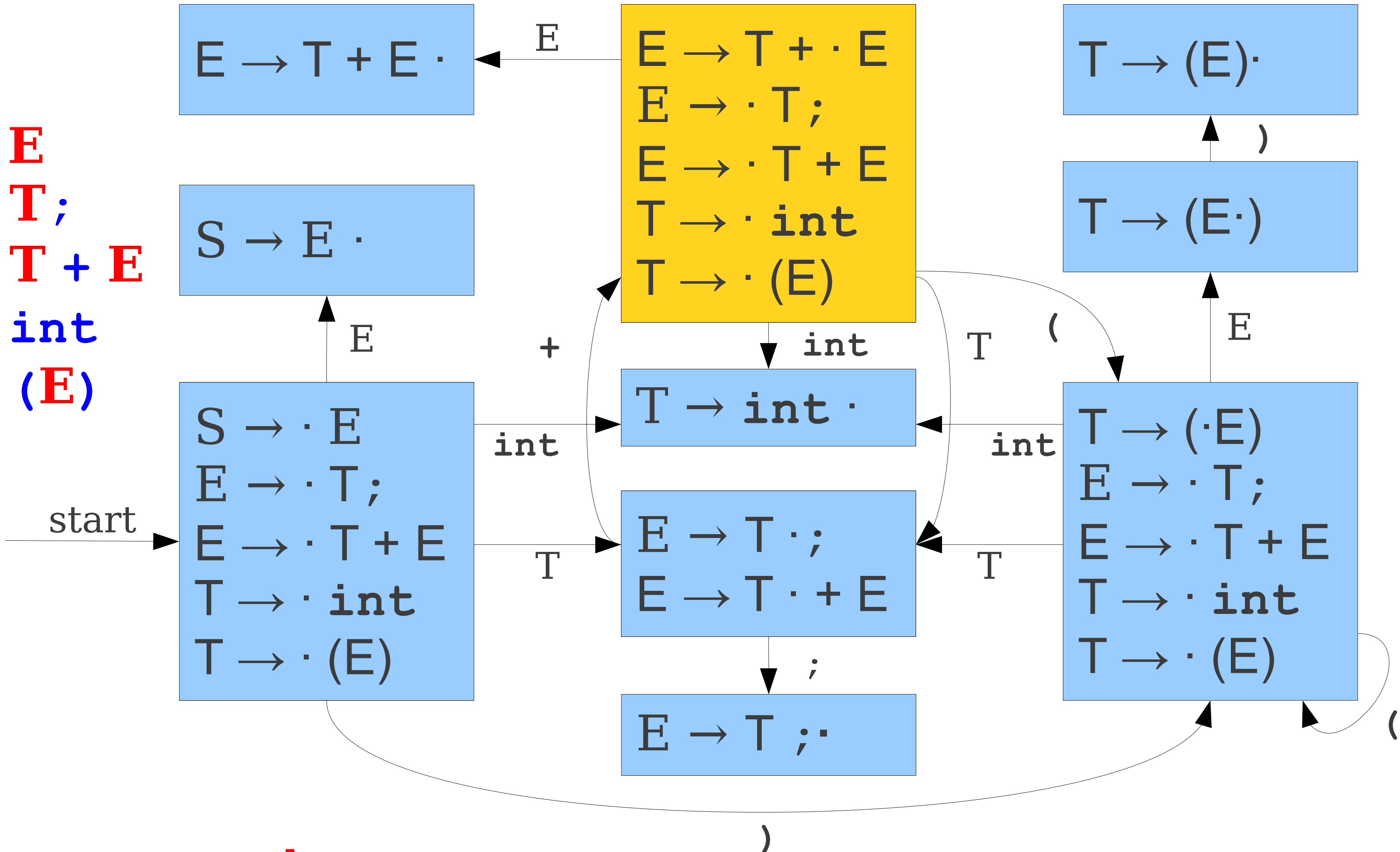
$S \to E$
$E \to T;$
$E \to T + E$
$T \to \texttt{int}$
$T \to (E)$



T	+		(	int	+	int	;	)	;

LR(0) Parsing

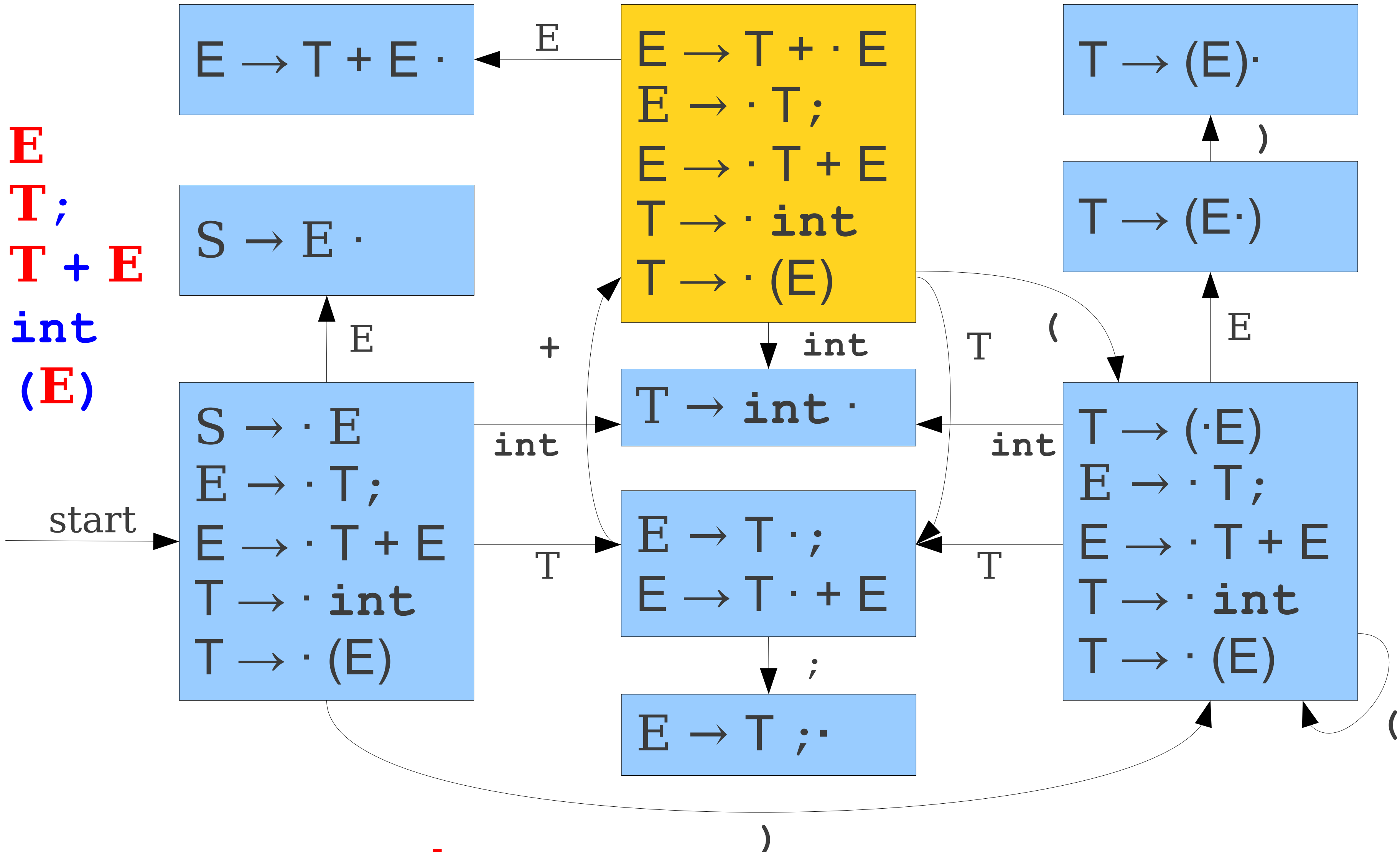
$S \rightarrow E$
$E \rightarrow T;$
$E \rightarrow T + E$
$T \rightarrow$ **int**
$T \rightarrow (E)$



T	+	(	int	+	int	;	)	;

LR(0) Parsing

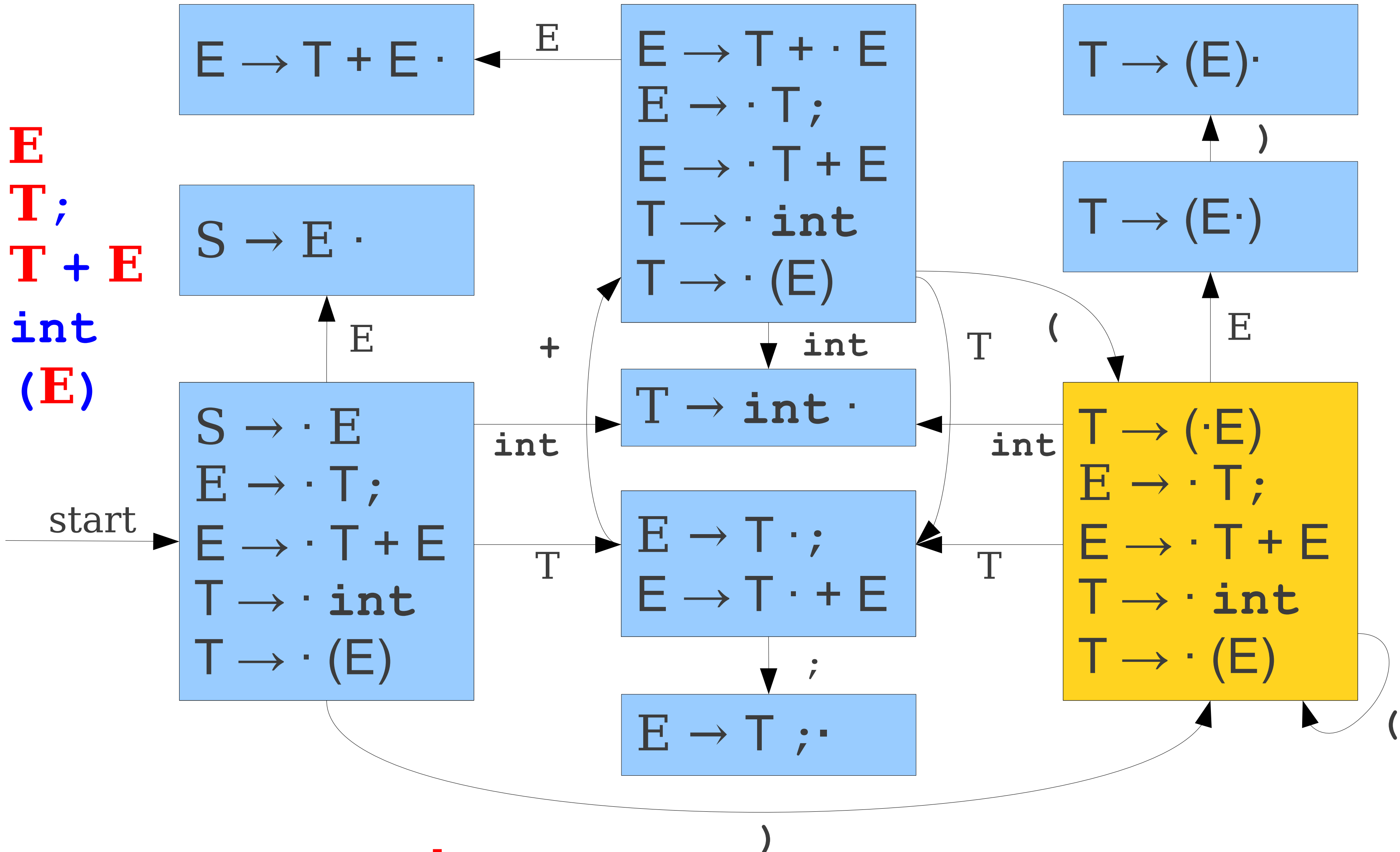
S → **E**
E → **T** ;
E → **T** + **E**
T → `int`
T → (**E**)



T	+	(	int	+	int	;	)	;

LR(0) Parsing

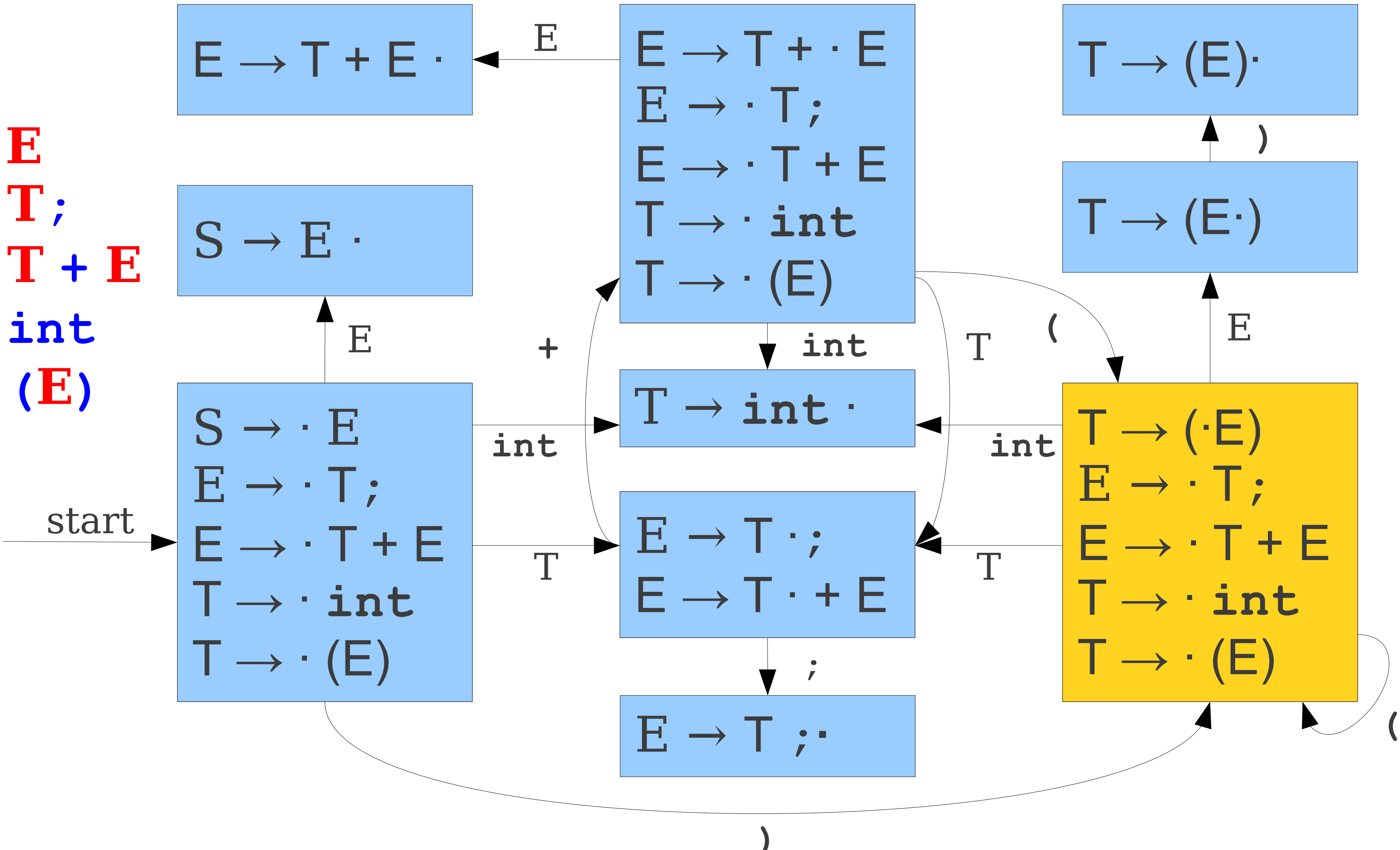
S → **E**
E → **T** `;`
E → **T** + **E**
T → `int`
T → `(`**E**`)`



T	+	(	int	+	int	;	)	;

LR(0) Parsing

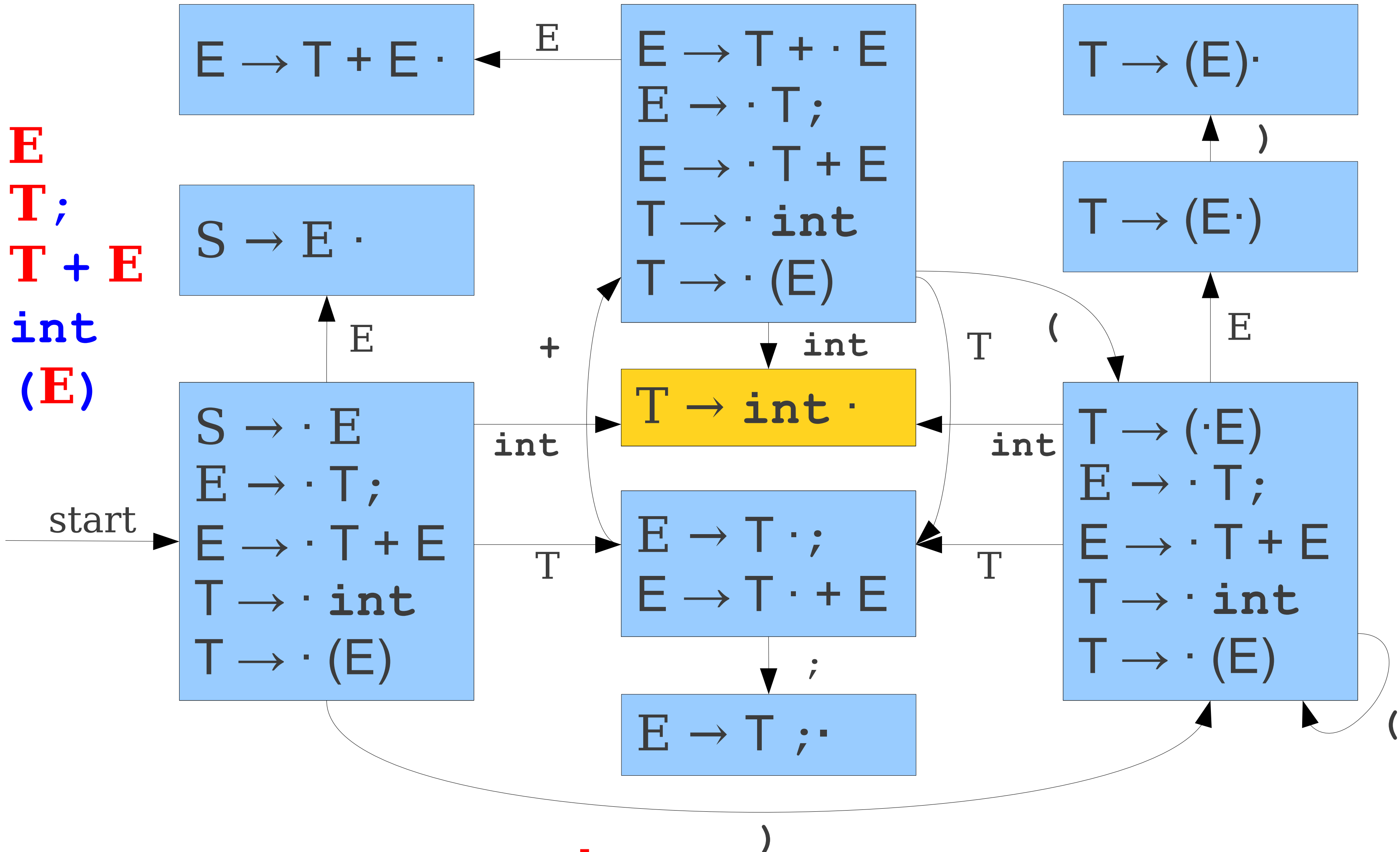
S → **E**
E → **T** ;
E → **T** + **E**
T → int
T → (**E**)



T	+	(	int	+	int	;	)	;

LR(0) Parsing

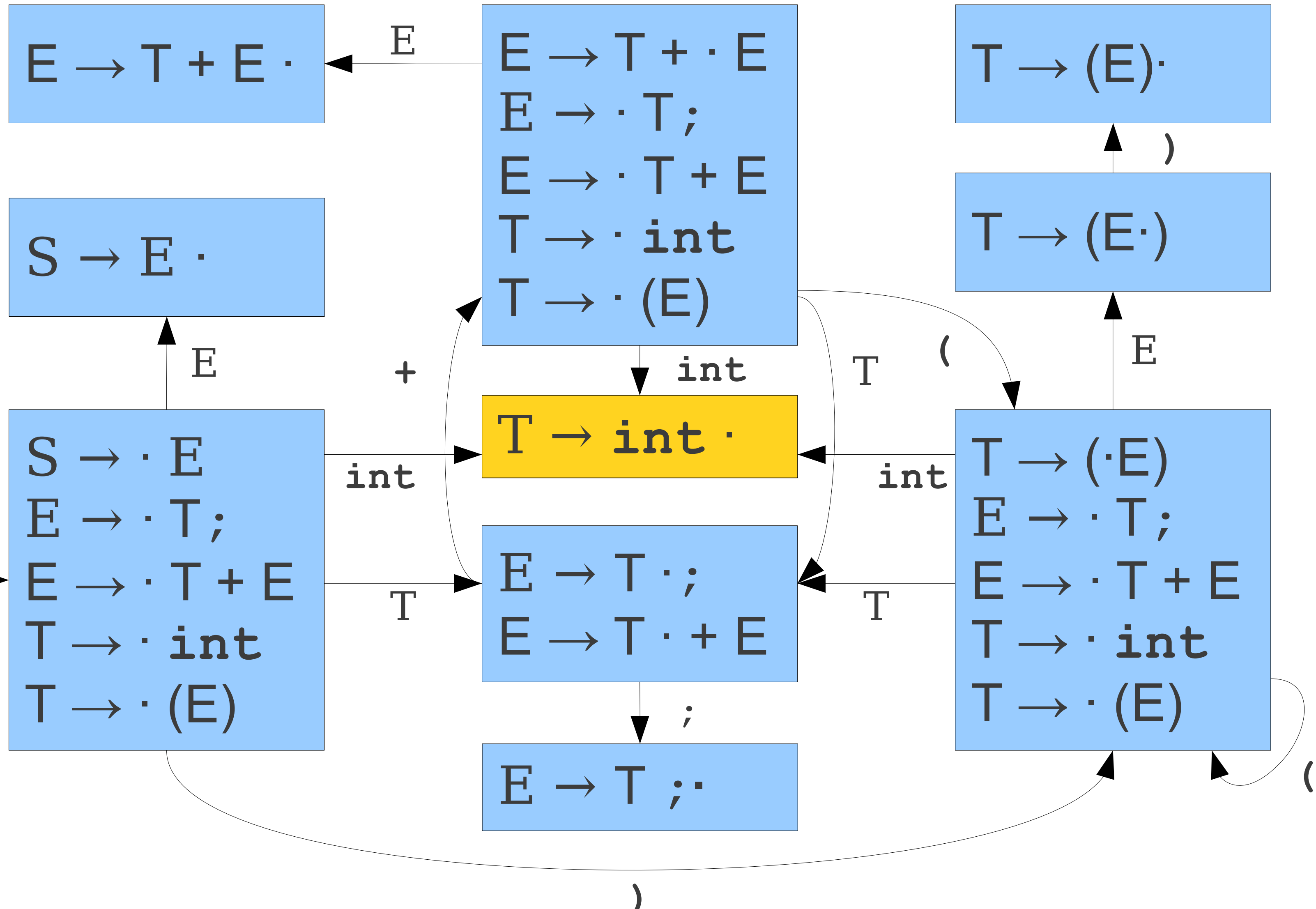
S → **E**
E → **T** `;`
E → **T** `+` **E**
T → `int`
T → `(`**E**`)`



T	+	(	int		+	int	;	)	;

LR(0) Parsing

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

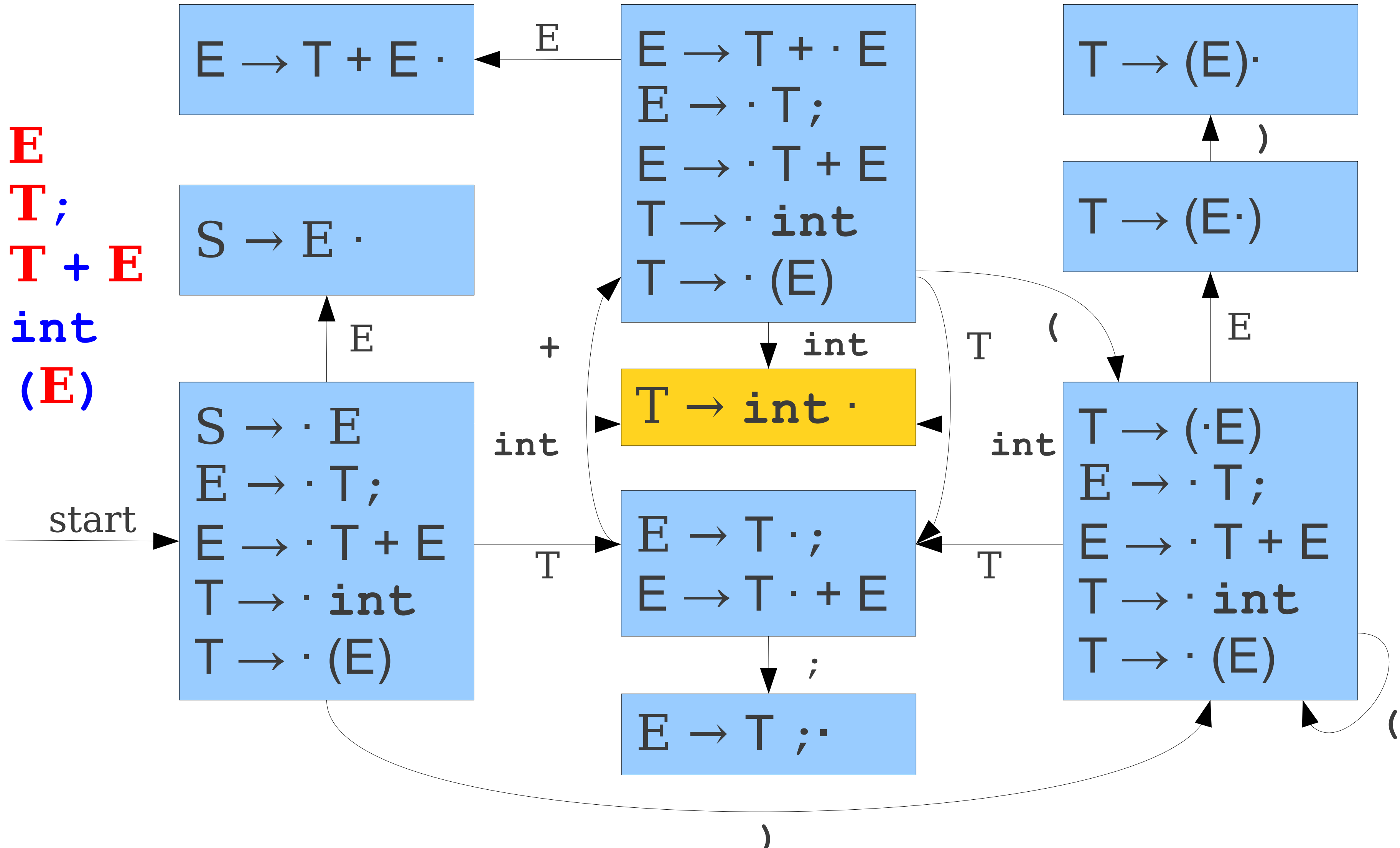


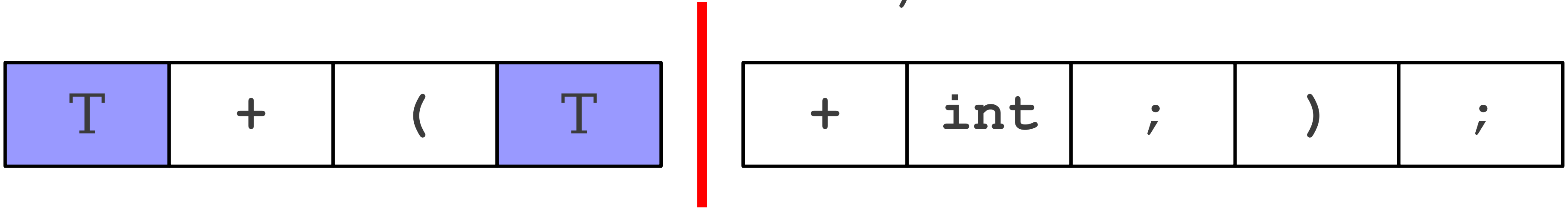
T	+	(

+	int	;	)	;

LR(0) Parsing

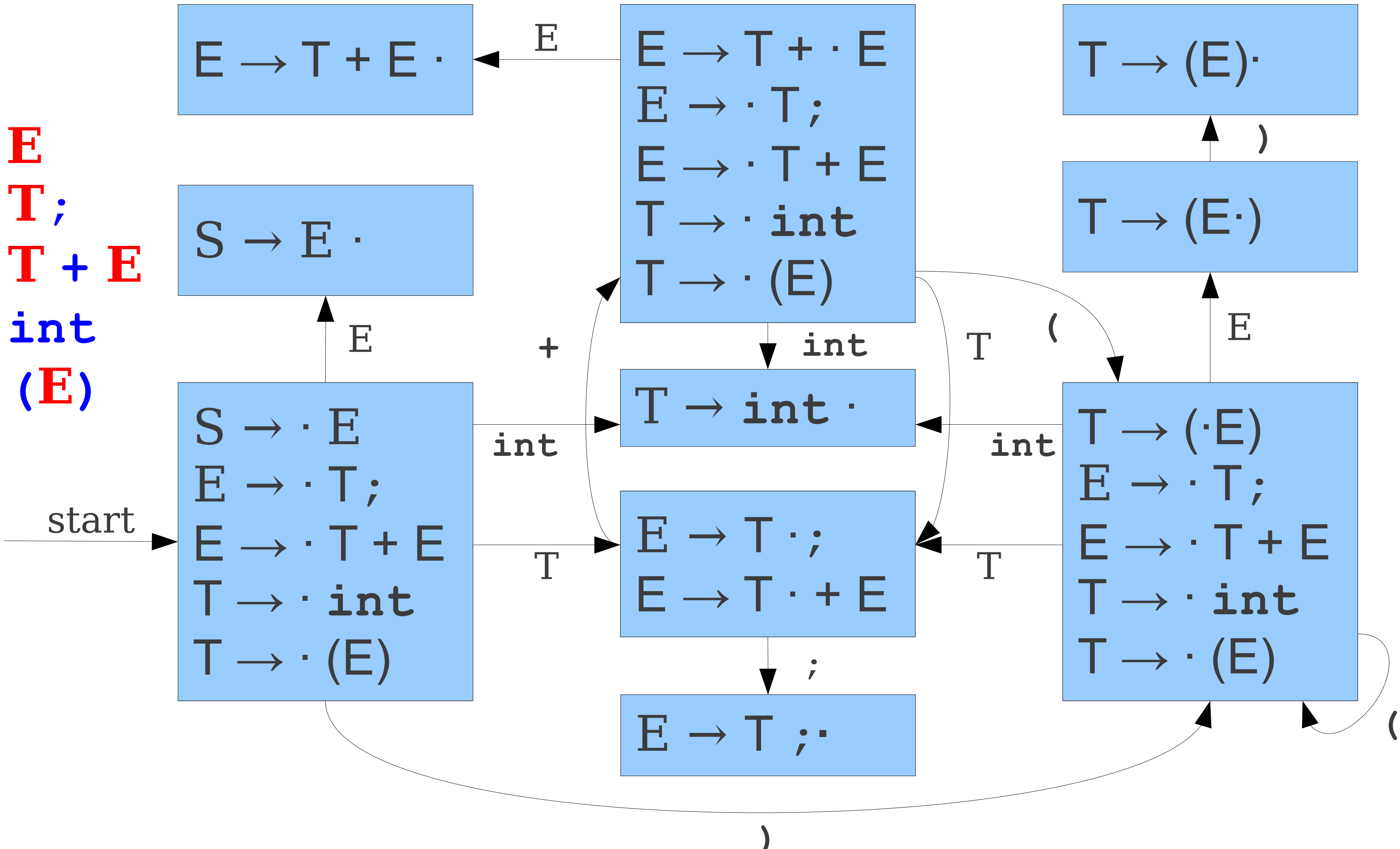
S → **E**
E → **T** ;
E → **T** + **E**
T → int
T → (**E**)





LR(0) Parsing

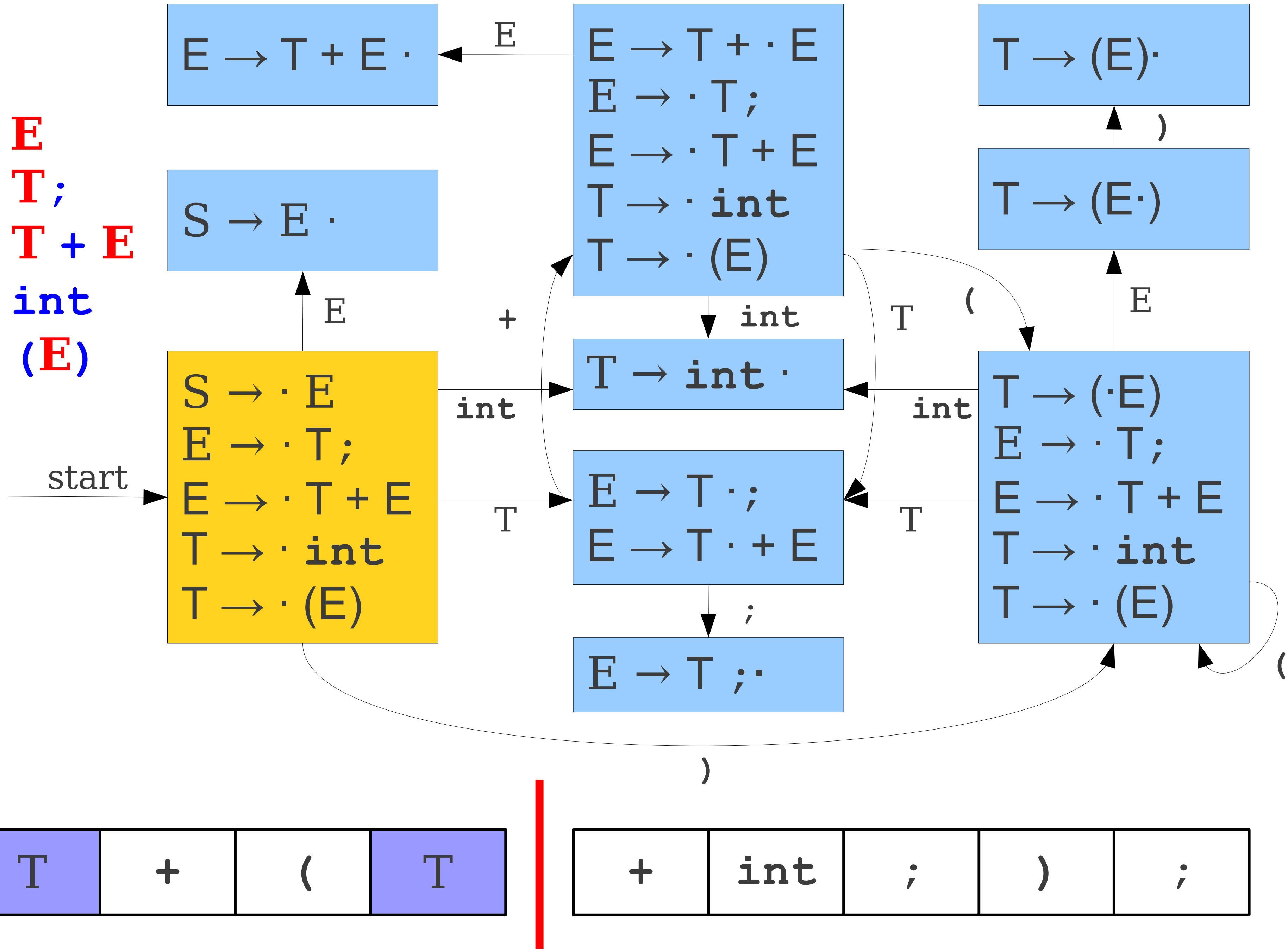
S → **E**
E → **T** `;`
E → **T** `+` **E**
T → `int`
T → `(E)`



T	+	(	T		+	int	;	)	;

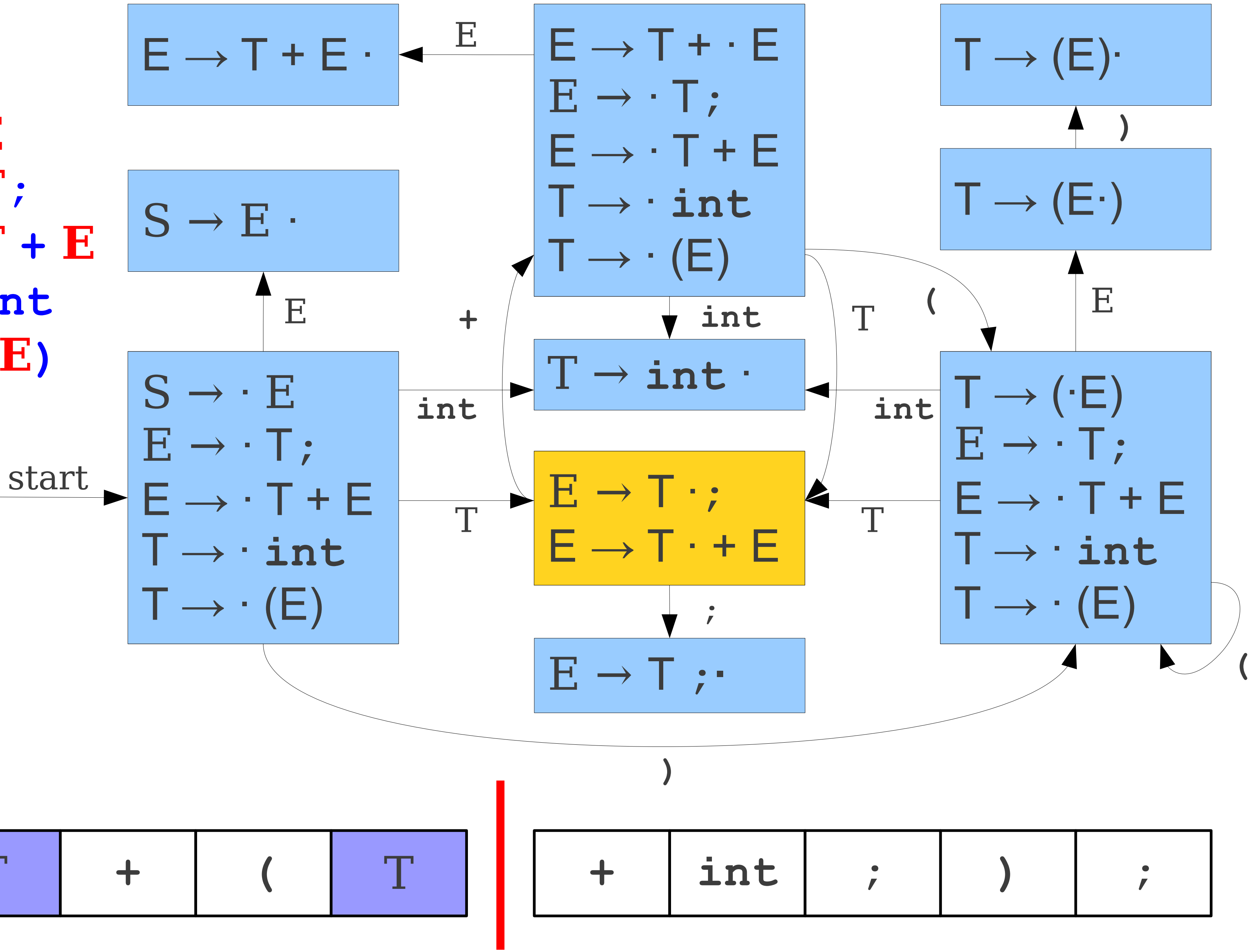
LR(0) Parsing

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



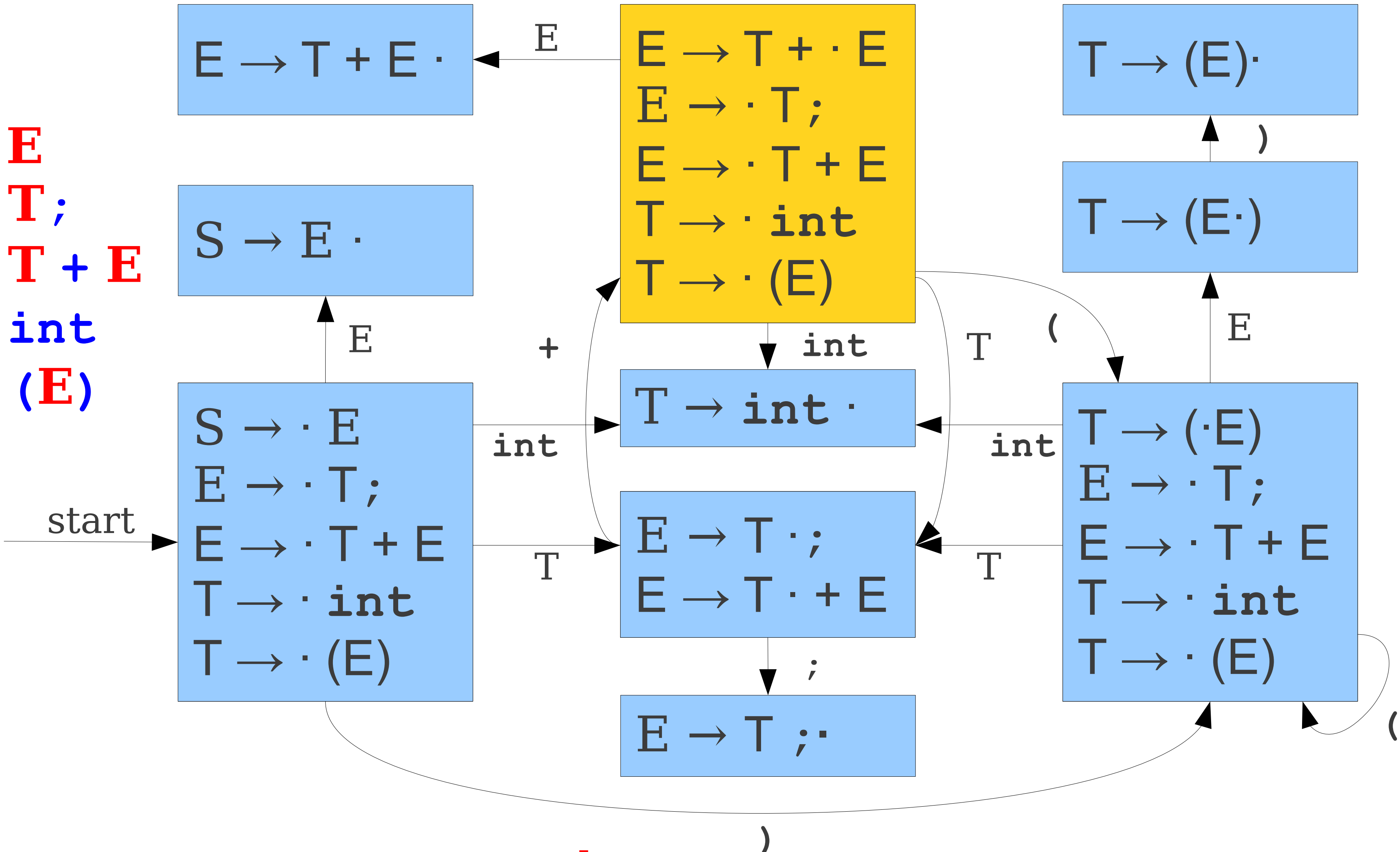
LR(0) Parsing

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



LR(0) Parsing

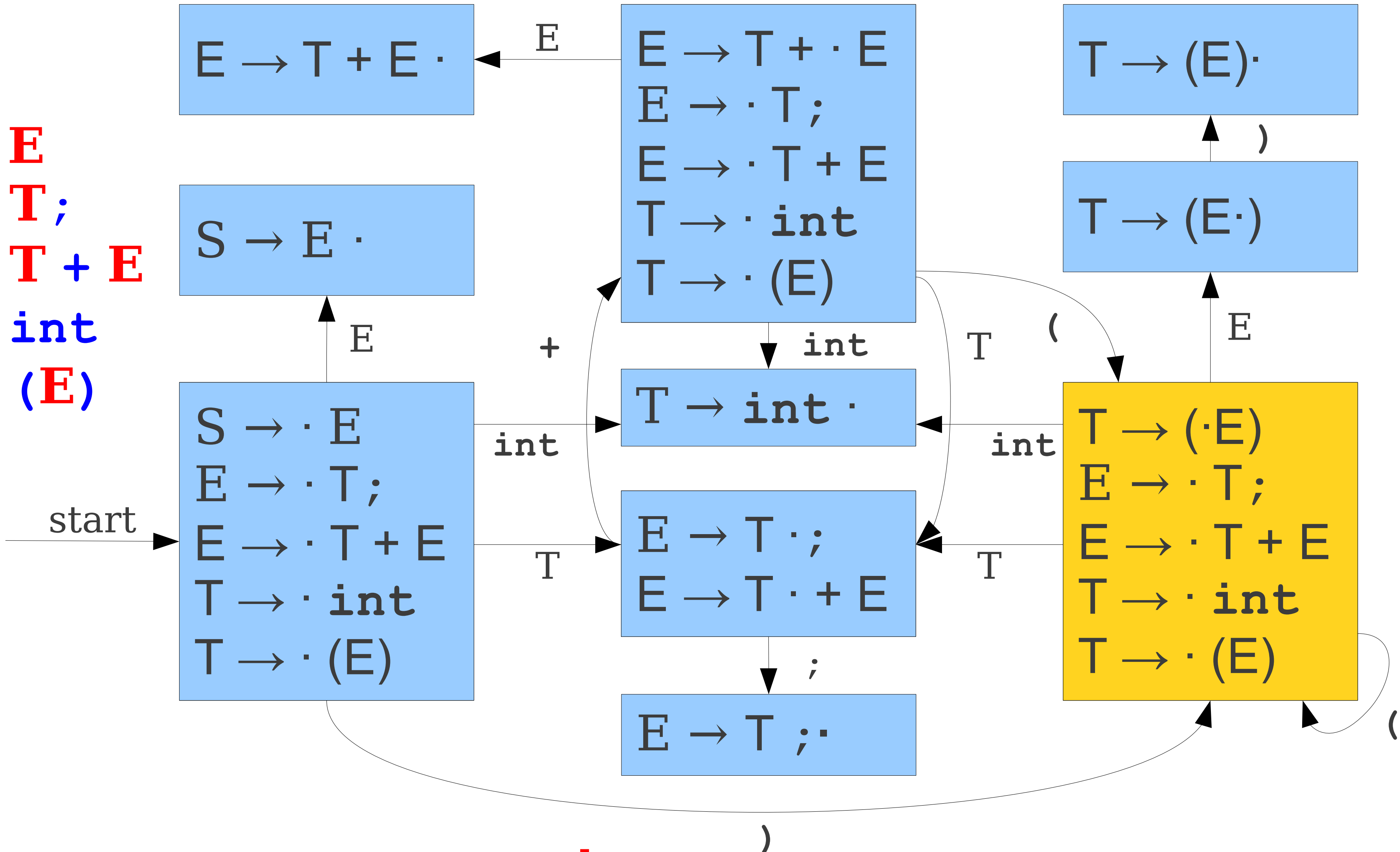
$S \rightarrow E$
$E \rightarrow T;$
$E \rightarrow T + E$
$T \rightarrow \texttt{int}$
$T \rightarrow (E)$



T	+	(	T	|	+	int	;	)	;

LR(0) Parsing

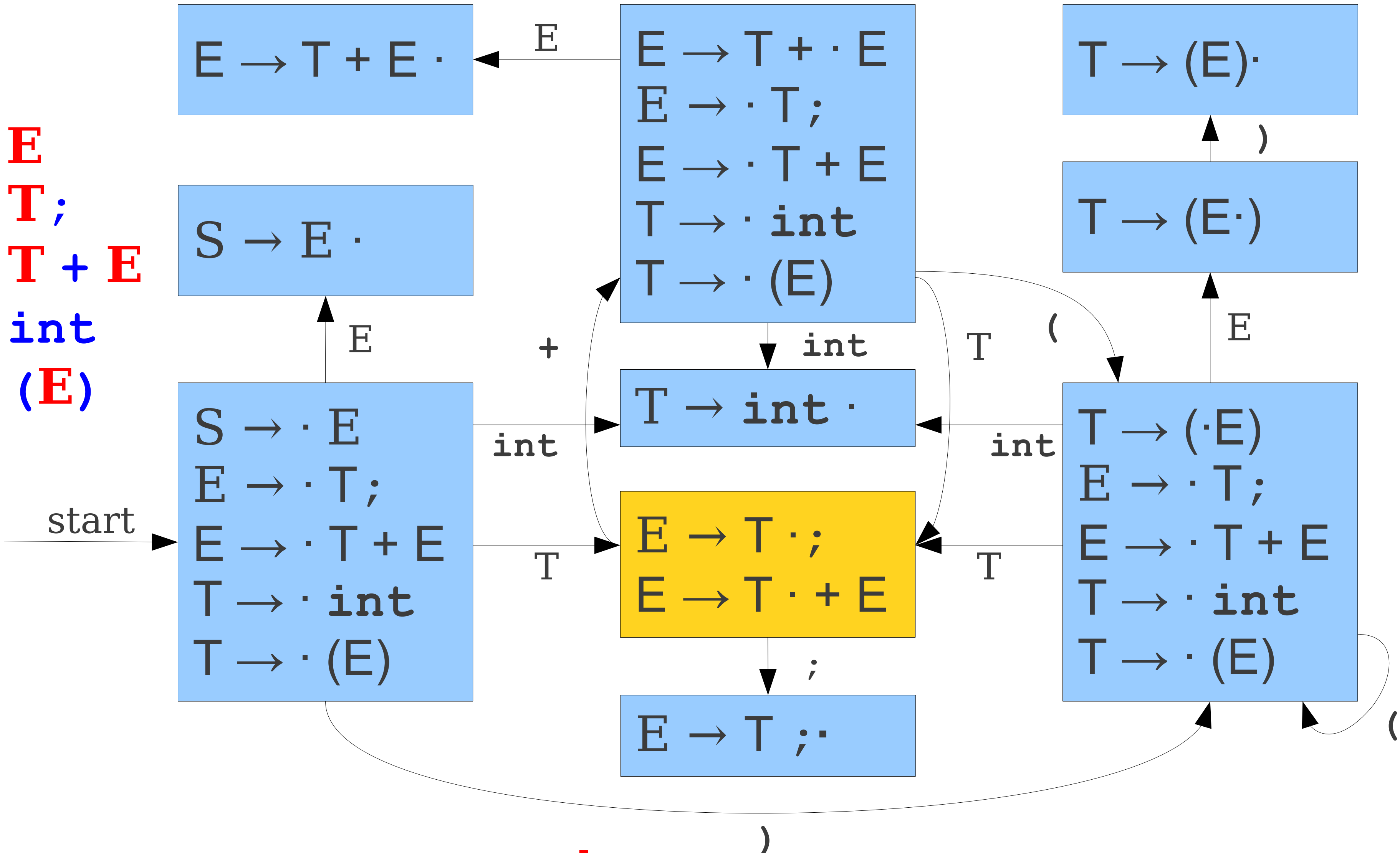
S → E
E → T;
E → T + E
T → int
T → (E)



T	+	(	T	|	+	int	;	)	;

LR(0) Parsing

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

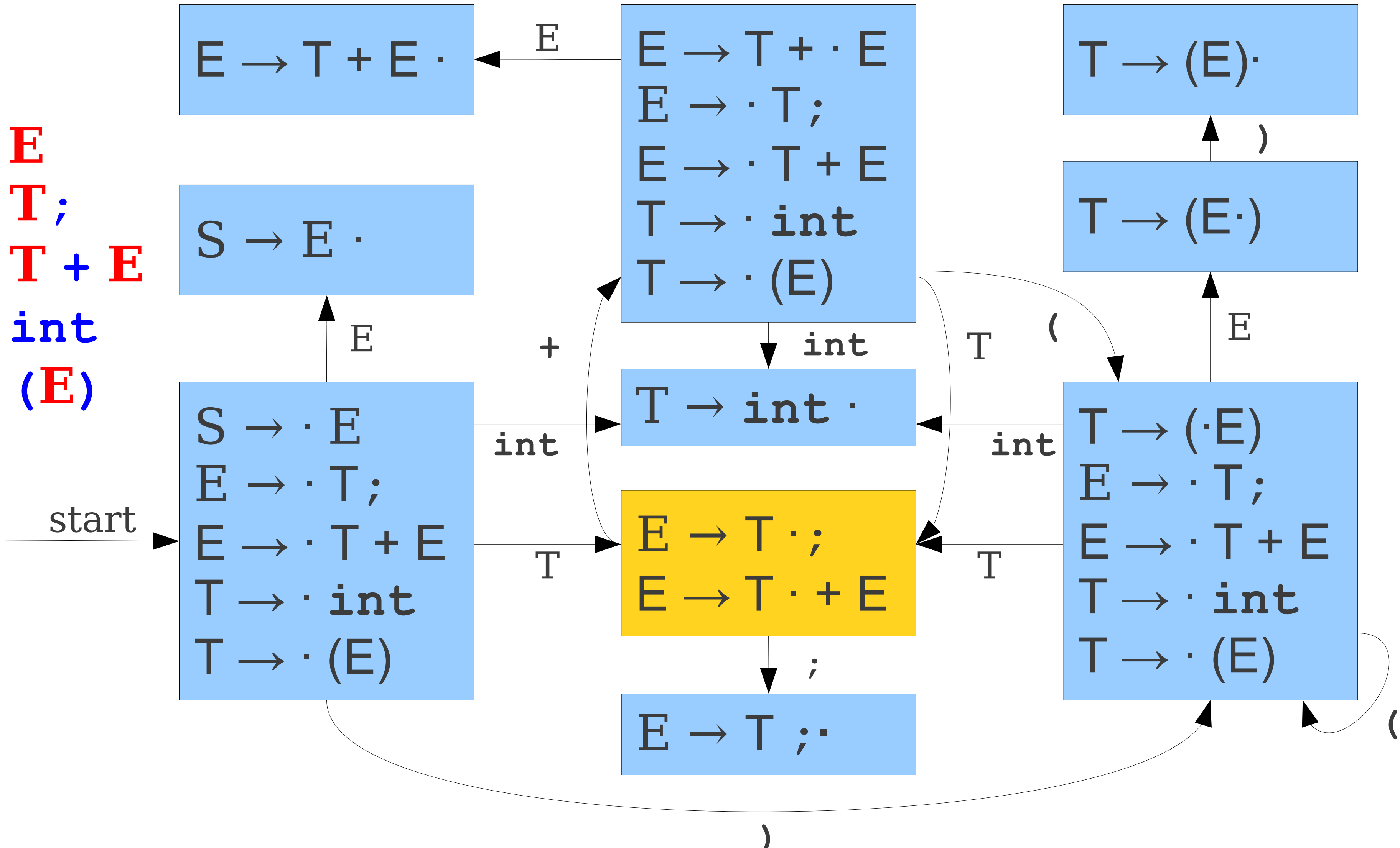


T	+	(	T

+	int	;	)	;

LR(0) Parsing

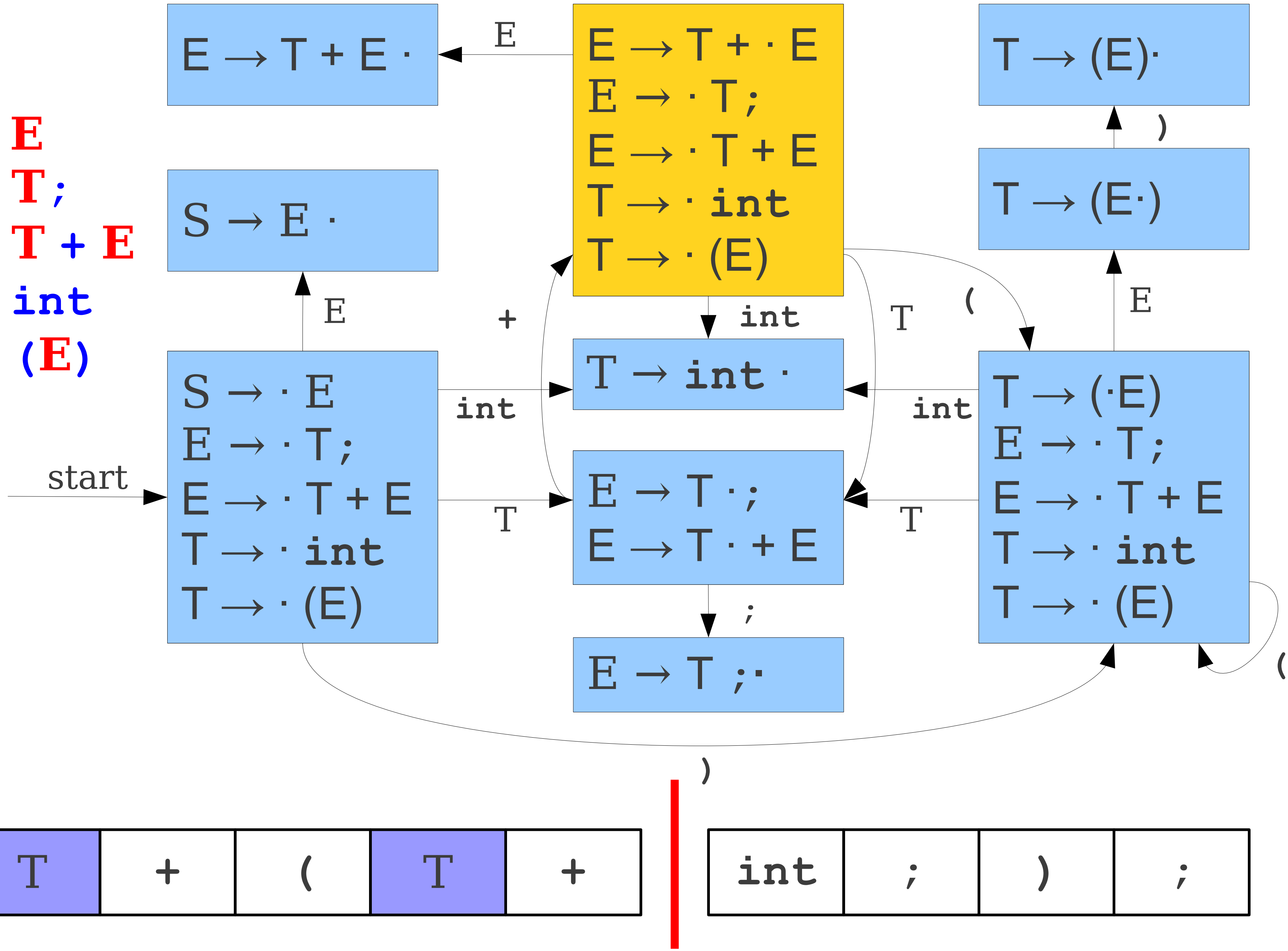
$S \to E$
$E \to T;$
$E \to T + E$
$T \to \texttt{int}$
$T \to (E)$



T	+	(	T	+	int	;	)	;

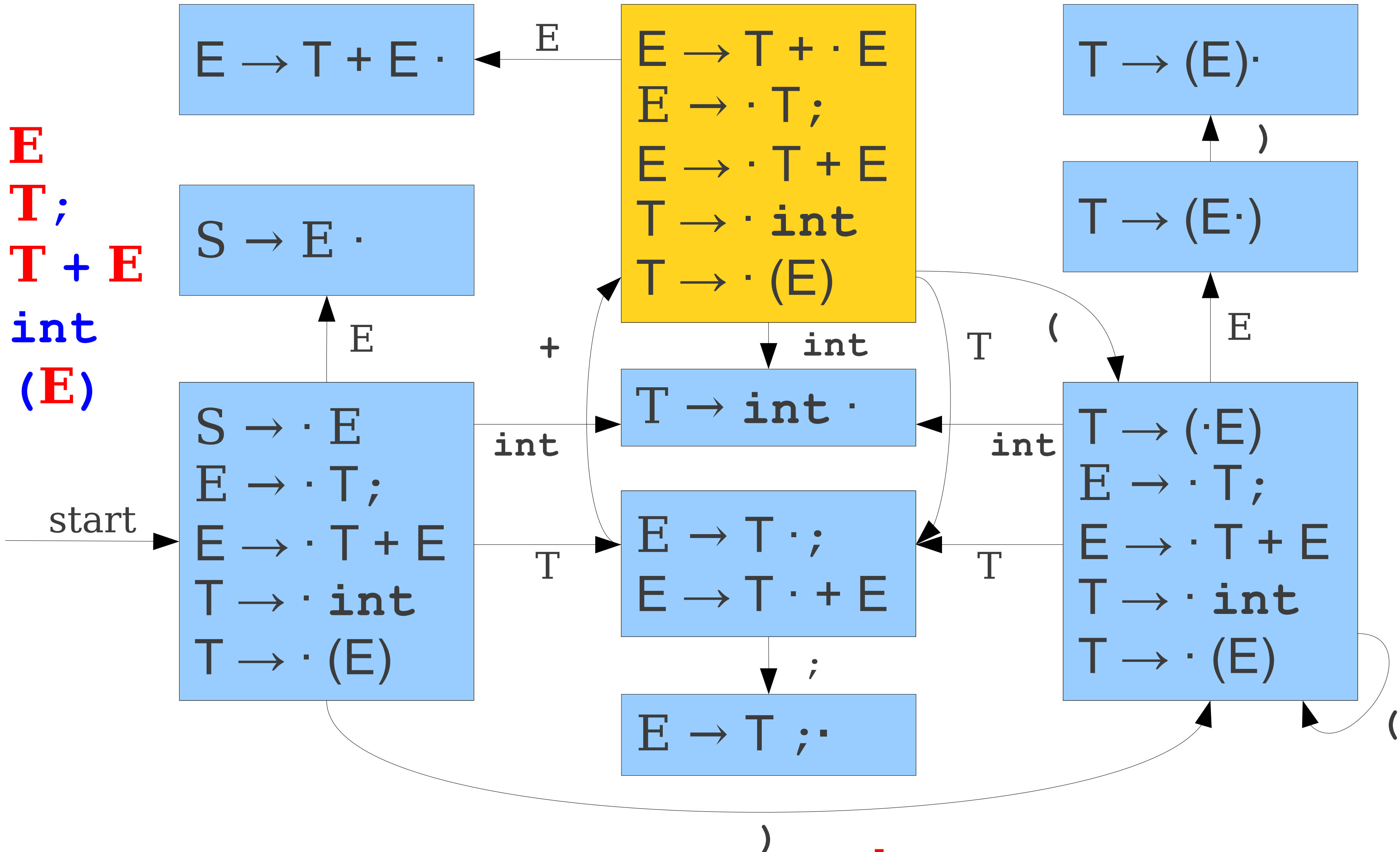
LR(0) Parsing

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



LR(0) Parsing

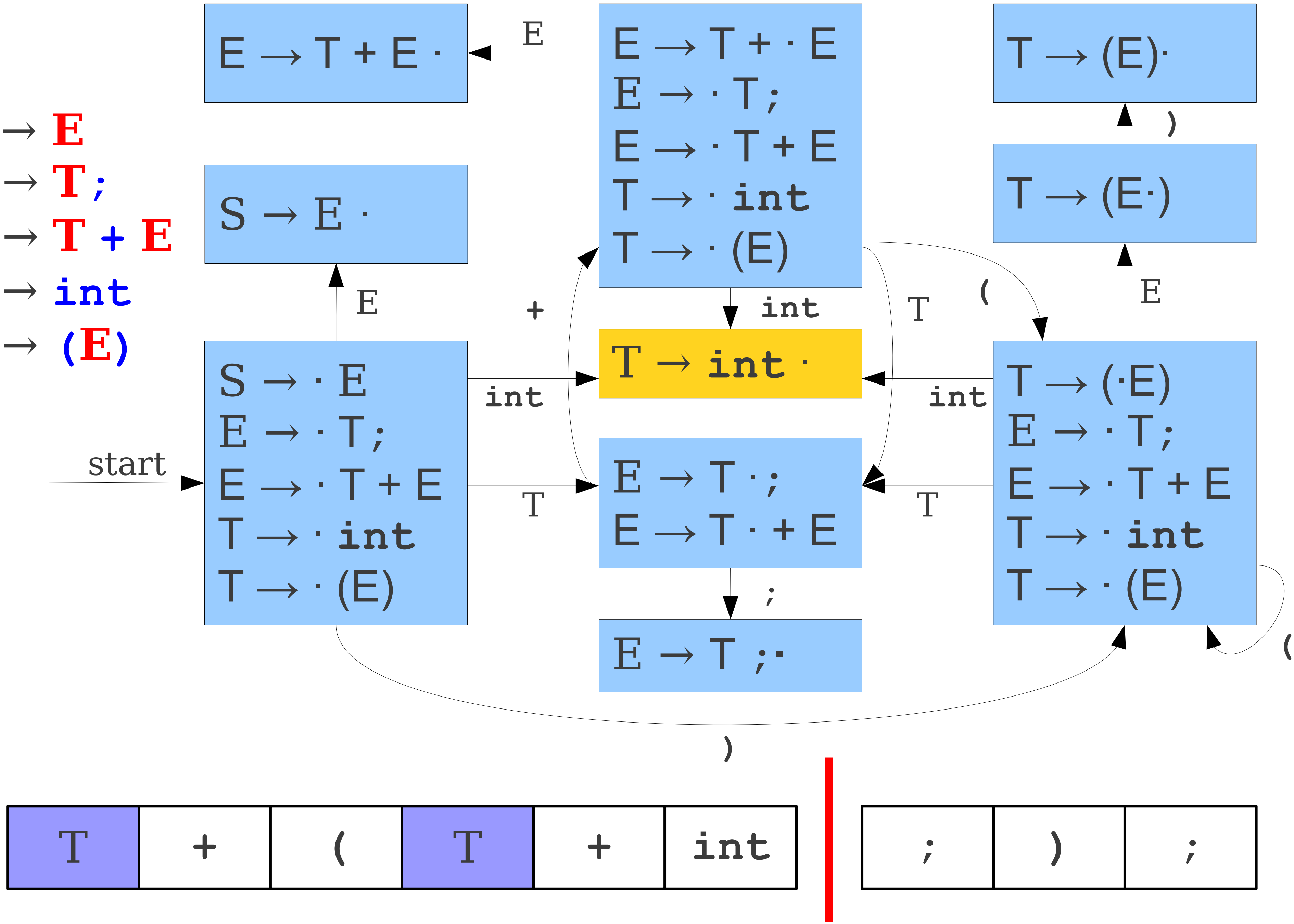
S → E
E → T;
E → T + E
T → int
T → (E)



T	+	(	T	+	int		;	)	;

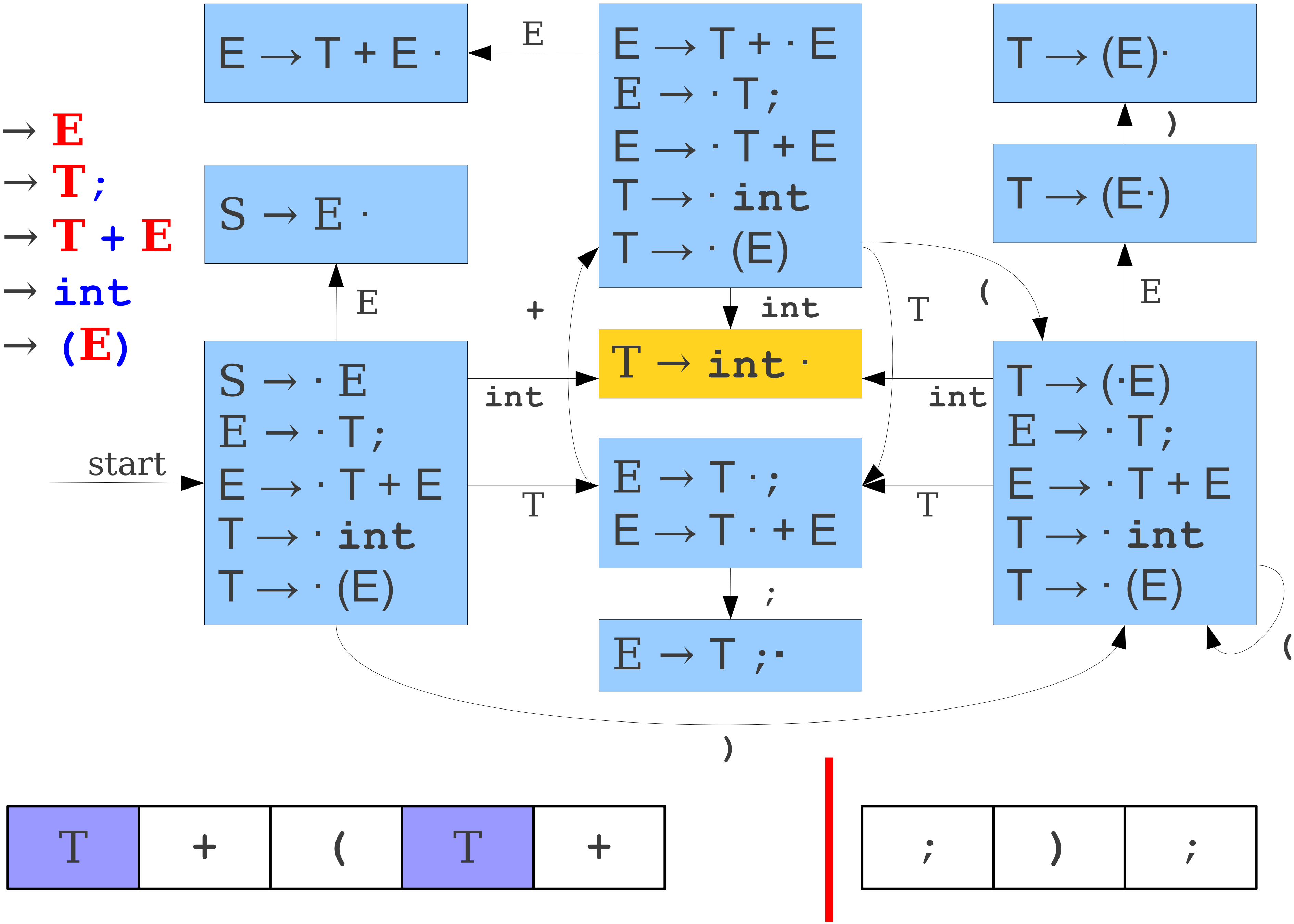
LR(0) Parsing

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



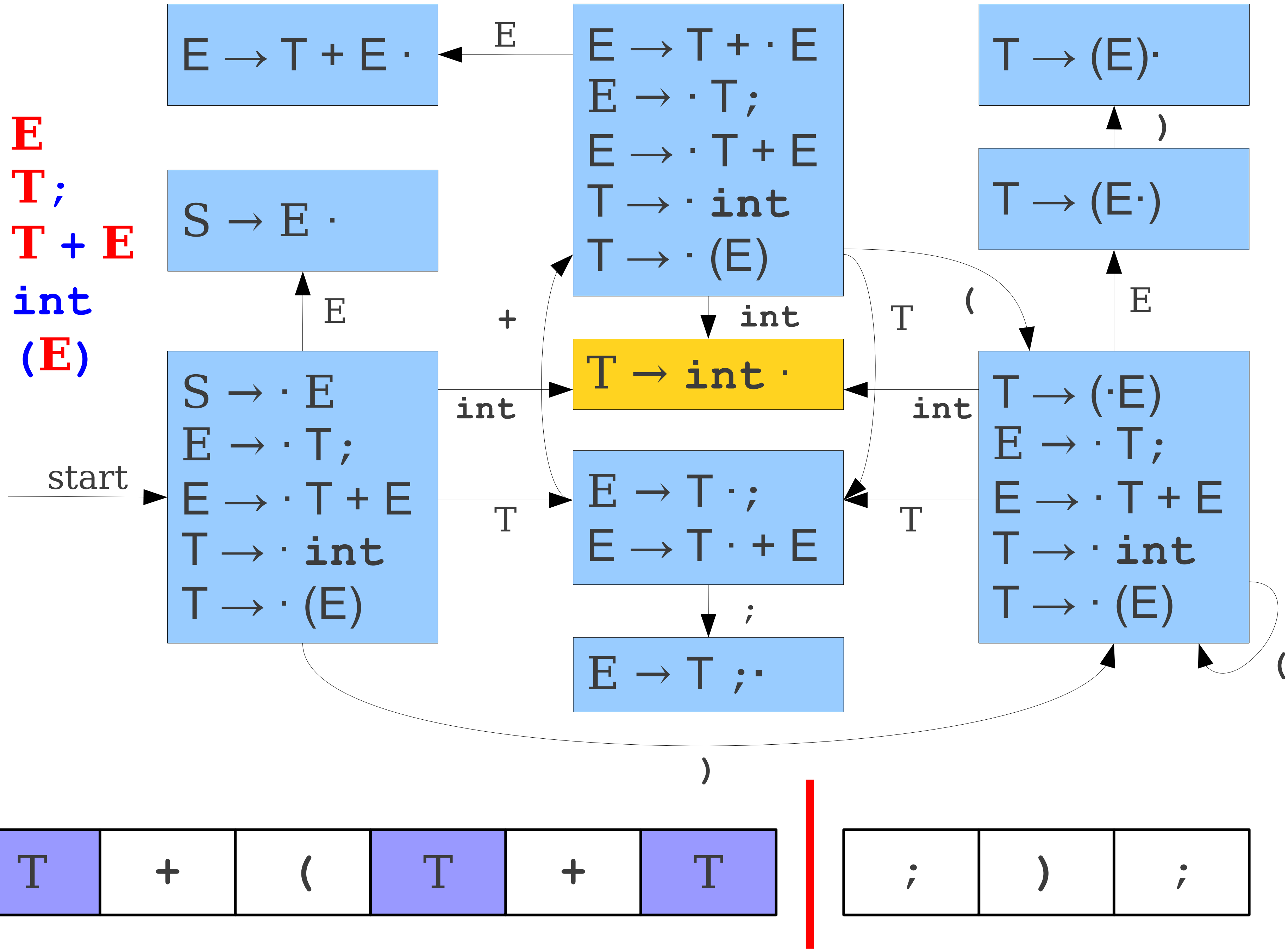
LR(0) Parsing

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



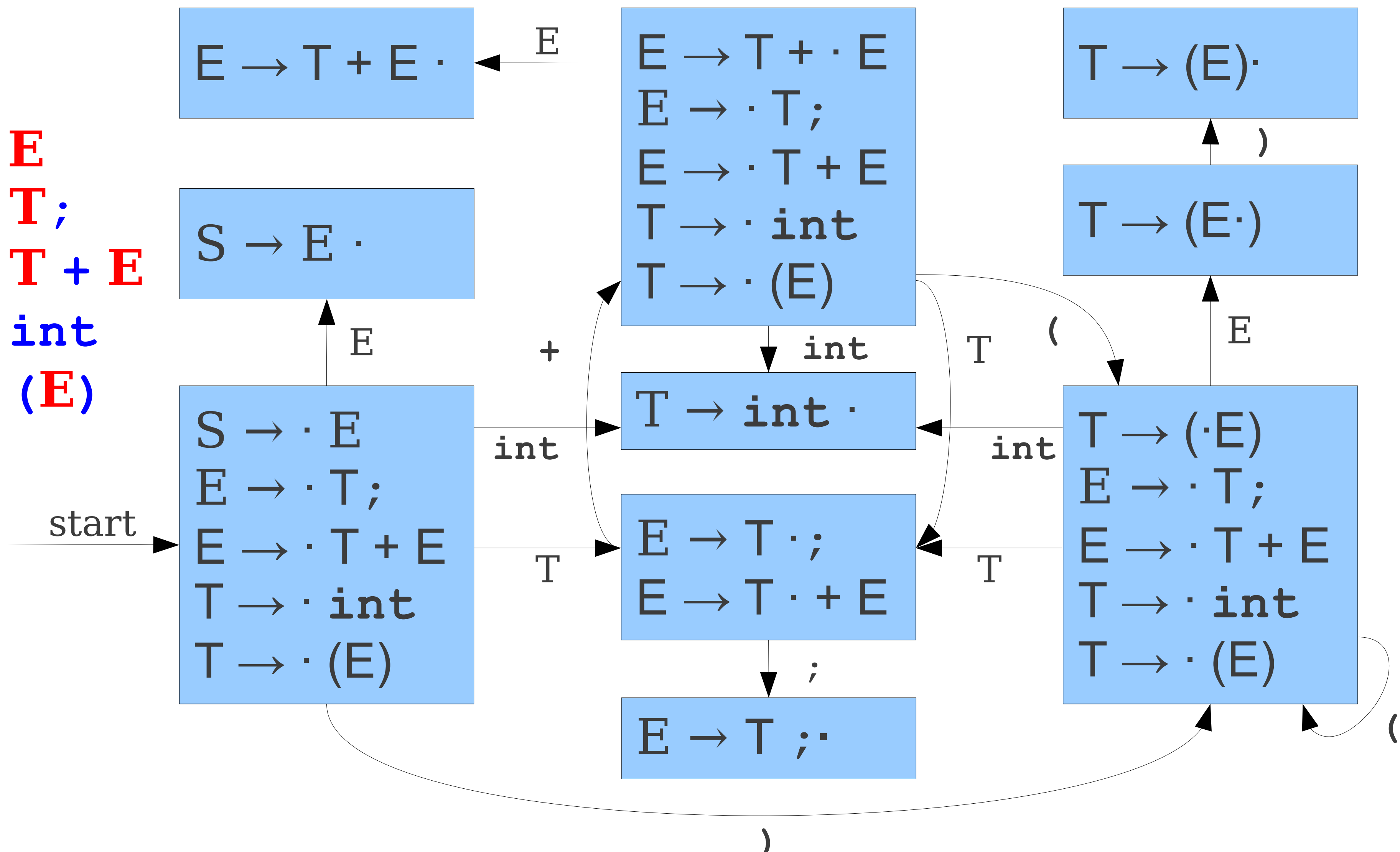
LR(0) Parsing

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



LR(0) Parsing

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



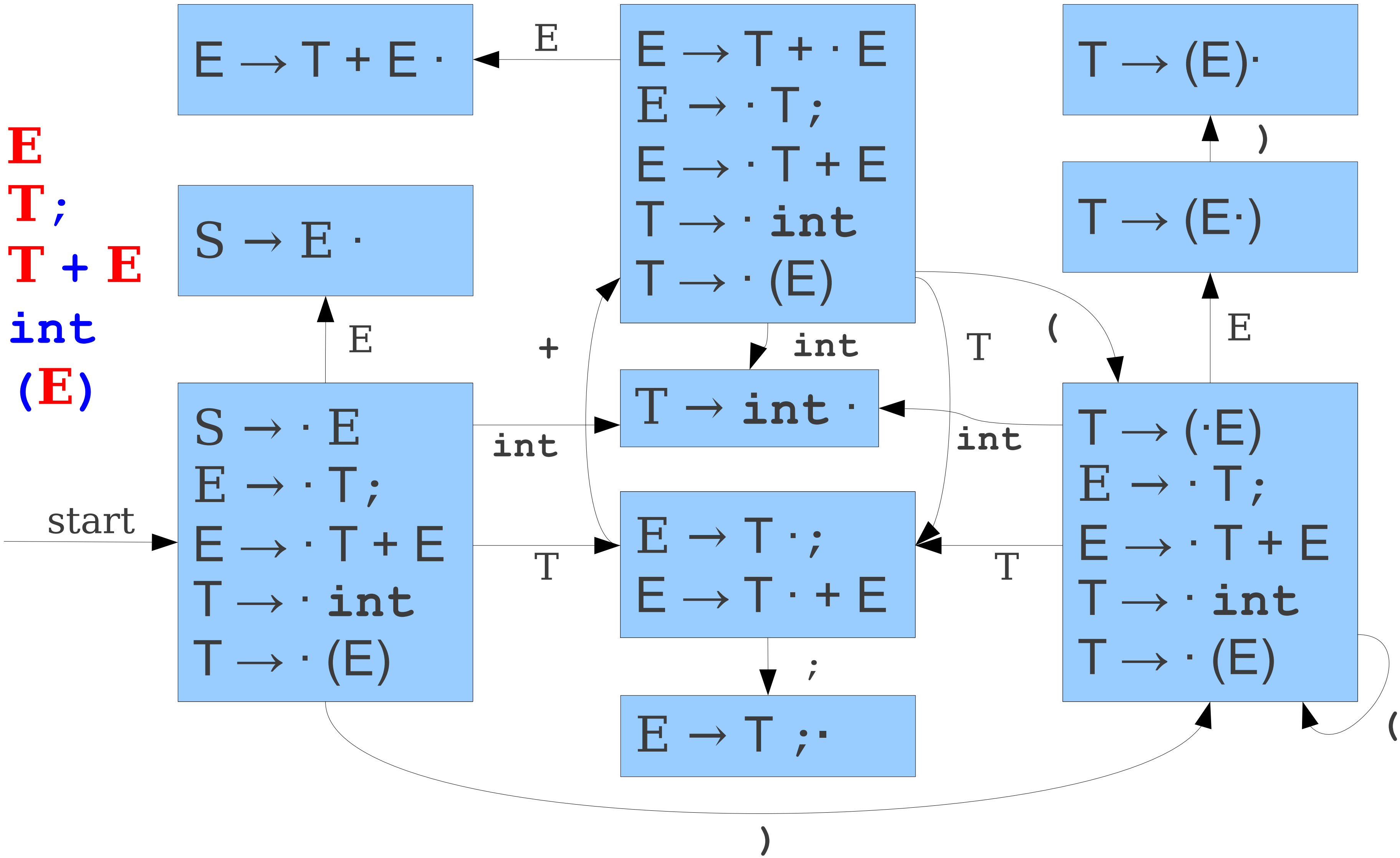
T	+	(	T	+	T	|	;	)	;

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

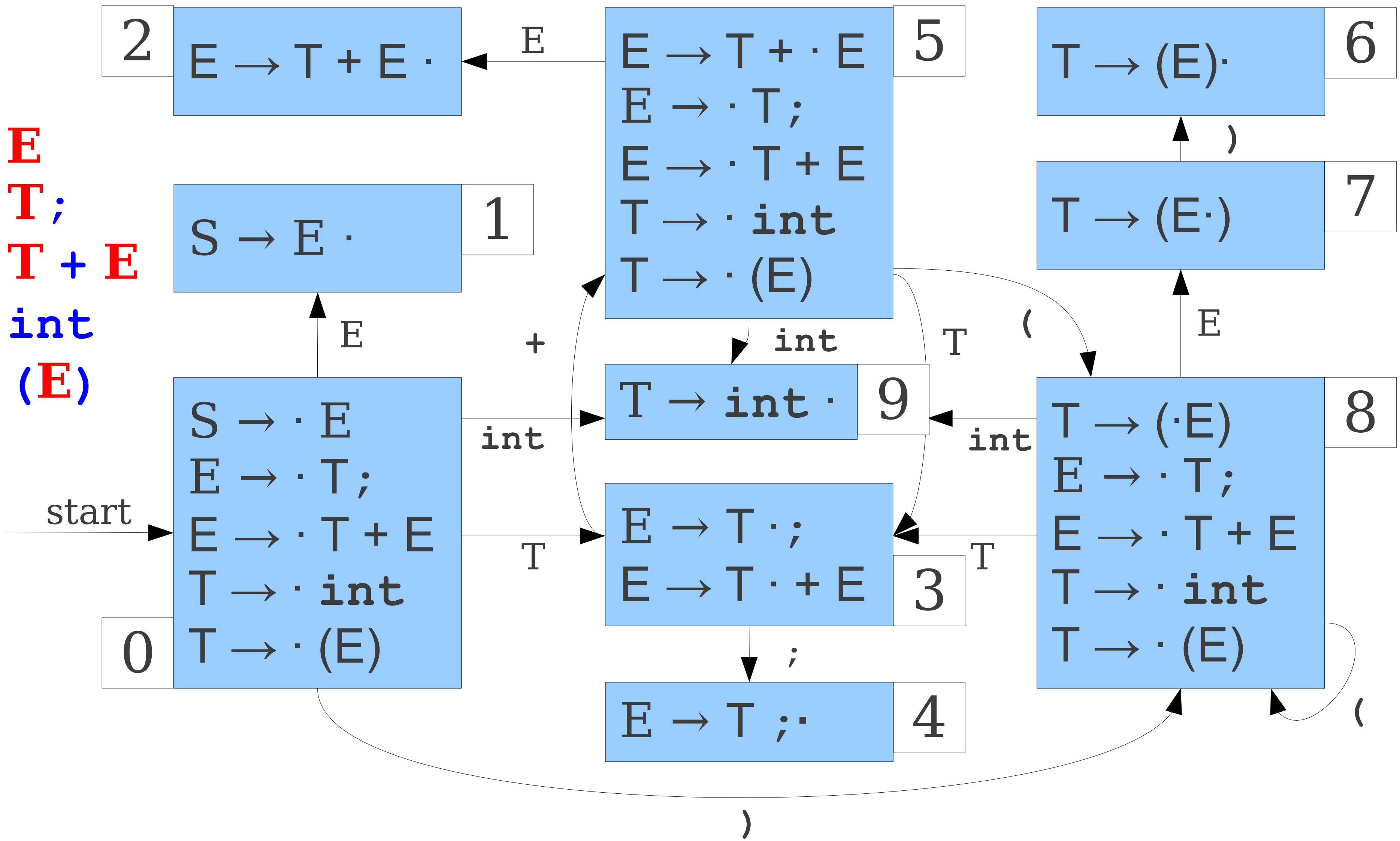
$S \rightarrow E$
$E \rightarrow T;$
$E \rightarrow T + E$
$T \rightarrow$ `int`
$T \rightarrow (E)$



int	+	(	int	+	int	;	)	;

LR(0) Parsing

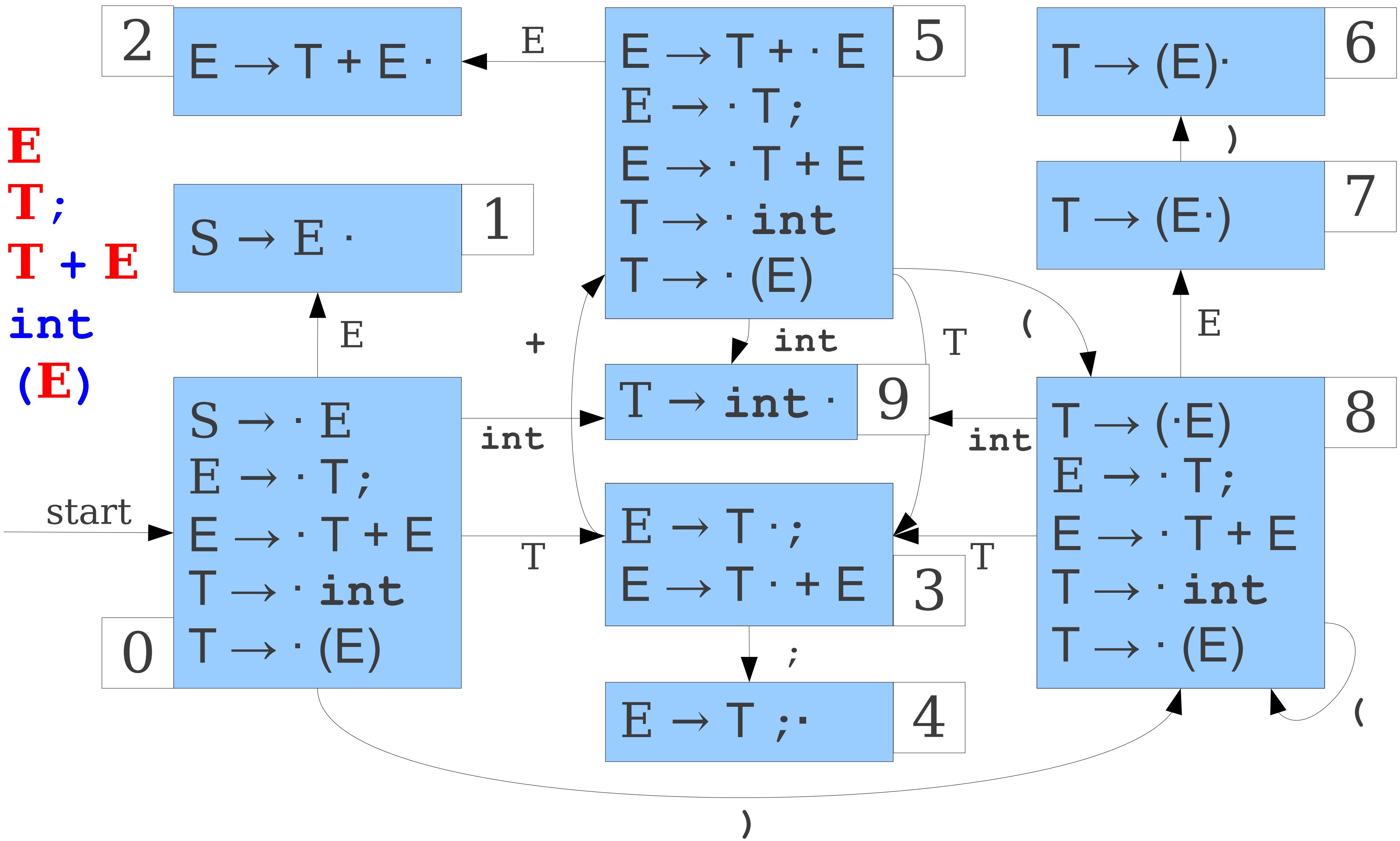
S → E
E → T;
E → T + E
T → int
T → (E)



int	+	(	int	+	int	;	)	;

LR(0) Parsing

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

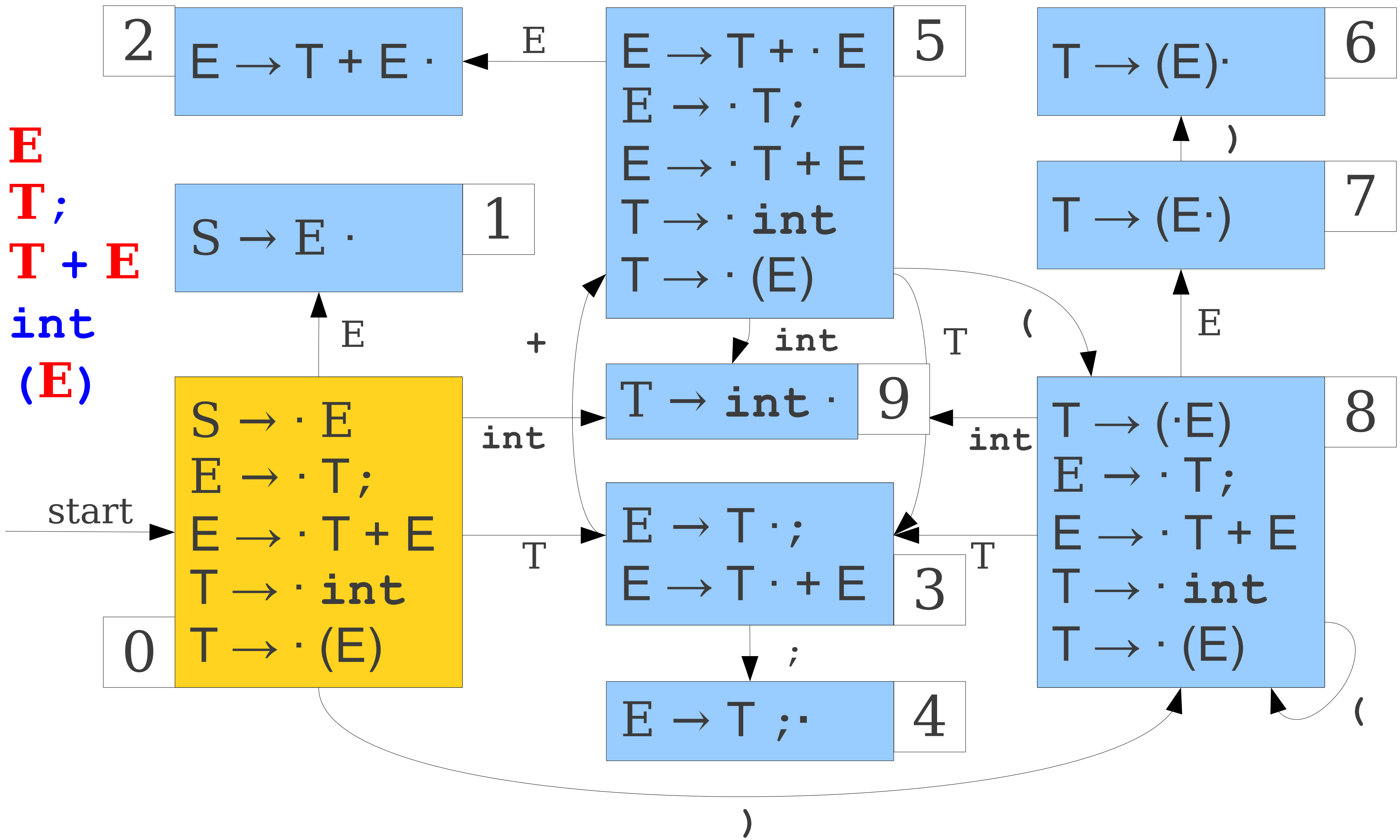


int	+	(	int	+	int	;	)	;

0

LR(0) Parsing

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

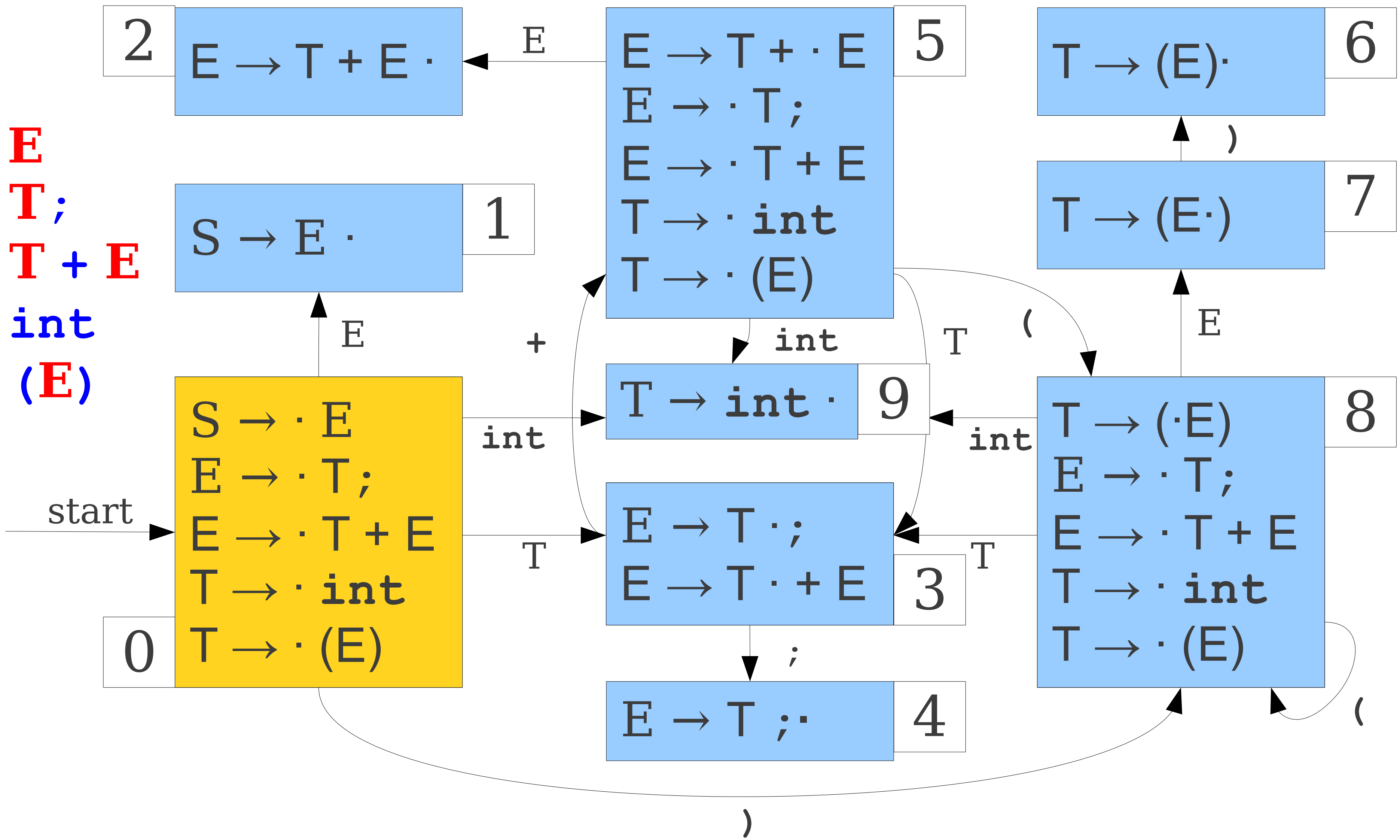


int	+	(	int	+	int	;	)	;

0

LR(0) Parsing

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

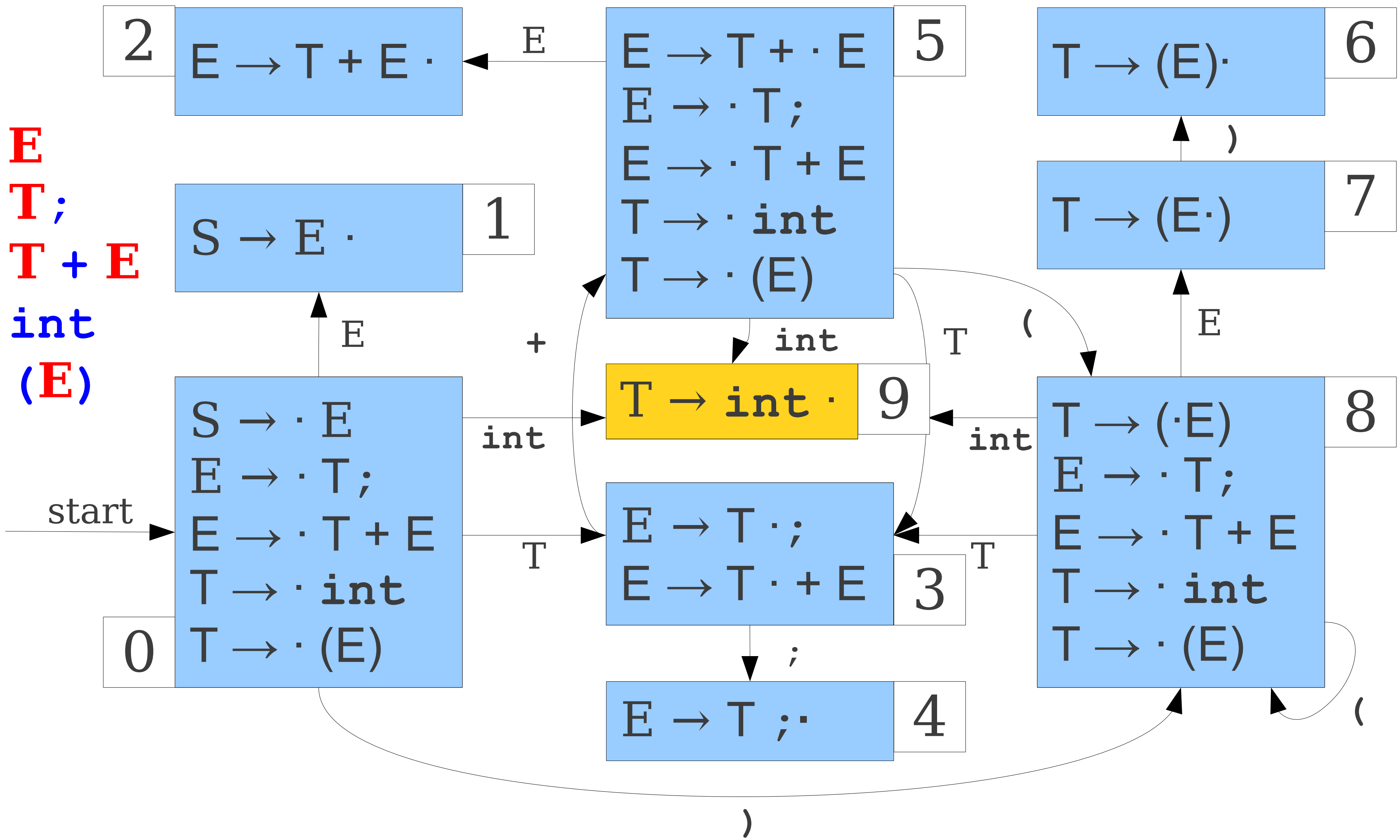


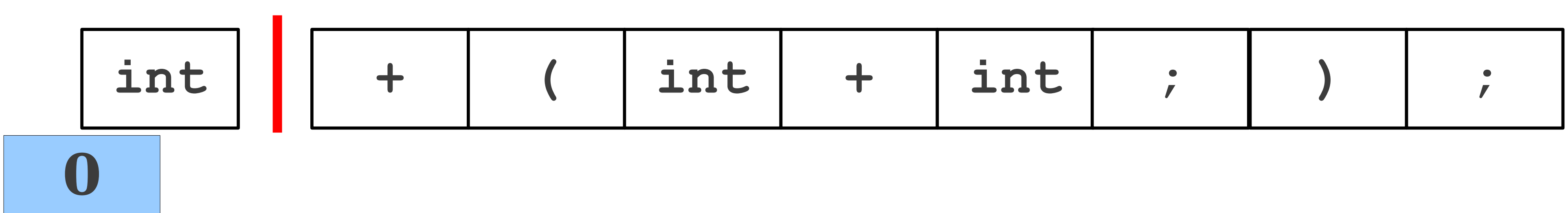
int		+	(	int	+	int	;	)	;

0

LR(0) Parsing

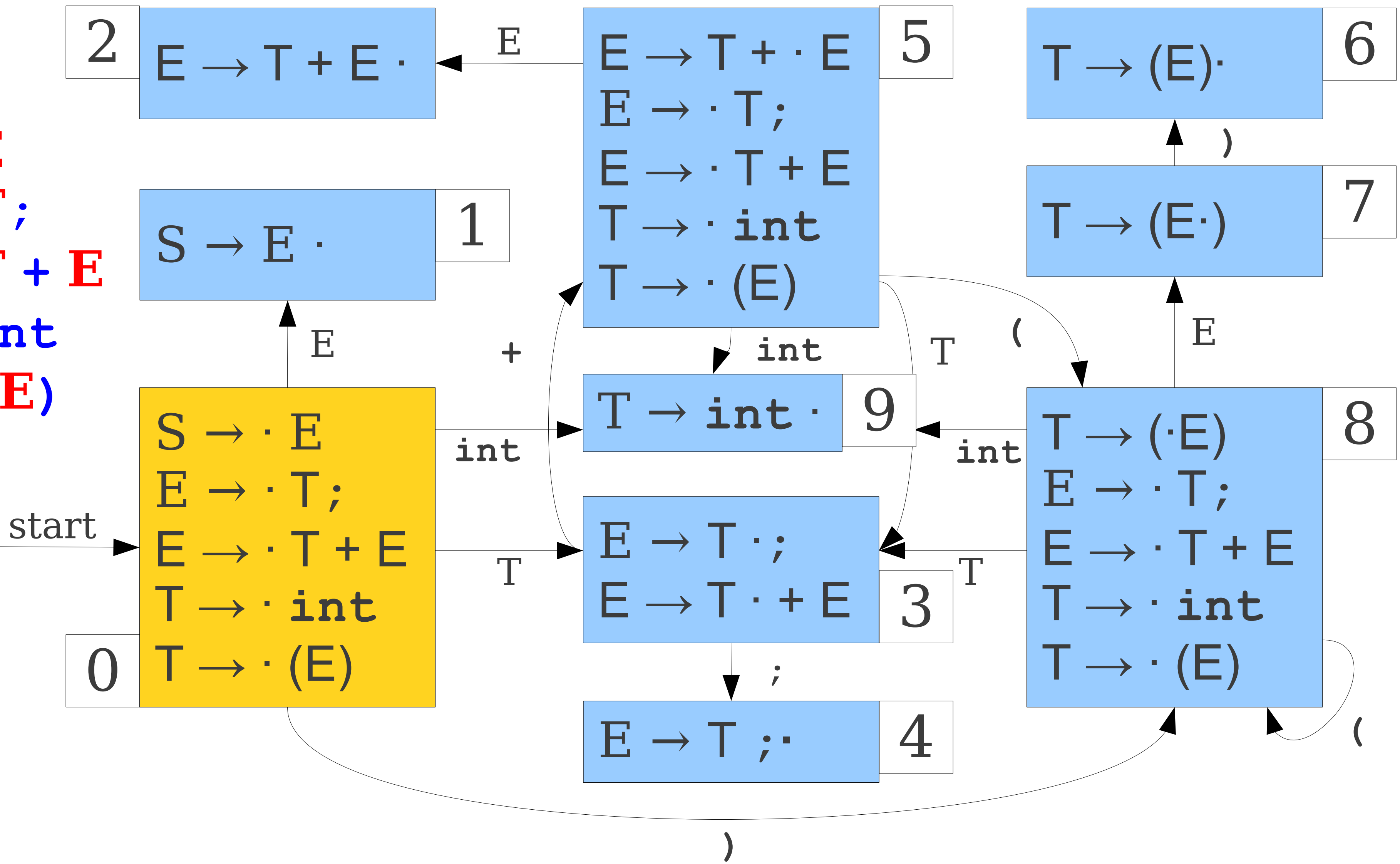
$S \rightarrow E$
$E \rightarrow T;$
$E \rightarrow T + E$
$T \rightarrow$ `int`
$T \rightarrow (E)$





LR(0) Parsing

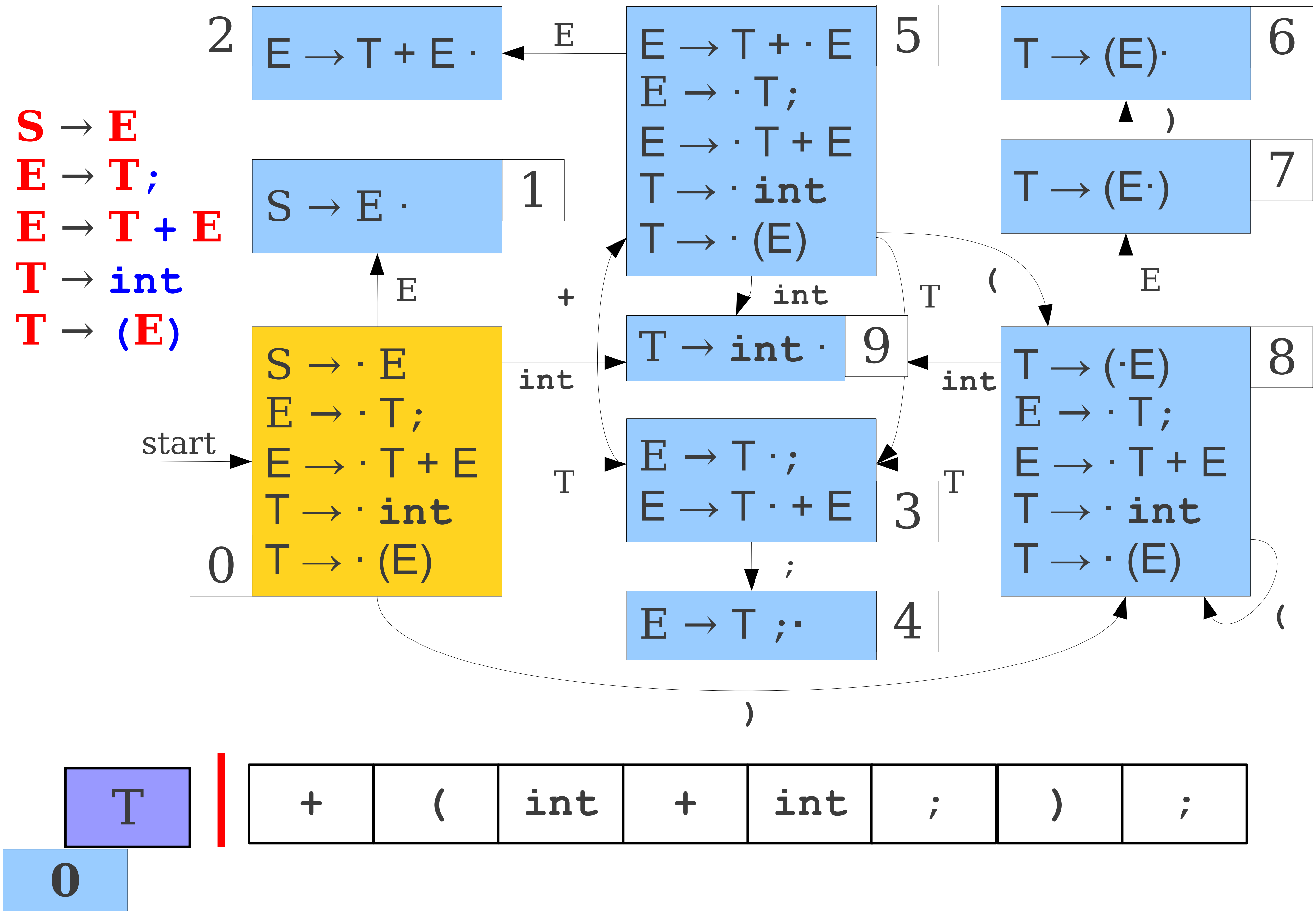
S → E
E → T;
E → T + E
T → int
T → (E)



+	(	int	+	int	;	)	;

0

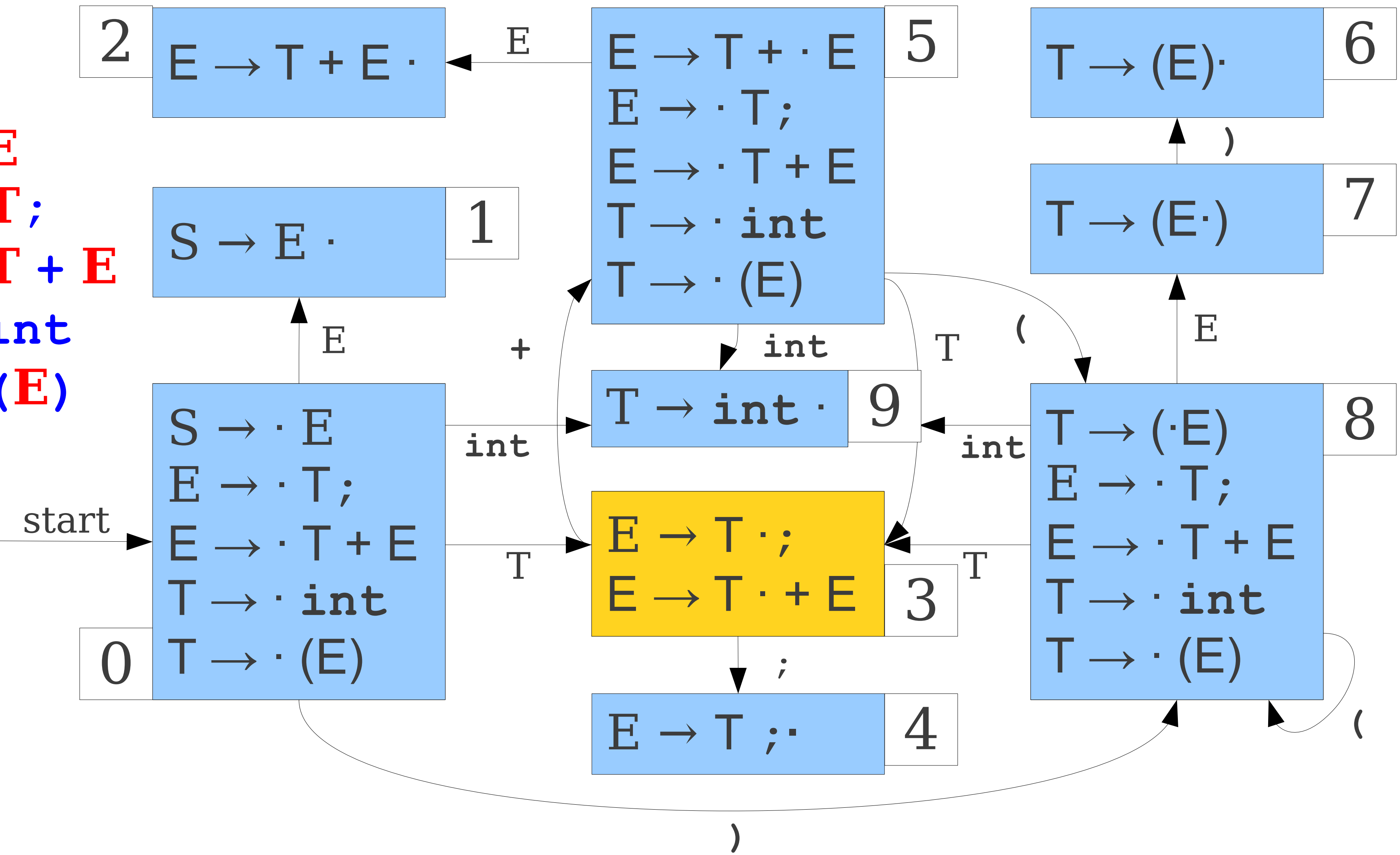
LR(0) Parsing

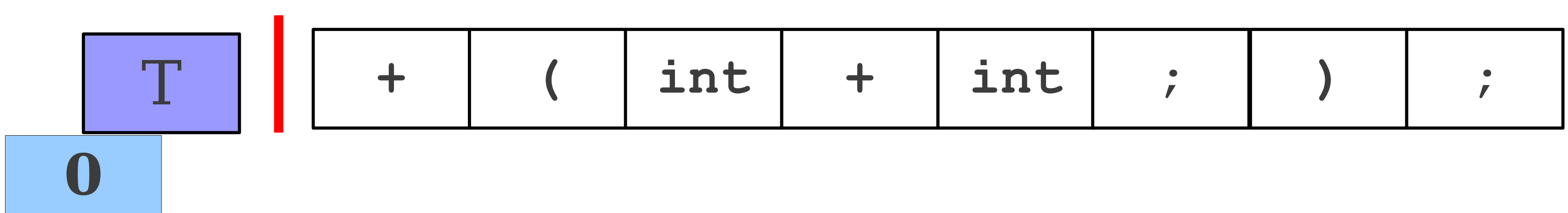


S → E
E → T;
E → T + E
T → int
T → (E)
start
0
S → · E
E → · T;
E → · T + E
T → · int
T → · (E)
1
S → E ·
2
E → T + E ·
3
E → T · ;
E → T · + E
4
E → T ;·
5
E → T + · E
E → · T;
E → · T + E
T → · int
T → · (E)
6
T → (E)·
7
T → (E·)
8
T → (·E)
E → · T;
E → · T + E
T → · int
T → · (E)
9
T → int ·
E
int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

LR(0) Parsing

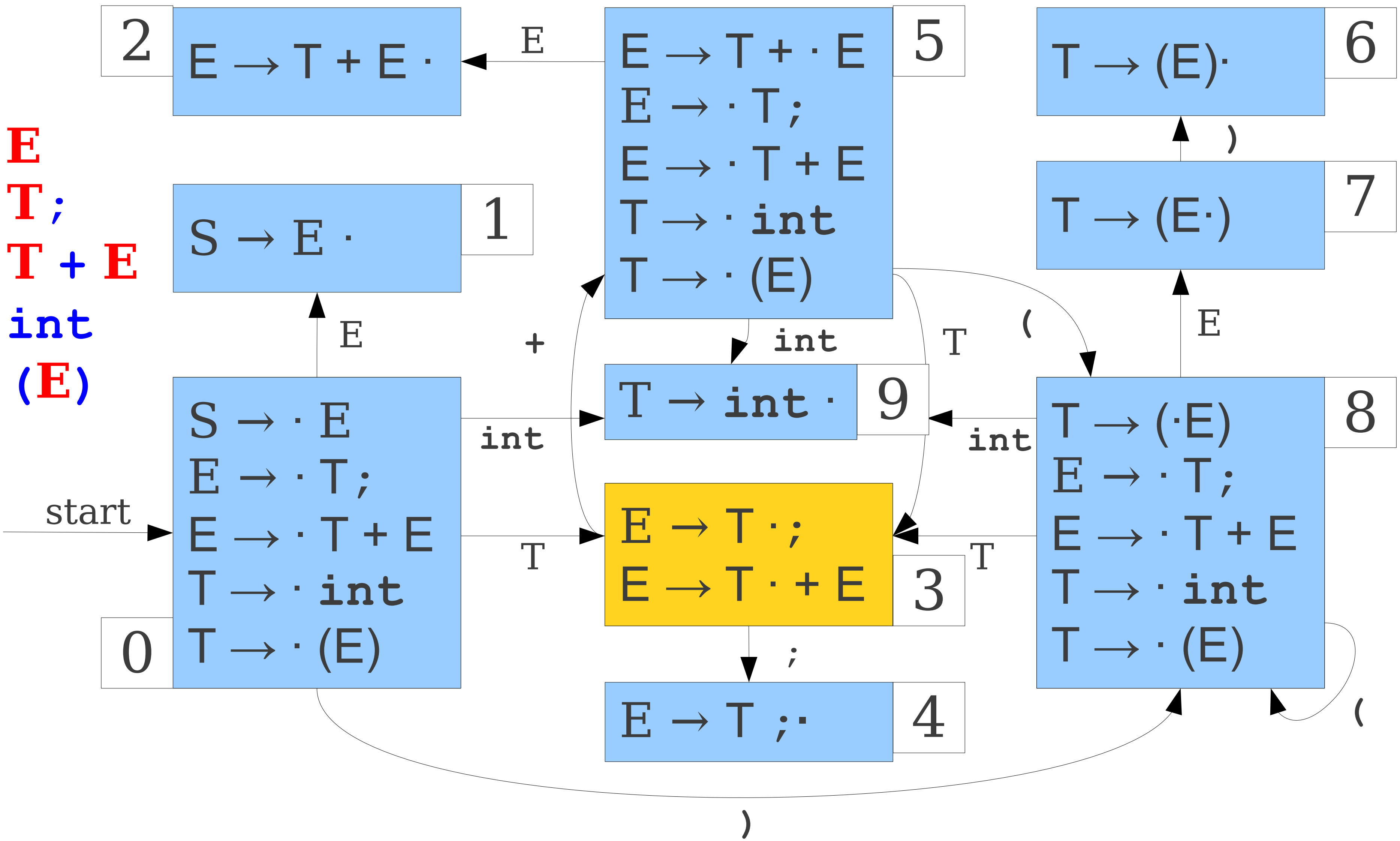
S → **E**
E → **T ;**
E → **T + E**
T → `int`
T → **(E)**

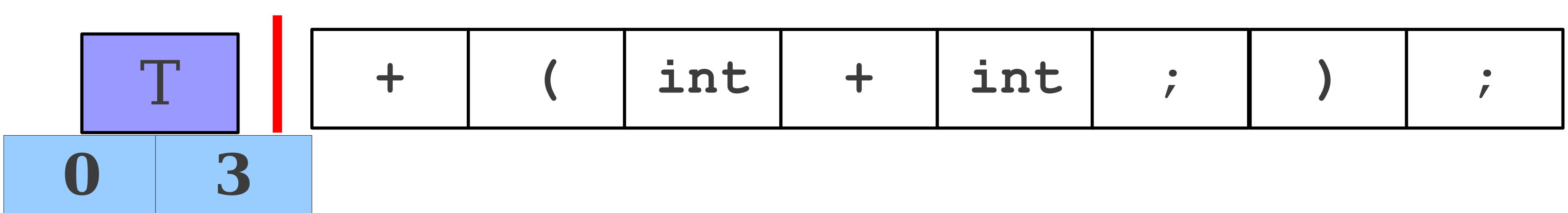




LR(0) Parsing

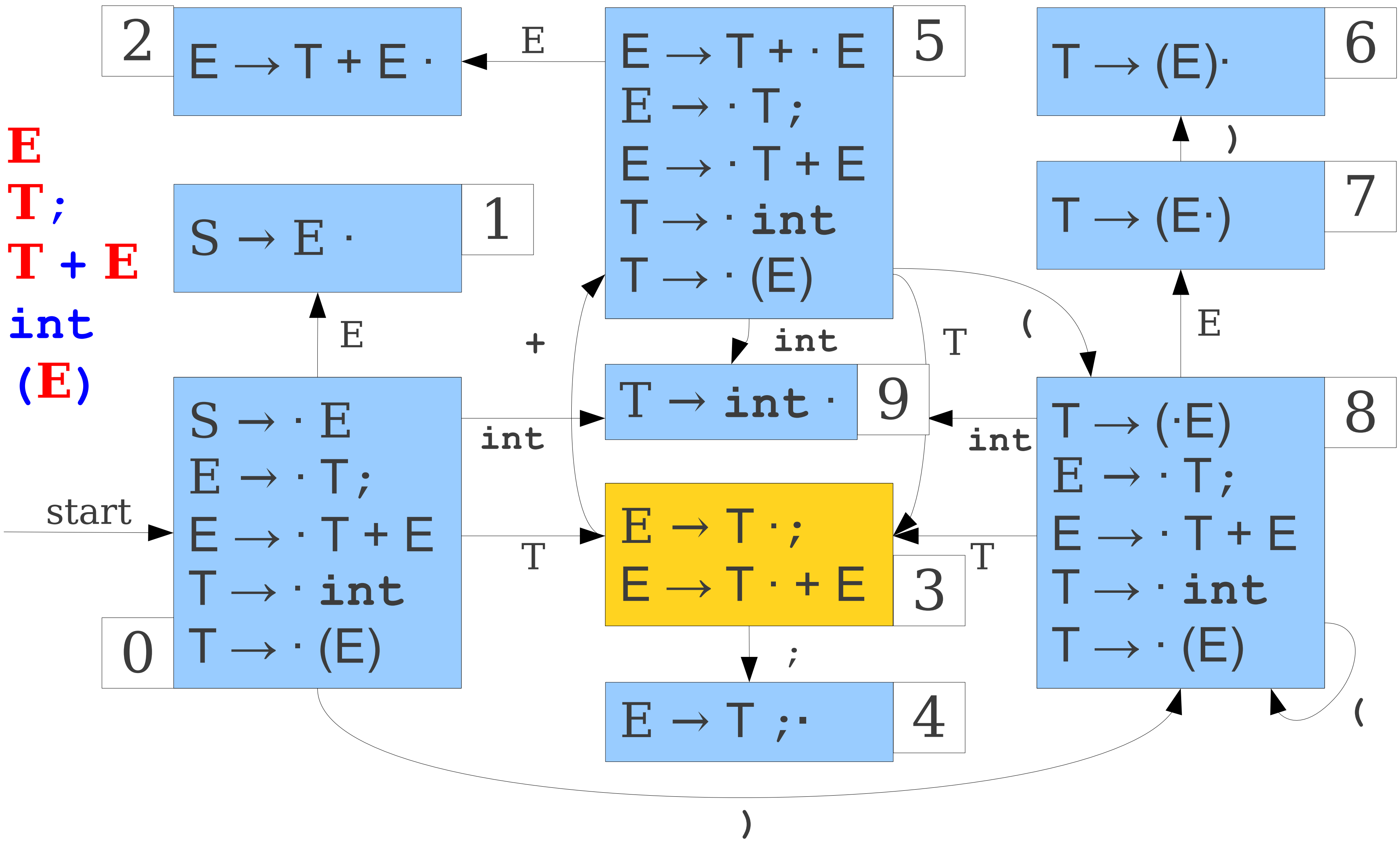
S → **E**
E → **T** `;`
E → **T** + **E**
T → `int`
T → `(E)`





LR(0) Parsing

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

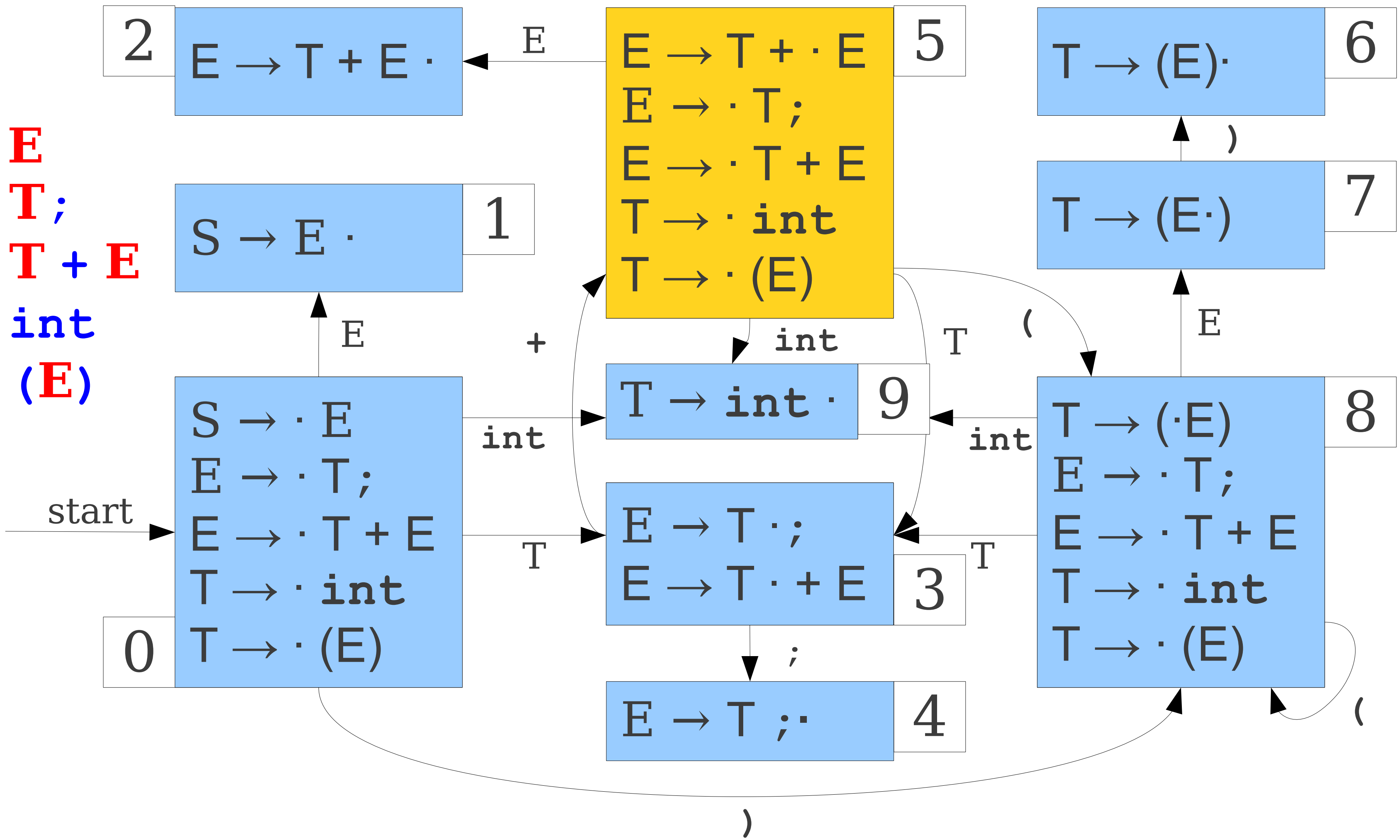


T	+
0	3

(	int	+	int	;	)	;

LR(0) Parsing

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

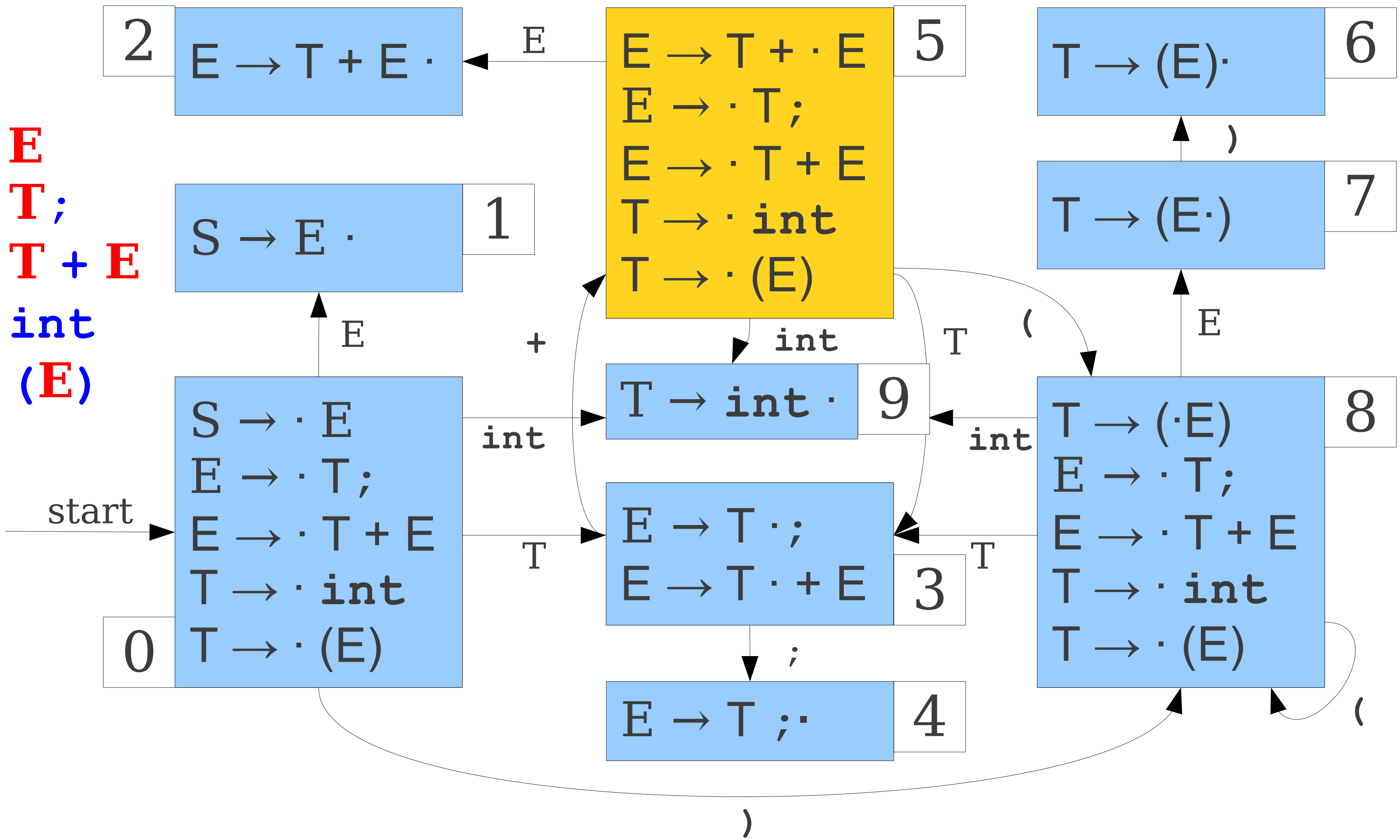


T	+
0	3

(	int	+	int	;	)	;

LR(0) Parsing

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

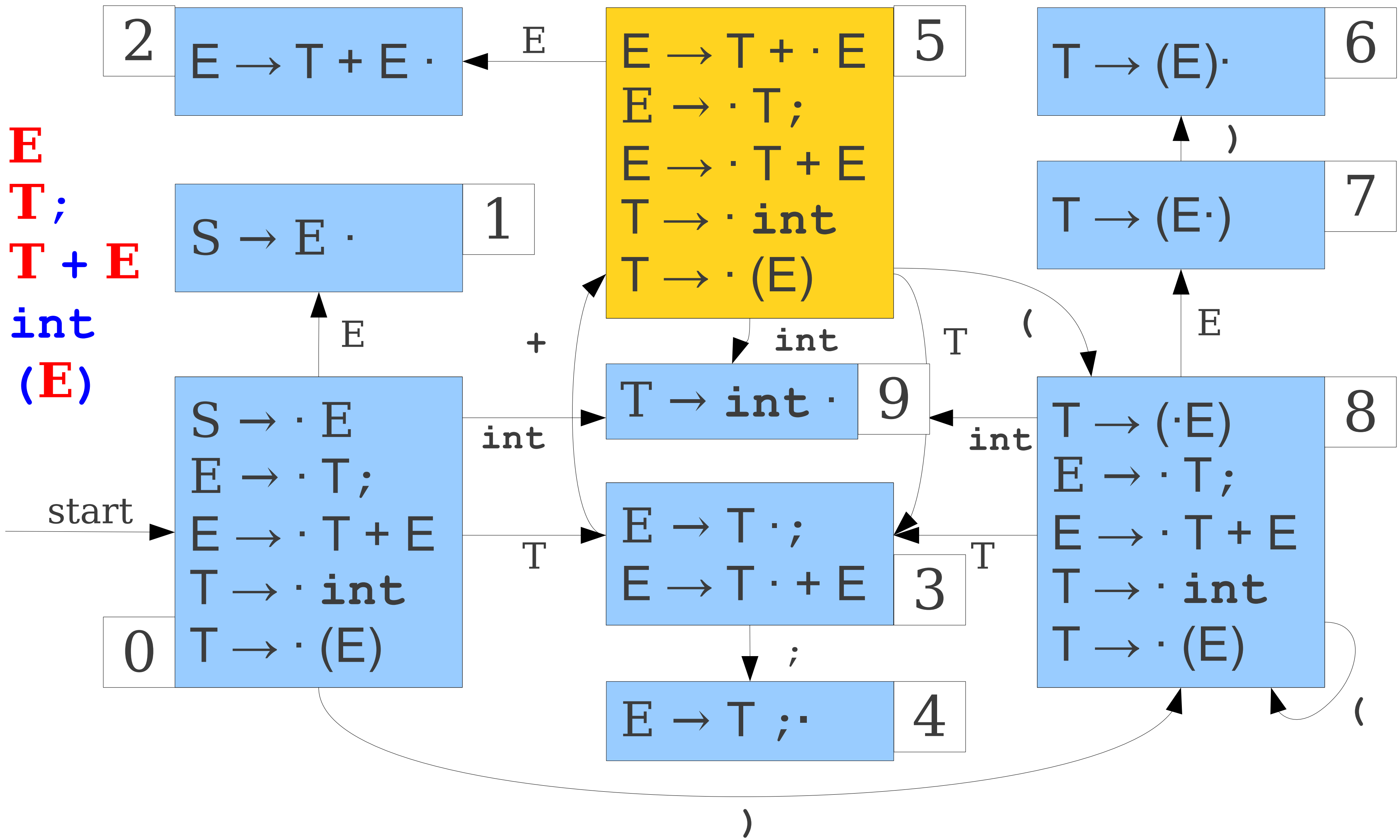


T	+		
0	3	5	

(	int	+	int	;	)	;

LR(0) Parsing

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

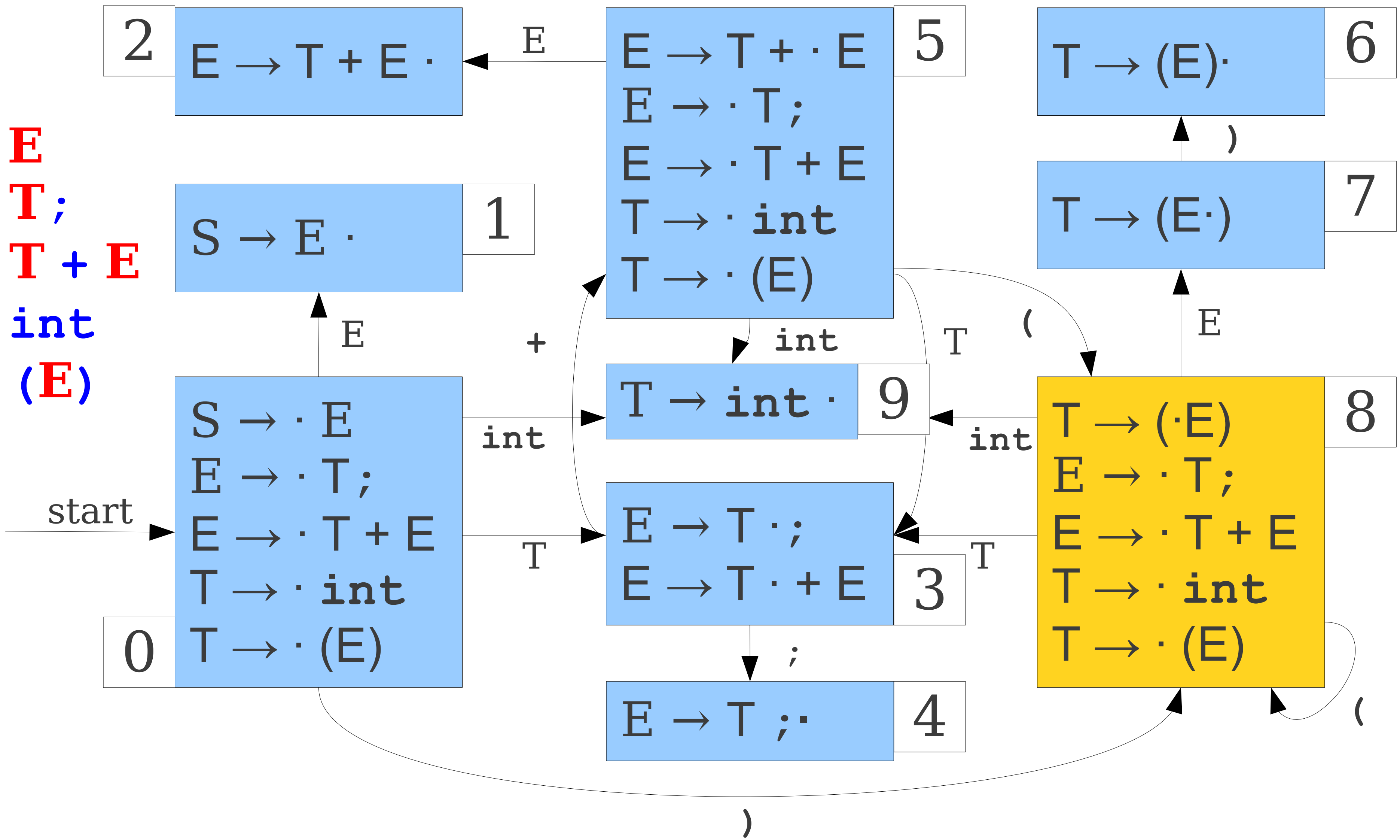


T	+	(
0	3	5

int	+	int	;	)	;

LR(0) Parsing

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

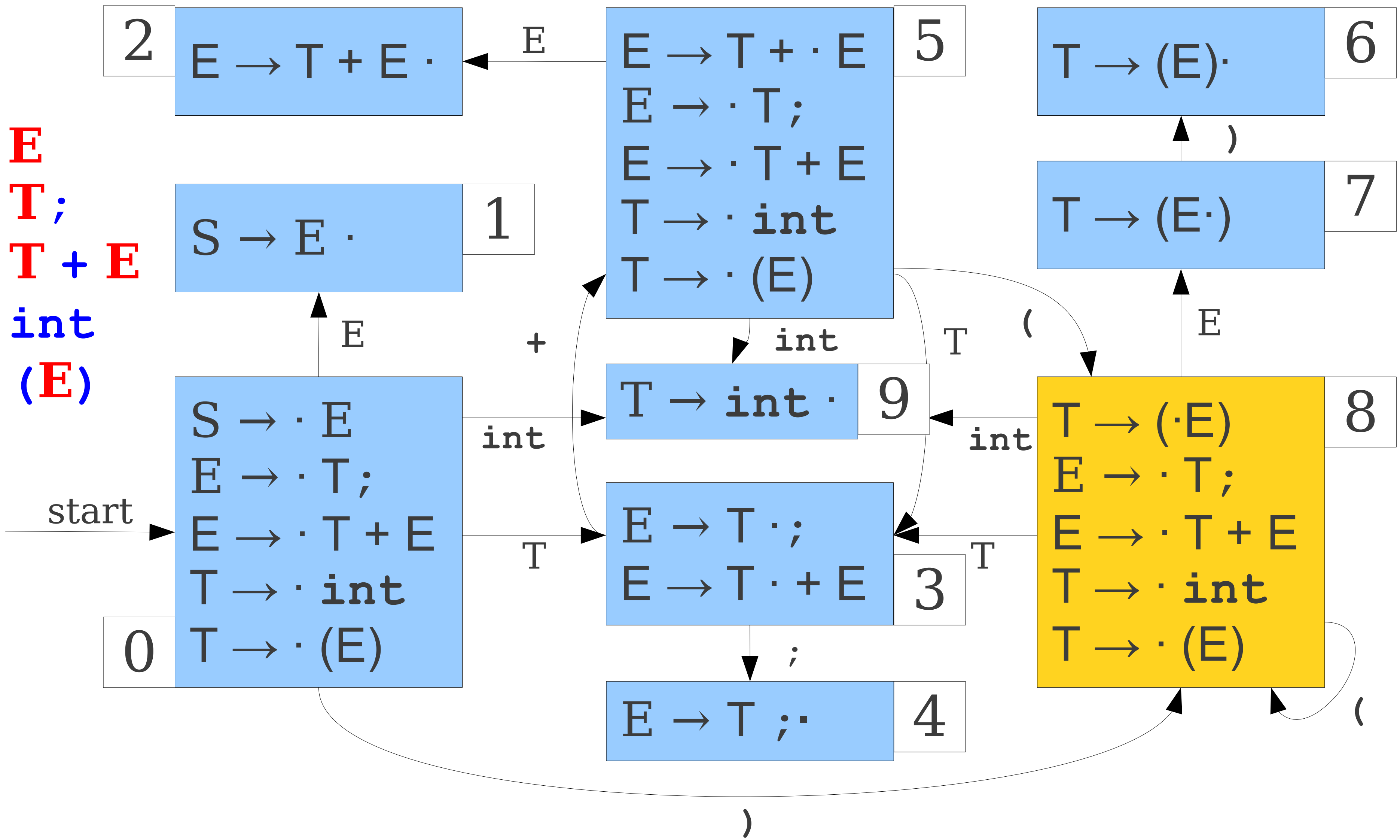


T	+	(
0	3	5

int	+	int	;	)	;

LR(0) Parsing

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

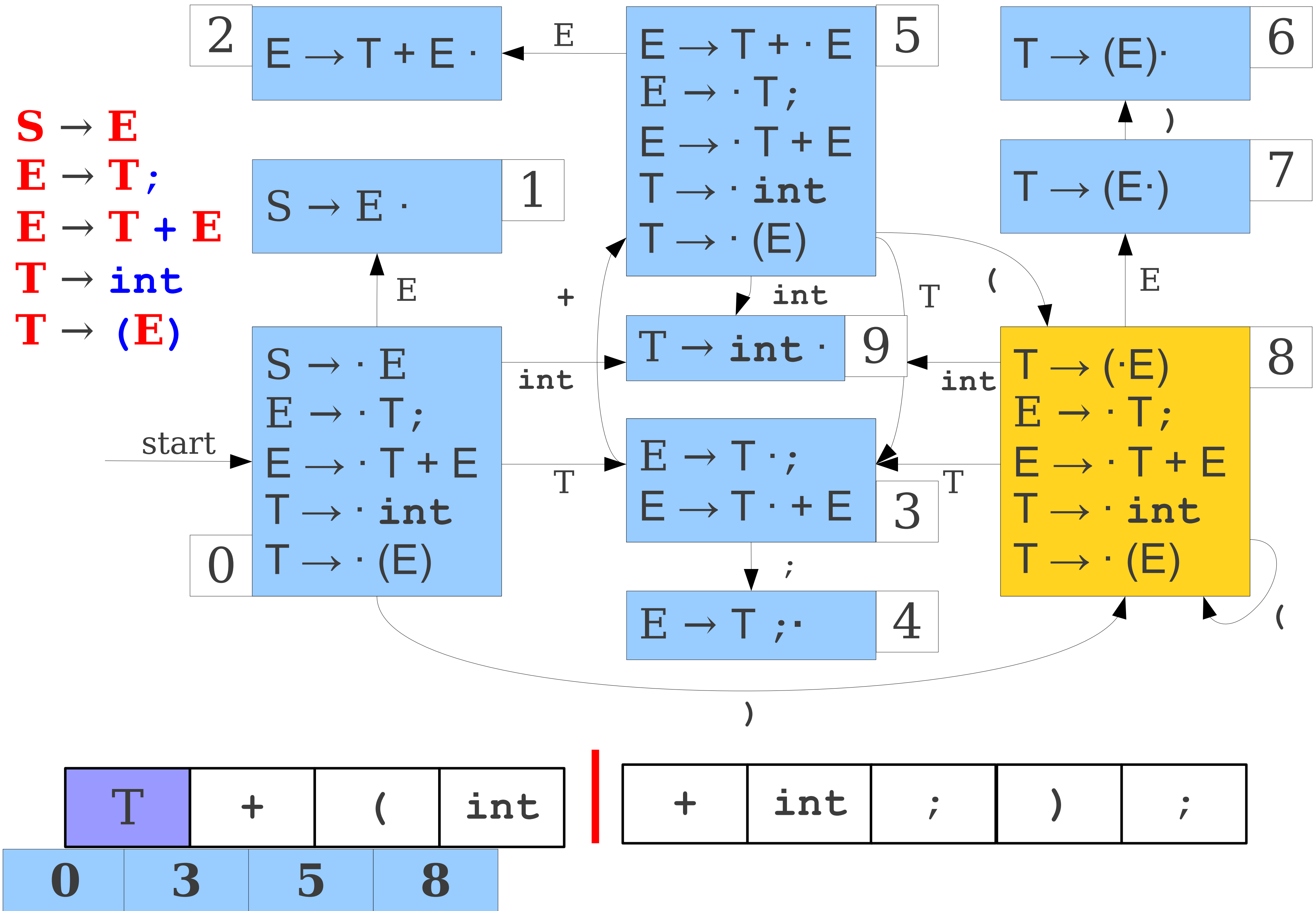


T	+	(

int	+	int	;	)	;

0	3	5	8

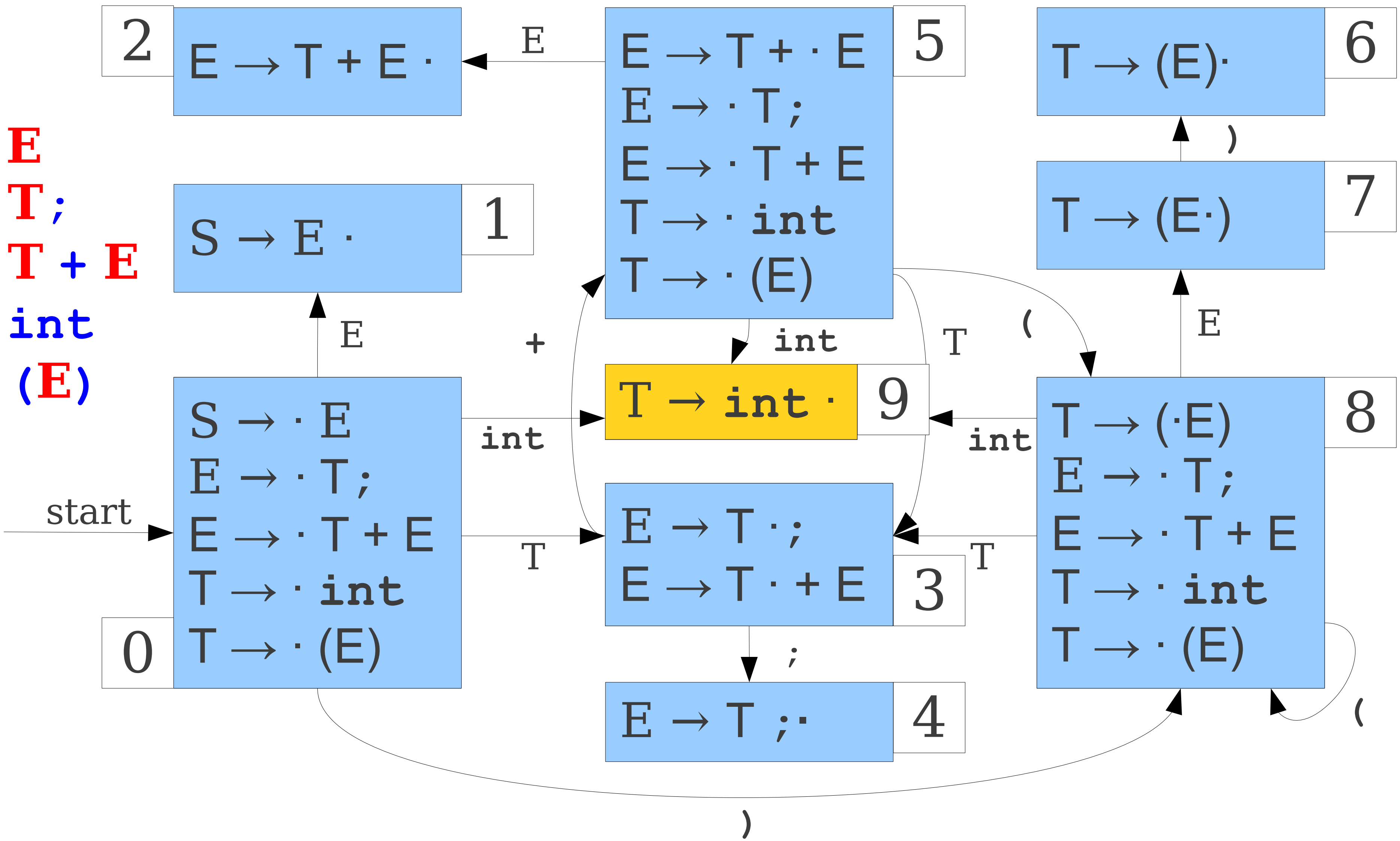
LR(0) Parsing



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

LR(0) Parsing

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

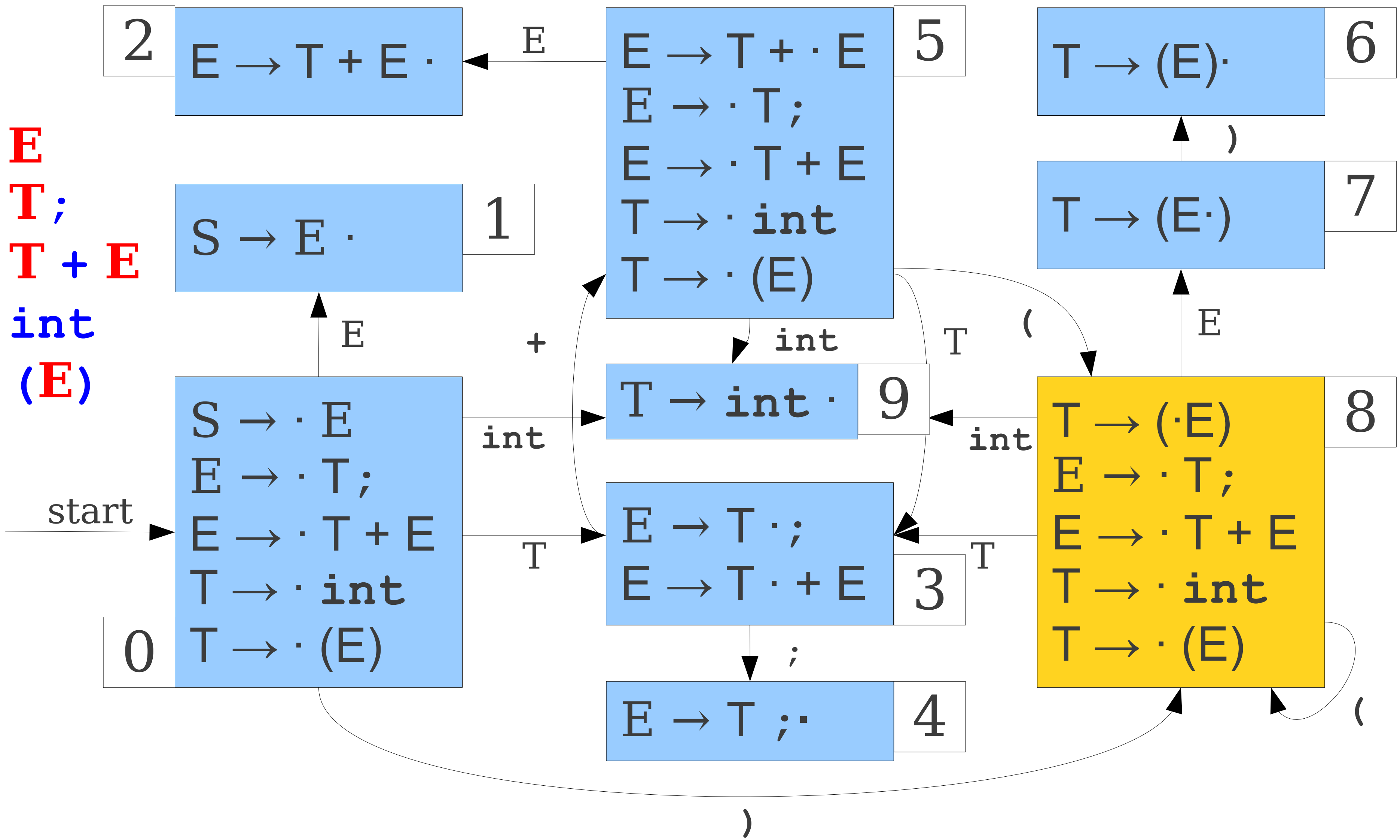


T	+	(	int
0	3	5	8

+	int	;	)	;

LR(0) Parsing

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

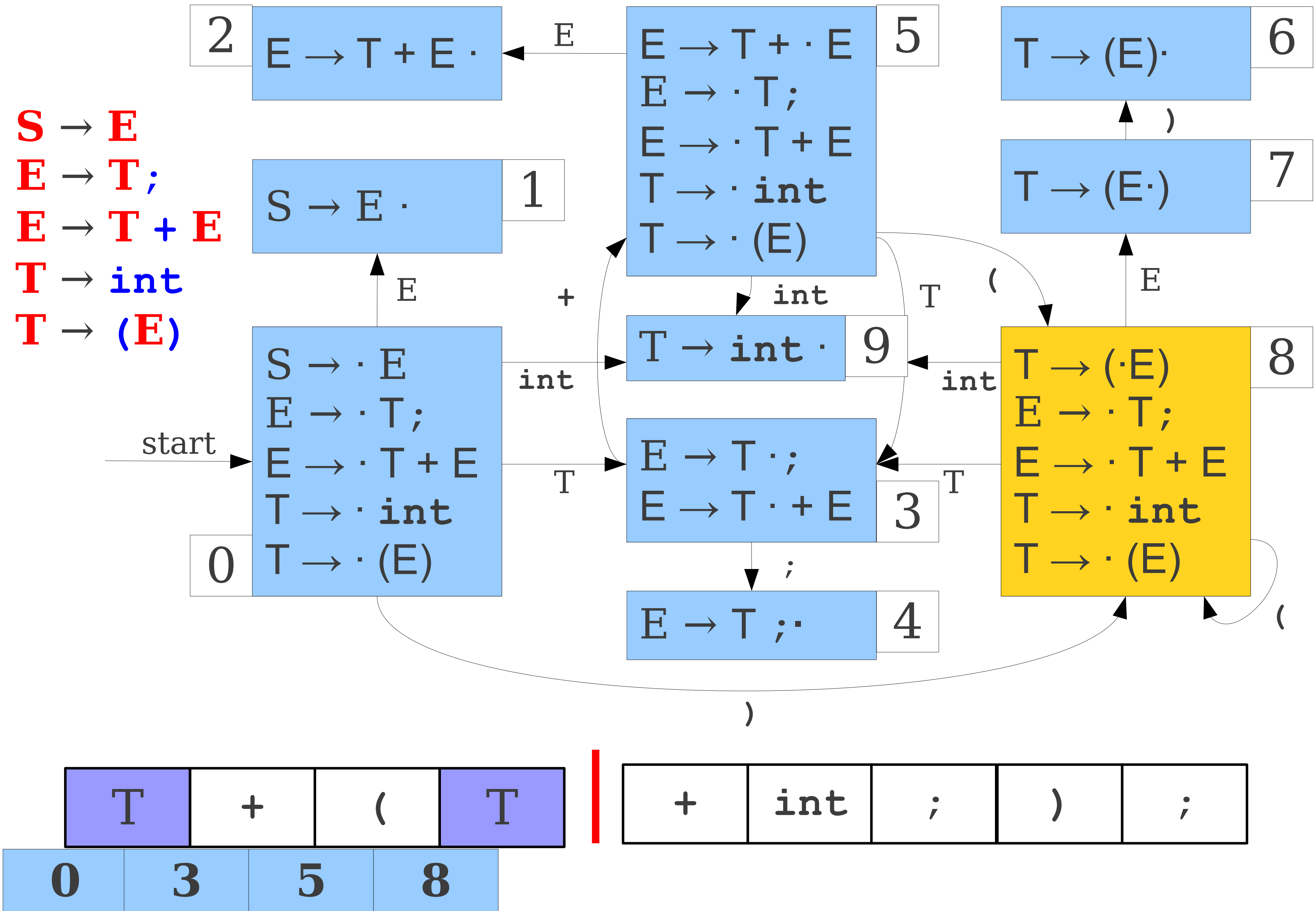


T	+	(
0	**3**	**5**	**8**

+	int	;	)	;

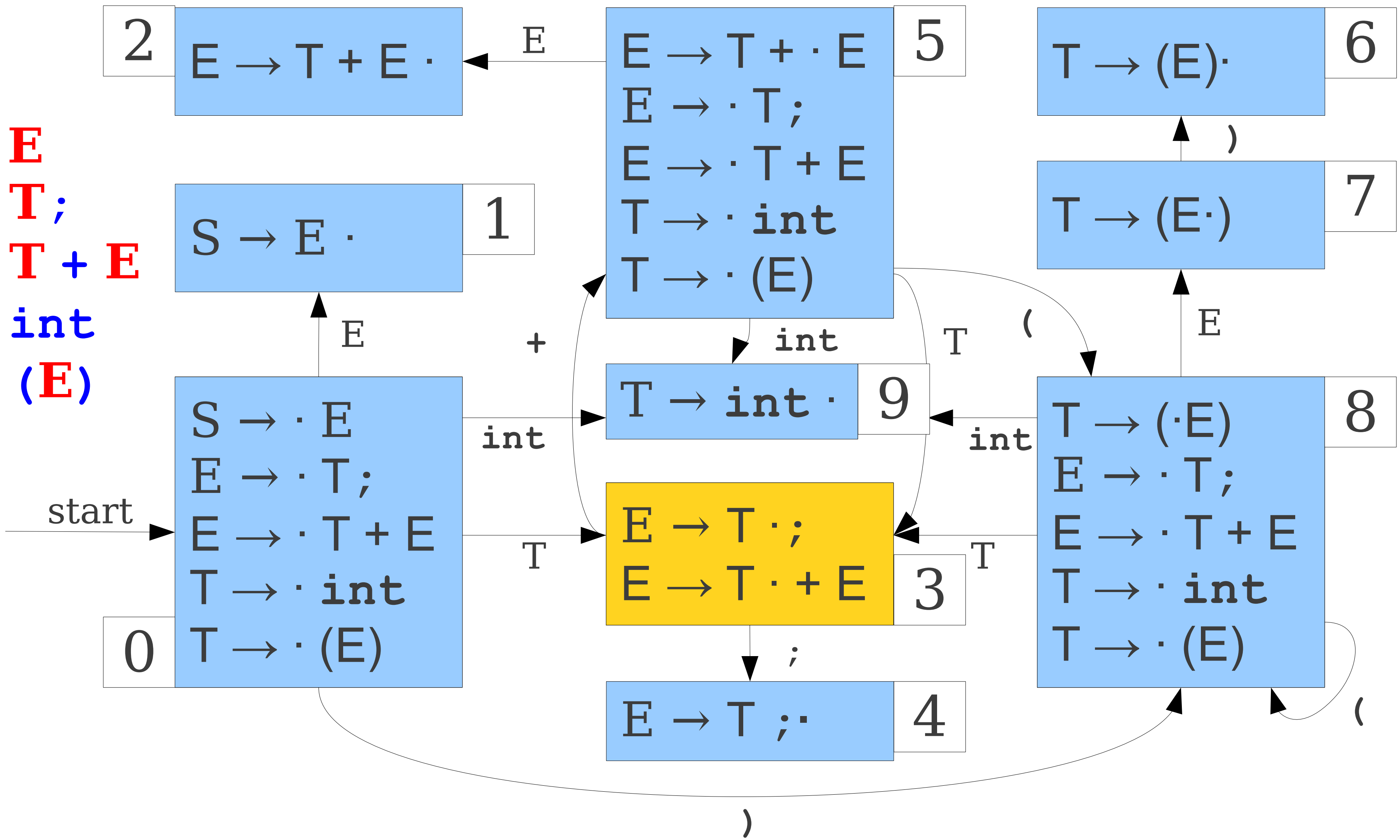
LR(0) Parsing

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**)

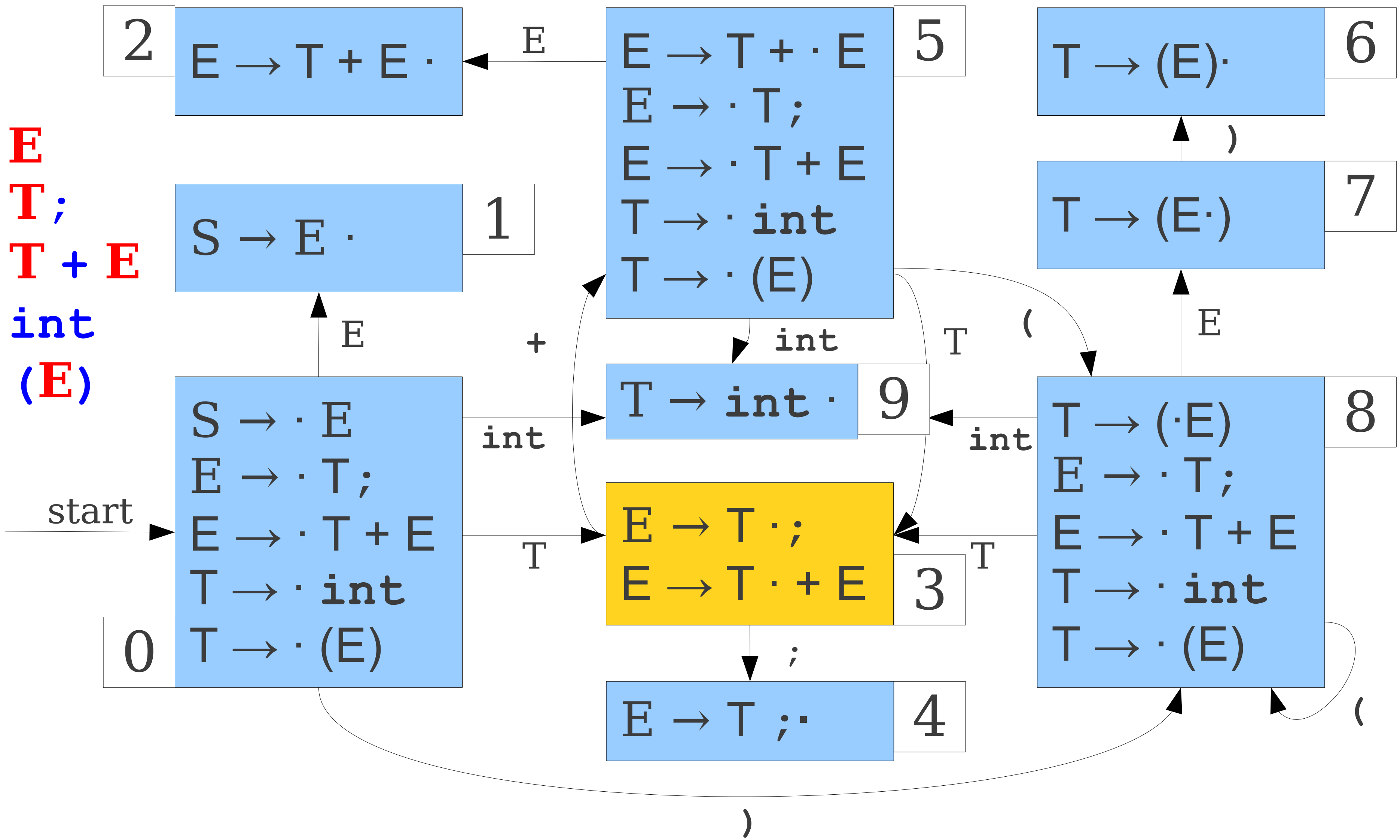


T	+	(	T
0	3	5	8

+	int	;	)	;

LR(0) Parsing

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



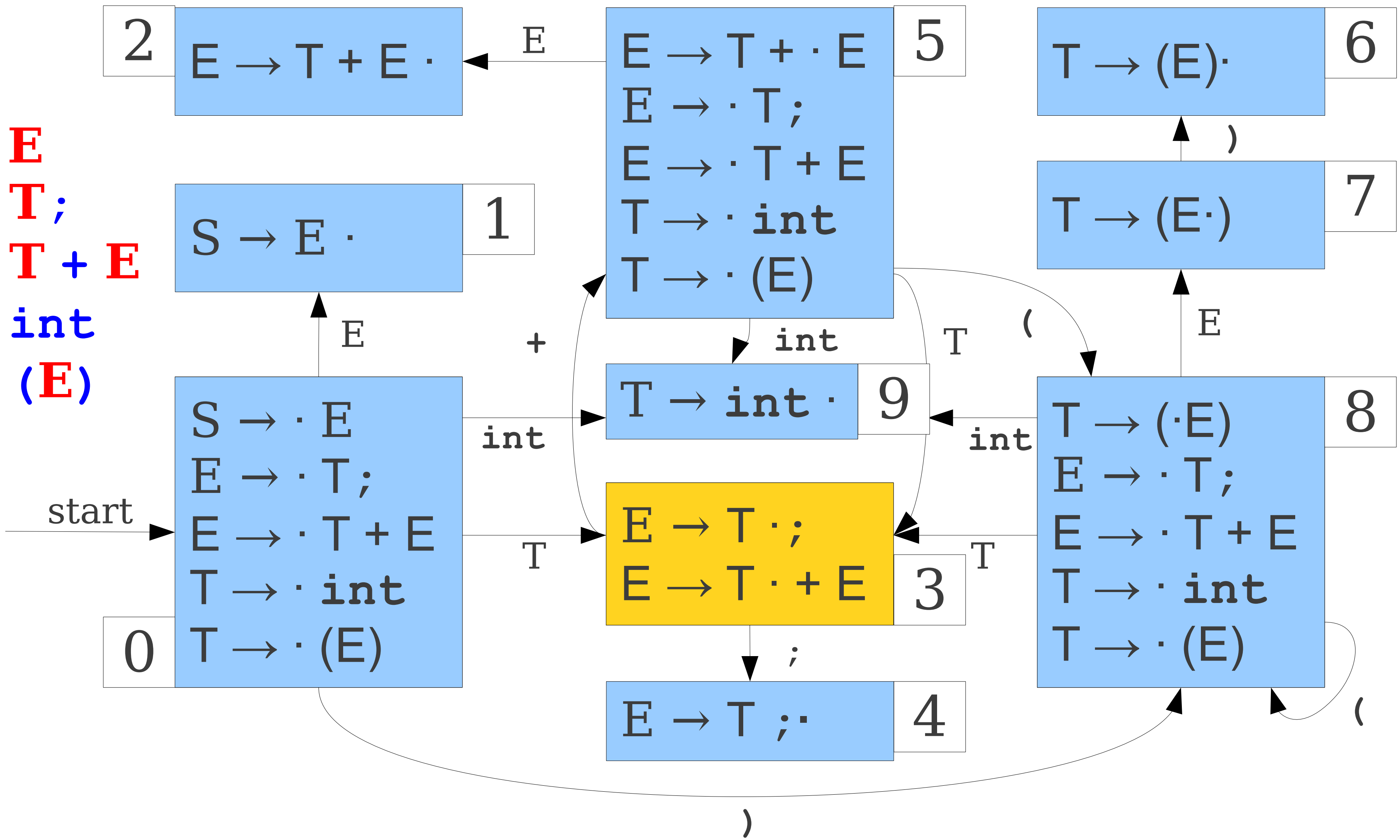
T	+	(	T

+	int	;	)	;

0	3	5	8	3

LR(0) Parsing

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

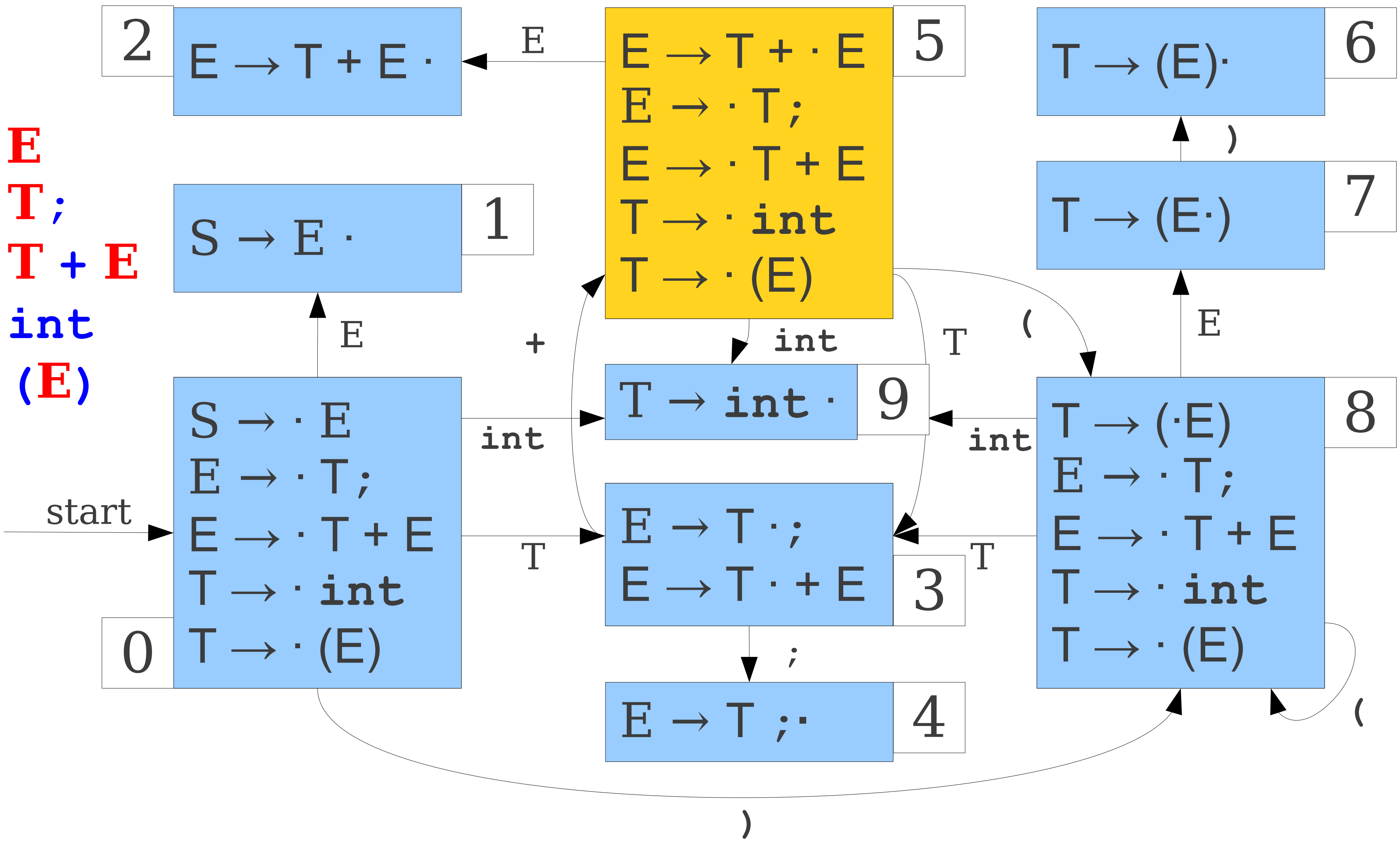


T	+	(	T	+
0	3	5	8	3

`int`	;	)	;

LR(0) Parsing

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

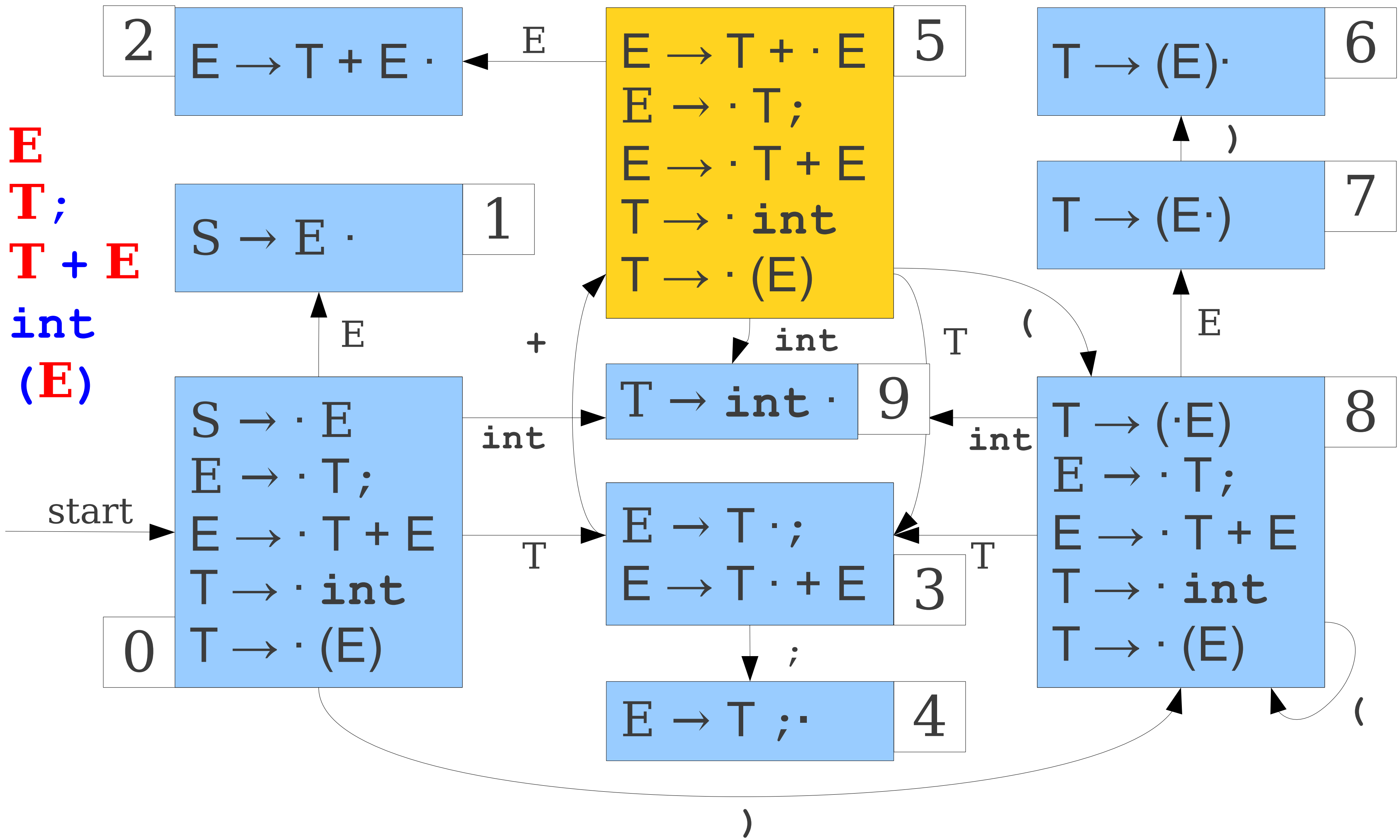


T	+	(	T	+
0	3	5	8	3

int	;	)	;

LR(0) Parsing

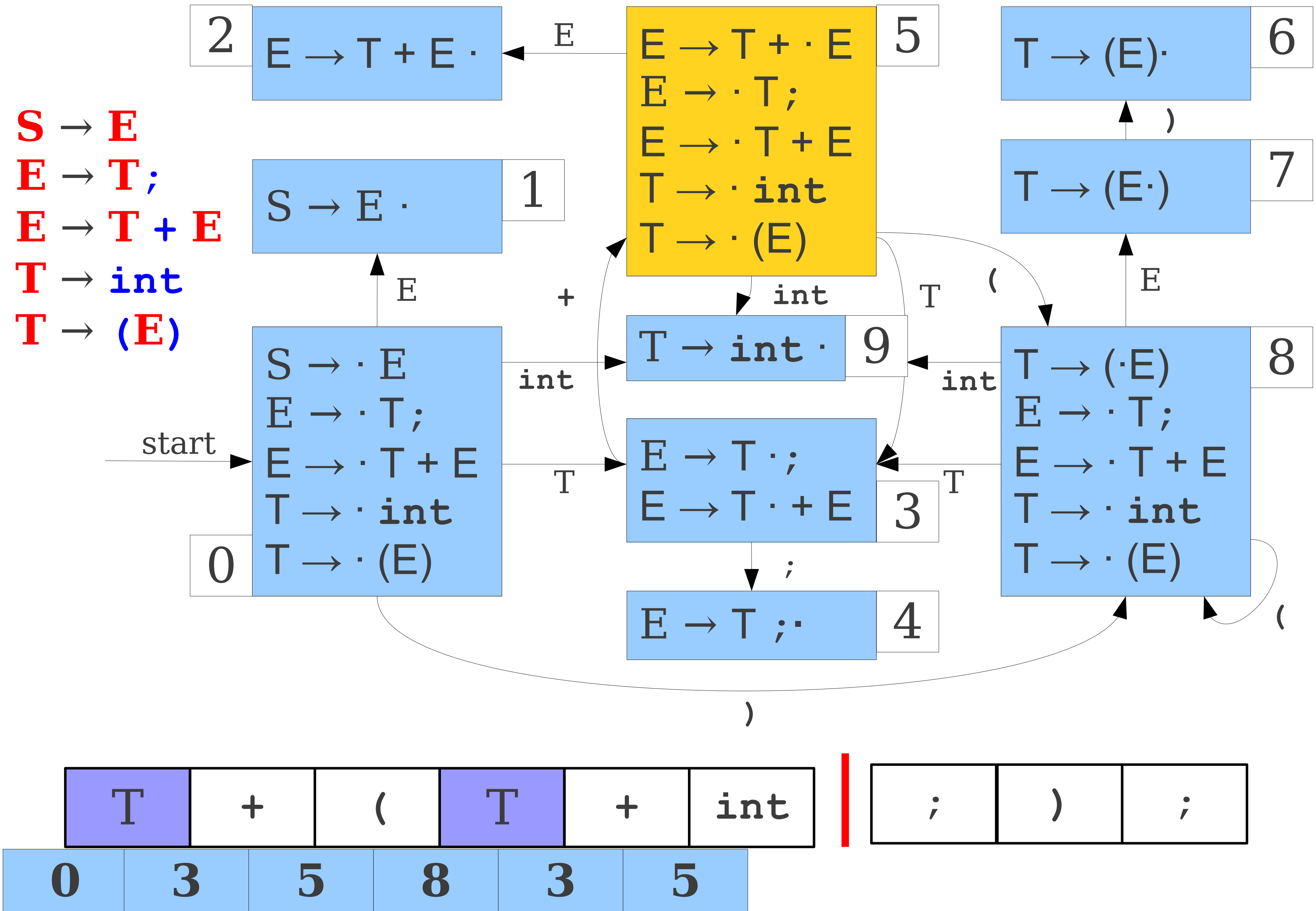
S → **E**
E → **T** ;
E → **T** + **E**
T → int
T → (**E**)



T	+	(	T	+	|	int	;	)	;
0	3	5	8	3	5				

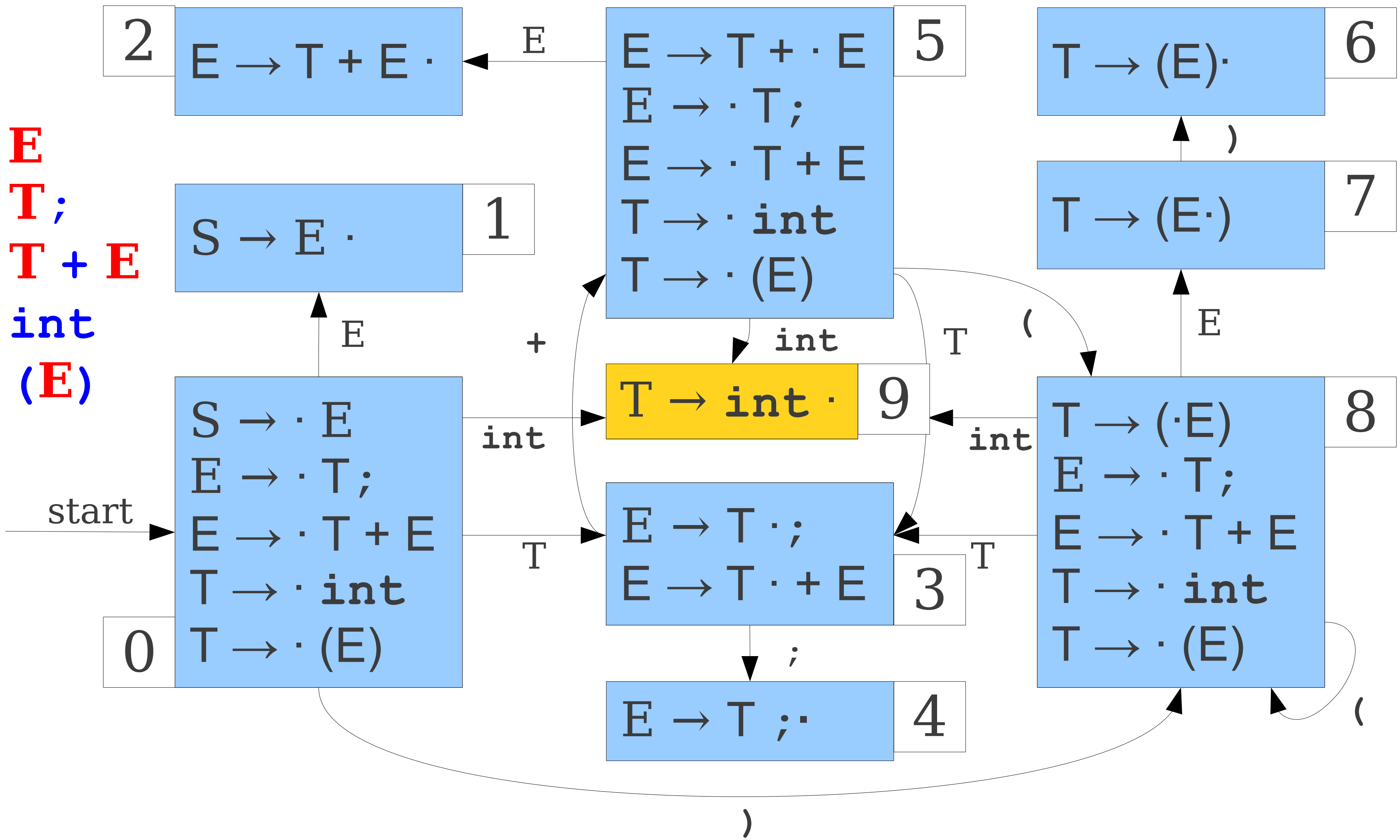
LR(0) Parsing

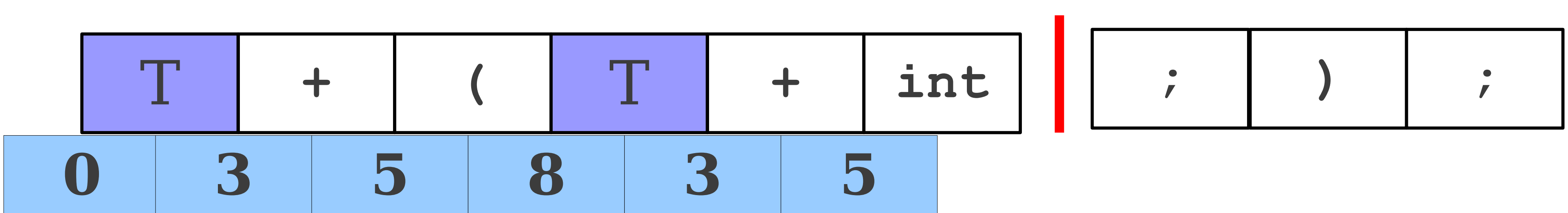
S → E
E → T;
E → T + E
T → int
T → (E)



LR(0) Parsing

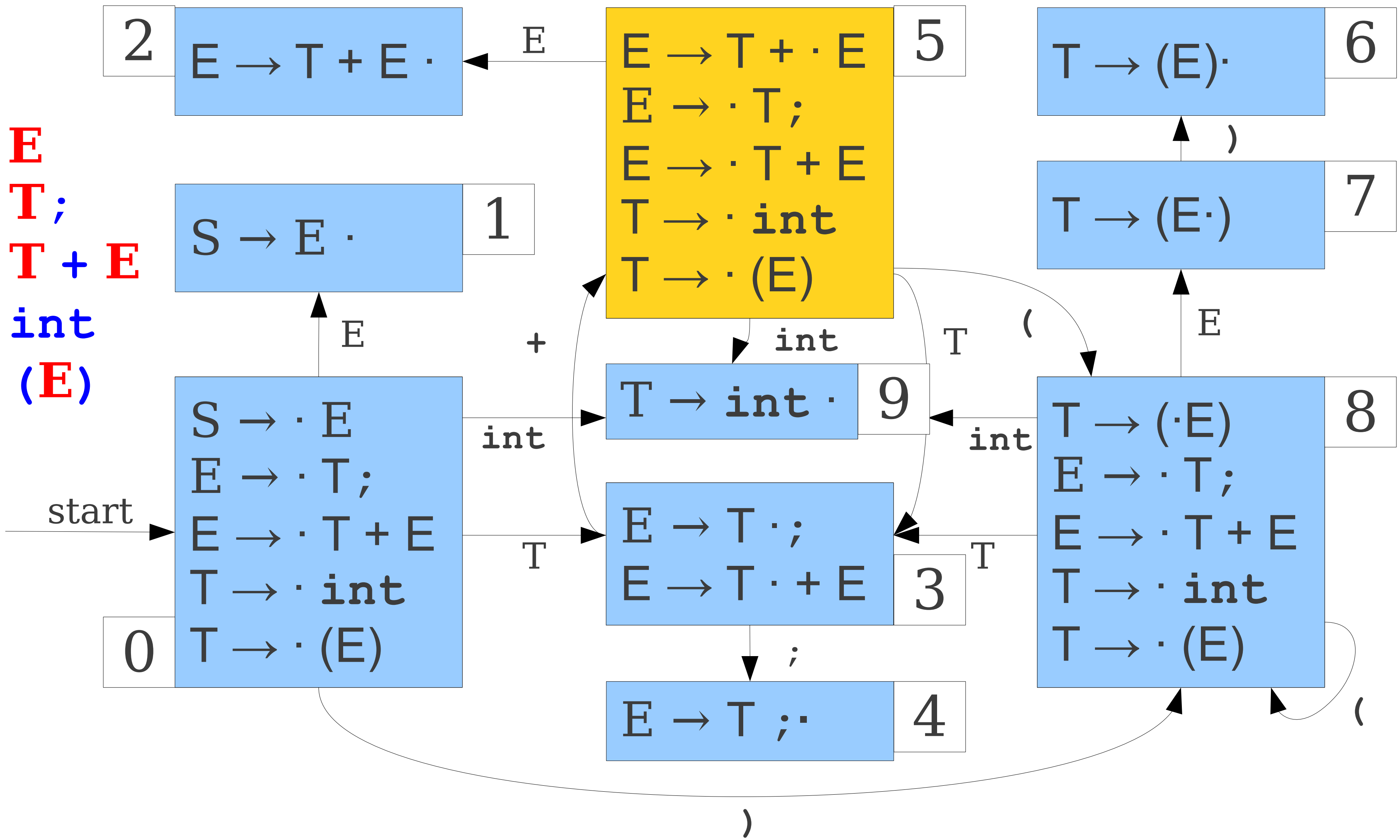
S → **E**
E → **T**;
E → **T** + **E**
T → **int**
T → **(E)**

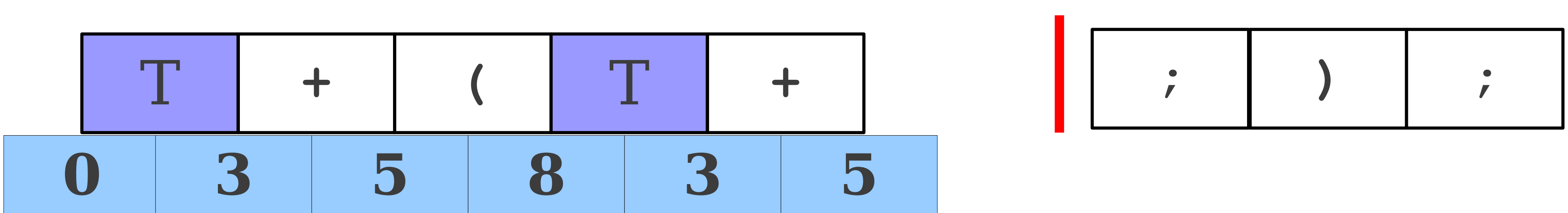




LR(0) Parsing

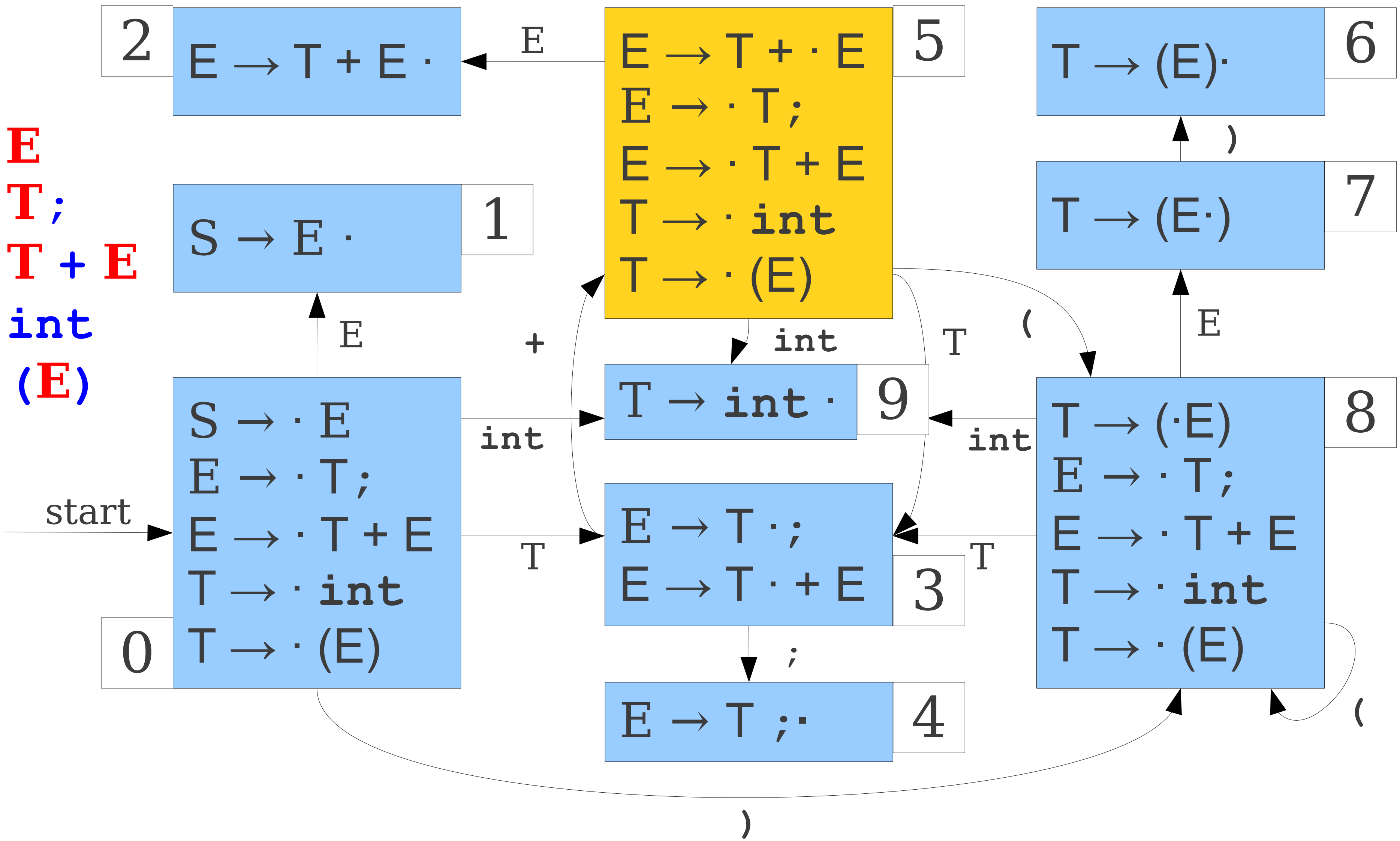
S → **E**
E → **T** ;
E → **T** + **E**
T → `int`
T → (**E**)





LR(0) Parsing

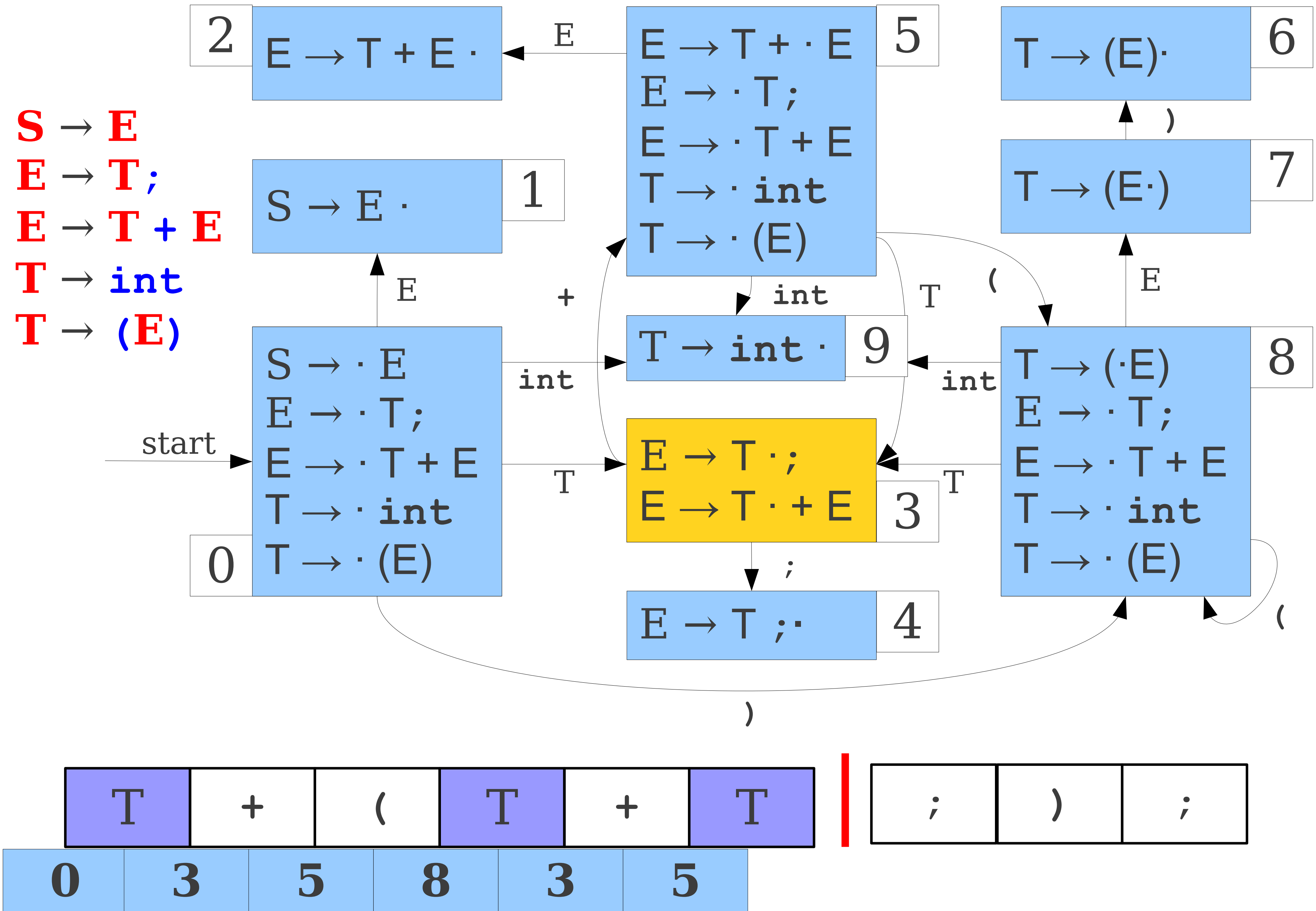
S → **E**
E → **T** `;`
E → **T** **+** **E**
T → `int`
T → `(`**E**`)`



T	+	(	T	+	T	|	;	)	;

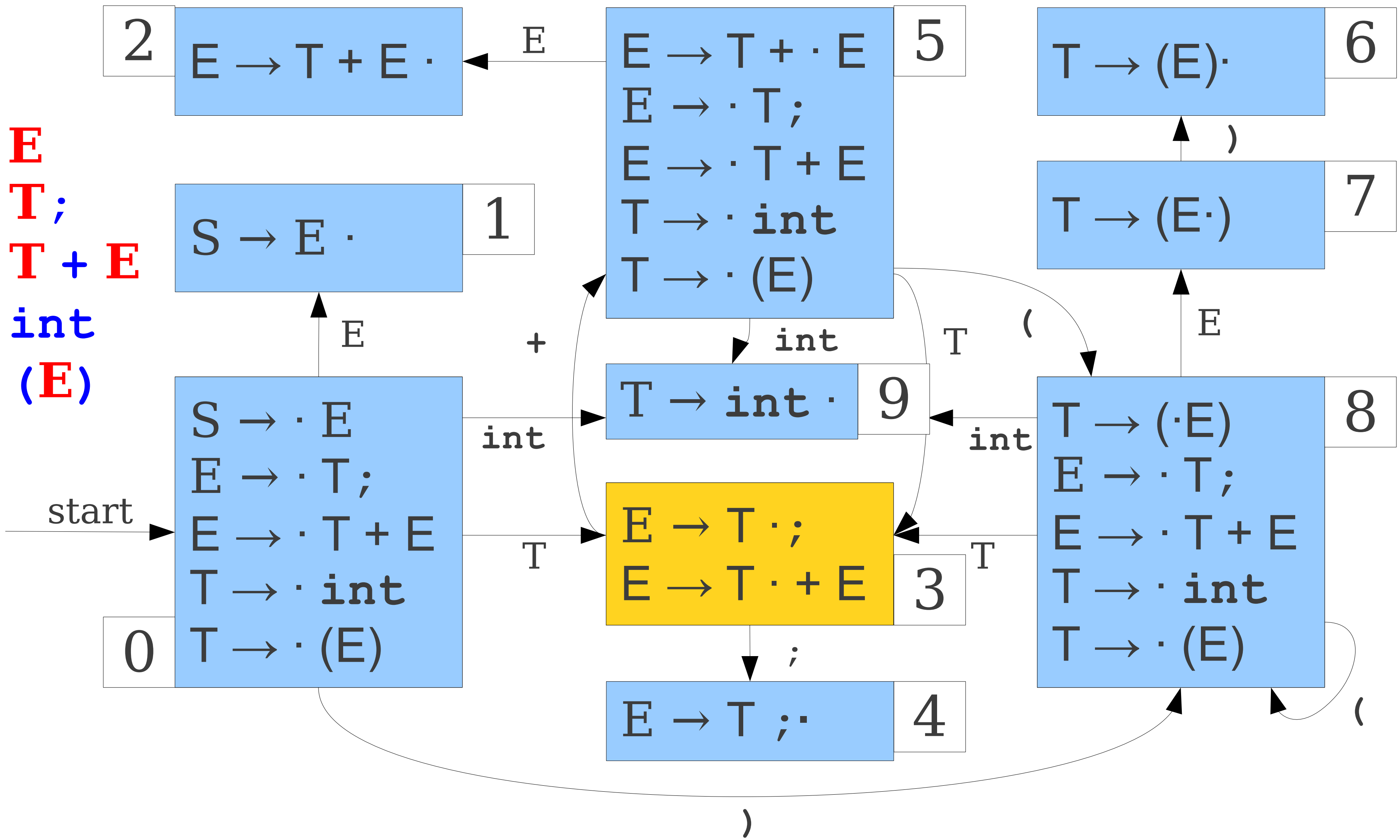
0	**3**	**5**	**8**	**3**	**5**

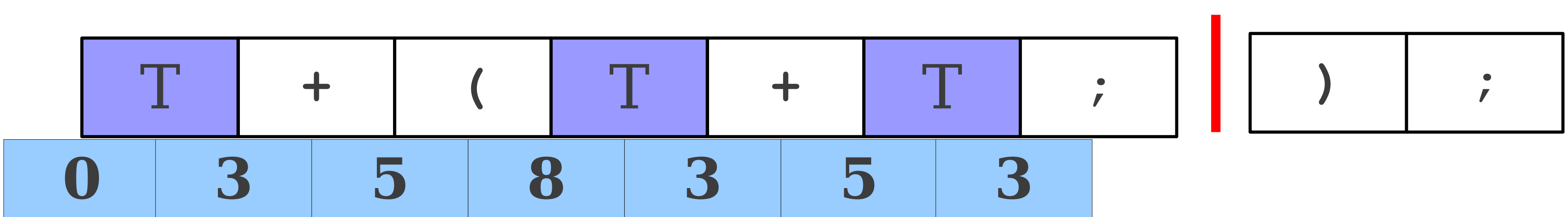
LR(0) Parsing


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

LR(0) Parsing

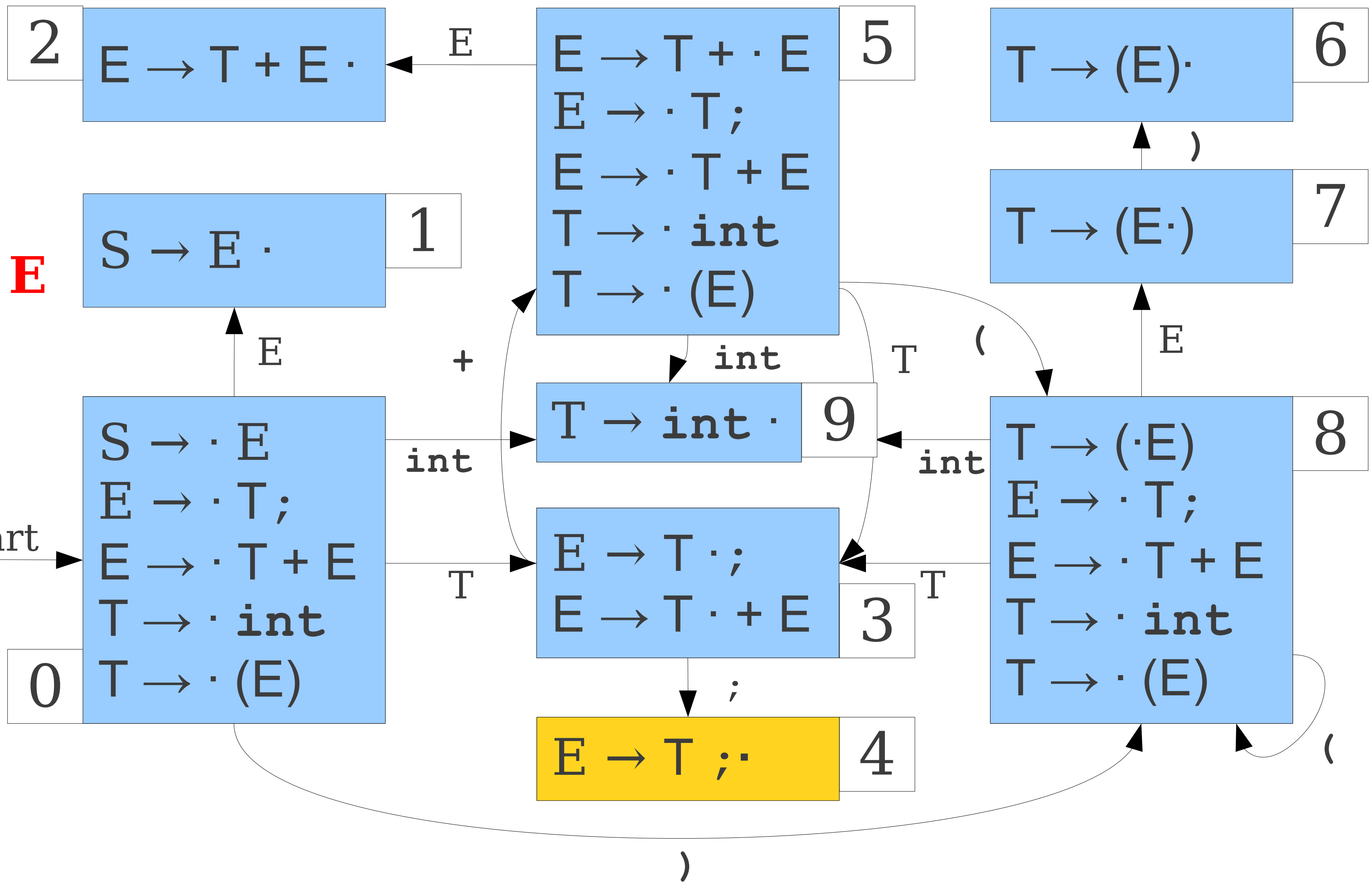
$S \rightarrow E$
$E \rightarrow T;$
$E \rightarrow T + E$
$T \rightarrow$ int
$T \rightarrow (E)$





LR(0) Parsing

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



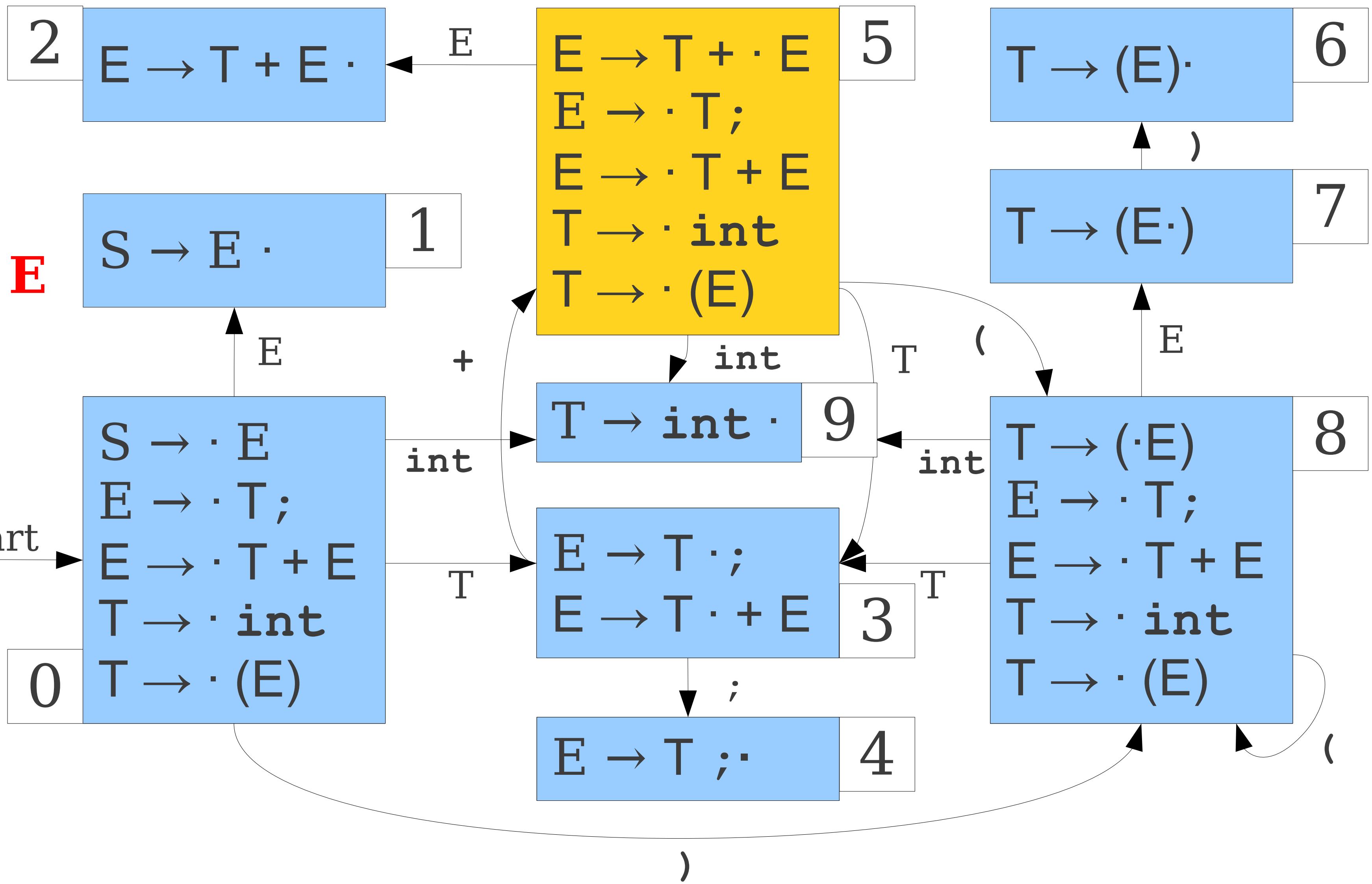
T	+	(	T	+	T	;

)	;

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

LR(0) Parsing

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



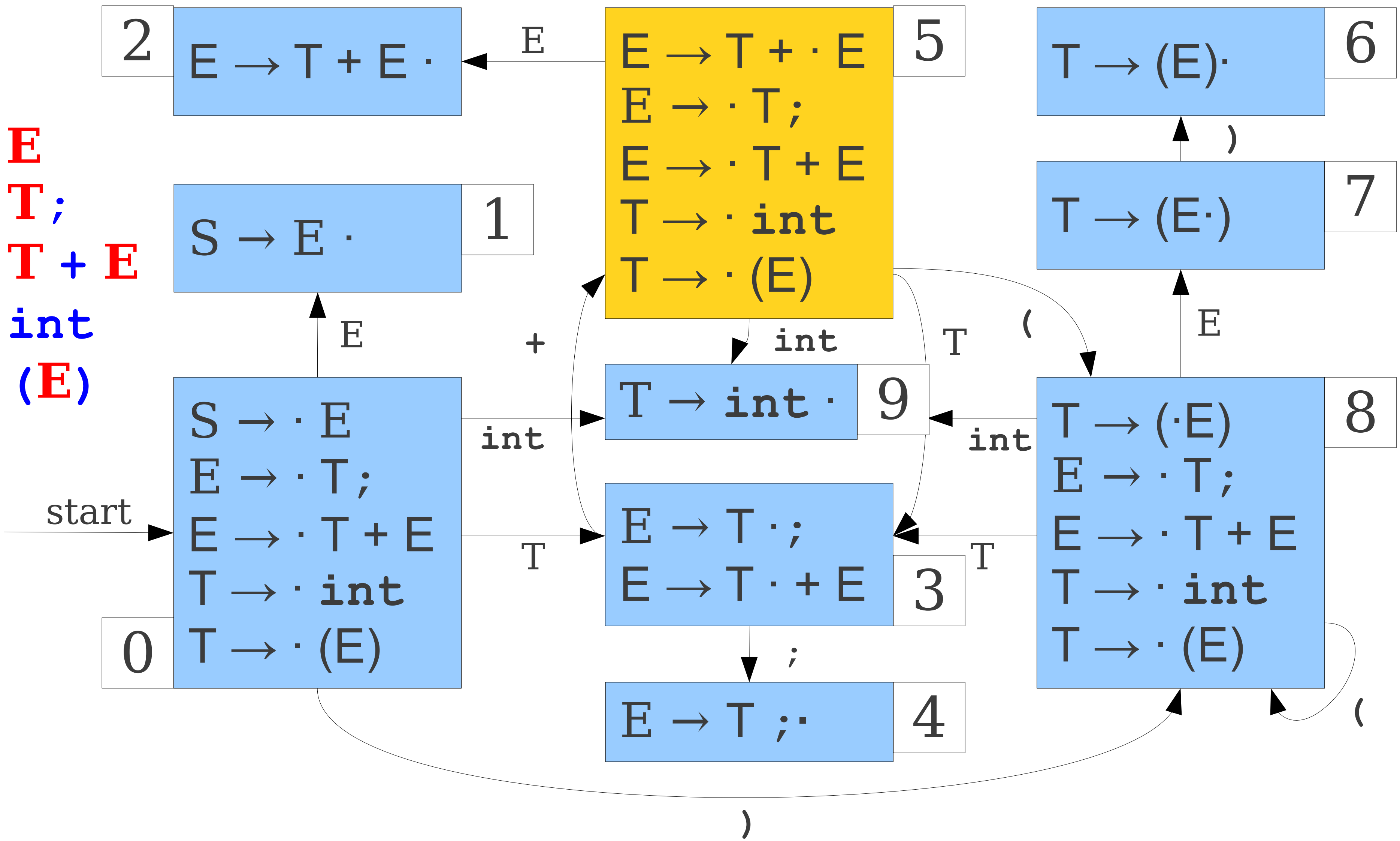
T	+	(	T	+

0	3	5	8	3	5

)	;

LR(0) Parsing

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

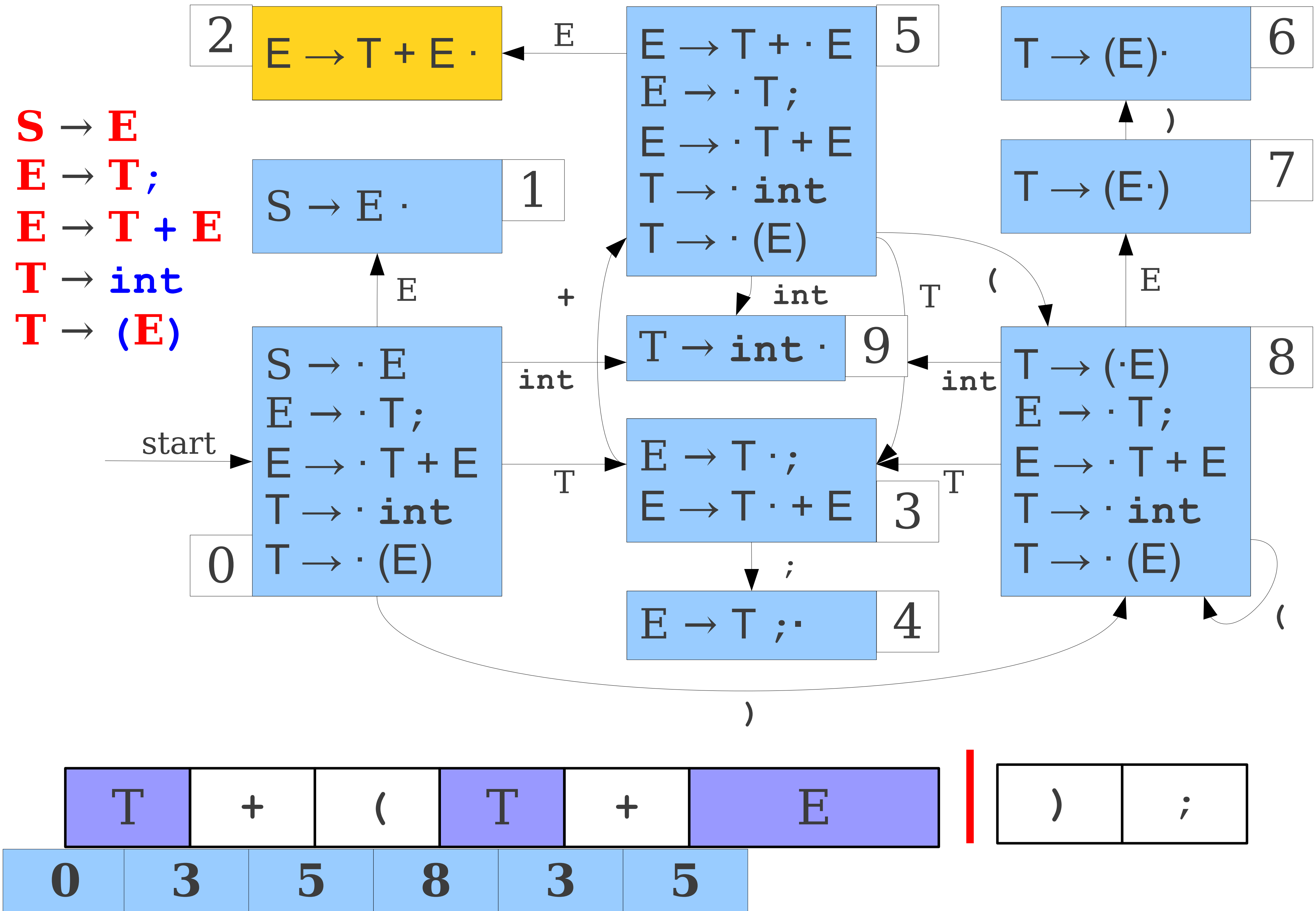


T	+	(	T	+	E

)	;

0	3	5	8	3	5

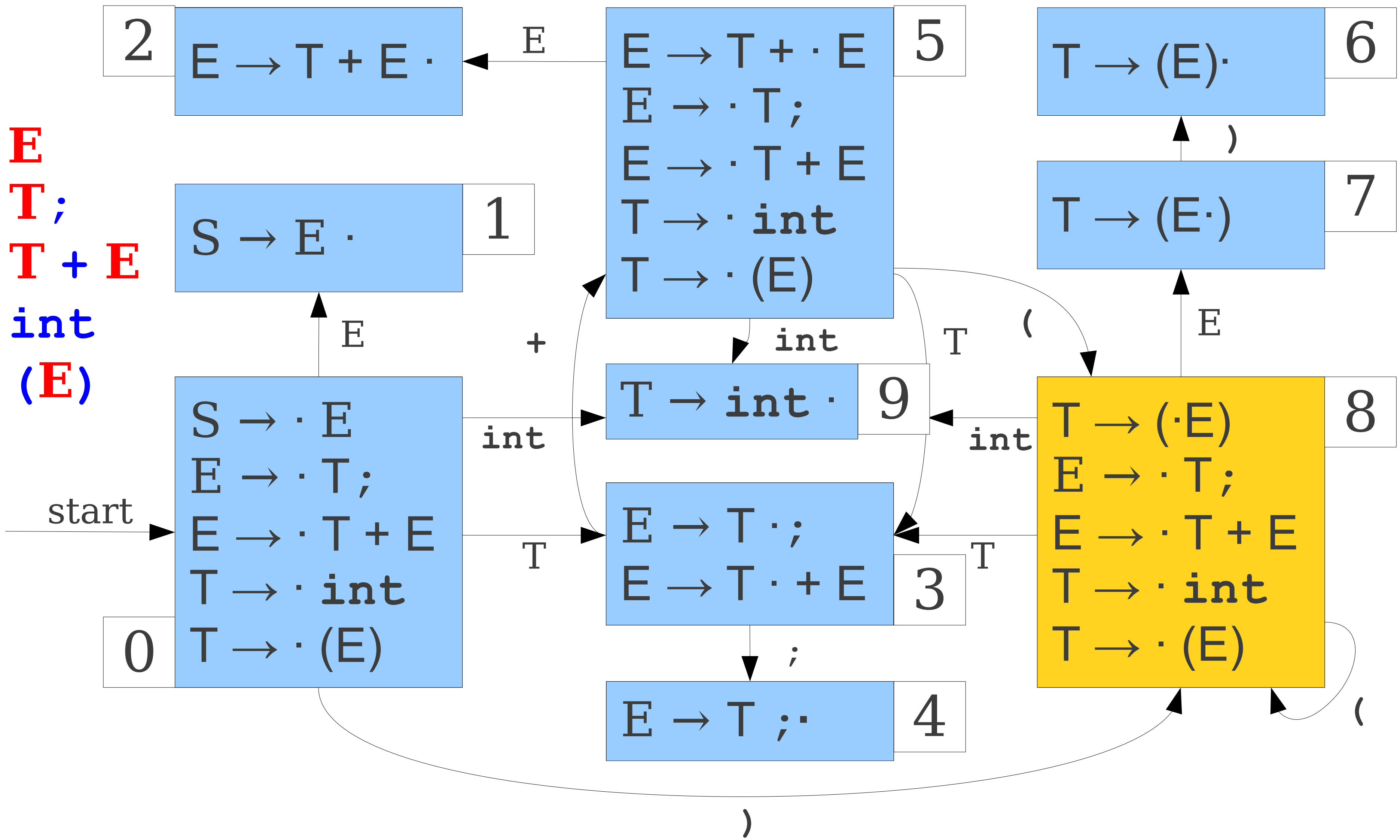
LR(0) Parsing



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

LR(0) Parsing

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



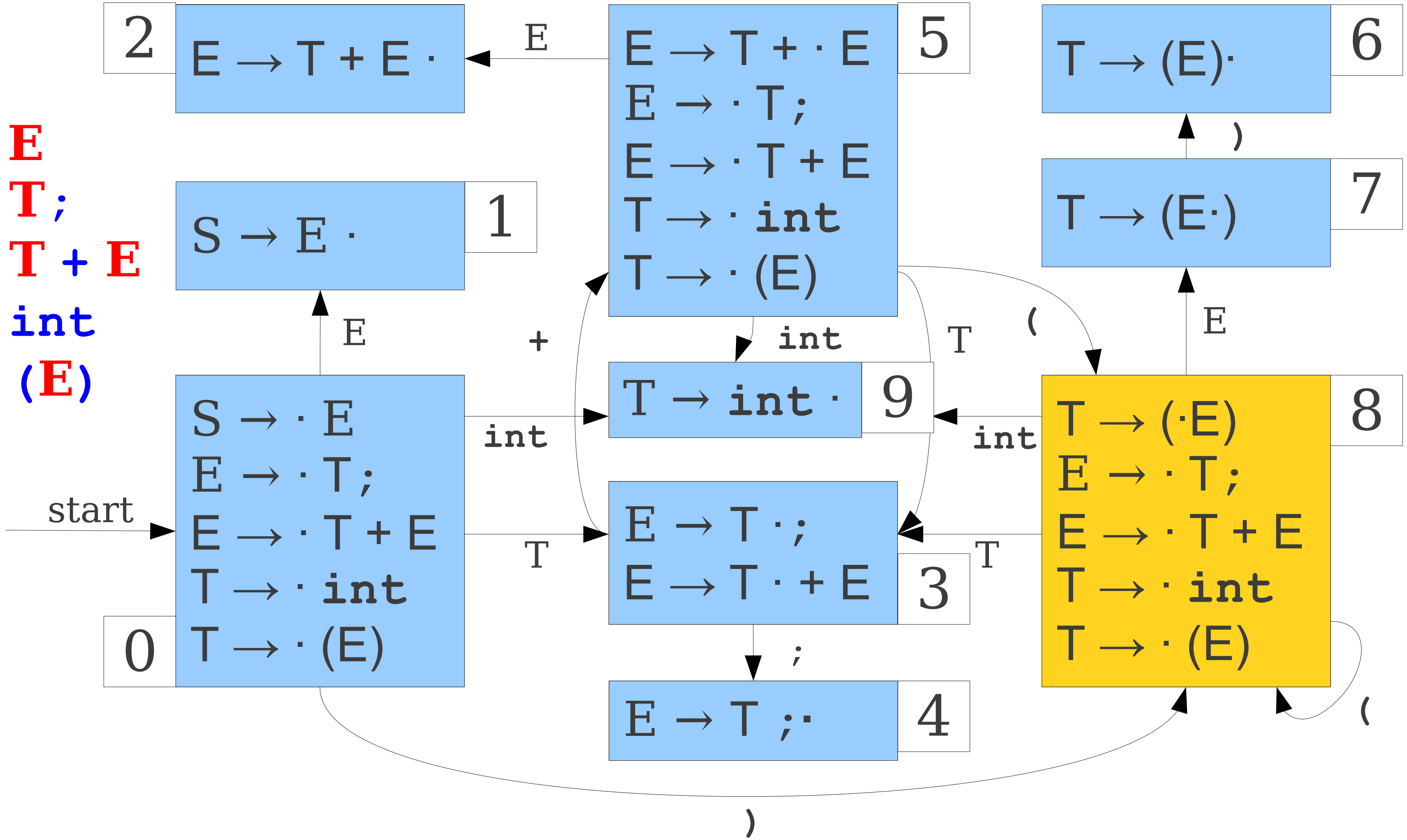
T	+	(

0	3	5	8

)	;

LR(0) Parsing

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

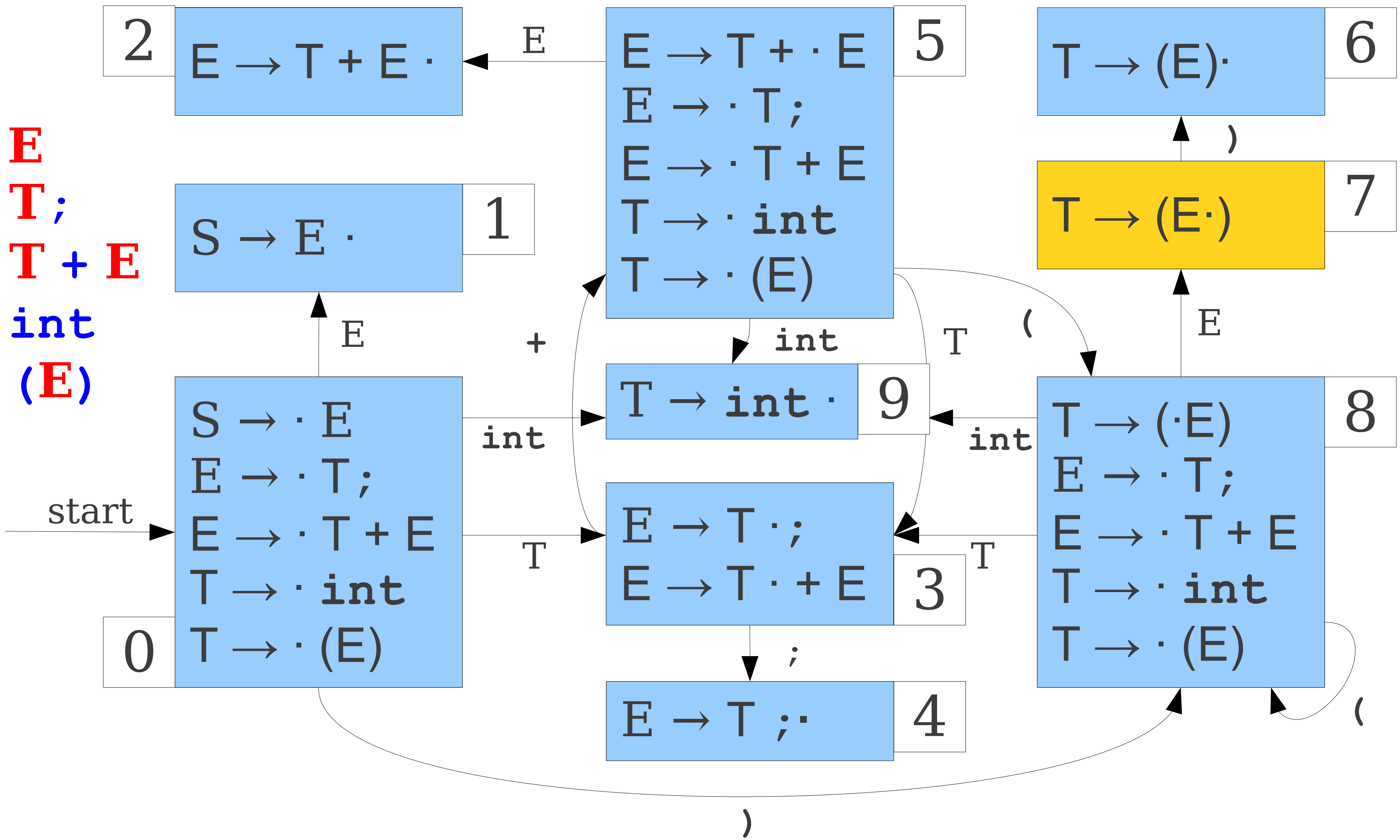


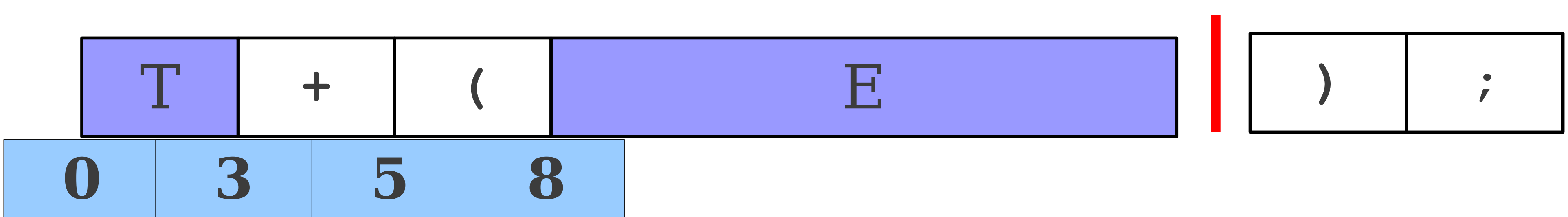
T	+	(	E
0	3	5	8

)	;

LR(0) Parsing

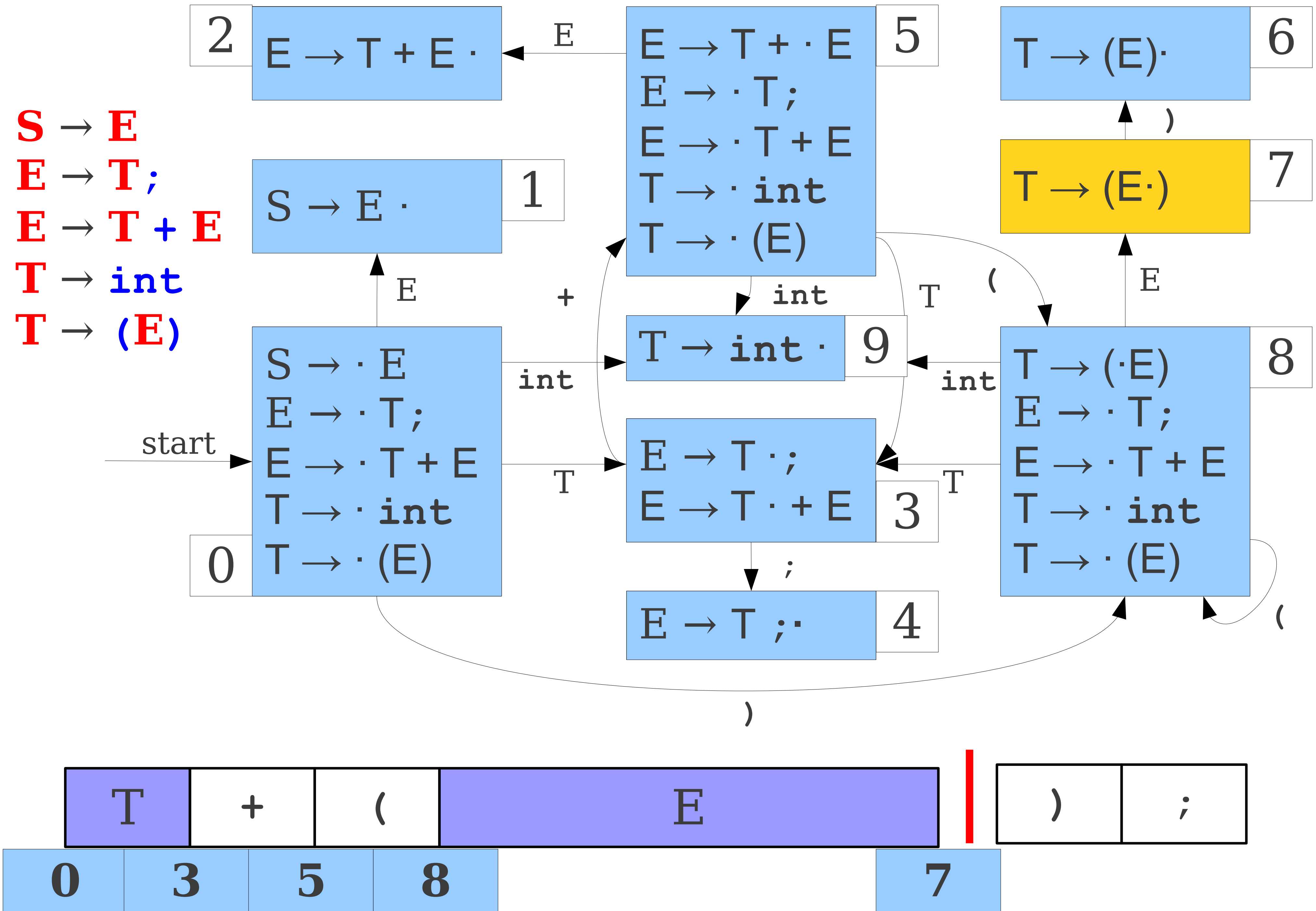
S → **E**
E → **T** ;
E → **T** + **E**
T → `int`
T → (**E**)





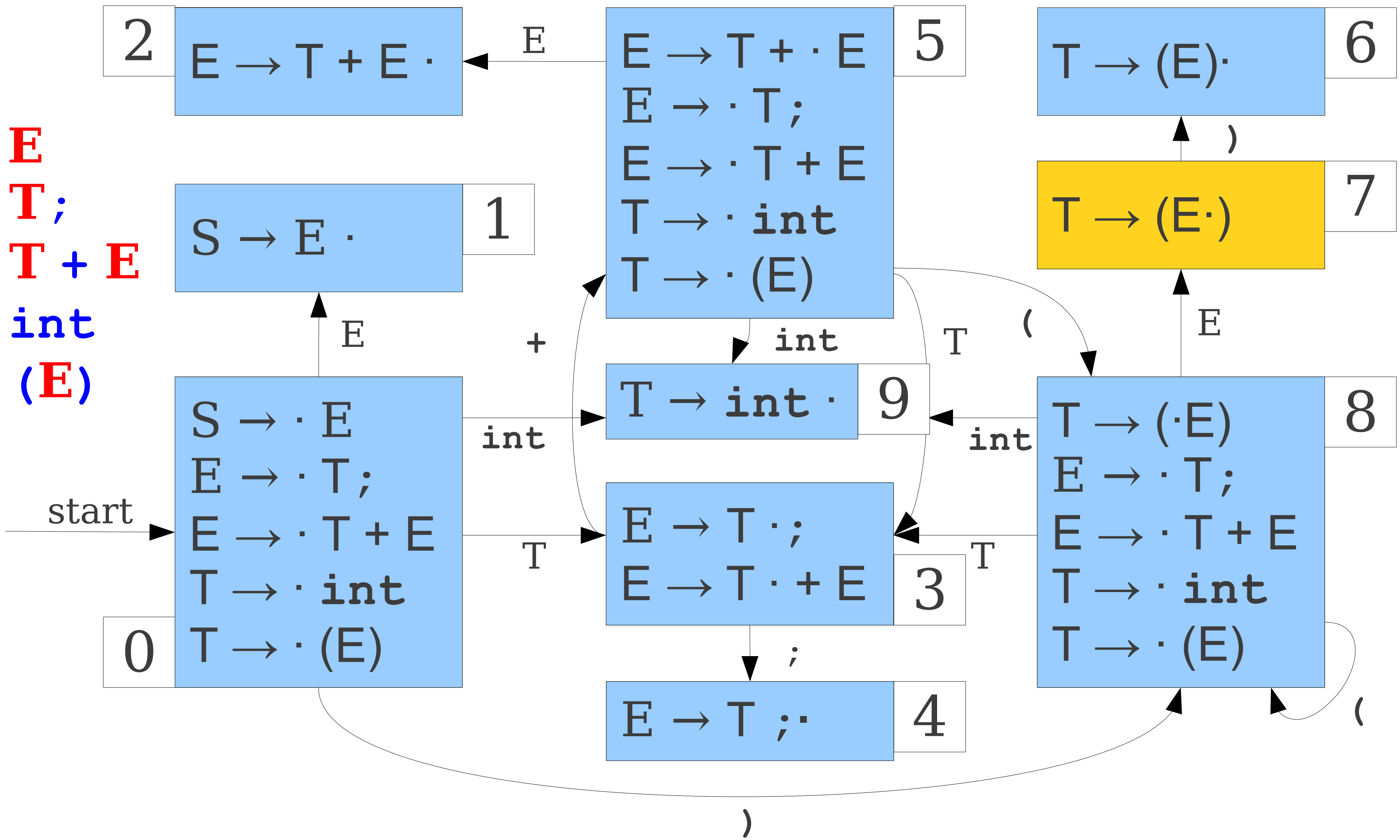
LR(0) Parsing

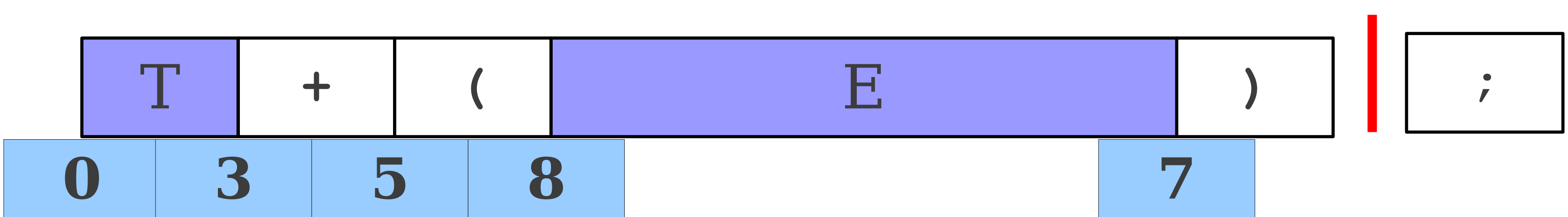
S → **E**
E → **T** ;
E → **T** + **E**
T → `int`
T → (**E**)



LR(0) Parsing

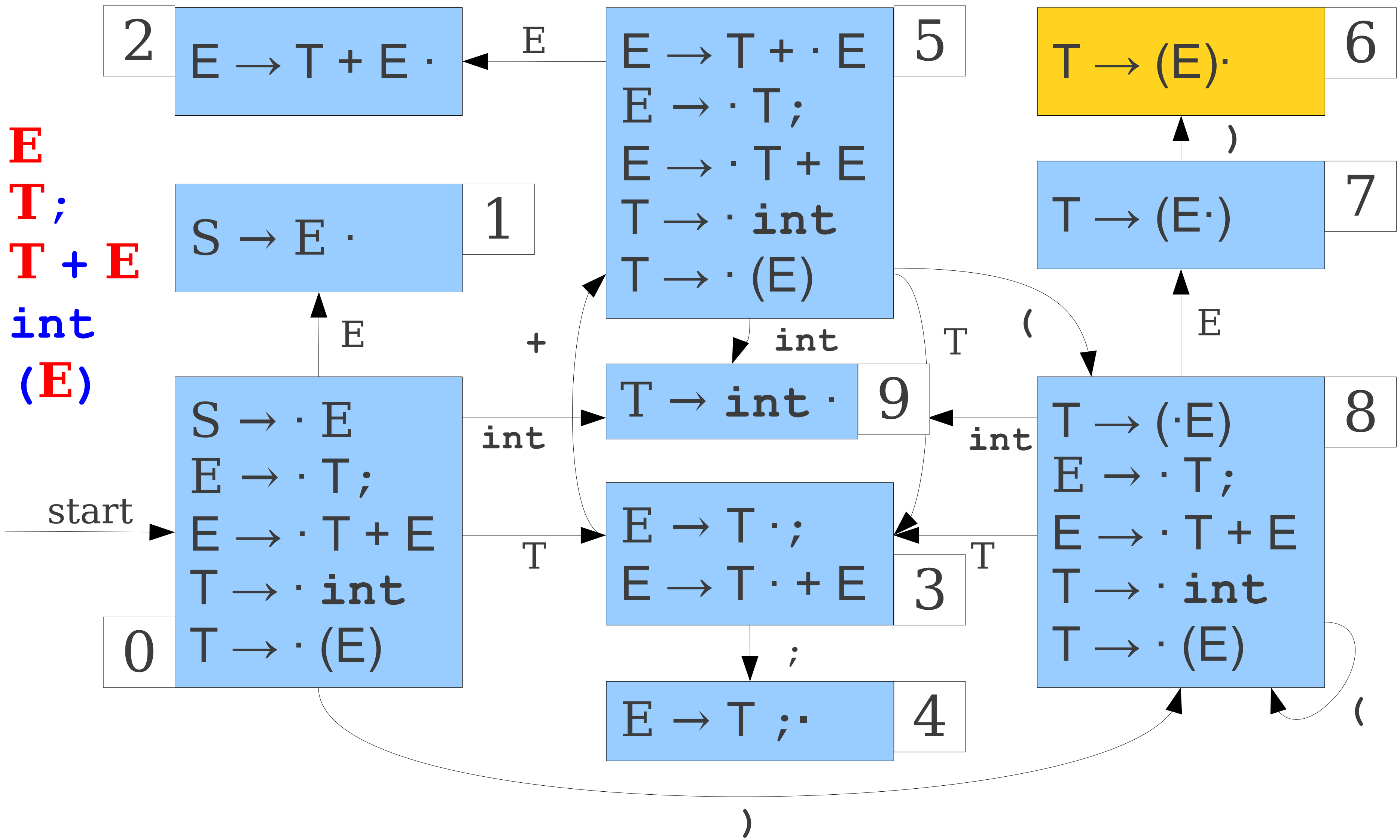
S → E
E → T;
E → T + E
T → int
T → (E)

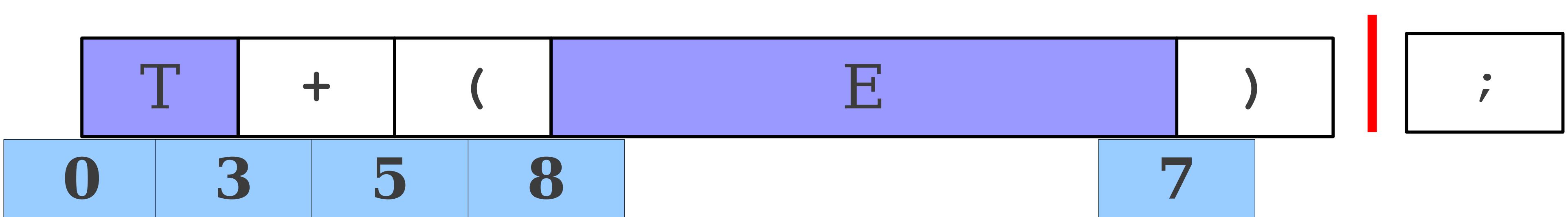




LR(0) Parsing

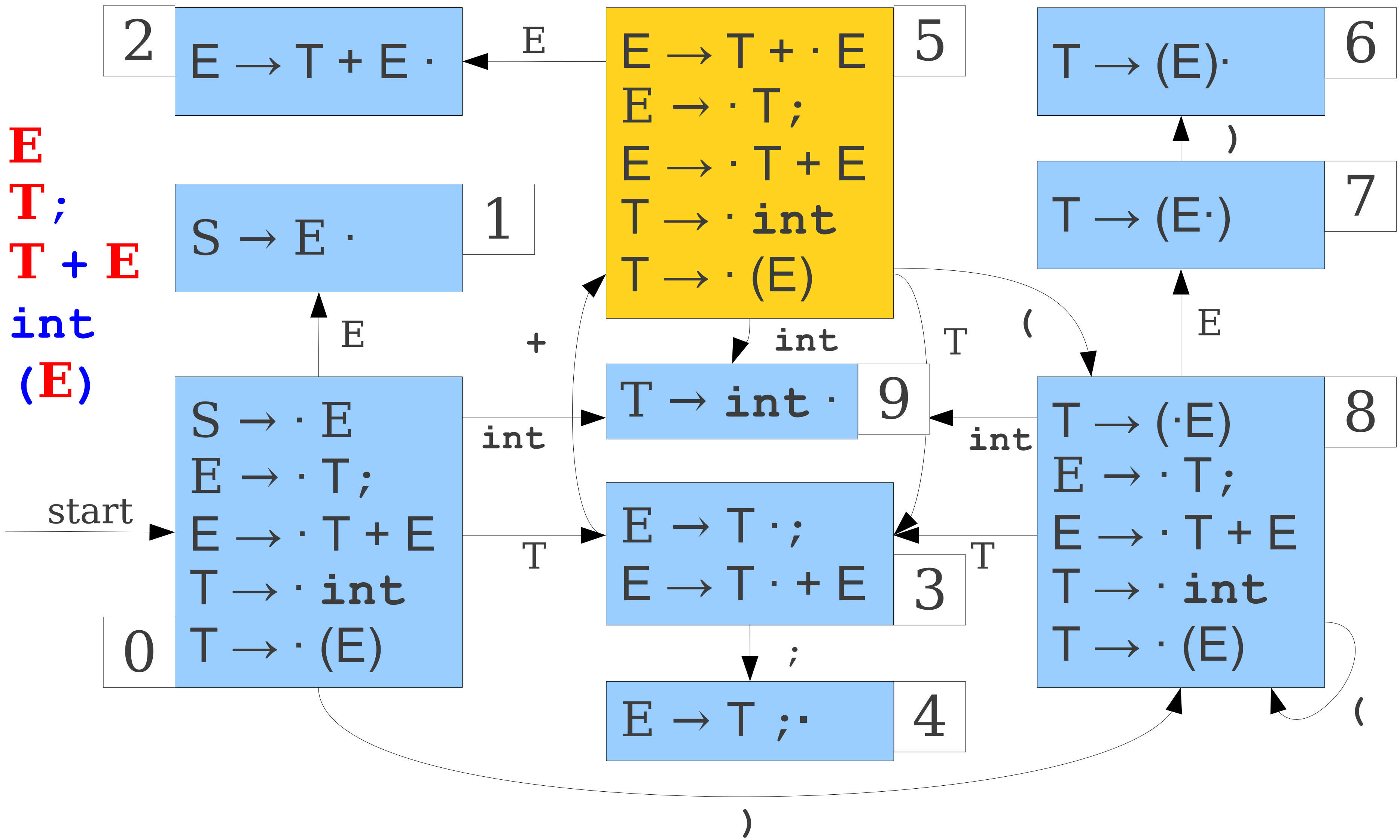
$S \rightarrow E$
$E \rightarrow T;$
$E \rightarrow T + E$
$T \rightarrow$ int
$T \rightarrow (E)$





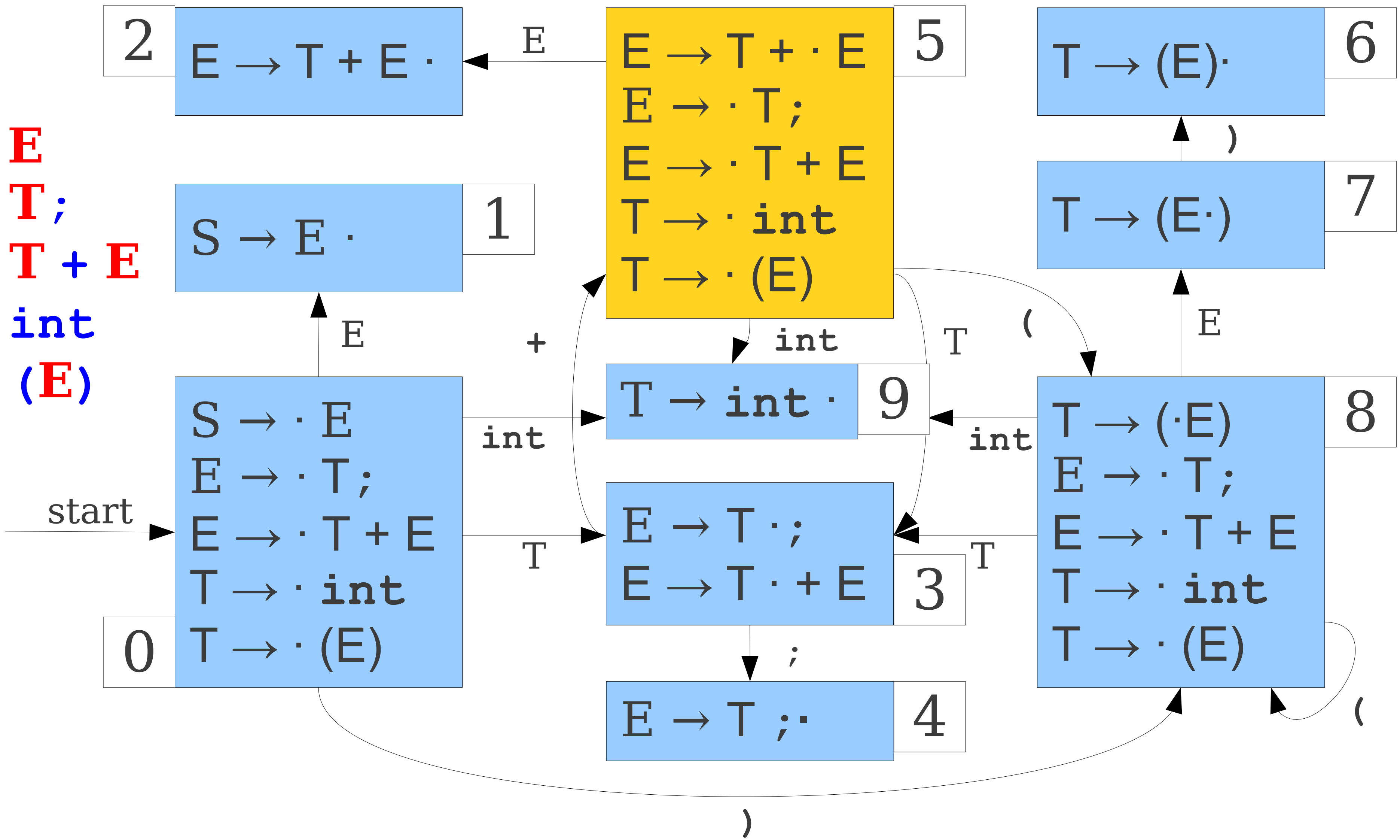
LR(0) Parsing

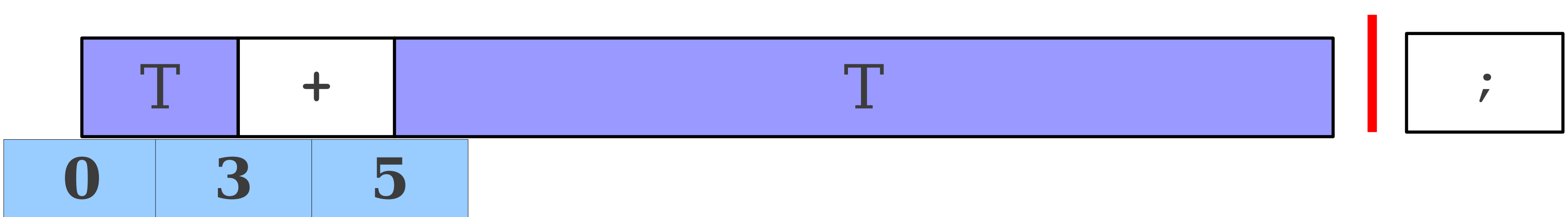
$S \to E$
$E \to T;$
$E \to T + E$
$T \to \texttt{int}$
$T \to (E)$



LR(0) Parsing

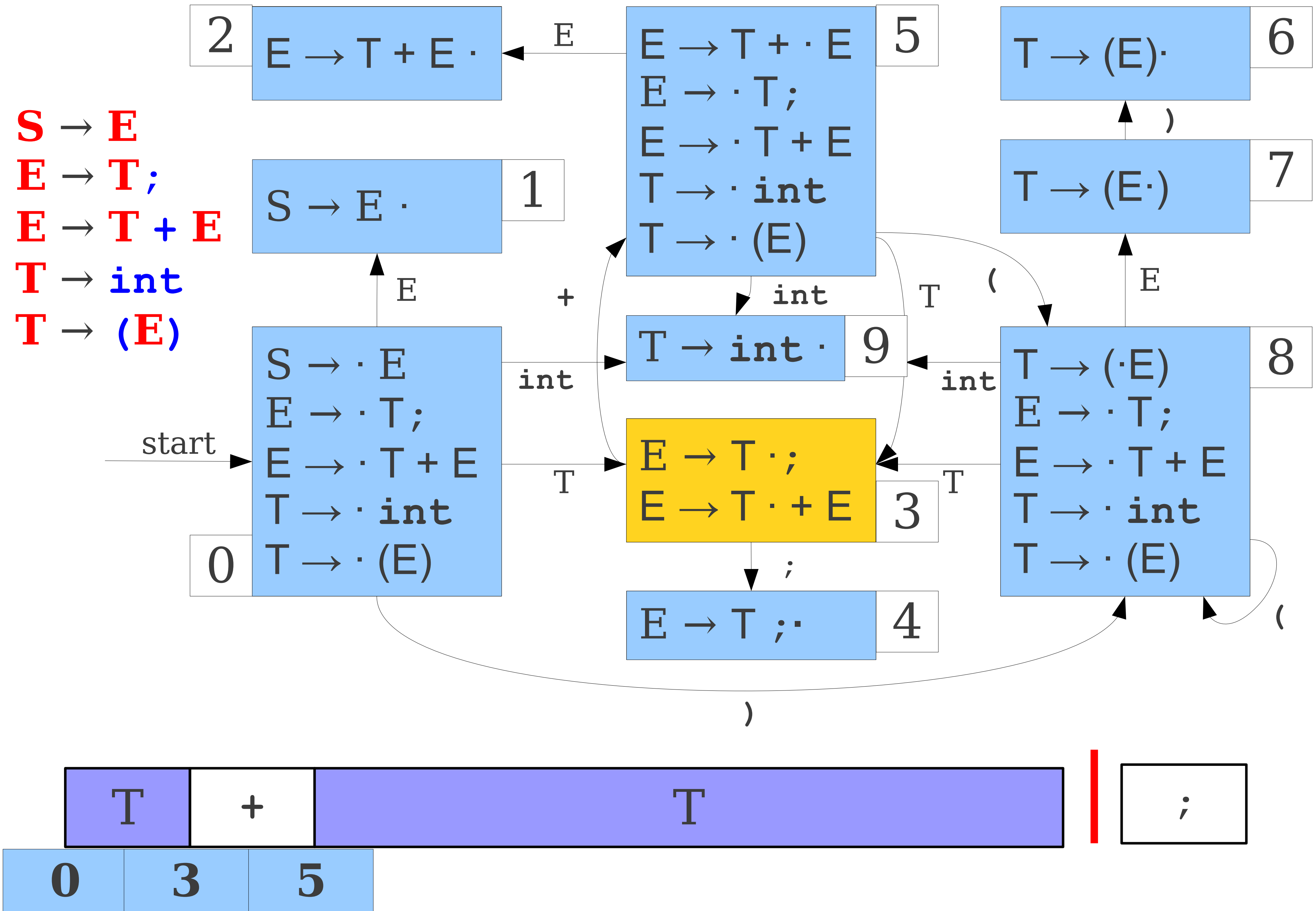
S → **E**
E → **T** ;
E → **T** + **E**
T → **int**
T → **(E)**





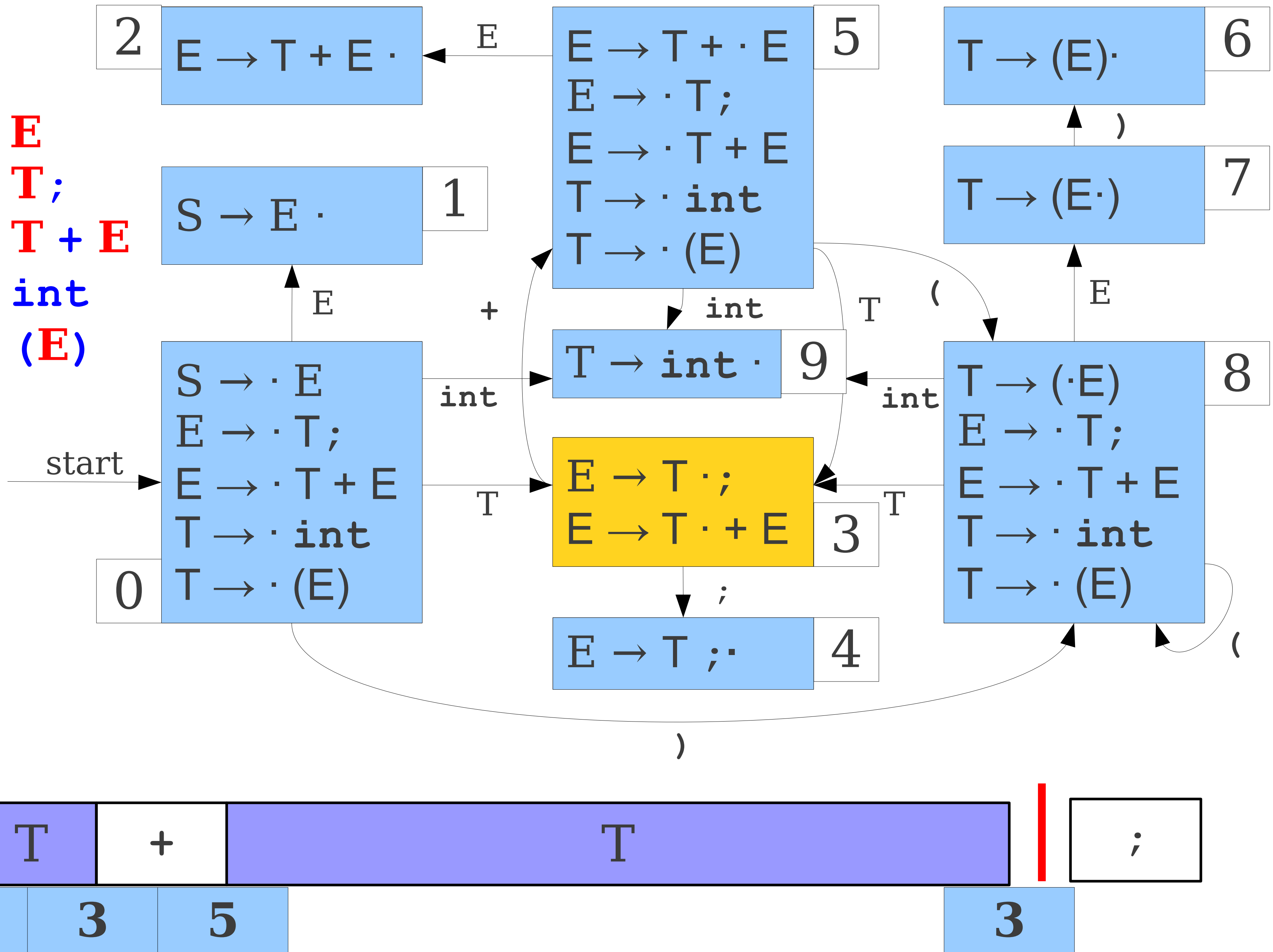
LR(0) Parsing

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



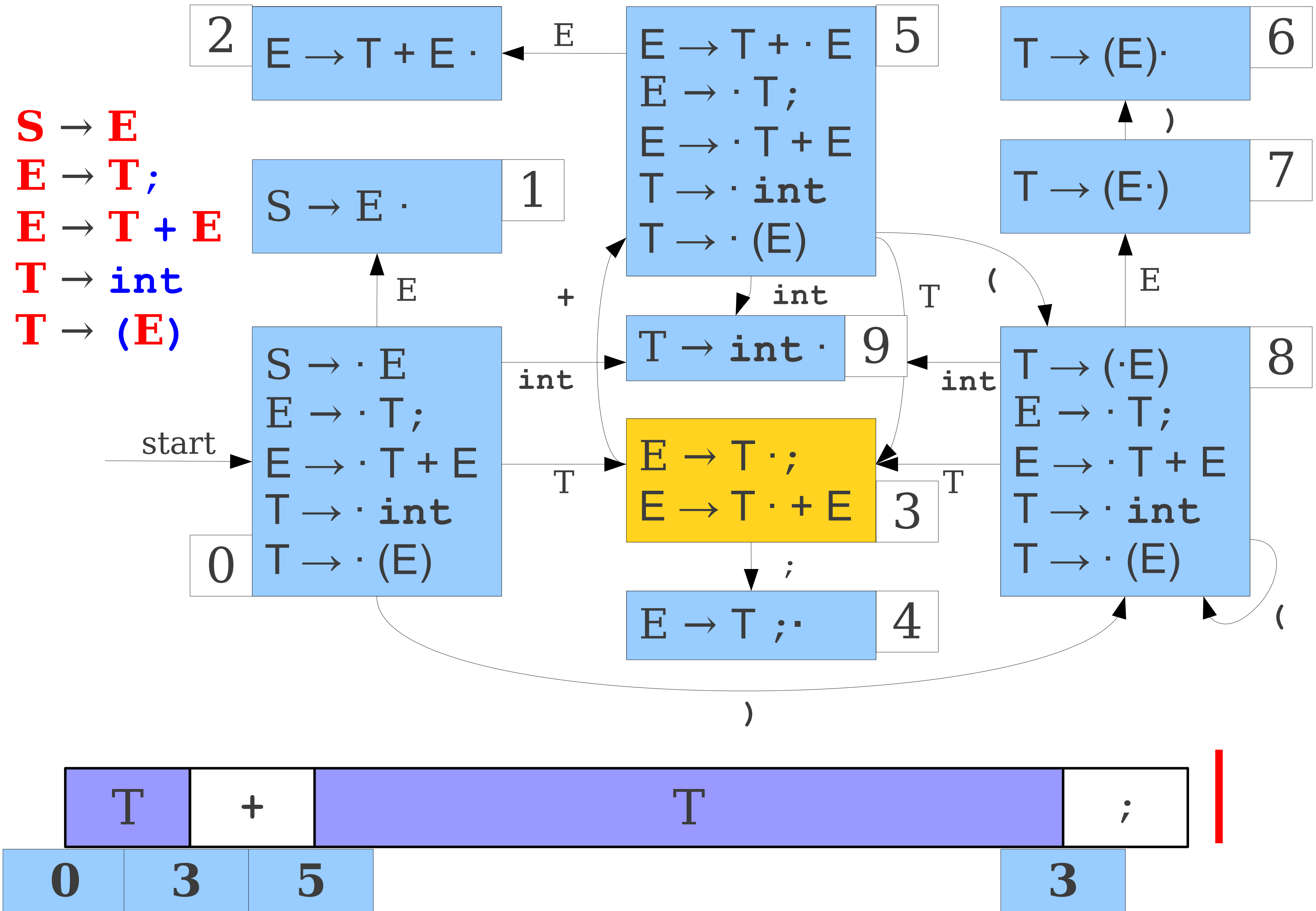
LR(0) Parsing

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



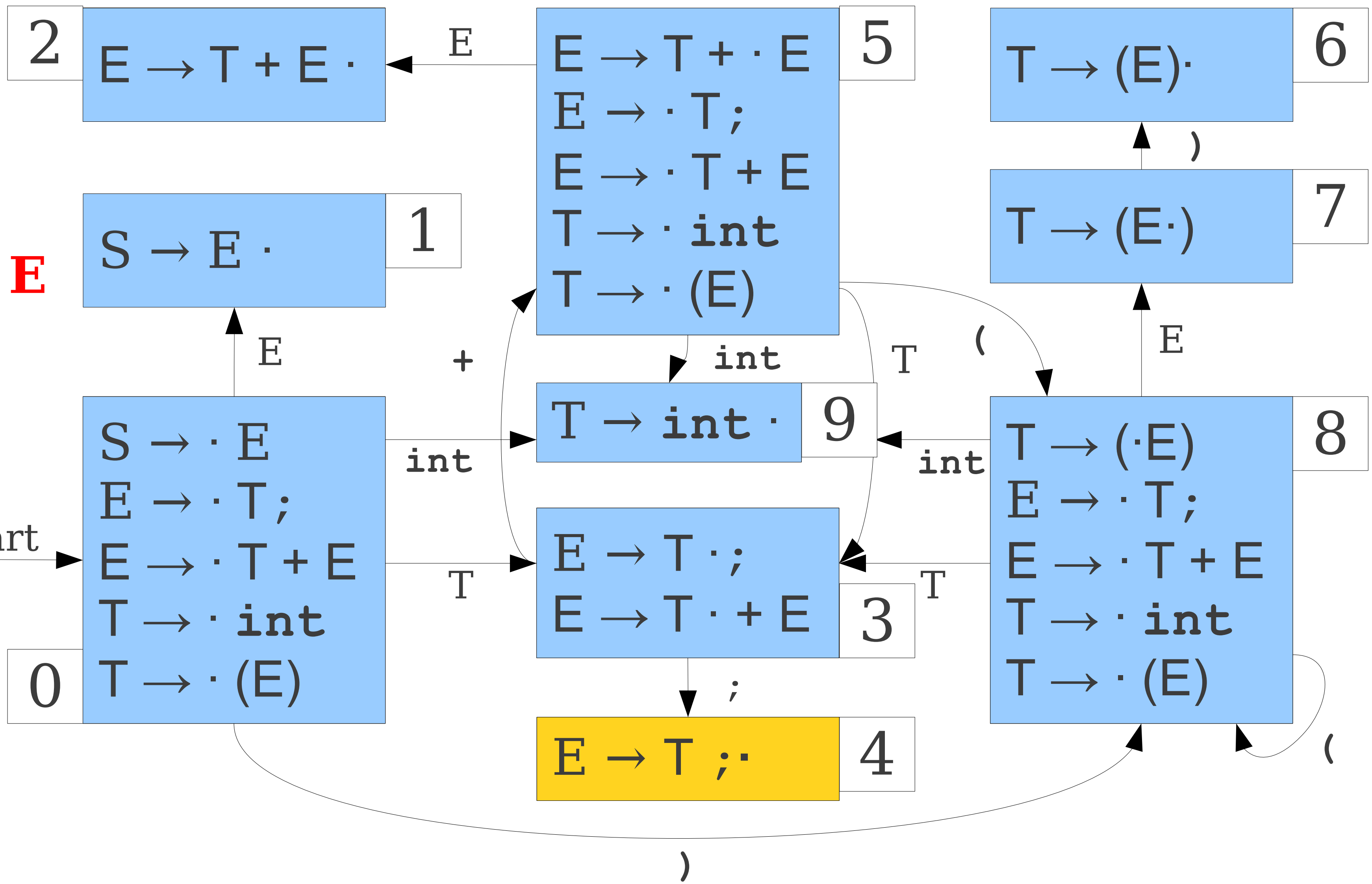
LR(0) Parsing

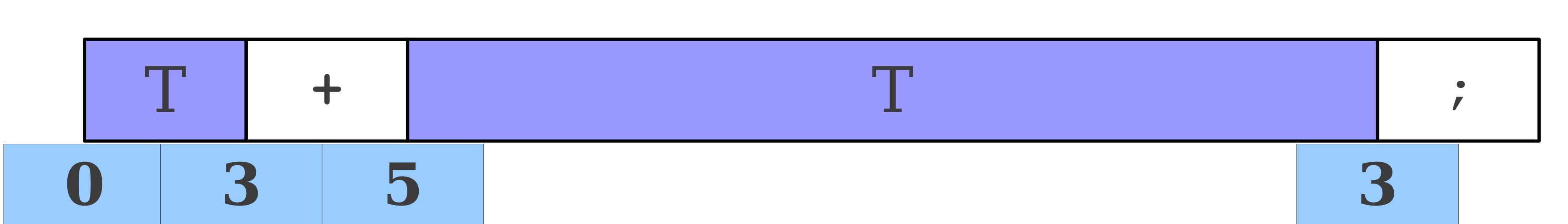
$S \to E$
$E \to T;$
$E \to T + E$
$T \to$ int
$T \to (E)$



LR(0) Parsing

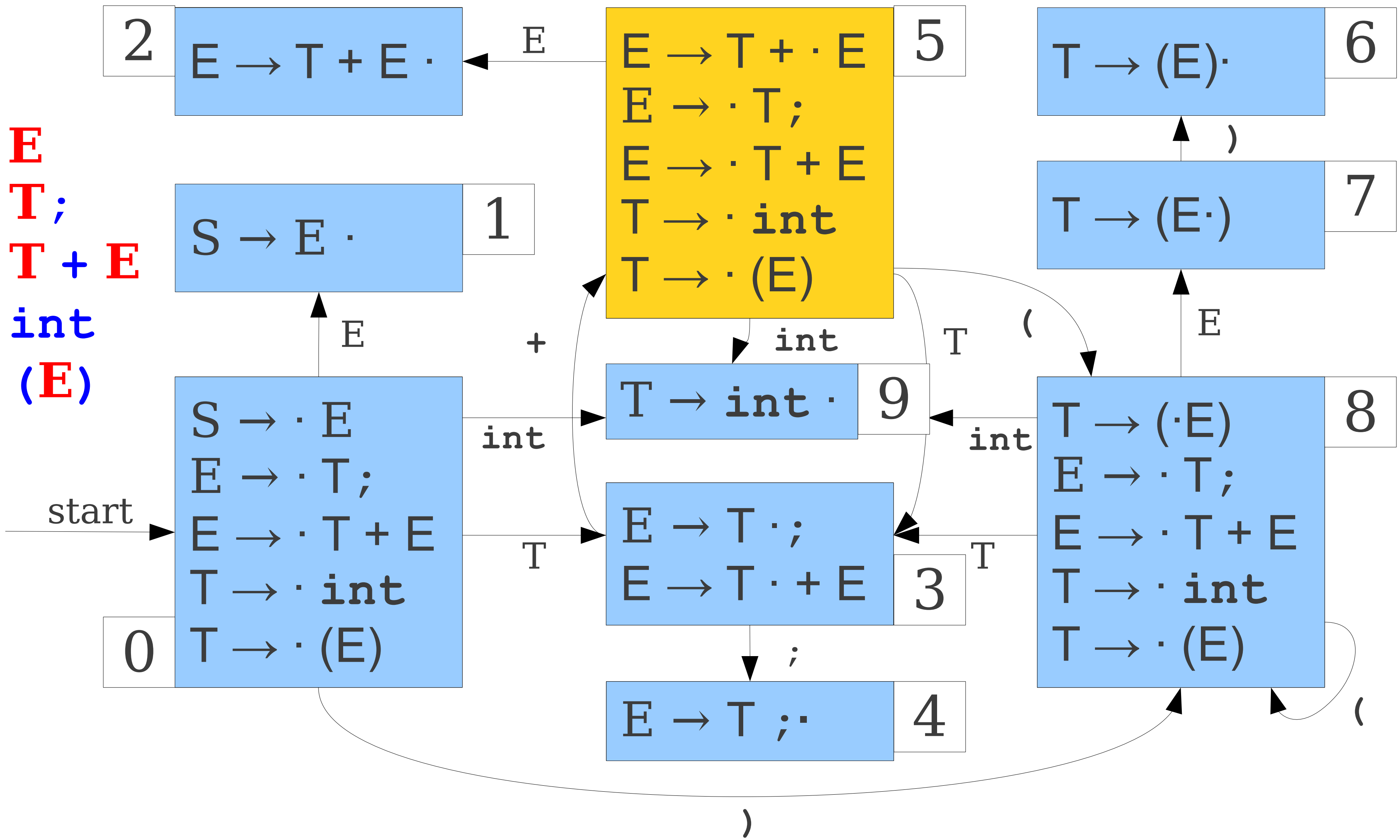
S → **E**
E → **T** ;
E → **T** + **E**
T → **int**
T → **(E)**

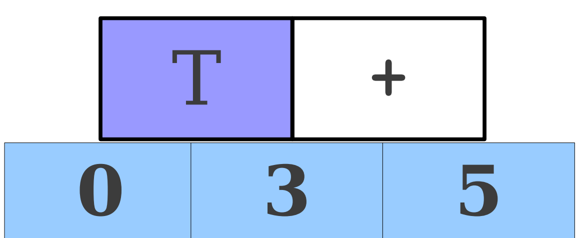




LR(0) Parsing

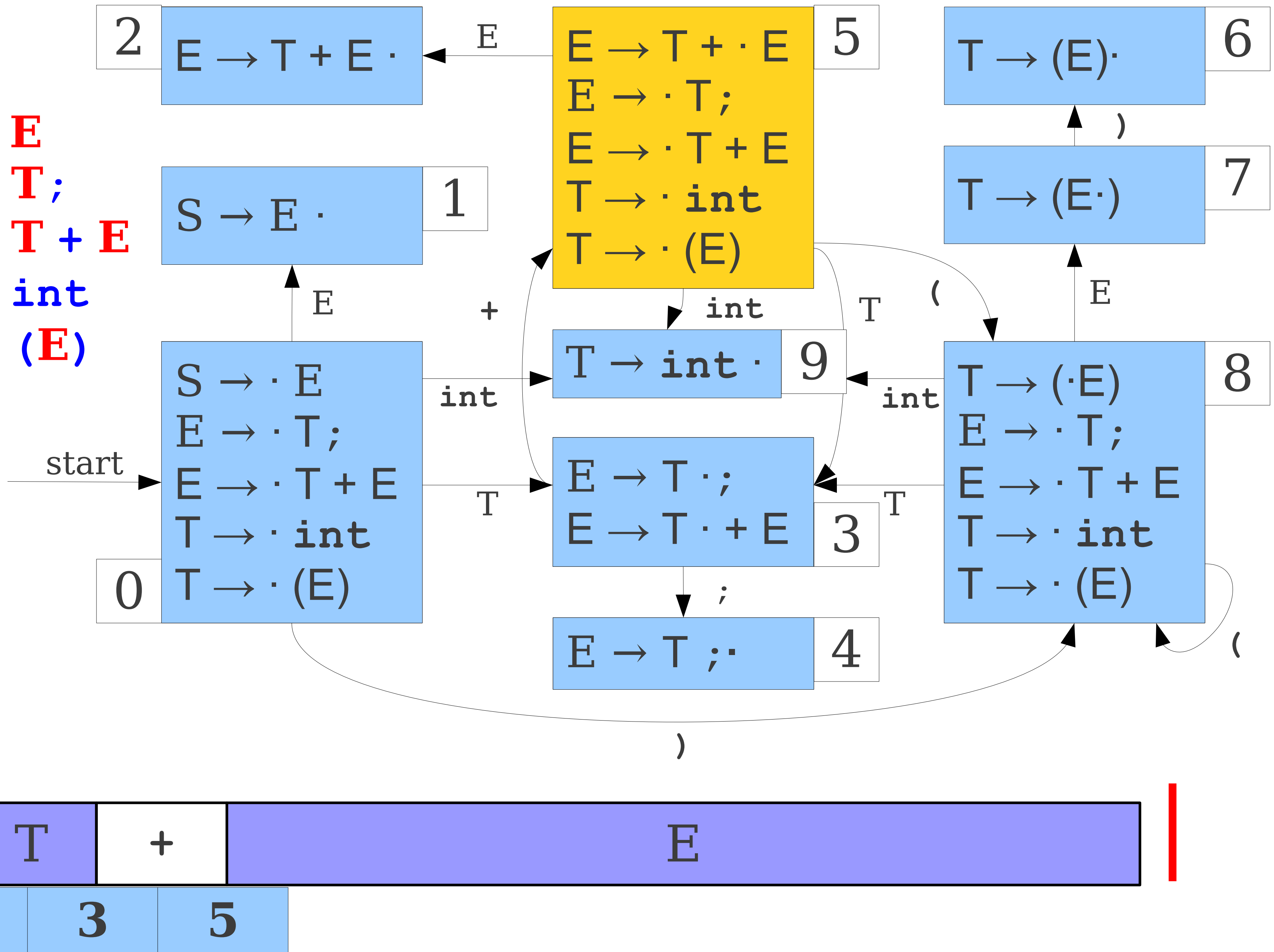
S → **E**
E → **T** ;
E → **T** + **E**
T → `int`
T → (**E**)





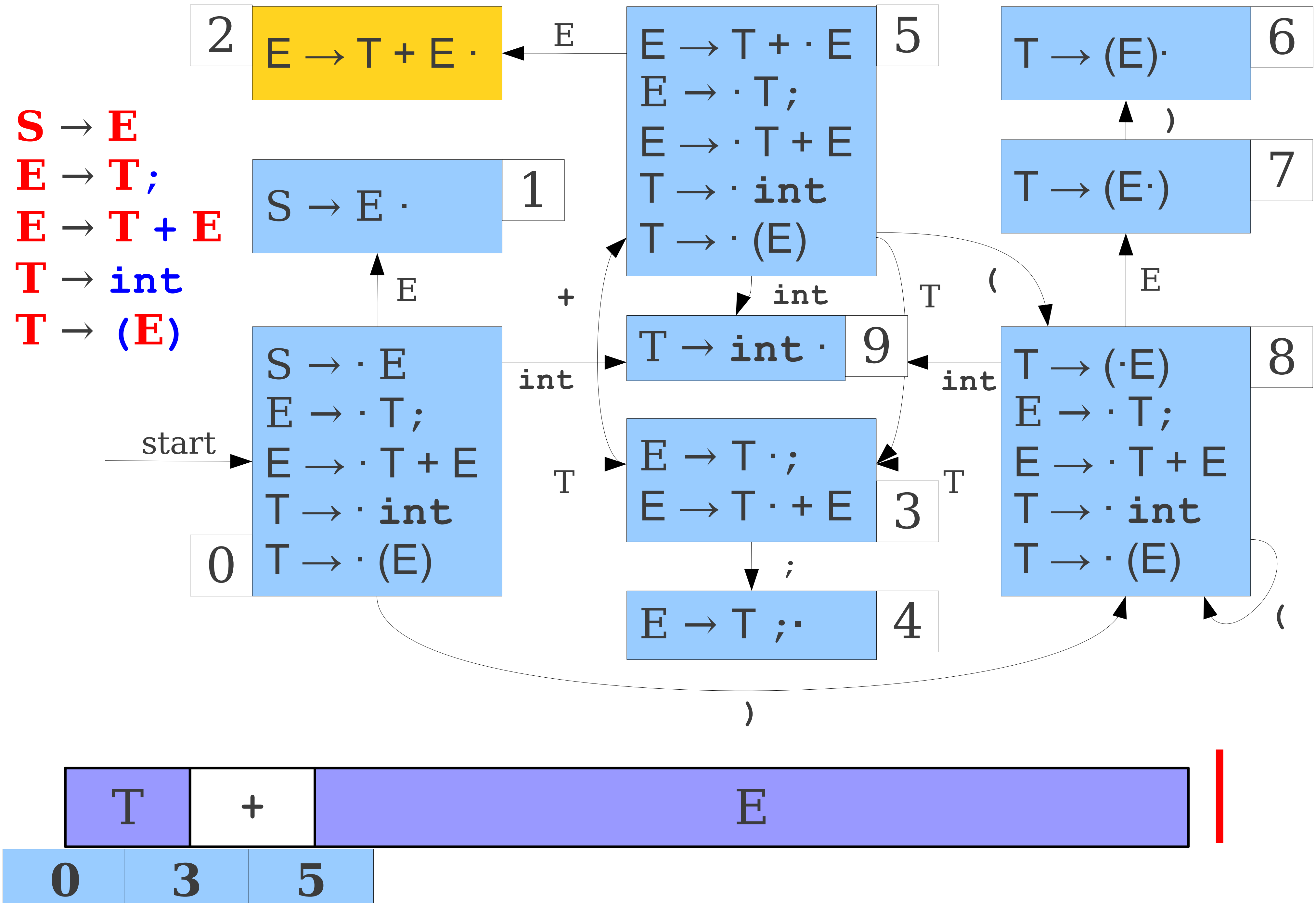
LR(0) Parsing

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



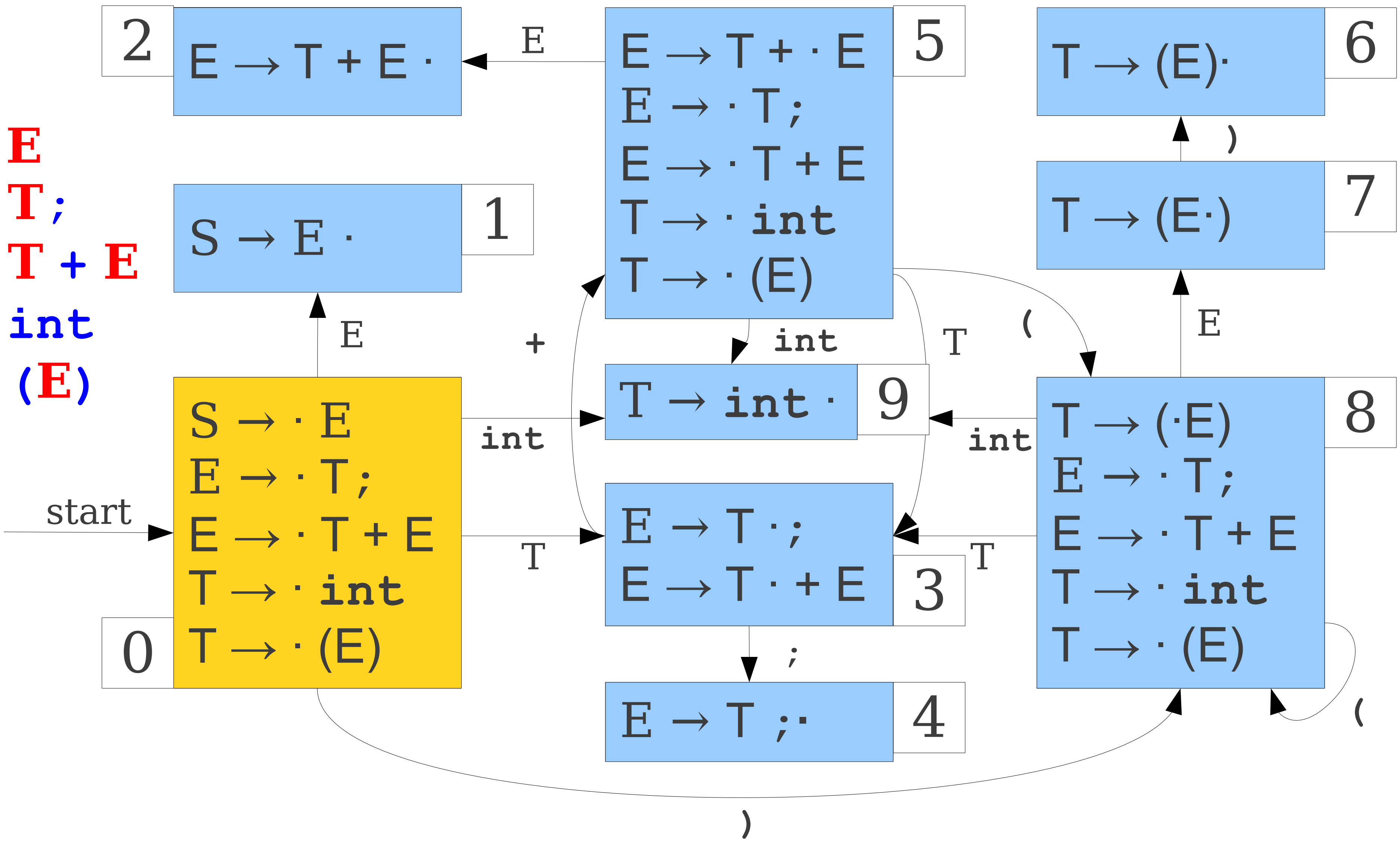
LR(0) Parsing

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



LR(0) Parsing

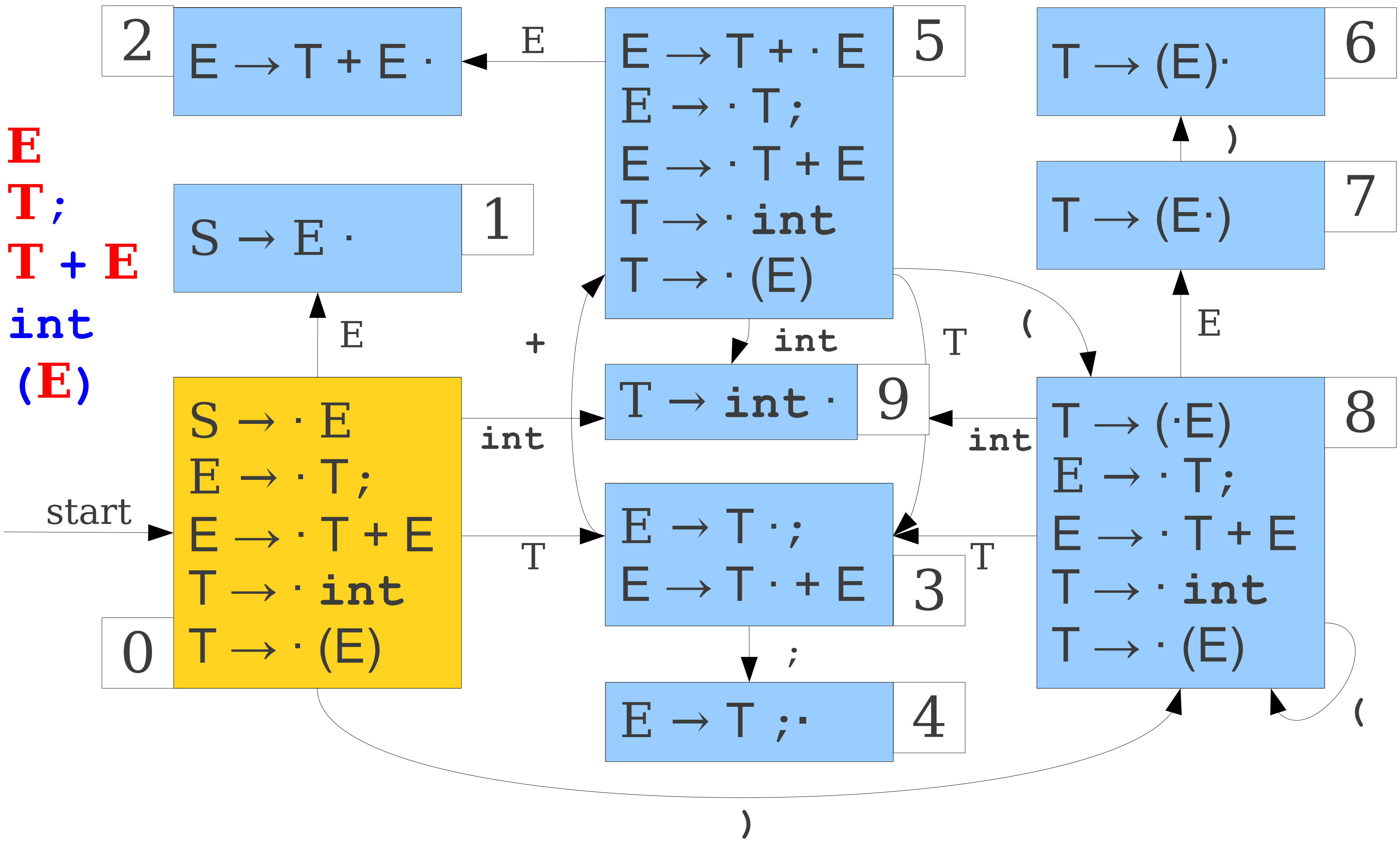
S → **E**
E → **T** ;
E → **T** + **E**
T → `int`
T → (**E**)

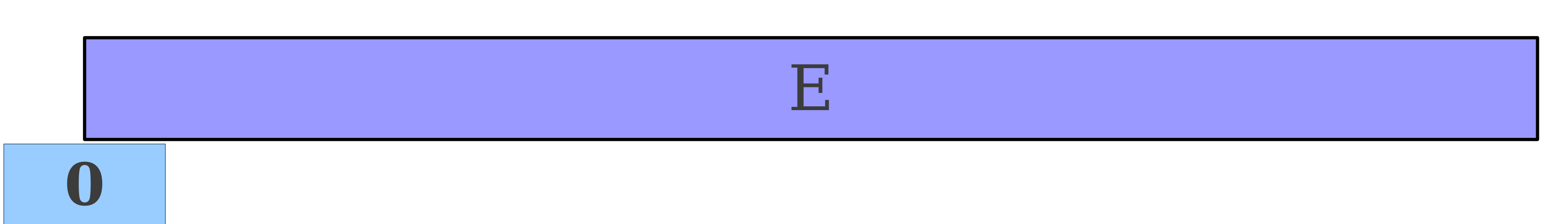


0

LR(0) Parsing

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





LR(0) Parsing

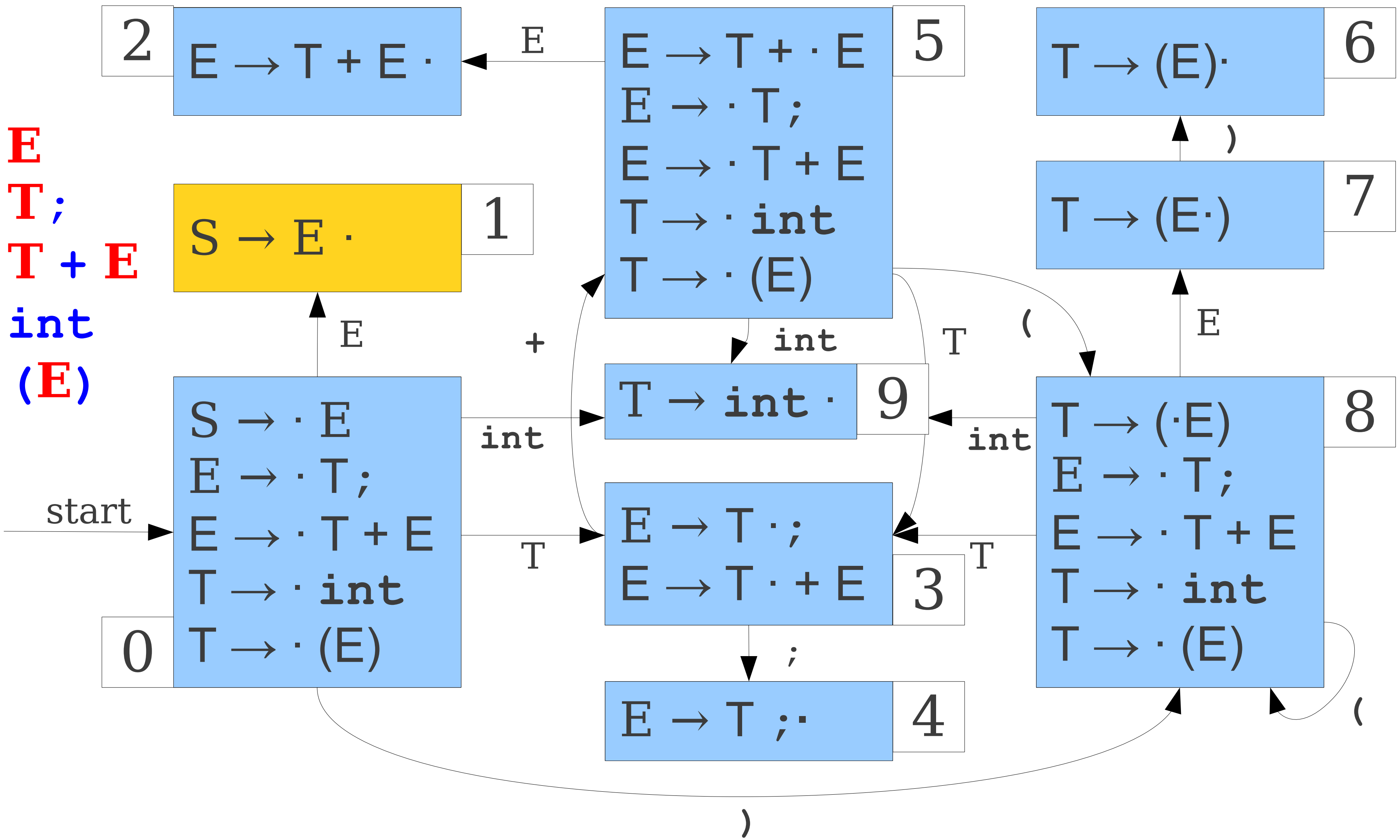
$S \to E$

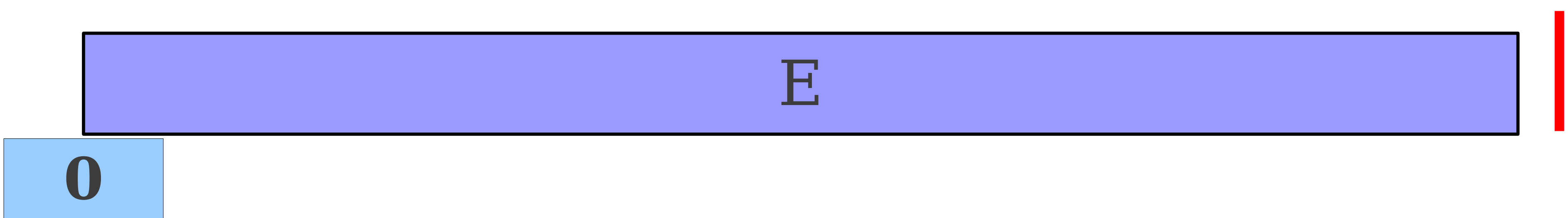
$E \to T;$

$E \to T + E$

$T \to$ **int**

$T \to (E)$



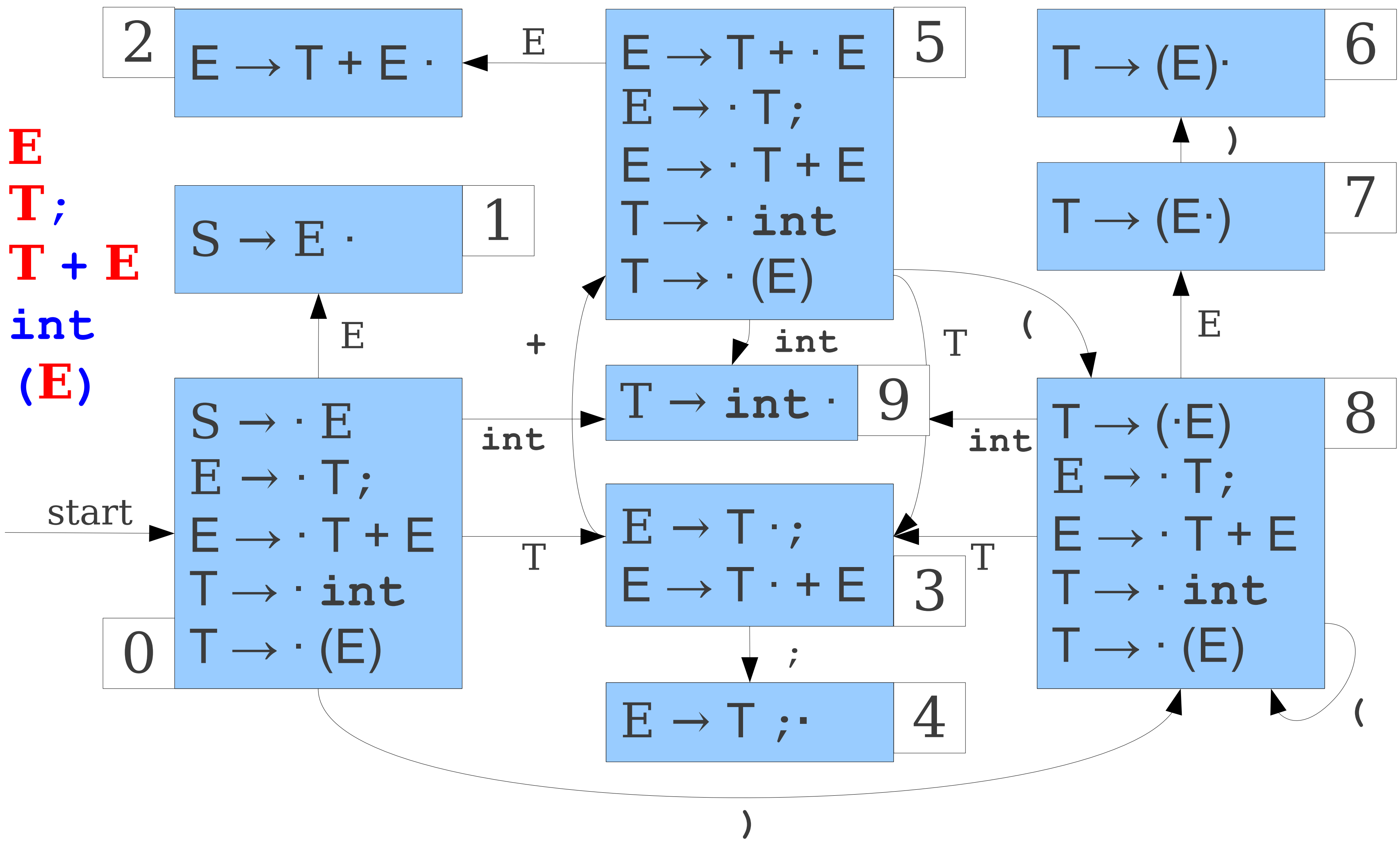


Representing the Automaton

- 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** **A** → ***ω***, which performs reduction **A** → ***ω***.
  - Any state of the form **A** → ***ω*** · 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**

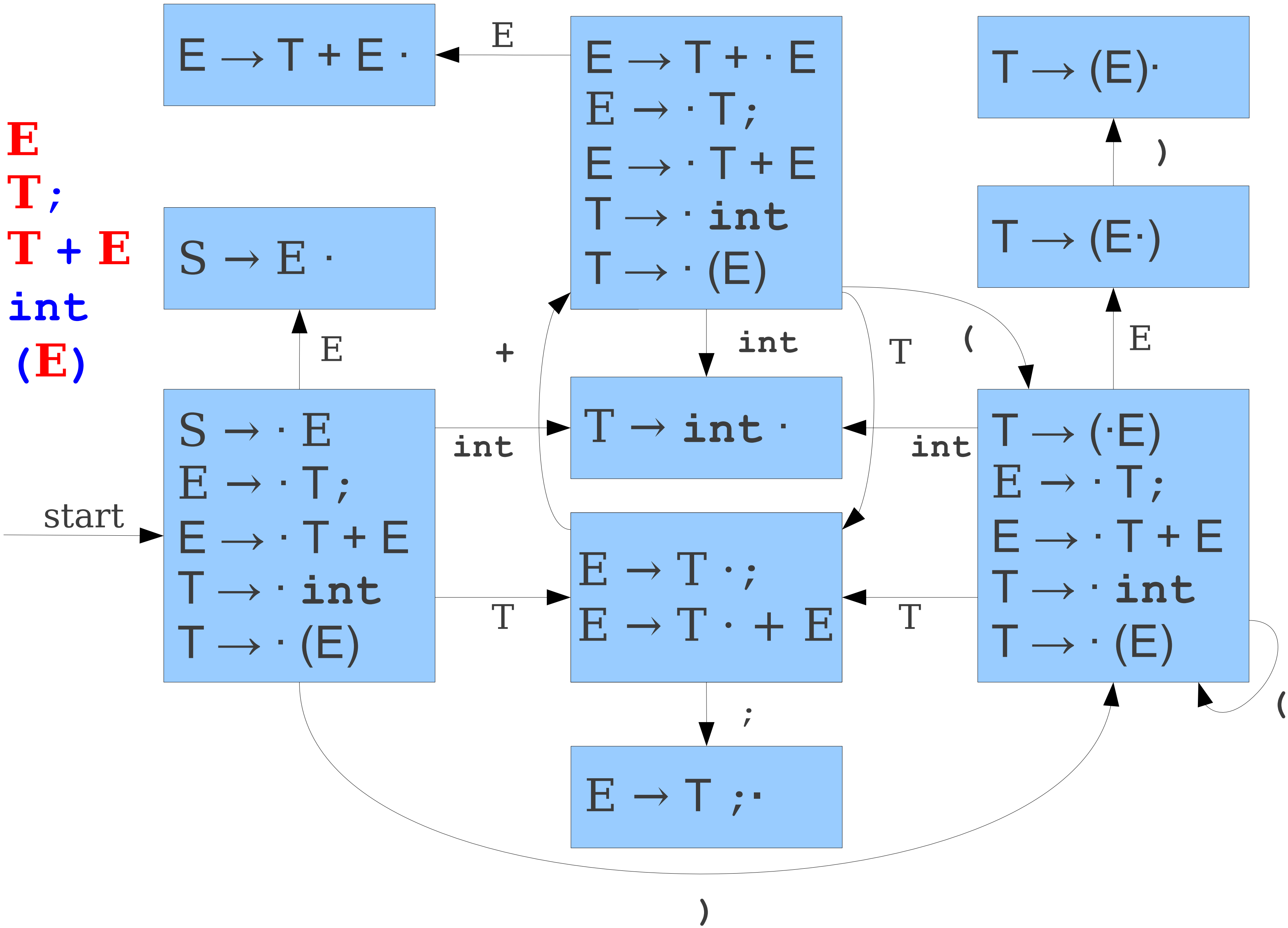
The LR(0) Algorithm

- 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** → ***ω***:
    - Remove |***ω***| 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)

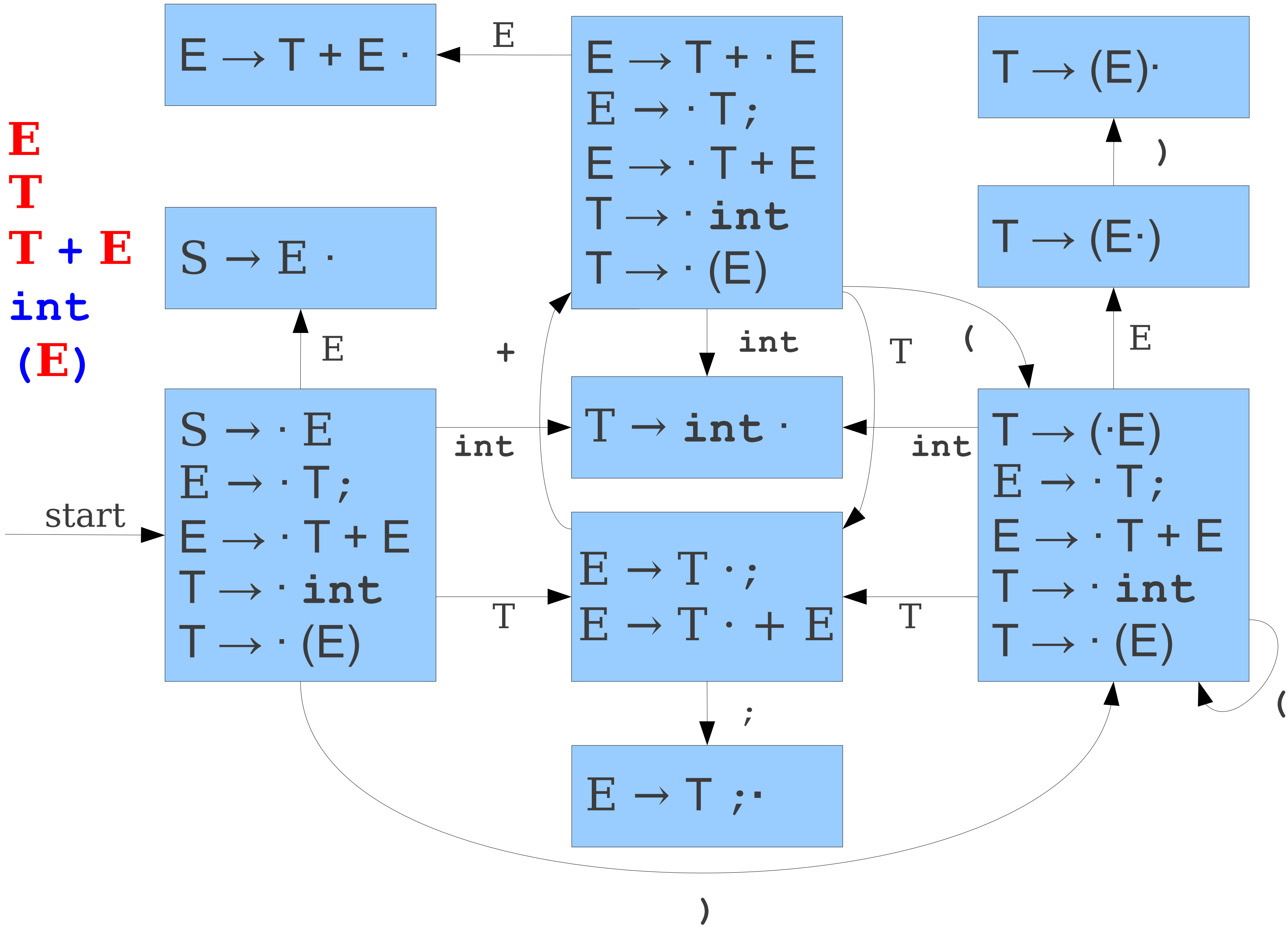
A Non-LR(0) Grammar

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



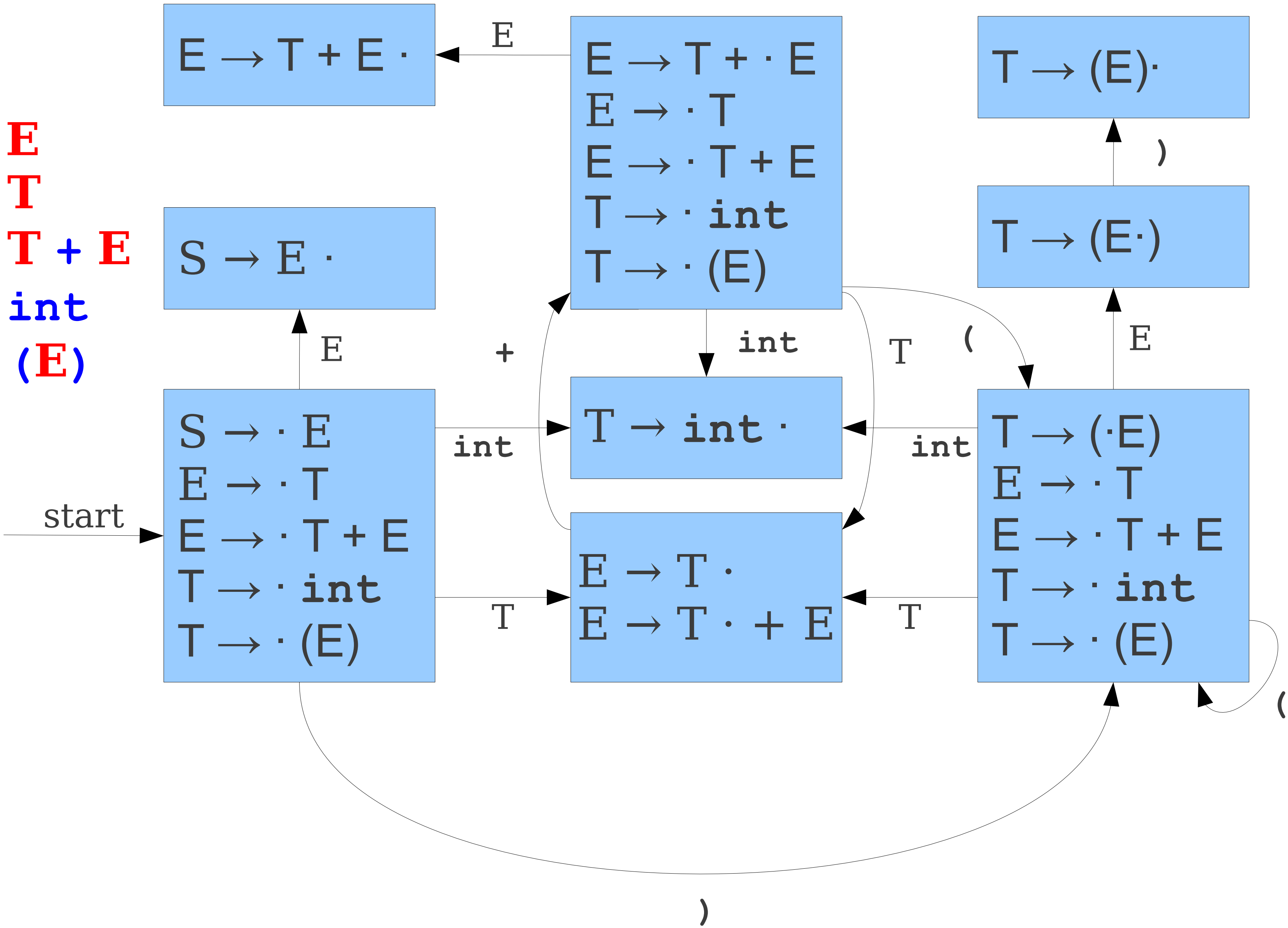
A Non-LR(0) Grammar

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



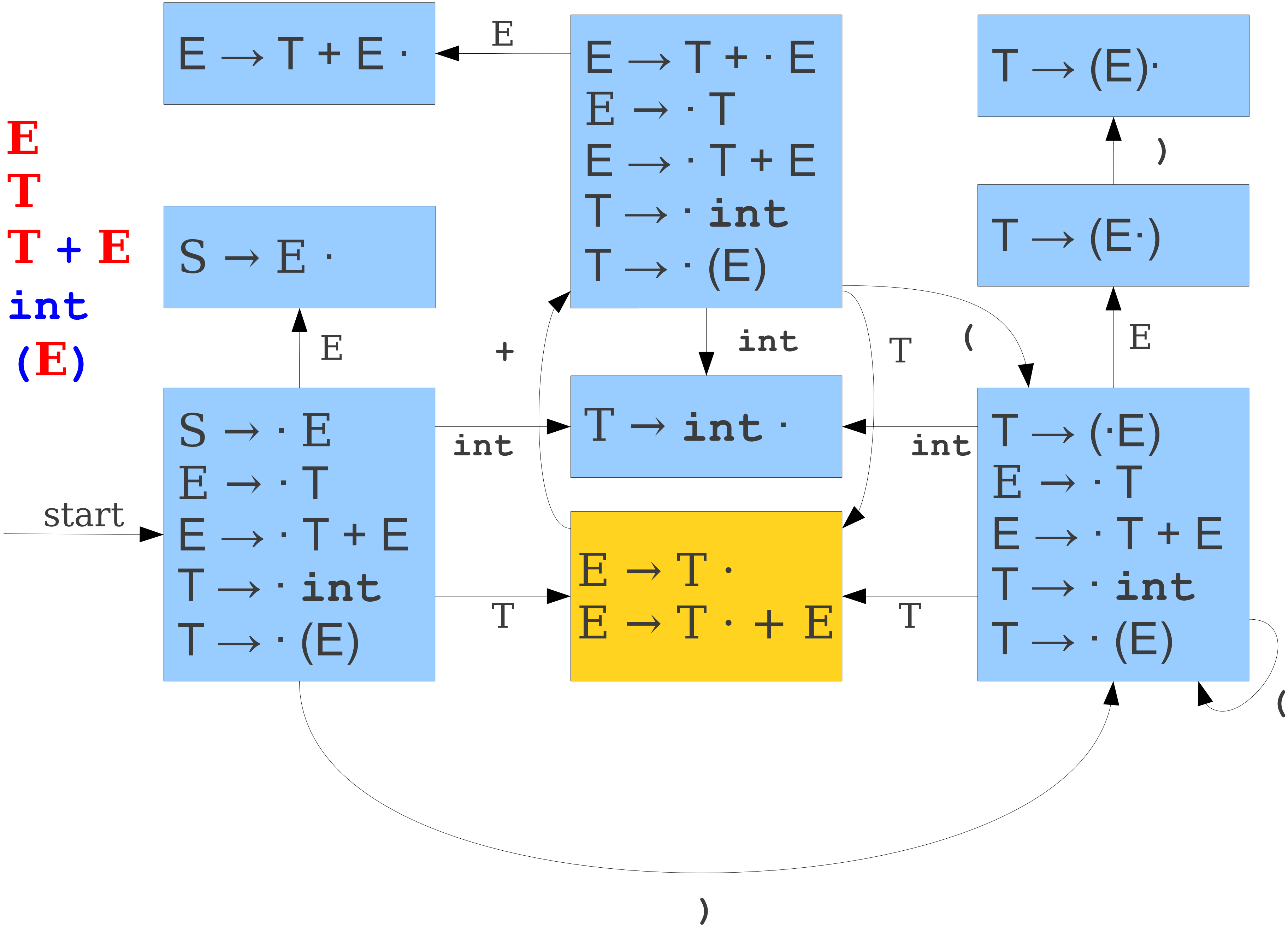
A Non-LR(0) Grammar

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)

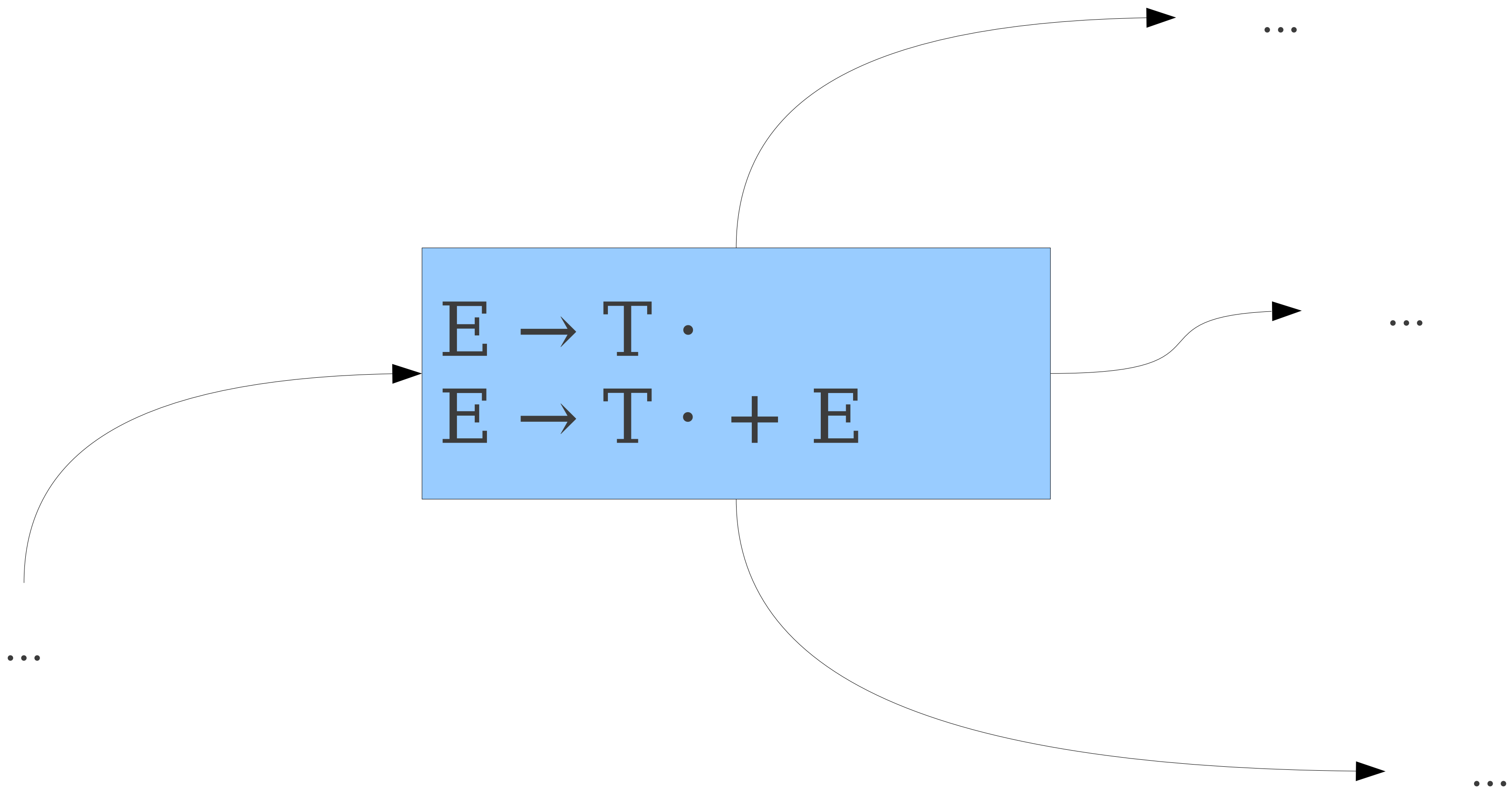


LR Conflicts

- 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?

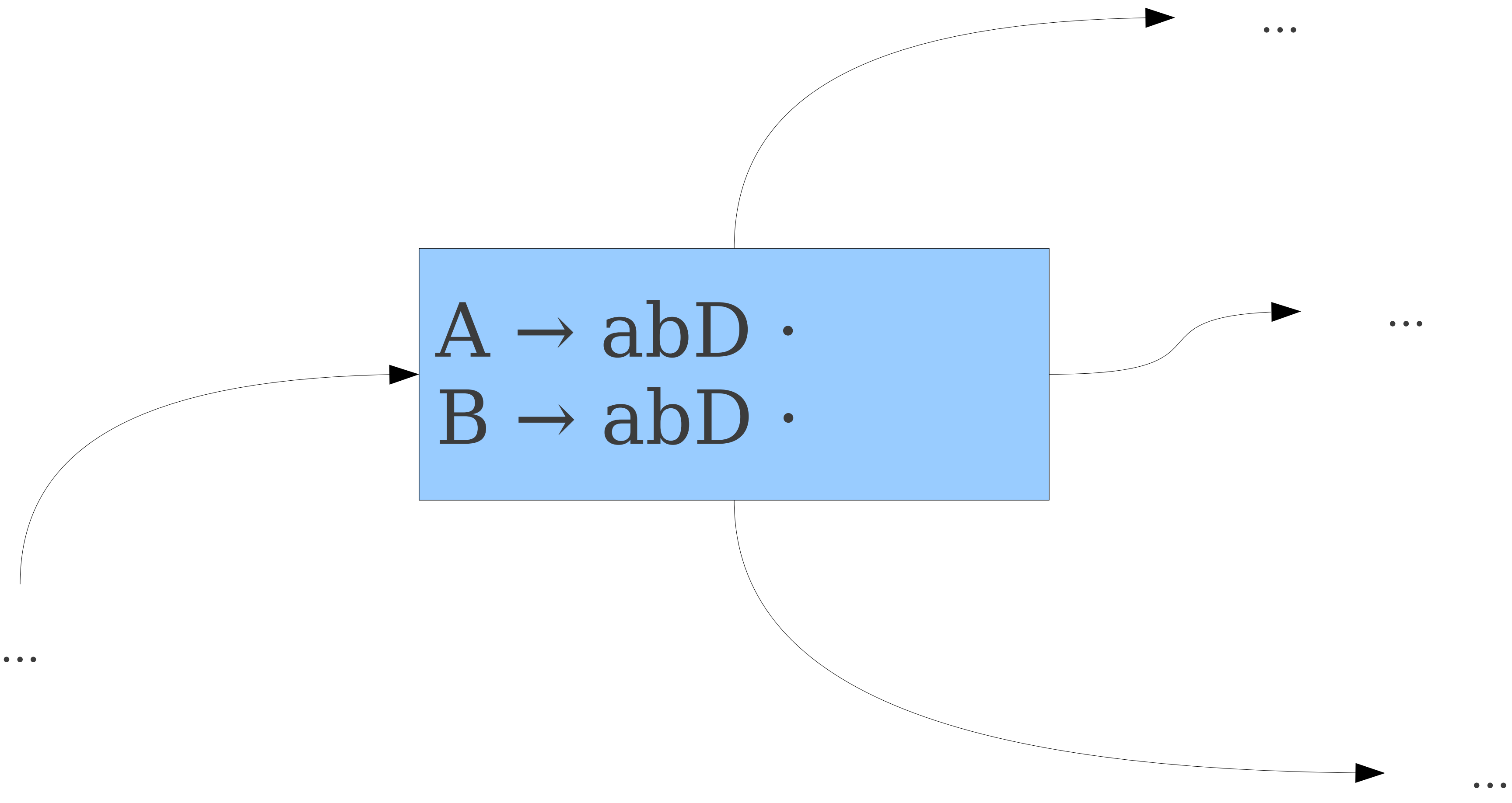
What error is this?

What error is this?



What about this?

What about this?



And what about this?

And 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.

Why LR(0) is Weak

- LR(0) only accepts languages where the handle can be found with no **right context**.
- Our shift/reduce parser only looks to the left of the handle, not to the right.
- How do we exploit the tokens after a possible handle to determine what to do?

A Powerful Parser: **LR(1)**

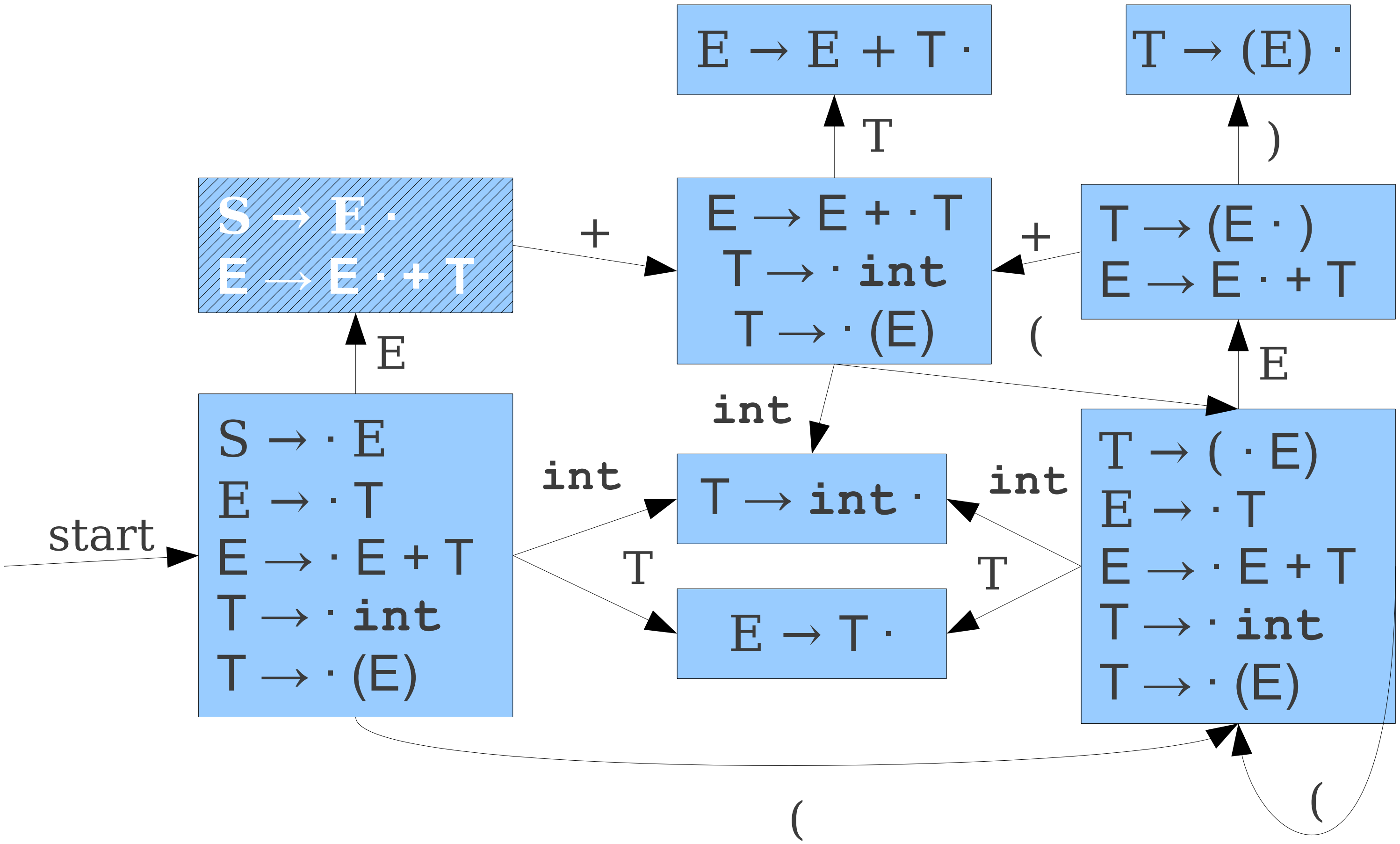
- Bottom-up predictive parsing with
  - **L**: **L**eft-to-right scan
  - **R**: **R**ightmost derivation
  - (**1**): One token lookahead
- *Substantially* more powerful than the other methods we've covered so far (more on that later).
- Tries to more intelligently find handles by using a lookahead token at each step.

LR(1) Parsing: The Intuition

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

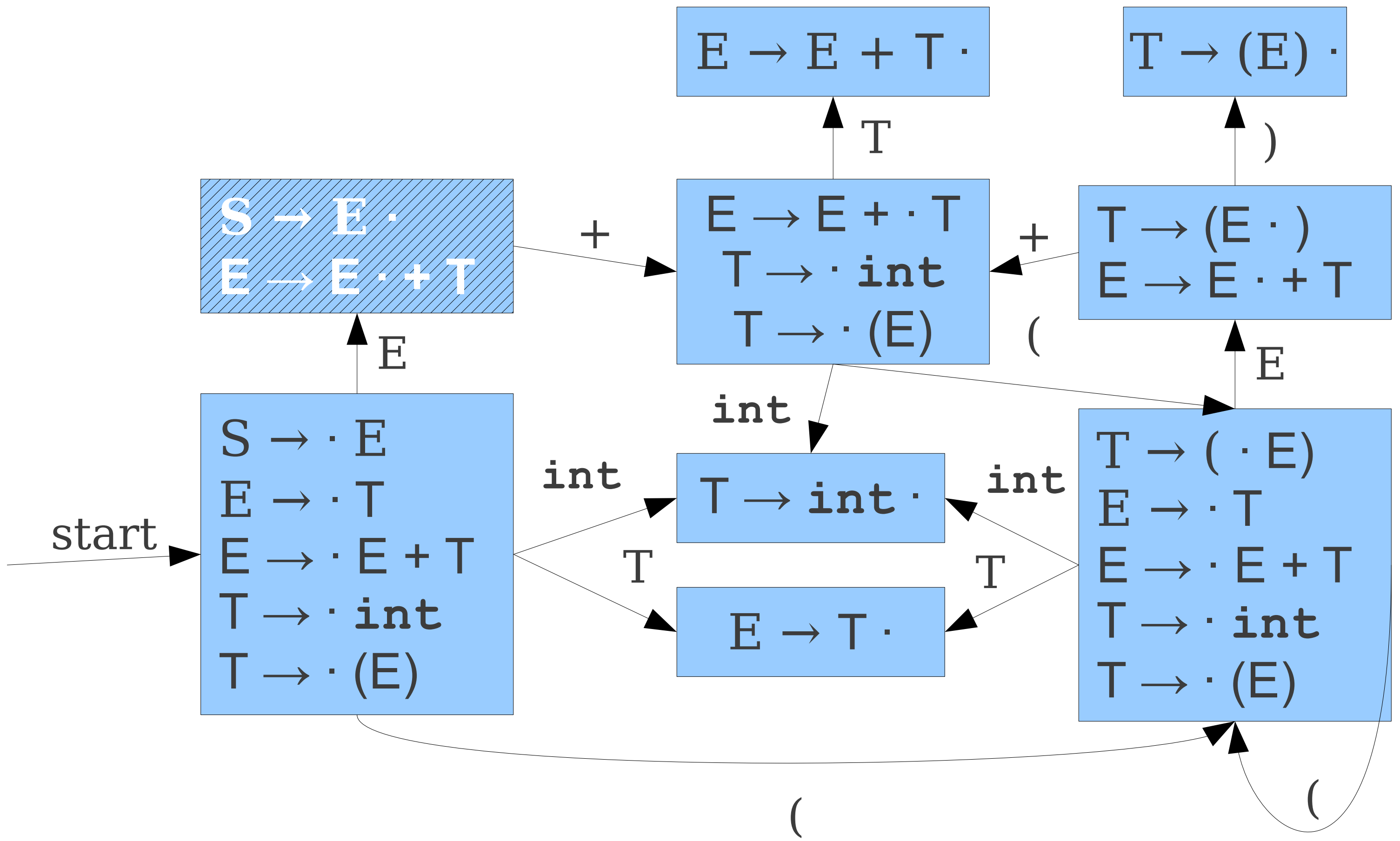
LR(1) Parsing: The Intuition

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



LR(1) Parsing: The Intuition

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

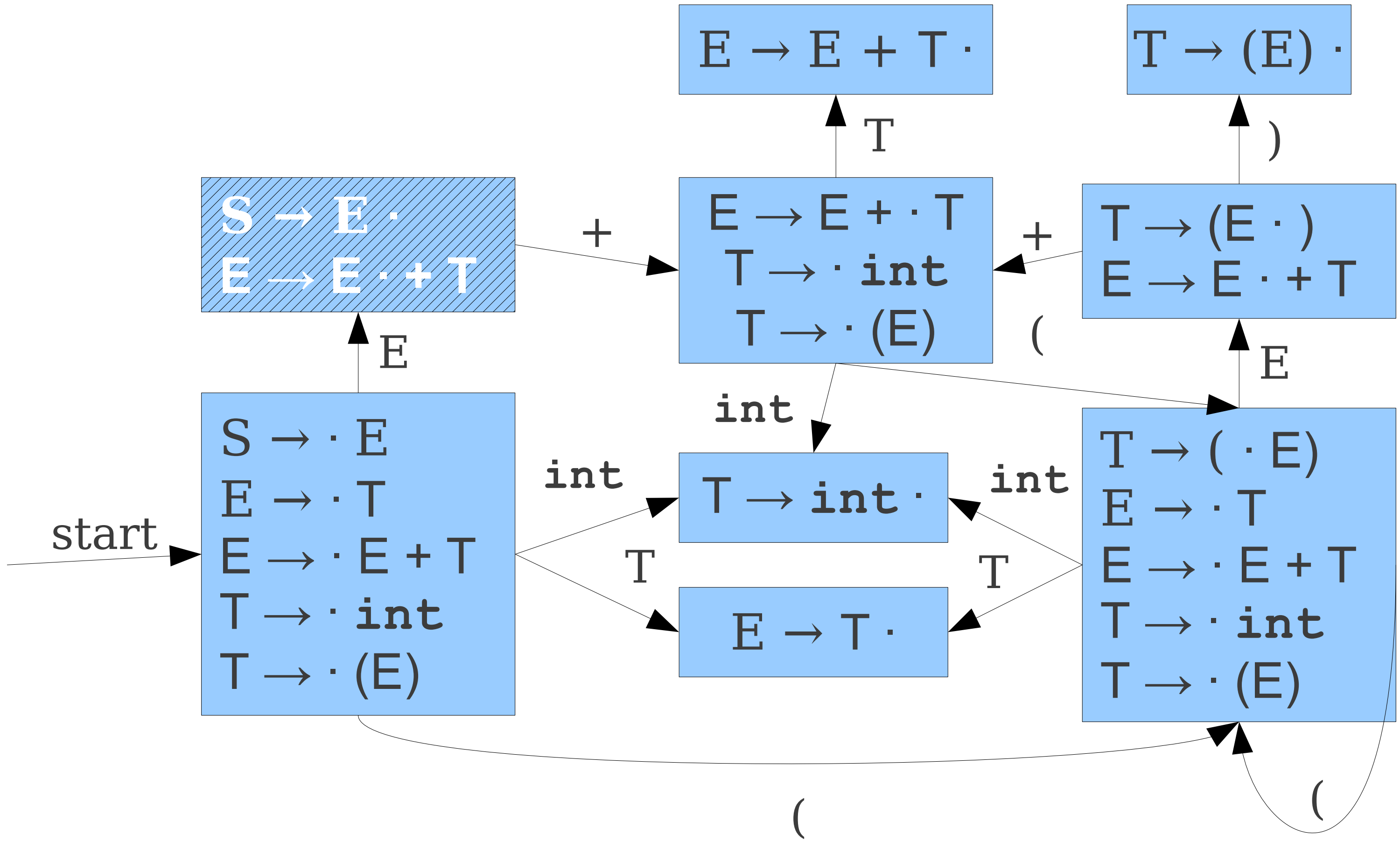


int	+	(	int	+	int	+	int	)	$

LR(1) Parsing: The Intuition

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

S → · E | $

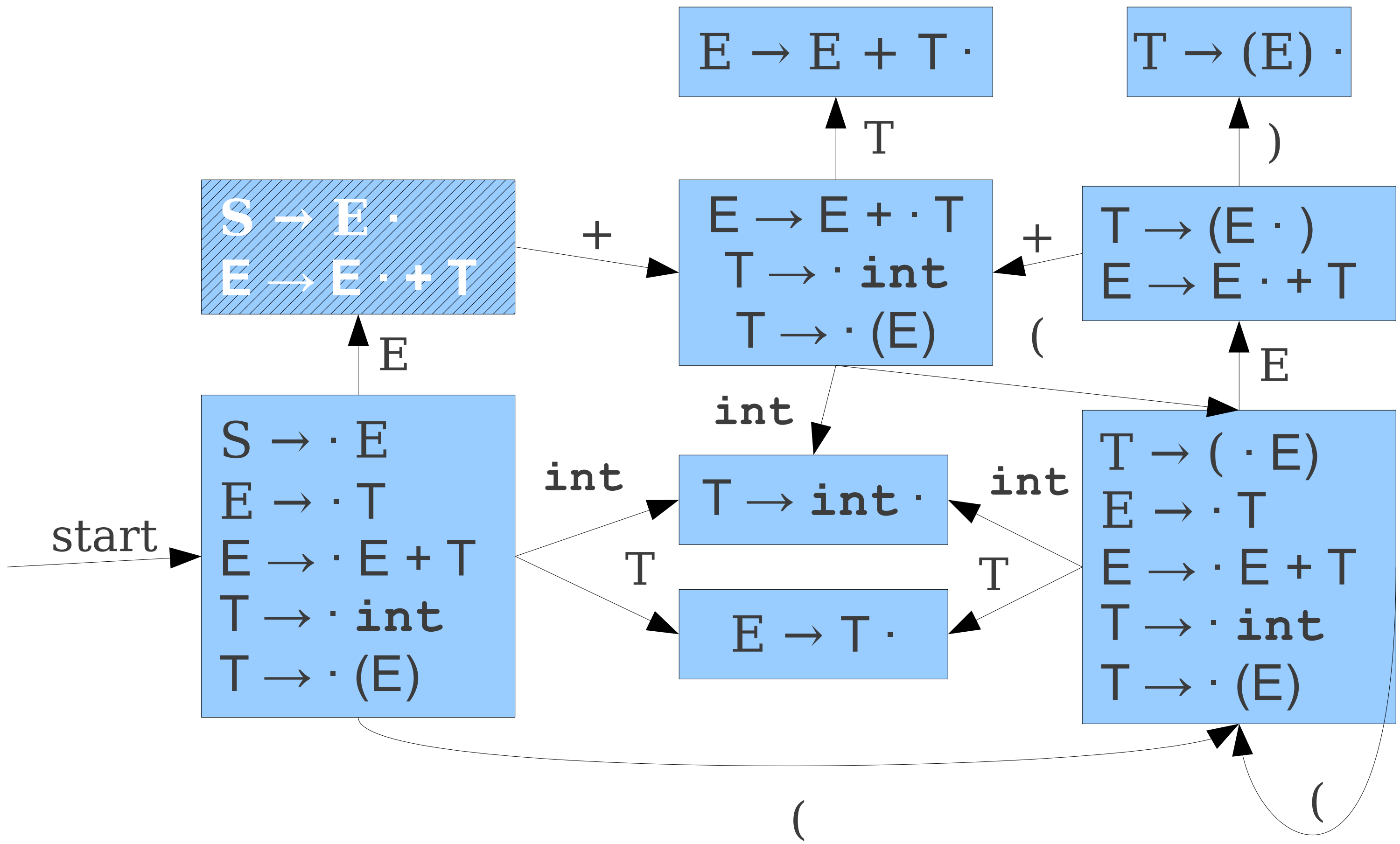


int	+	(	int	+	int	+	int	)	$

LR(1) Parsing: The Intuition

S → E
E → T
E → E + T
T → int
T → (E)

S → · E	$
E → · E + T	$

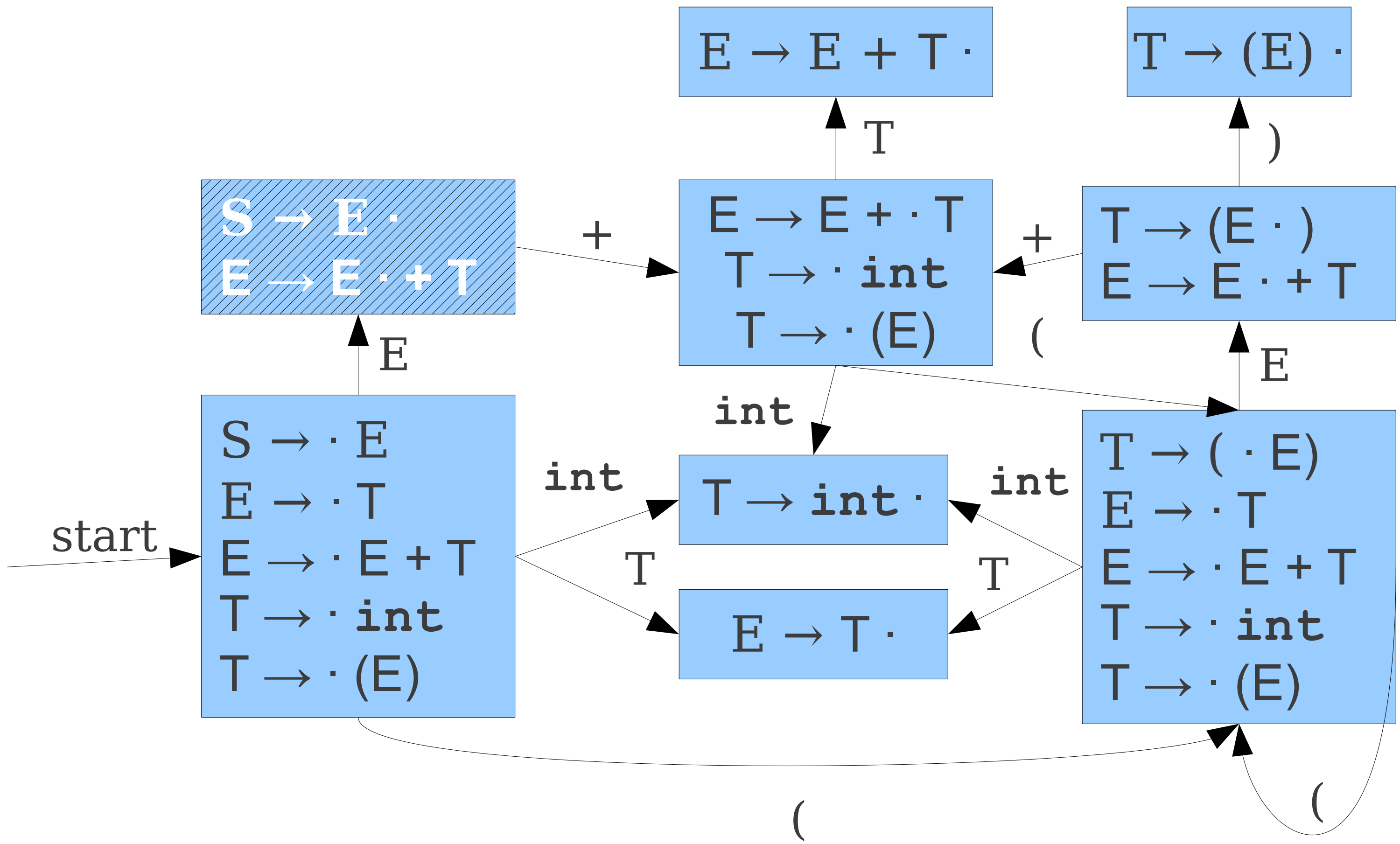


int	+	(	int	+	int	+	int	)	$

LR(1) Parsing: The Intuition

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

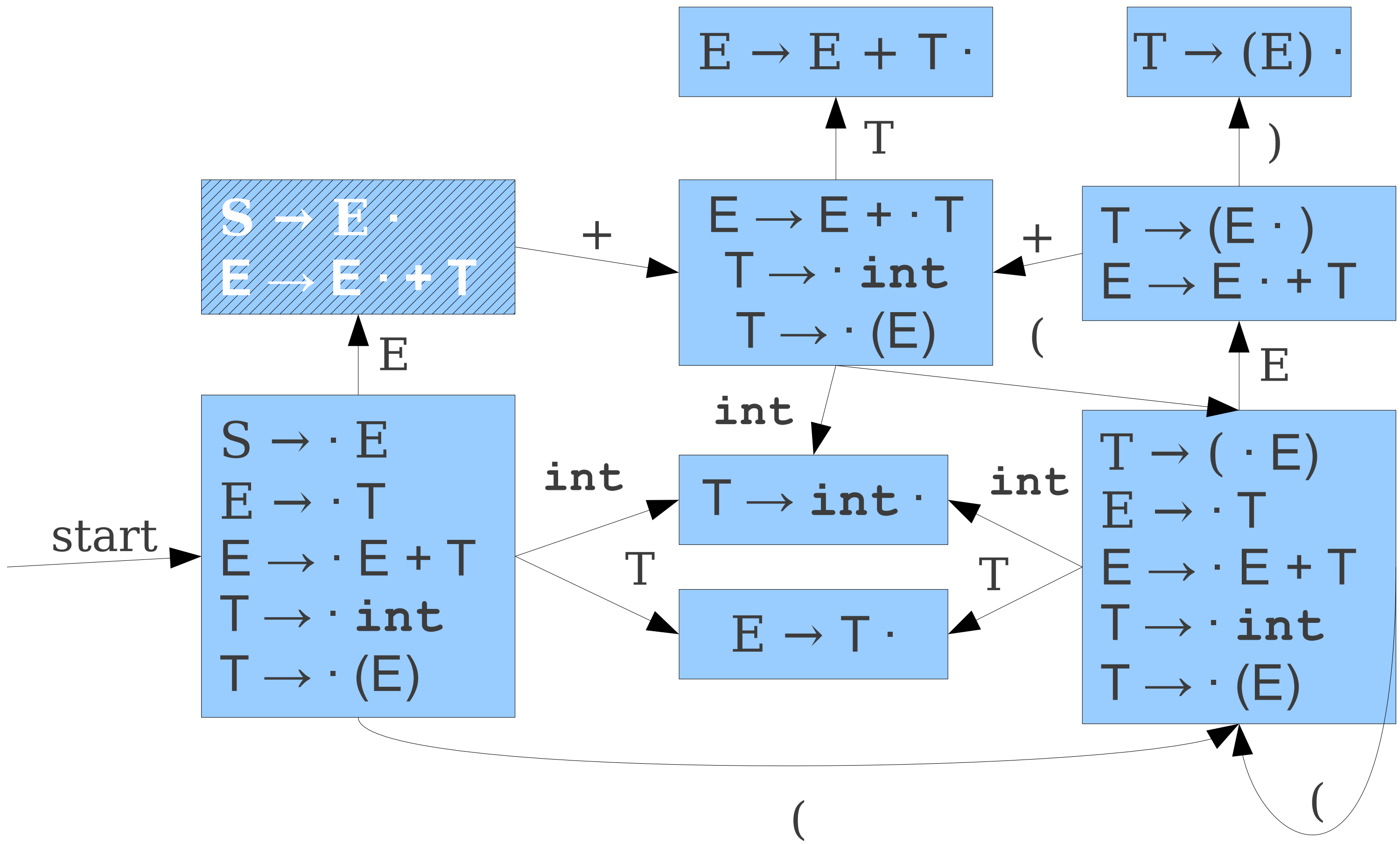
S → · E	$
E → · E + T	$
E → · T	+



int	+	(	int	+	int	+	int	)	$

LR(1) Parsing: The Intuition

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



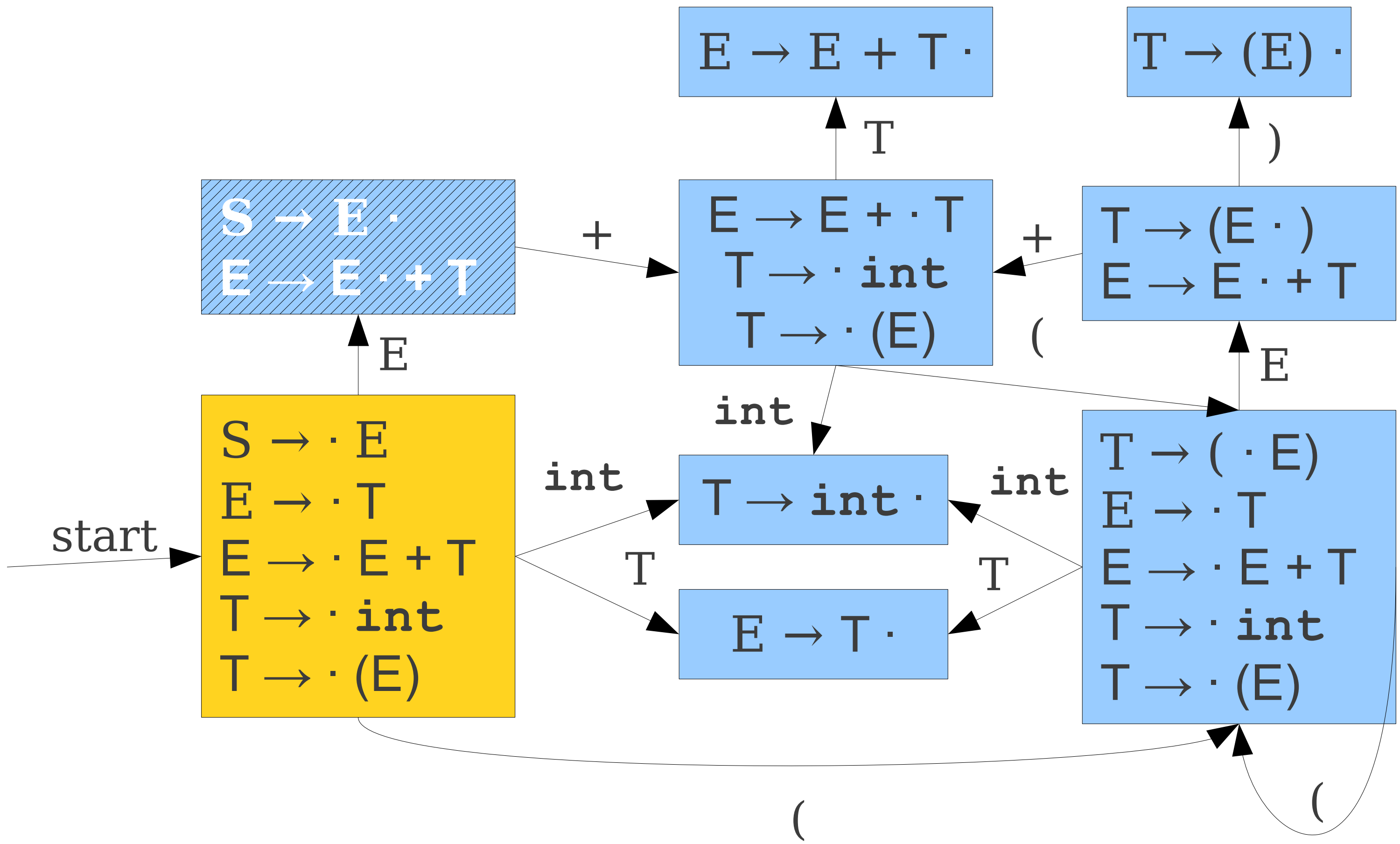
S → · E	$
E → · E + T	$
E → · T	+
T → · int	+

int	+	(	int	+	int	+	int	)	$

LR(1) Parsing: The Intuition

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

S → · E	$
E → · E + T	$
E → · T	+
T → · int	+

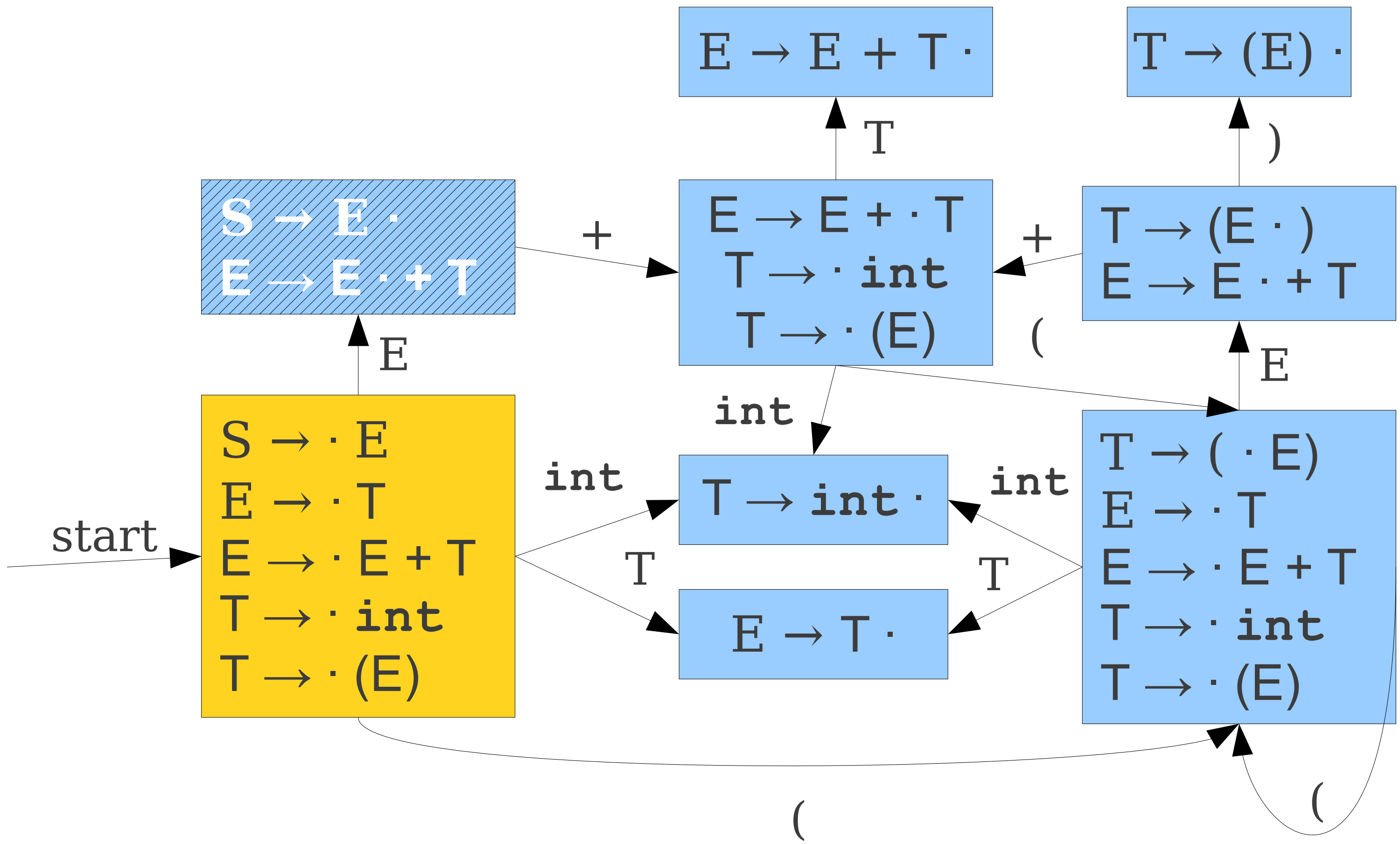


int	+	(	int	+	int	+	int	)	$

LR(1) Parsing: The Intuition

S → E
E → T
E → E + T
T → int
T → (E)

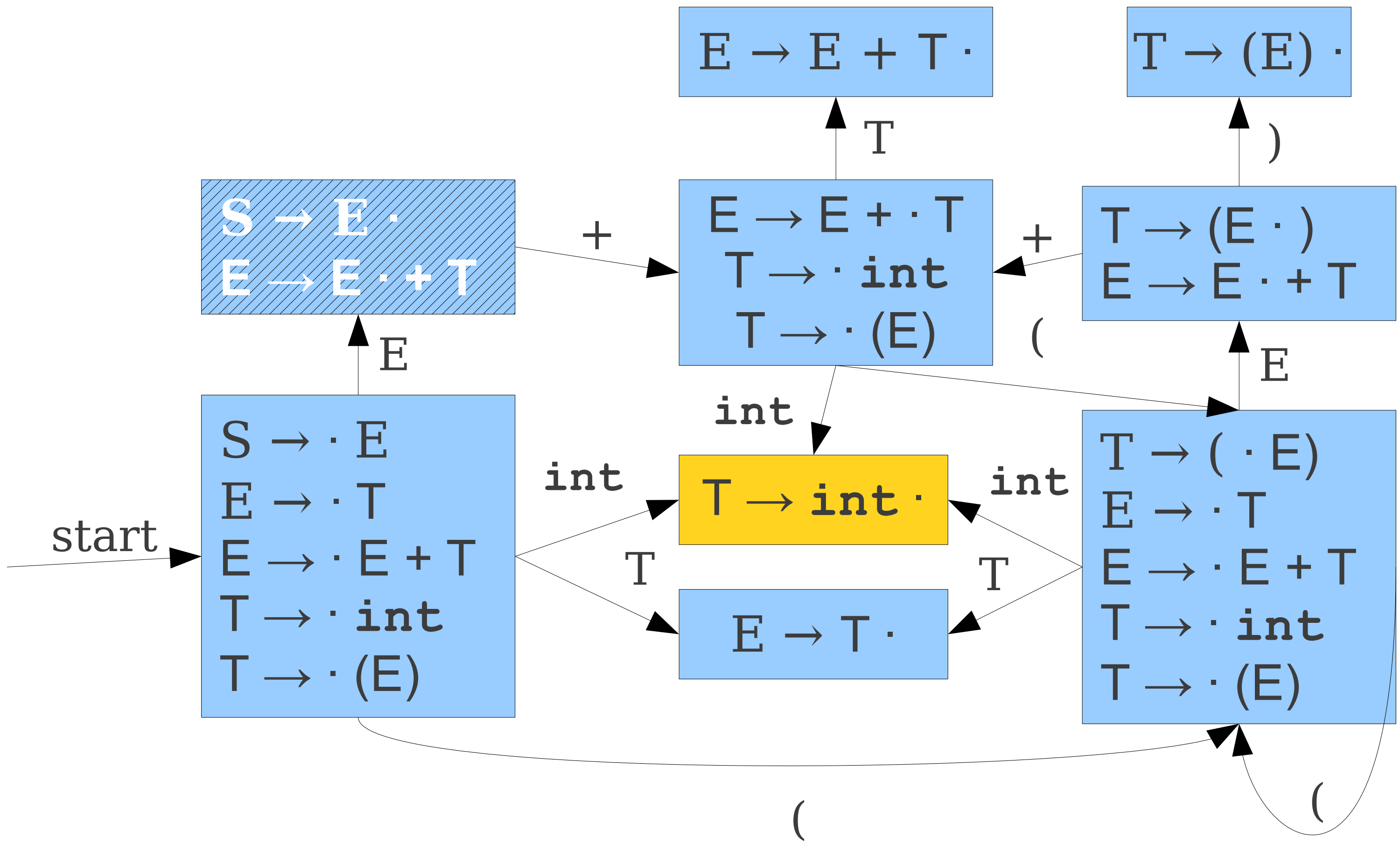
S → · E	$
E → · E + T	$
E → · T	+
T → · int	+



int		+	(	int	+	int	+	int	)	$

LR(1) Parsing: The Intuition

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



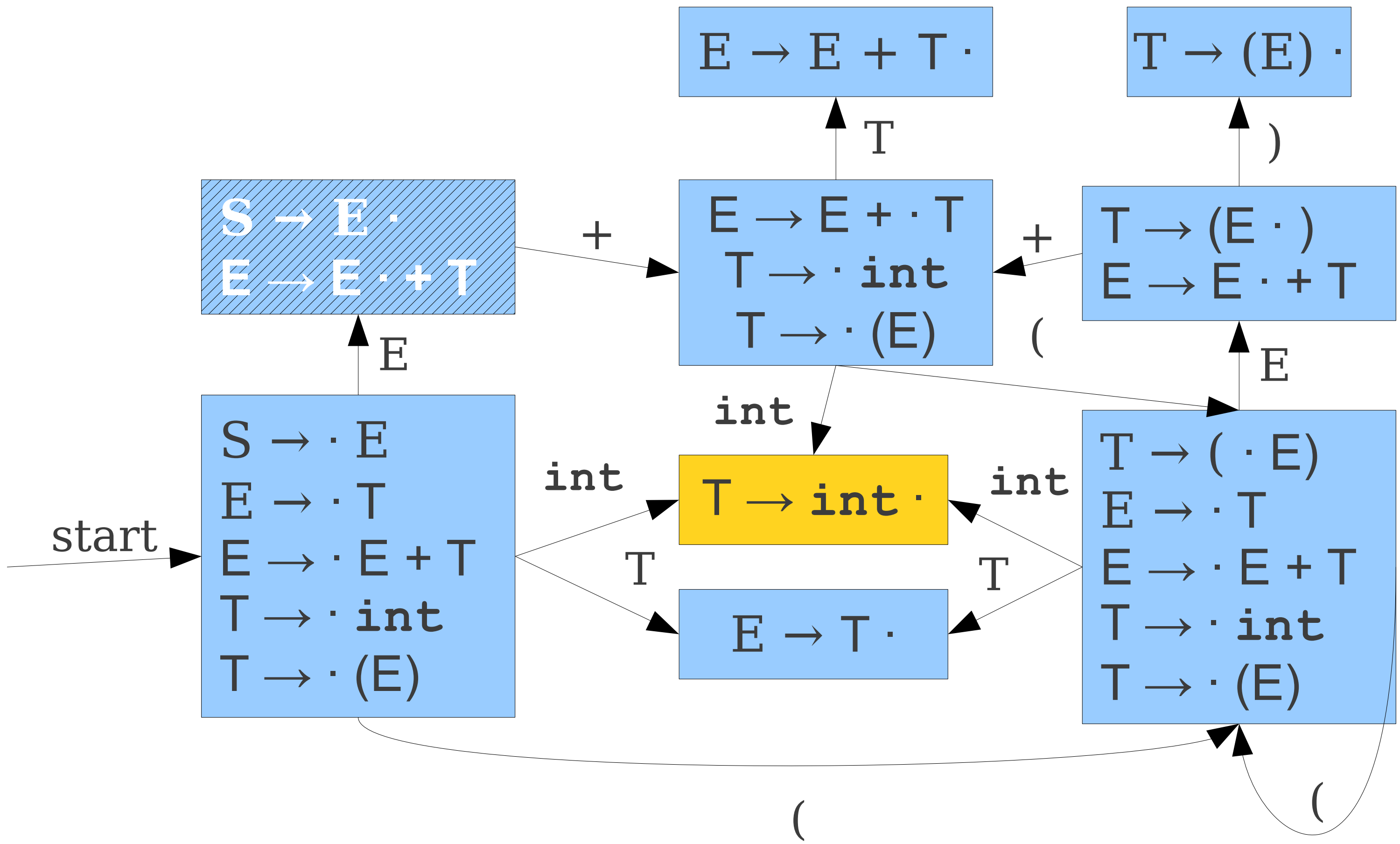
S → · E	$
E → · E + T	$
E → · T	+
T → · int	+

int	|	+	(	int	+	int	+	int	)	$

LR(1) Parsing: The Intuition

S → E
E → T
E → E + T
T → int
T → (E)

S → · E	$
E → · E + T	$
E → · T	+
T → int ·	+

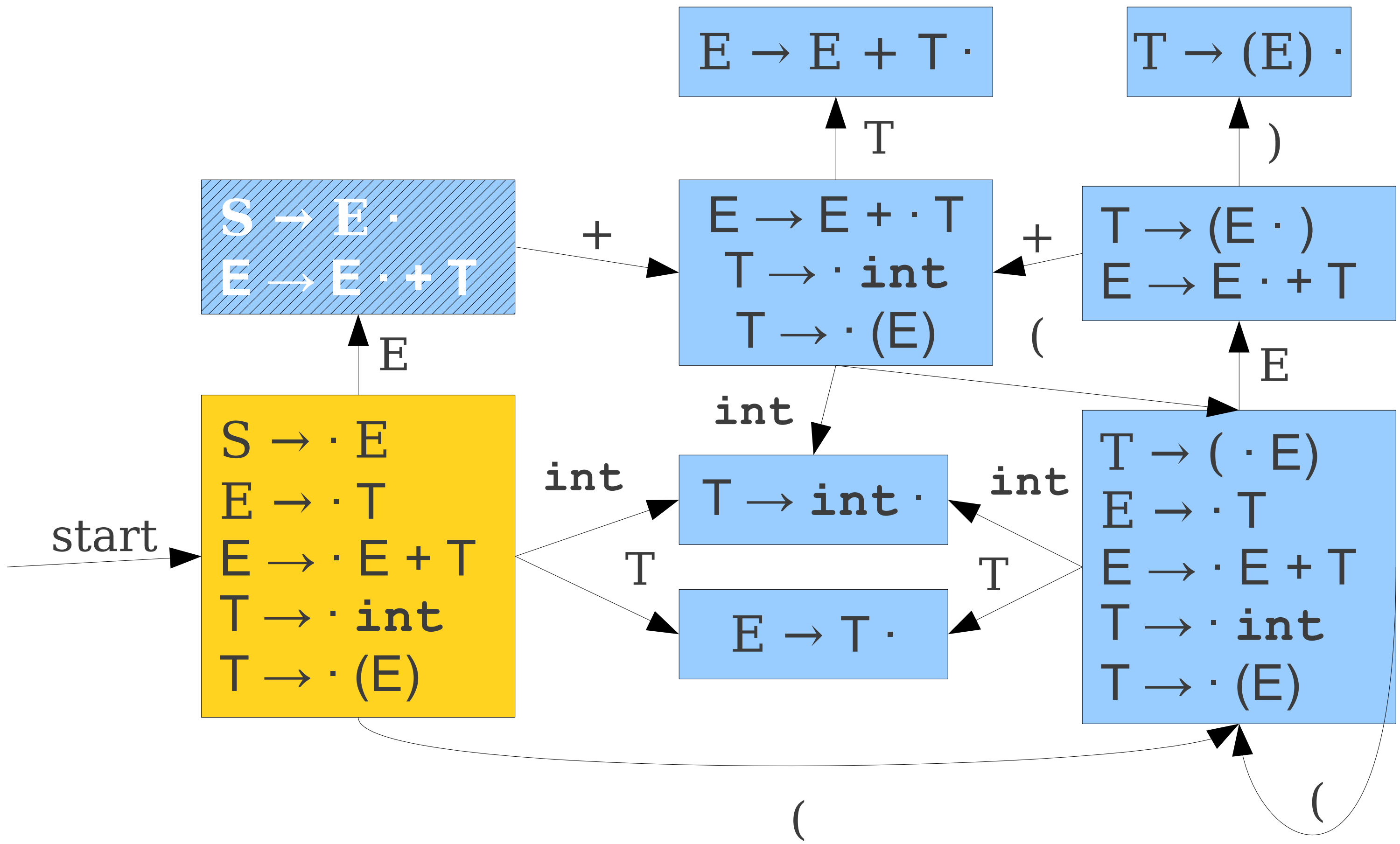


int	|	+	(	int	+	int	+	int	)	$

LR(1) Parsing: The Intuition

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

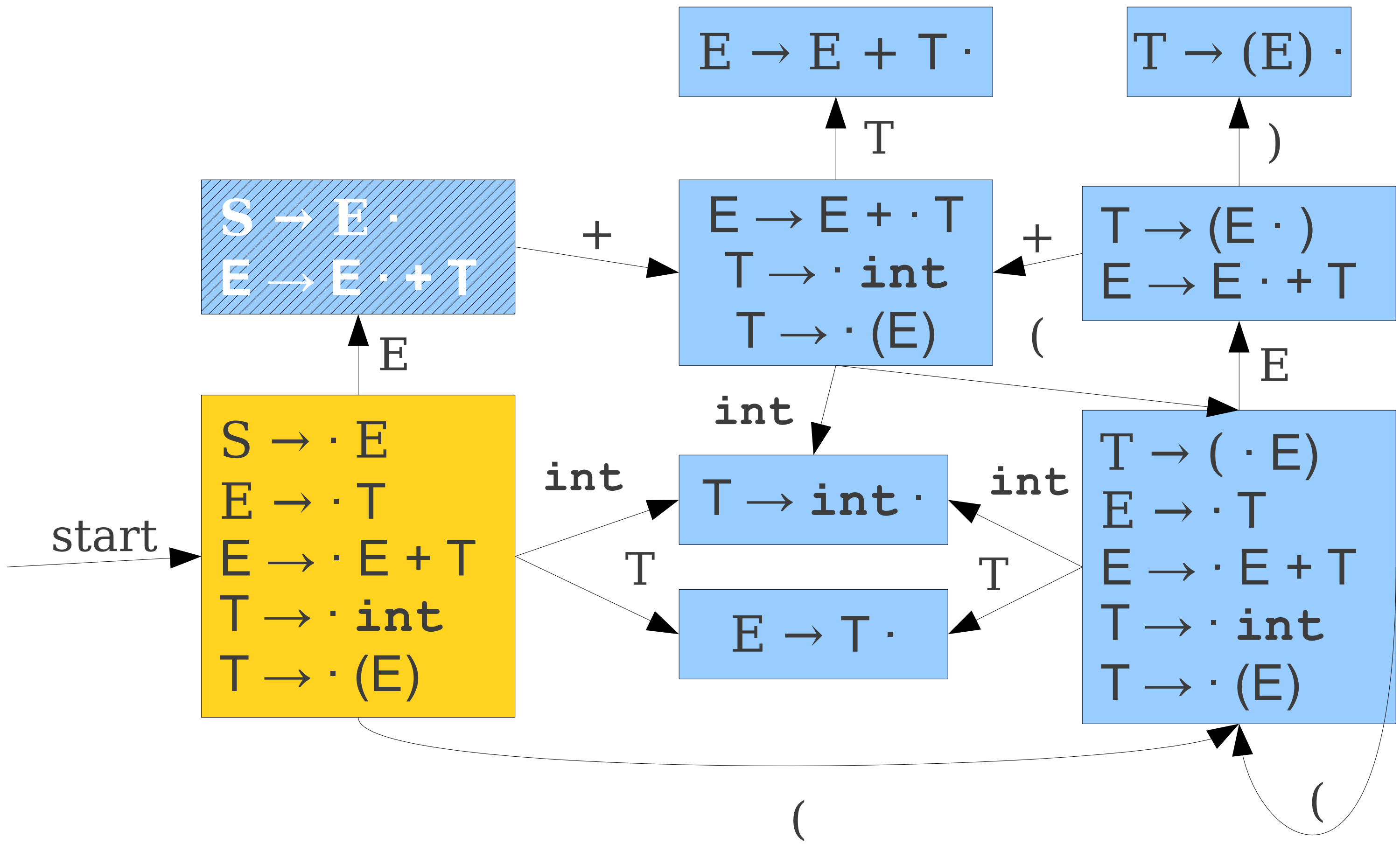
S → · E	$
E → · E + T	$
E → · T	+



+	(	int	+	int	+	int	)	$

LR(1) Parsing: The Intuition

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



S → · E	$
E → · E + T	$
E → · T	+

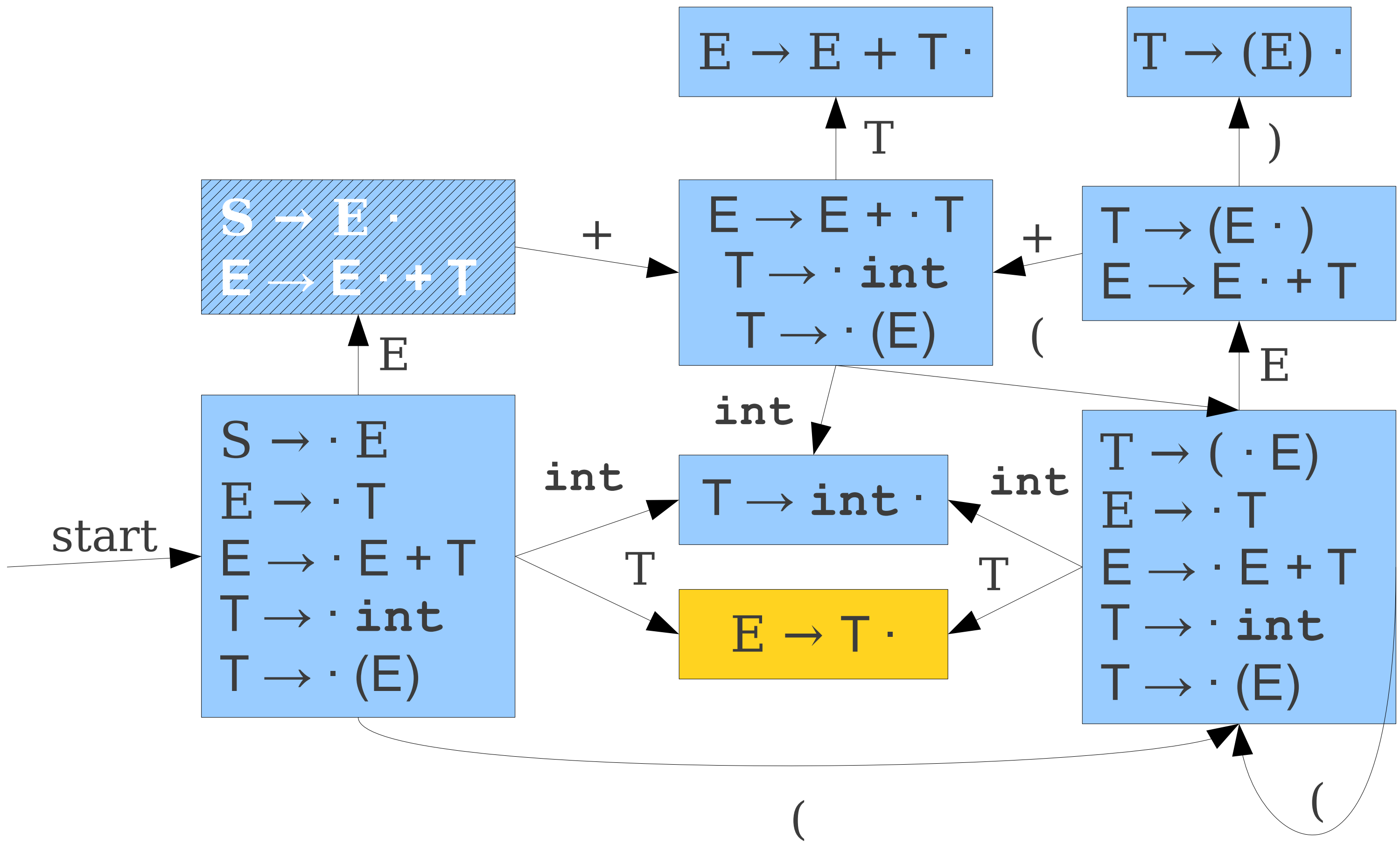
T

+	(	int	+	int	+	int	)	$

LR(1) Parsing: The Intuition

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

S → · E	$
E → · E + T	$
E → T ·	+

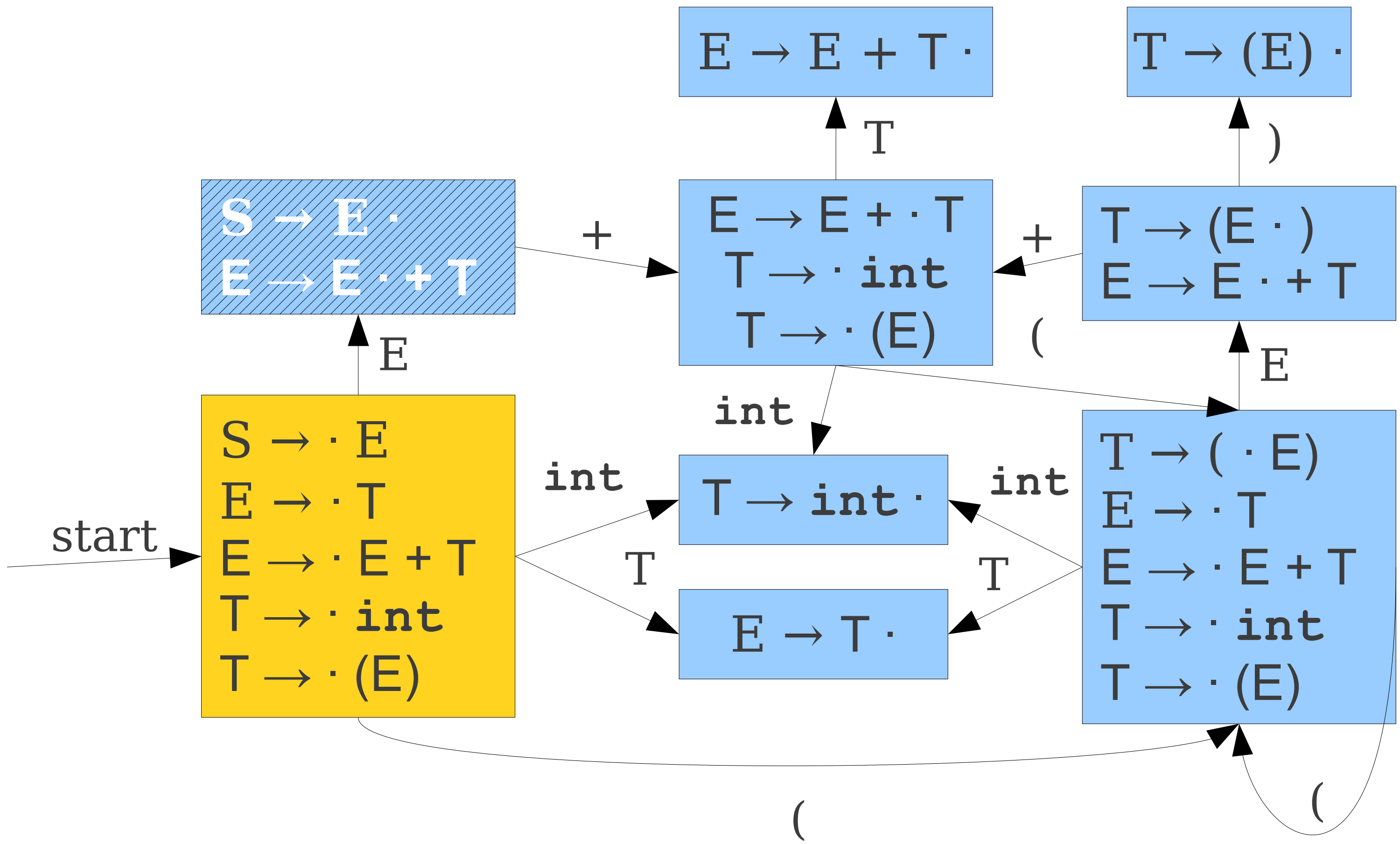


T	|	+	(	int	+	int	+	int	)	$

LR(1) Parsing: The Intuition

S → E
E → T
E → E + T
T → int
T → (E)

S → · E	$
E → · E + T	$

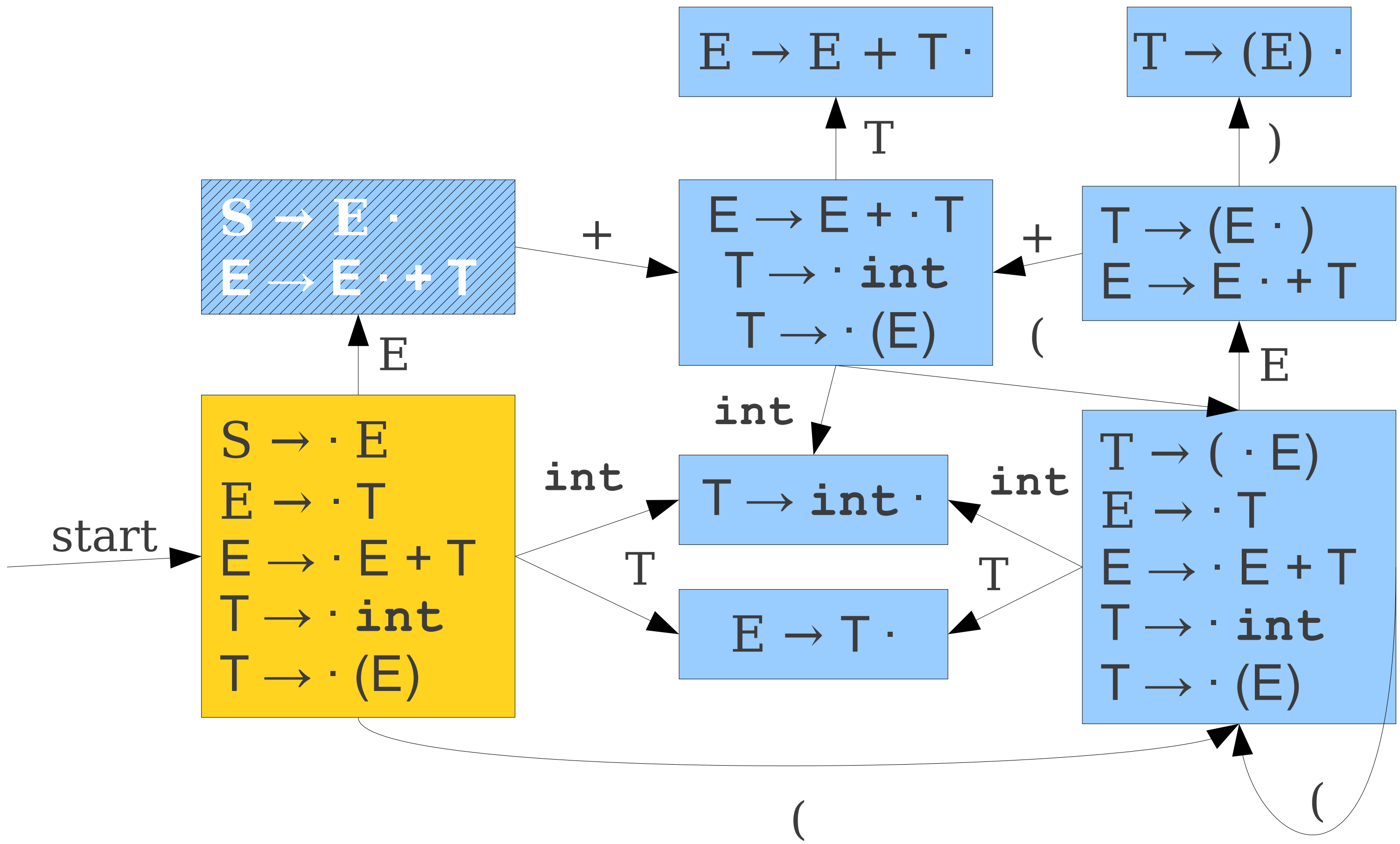


+	(	int	+	int	+	int	)	$

LR(1) Parsing: The Intuition

S → E
E → T
E → E + T
T → int
T → (E)

S → · E	$
E → · E + T	$

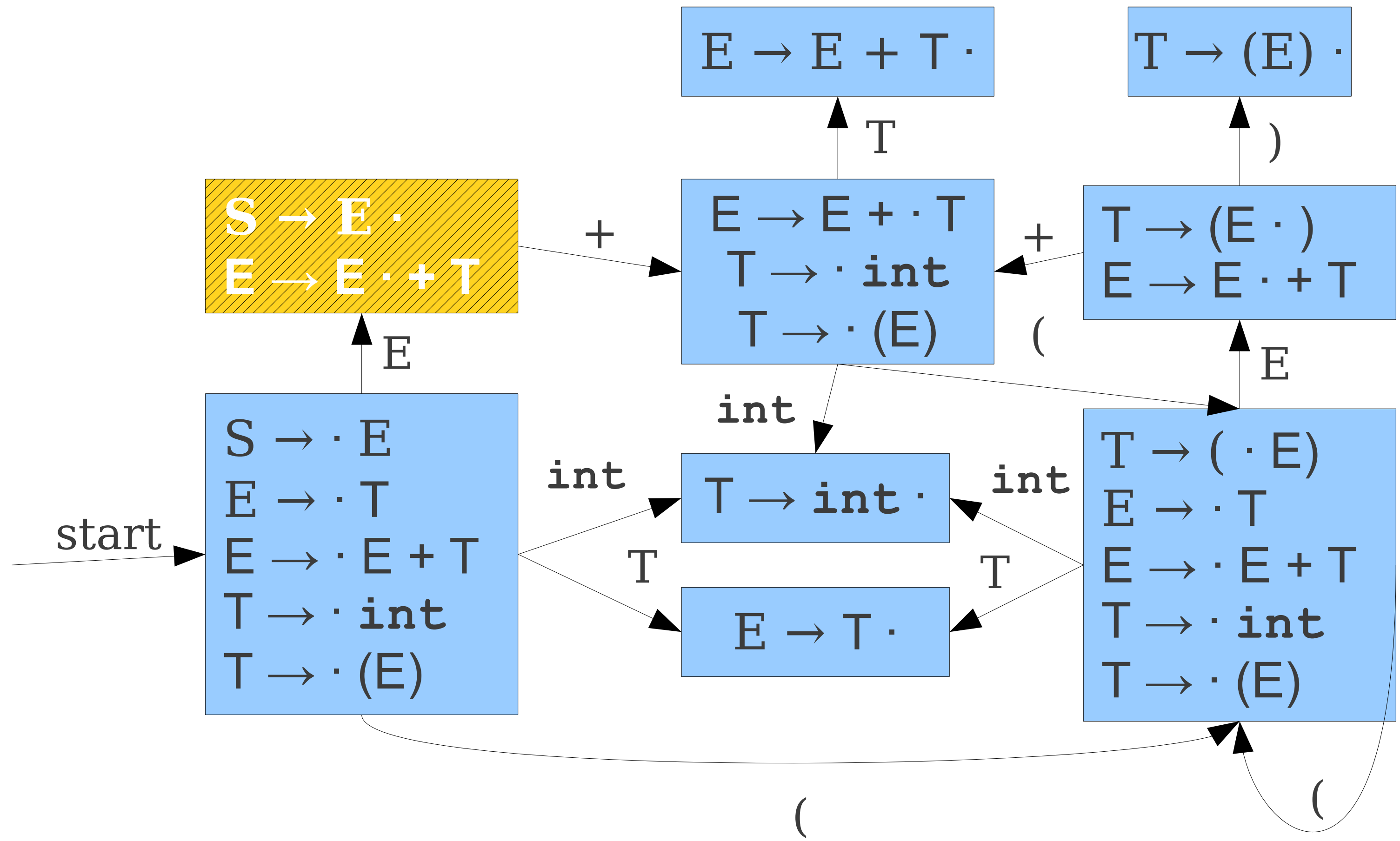


E		+	(	int	+	int	+	int	)	$

LR(1) Parsing: The Intuition

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

S → · E	$
E → E · + T	$

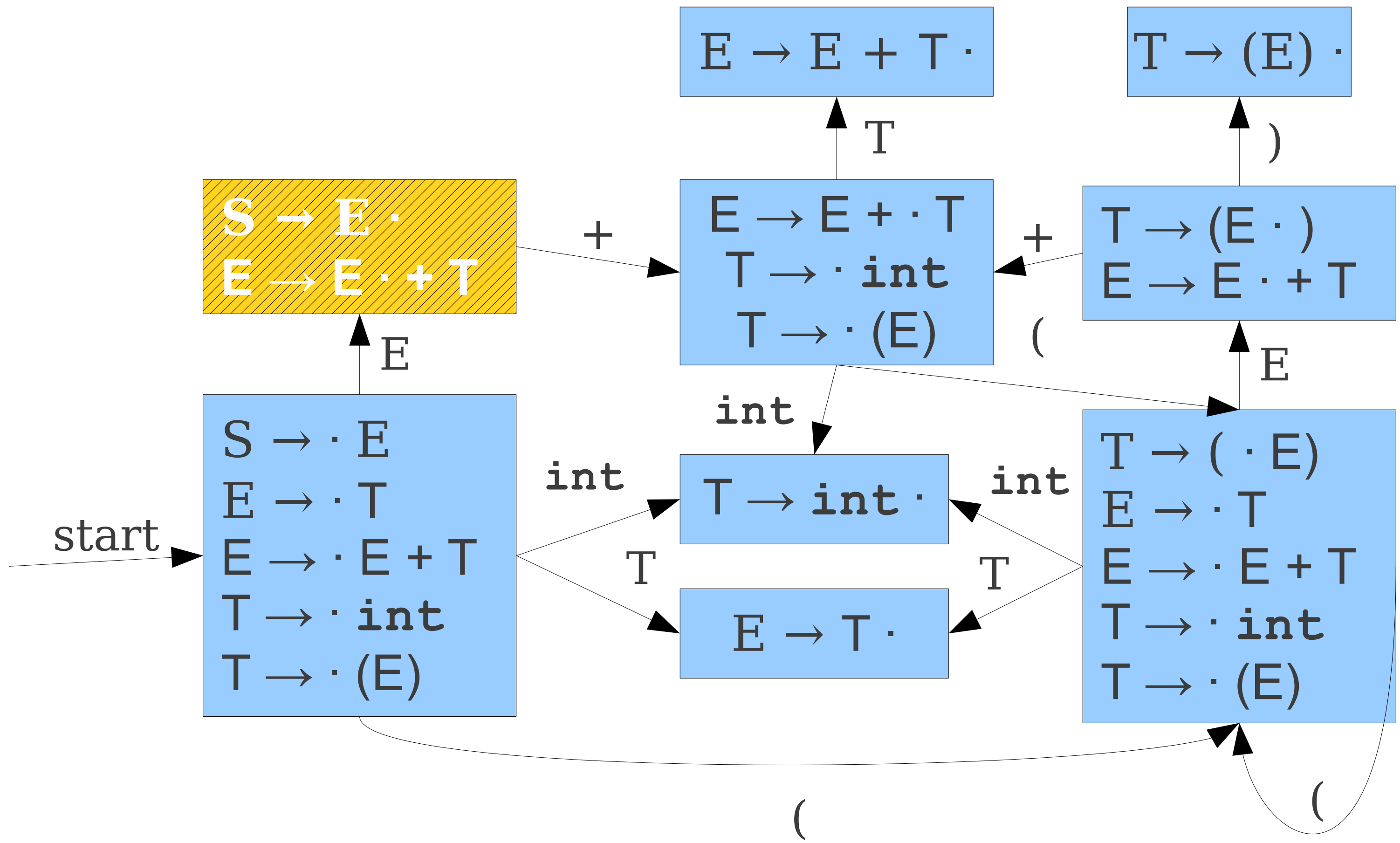


E | + | (| int | + | int | + | int |) | $

LR(1) Parsing: The Intuition

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

S → · E	$
E → E · + T	$

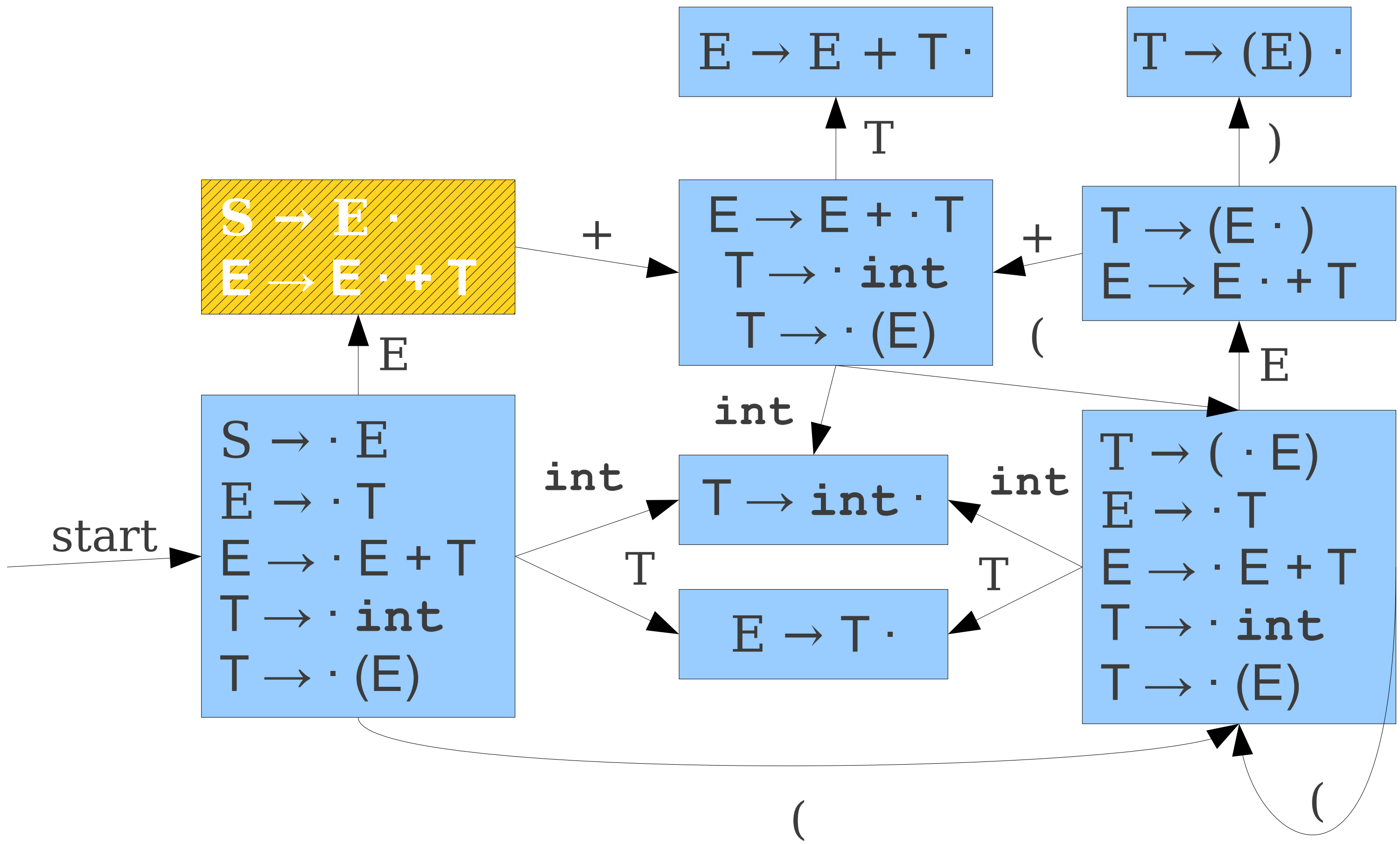


E		+	(	int	+	int	+	int	)	$

LR(1) Parsing: The Intuition

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

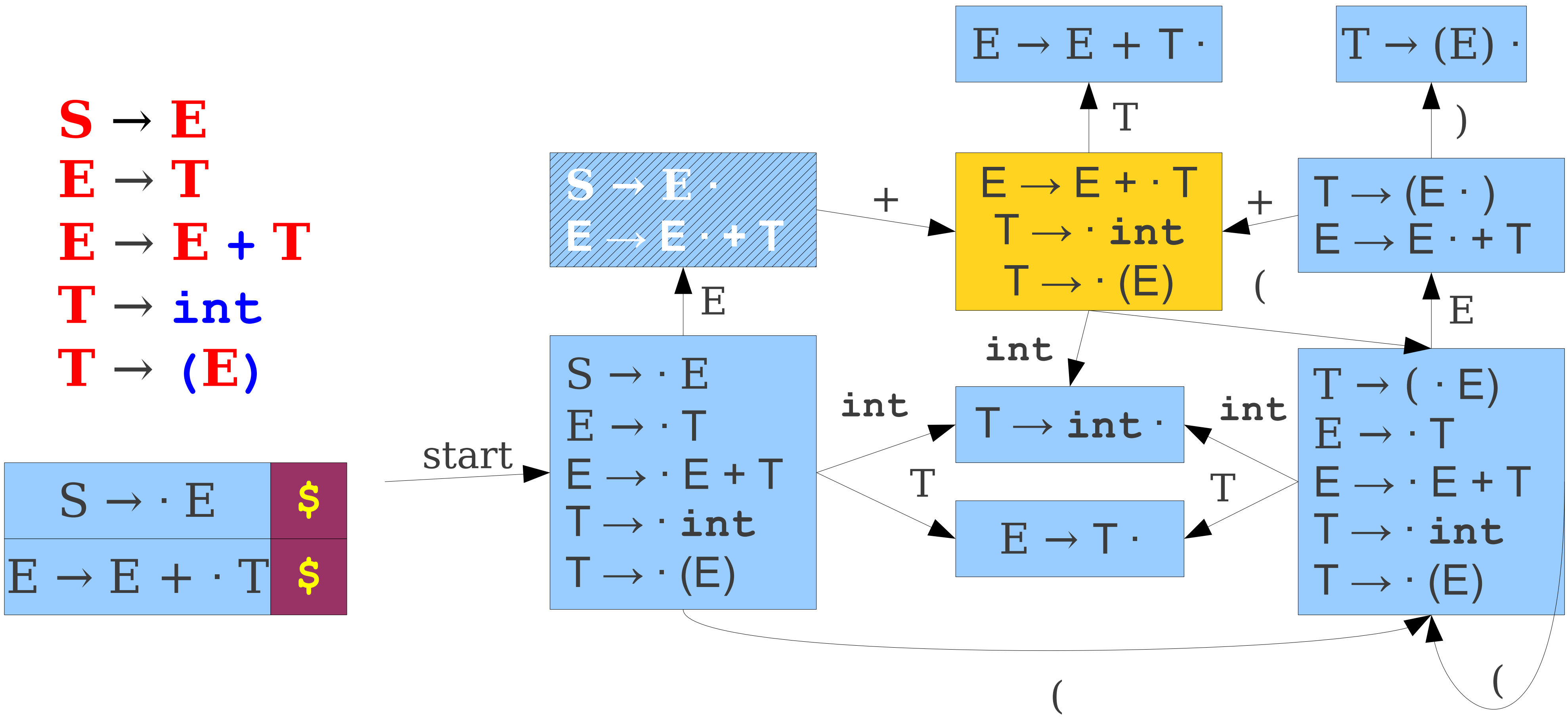
S → · E	$
E → E · + T	$



E	+	|	(	int	+	int	+	int	)	$

LR(1) Parsing: The Intuition

S → E
E → T
E → E + T
T → int
T → (E)



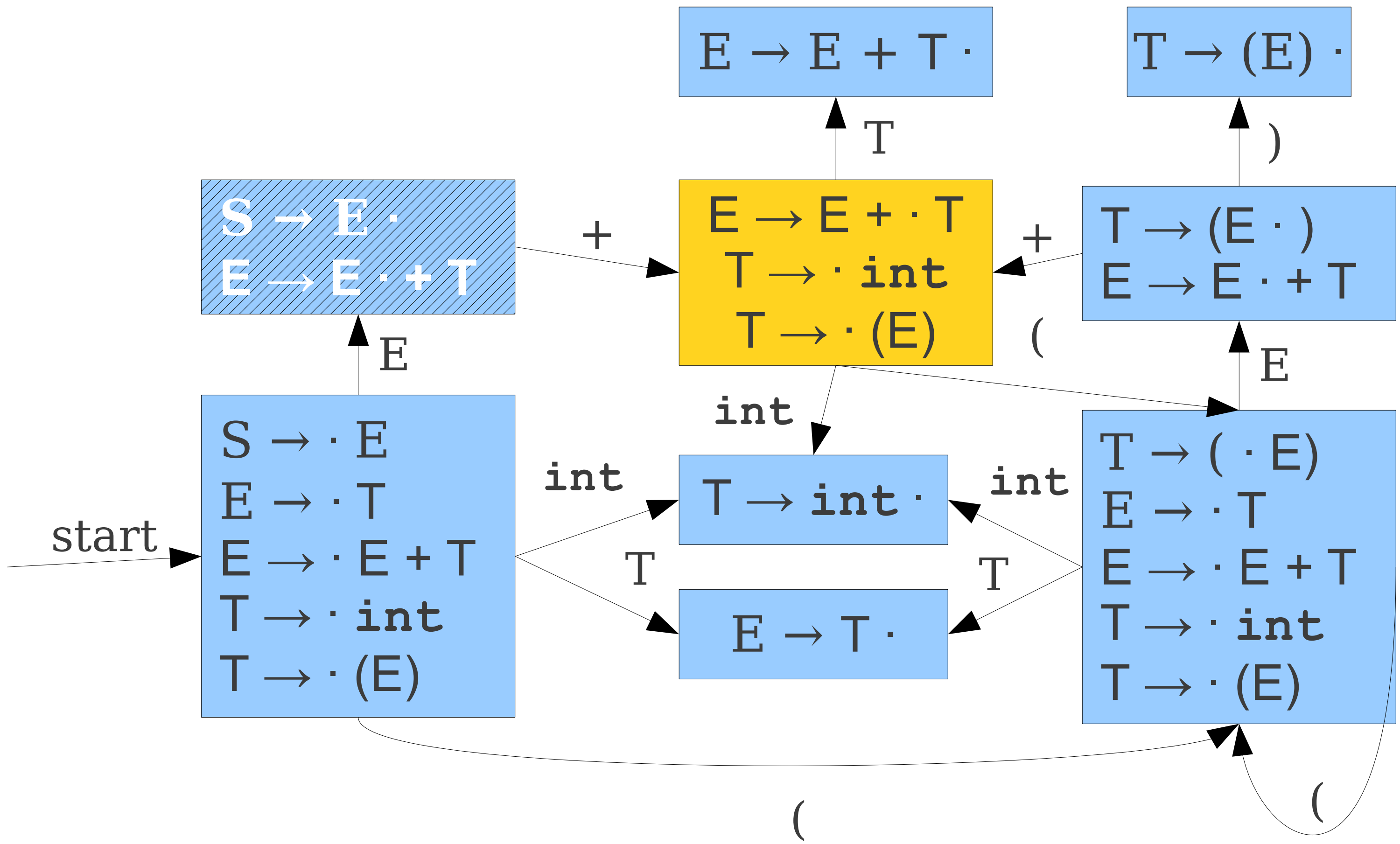
E	+

(	int	+	int	+	int	)	$

LR(1) Parsing: The Intuition

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

$S \rightarrow \cdot E$	$
$E \rightarrow E + \cdot T$	$
$T \rightarrow \cdot (E)$	$

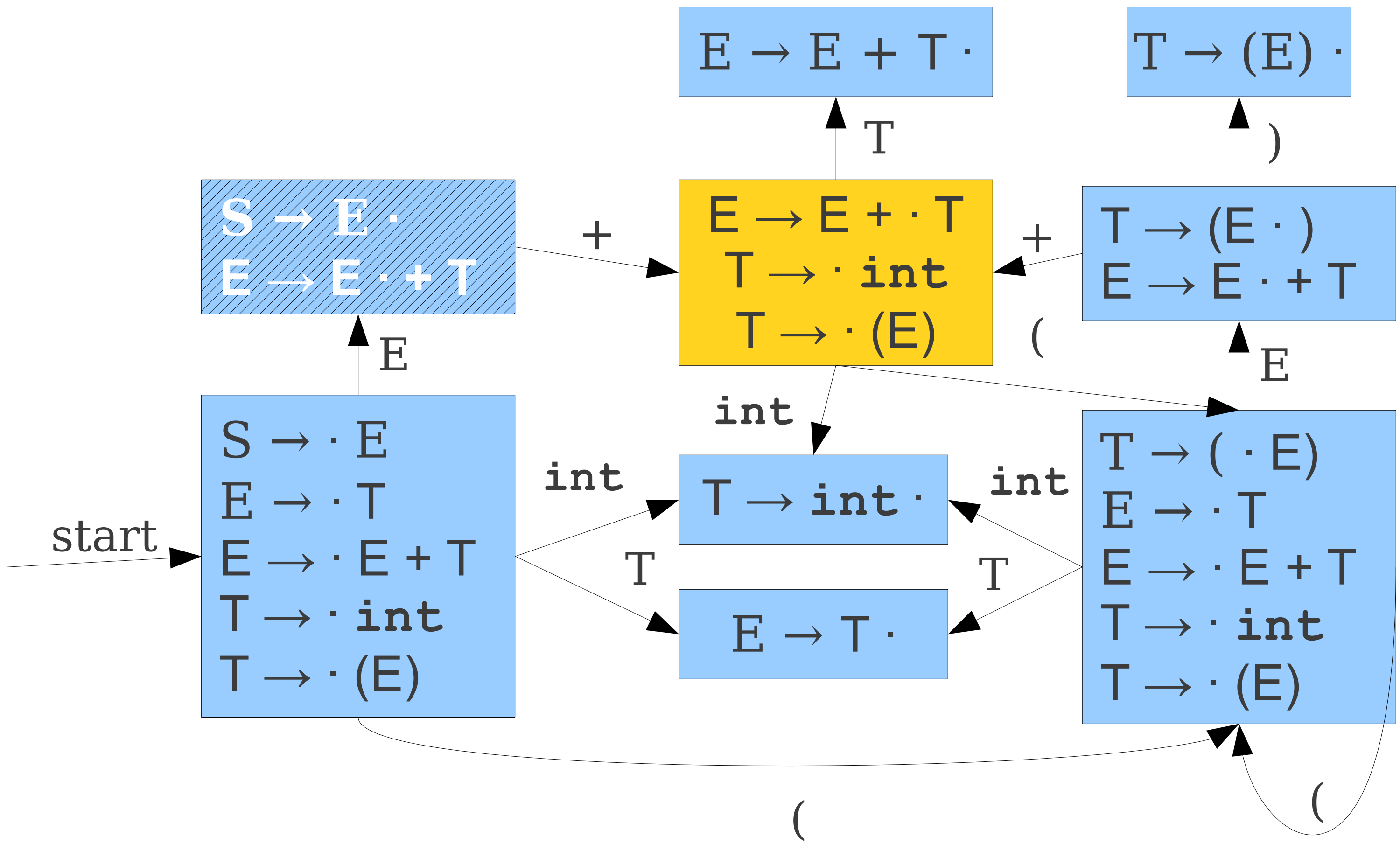


E	+	|	(	int	+	int	+	int	)	$

LR(1) Parsing: The Intuition

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

S → · E	$
E → E + · T	$
T → · (E)	$

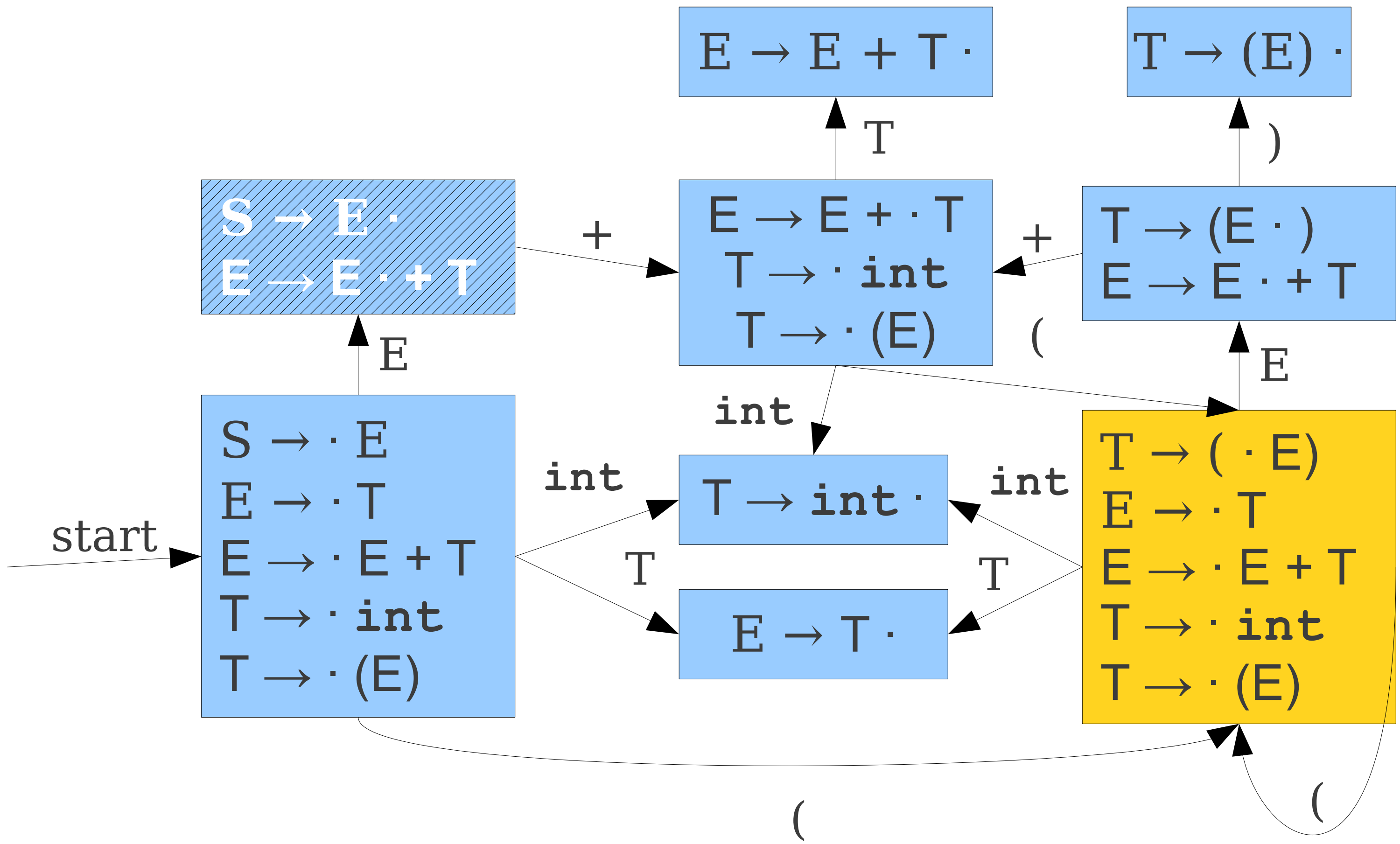


E	+	(		int	+	int	+	int	)	$

LR(1) Parsing: The Intuition

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

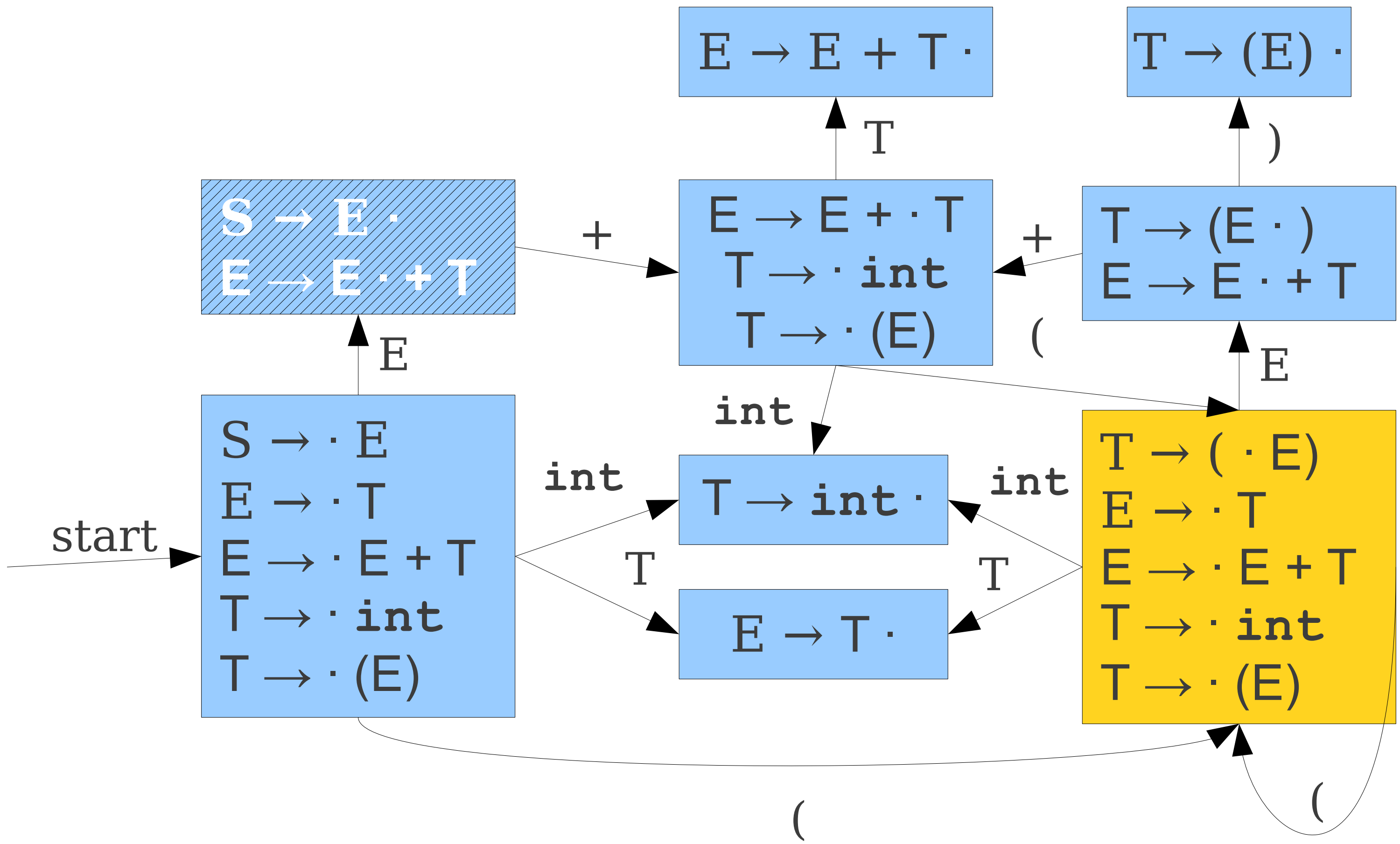
S → · E	$
E → E + · T	$
T → (· E)	$



E	+	(	|	int	+	int	+	int	)	$

LR(1) Parsing: The Intuition

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

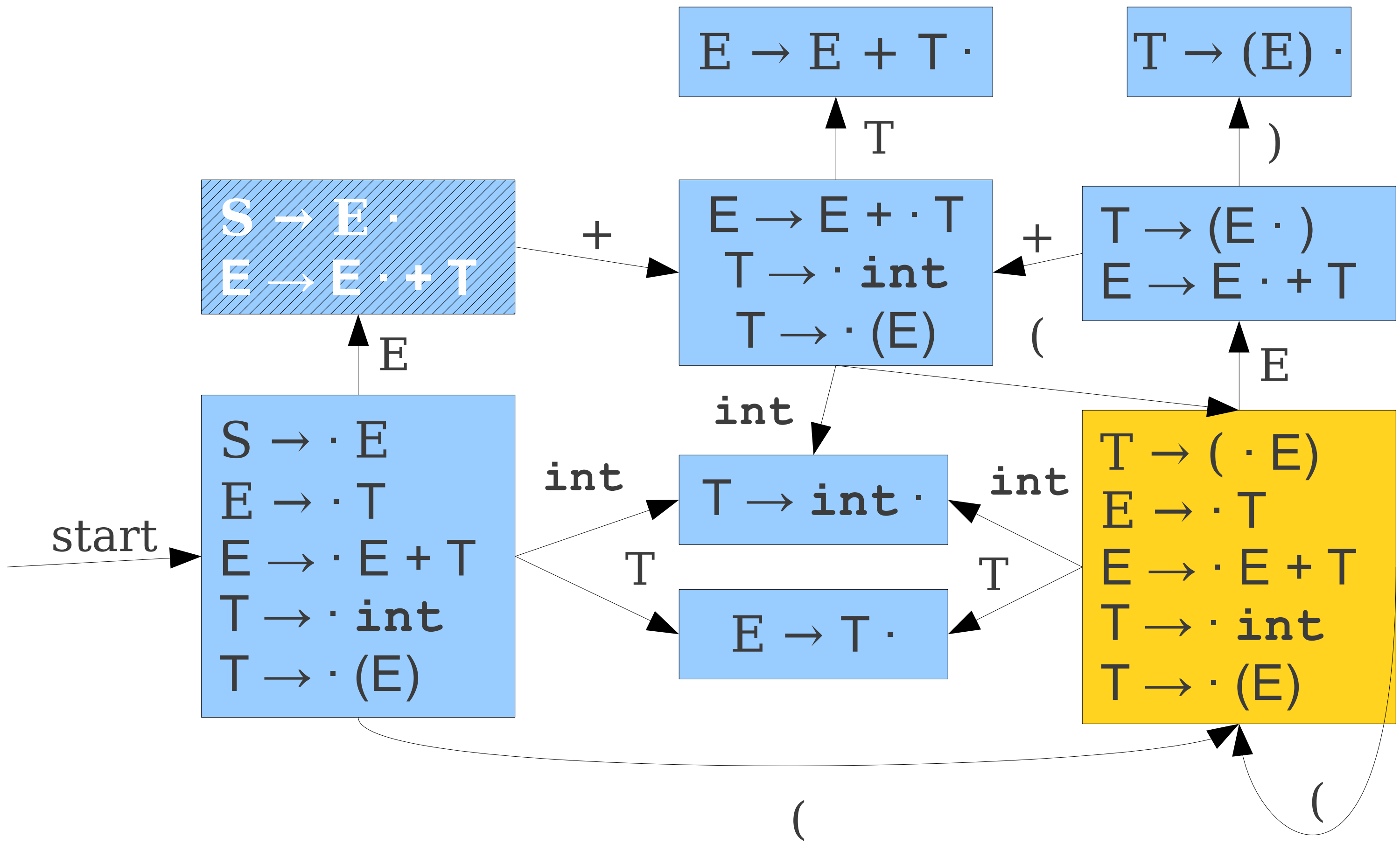


S → · E	$
E → E + · T	$
T → (· E)	$
E → · E + T	)

E	+	(		int	+	int	+	int	)	$

LR(1) Parsing: The Intuition

S → E
E → T
E → E + T
T → int
T → (E)

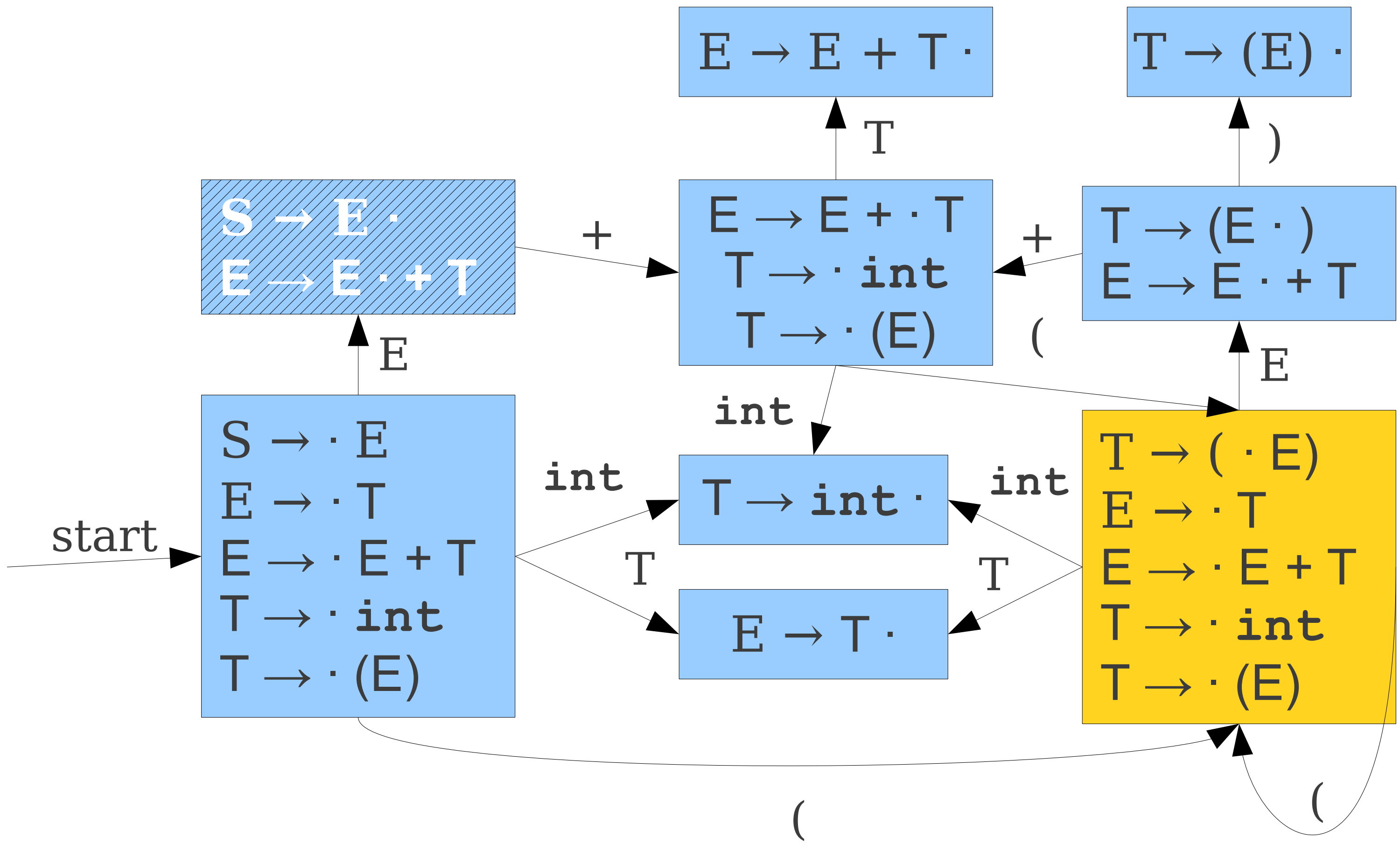


S → · E	$
E → E + · T	$
T → (· E)	$
E → · E + T	)
E → · T	+

E	+	(	|	int	+	int	+	int	)	$

LR(1) Parsing: The Intuition

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

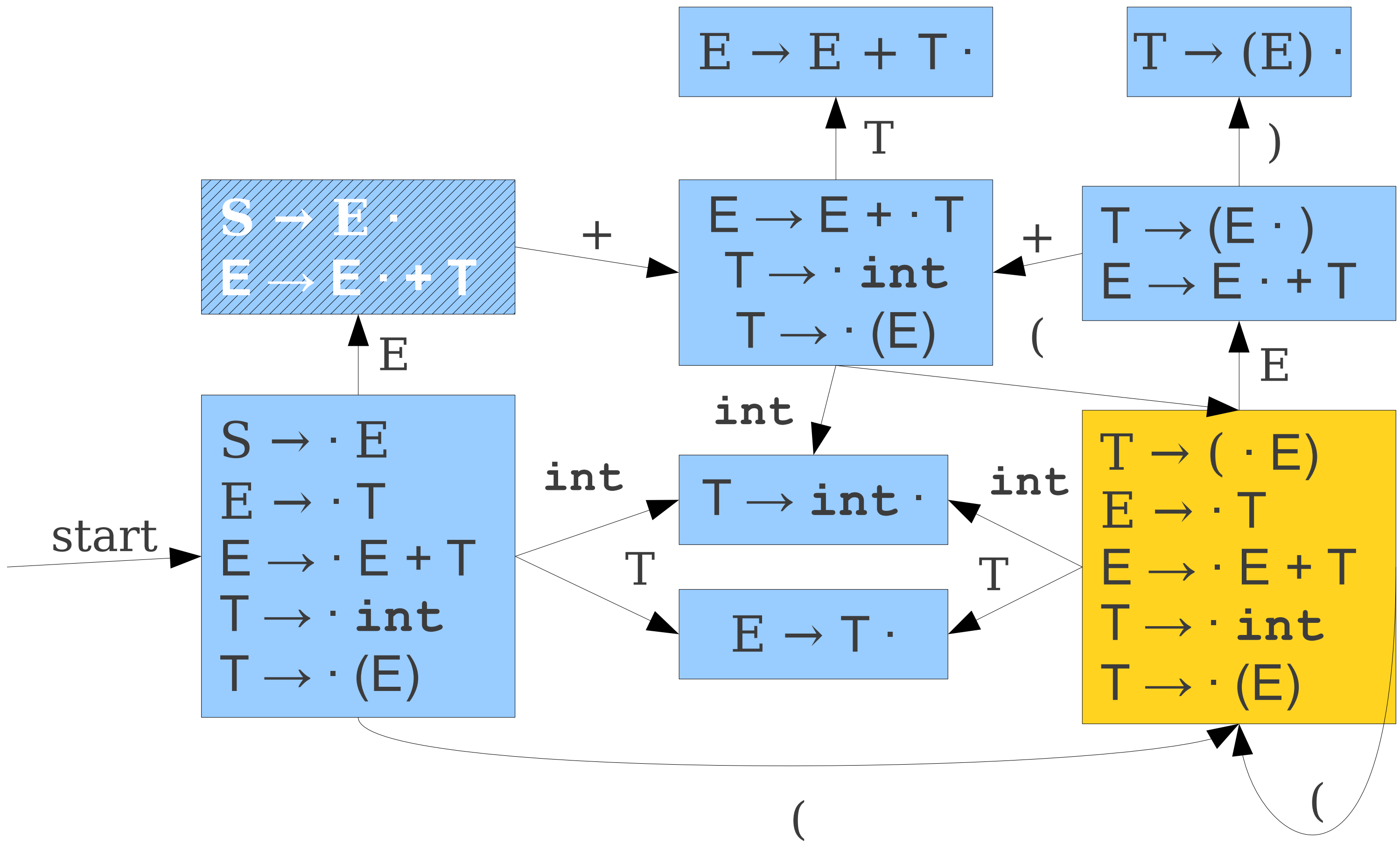


S → · E	$
E → E + · T	$
T → (· E)	$
E → · E + T	)
E → · T	+
T → · int	+

E	+	(	|	int	+	int	+	int	)	$

LR(1) Parsing: The Intuition

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

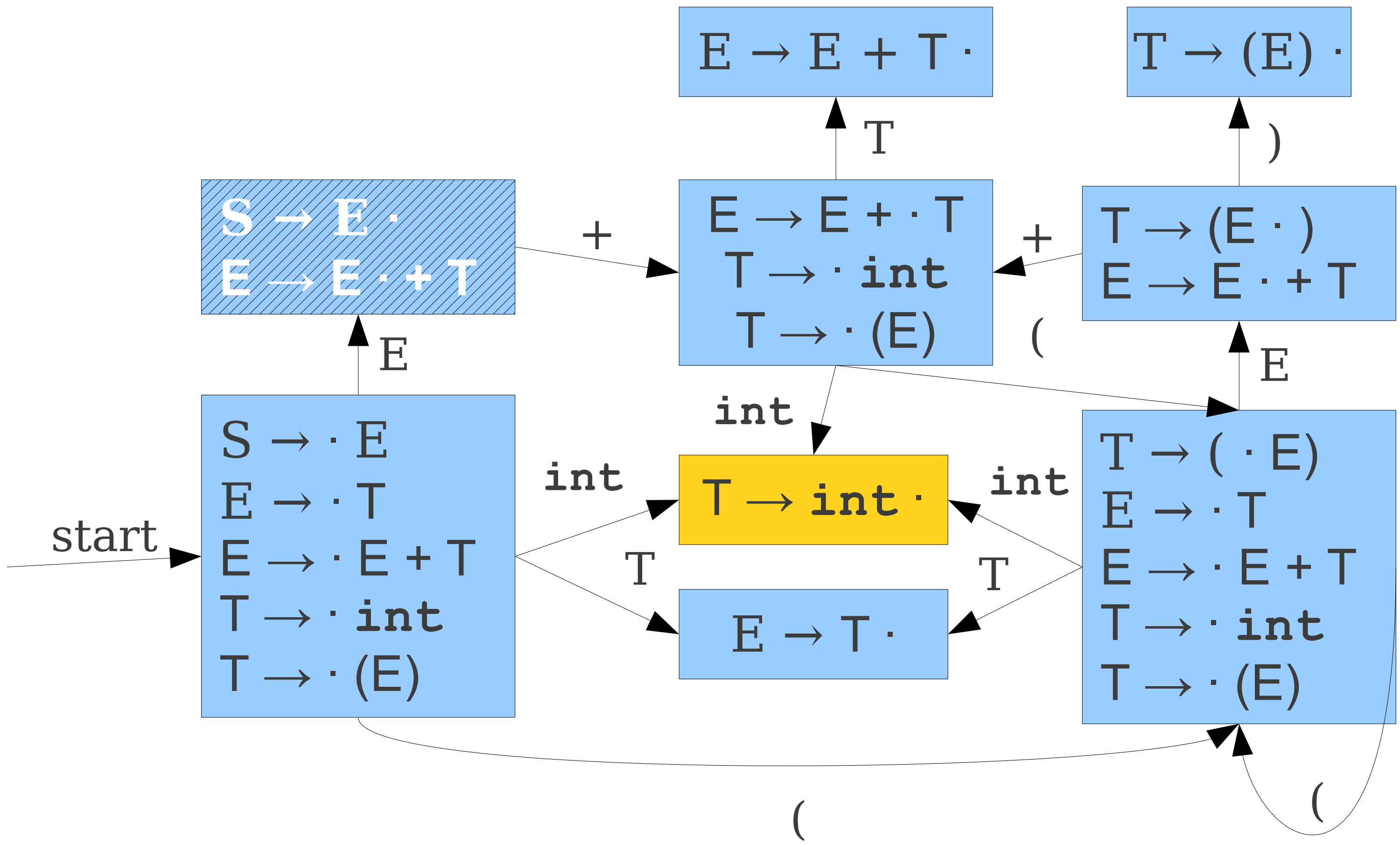


S → · E	$
E → E + · T	$
T → (· E)	$
E → · E + T	)
E → · T	+
T → · int	+

E	+	(	int	+	int	+	int	)	$

LR(1) Parsing: The Intuition

S → E
E → T
E → E + T
T → int
T → (E)



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

E	+	(	int	|	+	int	+	int	)	$

The Intuition behind LR(1)

- Guess which series of productions we are reversing.
- Use this information to maintain information about what lookahead to expect.
- When deciding whether to shift or reduce, use lookahead to disambiguate.

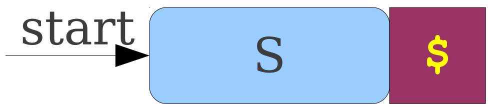
Tracking Lookaheads

- How do we know what lookahead to expect at each state?
- Observation:
  - There are only finitely many productions we can be in at any point.
  - There are only finitely many positions we can be in each production.
  - **There are only finitely many lookahead sets** at each point.
- Construct an automaton to track lookaheads!

Constructing LR(1) Automata

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

Constructing LR(1) Automata



S → **E**

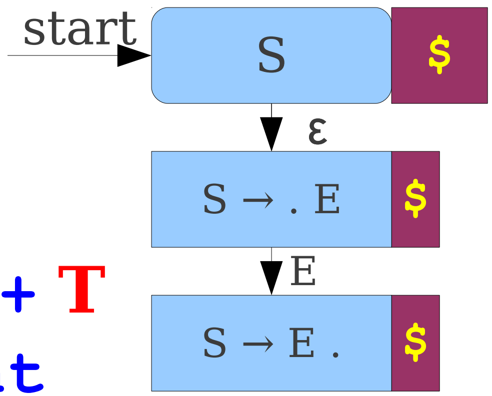
E → **T**

E → **E** **+** **T**

T → **int**

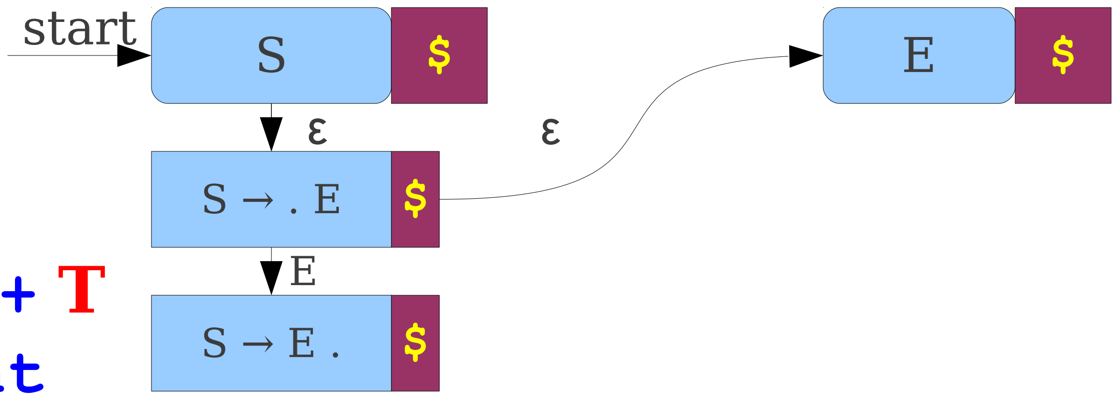
T → **(E)**

Constructing LR(1) Automata



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

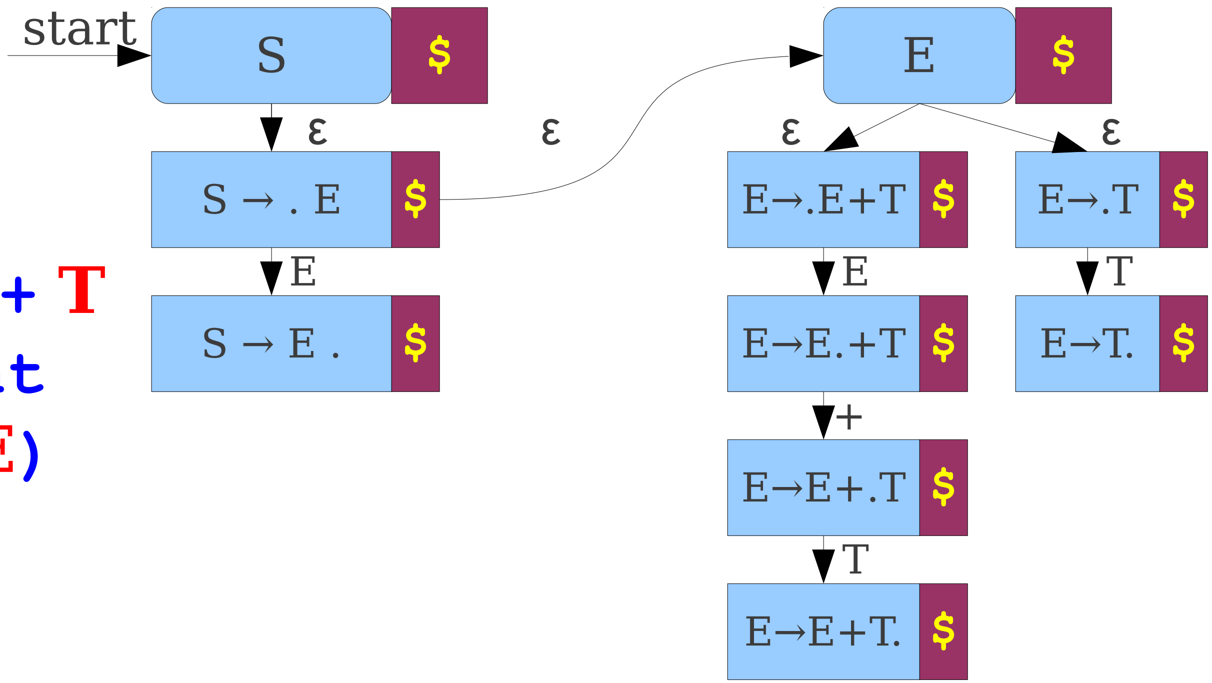
Constructing LR(1) Automata



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

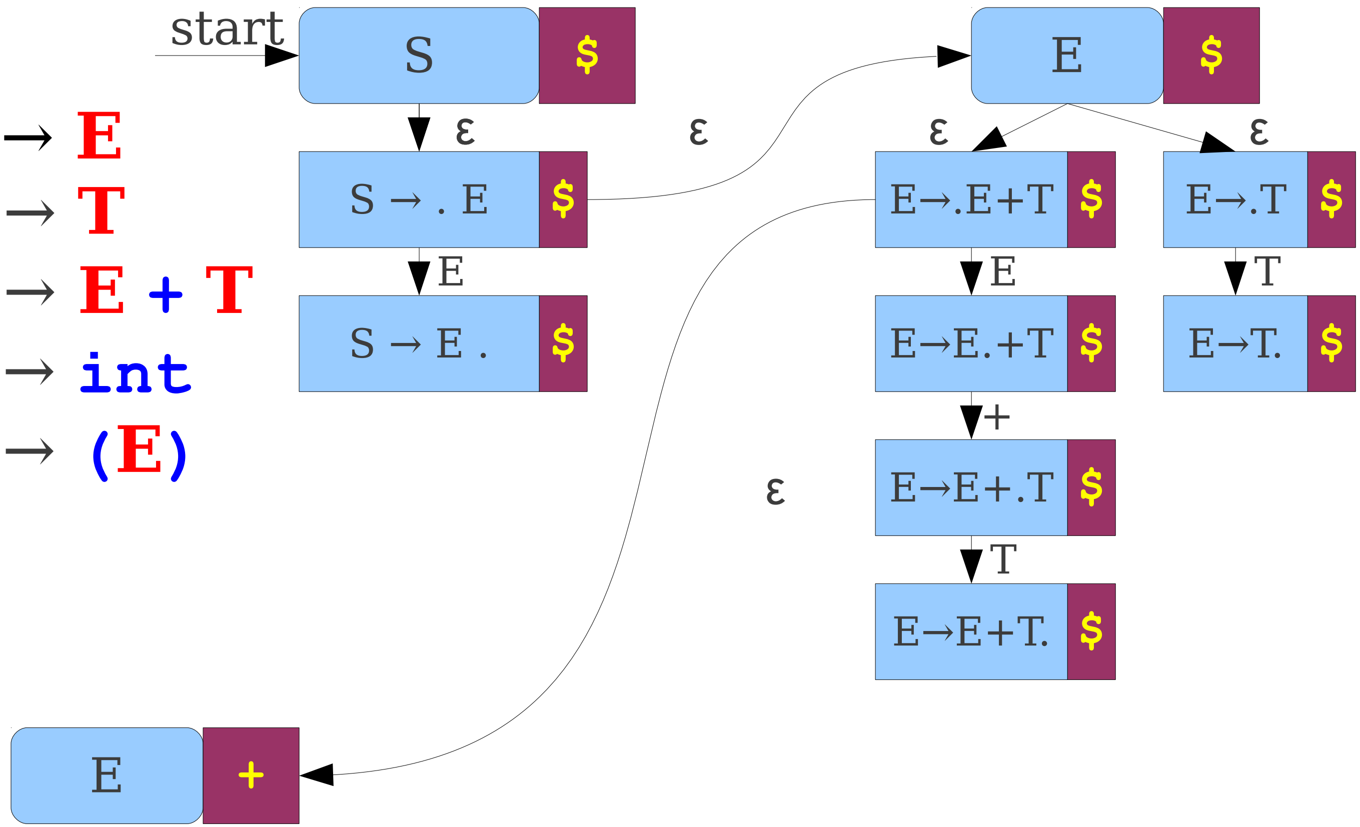
Constructing LR(1) Automata

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



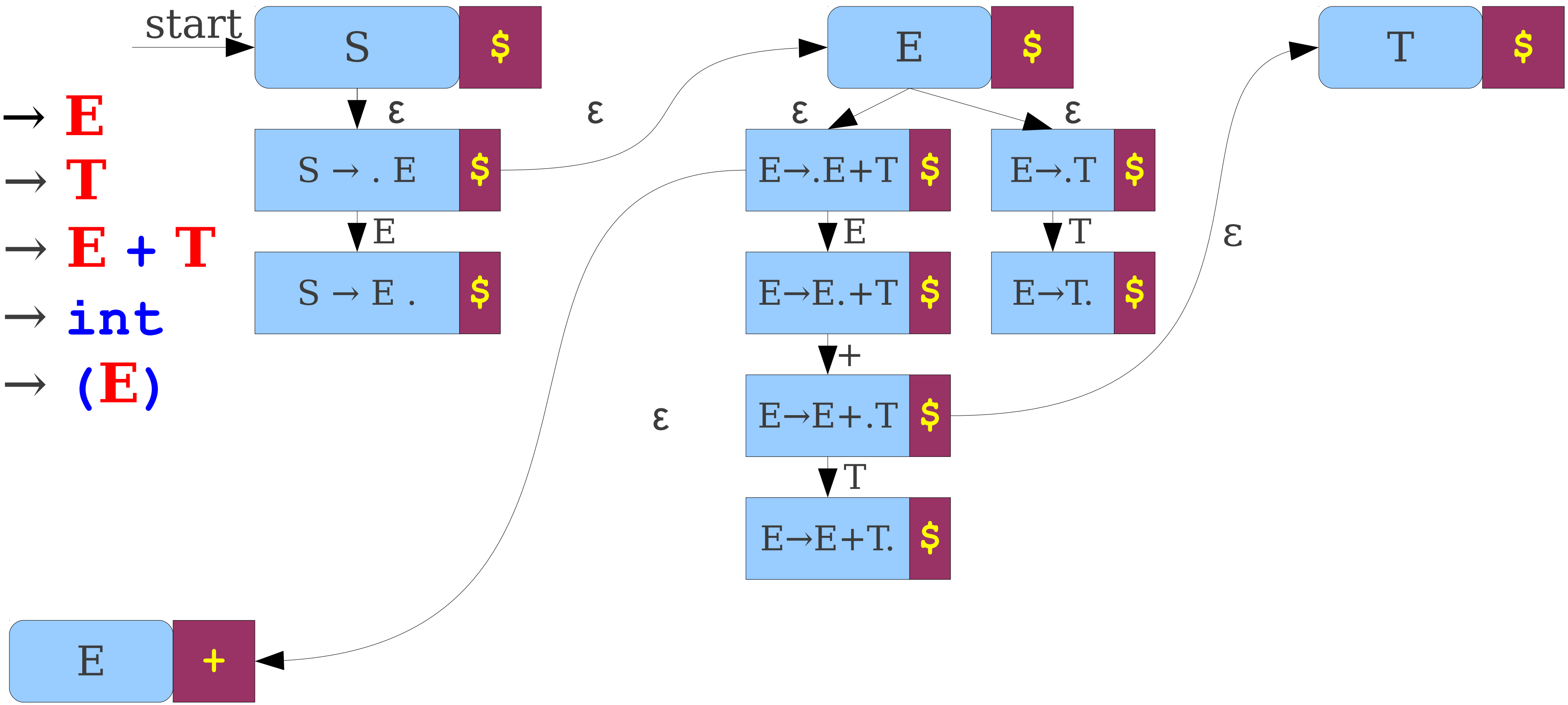
Constructing LR(1) Automata

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



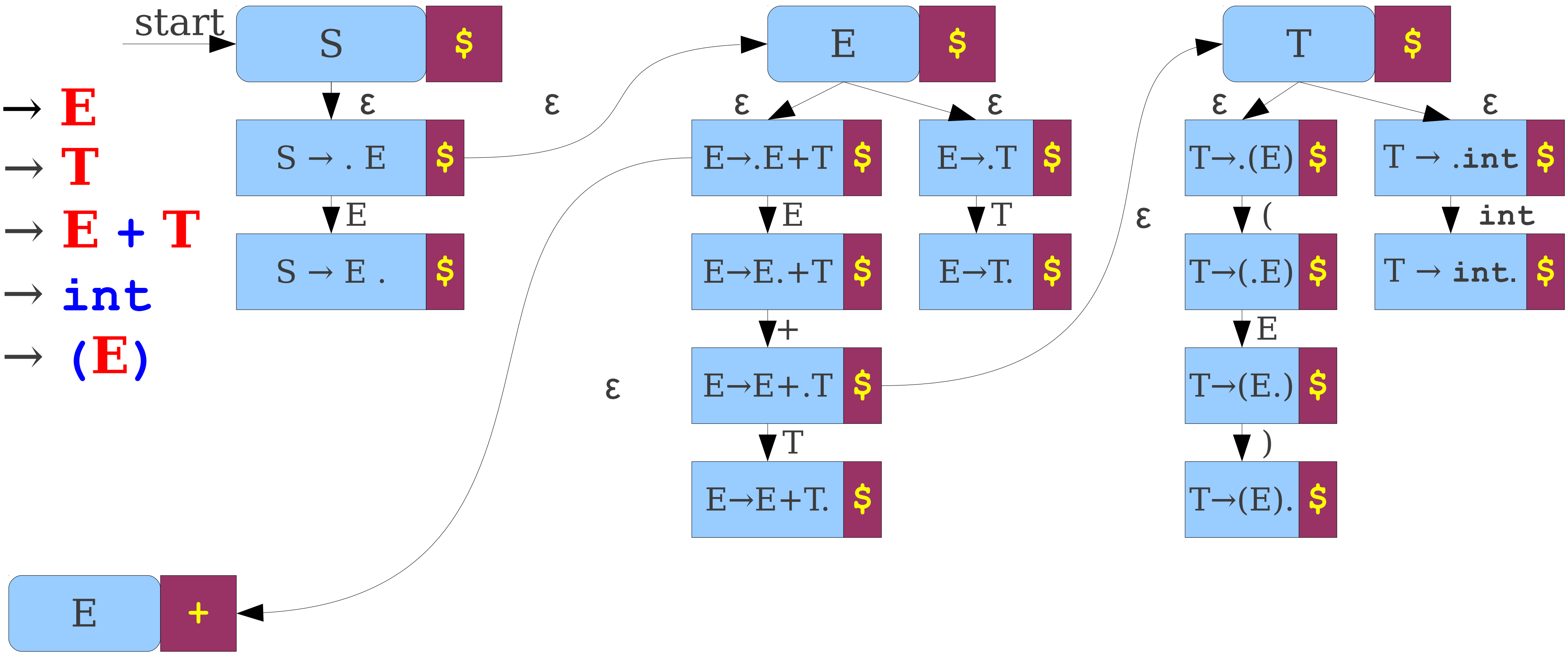
Constructing LR(1) Automata

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



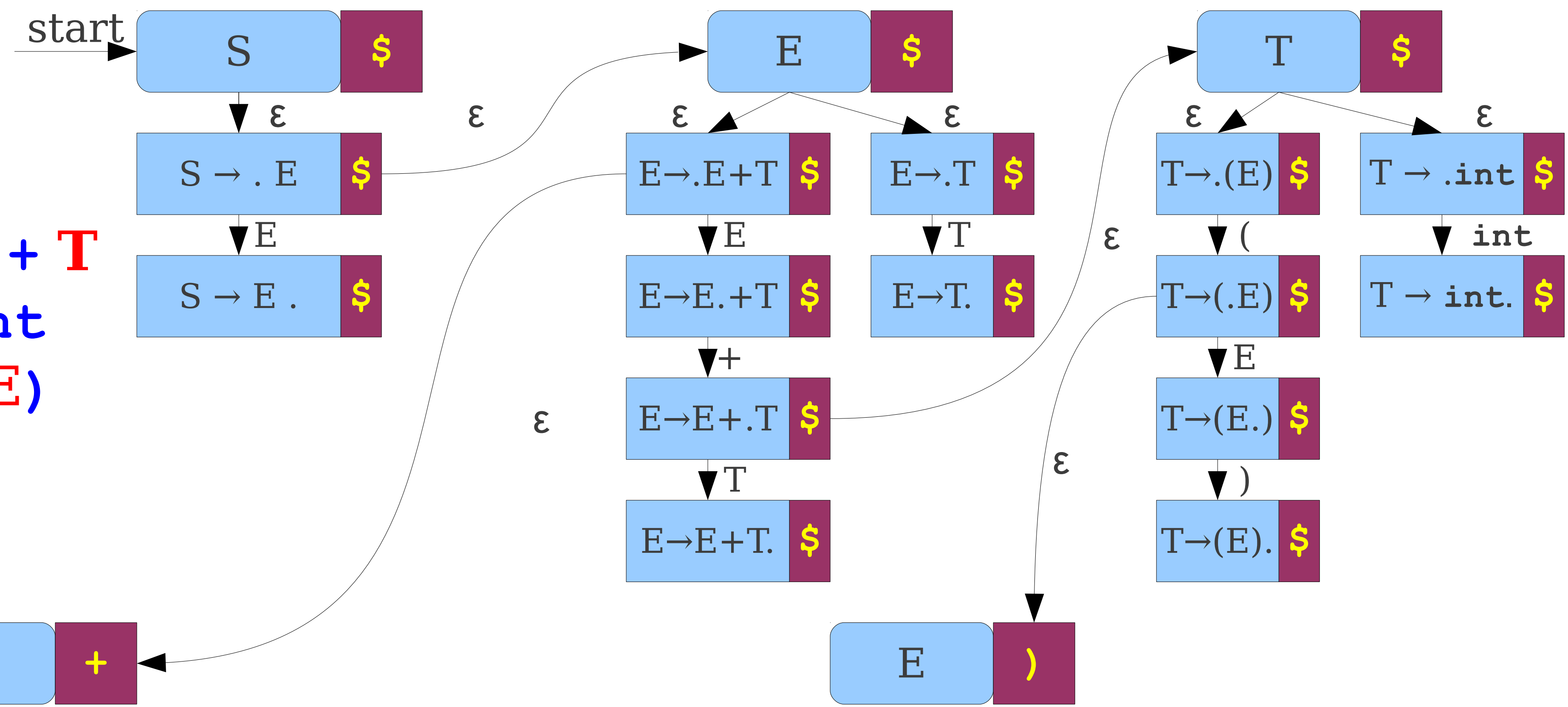
Constructing LR(1) Automata

S → E
E → T
E → E + T
T → int
T → (E)



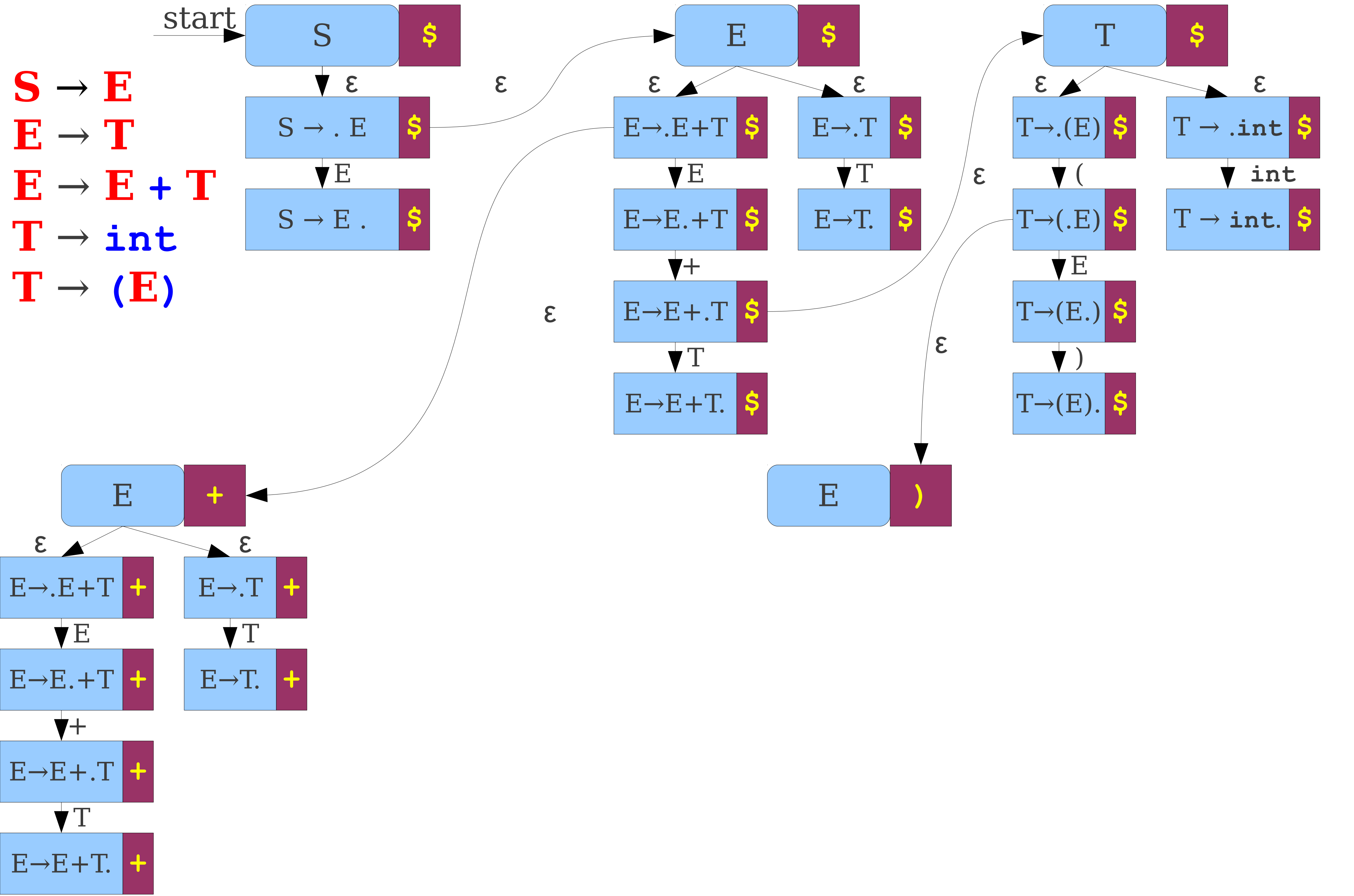
Constructing LR(1) Automata

S → E
E → T
E → E + T
T → int
T → (E)



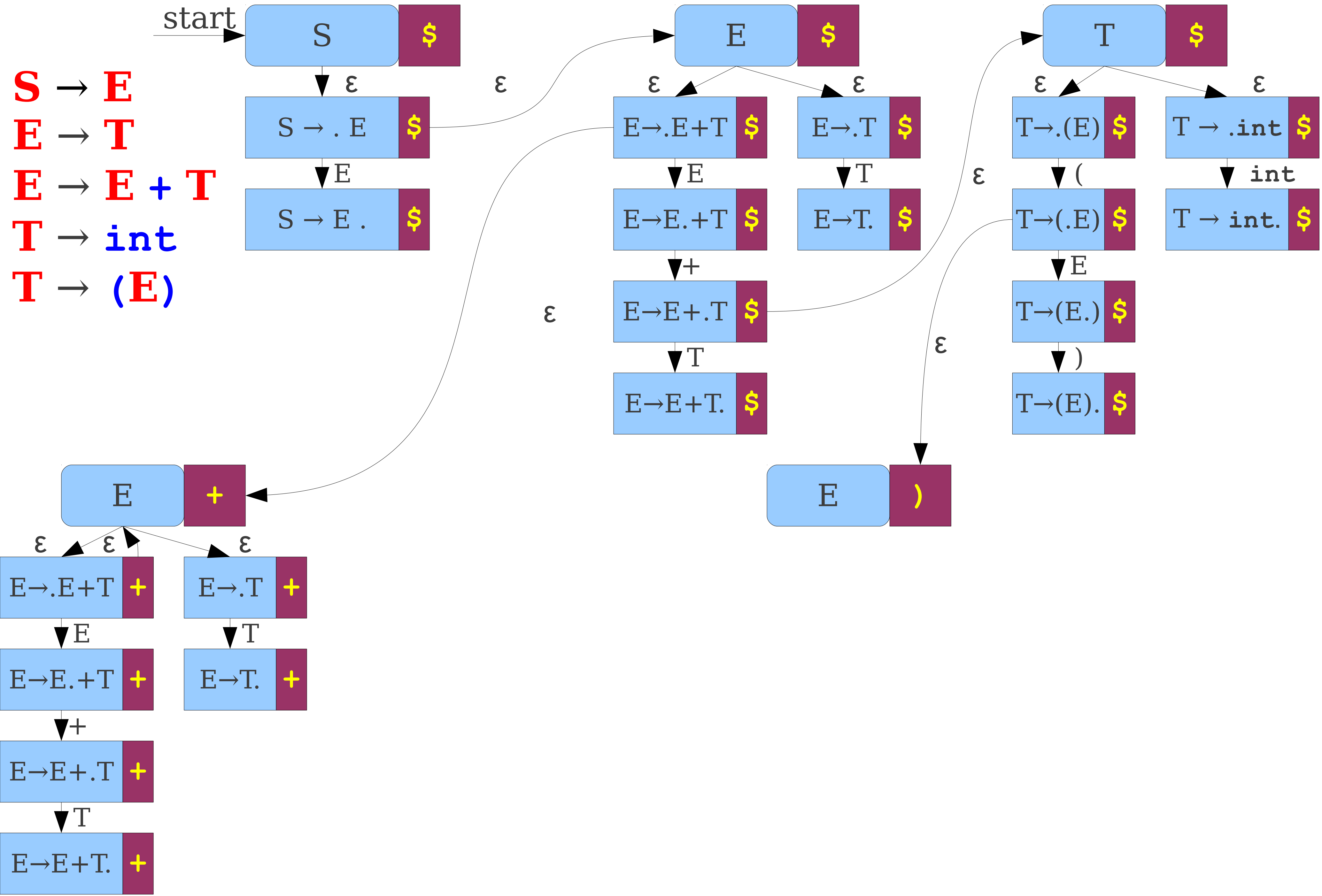
Constructing LR(1) Automata

S → E
E → T
E → E + T
T → int
T → (E)



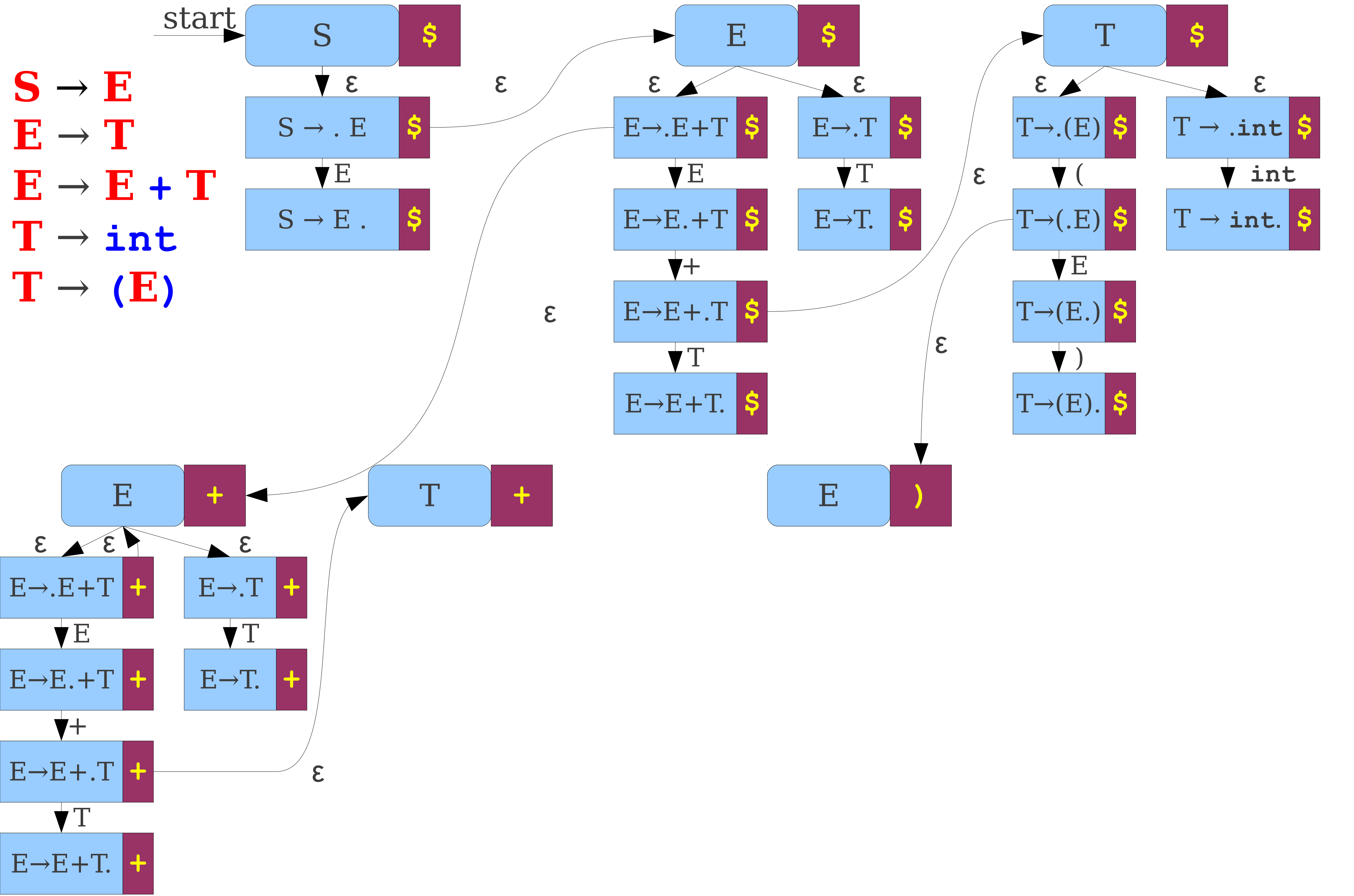
Constructing LR(1) Automata

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



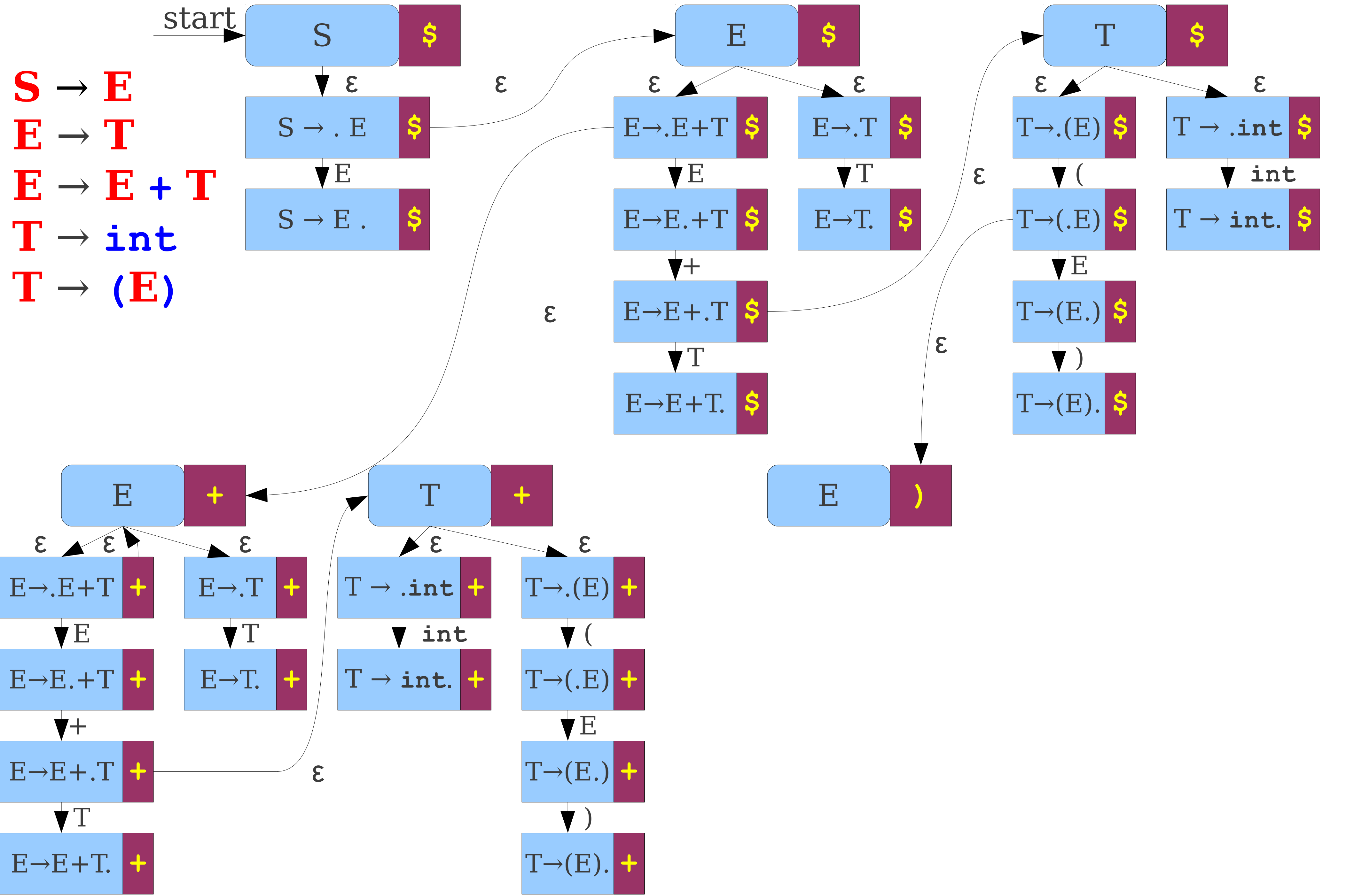
Constructing LR(1) Automata

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



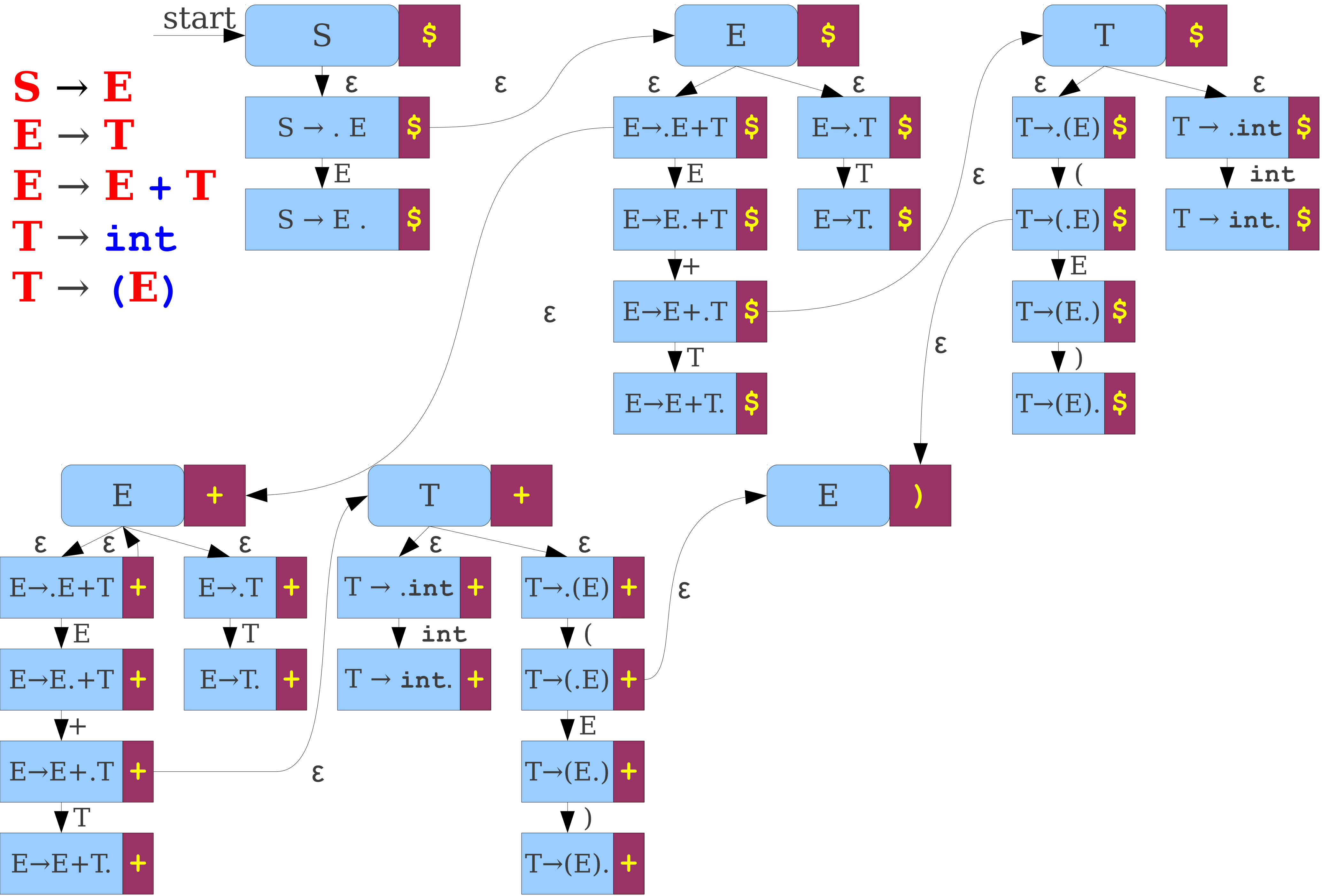
Constructing LR(1) Automata

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



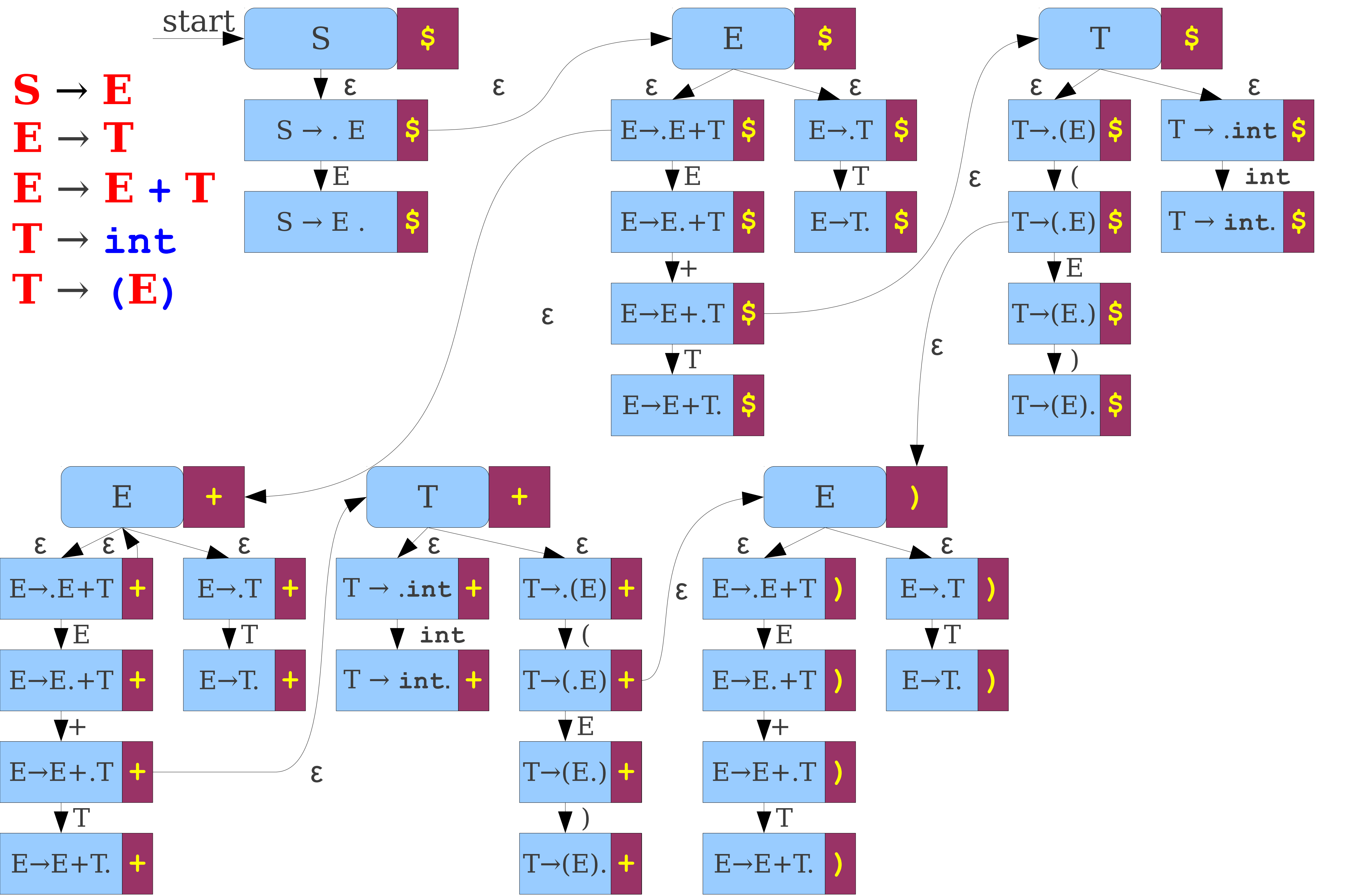
Constructing LR(1) Automata

S → E
E → T
E → E + T
T → int
T → (E)



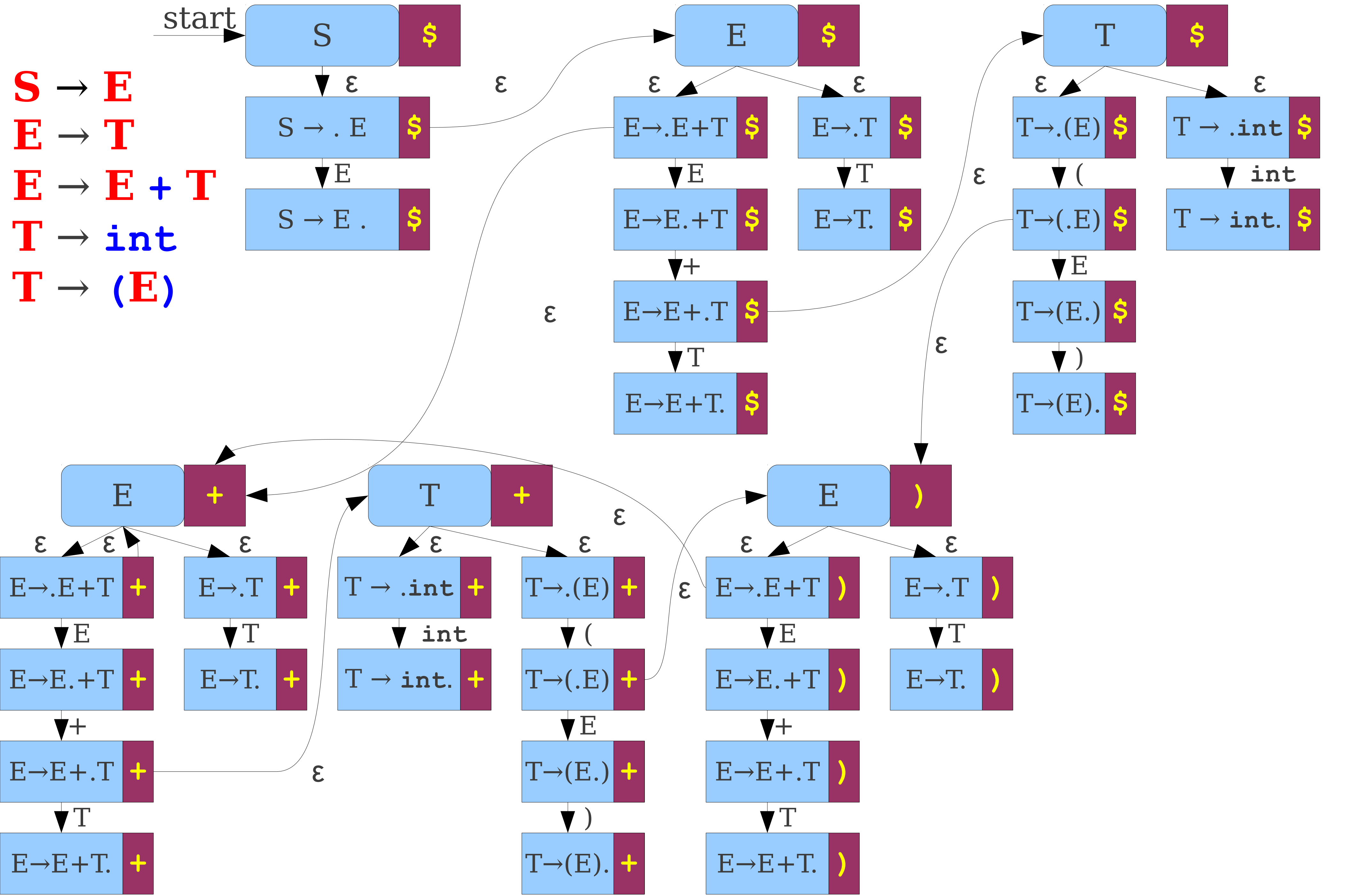
Constructing LR(1) Automata

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



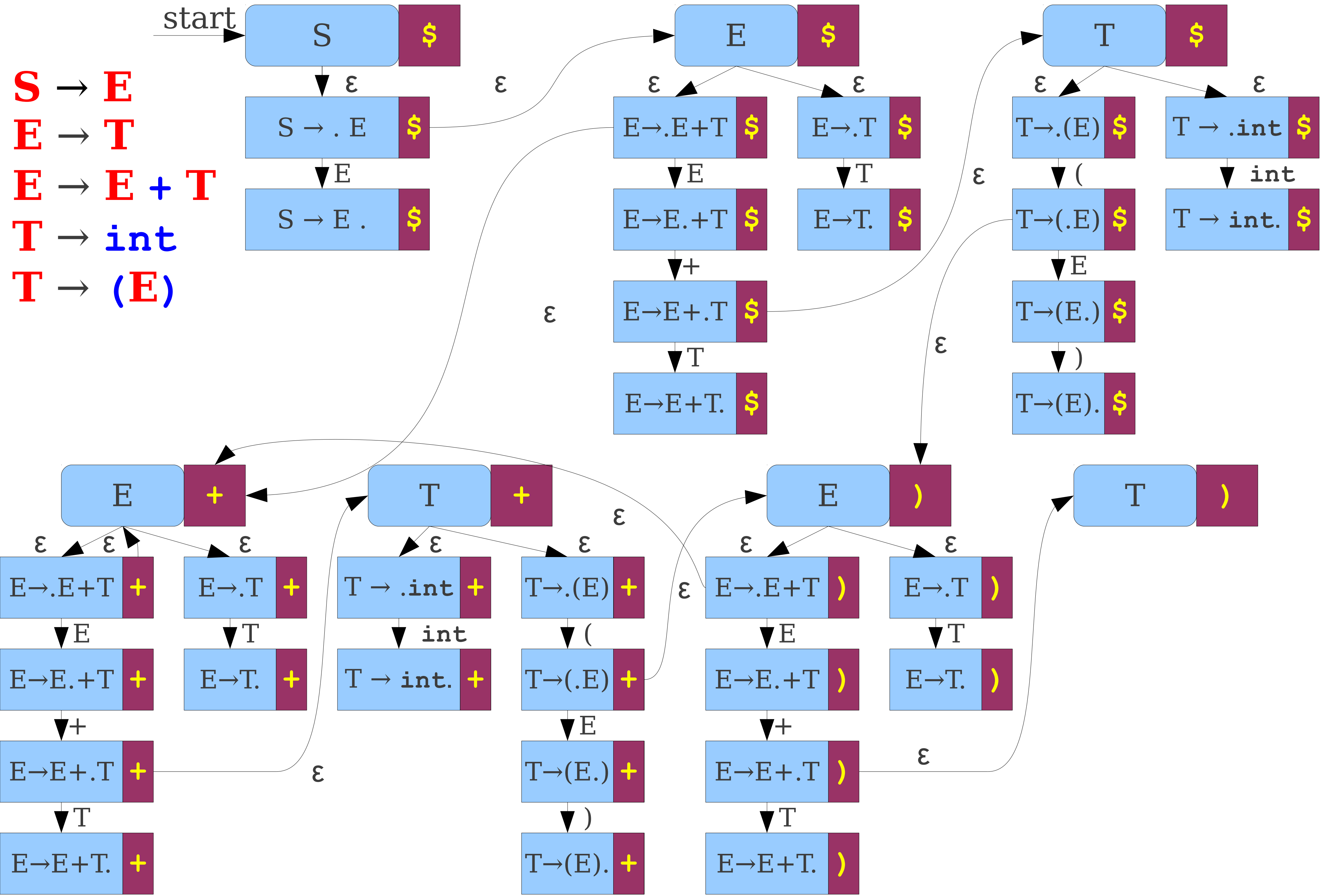
Constructing LR(1) Automata

S → E
E → T
E → E + T
T → int
T → (E)



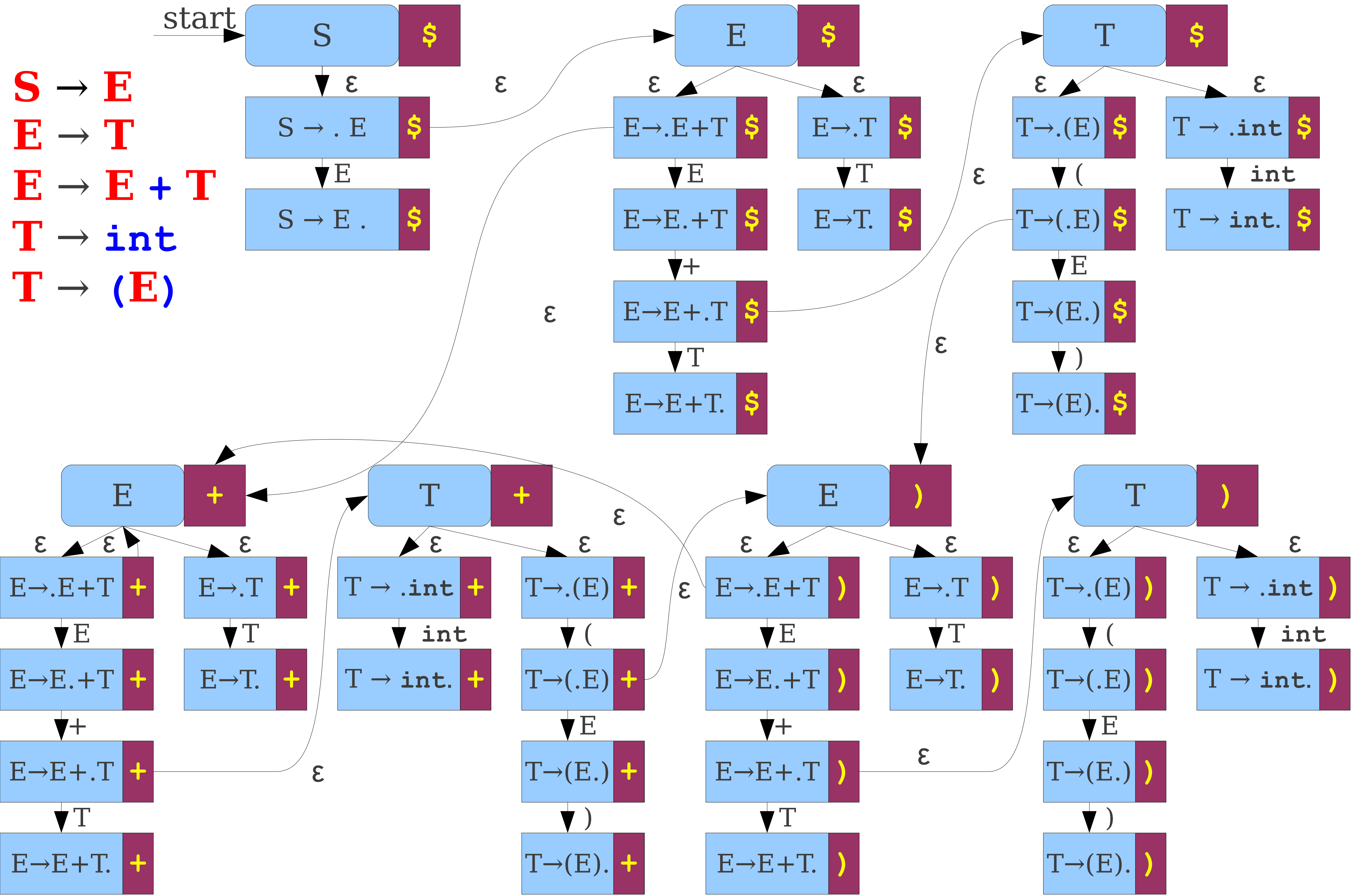
Constructing LR(1) Automata

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



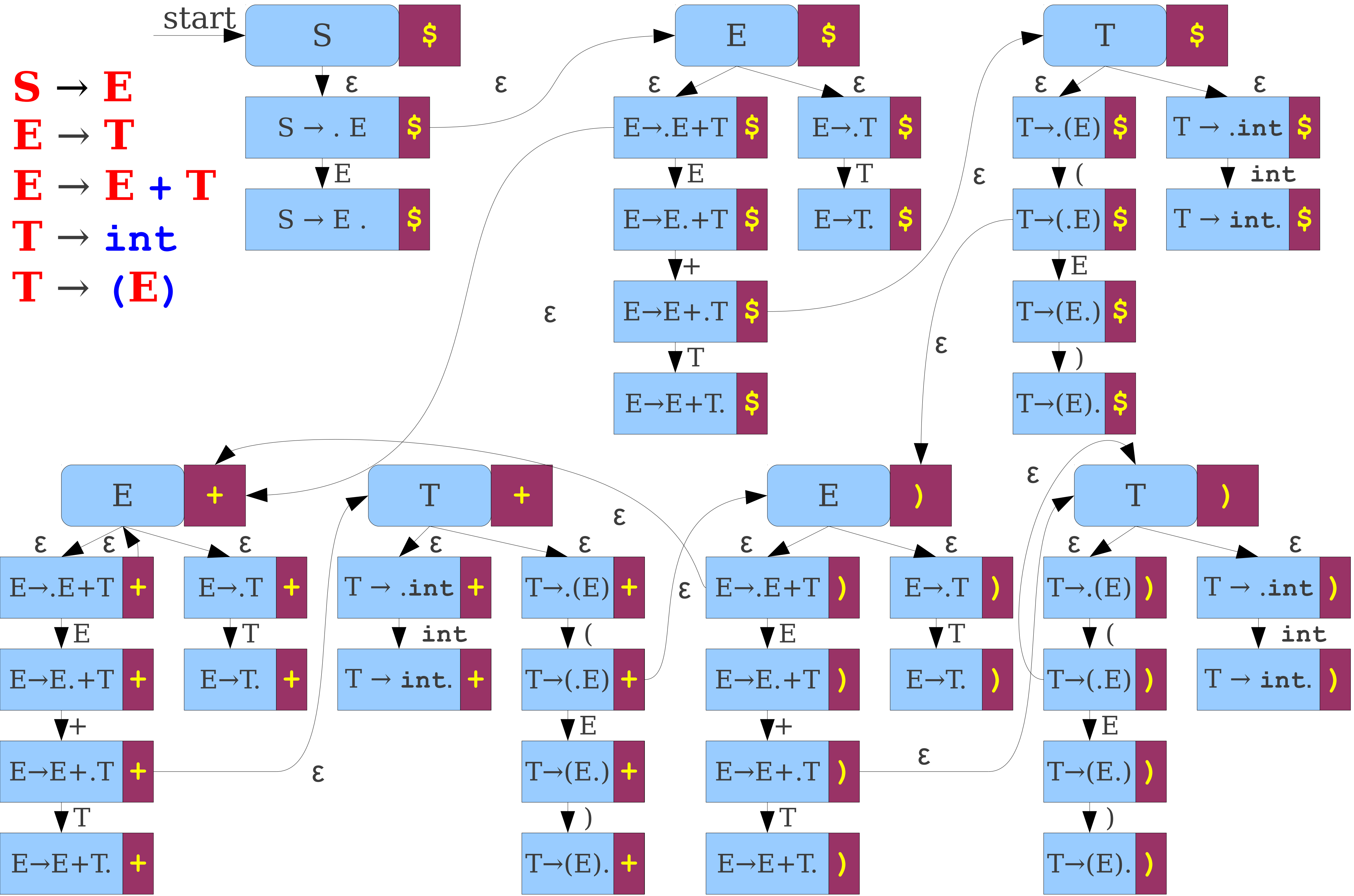
Constructing LR(1) Automata

S → E
E → T
E → E + T
T → int
T → (E)



Constructing LR(1) Automata

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



Constructing LR(1) Automata

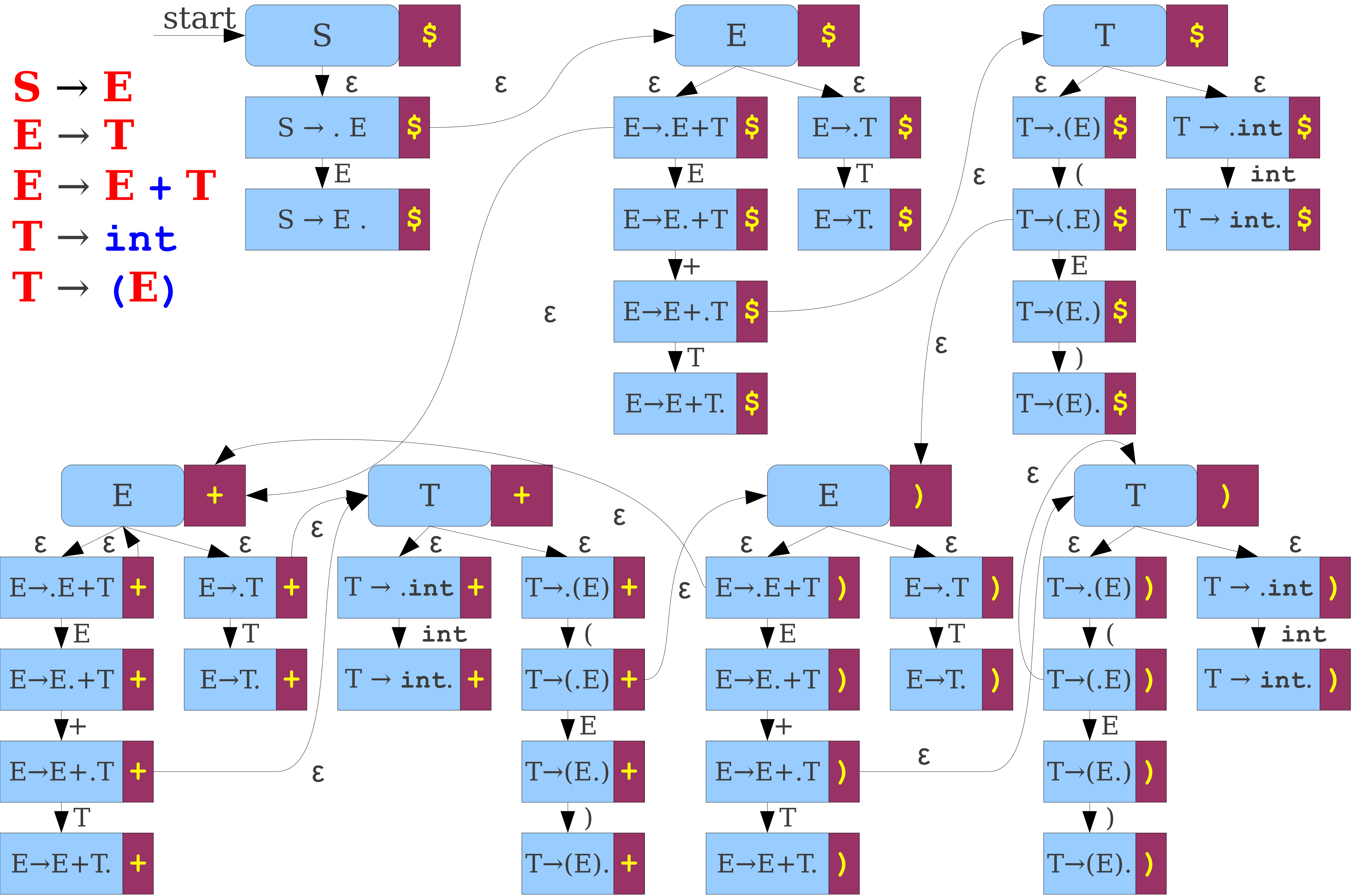
$S \rightarrow E$

$E \rightarrow T$

$E \rightarrow E + T$

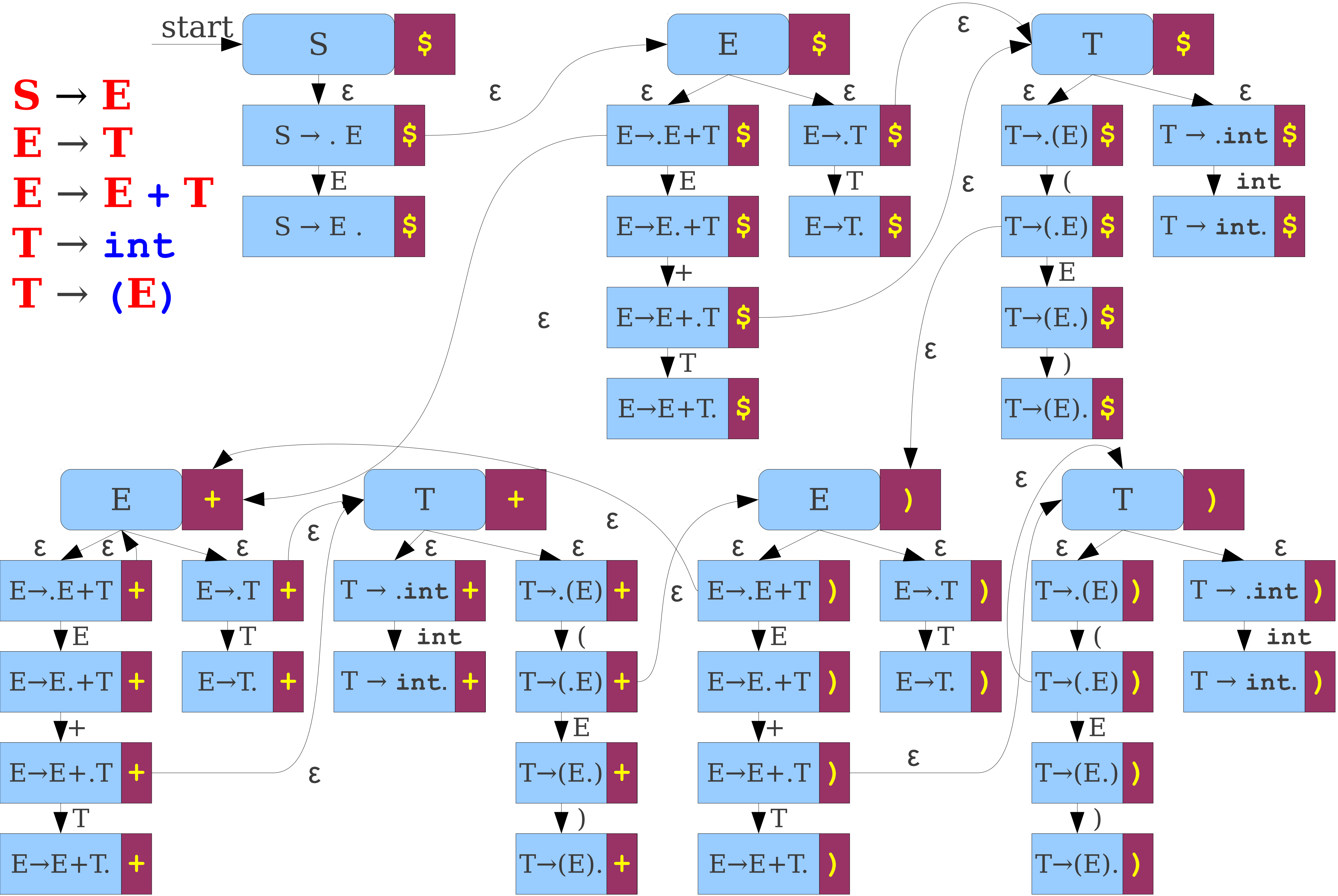
$T \rightarrow \texttt{int}$

$T \rightarrow (E)$



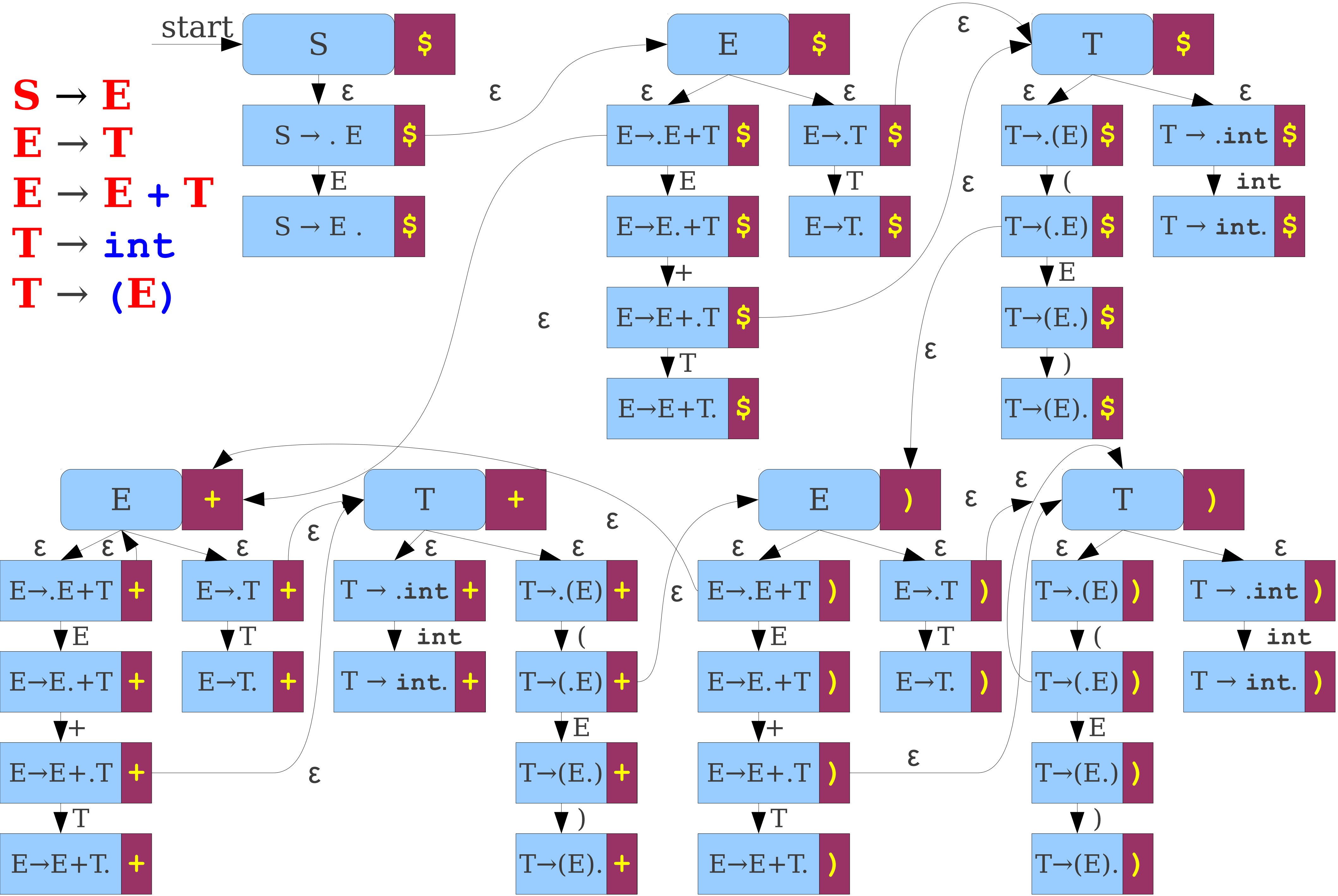
Constructing LR(1) Automata

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



Constructing LR(1) Automata

S → E
E → T
E → E + T
T → int
T → (E)

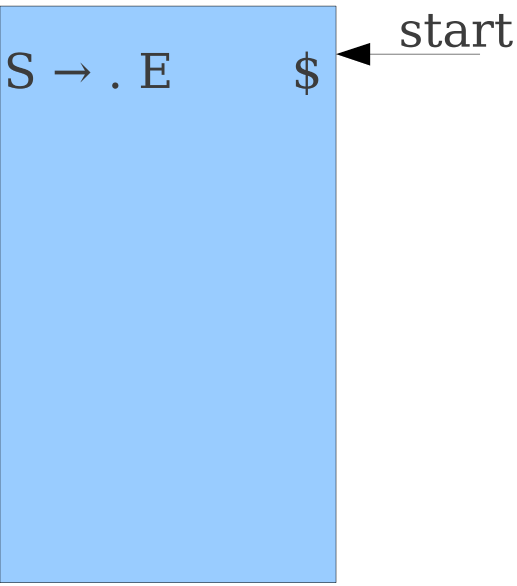


Constructing LR(1) Automata

- Begin with a state **S** [`$`].
- For each state **A** [`t`], for each production **A** → γ:
  - Construct states **A** → $\alpha \cdot \omega$ [`t`] for all possible ways of splitting $\gamma = \alpha\omega$.
  - Add an ε-transition from **A** [`t`] to each of these states.
  - Add transitions on x between **A** → $\alpha \cdot x\omega$ [`t`] and **A** → $\alpha x \cdot \omega$ [`t`]
- For each state **A** → $\alpha \cdot$ **B**ω [`t`], add an ε-transition from **A** → $\alpha \cdot$ **B**ω [`t`] to **B** [`r`] for each terminal `r` ∈ FIRST*(ω`t`).

Deterministic LR(1) Automata

Deterministic LR(1) Automata



Deterministic LR(1) Automata

start

$S \rightarrow .\ E \quad \$$
$E \rightarrow .\ T \quad \$$
$E \rightarrow .\ E + T \quad \$$

Deterministic LR(1) Automata

start

$S \rightarrow .\ E$	$\$$
$E \rightarrow .\ T$	$\$$
$E \rightarrow .\ E + T$	$\$$
$E \rightarrow .\ T$	$+$
$E \rightarrow .\ E + T$	$+$

Deterministic LR(1) Automata

start

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

Deterministic LR(1) Automata

start

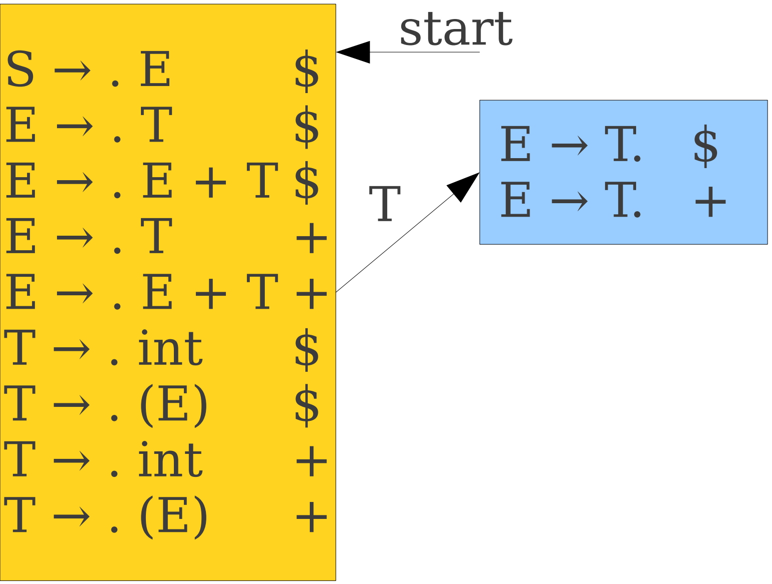
$S \rightarrow .\ E$	\$
$E \rightarrow .\ T$	\$
$E \rightarrow .\ E + T$	\$
$E \rightarrow .\ T$	+
$E \rightarrow .\ E + T$	+
$T \rightarrow .\ \text{int}$	\$
$T \rightarrow .\ (E)$	\$
$T \rightarrow .\ \text{int}$	+
$T \rightarrow .\ (E)$	+

Deterministic LR(1) Automata

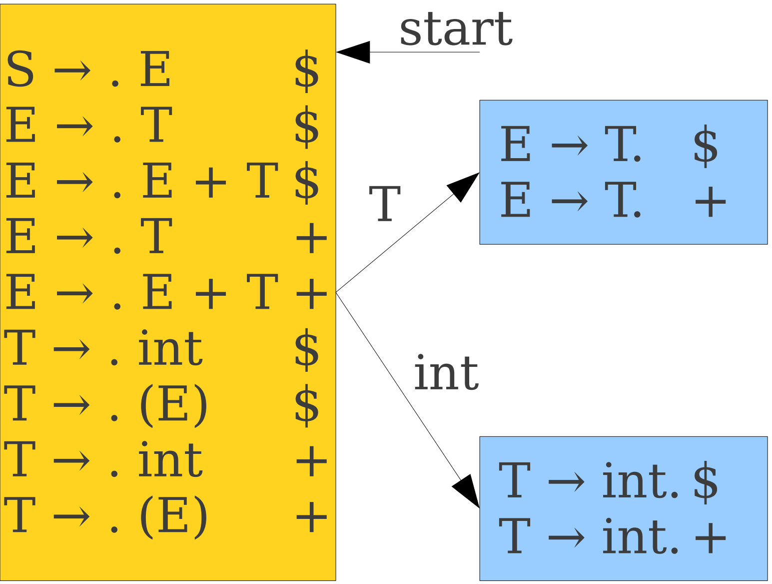
start

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

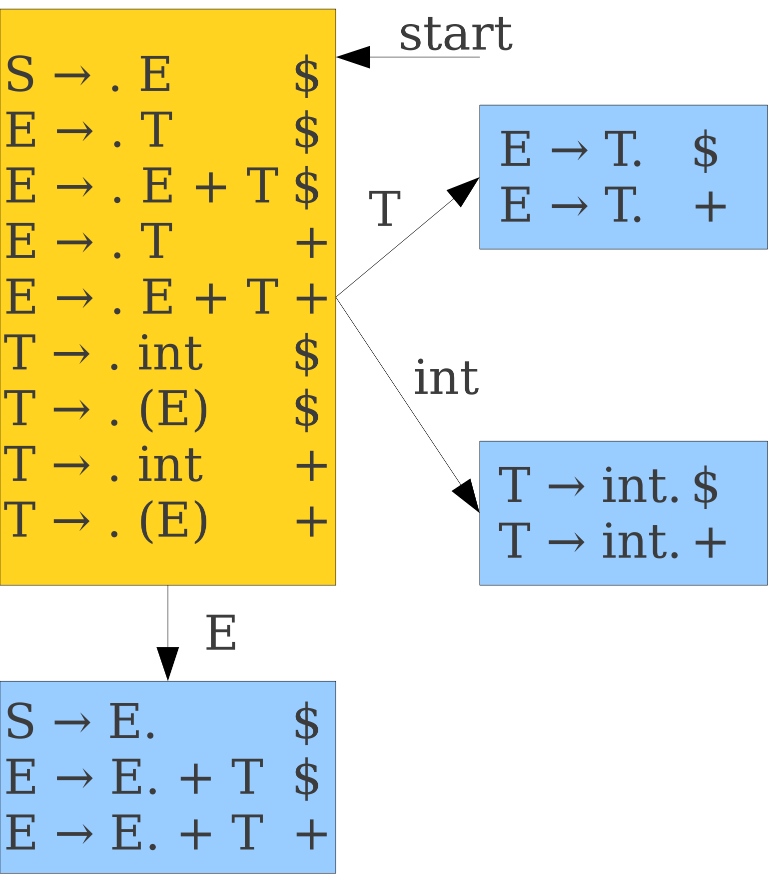
Deterministic LR(1) Automata



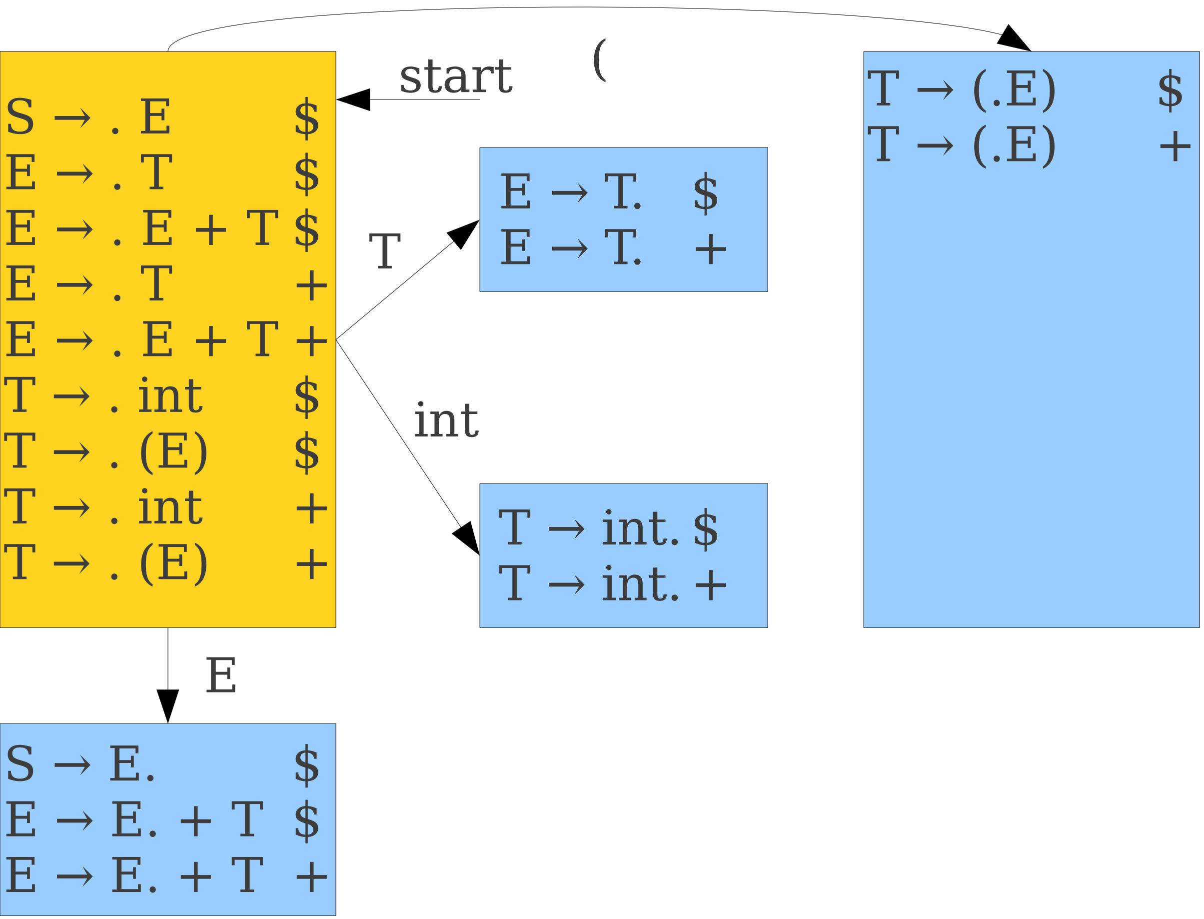
Deterministic LR(1) Automata



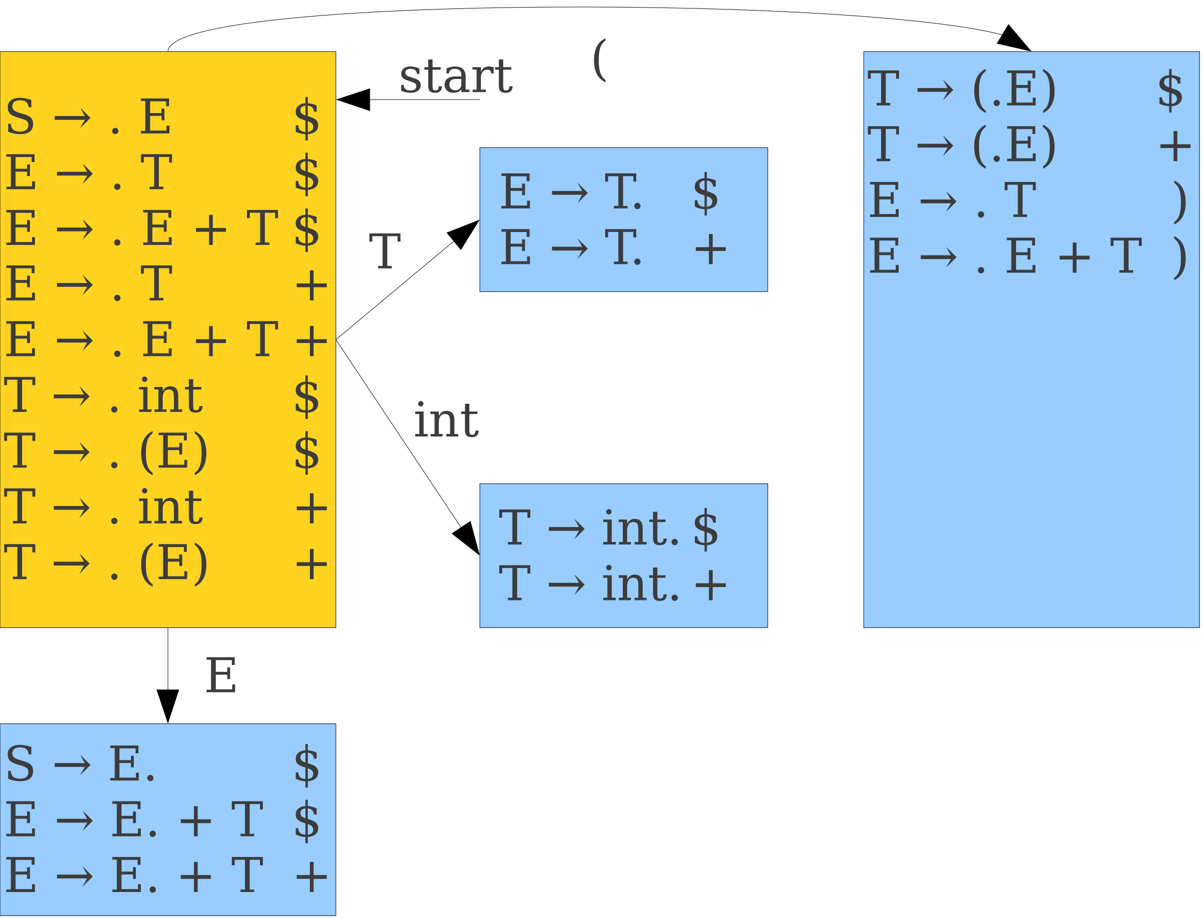
Deterministic LR(1) Automata



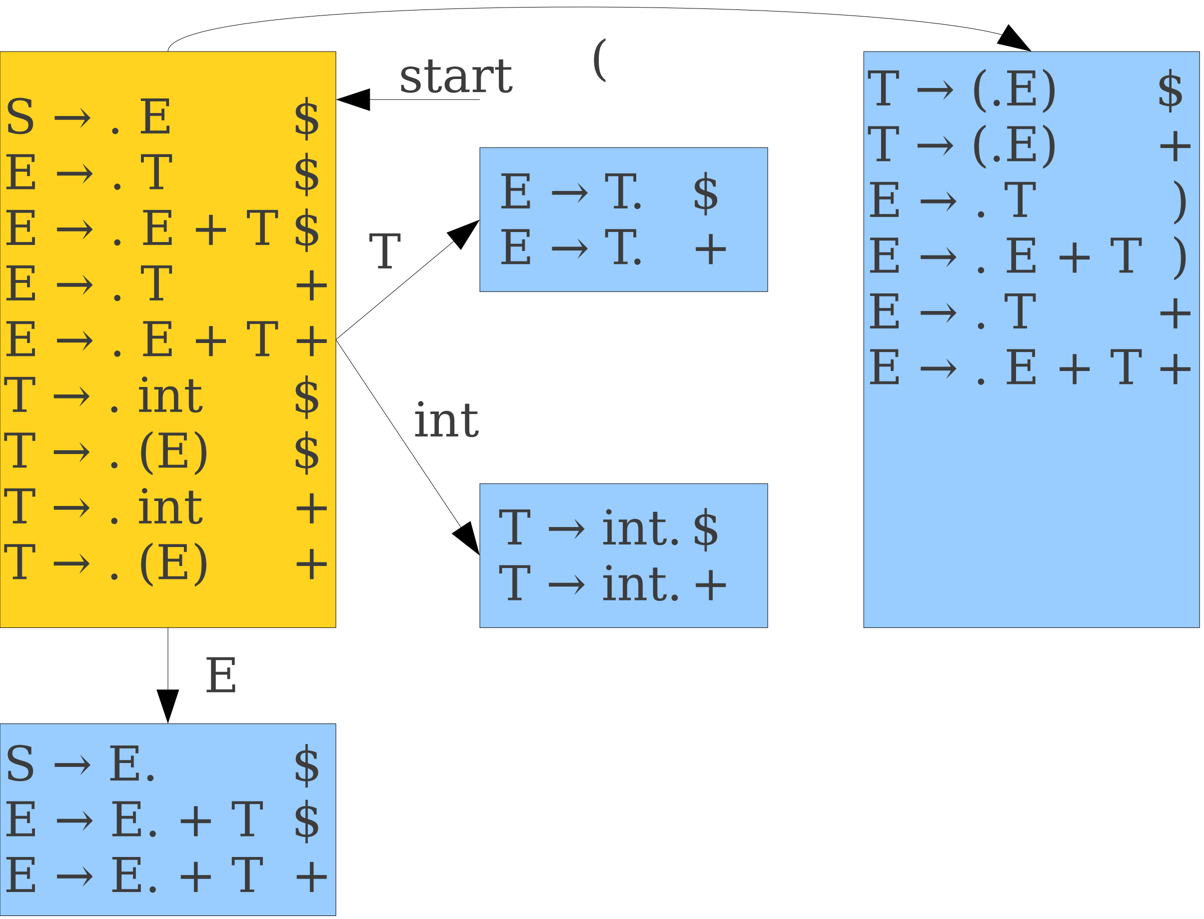
Deterministic LR(1) Automata



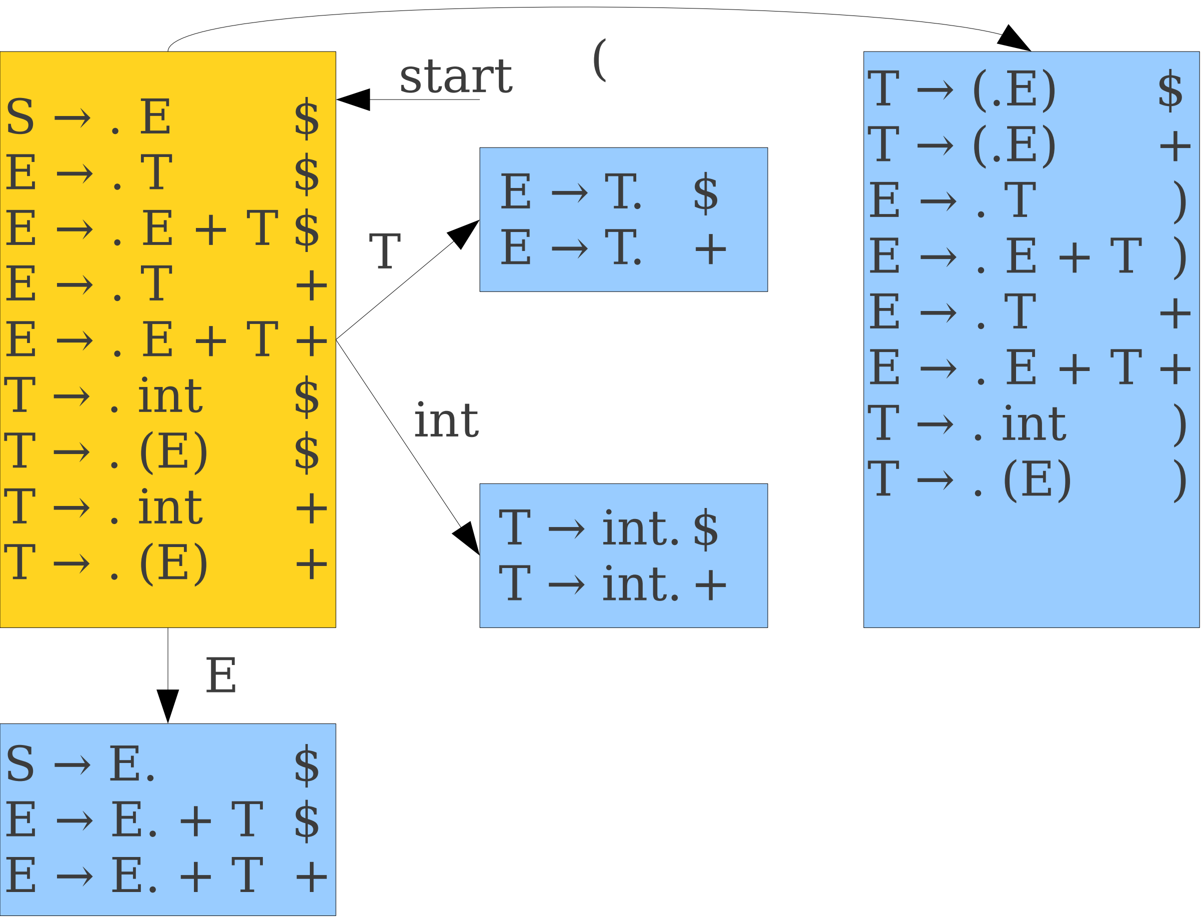
Deterministic LR(1) Automata



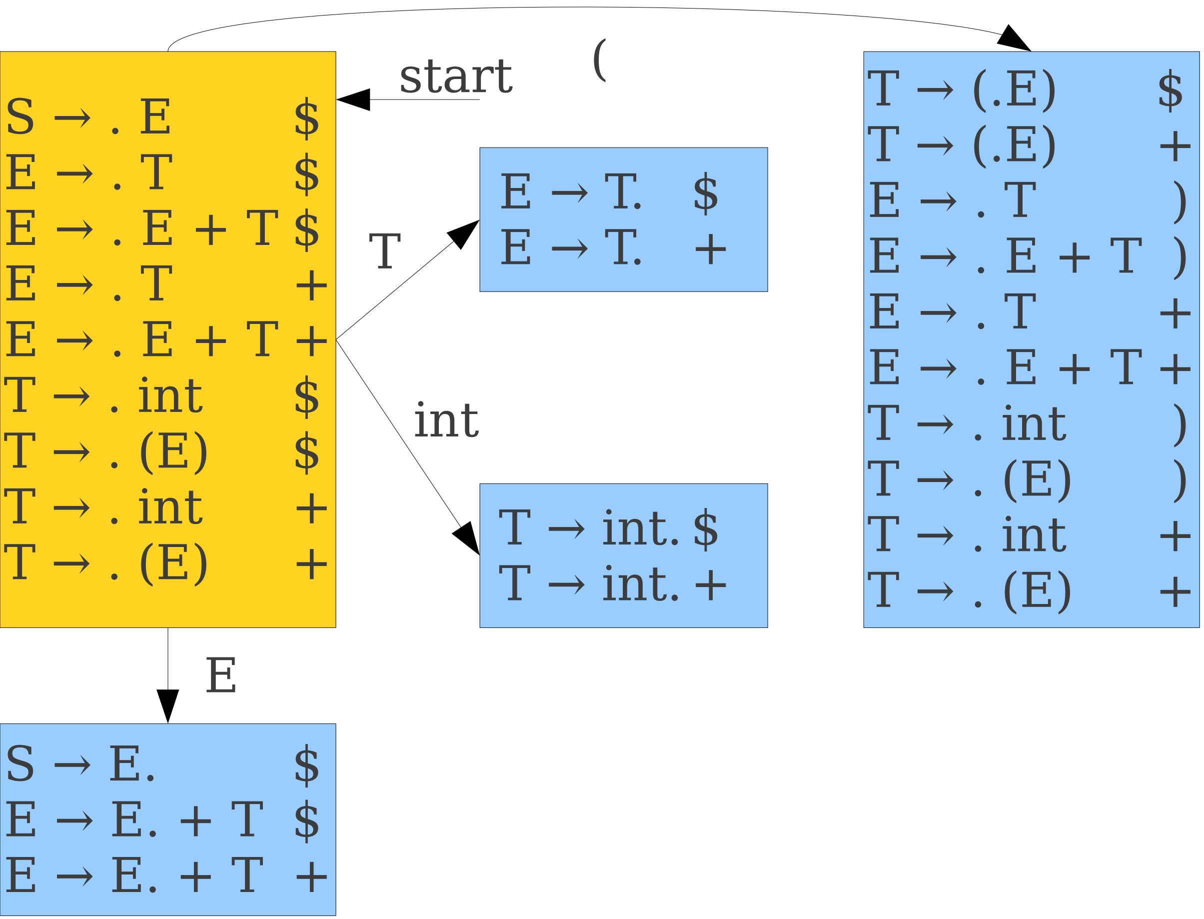
Deterministic LR(1) Automata



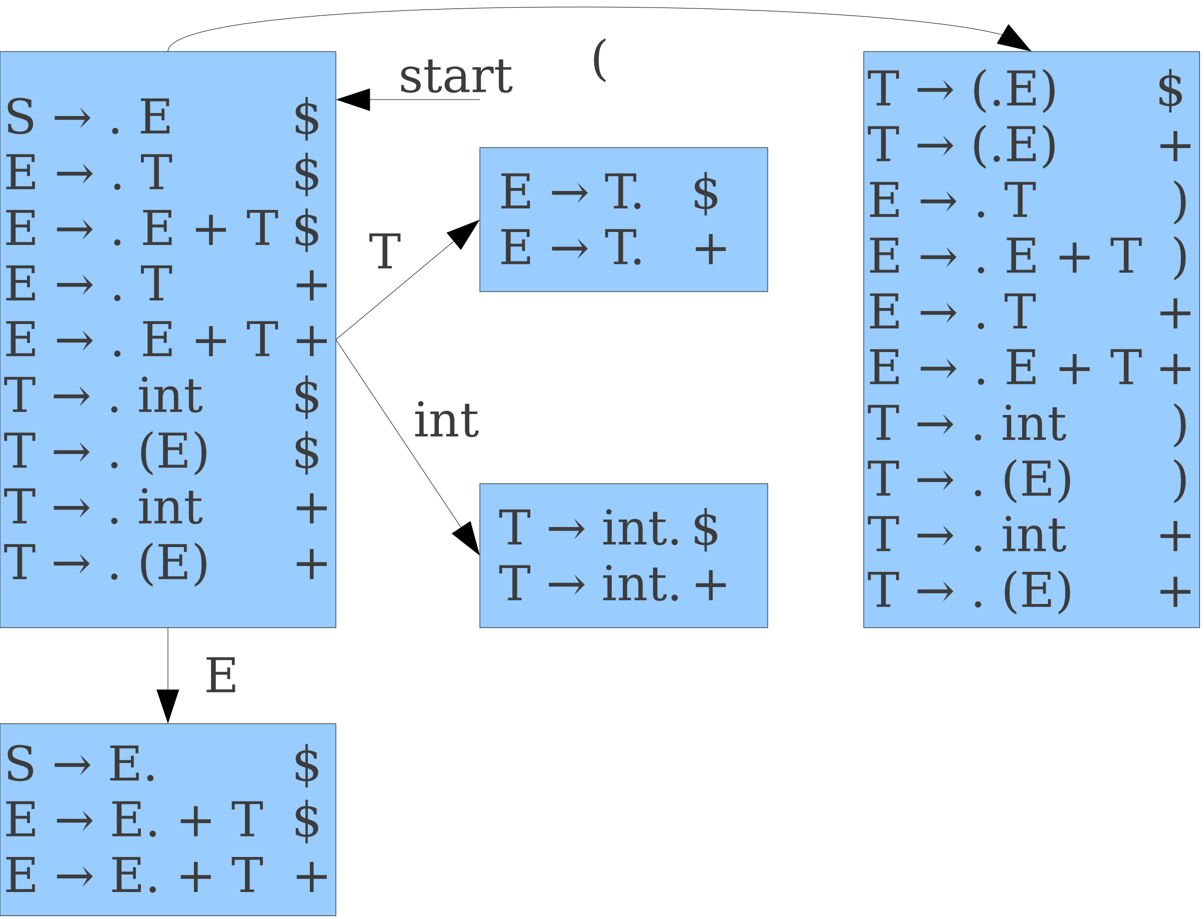
Deterministic LR(1) Automata



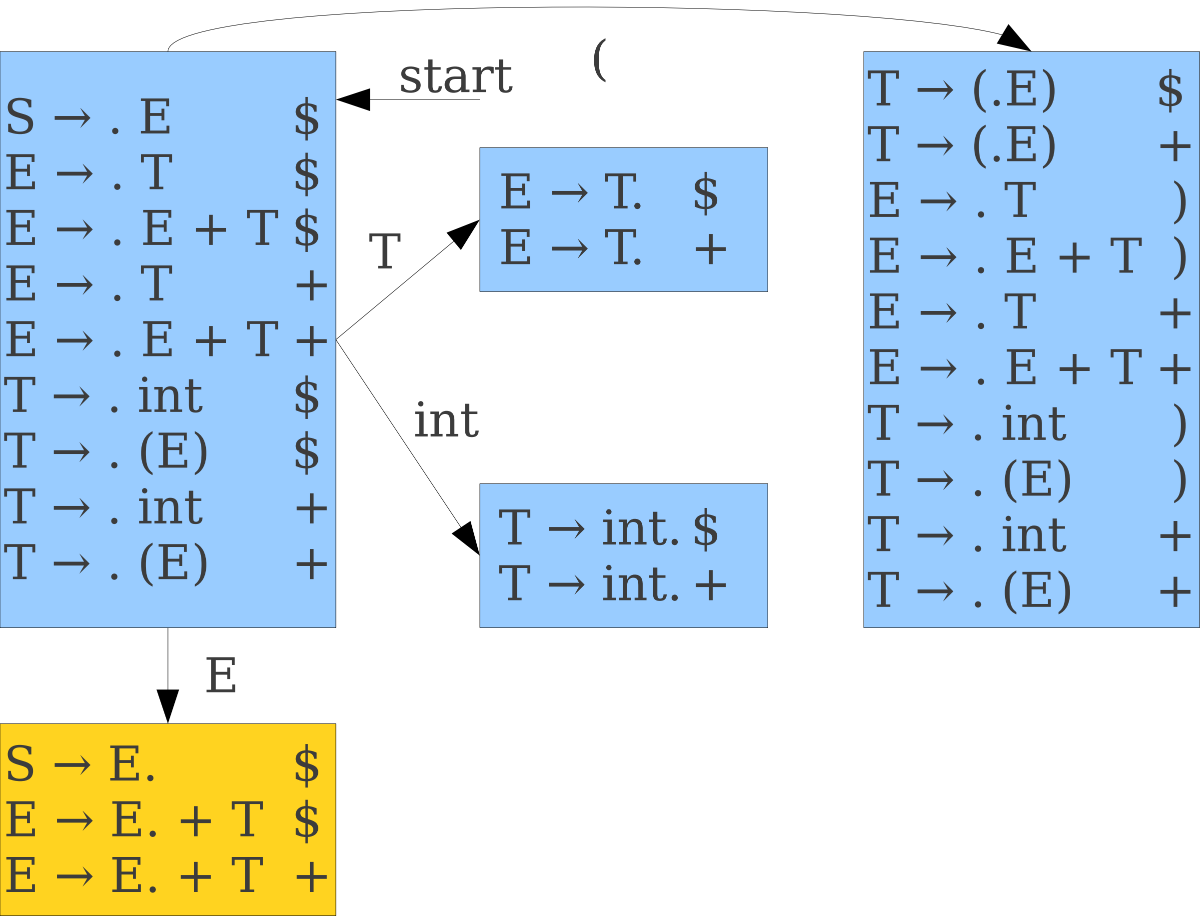
Deterministic LR(1) Automata



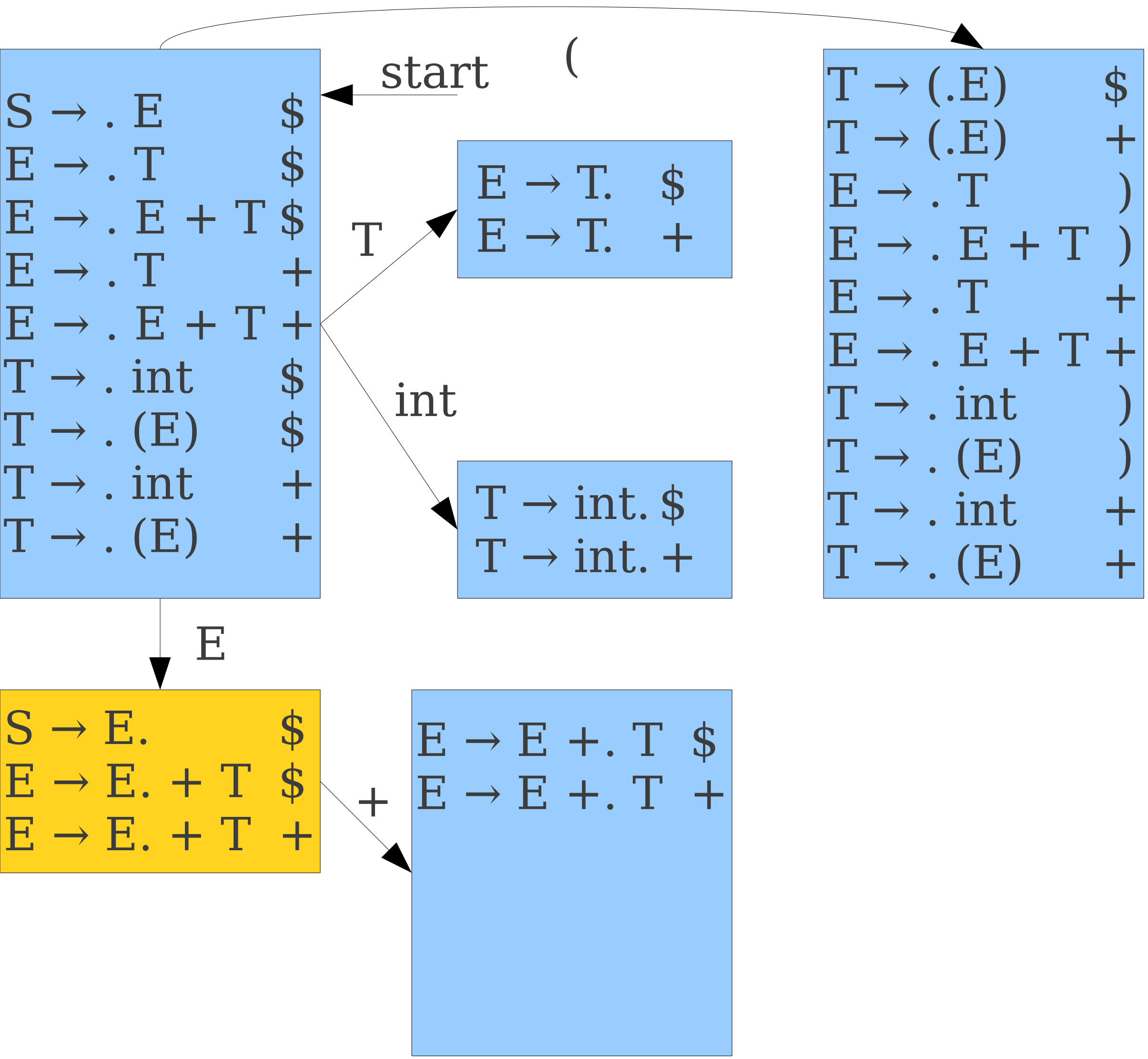
Deterministic LR(1) Automata



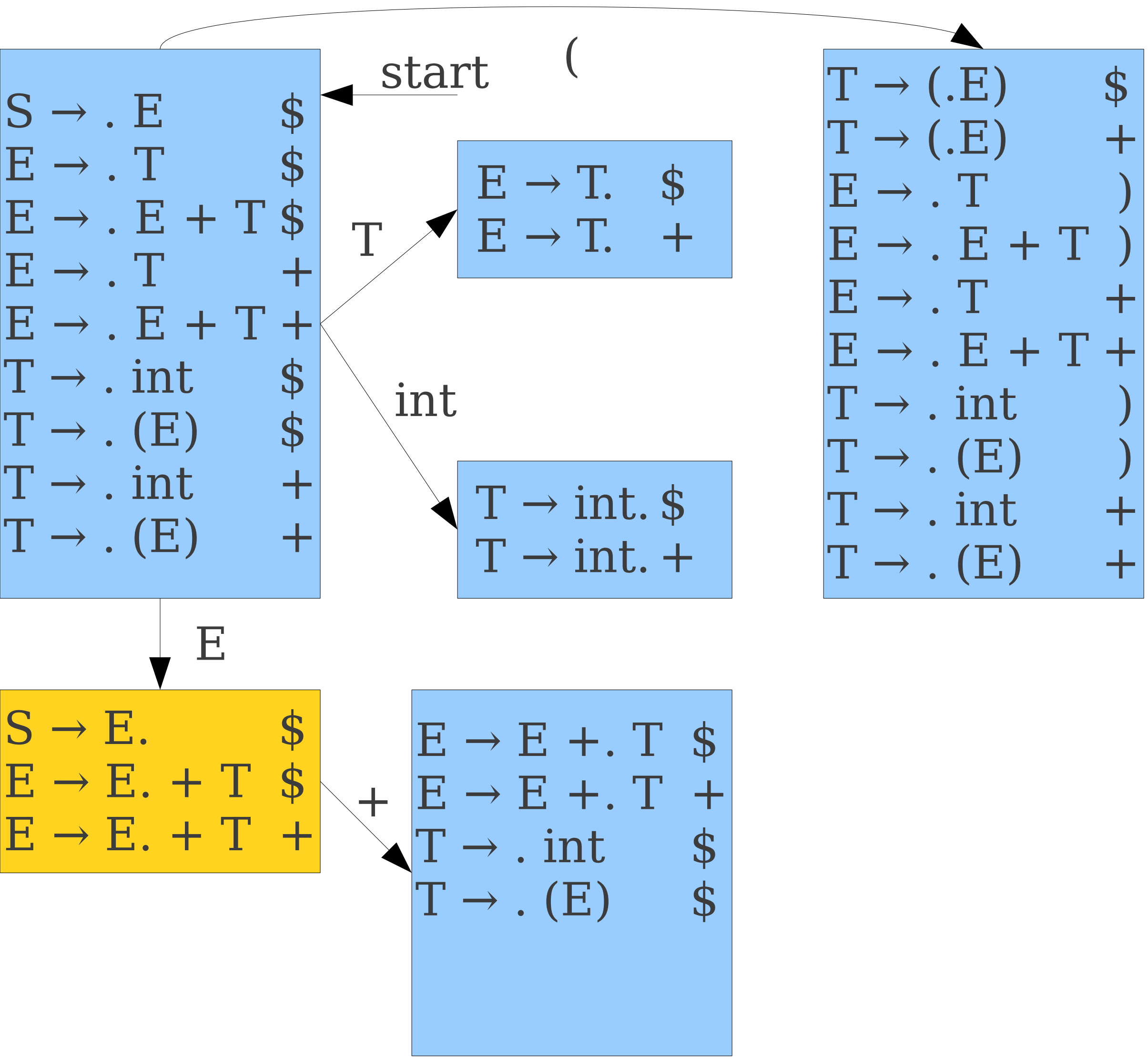
Deterministic LR(1) Automata



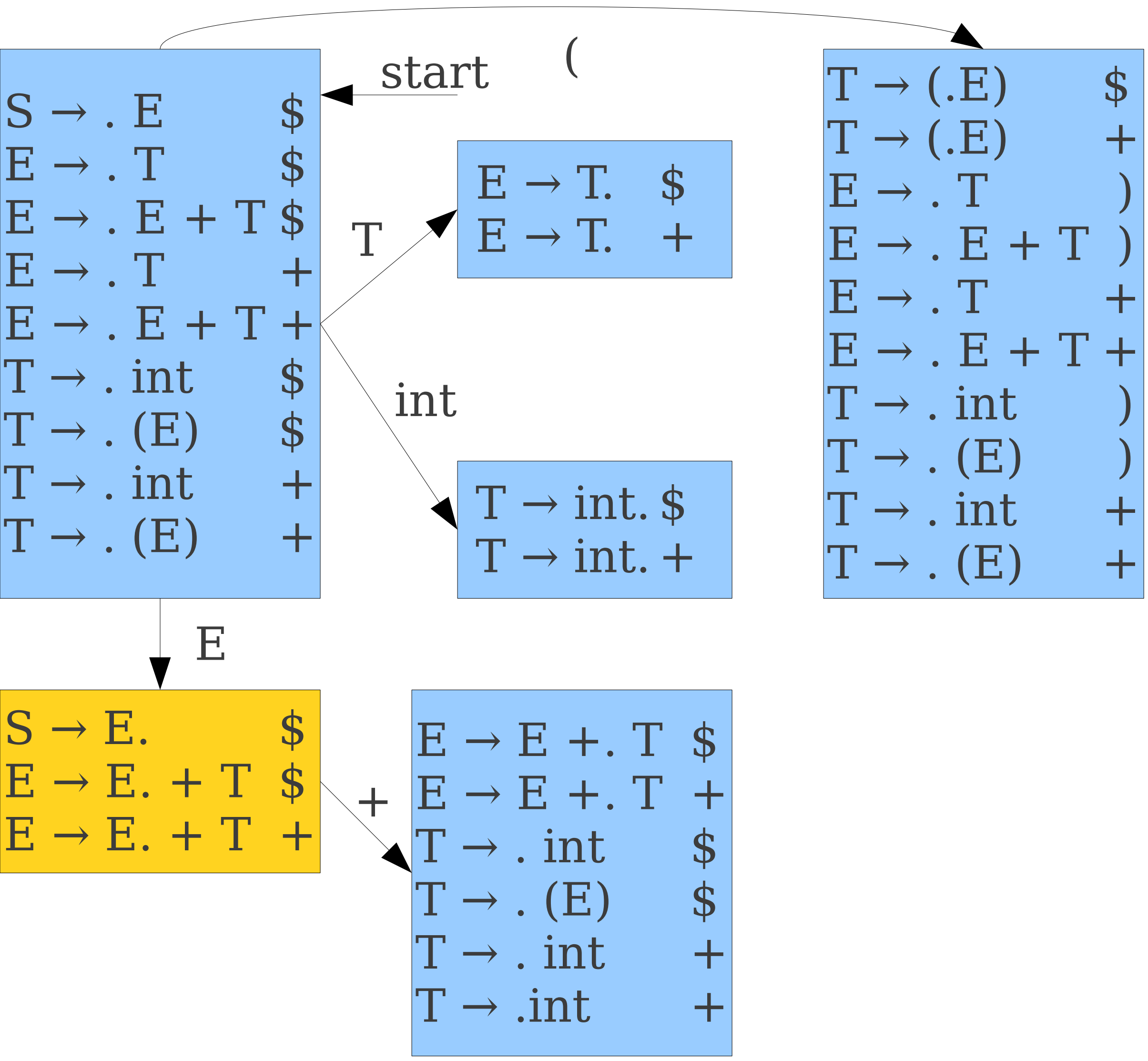
Deterministic LR(1) Automata



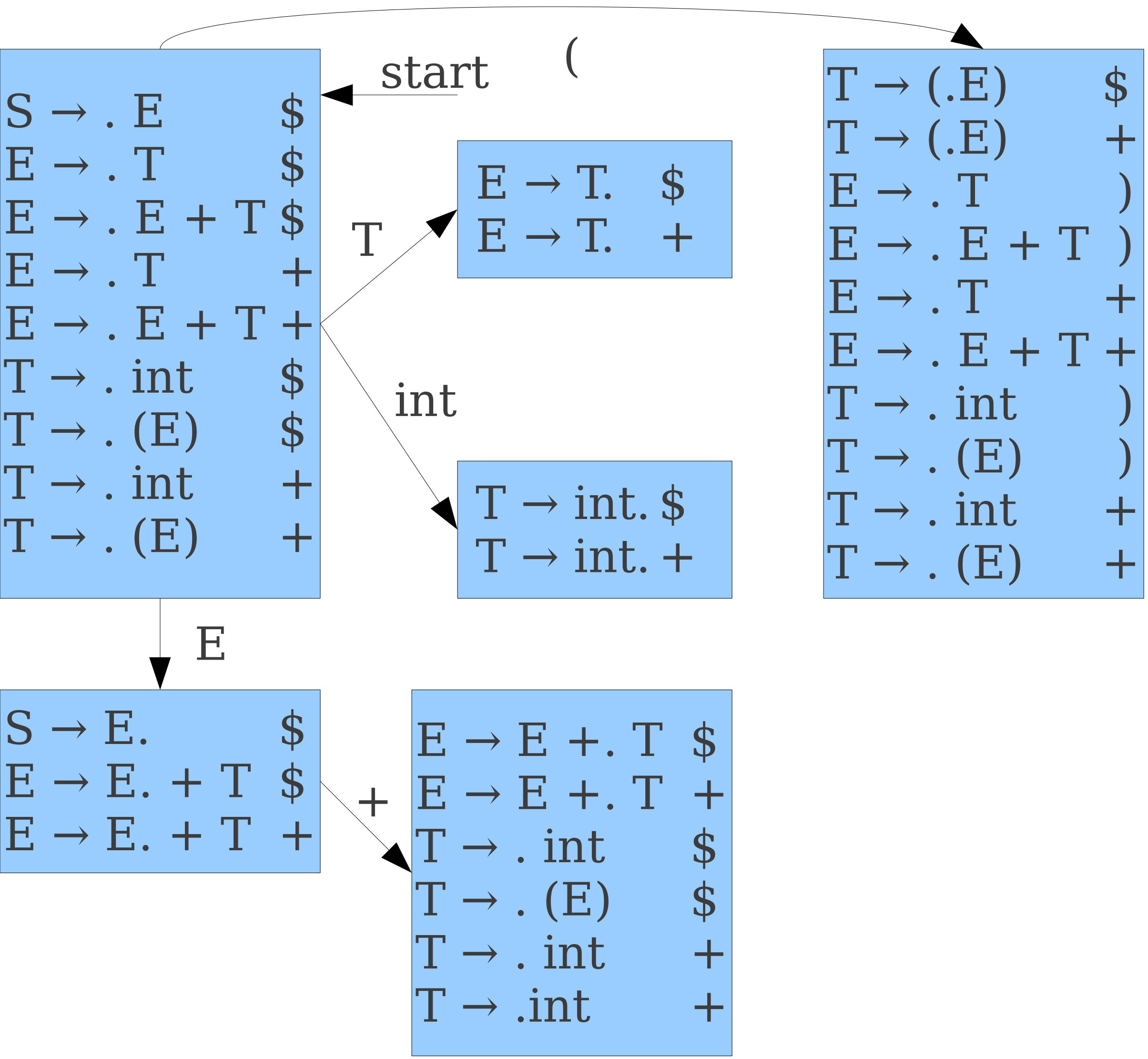
Deterministic LR(1) Automata



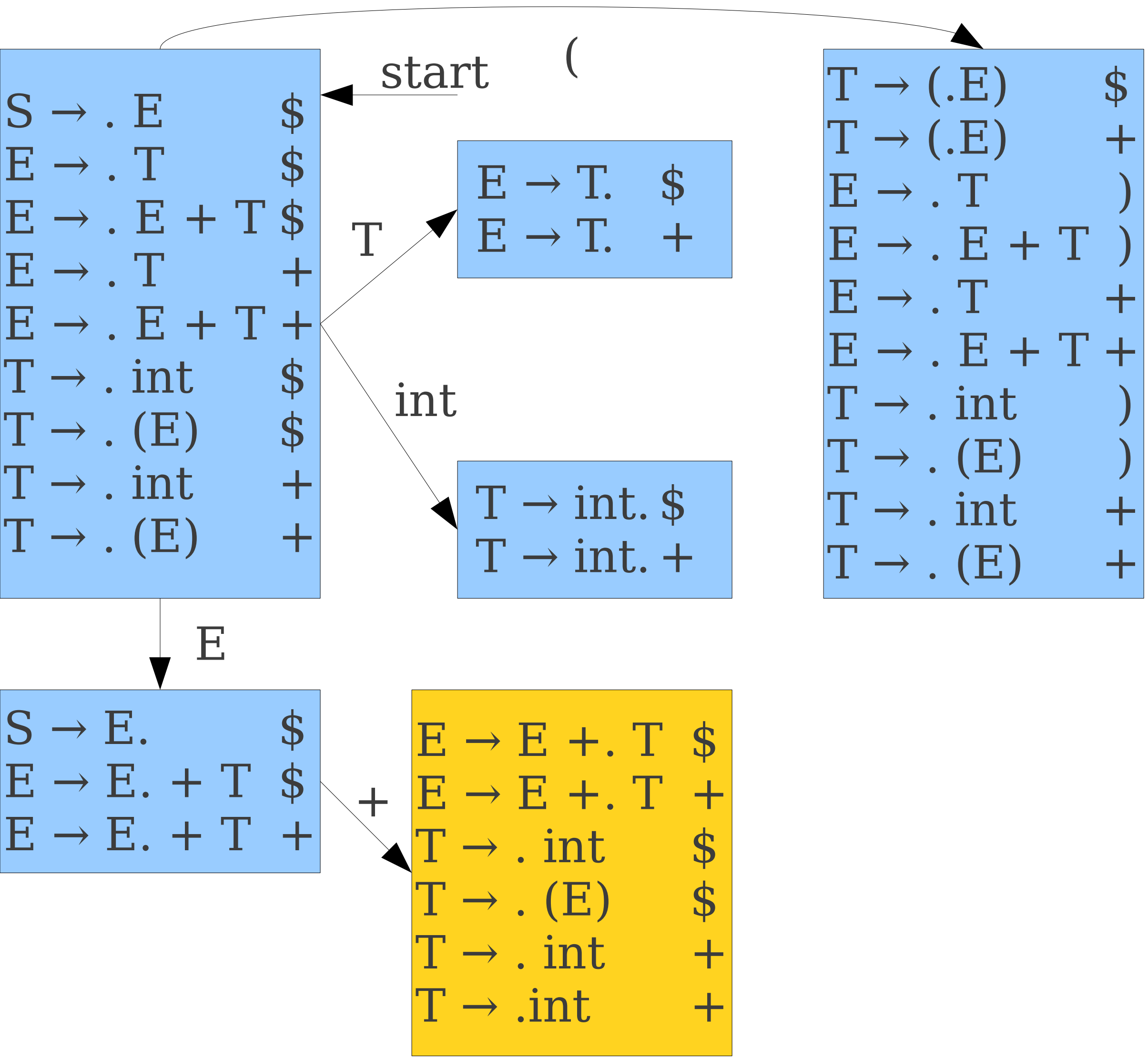
Deterministic LR(1) Automata



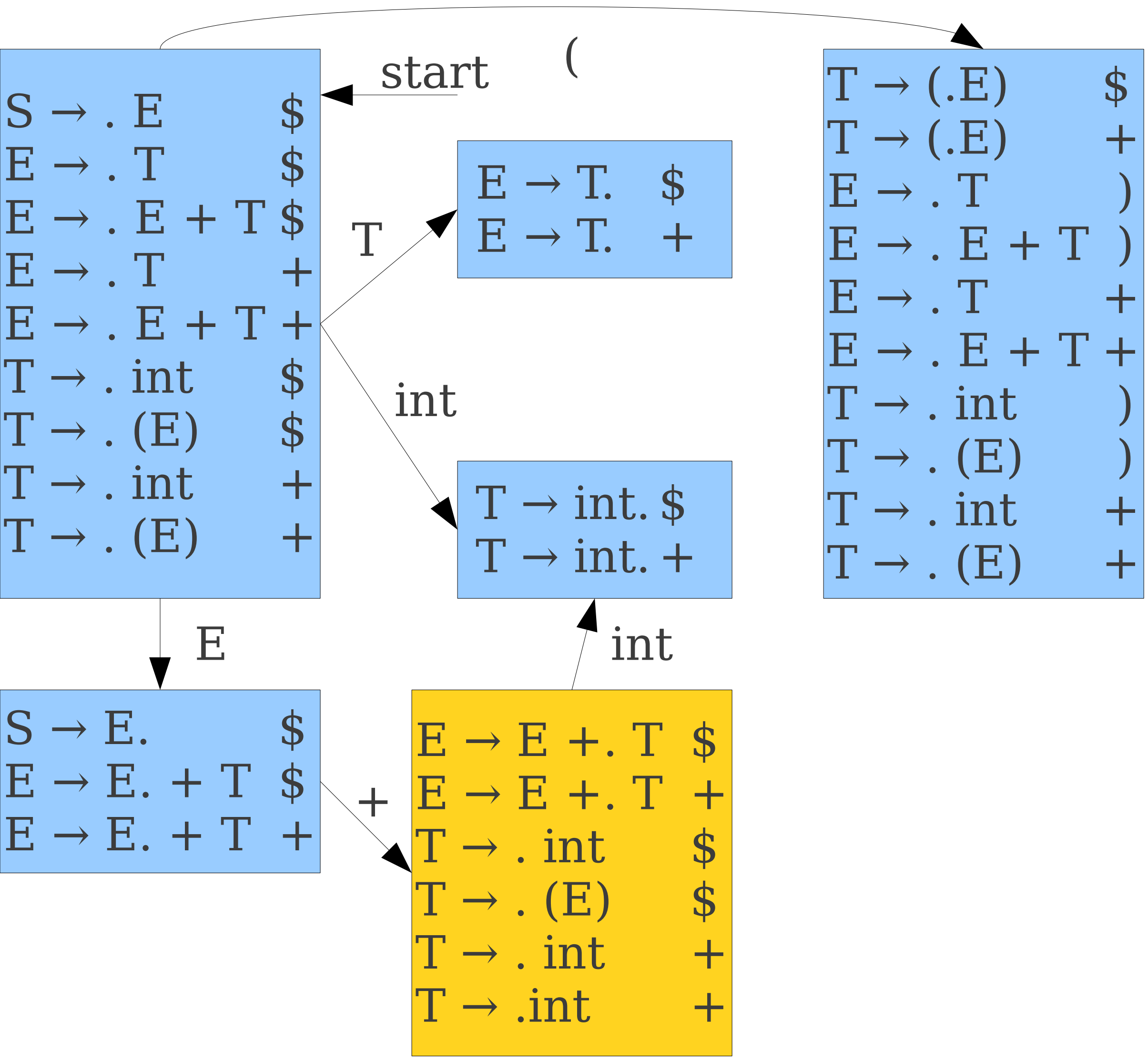
Deterministic LR(1) Automata



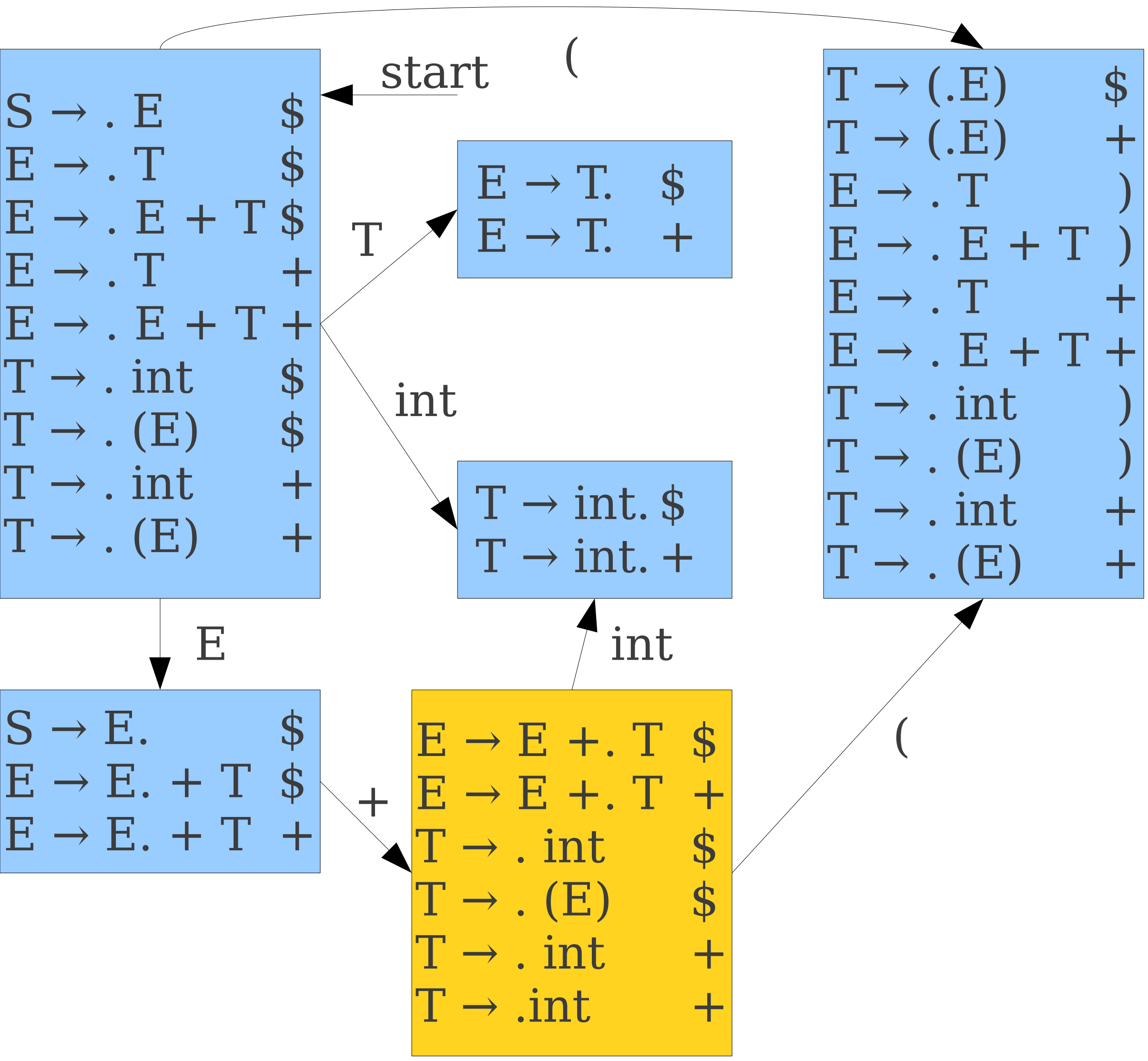
Deterministic LR(1) Automata



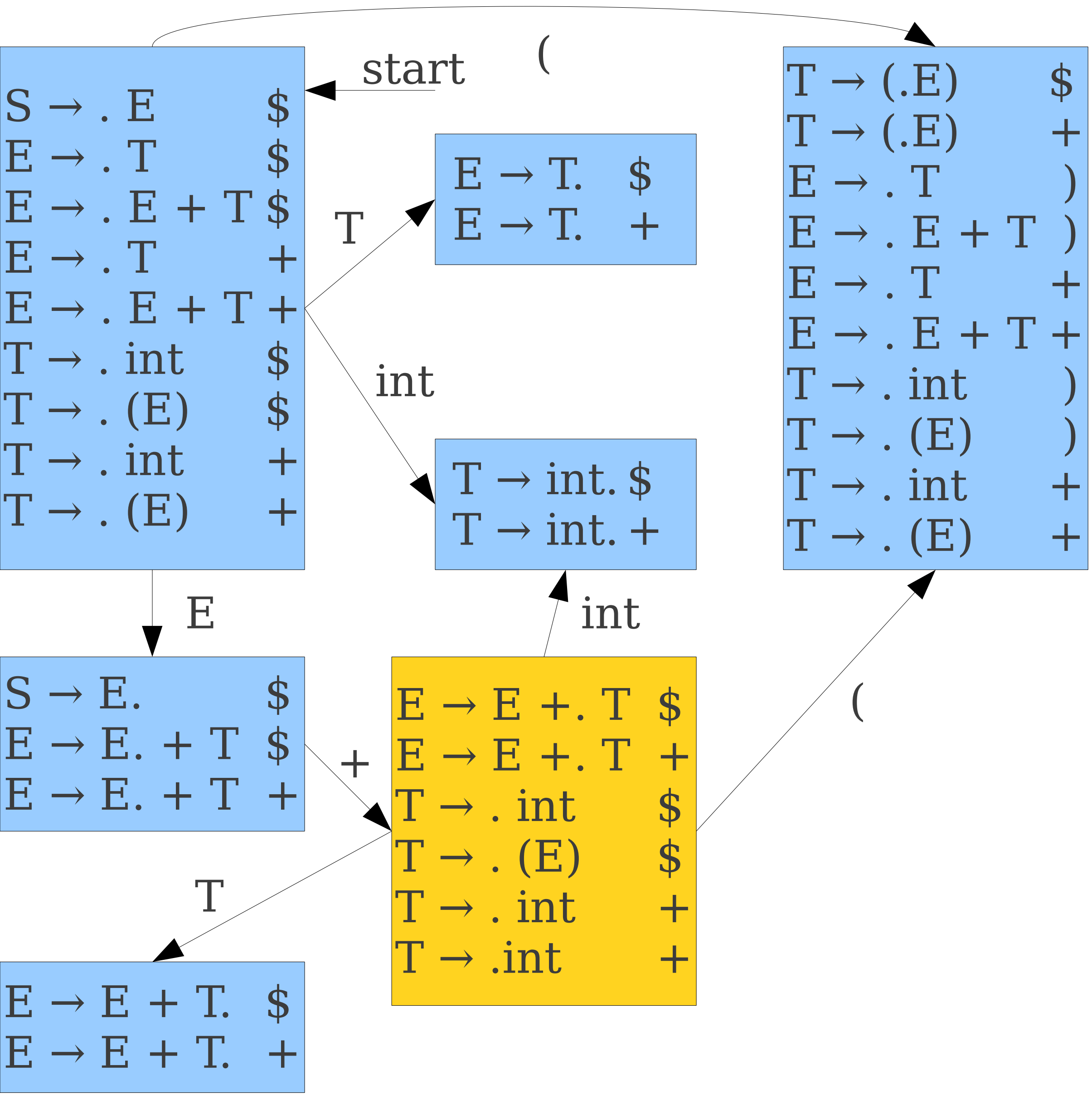
Deterministic LR(1) Automata



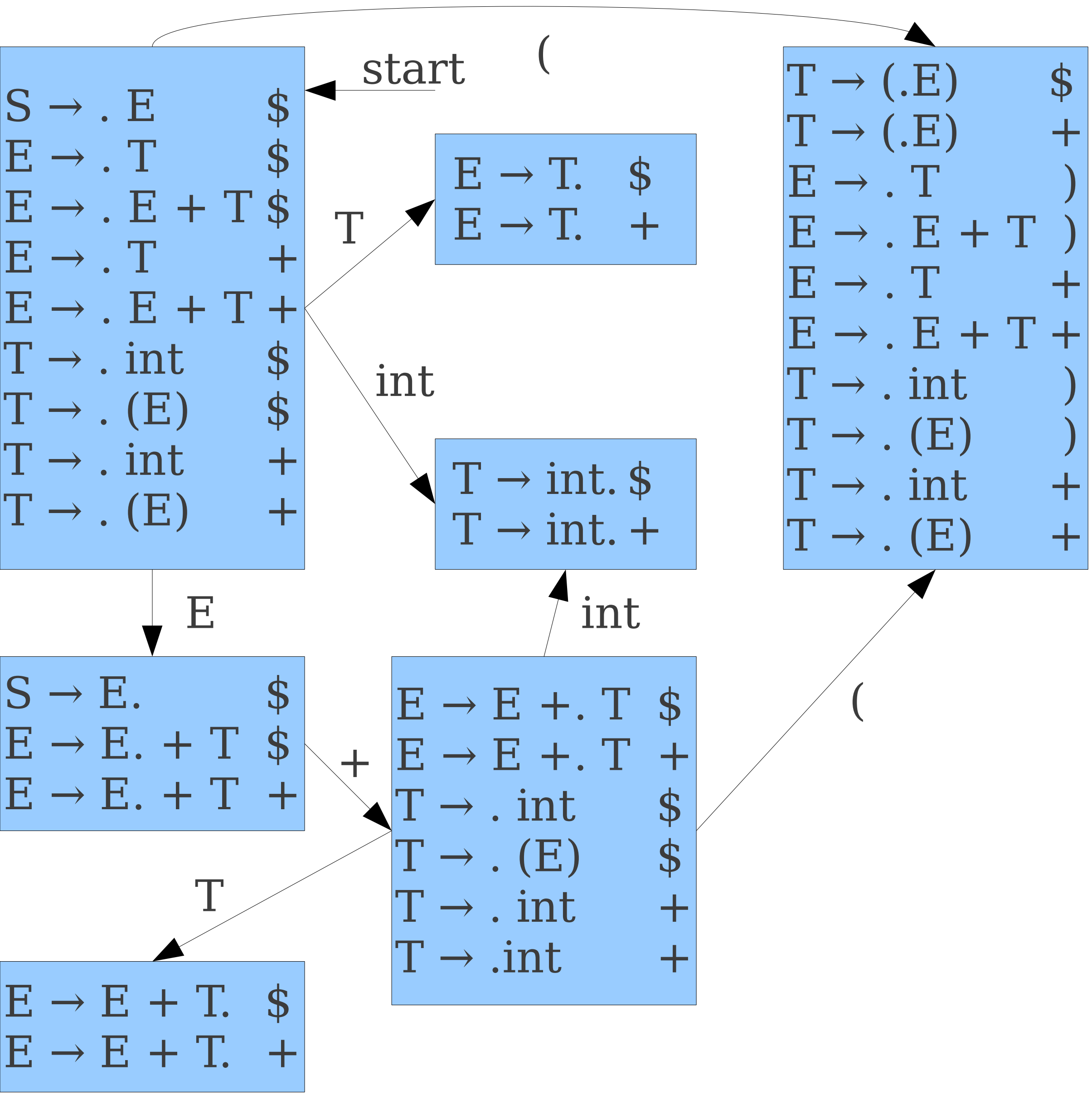
Deterministic LR(1) Automata



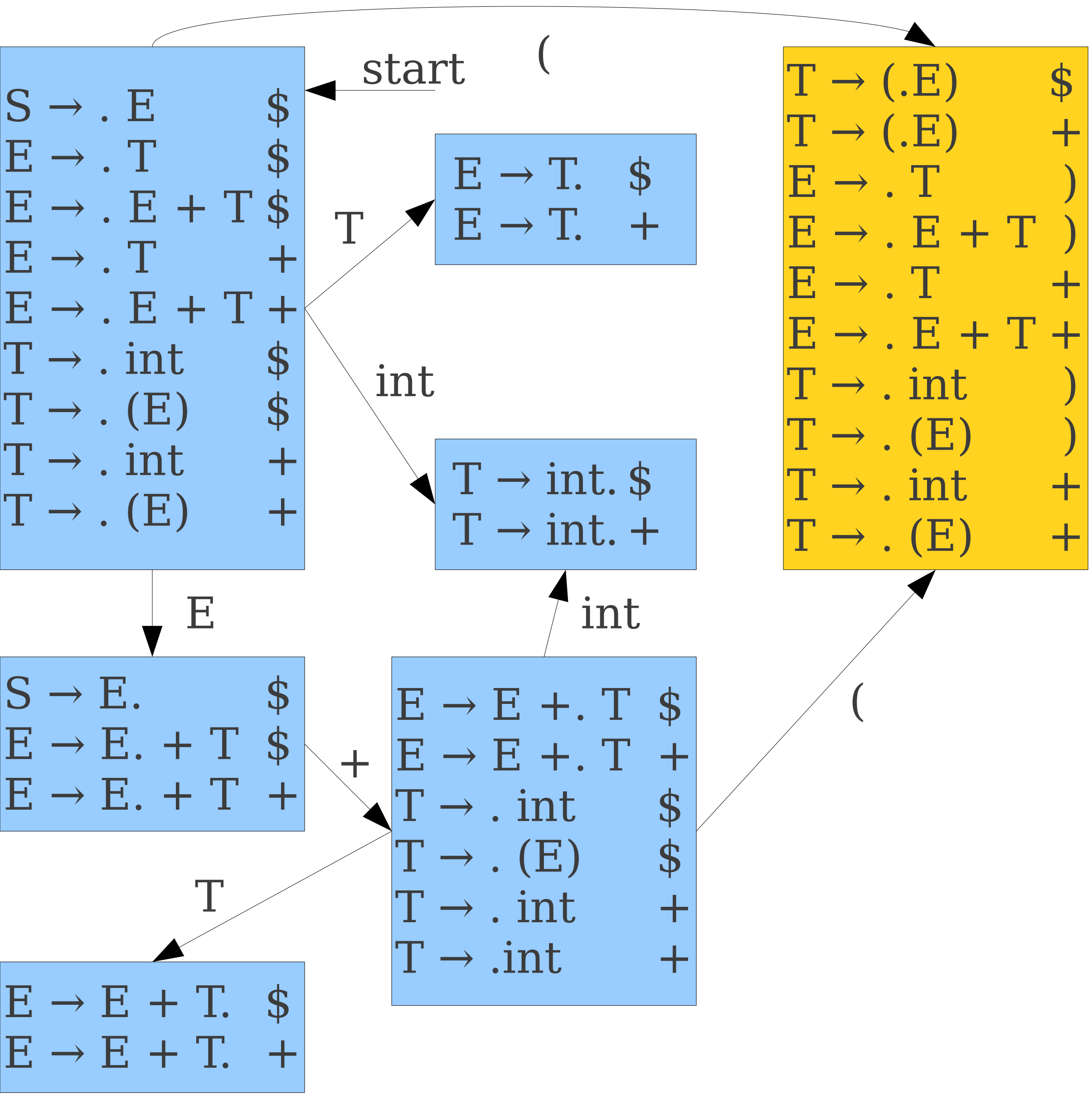
Deterministic LR(1) Automata



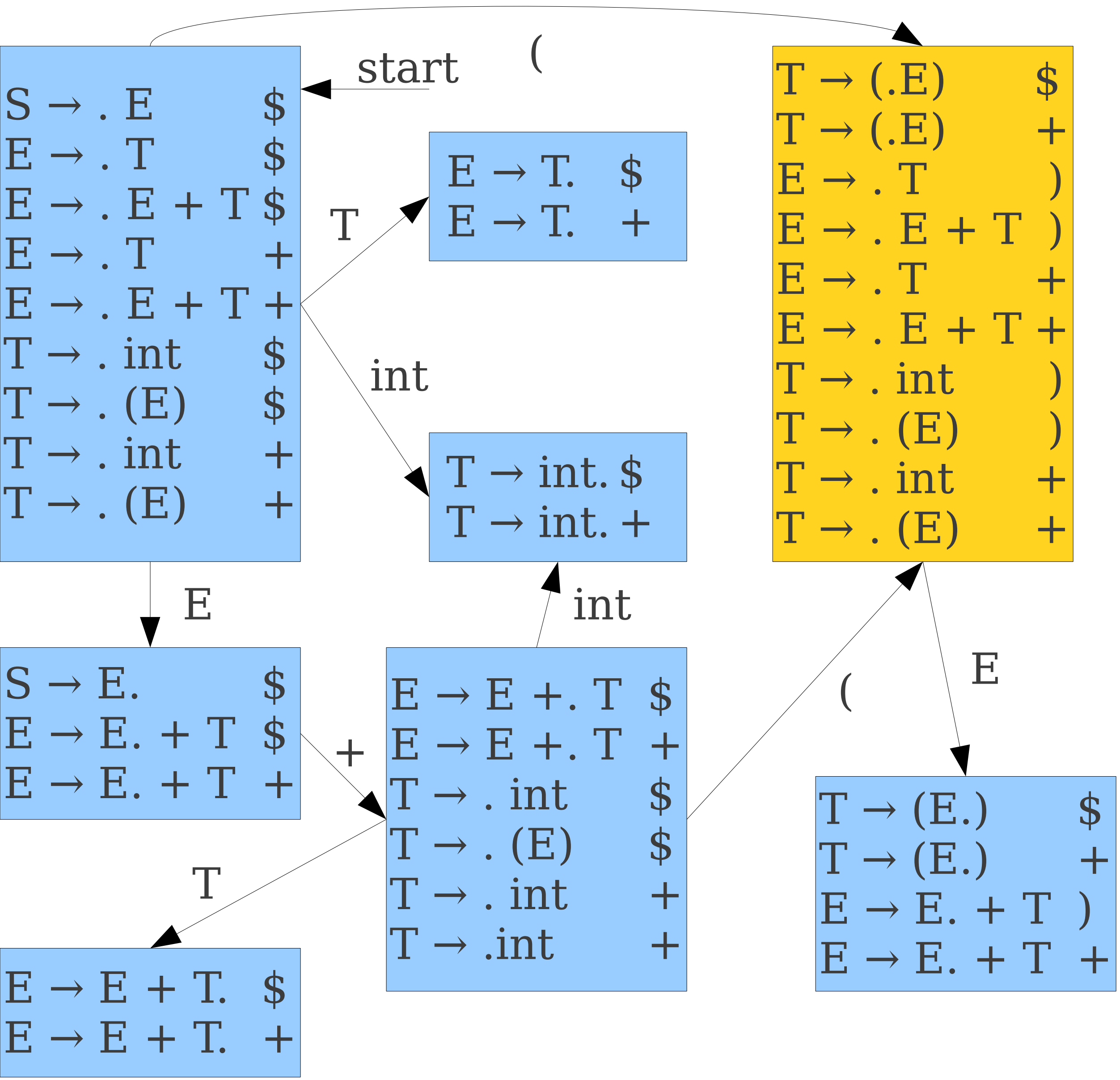
Deterministic LR(1) Automata



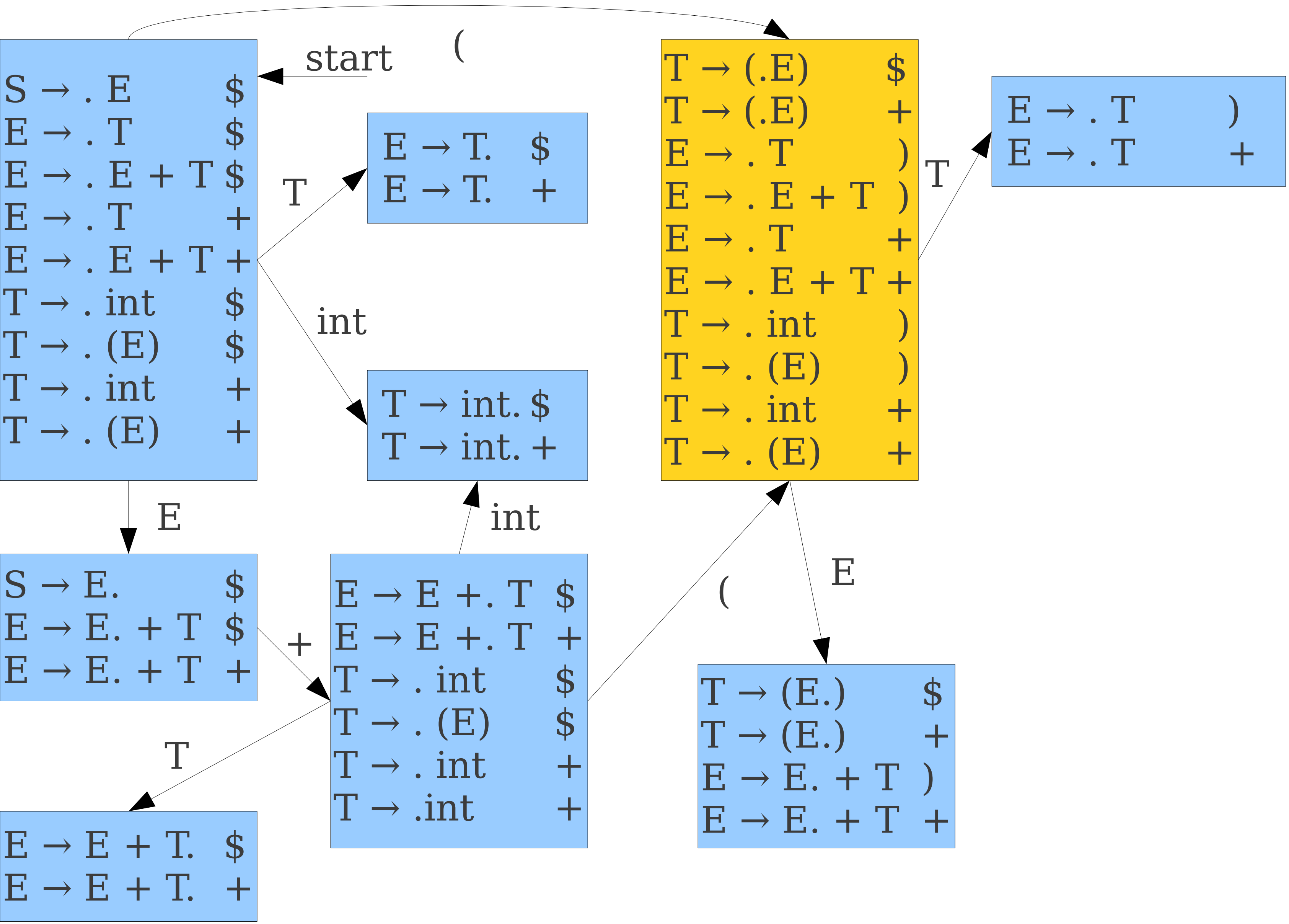
Deterministic LR(1) Automata



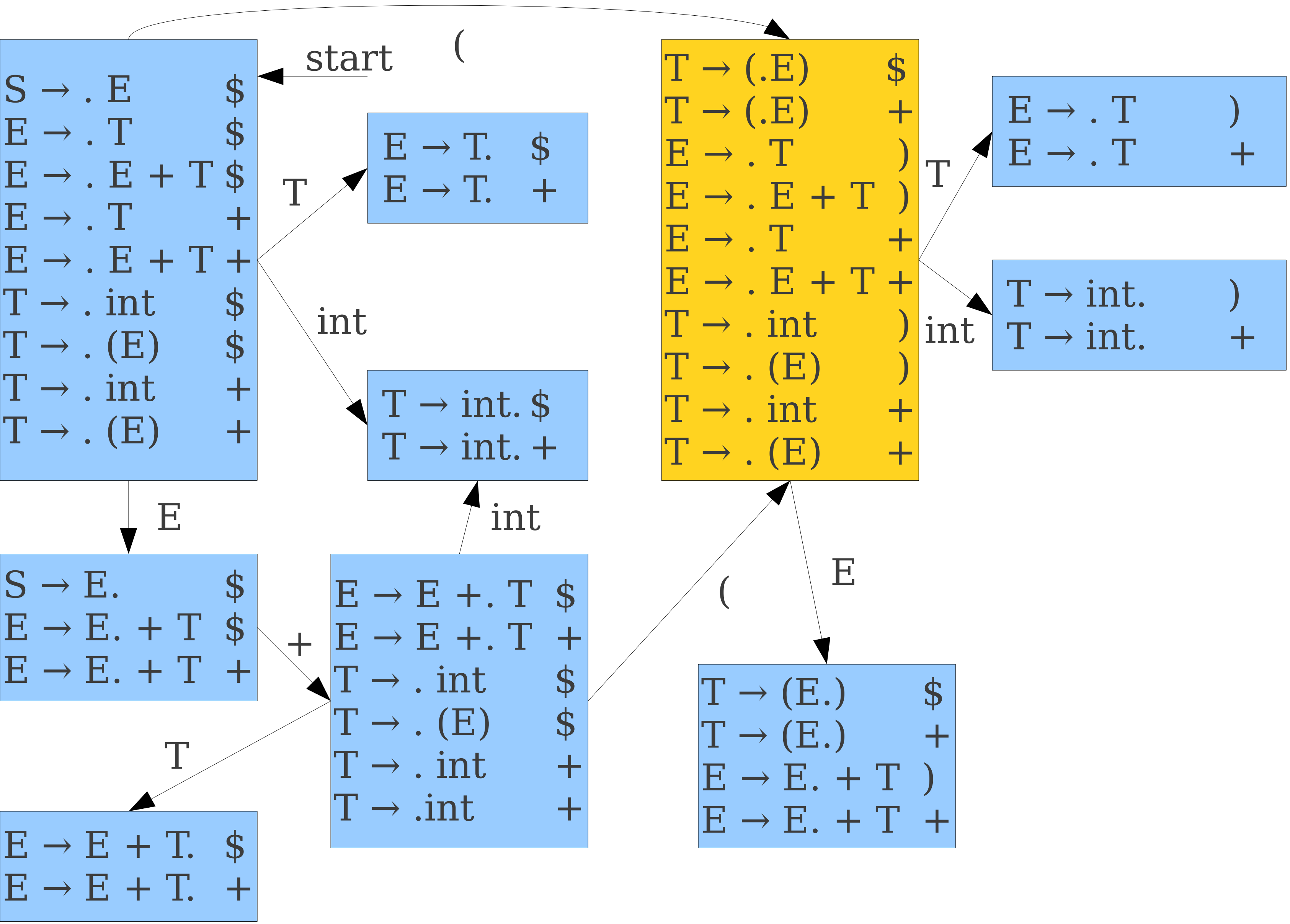
Deterministic LR(1) Automata



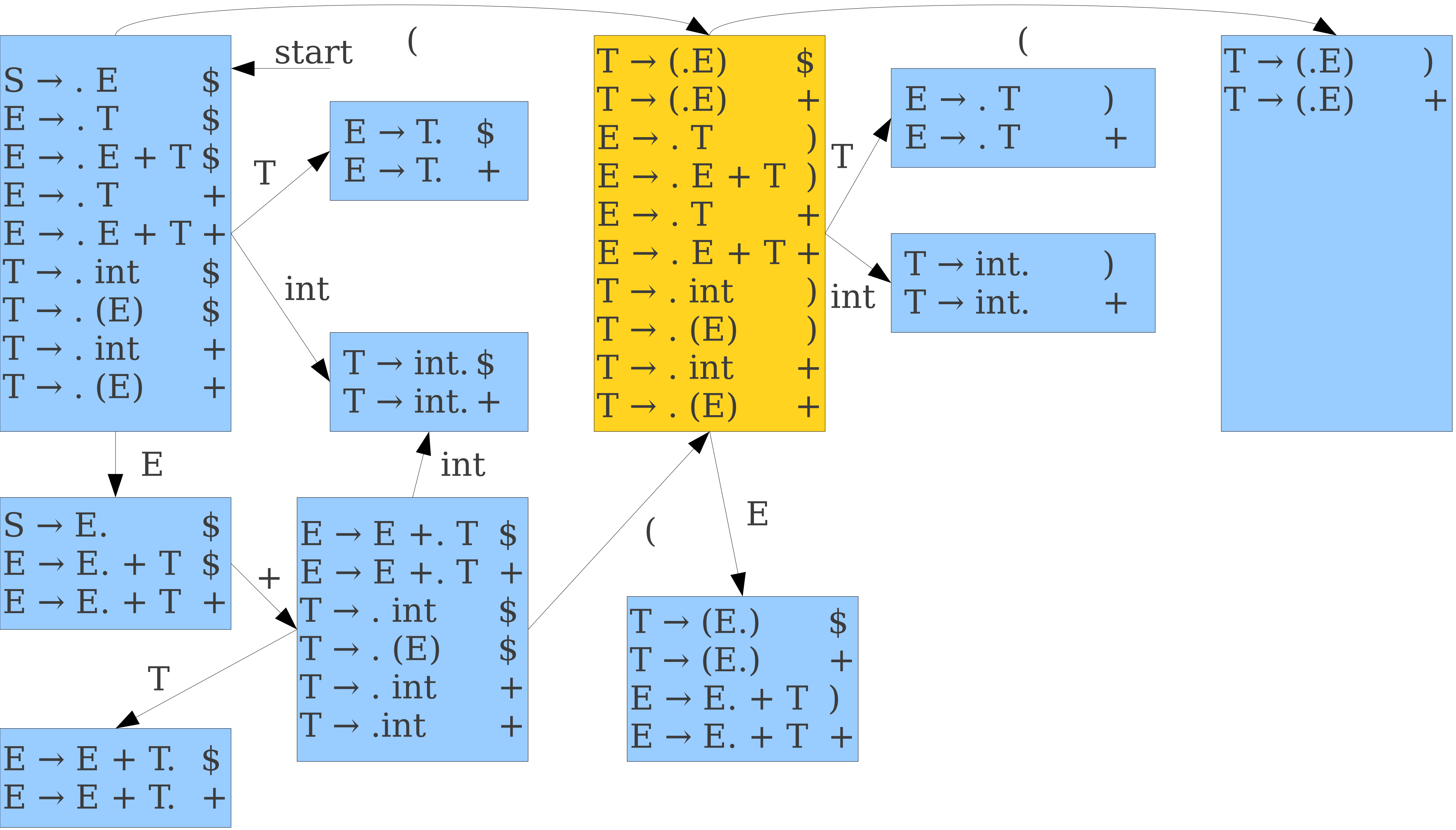
Deterministic LR(1) Automata



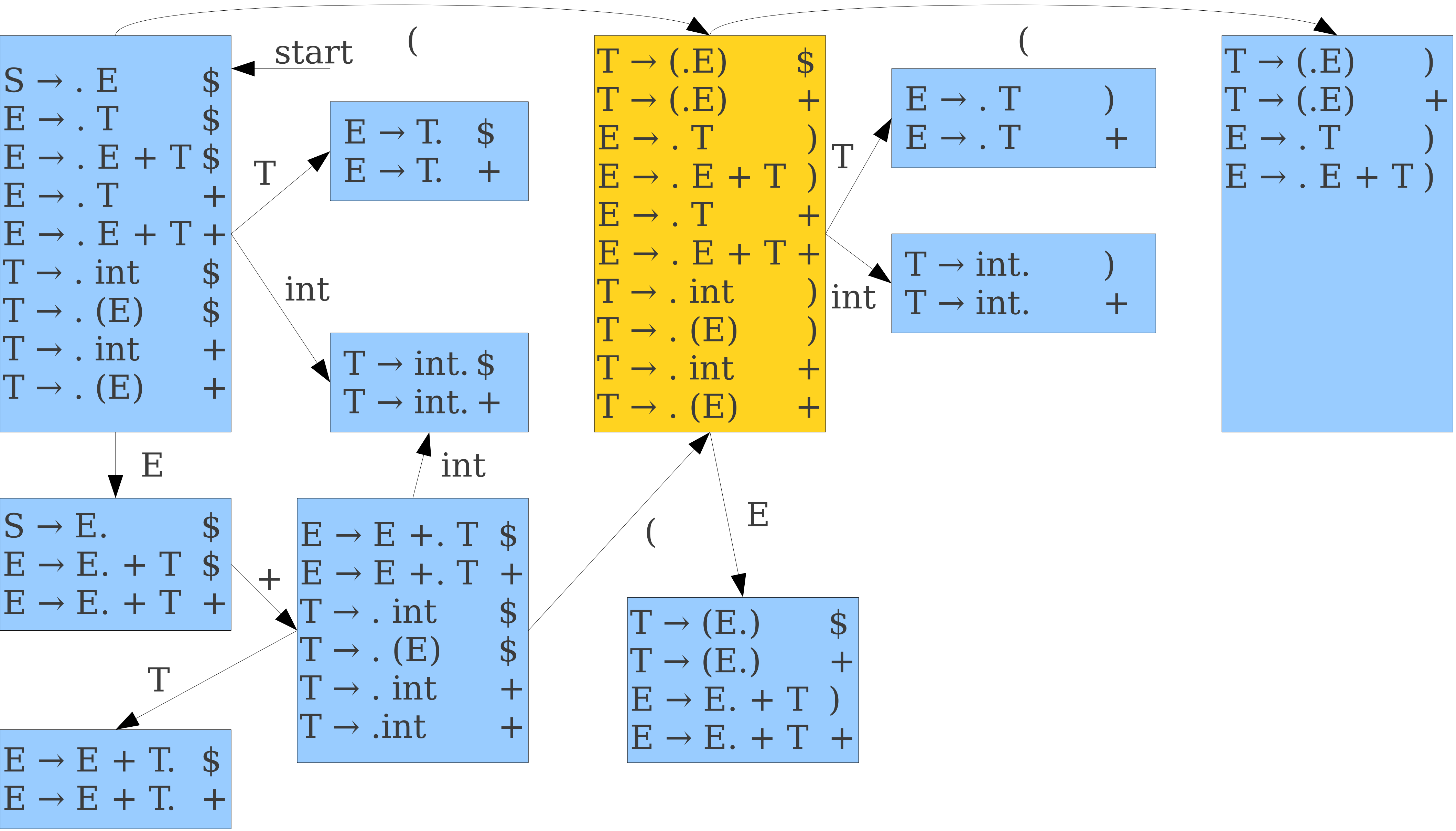
Deterministic LR(1) Automata



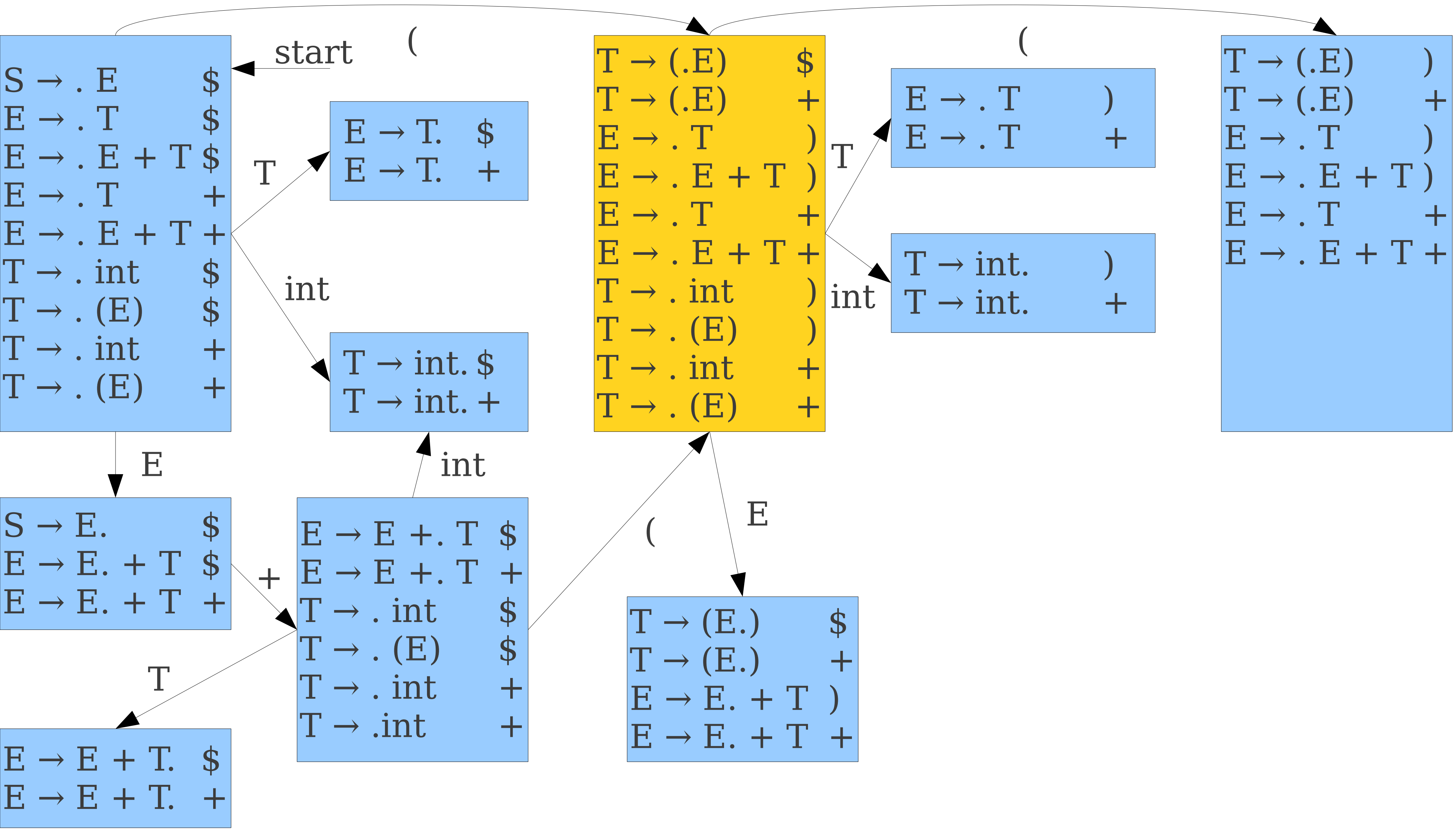
Deterministic LR(1) Automata



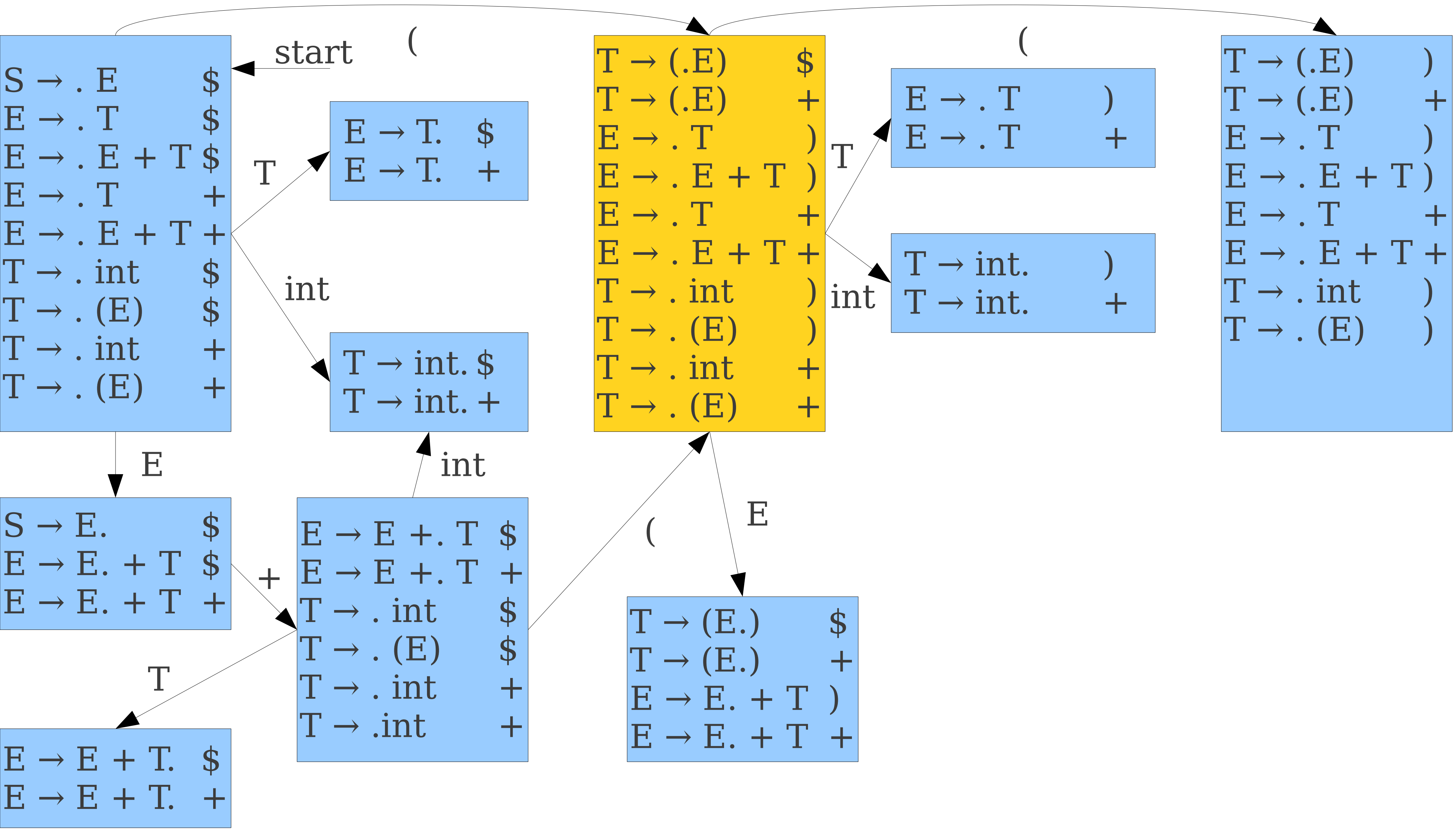
Deterministic LR(1) Automata



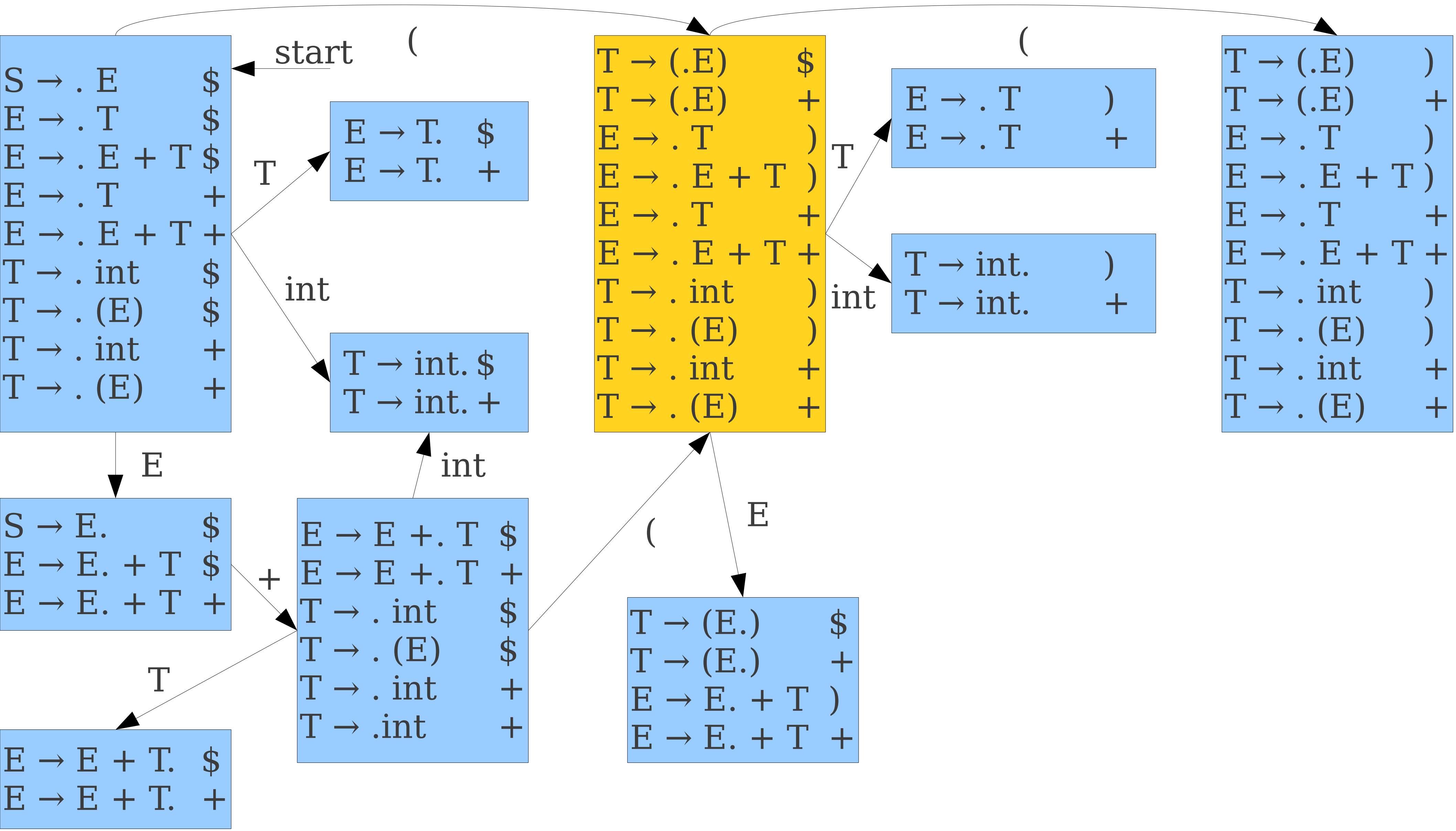
Deterministic LR(1) Automata



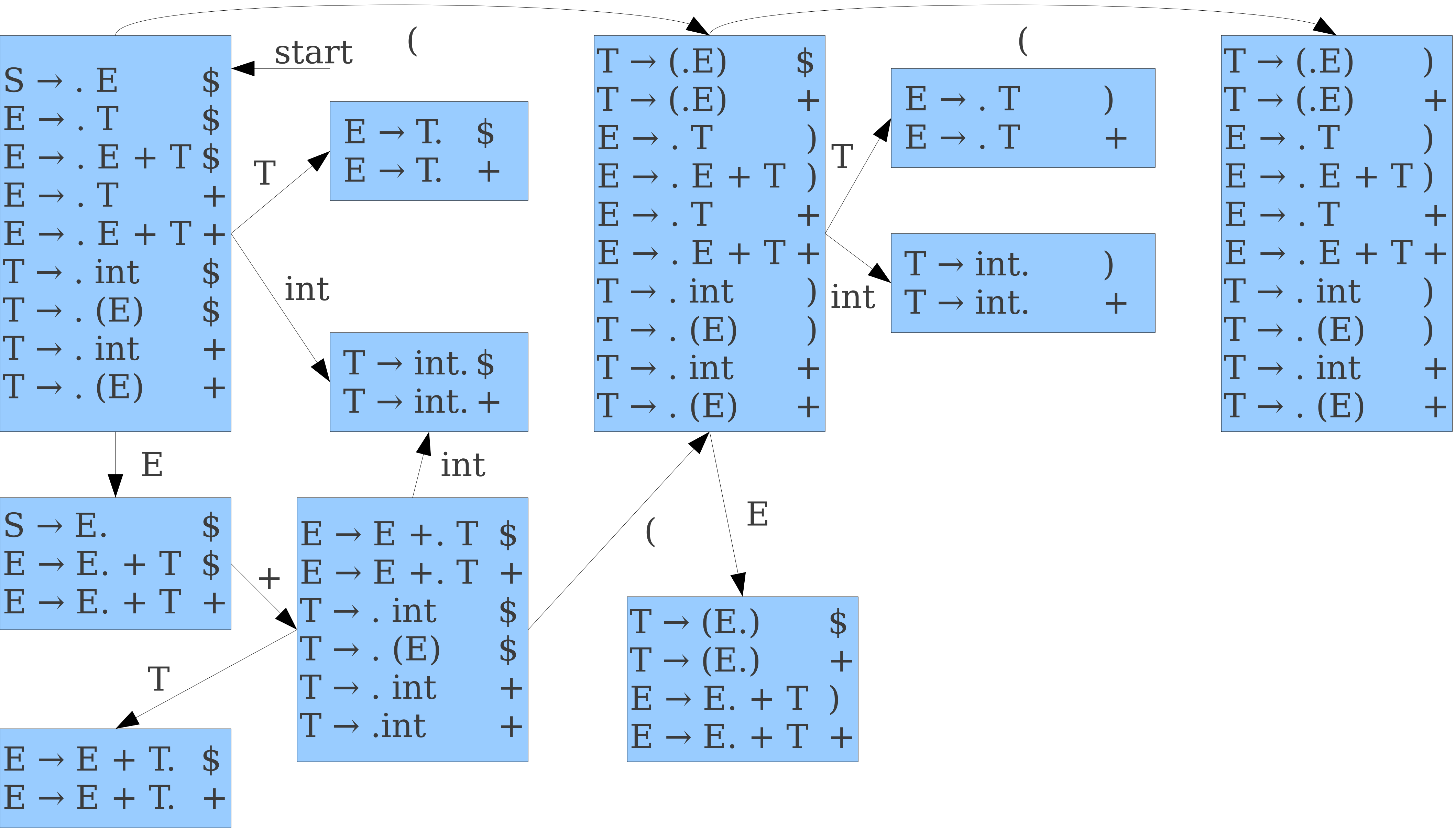
Deterministic LR(1) Automata



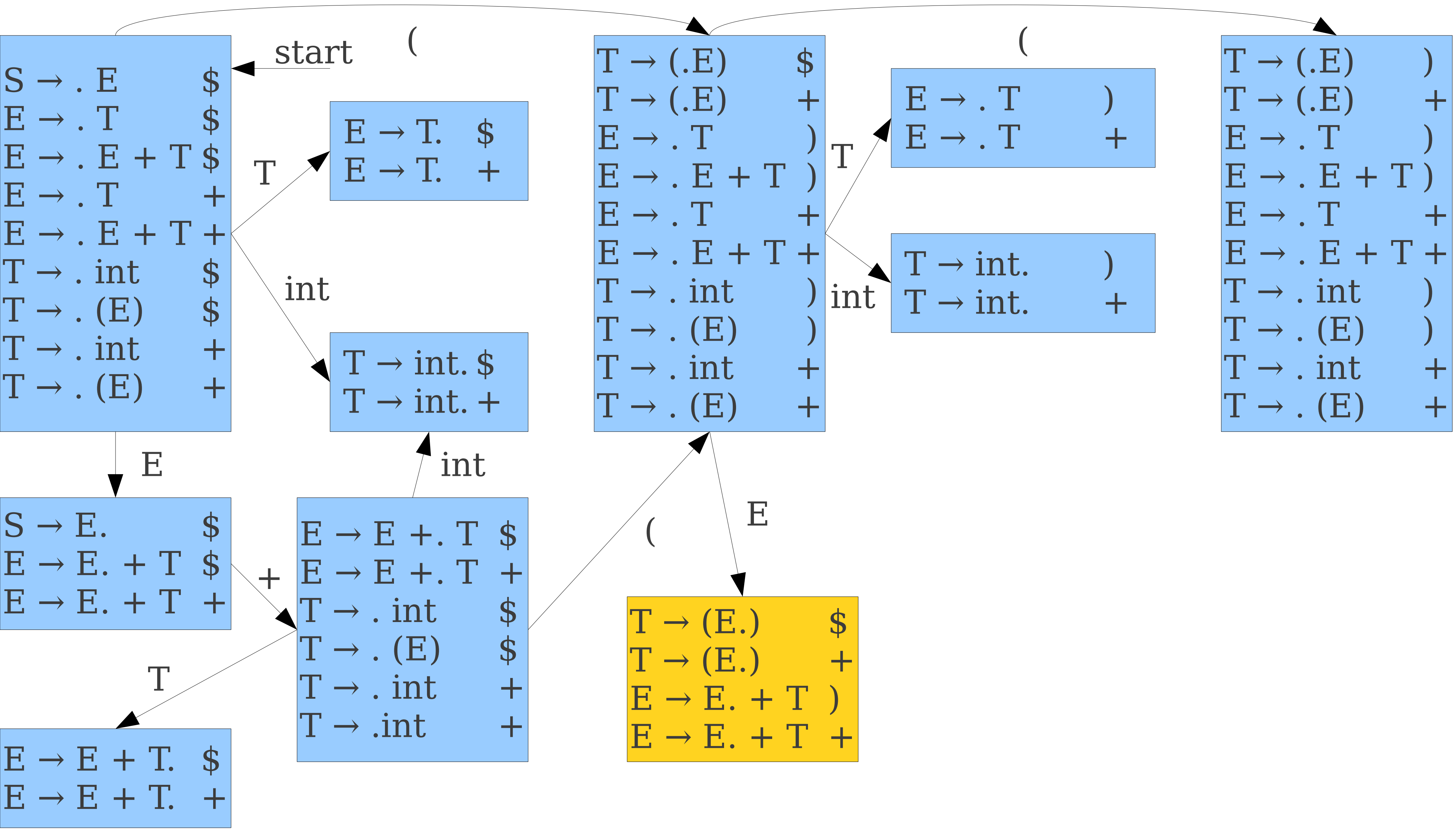
Deterministic LR(1) Automata



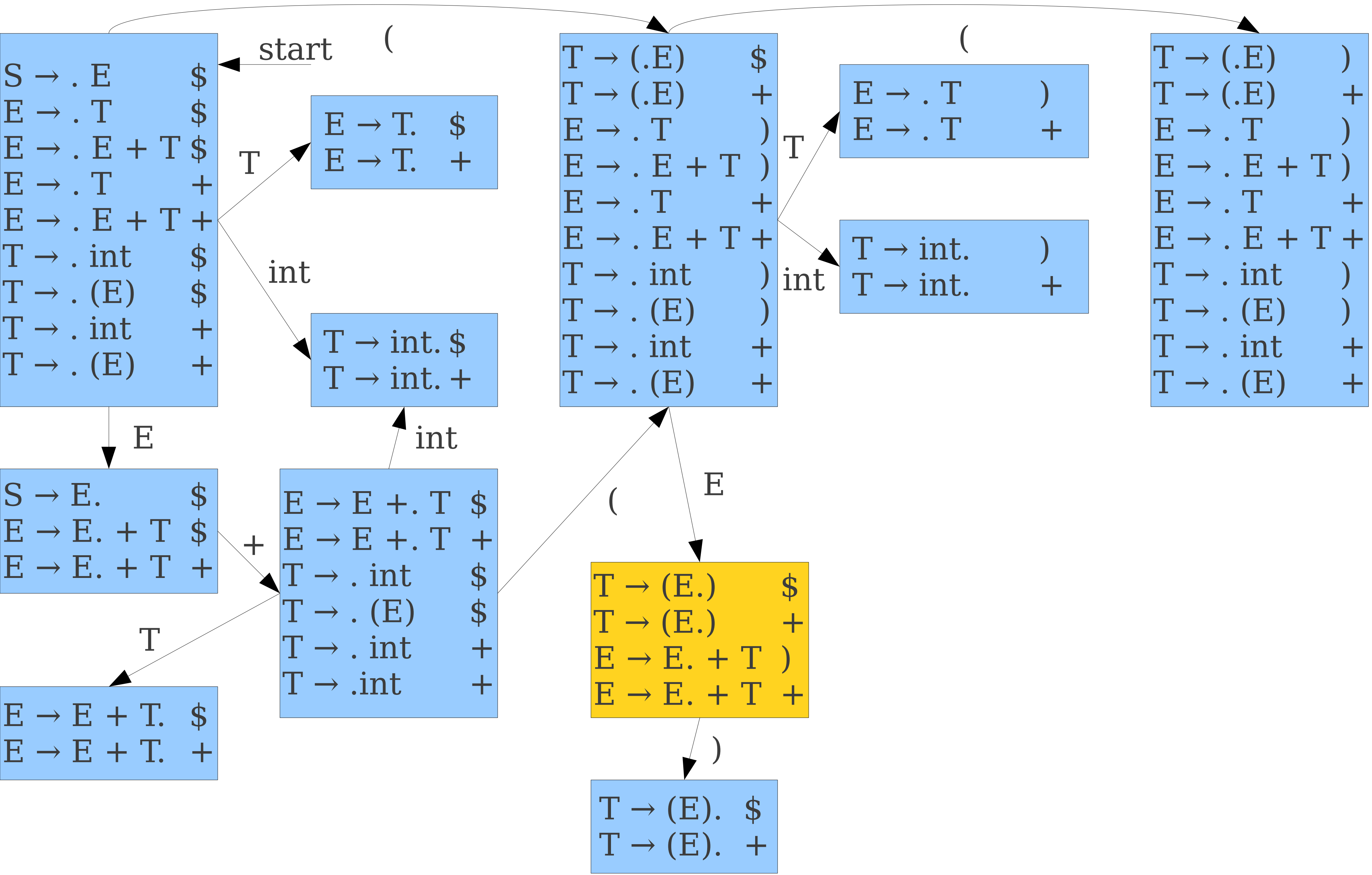
Deterministic LR(1) Automata



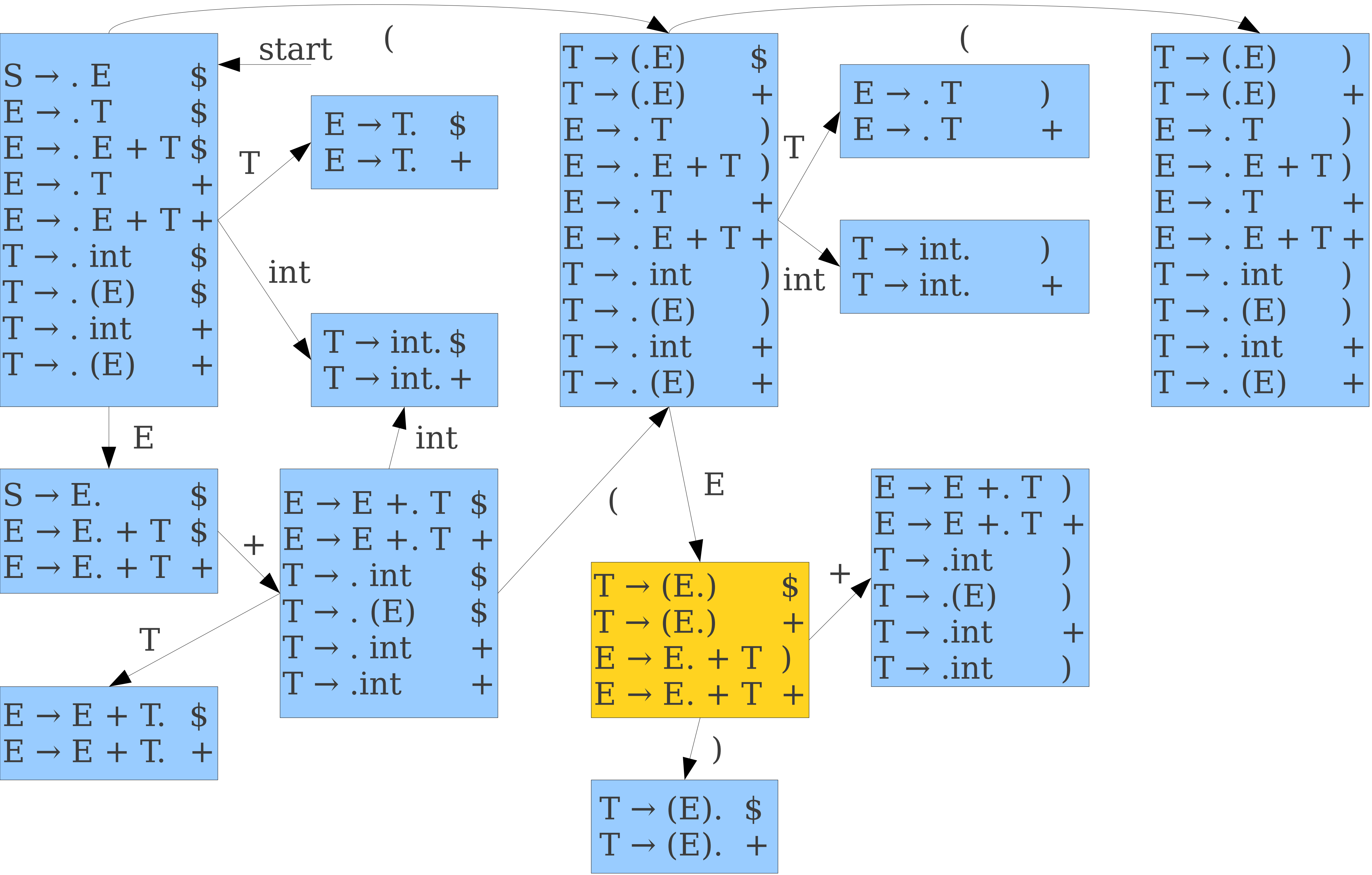
Deterministic LR(1) Automata



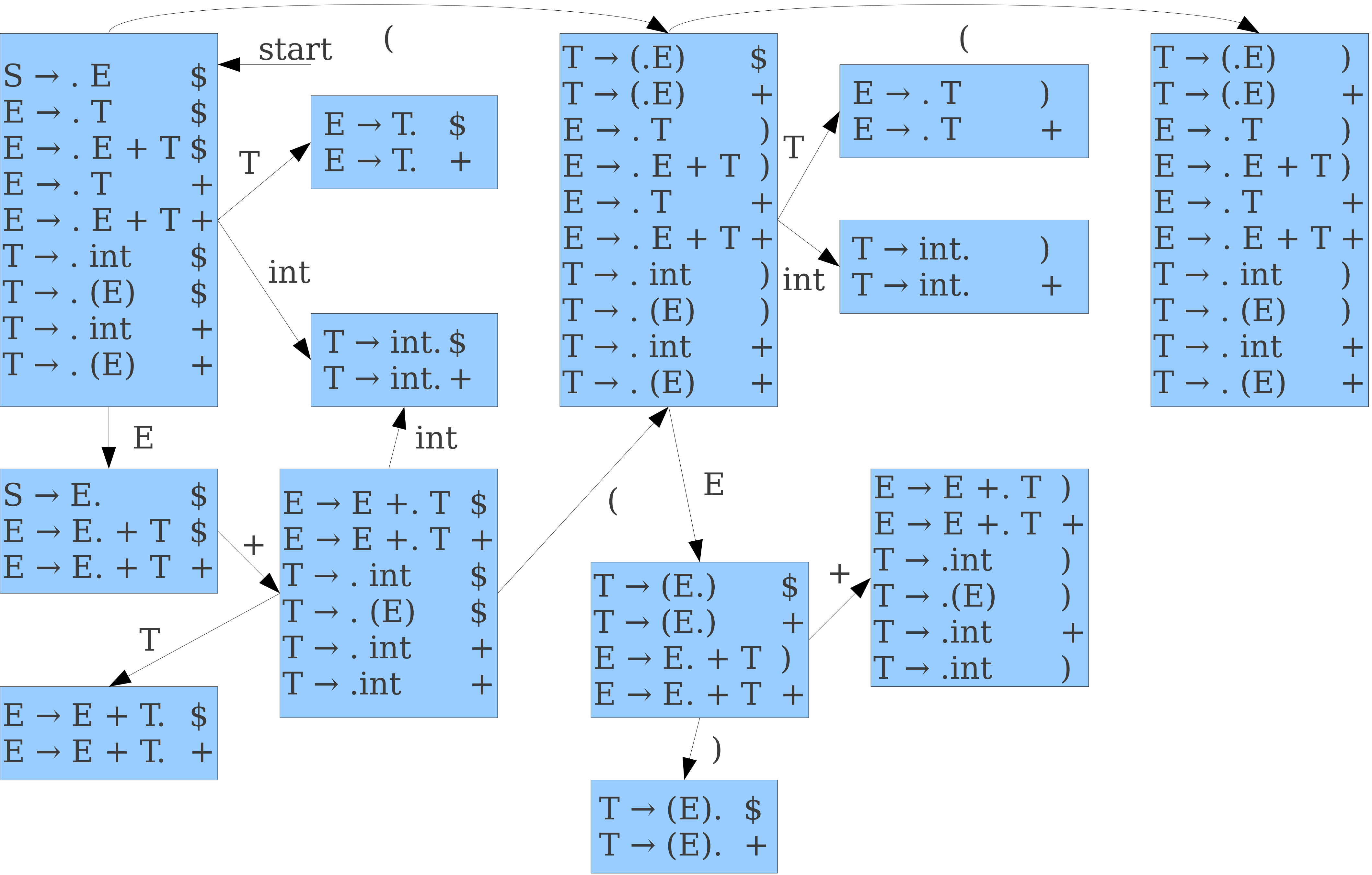
Deterministic LR(1) Automata



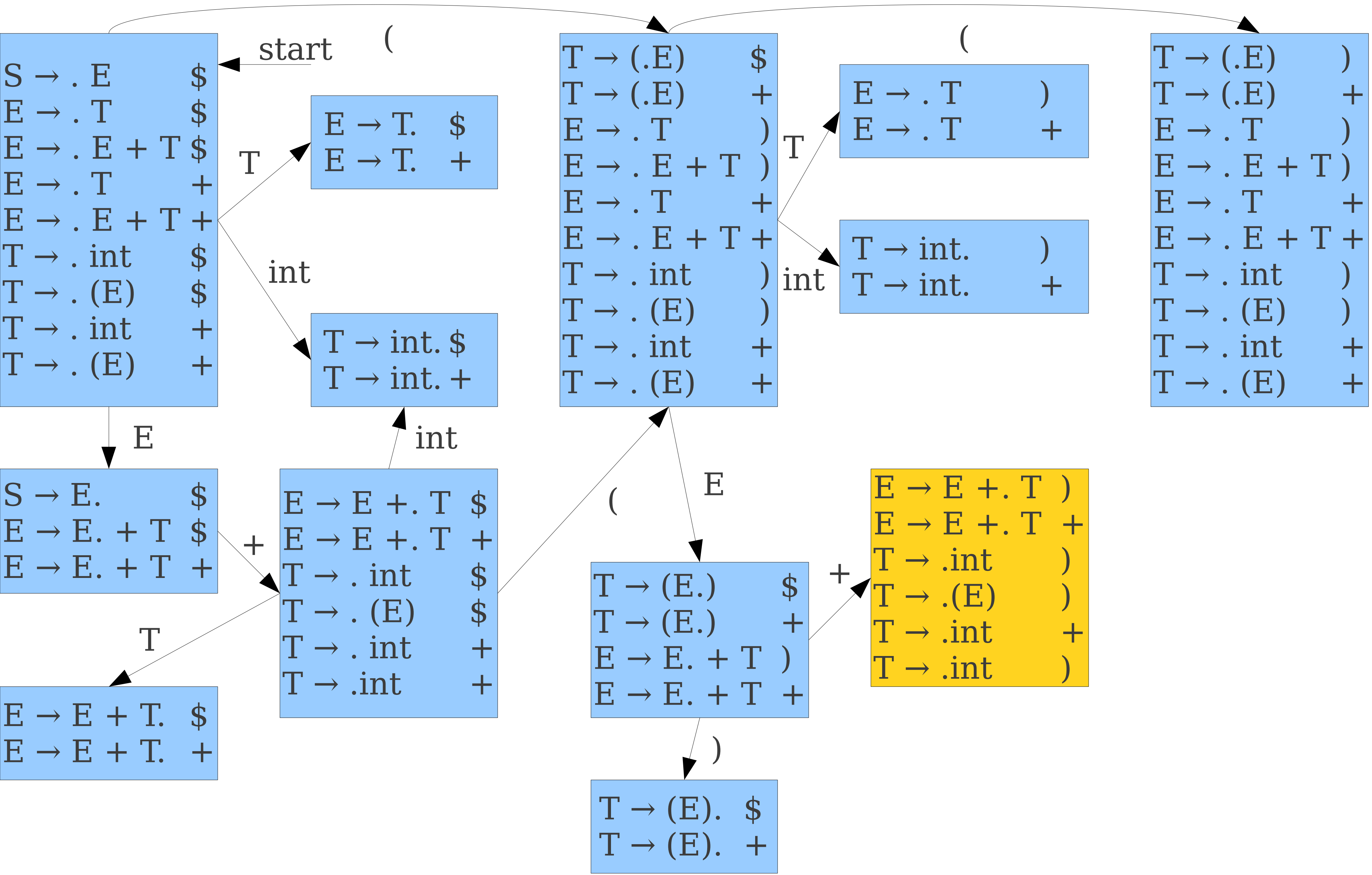
Deterministic LR(1) Automata



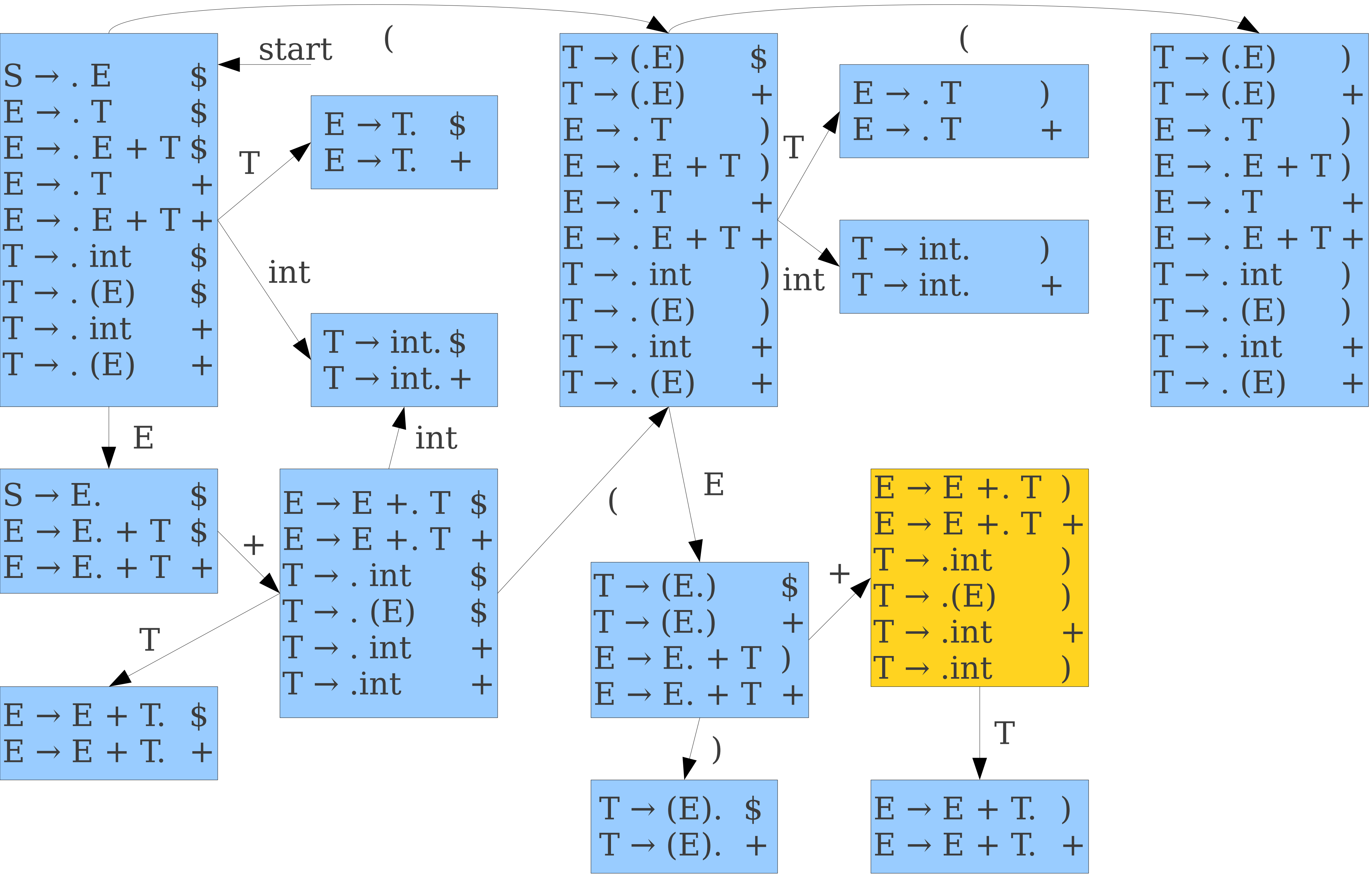
Deterministic LR(1) Automata



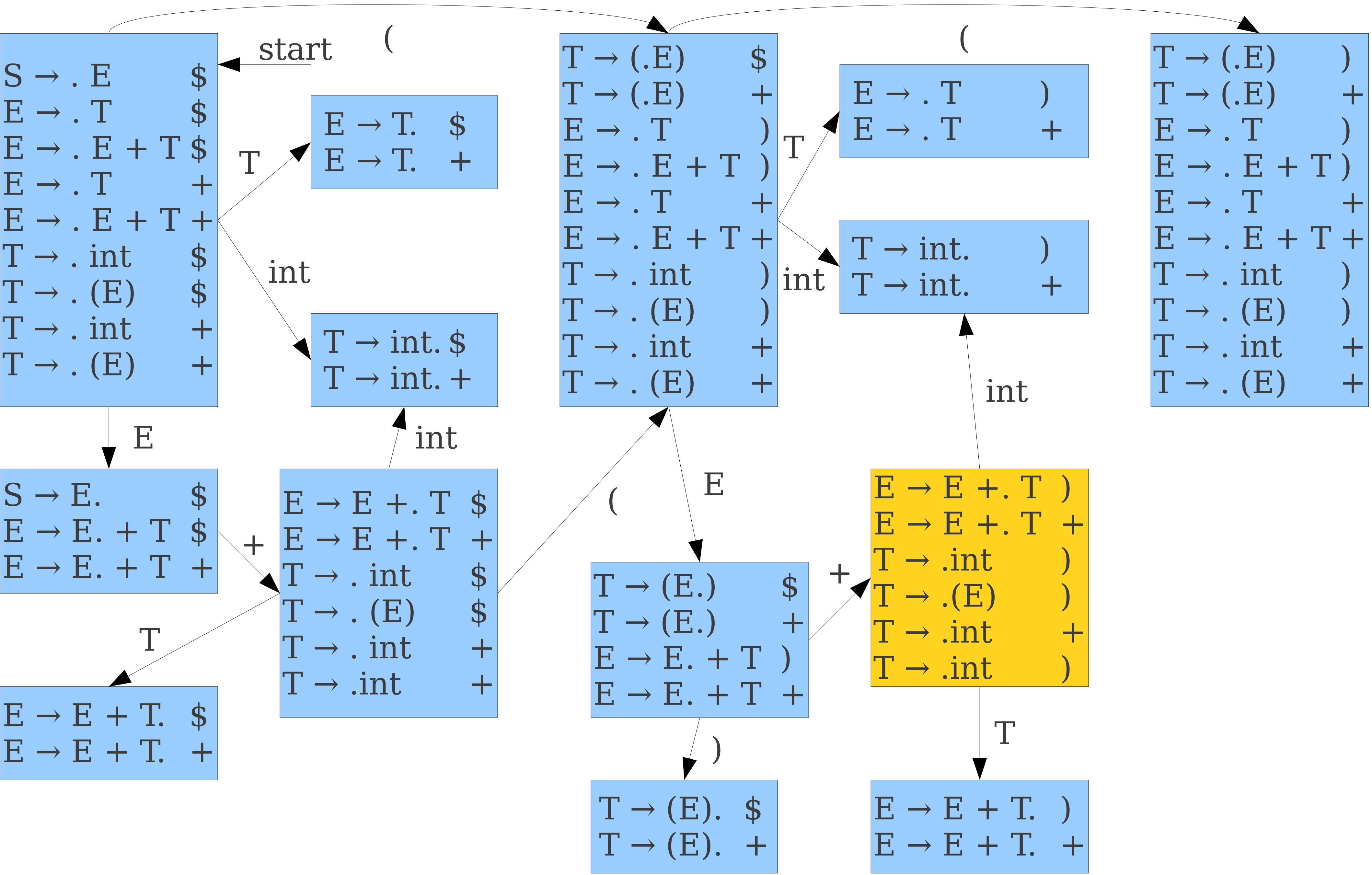
Deterministic LR(1) Automata



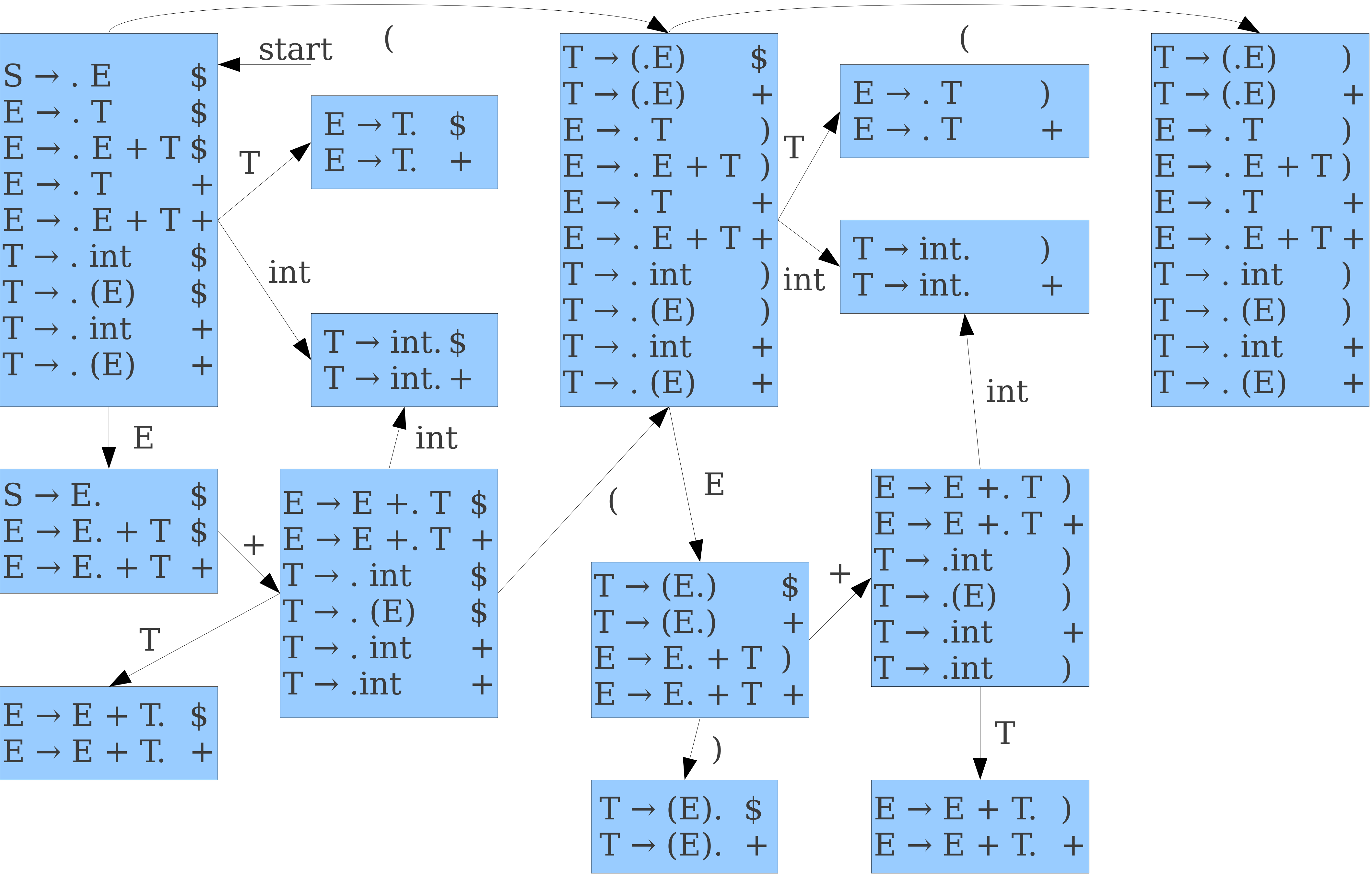
Deterministic LR(1) Automata



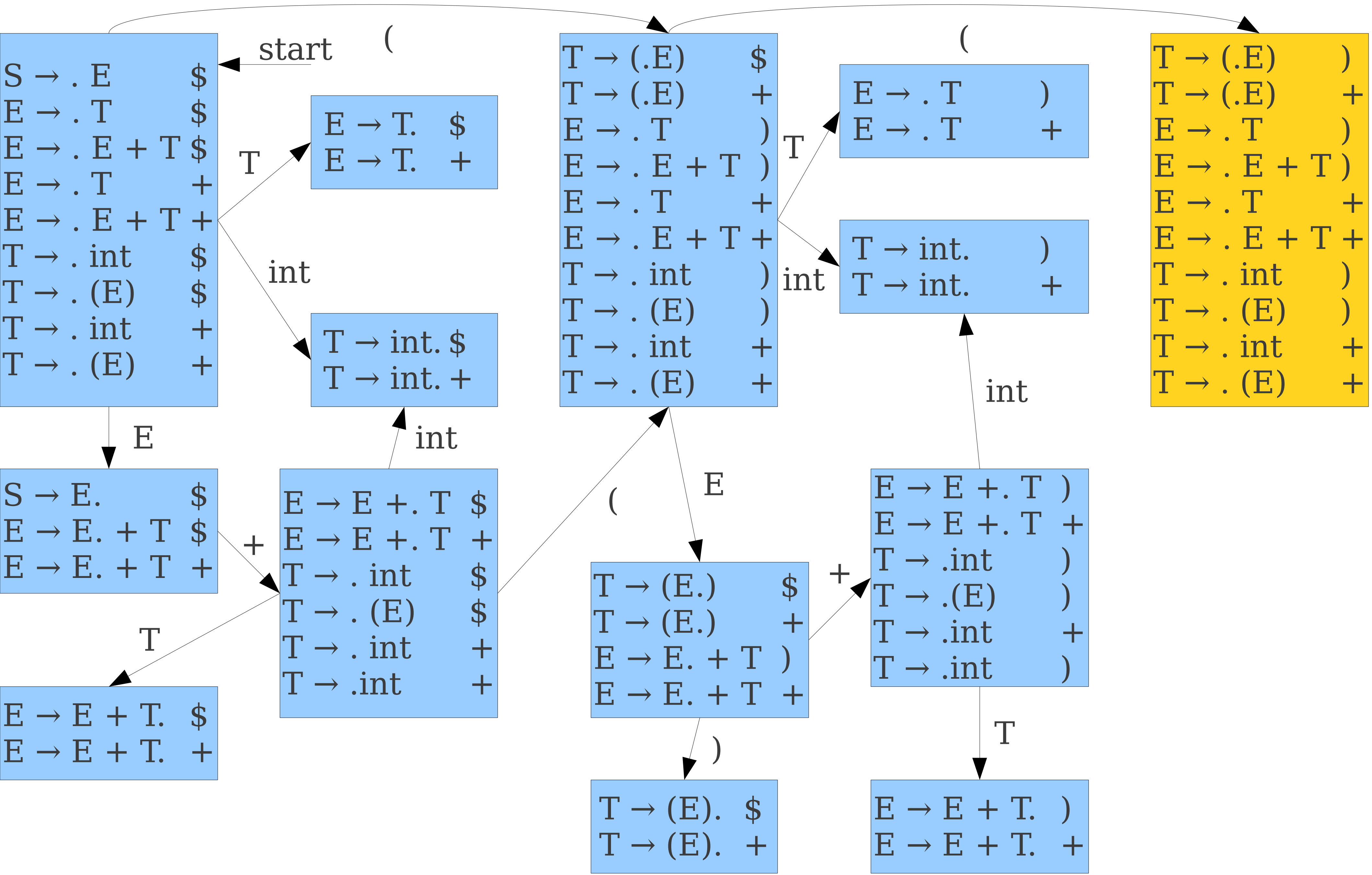
Deterministic LR(1) Automata



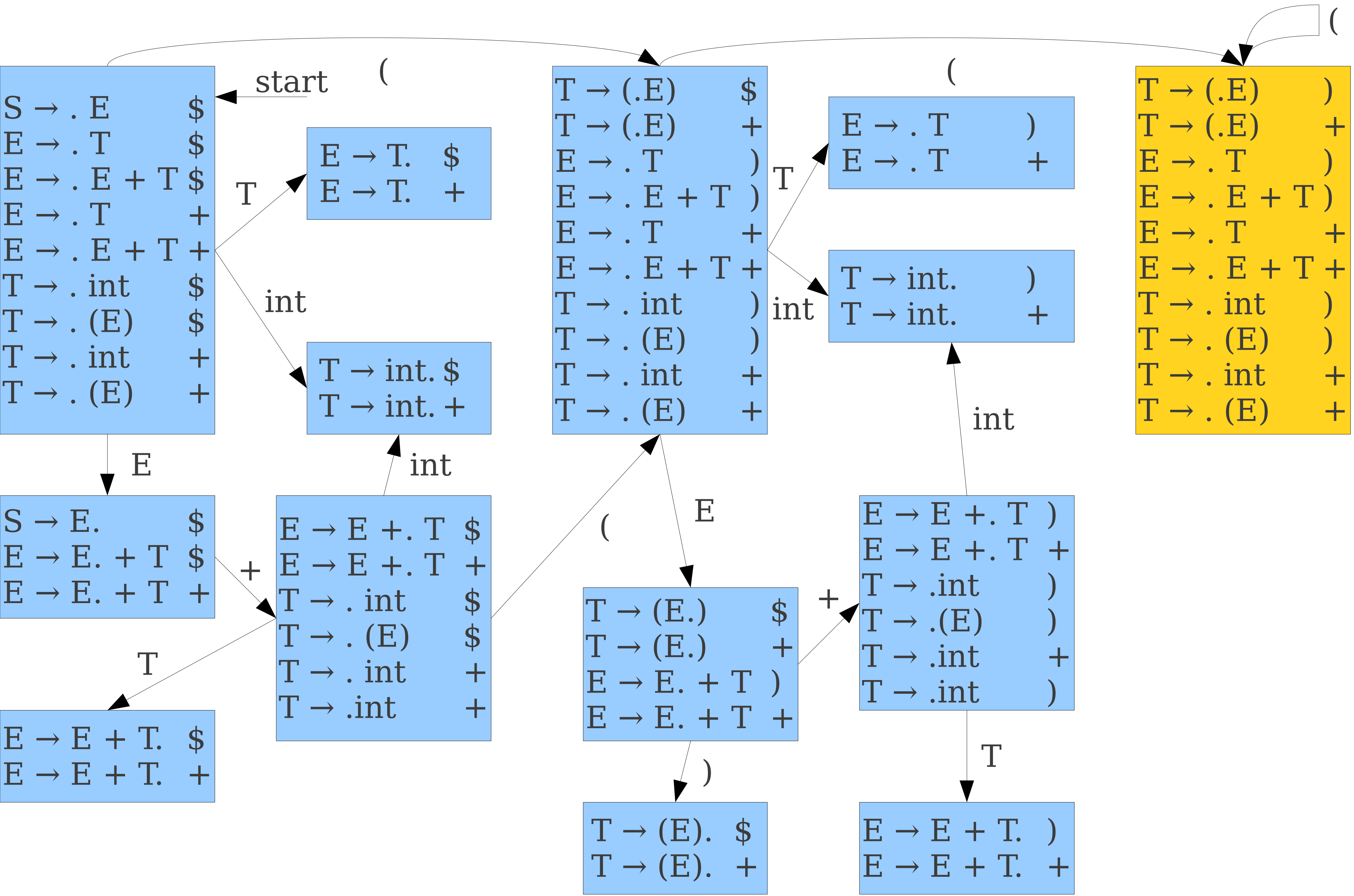
Deterministic LR(1) Automata



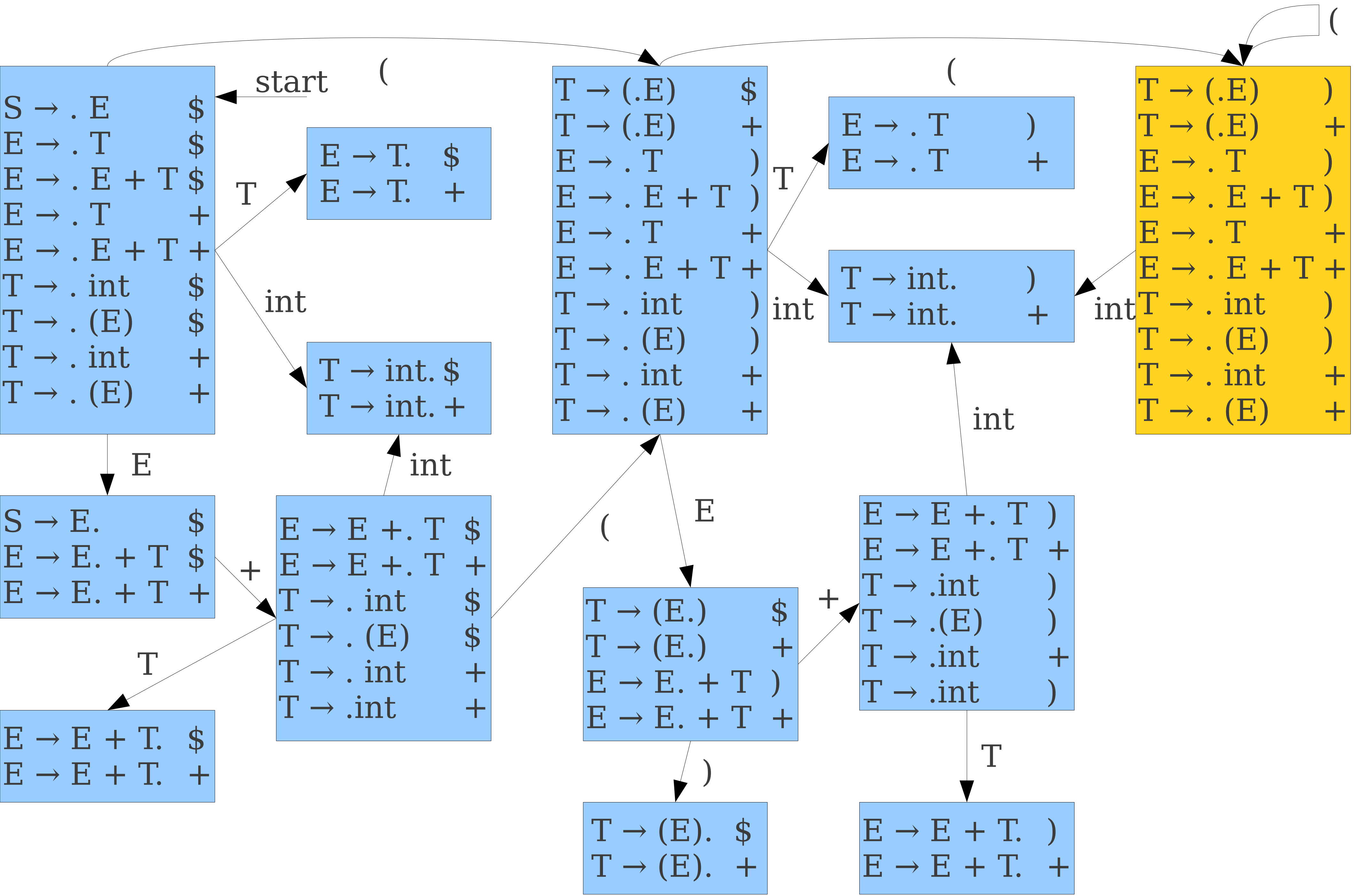
Deterministic LR(1) Automata



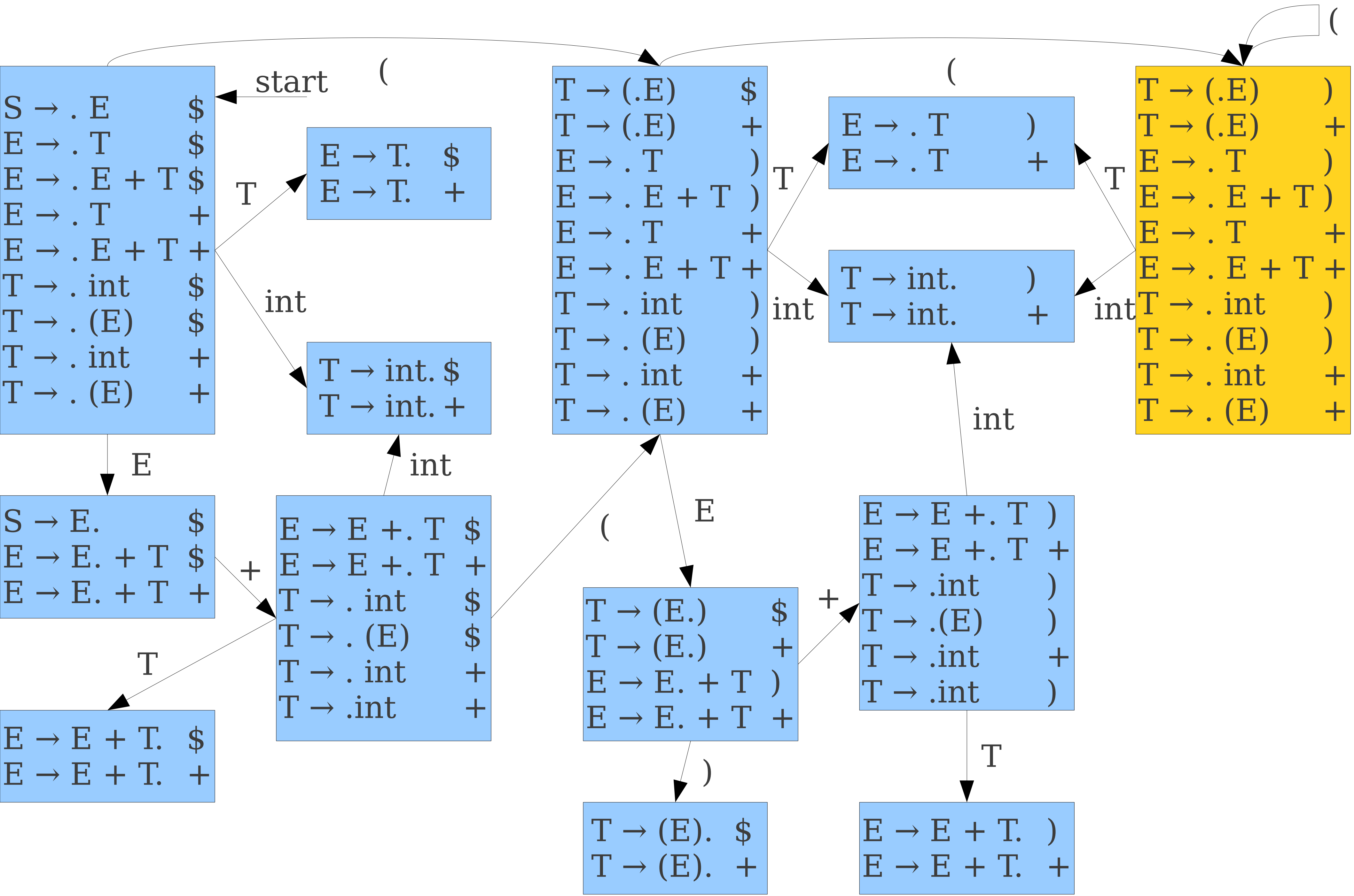
Deterministic LR(1) Automata



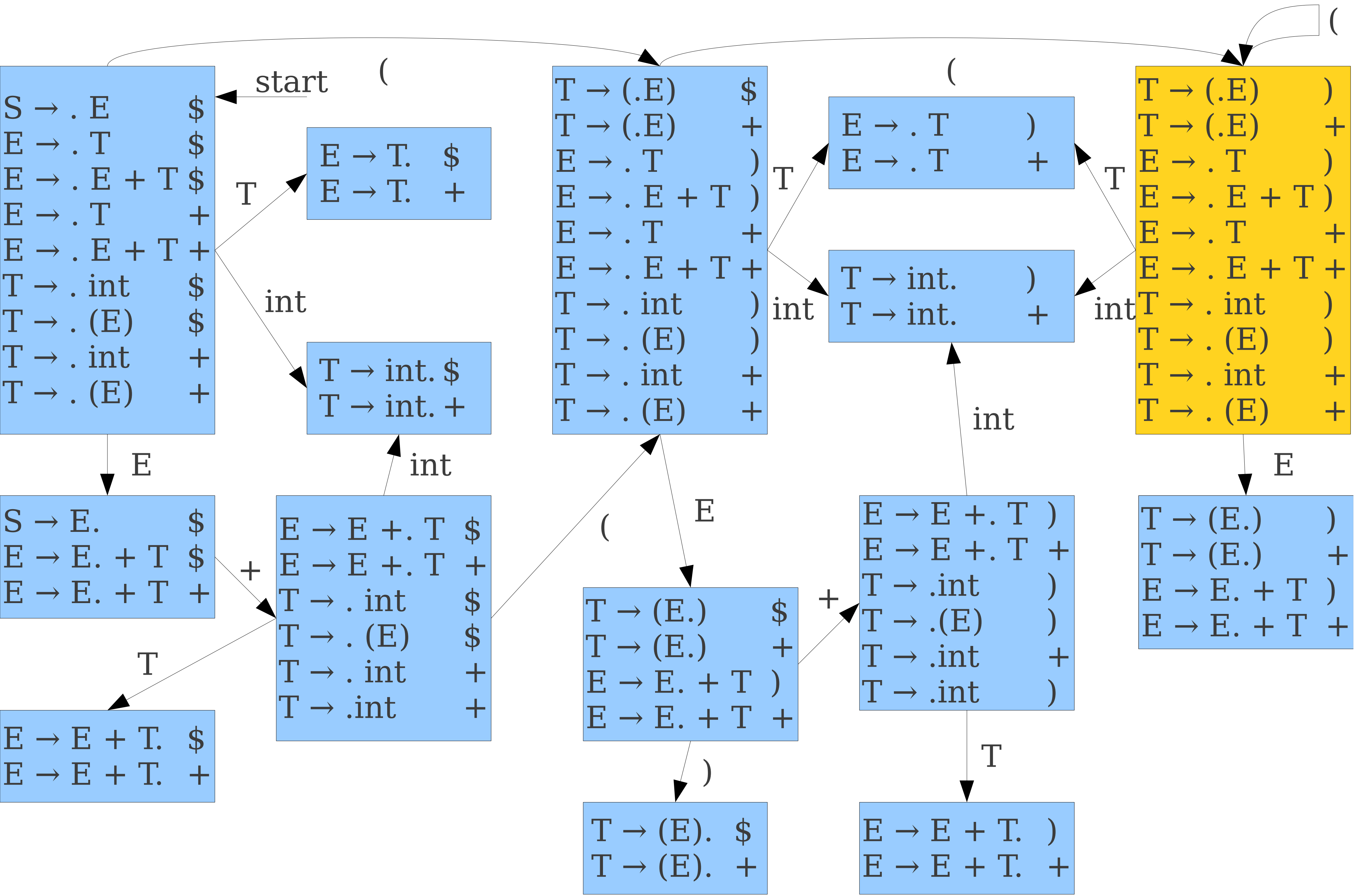
Deterministic LR(1) Automata



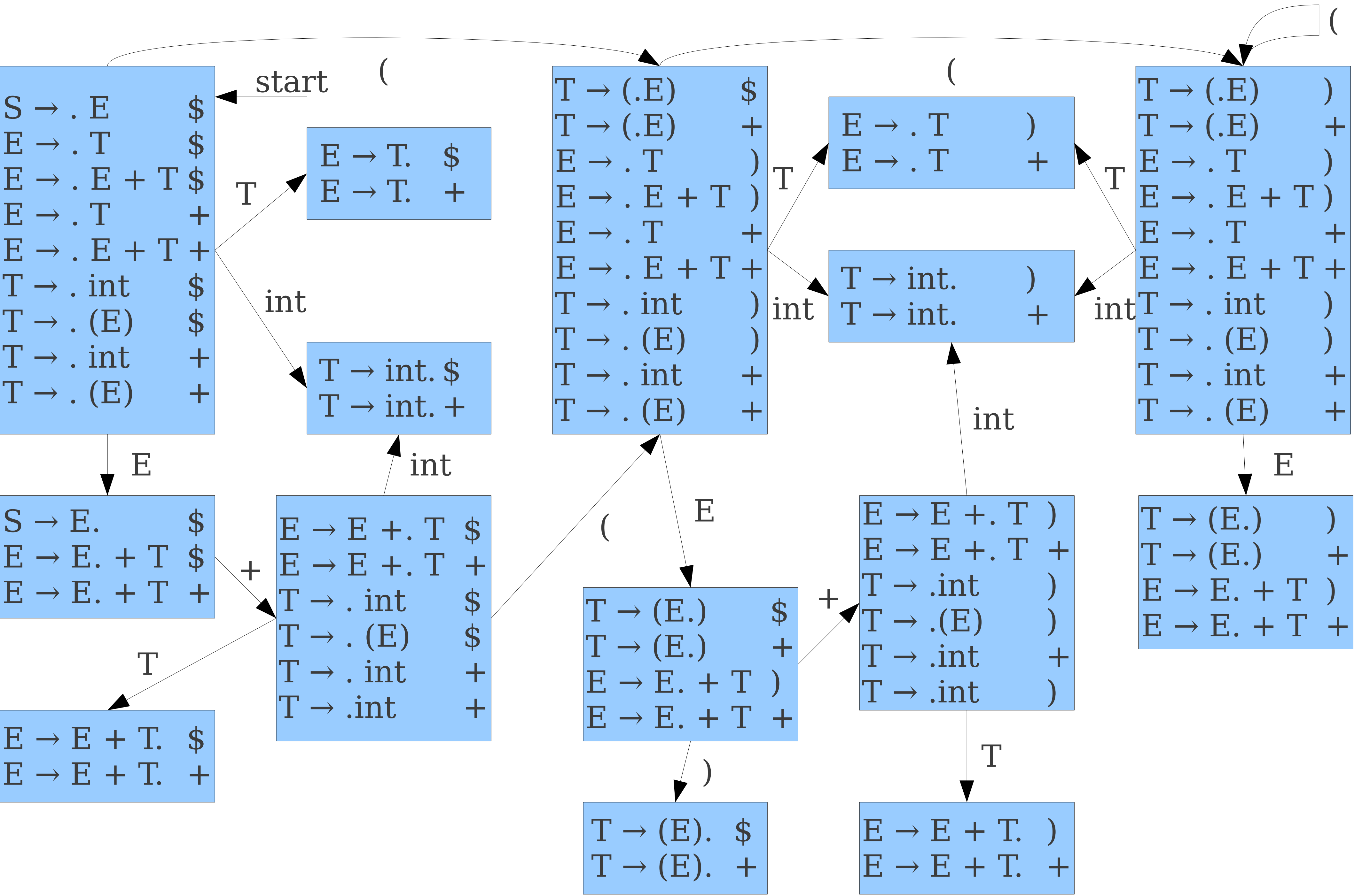
Deterministic LR(1) Automata



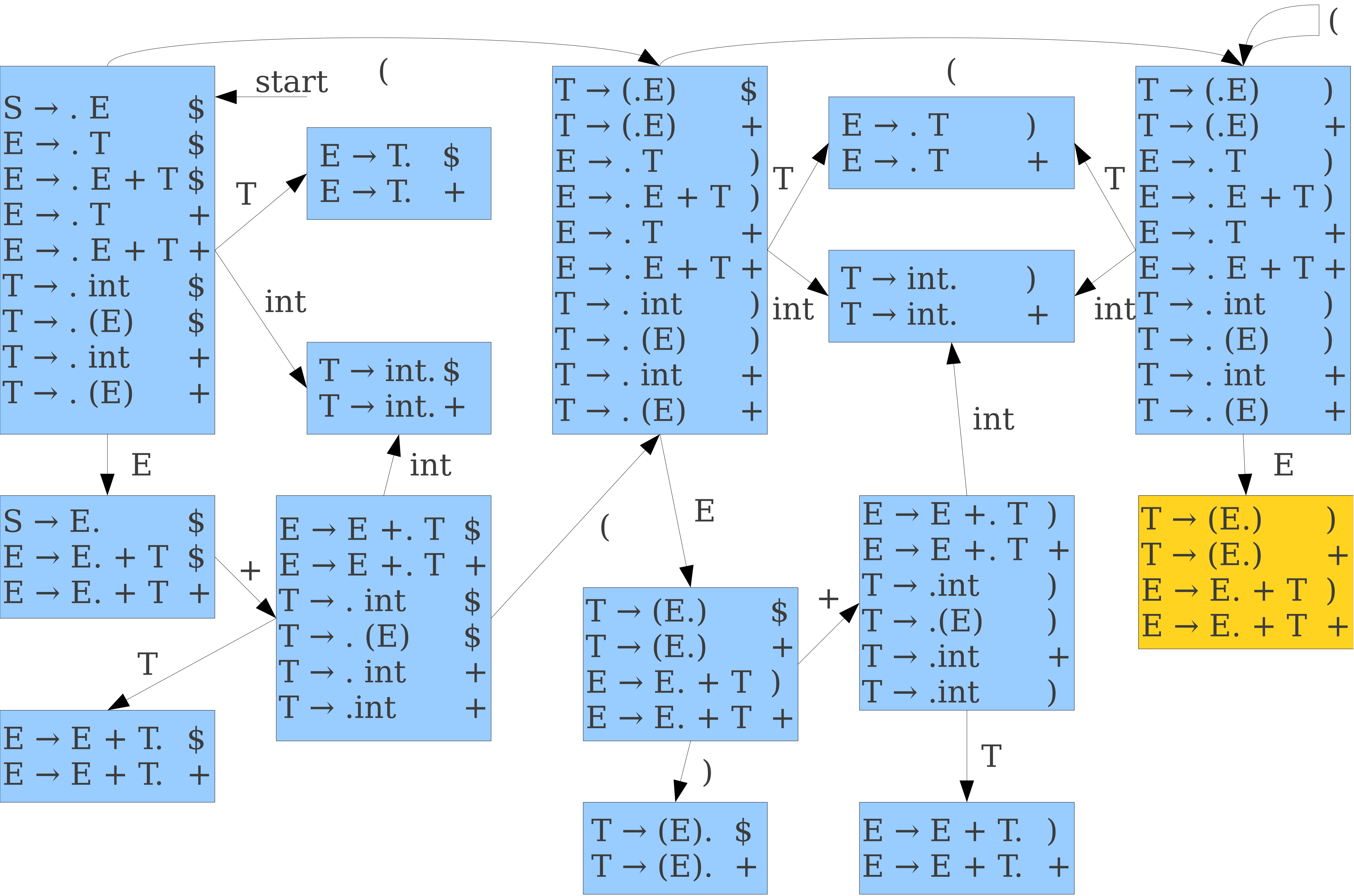
Deterministic LR(1) Automata



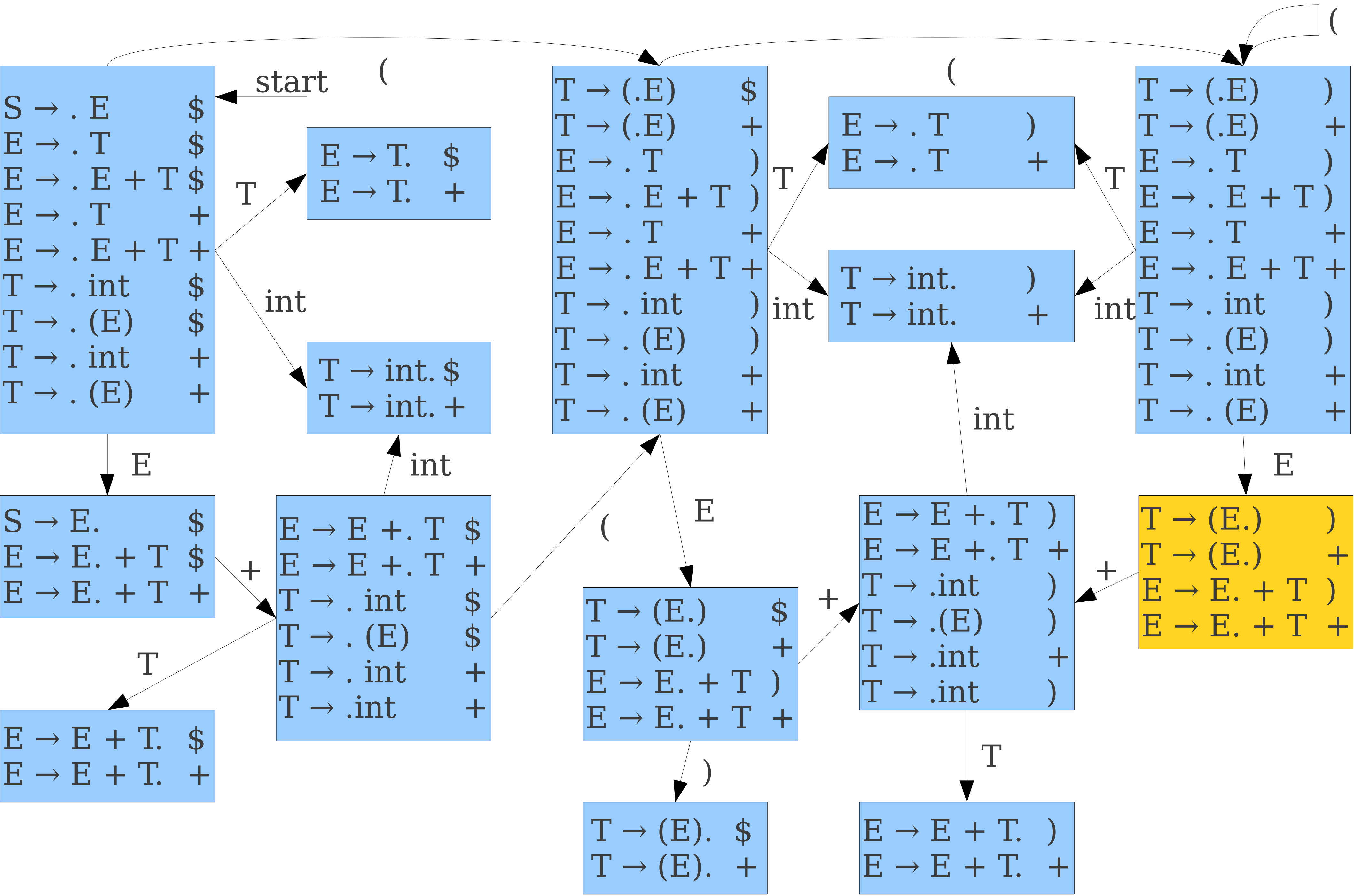
Deterministic LR(1) Automata



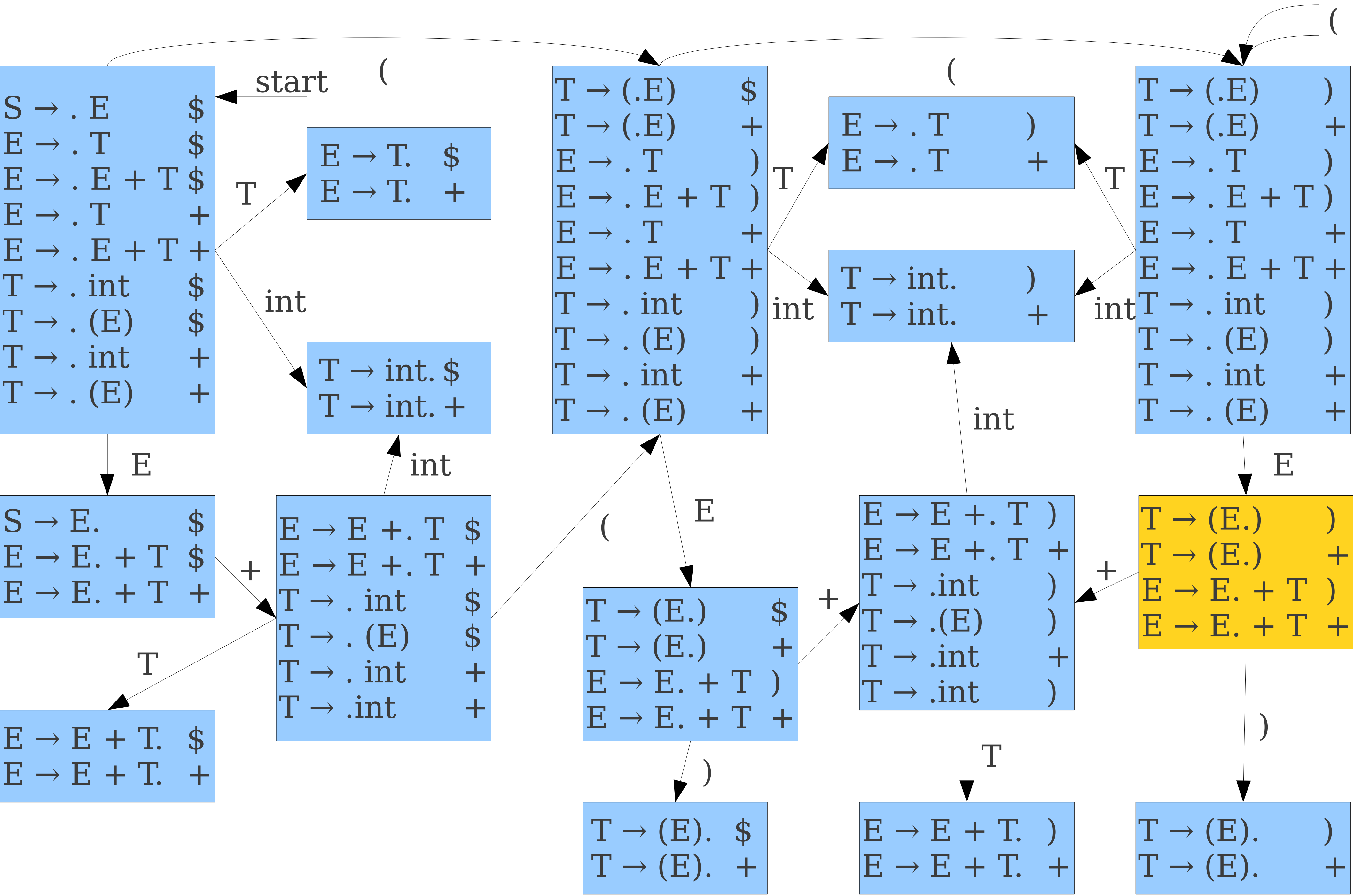
Deterministic LR(1) Automata



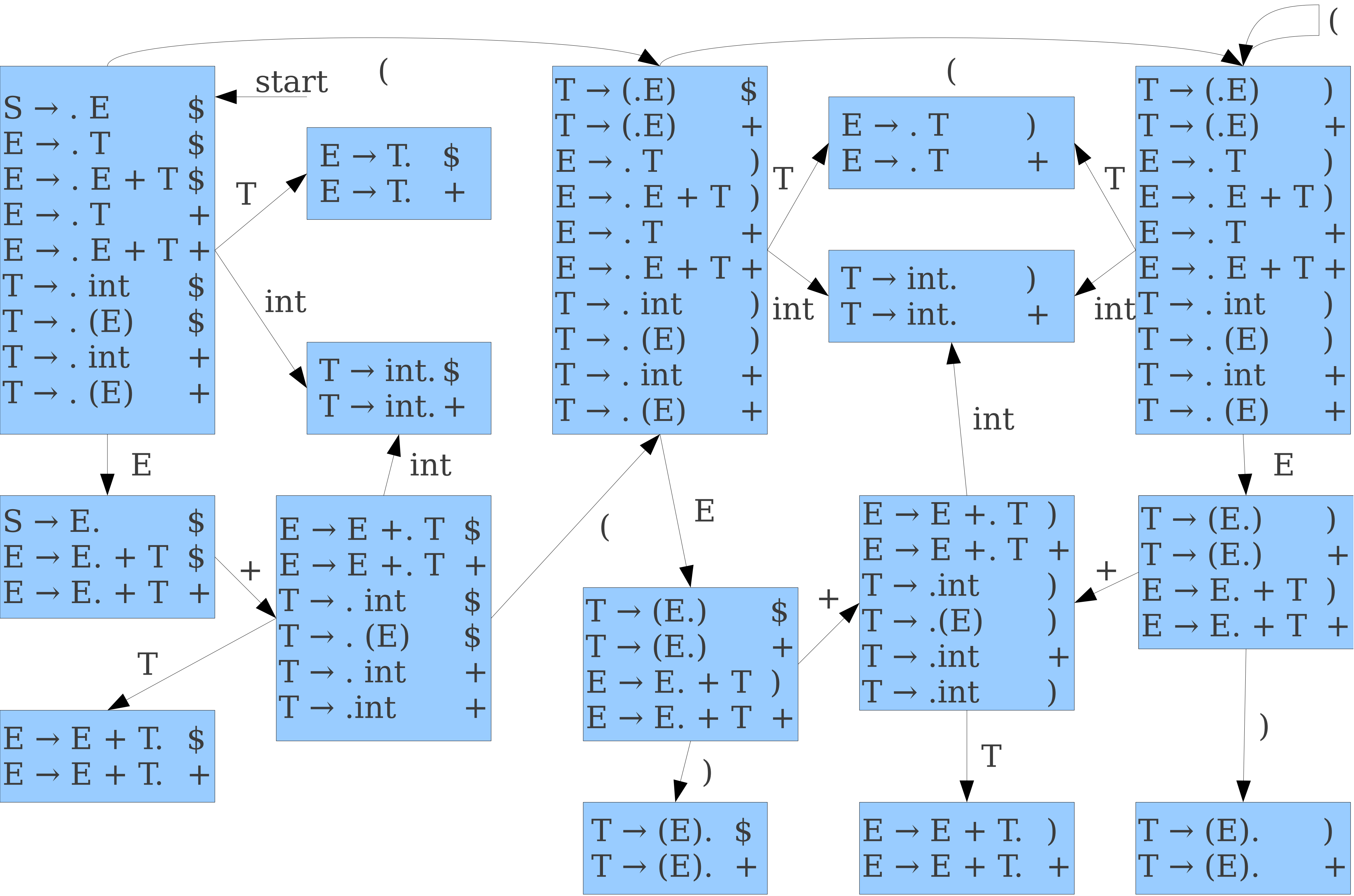
Deterministic LR(1) Automata



Deterministic LR(1) Automata



Deterministic LR(1) Automata



Constructing LR(1) Automata II

- Begin in a state containing $\mathbf{S} \rightarrow \cdot\ \mathbf{E}$ [`$`], where **S** is the start symbol.
- Compute the **closure** of the state:
  - If $\mathbf{A} \rightarrow \boldsymbol{\alpha} \cdot \mathbf{B}\boldsymbol{\omega}$ [`t`] is in the state, add $\mathbf{B} \rightarrow \cdot\ \boldsymbol{\gamma}$ [`t`] to the state for each production $\mathbf{B} \rightarrow \boldsymbol{\gamma}$ and for each terminal `t` ∈ FIRST*($\boldsymbol{\omega}$`t`)
- Repeat until no new states are added:
  - If a state contains a production $\mathbf{A} \rightarrow \boldsymbol{\alpha} \cdot \boldsymbol{x}\boldsymbol{\omega}$ [`t`], add a transition on $\boldsymbol{x}$ from that state to the state containing the closure of $\mathbf{A} \rightarrow \boldsymbol{\alpha}\boldsymbol{x} \cdot \boldsymbol{\omega}$ [`t`].

Structure of LR(1) Automata

- Every LR(1) automaton simulates two processes simultaneously:
  - An **LR(0) automaton** for finding handles.
  - A **lookahead tracker** for determining what the lookahead is.
- Removing the lookaheads from an LR(1) automaton results in a (much larger) LR(0) automaton for the same grammar.

Representing LR(1) Automata

- As with LR(0), use **action** and **goto** tables.
- **goto** table defined as before; encodes transition table as map from (state, token) to states.
- **action** table maps pairs (state, lookahead) to actions.
- Commonly combined into a single **action/goto** table.

$$\begin{aligned} S &\rightarrow E && (1) \\ E &\rightarrow T && (2) \\ E &\rightarrow E + T && (3) \\ T &\rightarrow \texttt{int} && (4) \\ T &\rightarrow (E) && (5) \end{aligned}$$

	int	(	)	+	$	T	E
1	s5					s4	s2
2				s6	ACCEPT		
3				r3	r3		
4				r2	r2		
5				r5	r5		
6	s5	s7				s3	
7	s10	s14				s10	s8
8			s9	s12			
9				r5	r5		
10			r2	r2			
11			r4	r4			
12	s11					s13	
13			r3	r3			
14	s11		s14			s10	s15
15			s16	s12			
16			r5	r5			

The LR(1) Parsing Algorithm

- Begin with an empty stack and the input set to ***ω*`$`**, where ***ω*** is the string to parse. Set **state** to the initial state.
- Repeat the following:
  - Let the next symbol of input be **`t`**.
  - If **action[state, `t`]** is **shift**, then shift the input and set **state** = **goto[state, `t`]**.
  - If **action[state, `t`]** is **reduce A** → ***ω***:
    - Pop |***ω***| symbols off the stack; replace them with **A**.
    - Let the state atop the stack be **top-state**.
    - Set **state** = **goto[top-state**, **A]**
  - If **action[state, `t`]** is **accept**, then the parse is done.
  - If **action[state, `t`]** is **error**, report an error.

Constructing LR(1) Parse Tables

- For each state X:
  - If there is a production $\mathbf{A} \to \boldsymbol{\omega} \cdot$ [`t`], set **action[**X**,** `t`**]** = reduce $\mathbf{A} \to \boldsymbol{\omega}$.
  - If there is the special production $\mathbf{S} \to \mathbf{E} \cdot$ [`$`], where $\mathbf{S}$ is the start symbol, set **action[**X**,** `t`**]** = **accept**.
  - If there is a transition out of s on symbol `t`, set **action[**X**,** `t`**] = shift**.
- Set all other actions to **error**.
- If any table entry contains two or more actions, the grammar is not LR(1).

Next Time

- **SLR(1) Parsing**
  - A smaller, simpler, and weaker variant of LR(1).
- **LALR(1) Parsing**
  - An excellent tradeoff between SLR(1) and LR(1).
- **Parsing Ambiguous Grammars**
  - Manually tweaking LR parsers.
- **Error Recovery**
  - Report all the errors!