Compilers and Interpreters

Bottom-Up Parsing

What is Bottom-Up Parsing?

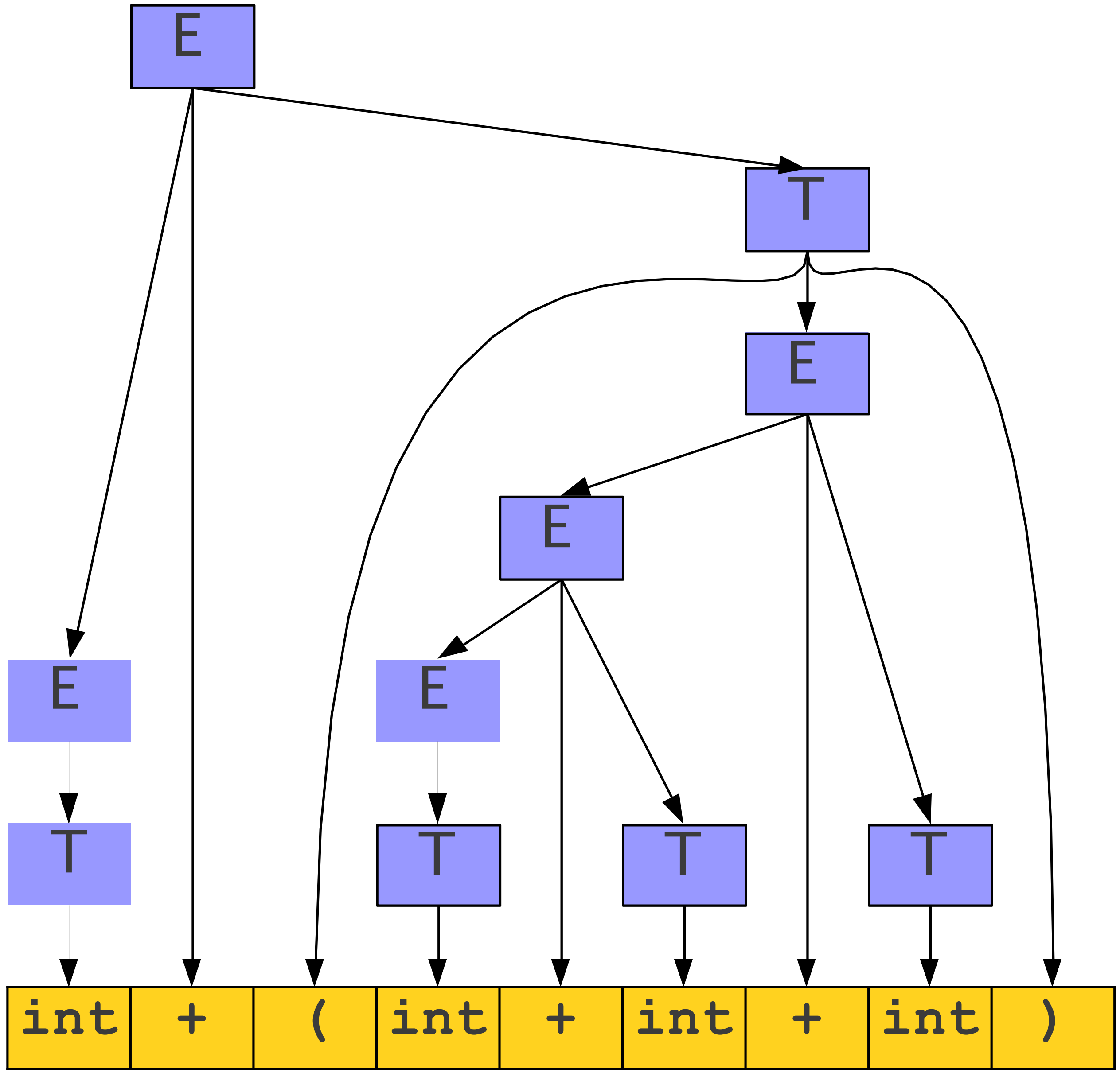
- Idea: Apply productions **in reverse** to convert the user's program to the start symbol.
  - We can think of bottom-up parsing as the process of "reducing" a string *w* to the start symbol of the grammar. At each reduction step, a specific substring matching the body of a production is replaced by the nonterminal at the head of that production.
- Keywords
  - Reductions, handle, shift-reduce parsing, conflicts, LR grammars

What is Bottom-Up Parsing?

- We'll be exploring four **directional**, **predictive** bottom-up parsing techniques:
  - **Directional**: Scan the input from left-to-right.
  - **Predictive**: Guess which production should be inverted.
  - The largest class of grammars for which shift-reduce parsers can be built, the LR grammars: **LR(0), SLR(0), LR(1), LALR(1)**

One View of a Bottom-Up Parse

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



A Second View of a Bottom-Up Parse

E → T
E → E + T
T → int
T → (E)

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

int + (int + int + int)

+ int + int)

+ int + int)

+ T + int)

⇒ E + (E + int)

⇒ E + (E + T)

⇒ E + (E)

⇒ E + T

⇒ E

A Second View of a Bottom-Up Parse

E → T
E → E + T
T → int
T → (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.

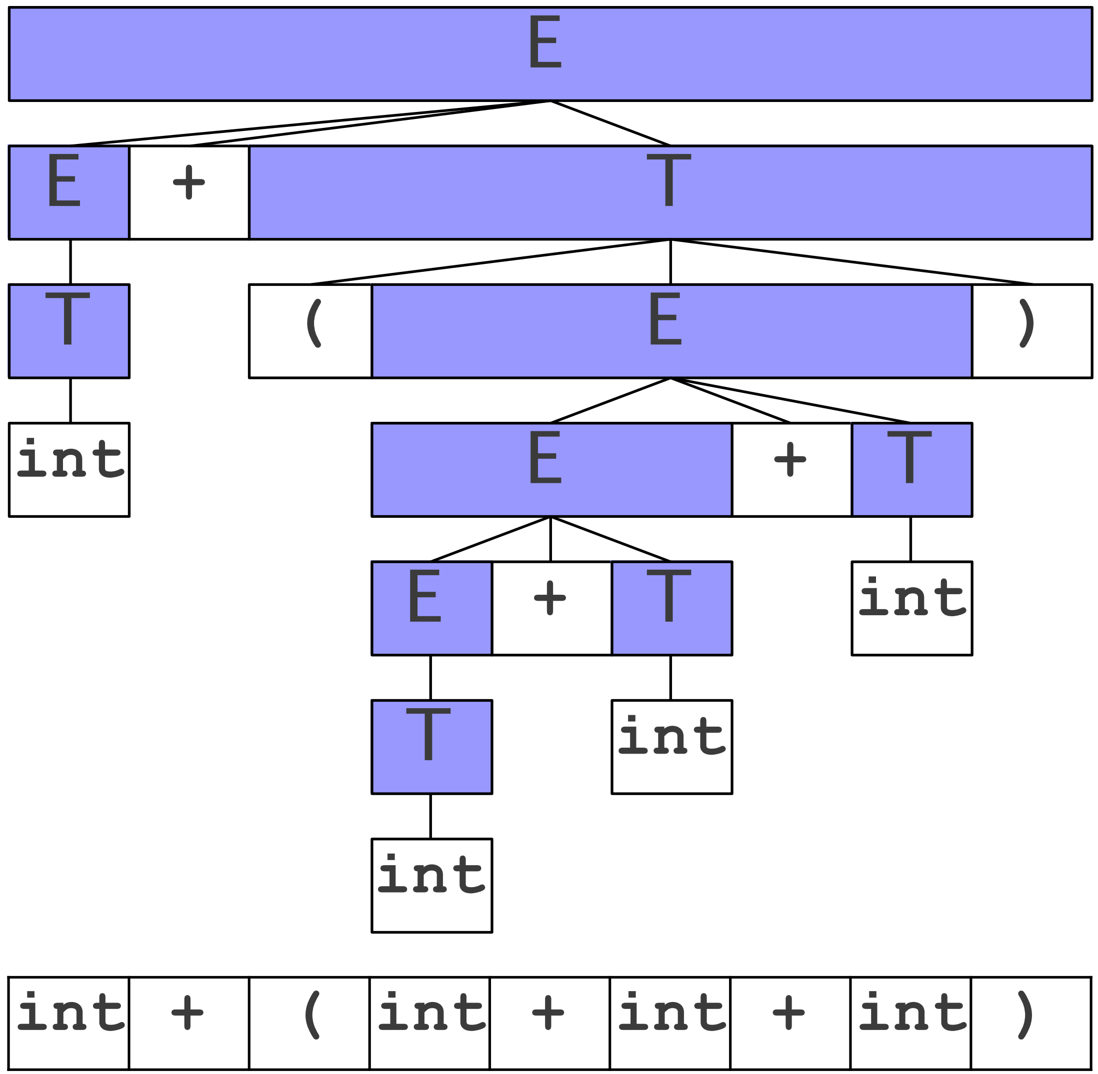
A Third View of a Bottom-Up Parse

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

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

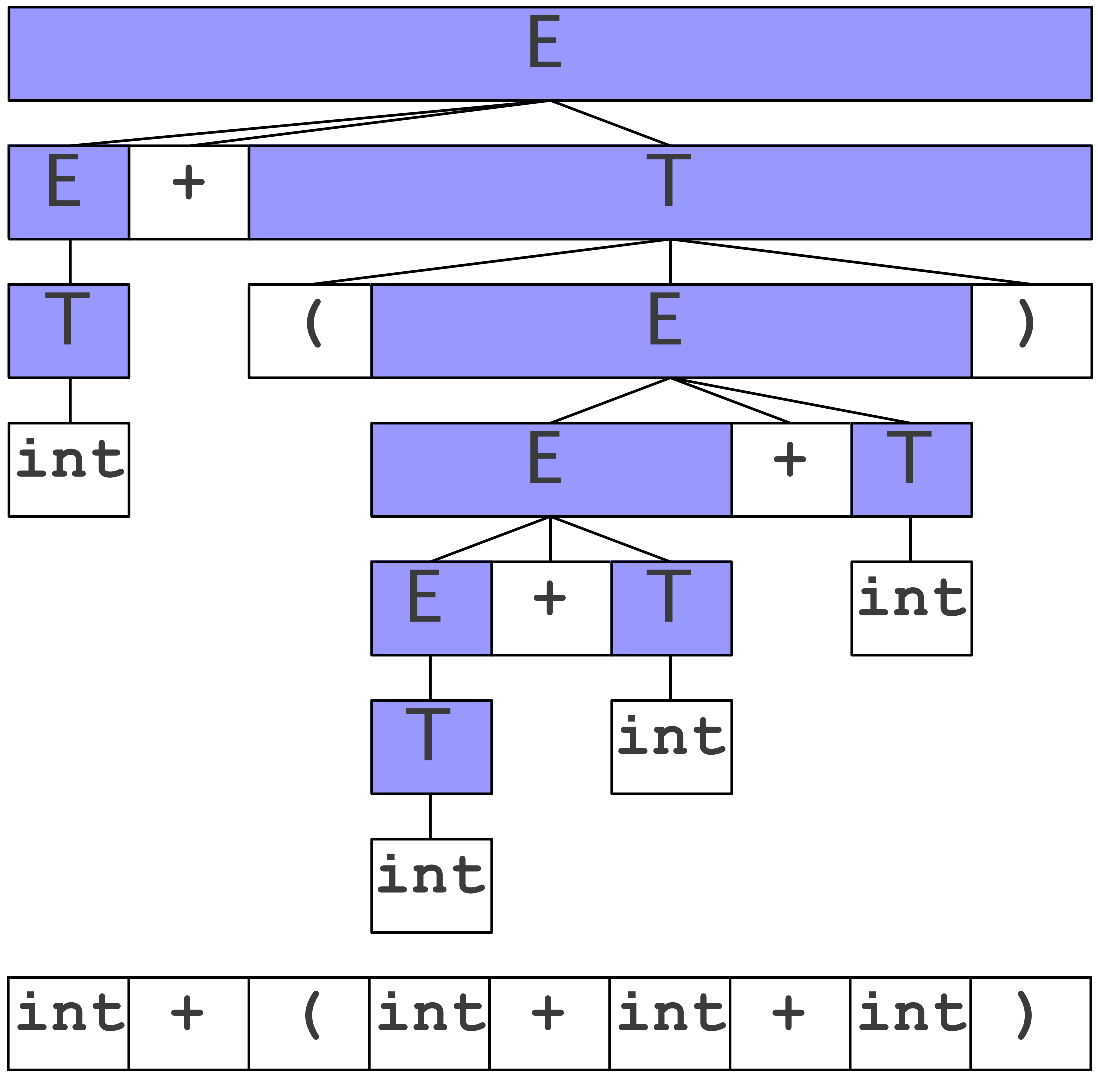
A Third View of a Bottom-Up Parse

```
  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
```



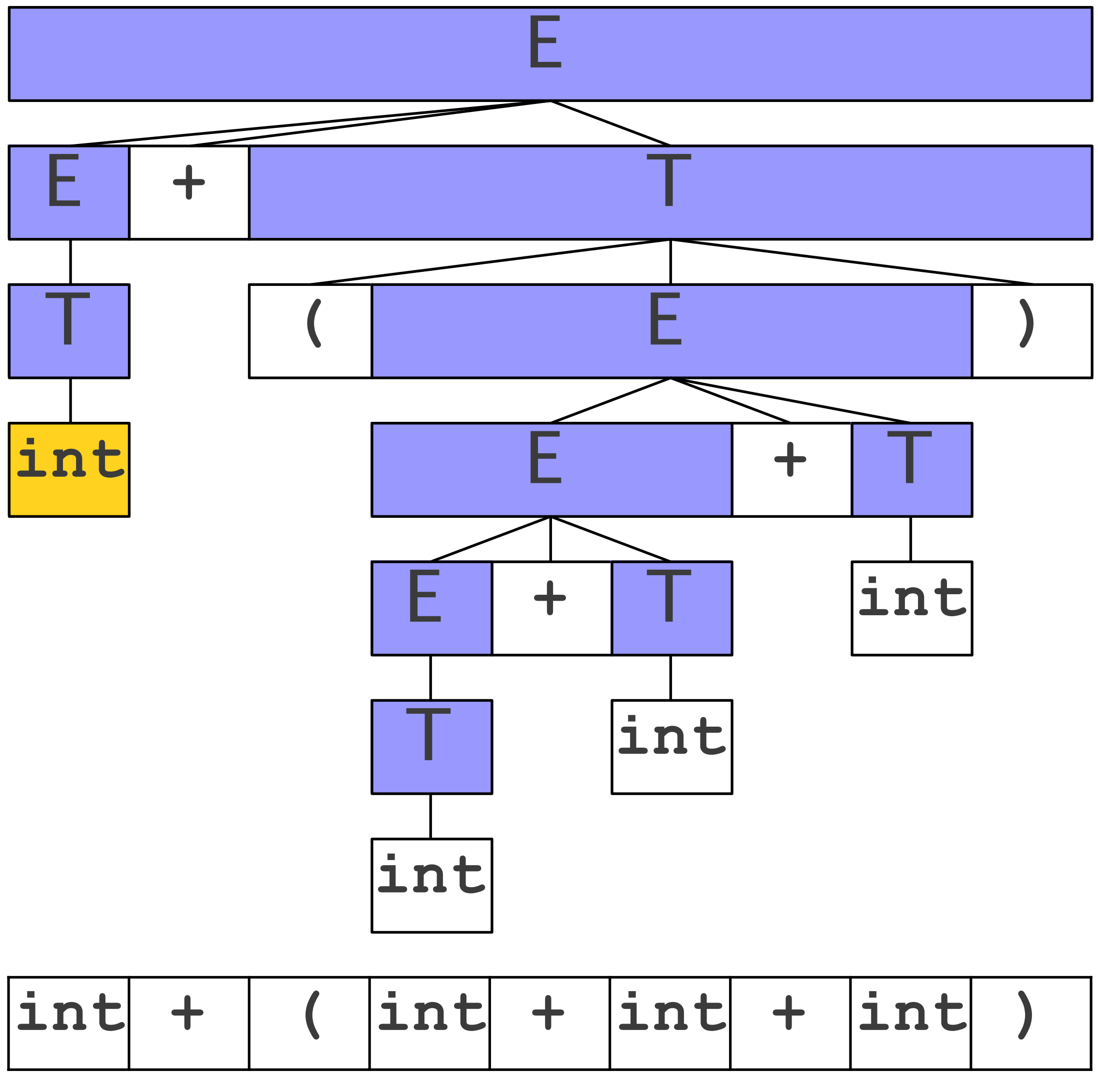
A Third View of a Bottom-Up Parse

```
  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
```



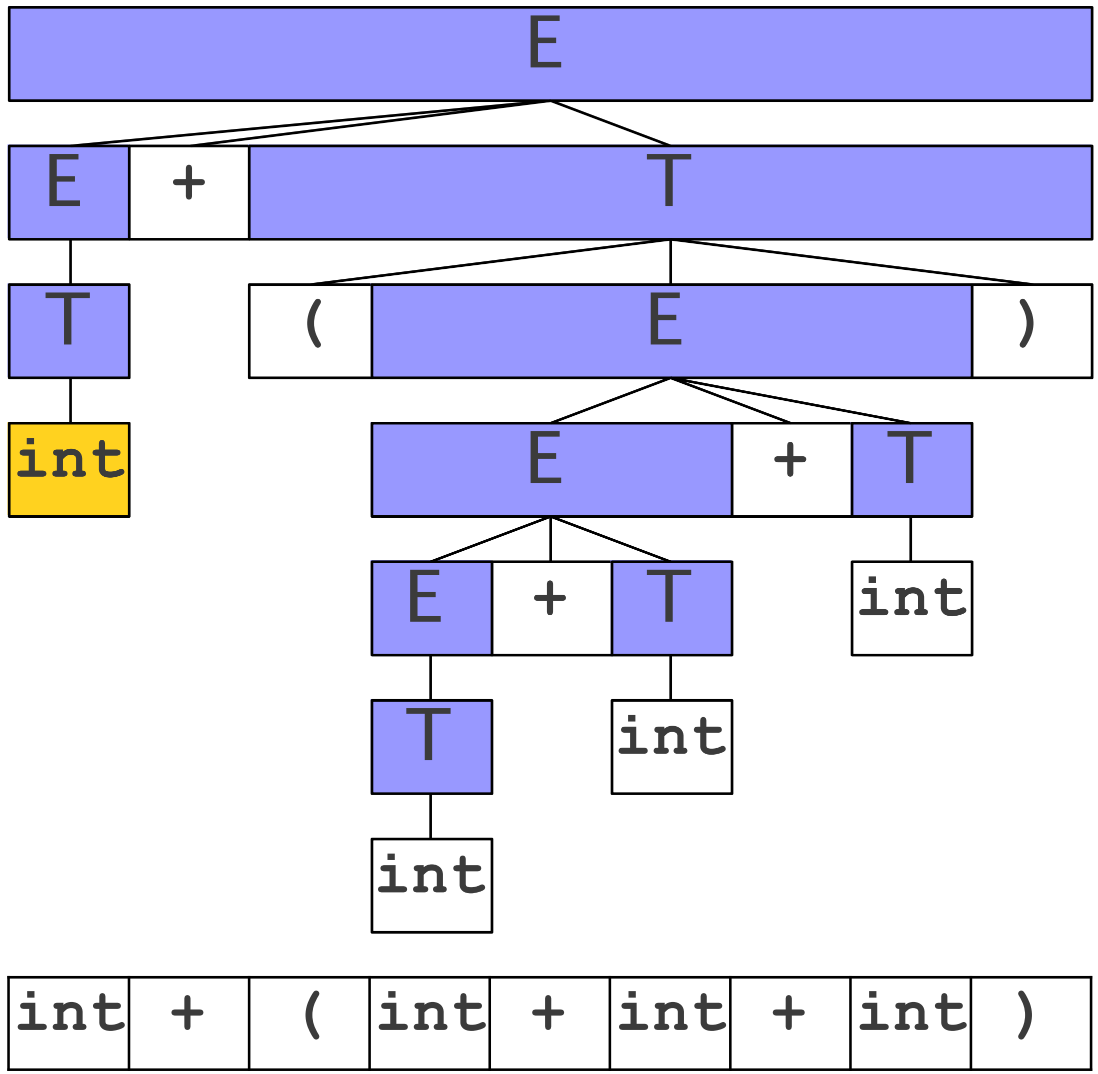
A Third View of a Bottom-Up Parse

```
  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
```



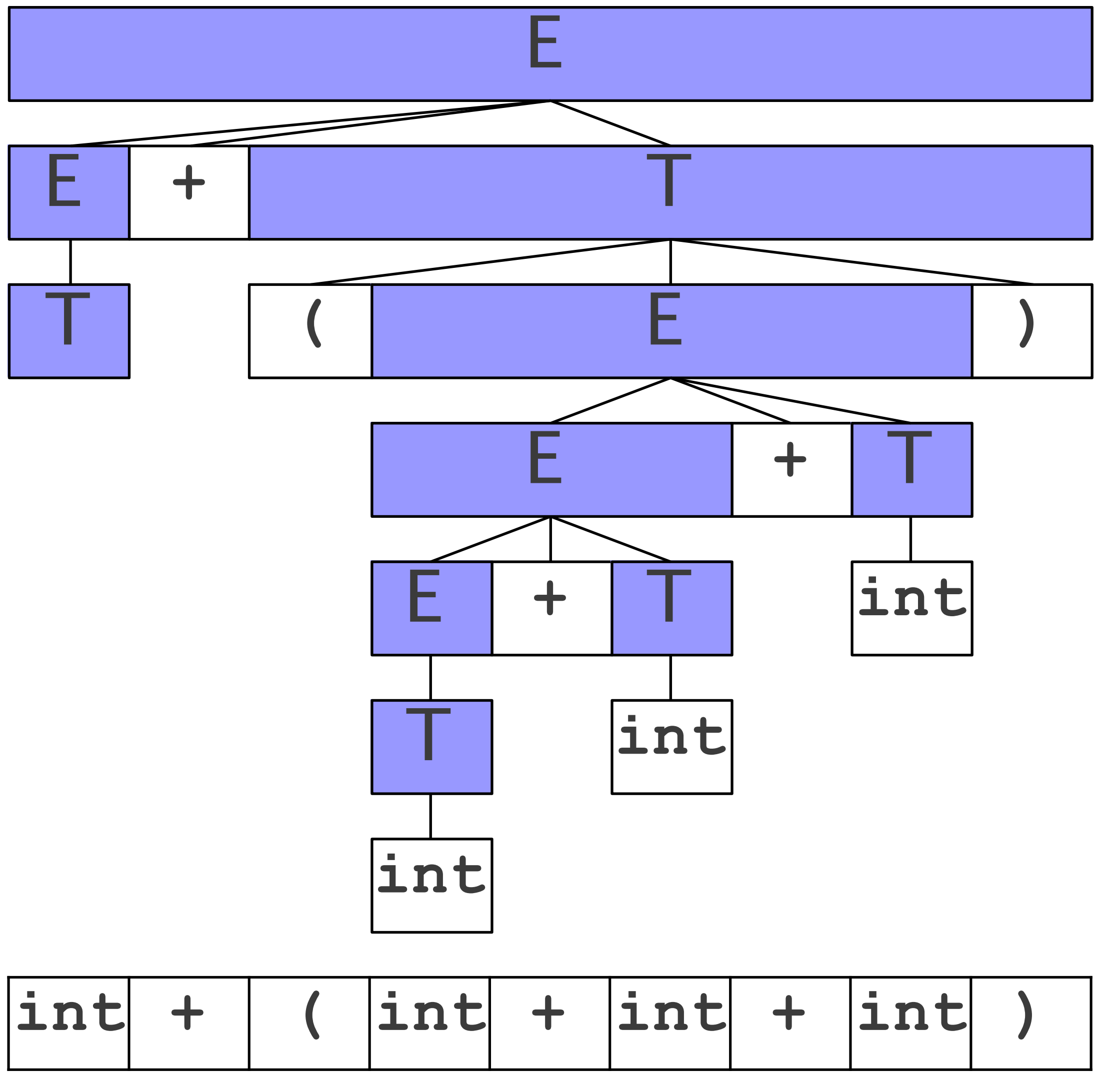
A Third View of a Bottom-Up Parse

```
  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
```



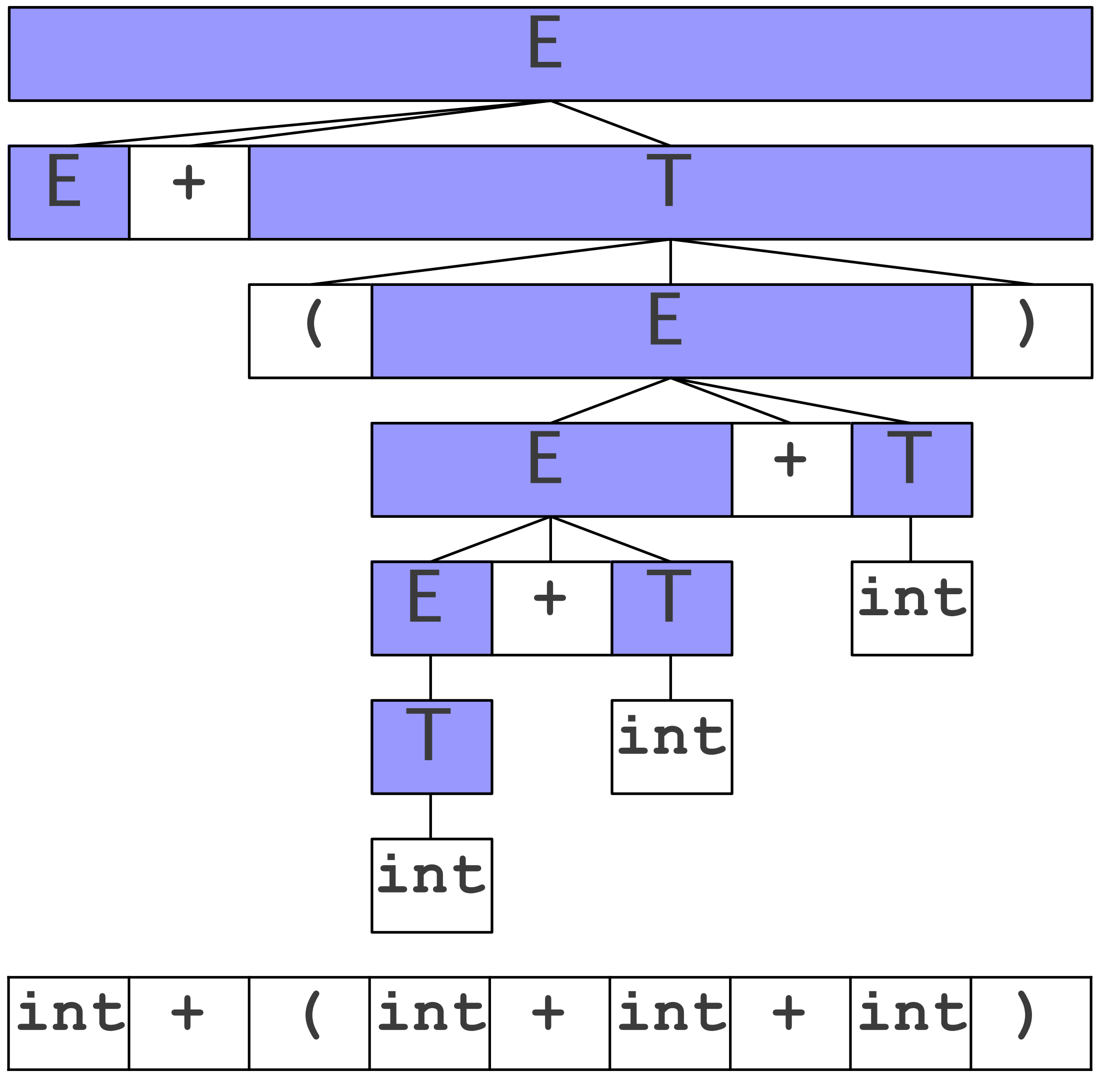
A Third View of a Bottom-Up Parse

⇒ 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



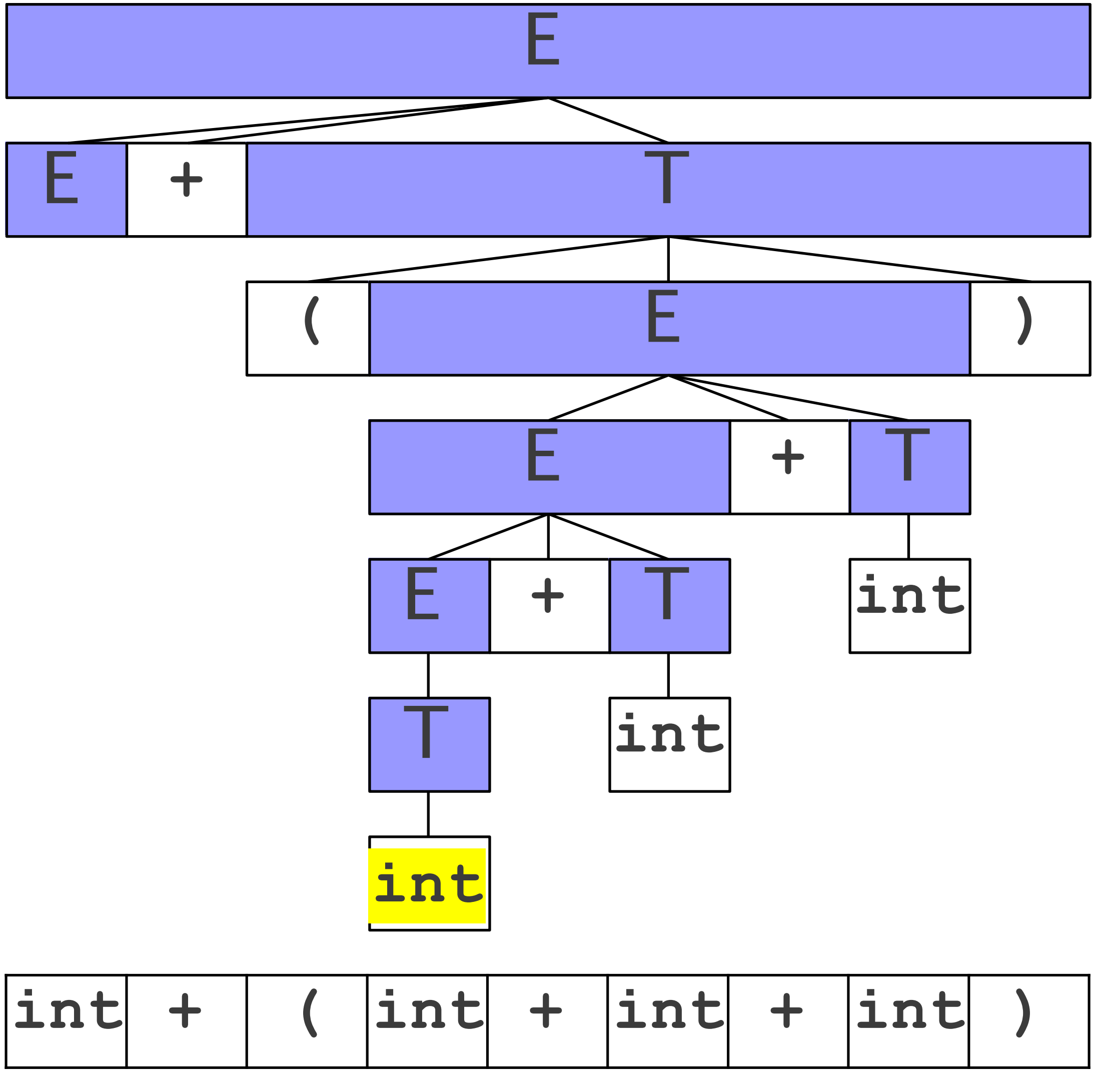
A Third View of a Bottom-Up Parse

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



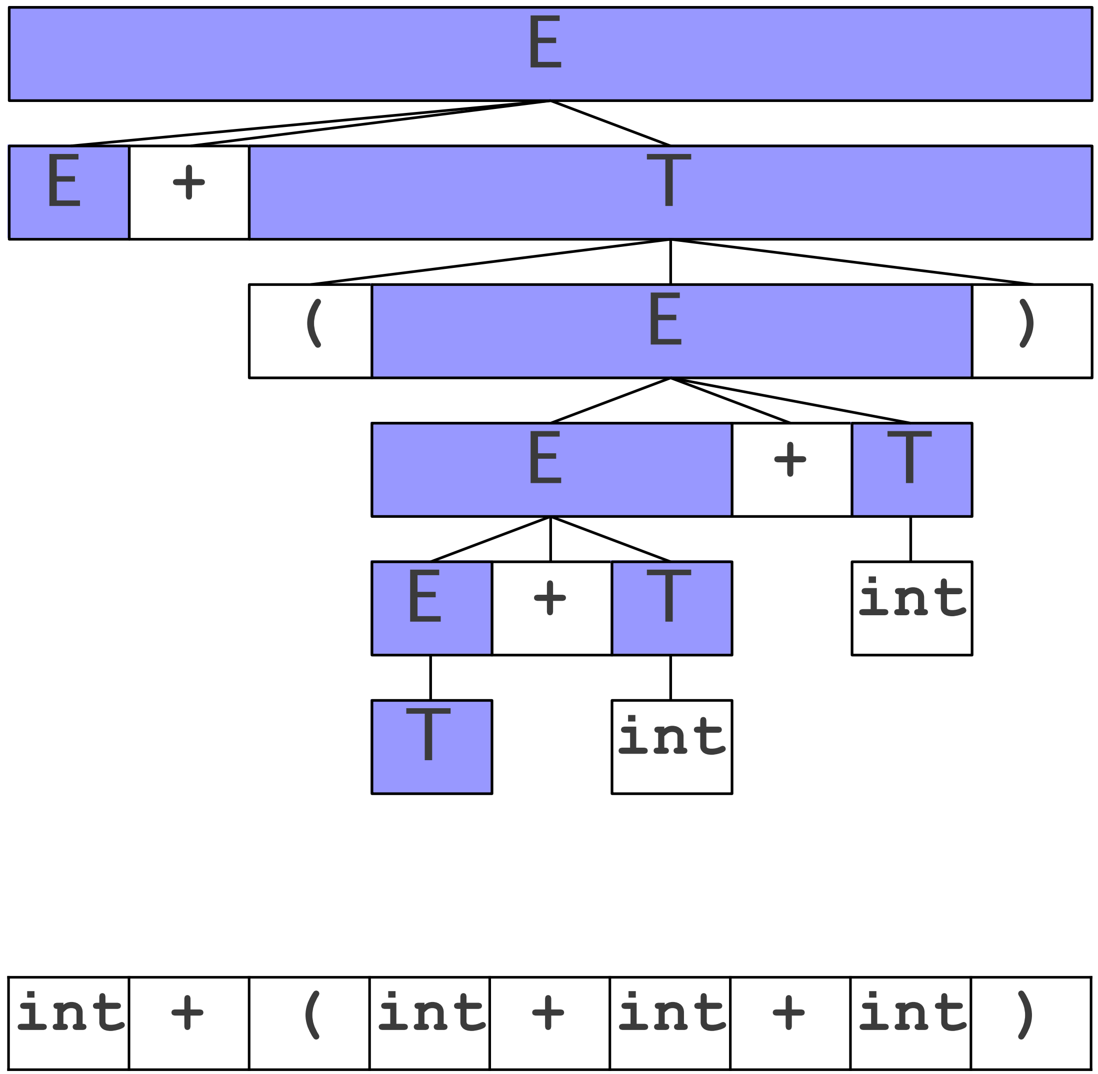
A Third View of a Bottom-Up Parse

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



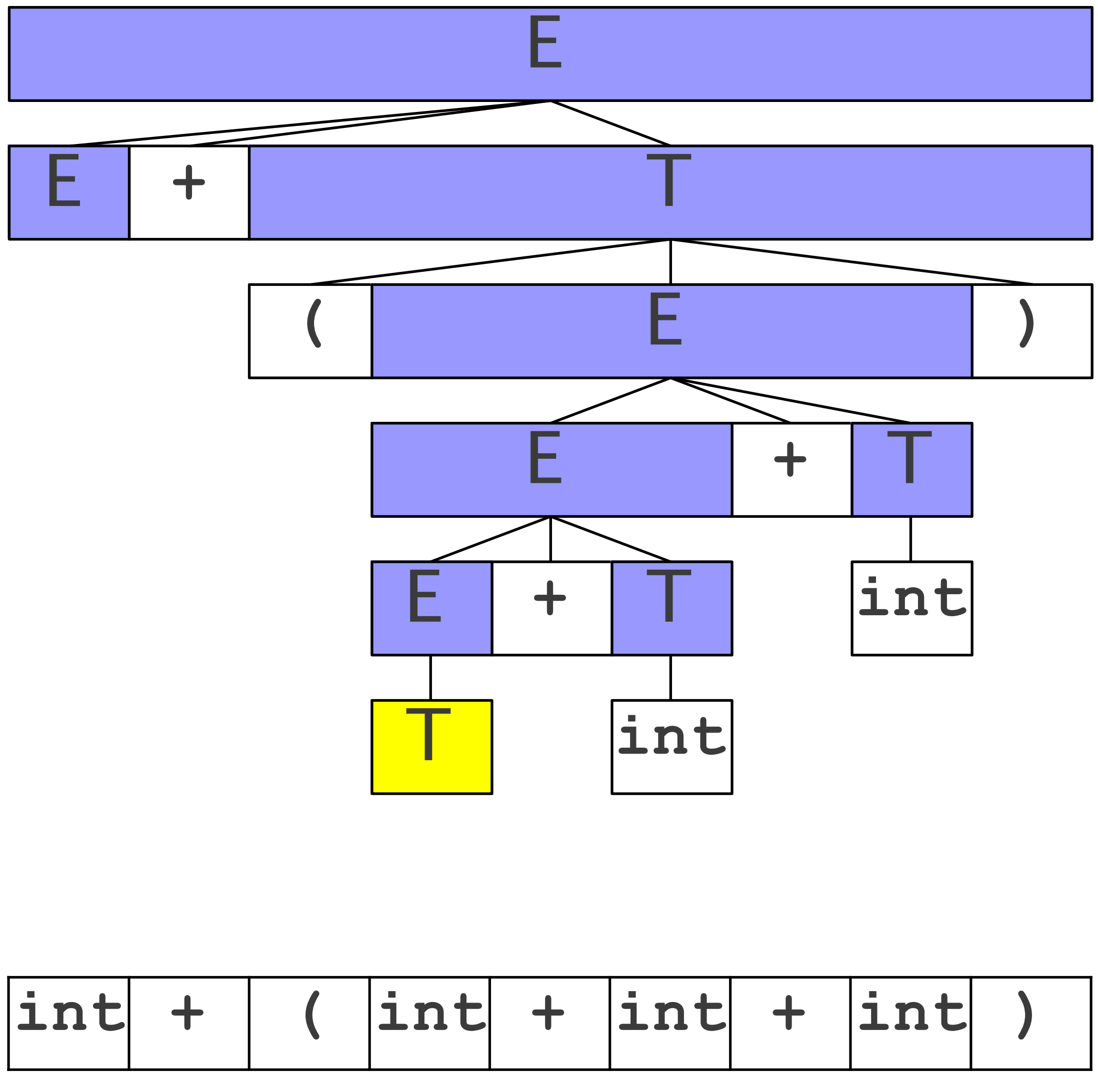
A Third View of a Bottom-Up Parse

$\Rightarrow$ `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`



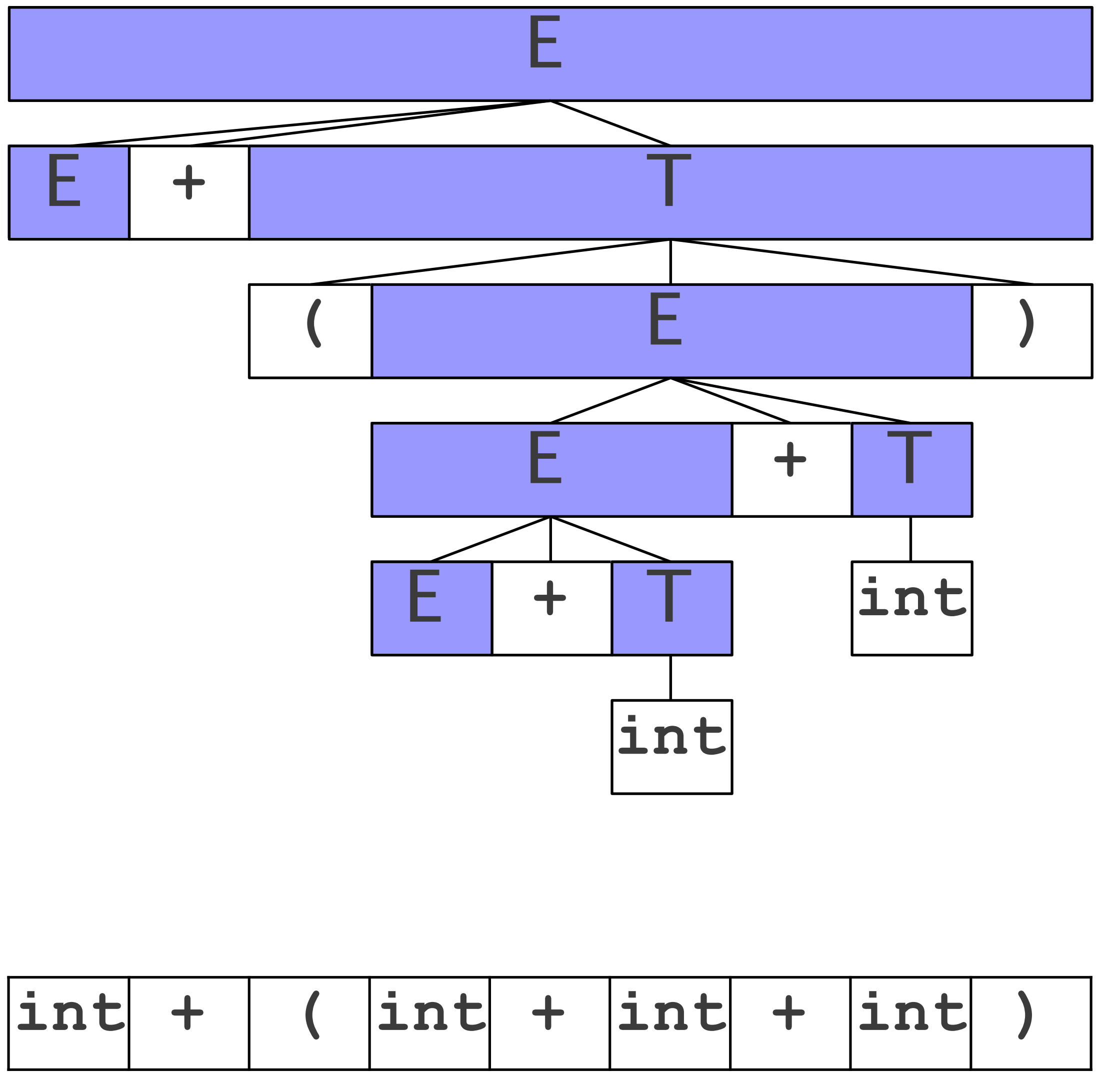
A Third View of a Bottom-Up Parse

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



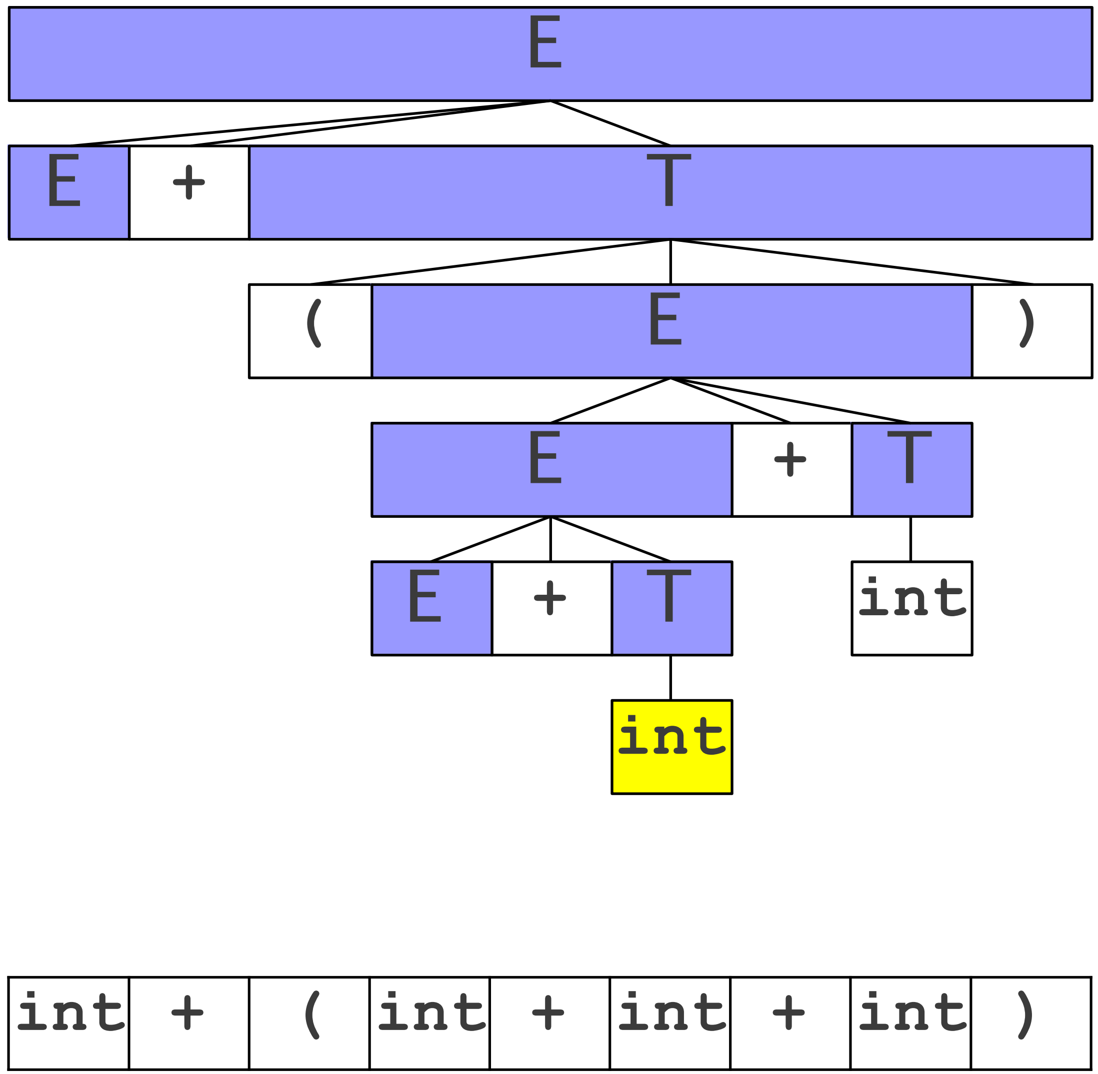
A Third View of a Bottom-Up Parse

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

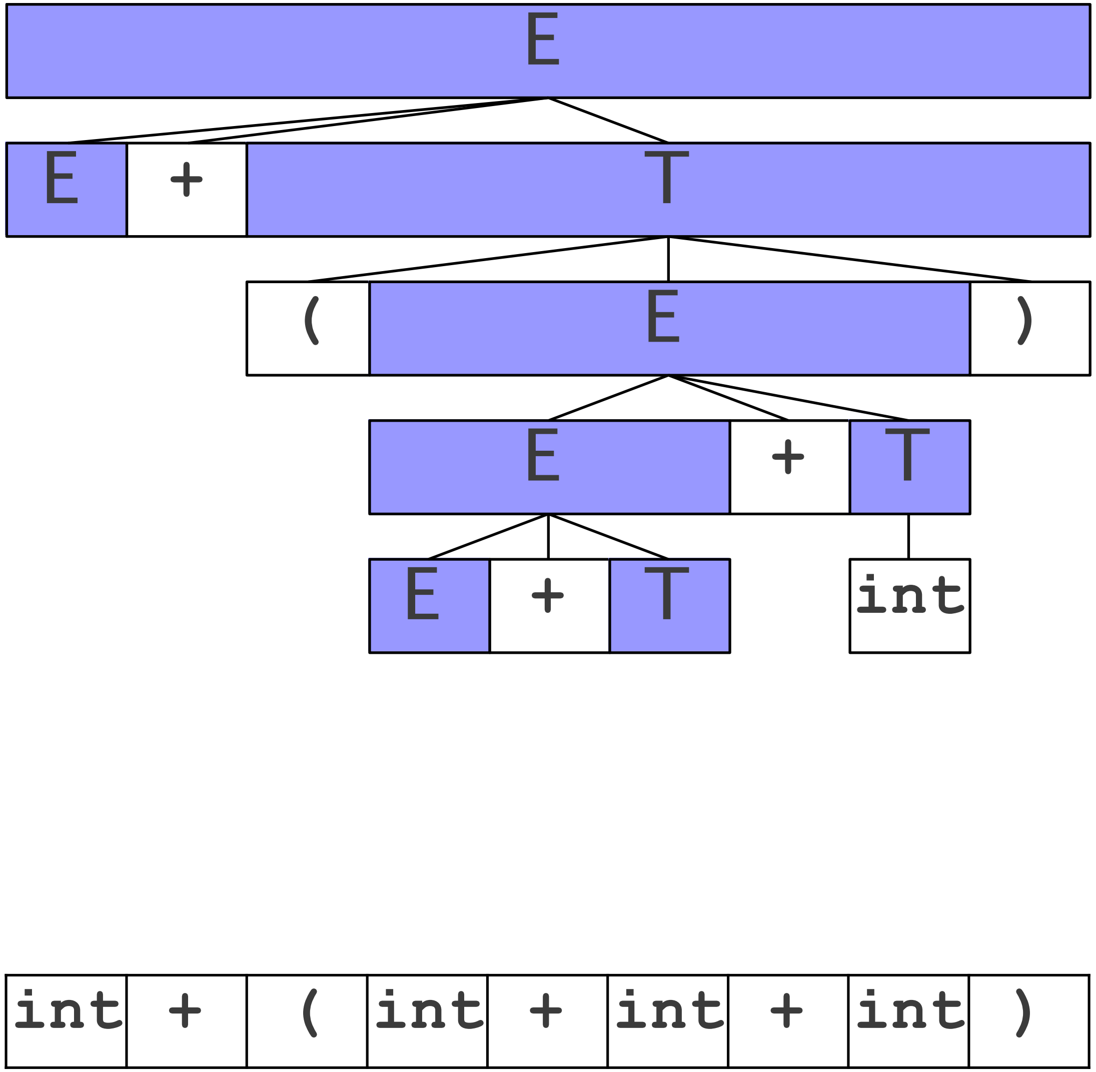


A Third View of a Bottom-Up Parse

⇒ 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)**

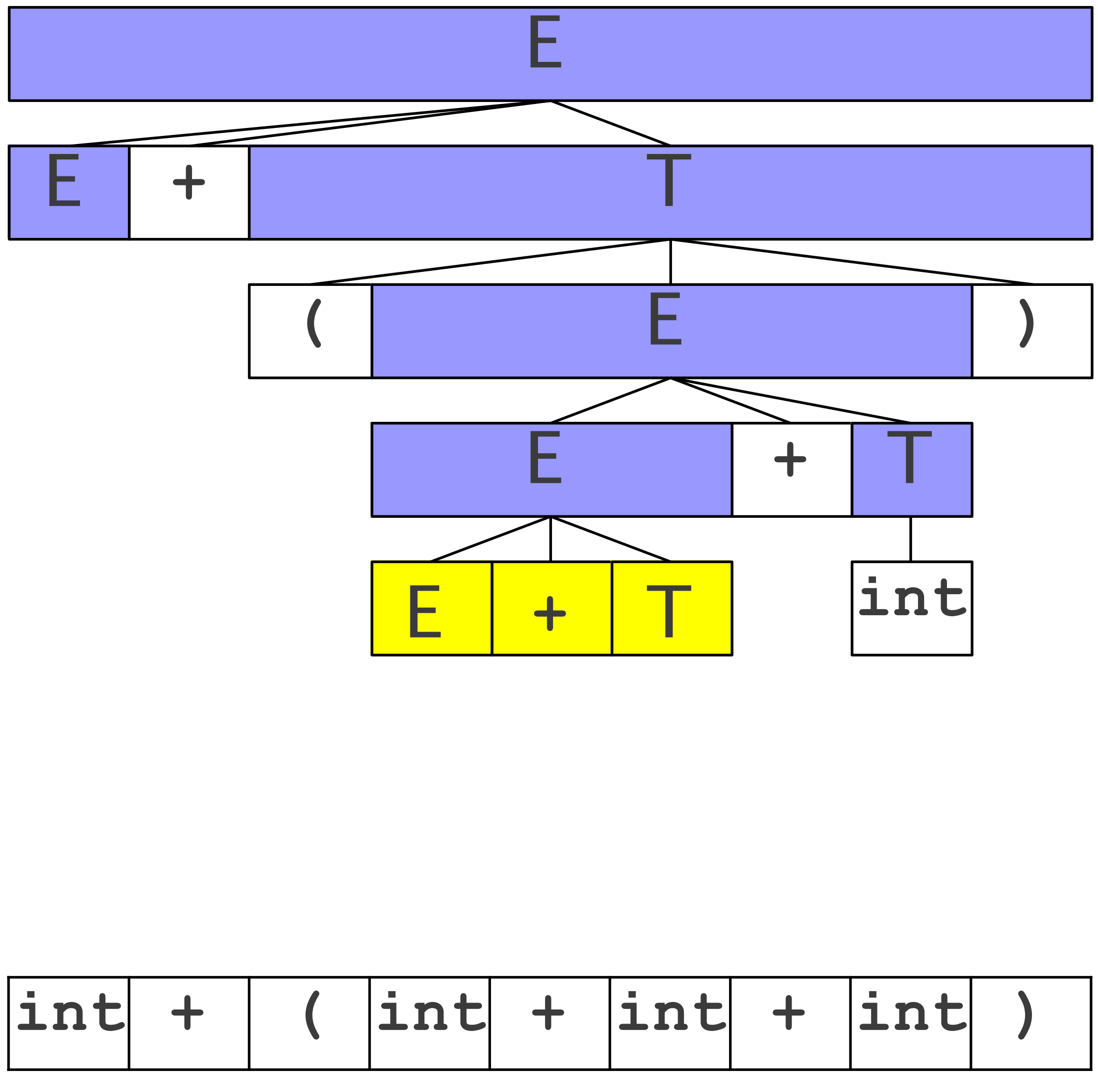
$\Rightarrow$ **E + (E + T)**

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

$\Rightarrow$ **E + T**

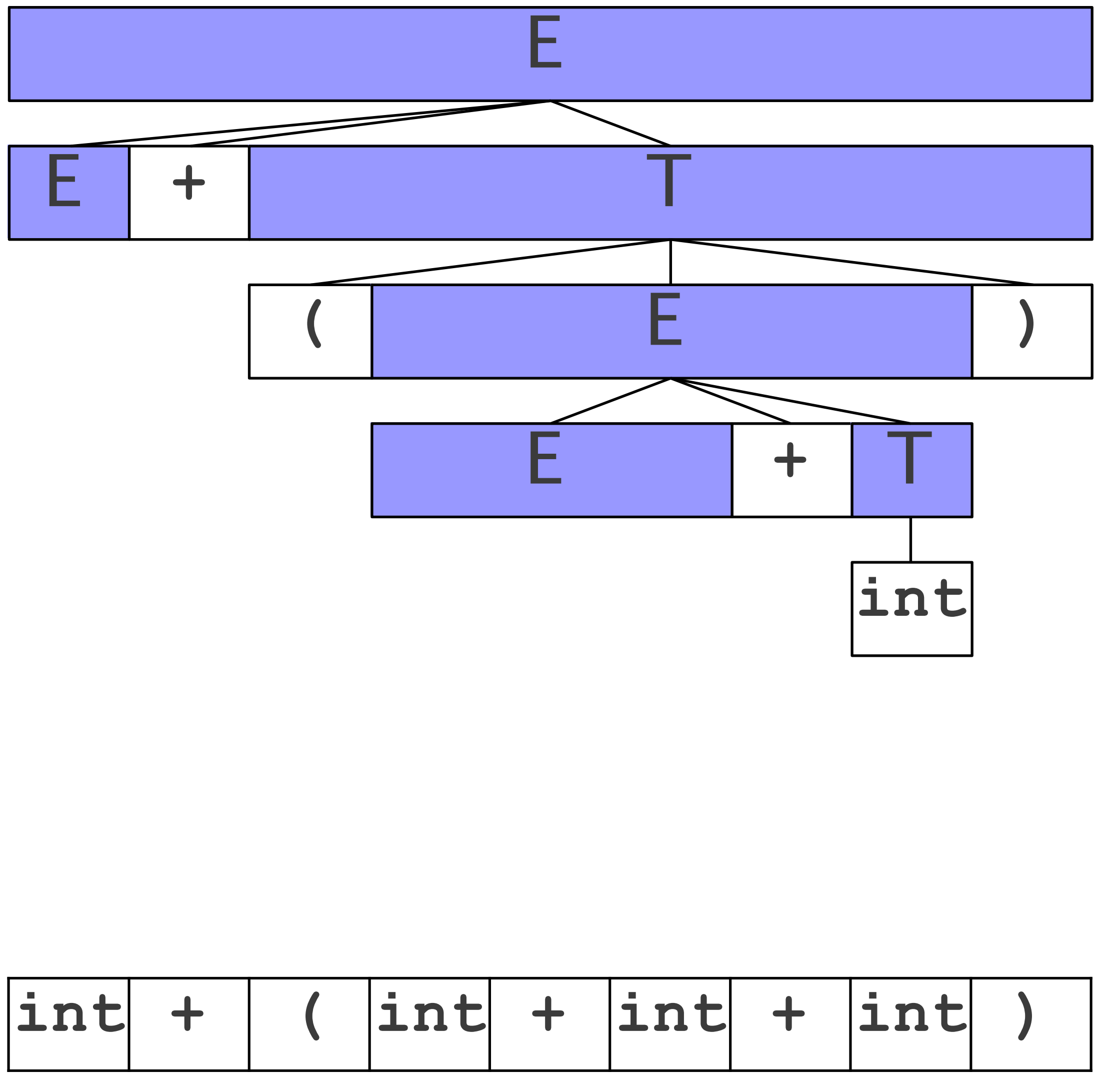
$\Rightarrow$ **E**

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



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

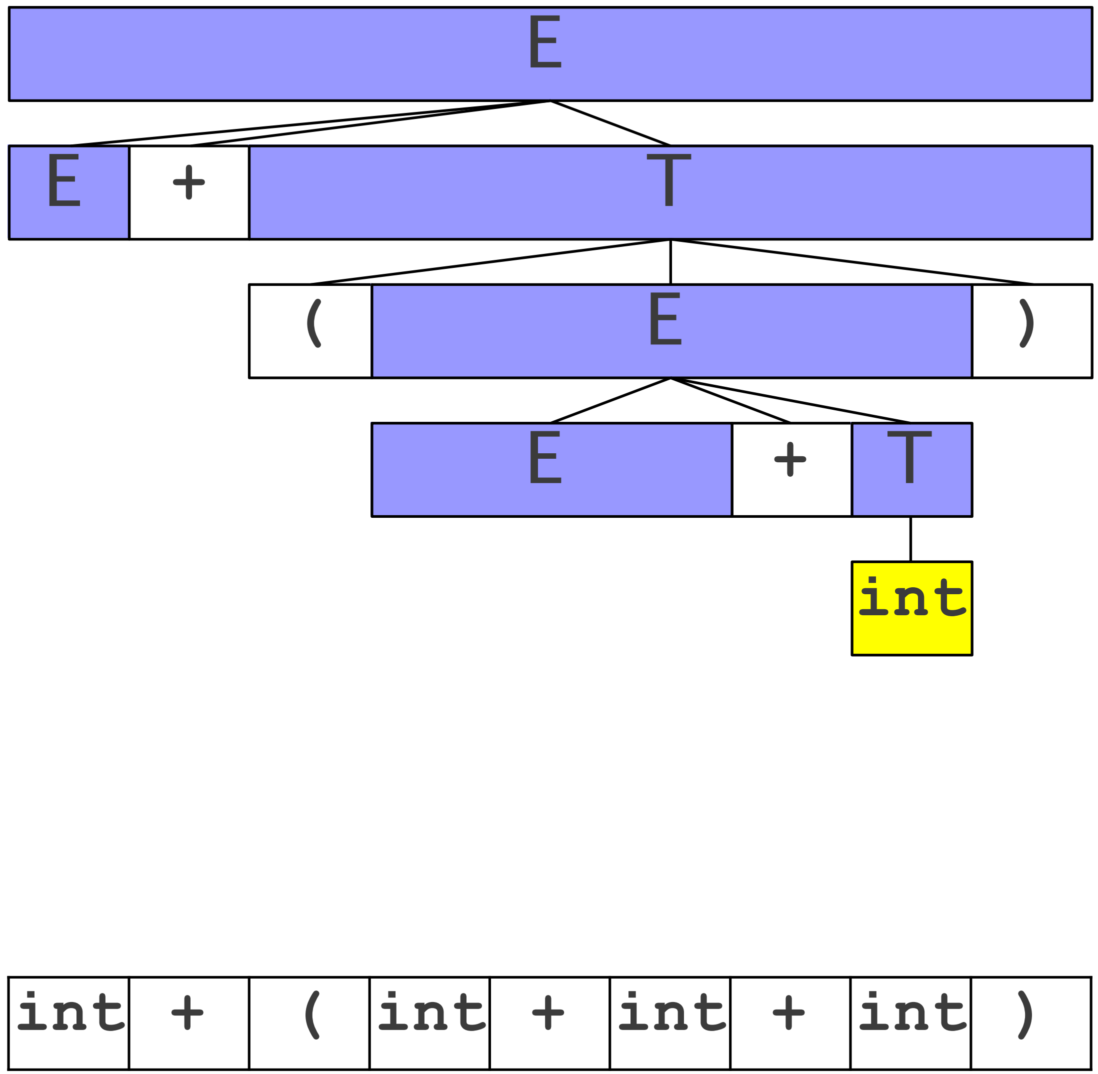
$\Rightarrow$ `E + (E + T)`

$\Rightarrow$ `E + (E)`

$\Rightarrow$ `E + T`

$\Rightarrow$ `E`

A Third View of a Bottom-Up Parse



⇒ E + (E + int)

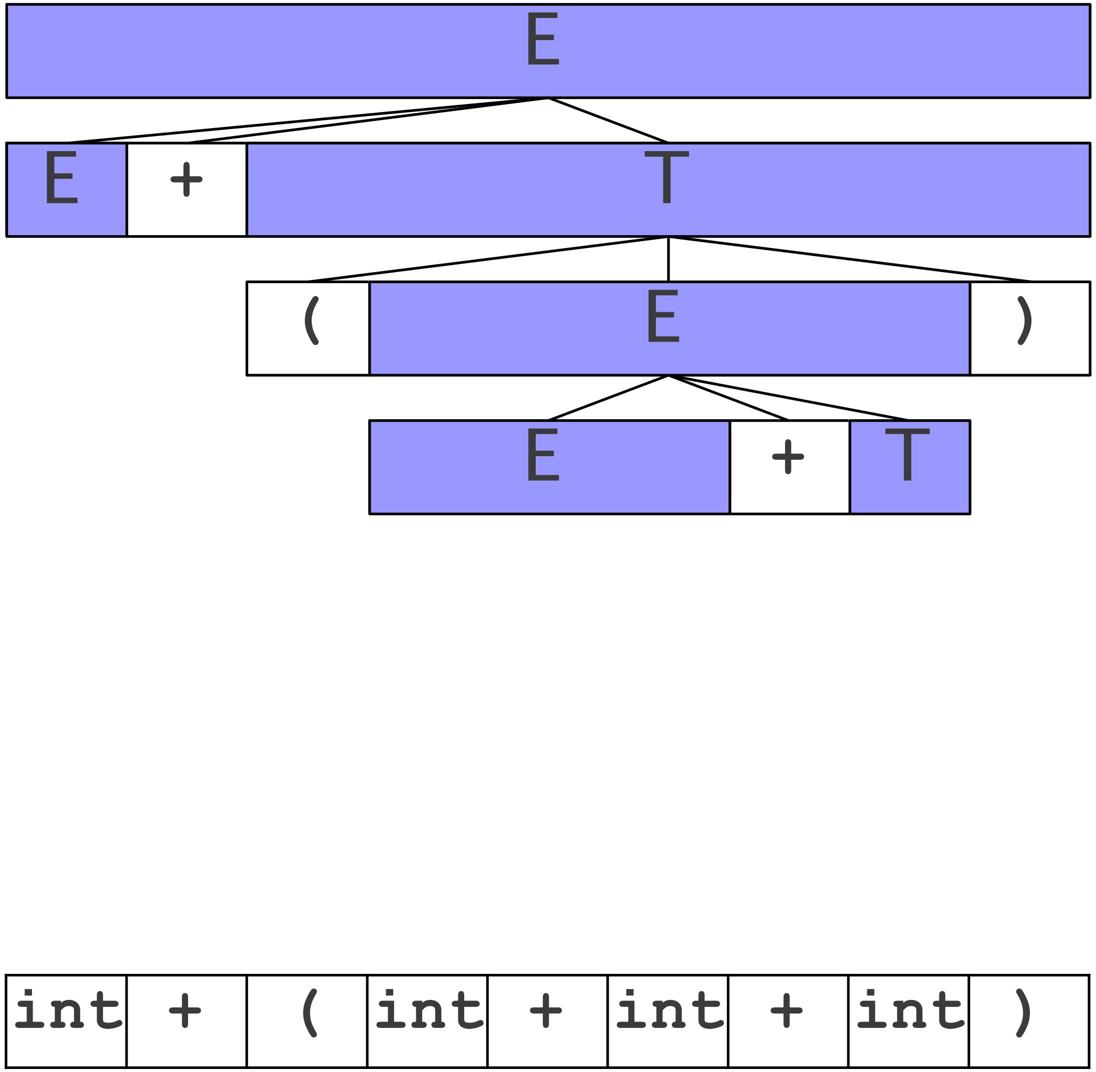
⇒ E + (E + T)

⇒ E + (E)

⇒ E + T

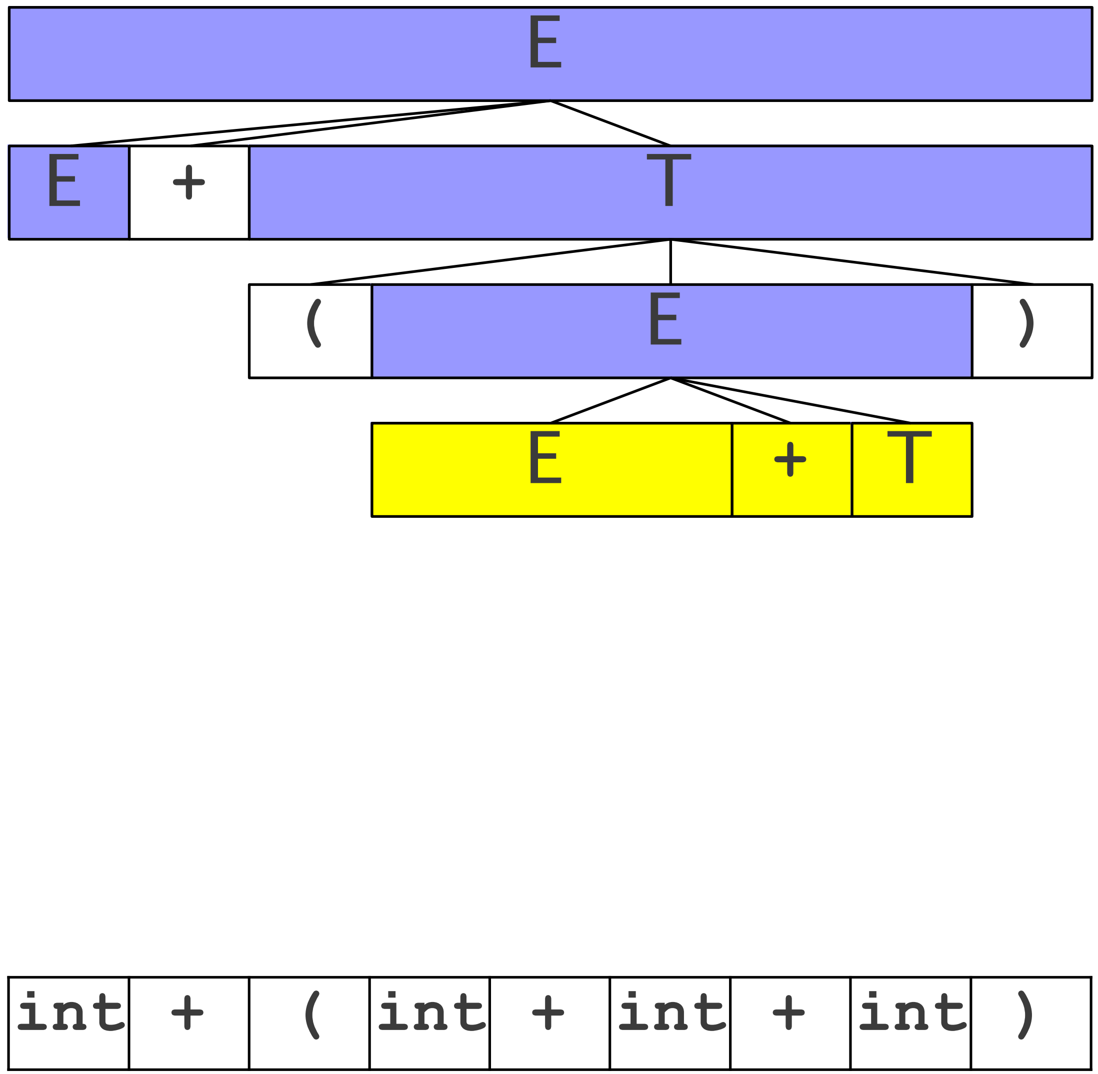
⇒ E

A Third View of a Bottom-Up Parse



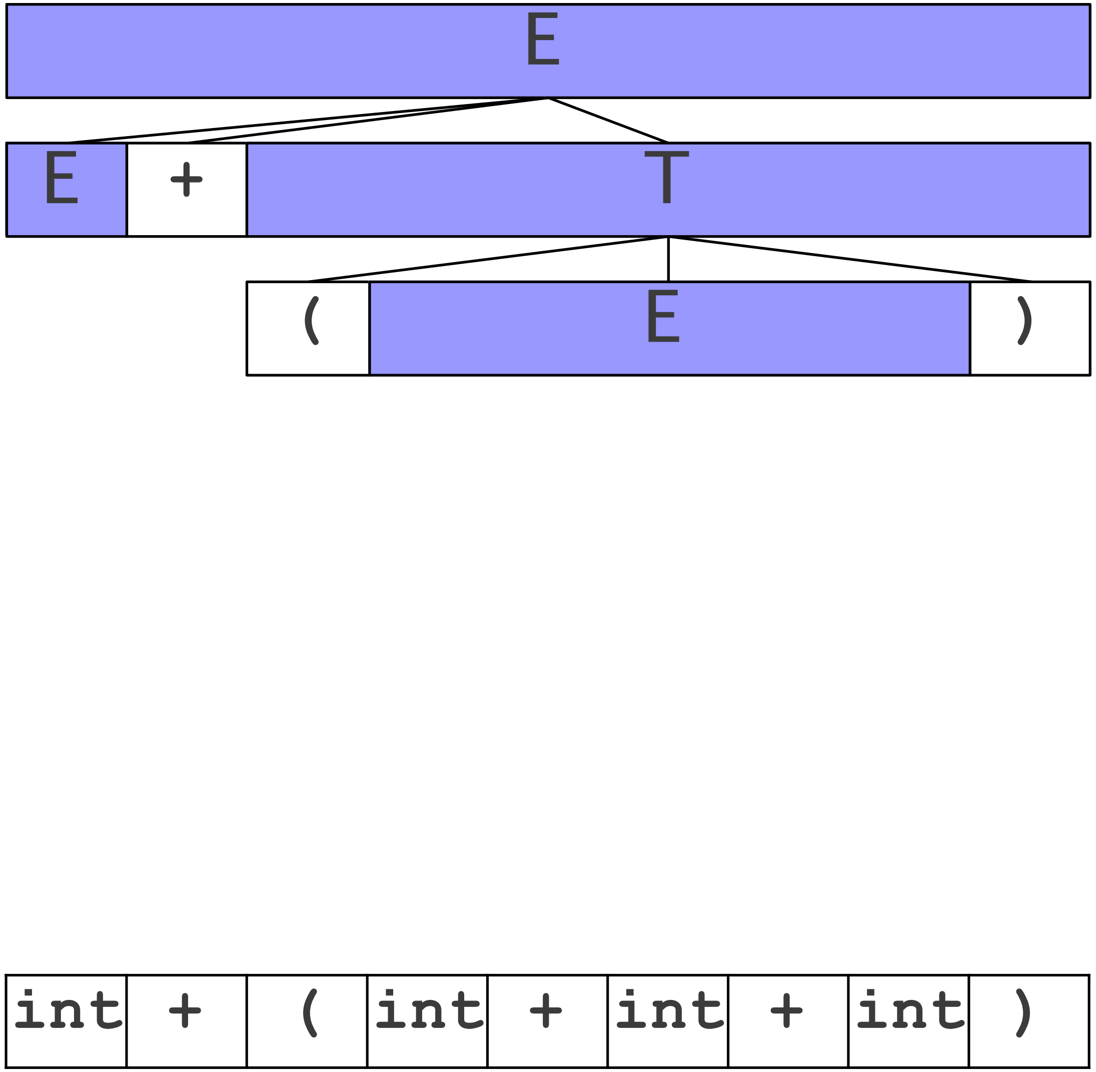
⇒ **E + (E + T)**
⇒ **E + (E)**
⇒ **E + T**
⇒ **E**

A Third View of a Bottom-Up Parse



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

A Third View of a Bottom-Up Parse

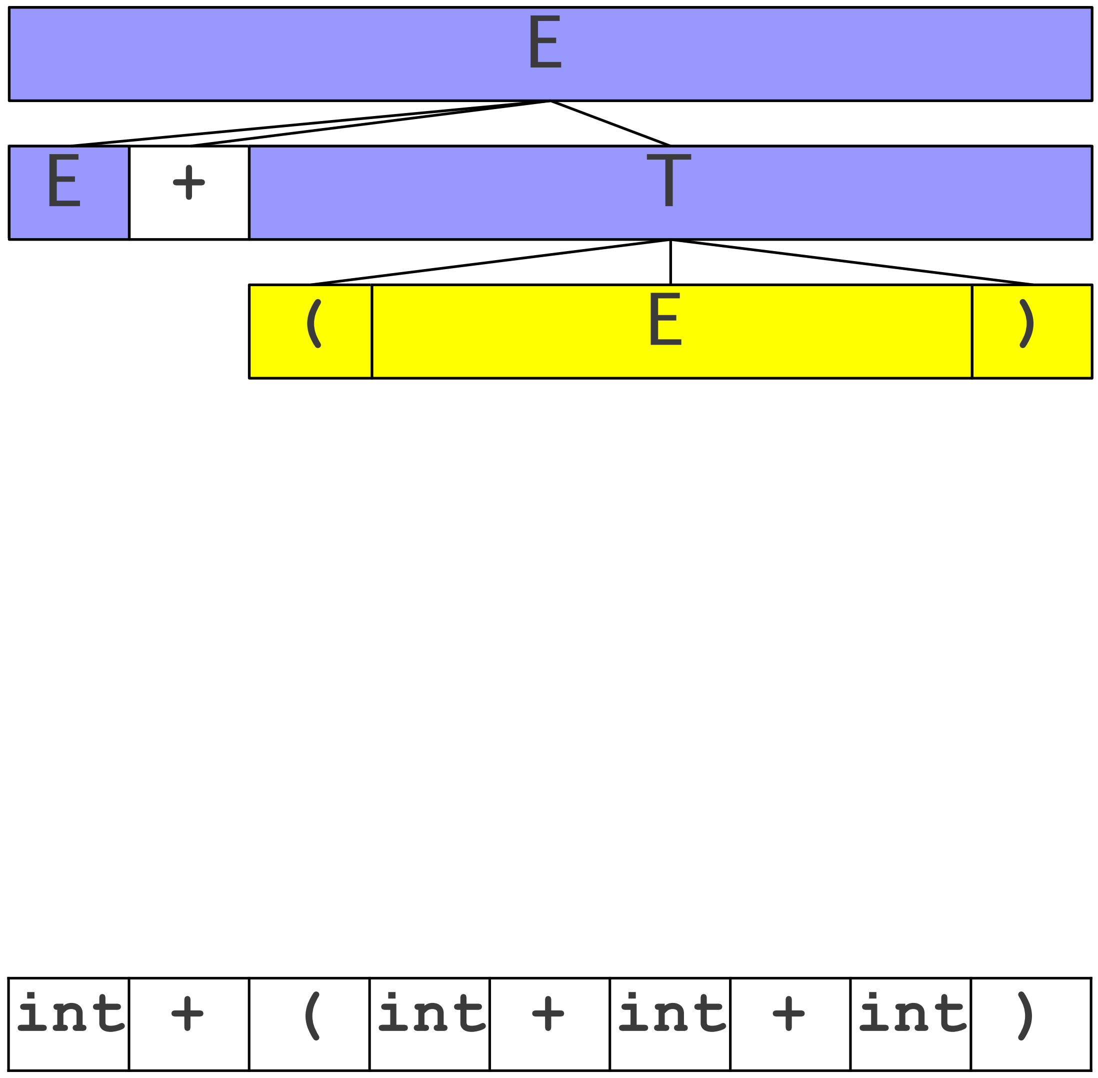


$\Rightarrow$ **E + (E)**

$\Rightarrow$ **E + T**

$\Rightarrow$ **E**

A Third View of a Bottom-Up Parse

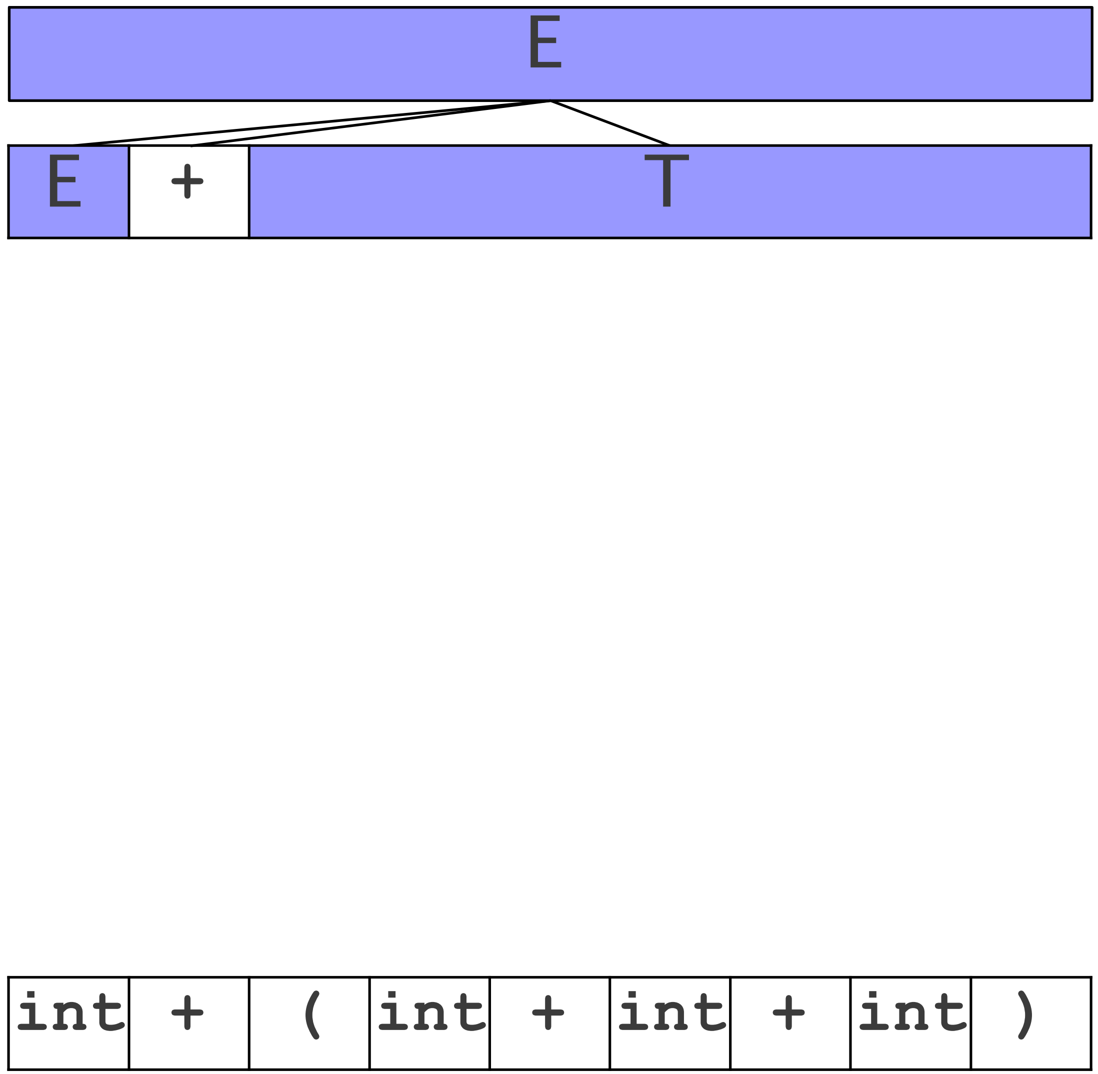


$\Rightarrow$ **E + (E)**

$\Rightarrow$ **E + T**

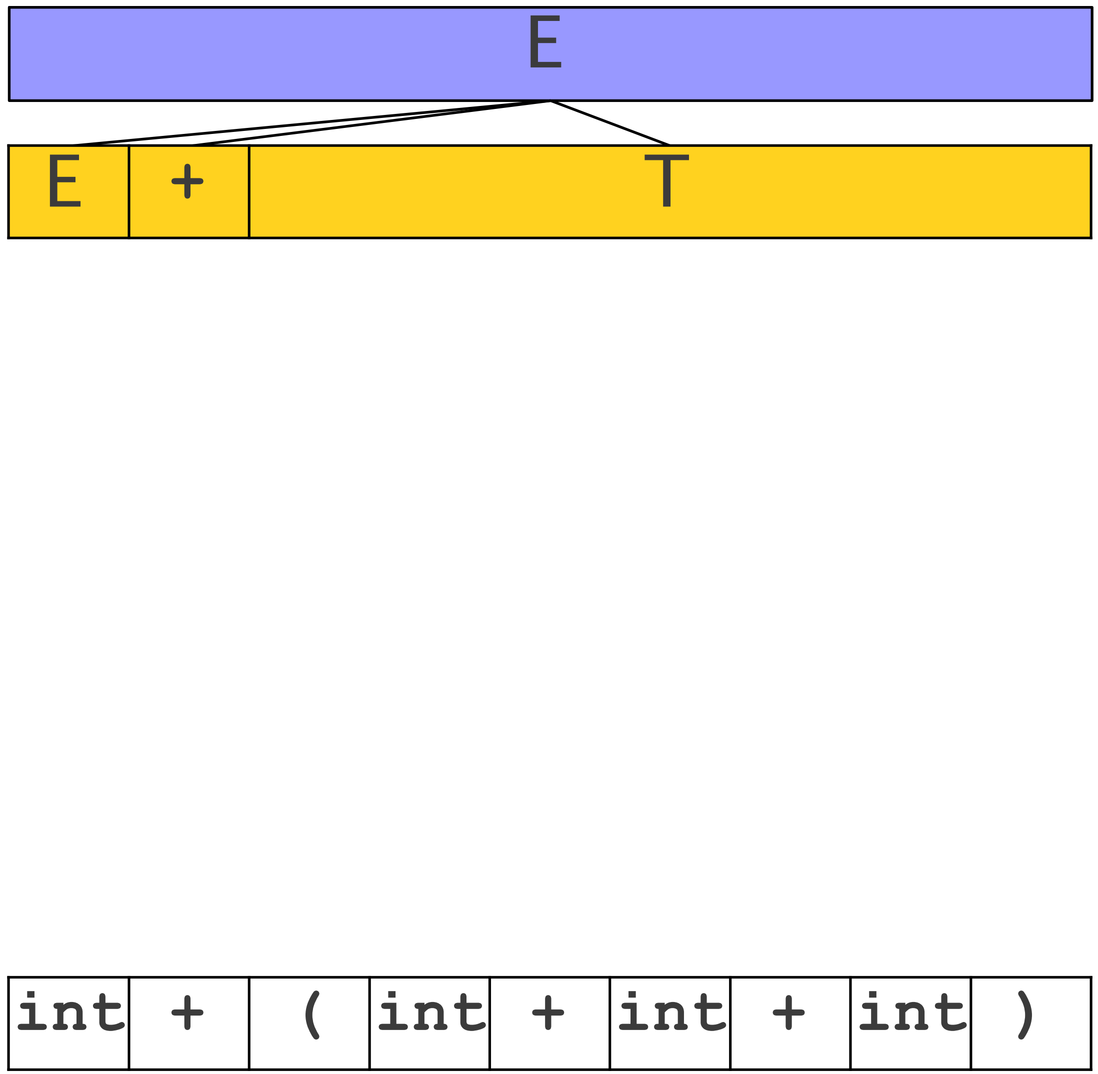
$\Rightarrow$ **E**

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.

Handles

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

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

- A **handle** of a right sentential form γ ($\equiv \alpha\beta\omega$) is a production rule $A \rightarrow \beta$ and a position of γ where the string β may be found and replaced by A to produce the previous right-sentential form in a rightmost derivation of γ.

$$S \underset{rm}{\Rightarrow}^{*} \alpha \mathbf{A} \omega \underset{rm}{\Rightarrow}^{*} \alpha \boldsymbol{\beta} \omega$$

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

Handle Pruning

- A right-most derivation in reverse can be obtained by **handle-pruning**.

$$S=\gamma_0 \overset{rm}{\Rightarrow} \gamma_1 \overset{rm}{\Rightarrow} \gamma_2 \overset{rm}{\Rightarrow} \ldots \overset{rm}{\Rightarrow} \gamma_{n-1} \overset{rm}{\Rightarrow} \gamma_n = \omega$$

input string

- Start from γ_n, find a handle $A_n \rightarrow \beta_n$ in γ_n, and replace β_n in by A_n to get γ_{n-1}.
- Then find a handle $A_{n-1} \rightarrow \beta_{n-1}$ in γ_{n-1}, and replace β_{n-1} in by A_{n-1} to get γ_{n-2}.
- Repeat this, until reach the start nonterminal *S*.

A Shift-Reduce Parser

$E \rightarrow E+T \mid T$

$T \rightarrow T*F \mid F$

$F \rightarrow (E) \mid id$

Right-Most Derivation of id+id*id

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

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

Right-Most Sentential Form	Reducing Production
$\underline{id}+id*id$	$F \rightarrow id$
$\underline{F}+id*id$	$T \rightarrow F$
$\underline{T}+id*id$	$E \rightarrow T$
$E+\underline{id}*id$	$F \rightarrow id$
$E+\underline{F}*id$	$T \rightarrow F$
$E+T*\underline{id}$	$F \rightarrow id$
$E+\underline{T*F}$	$T \rightarrow T*F$
$\underline{E+T}$	$E \rightarrow E+T$
E	

Handles are red and underlined in the right-sentential forms.

A Stack Implementation of A Shift-Reduce Parser

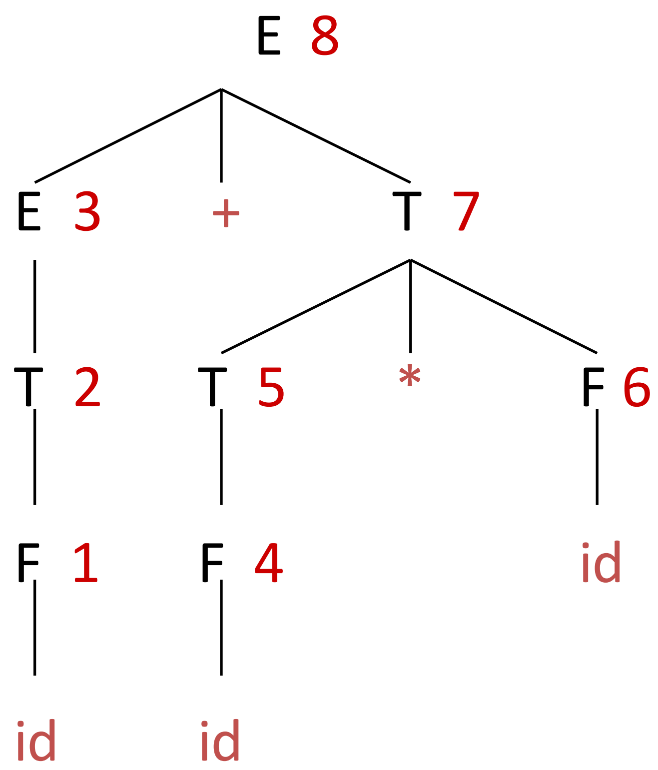
- There are four possible actions of a shift-parser action:
  - **Shift** : The next input symbol is shifted onto the top of the stack.
  - **Reduce**: Replace the handle on the top of the stack by the non-terminal.
  - **Accept**: Successful completion of parsing.
  - **Error**: Parser discovers a syntax error, and calls an error recovery routine.

- Initial stack just contains only the end-marker **$**.
- The end of the input string is marked by the end-marker **$**.

A Stack Implementation of A Shift-Reduce Parser

Stack	**Input**	**Action**
\$	id+id*id\$	shift
\$id	+id*id\$	reduce by F → id
\$F	+id*id\$	reduce by T → F
\$T	+id*id\$	reduce by E → T
\$E	+id*id\$	shift
\$E+	id*id\$	shift
\$E+id	*id\$	reduce by F → id
\$E+F	*id\$	reduce by T → F
\$E+T	*id\$	shift
\$E+T*	id\$	shift
\$E+T*id	\$	reduce by F → id
\$E+T*F	\$	reduce by T → T*F
\$E+T	\$	reduce by E → E+T
\$E	\$	accept

Parse Tree



Summarizing the Intuition

- The first intuition (reconstructing the parse tree bottom-up) motivates how the parsing should work.
- The second intuition (rightmost derivation in reverse) describes the order in which we should build the parse tree.
- The third intuition (handle pruning) is the basis for the bottom-up parsing algorithms we will explore.

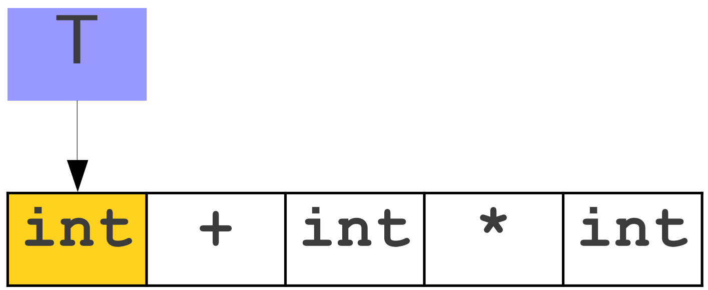
A Detail about Handles

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

int	+	int	*	int

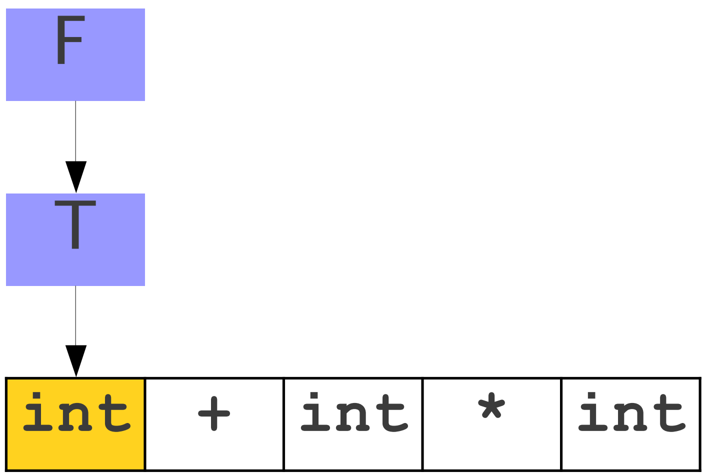
A Detail about Handles

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



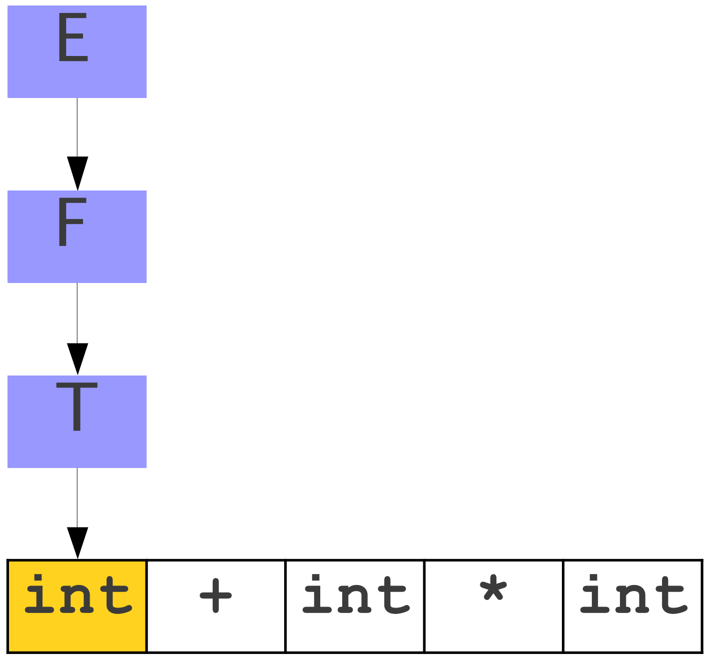
A Detail about Handles

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



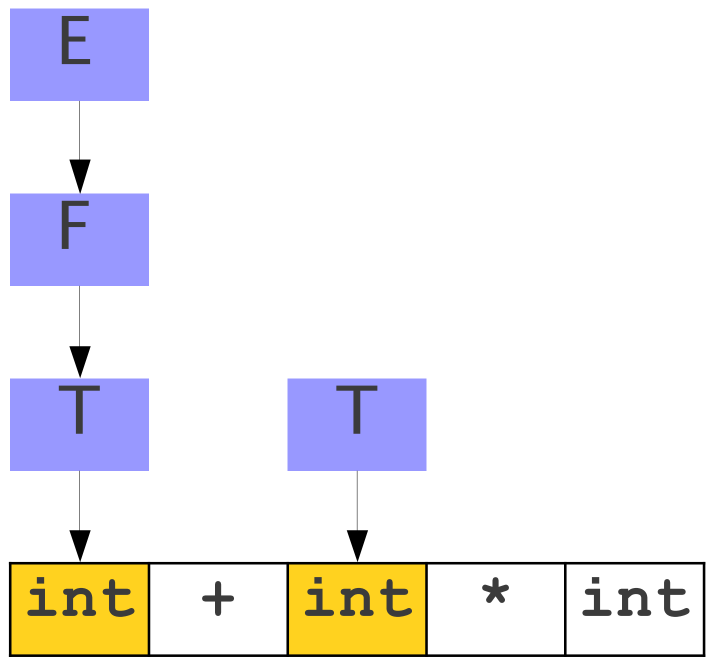
A Detail about Handles

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



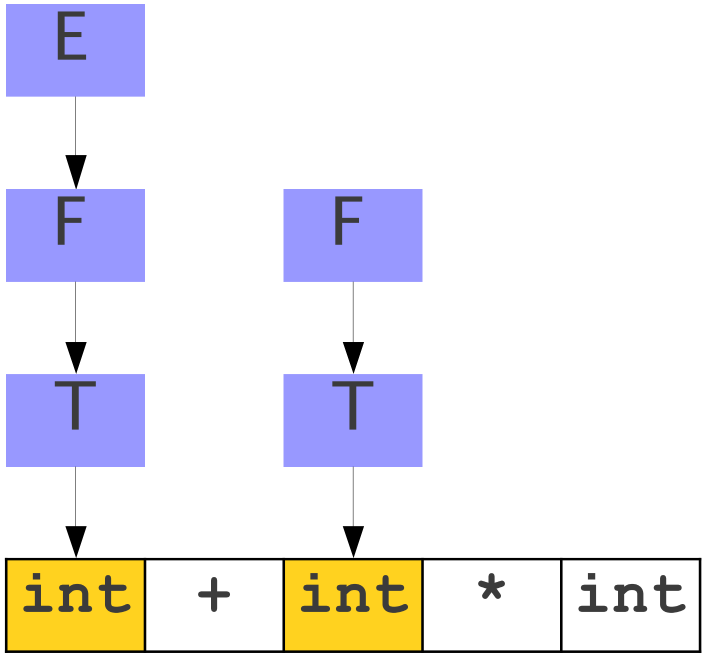
A Detail about Handles

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



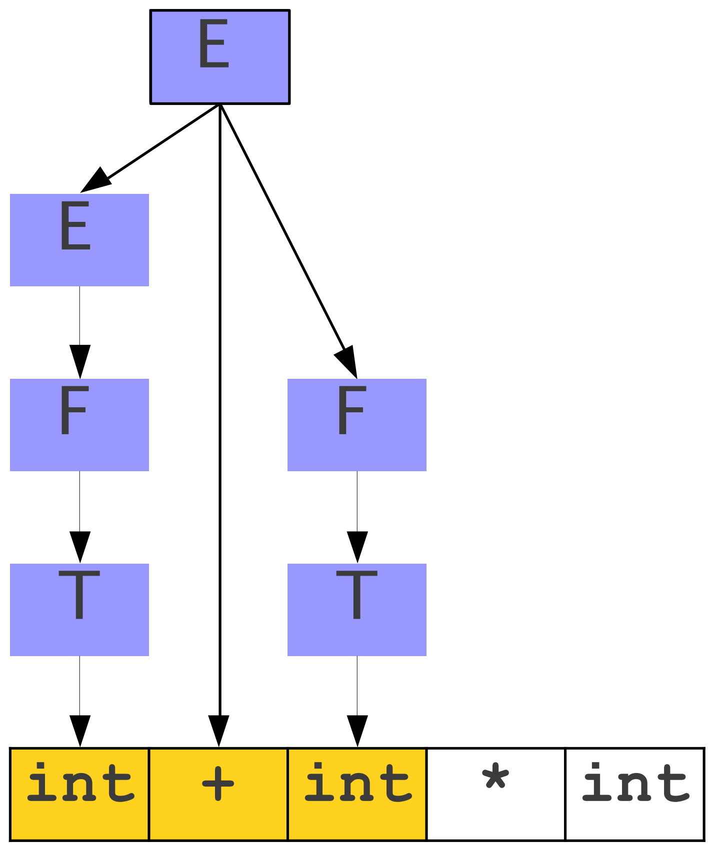
A Detail about Handles

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



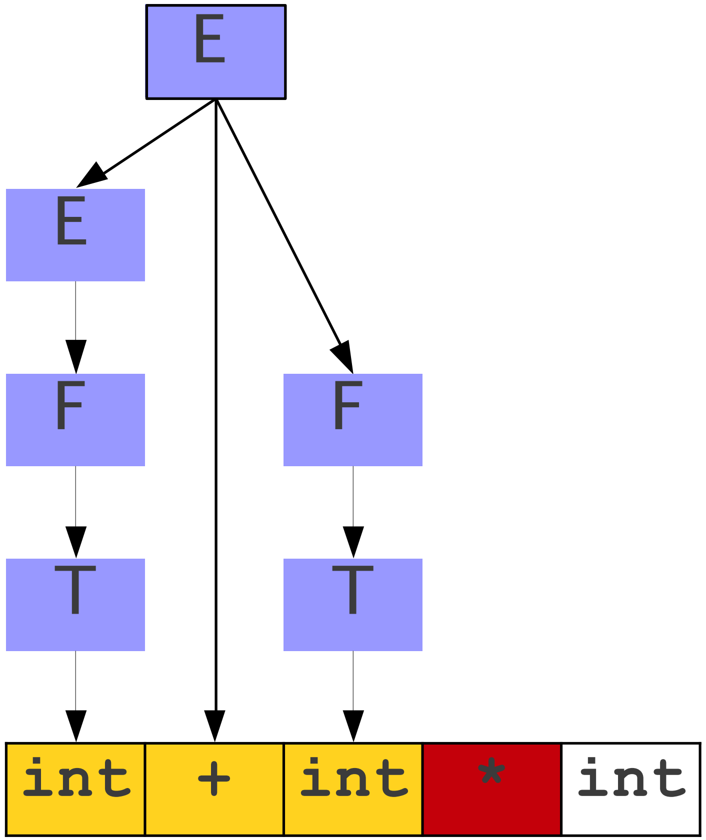
A Detail about Handles

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



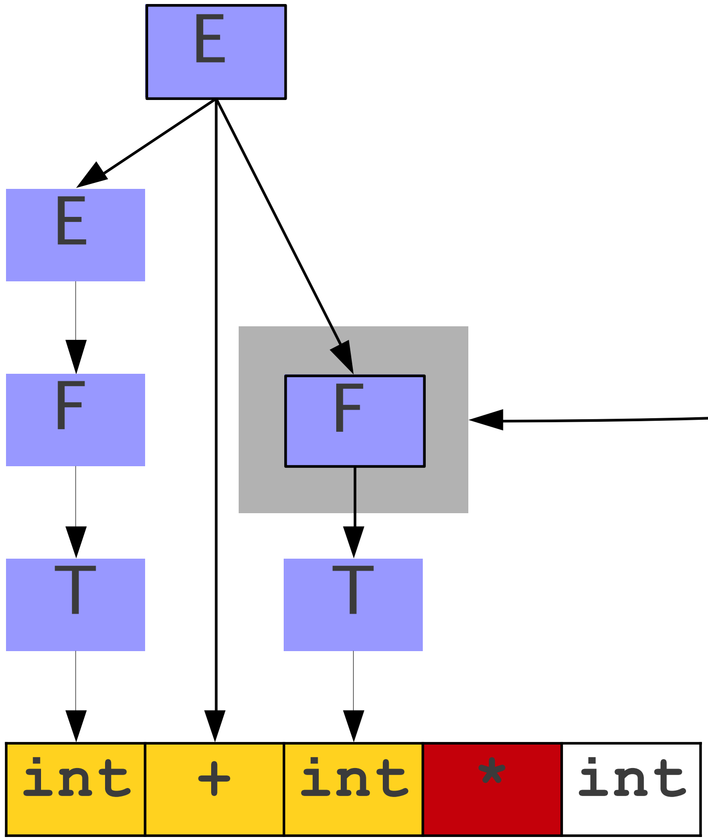
A Detail about Handles

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



A Detail about Handles

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



This reduction wasn't a handle!

The leftmost reduction isn't always the handle.

Finding Handles

- Where do we look for handles?
  – Where in the string might the handle be?
- How do we search for possible handles?
  – Once we know where to search, how do we identify candidate handles?
- How do we recognize handles?
  – Once we've found a candidate handle, how do we check that it really is the handle?

Question One:

Where are handles?

Where are Handles?

- Recall: A left-to-right, bottom-up parse traces a rightmost derivation in reverse.
- Each time we do a reduction, we are reversing a production applied to the *rightmost* nonterminal symbol.
- Suppose that our current sentential form is $\alpha\gamma\omega$, where γ is the handle and $\mathbf{A} \rightarrow \gamma$ is a production rule.
- After reducing γ back to $\mathbf{A}$, we have the string $\alpha\mathbf{A}\omega$. Thus ω must consist purely of terminals, since

  otherwise the reduction we just did was not for the rightmost terminal.

Why This Matters

- Suppose we want to parse the string γ. We will break γ into two parts, α and ω,
- Where
  - α consists of both terminals and nonterminals, and
  - ω consists purely of terminals.
- Our search for handles will concentrate purely in α.
- As necessary, we will start moving terminals from ω over into α.

Shift/Reduce Parsing

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

A Sample Shift/Reduce Parse

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

int	+	int	*	int	+	int

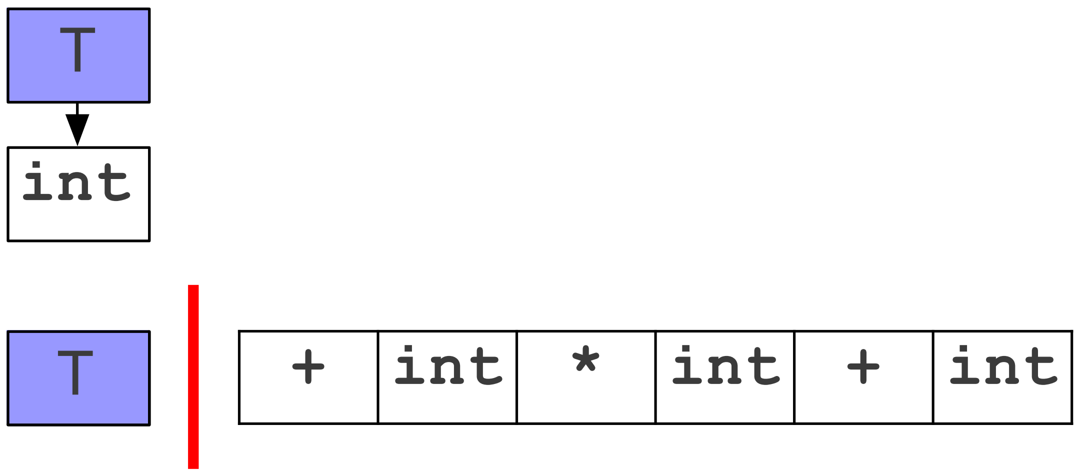
A Sample Shift/Reduce Parse

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

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

A Sample Shift/Reduce Parse

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



A Sample Shift/Reduce Parse

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

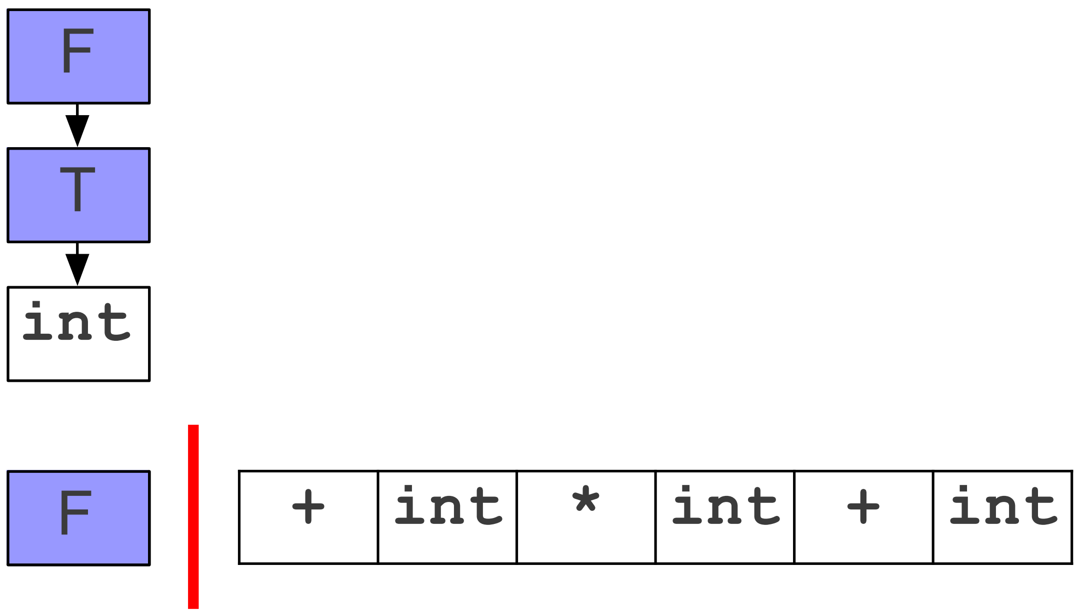
$\mathbf{E} \rightarrow \mathbf{E} + \mathbf{F}$

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

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

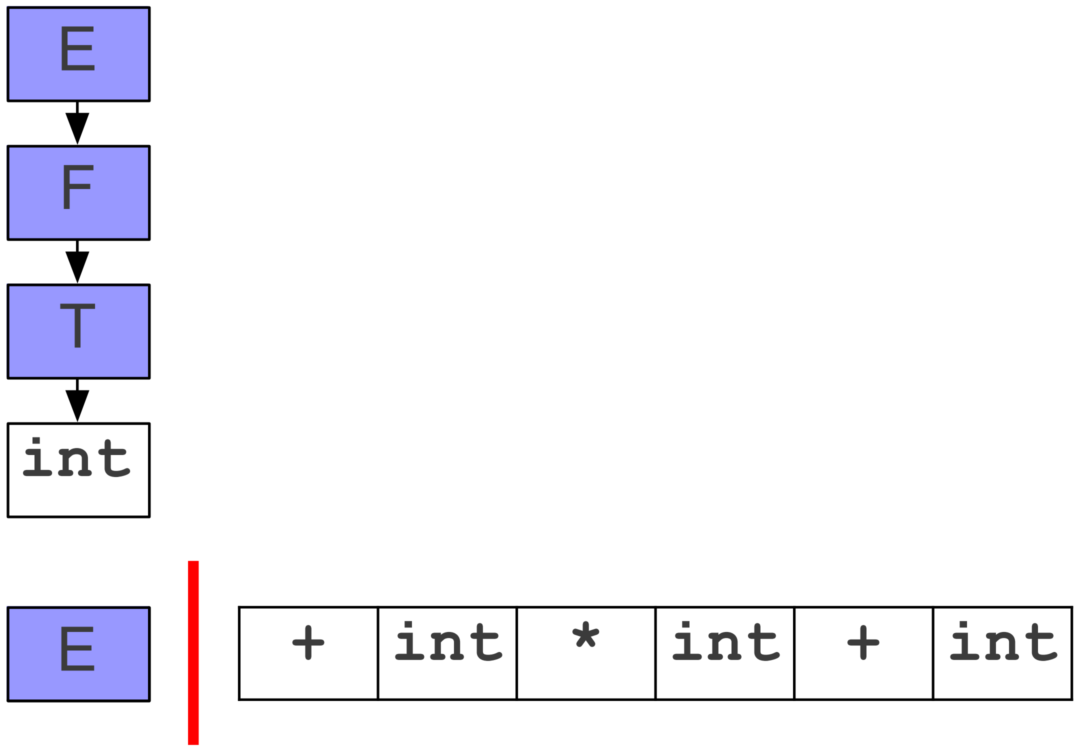
$\mathbf{T} \rightarrow$ `int`

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



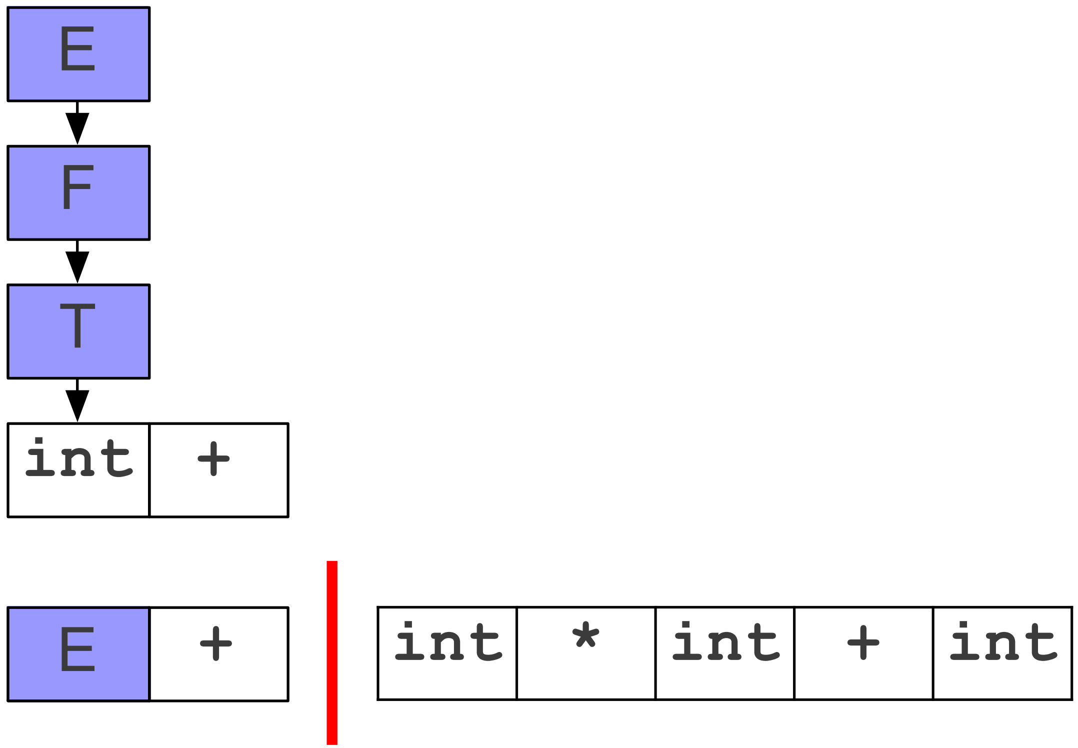
A Sample Shift/Reduce Parse

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



A Sample Shift/Reduce Parse

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



A Sample Shift/Reduce Parse

$E \rightarrow F$

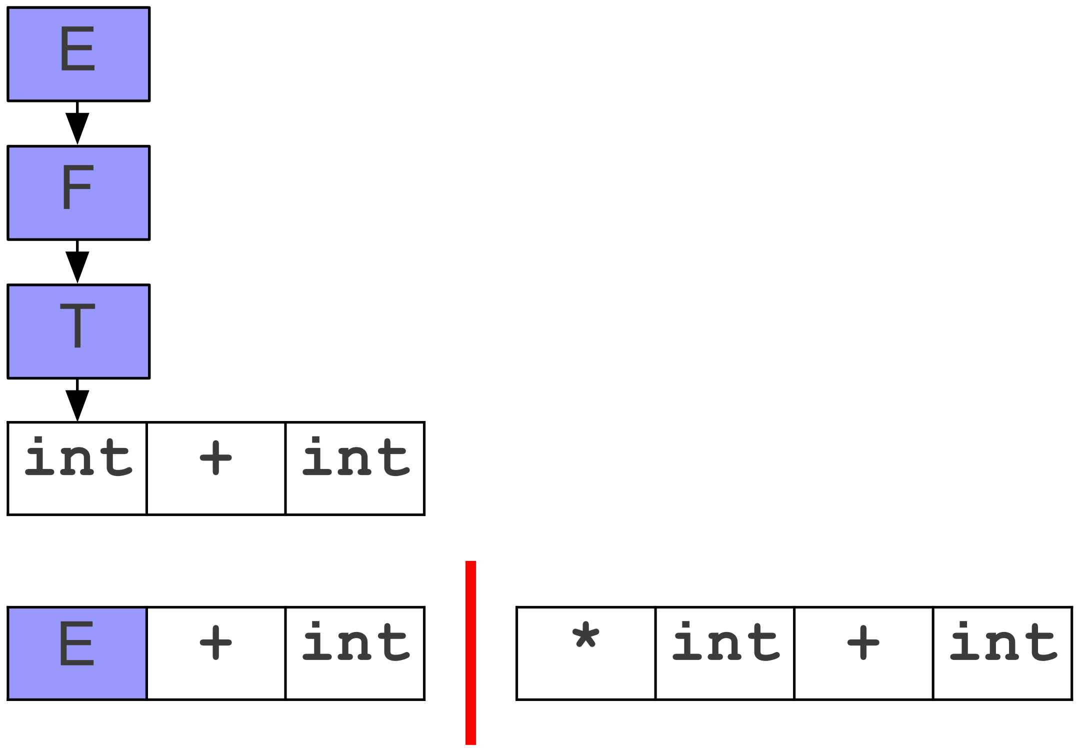
$E \rightarrow E + F$

$F \rightarrow F * T$

$F \rightarrow T$

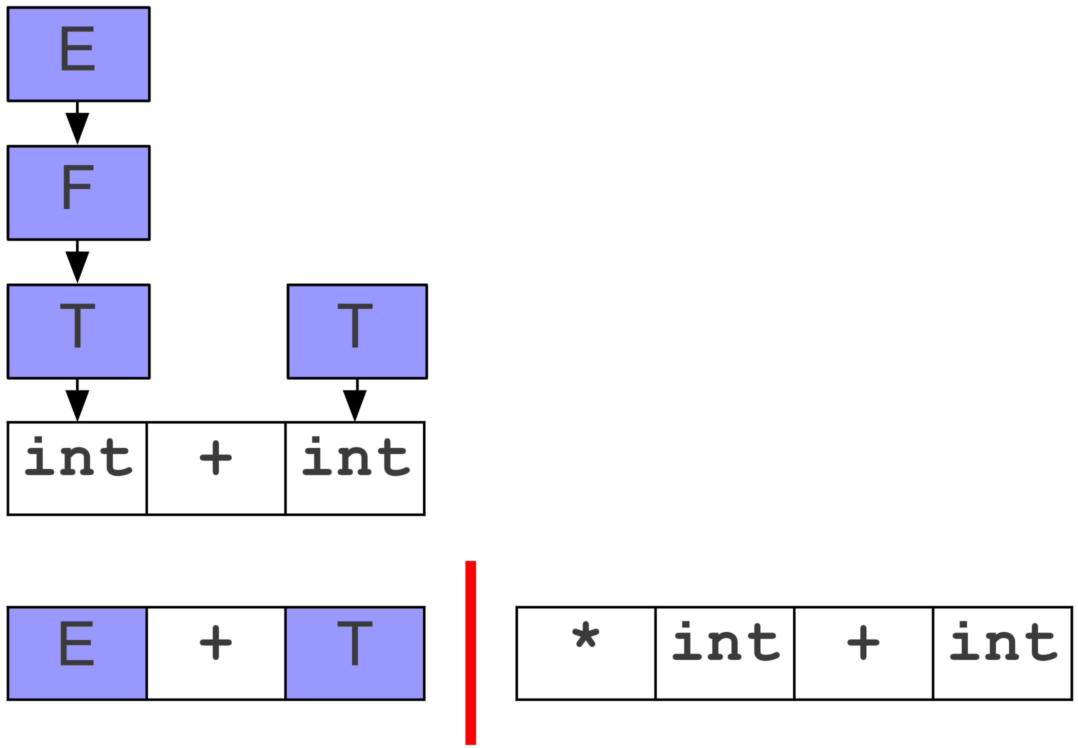
$T \rightarrow \texttt{int}$

$T \rightarrow (E)$



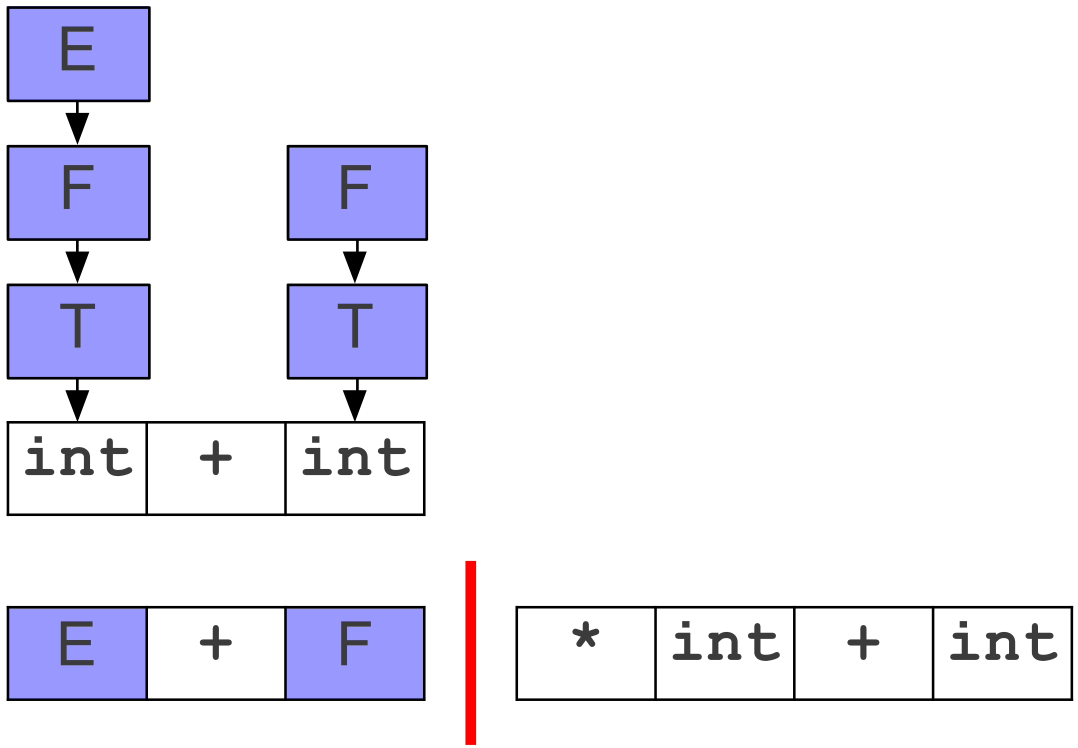
A Sample Shift/Reduce Parse

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



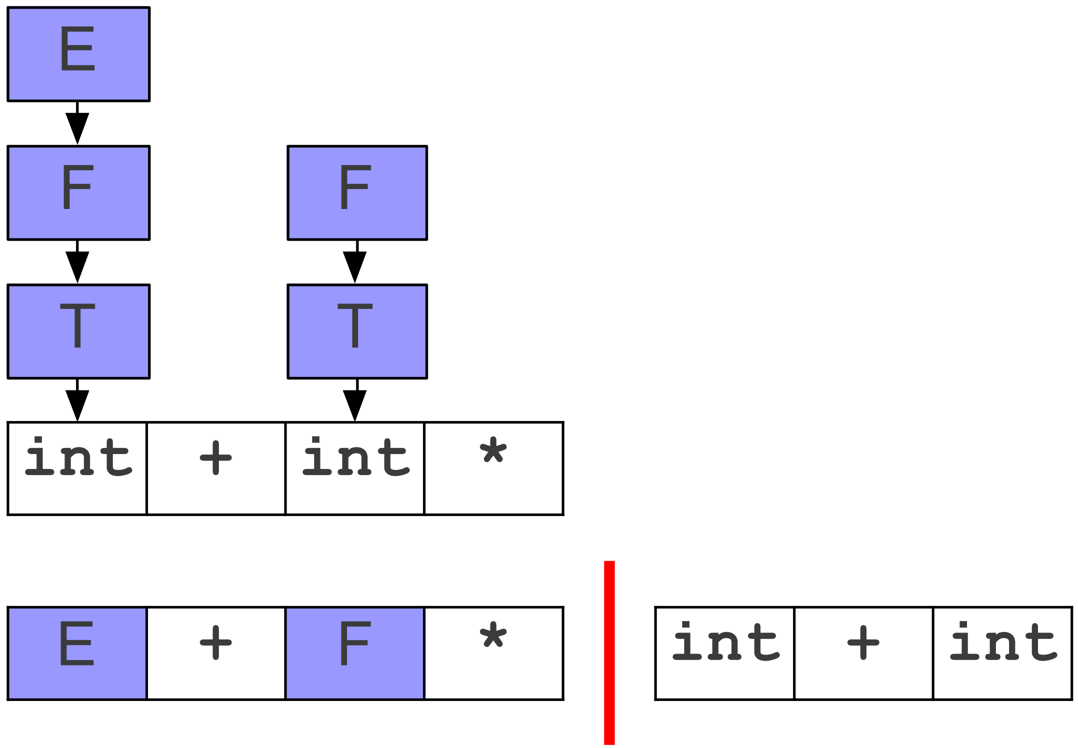
A Sample Shift/Reduce Parse

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



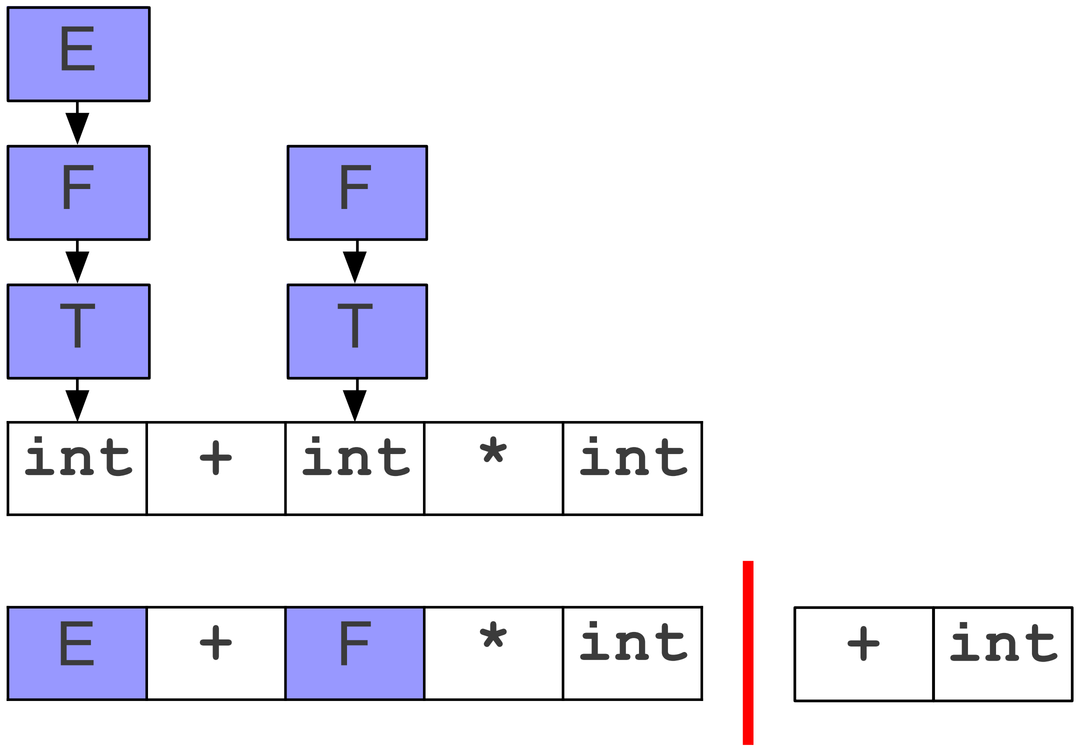
A Sample Shift/Reduce Parse

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



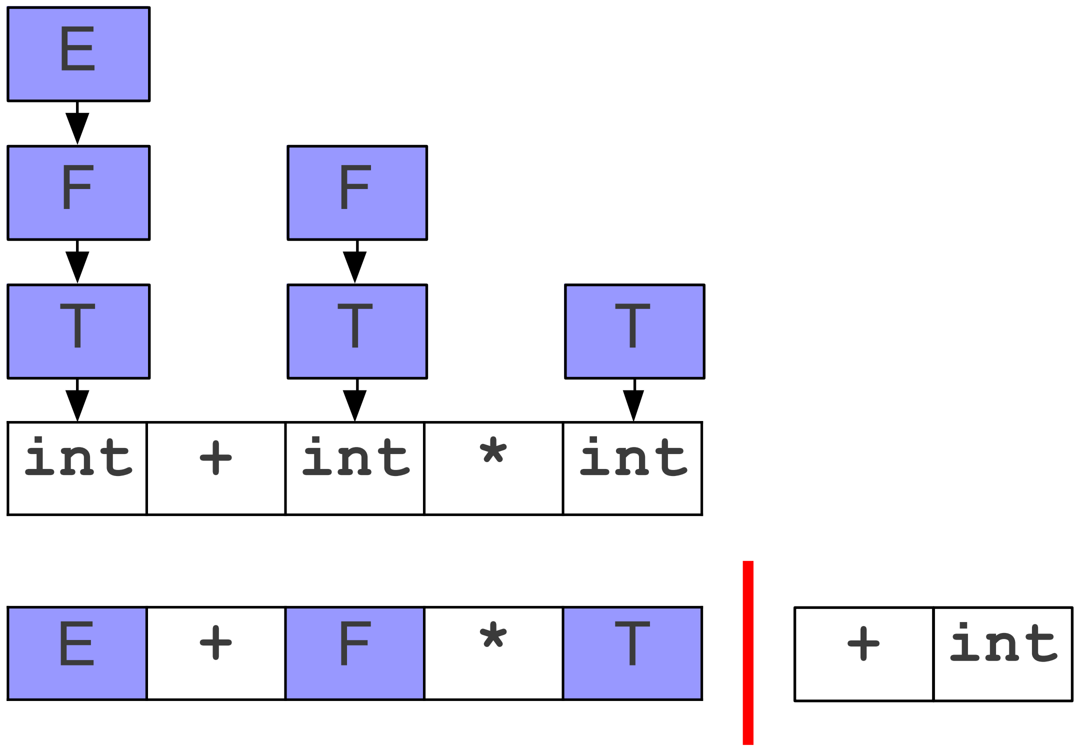
A Sample Shift/Reduce Parse

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



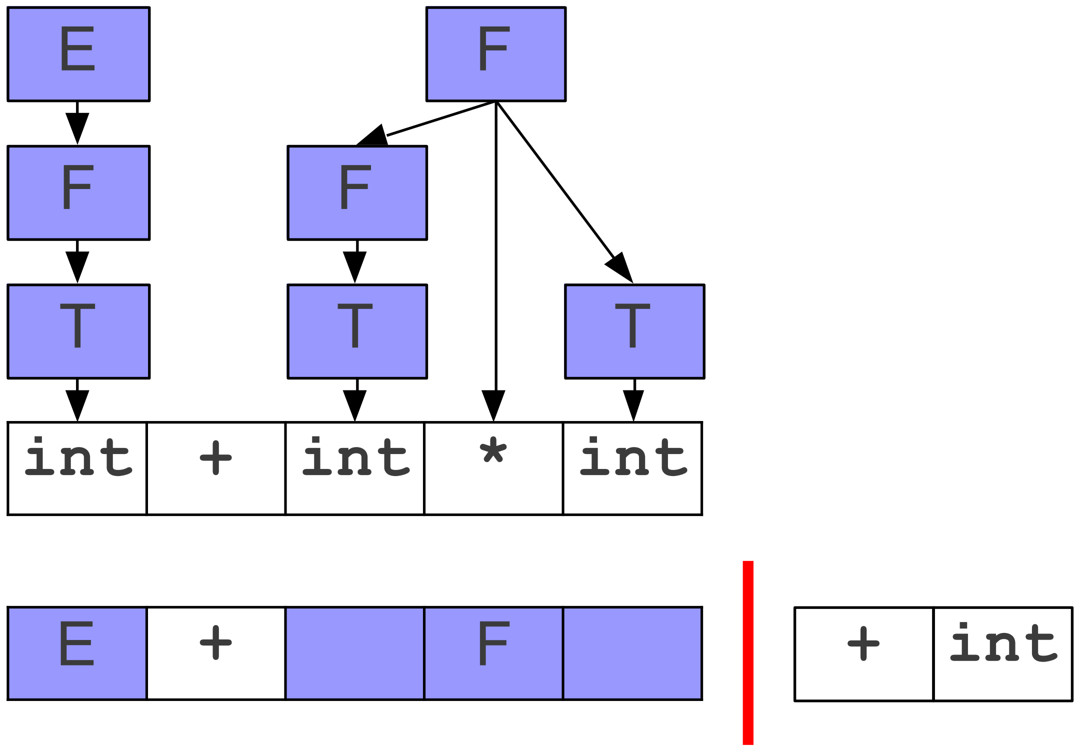
A Sample Shift/Reduce Parse

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



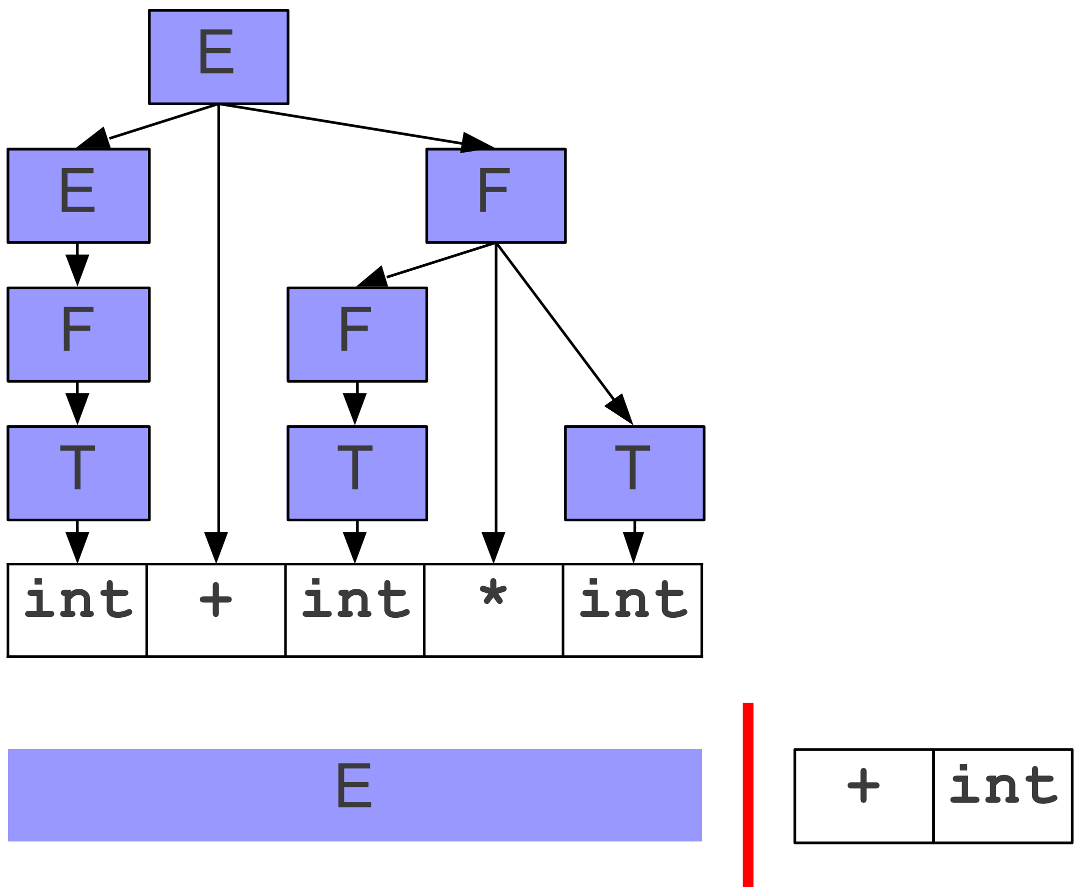
A Sample Shift/Reduce Parse

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



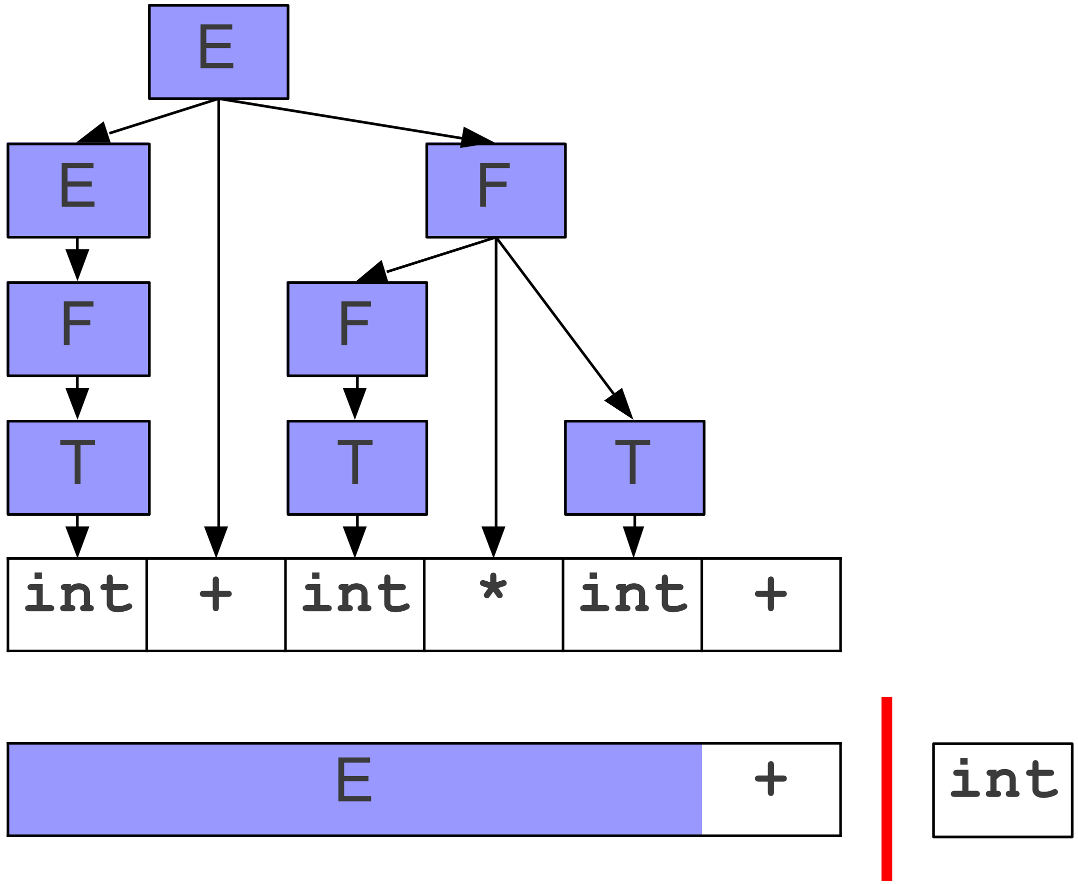
A Sample Shift/Reduce Parse

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



A Sample Shift/Reduce Parse

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



A Sample Shift/Reduce Parse

$E \rightarrow F$

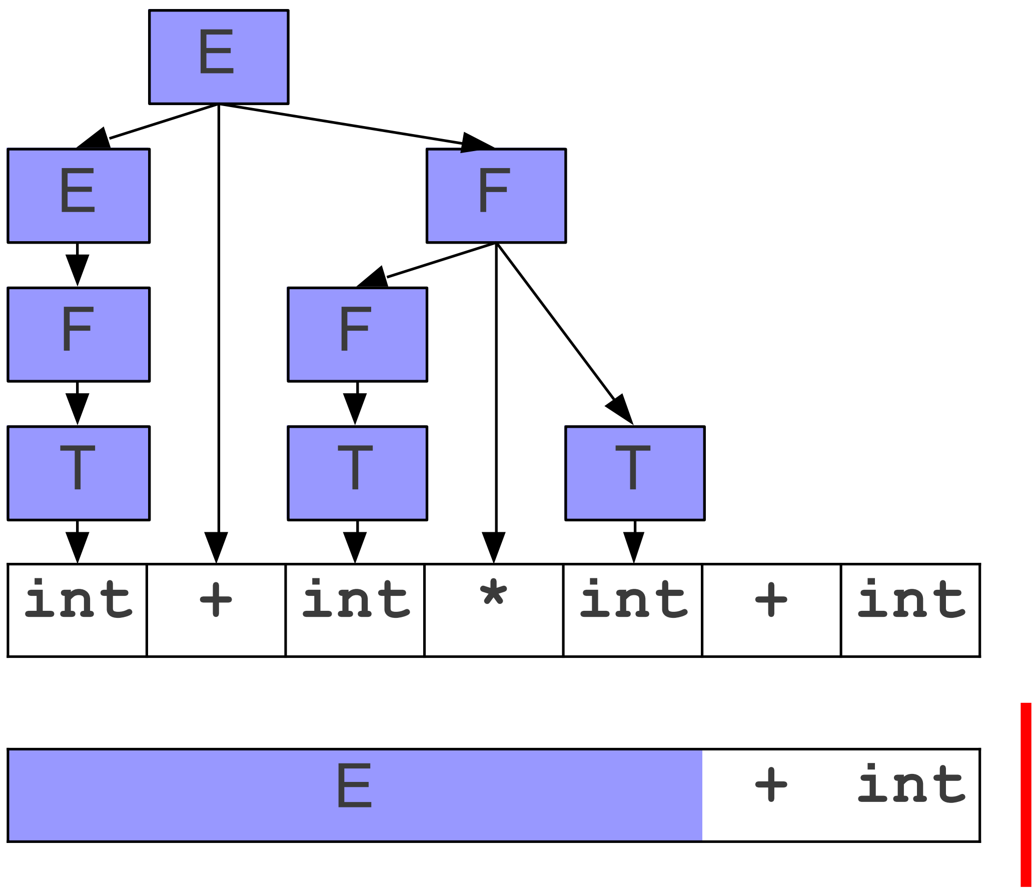
$E \rightarrow E + F$

$F \rightarrow F * T$

$F \rightarrow T$

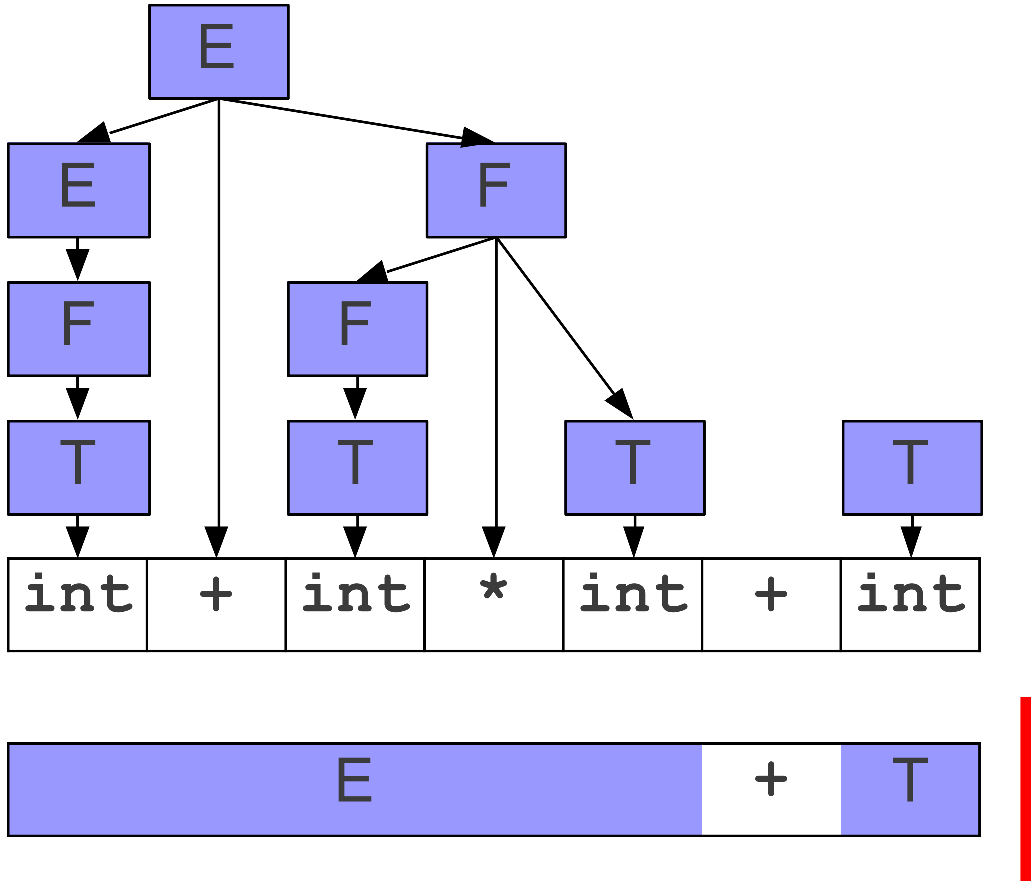
$T \rightarrow$ int

$T \rightarrow (E)$



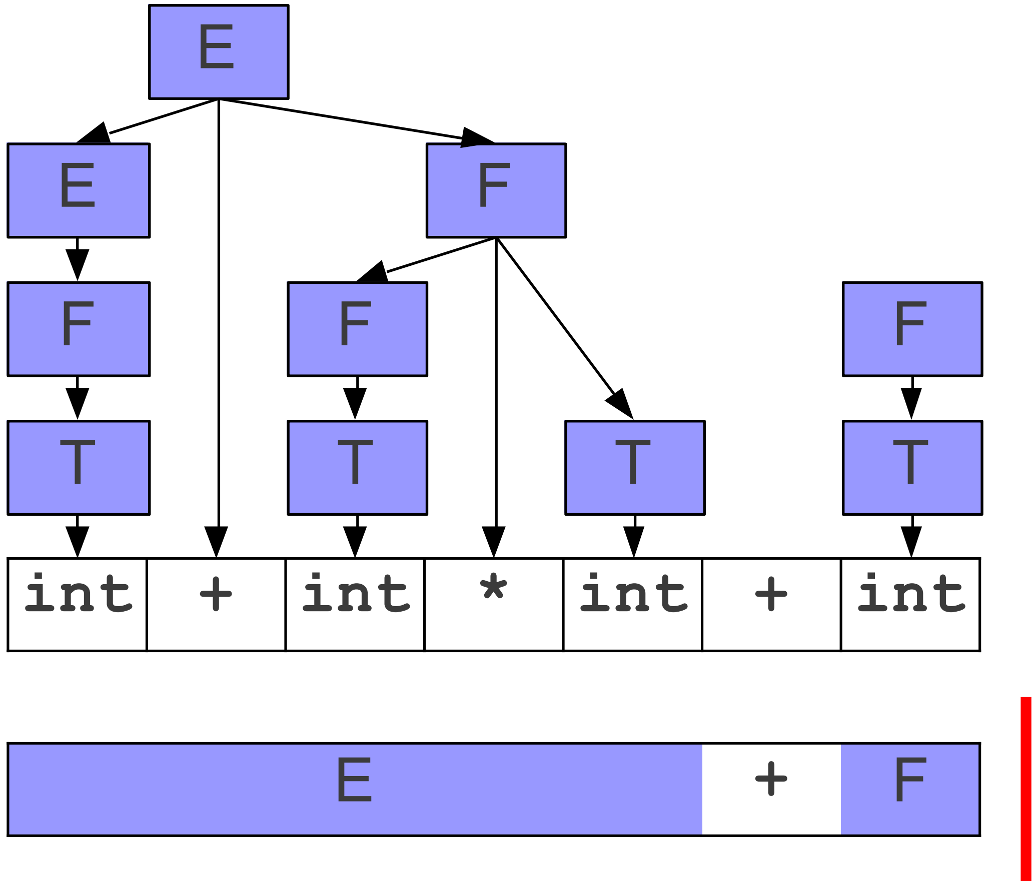
A Sample Shift/Reduce Parse

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



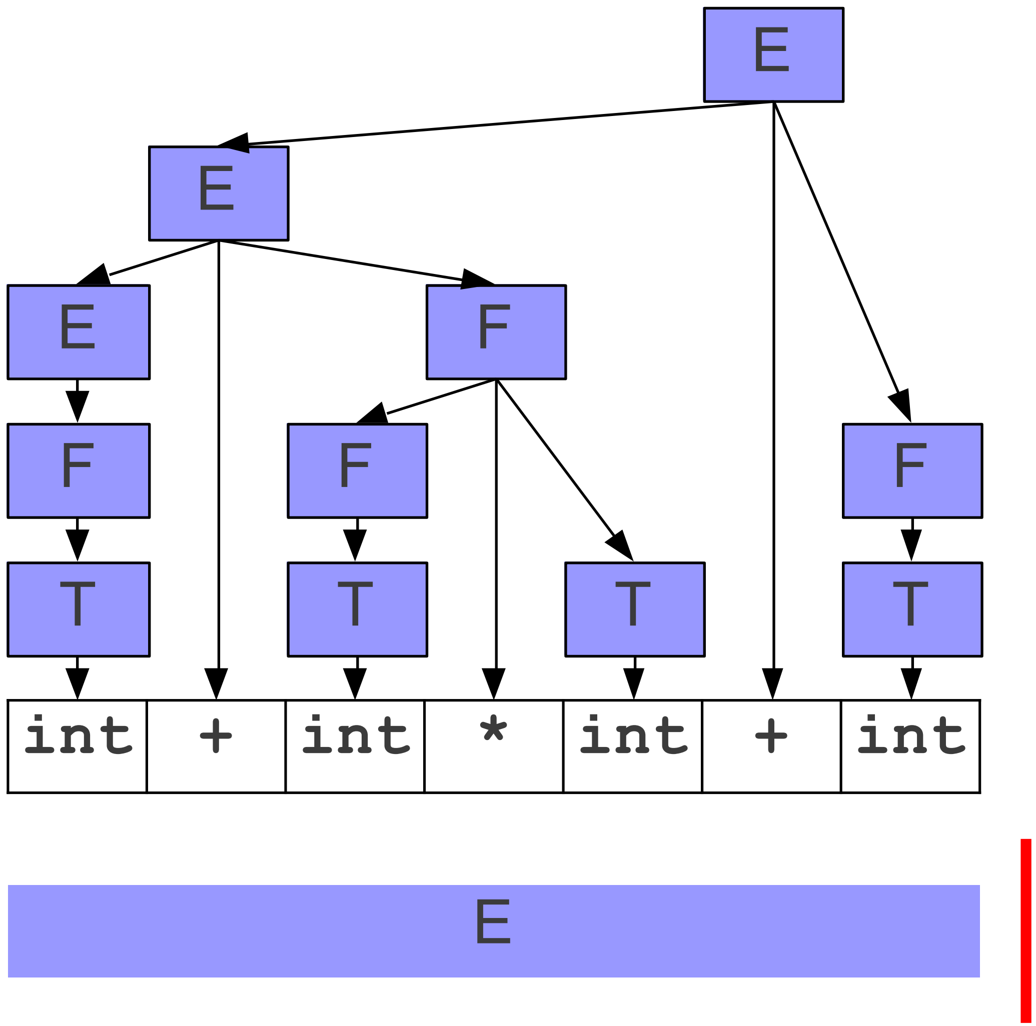
A Sample Shift/Reduce Parse

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)**



An Important Observation

- All of the reductions we applied were to the far right end of the left area.
- This is not a coincidence; all reductions are always applied all the way to the end of the left area.
- Inductive proof sketch:
  - After no reduces, the first reduction can be done at the right end of the left area.
  - After at least one reduce, the very right of the left area is a nonterminal. This nonterminal must be part of the next reduction, since we're tracing a rightmost derivation backwards.

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.

Simplifying our Terminology

- All activity in a shift/reduce parser is at the far right end of the left area.
- Idea: Represent the left area as a stack.
- Shift: Push the next terminal onto the
- stack.
- Reduce: Pop some number of symbols from the stack, then push the appropriate nonterminal.

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?

Finding Handles

- Where do we look for handles?
  - **At the top of the stack.**
- How do we search for possible handles?
  - Once we know where to search, how do we identify candidate handles?
- How do we recognize handles?
  - Once we've found a candidate handle, how do we check that it really is the handle?

Question Two:

How do we search for handles?

Searching for Handles

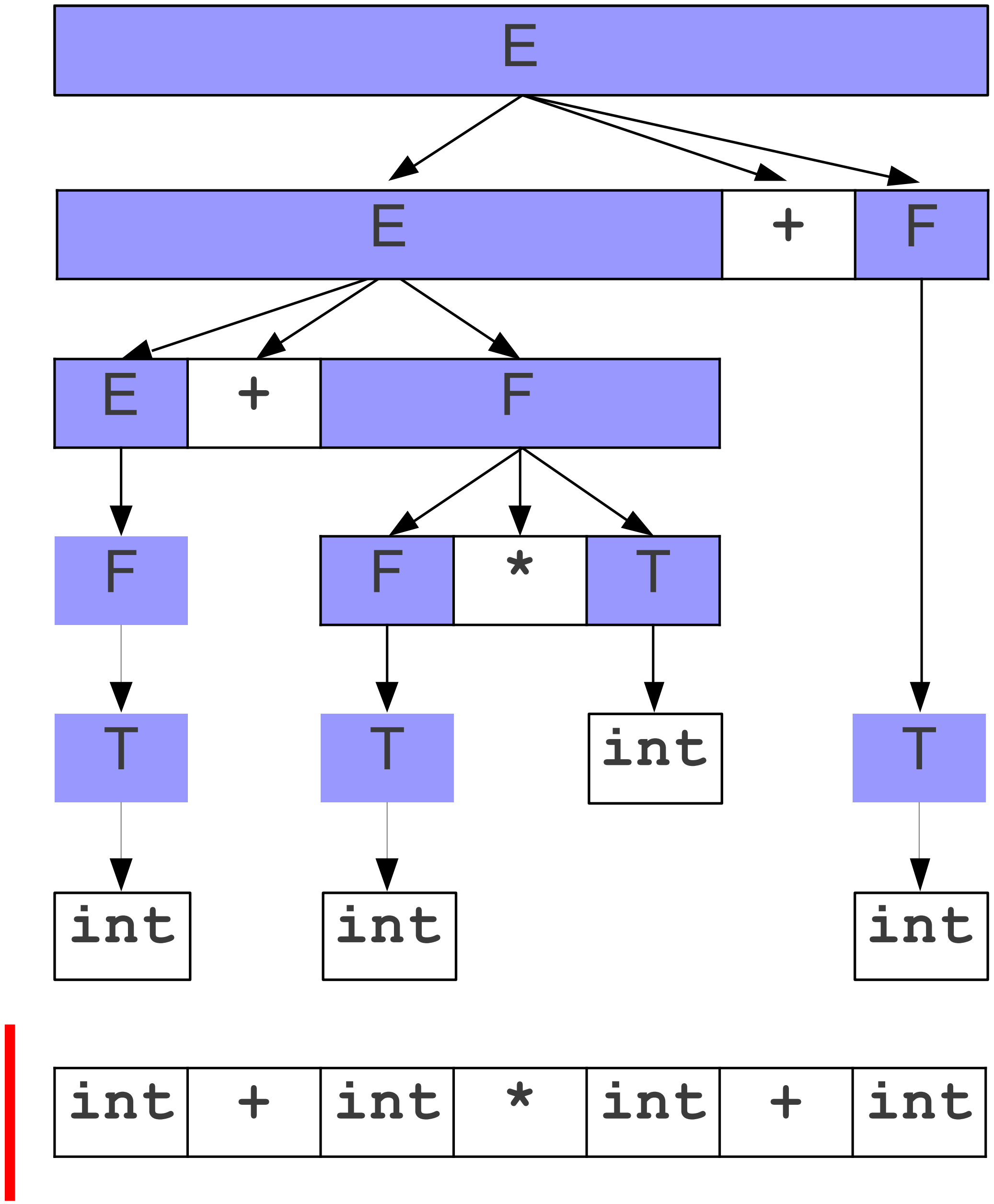
- When using a shift/reduce parser, we must decide whether to shift or reduce at each point.
- We only want to reduce when we know we have a handle.
- **Question:** How can we tell that we might be looking at a handle?

Exploring the Left Side

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

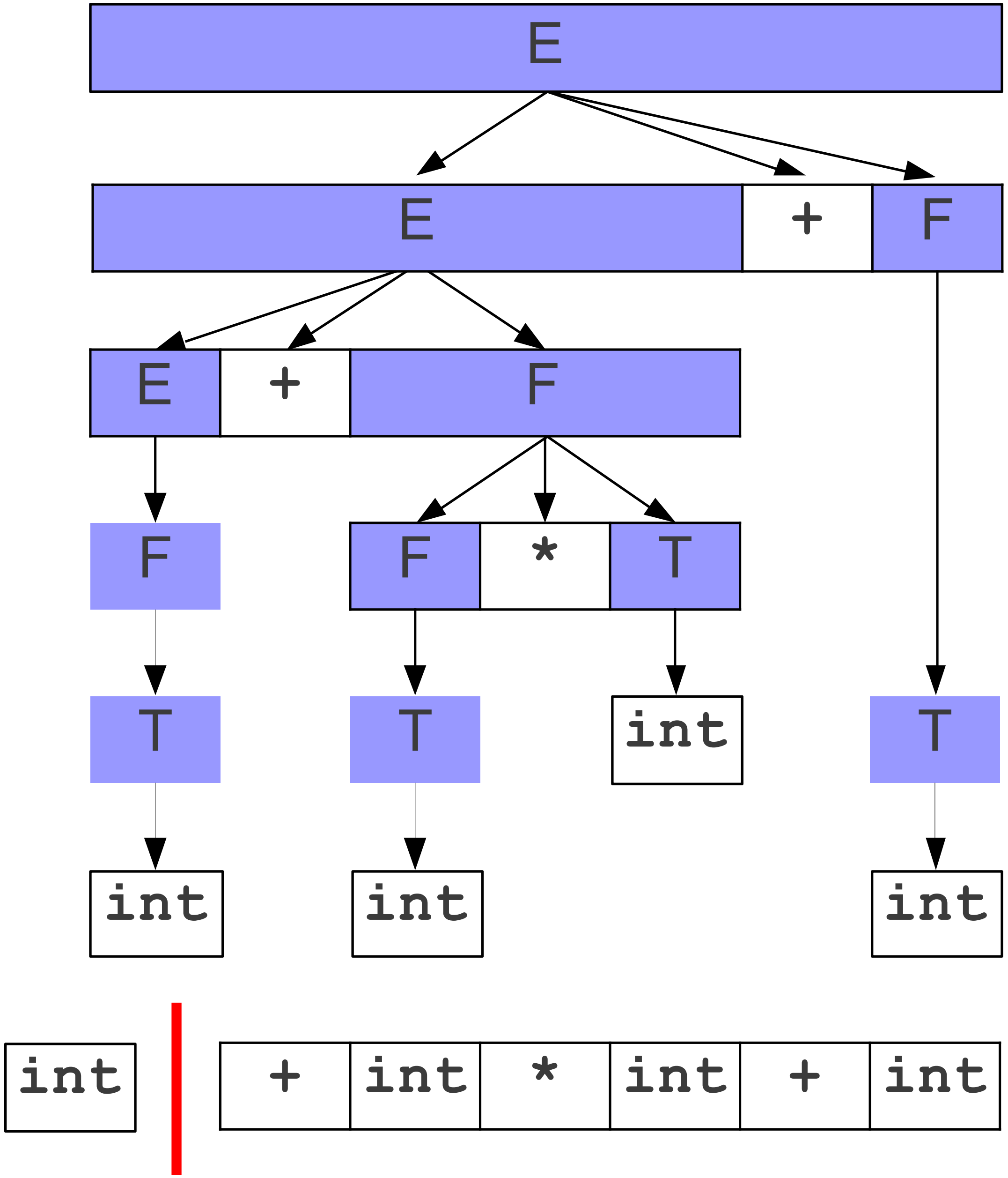
Another Look at Handles

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



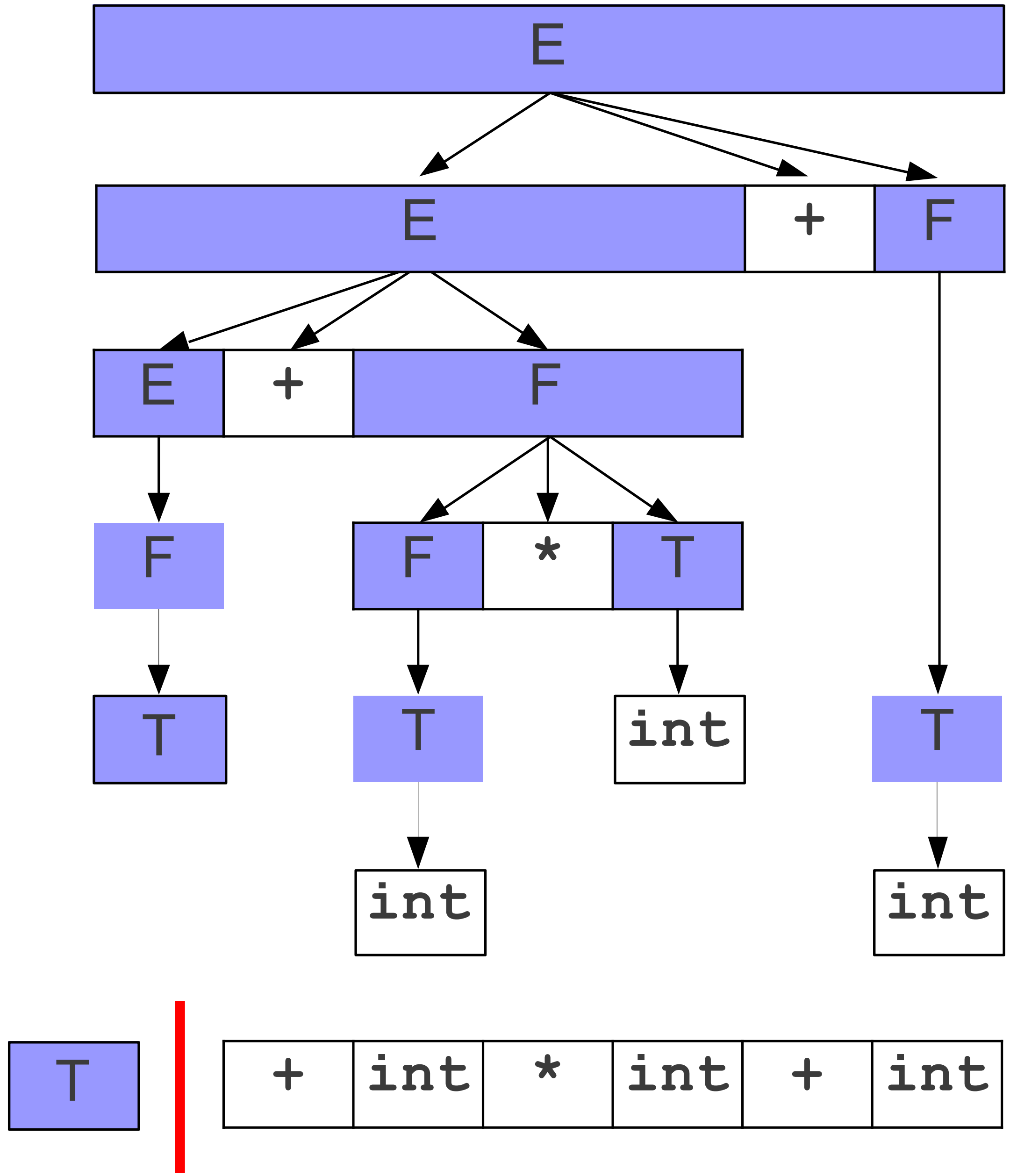
Another Look at Handles

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



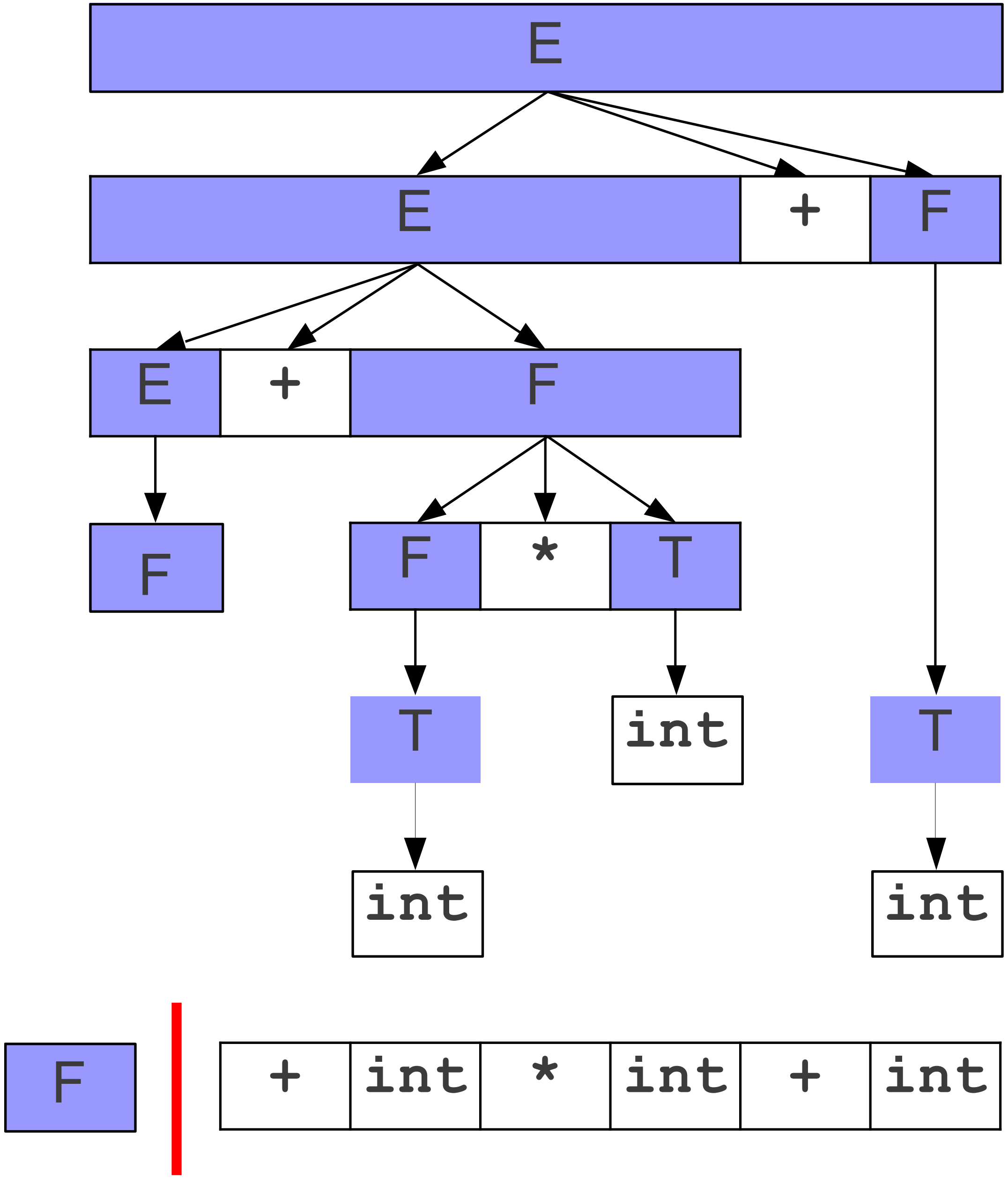
Another Look at Handles

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



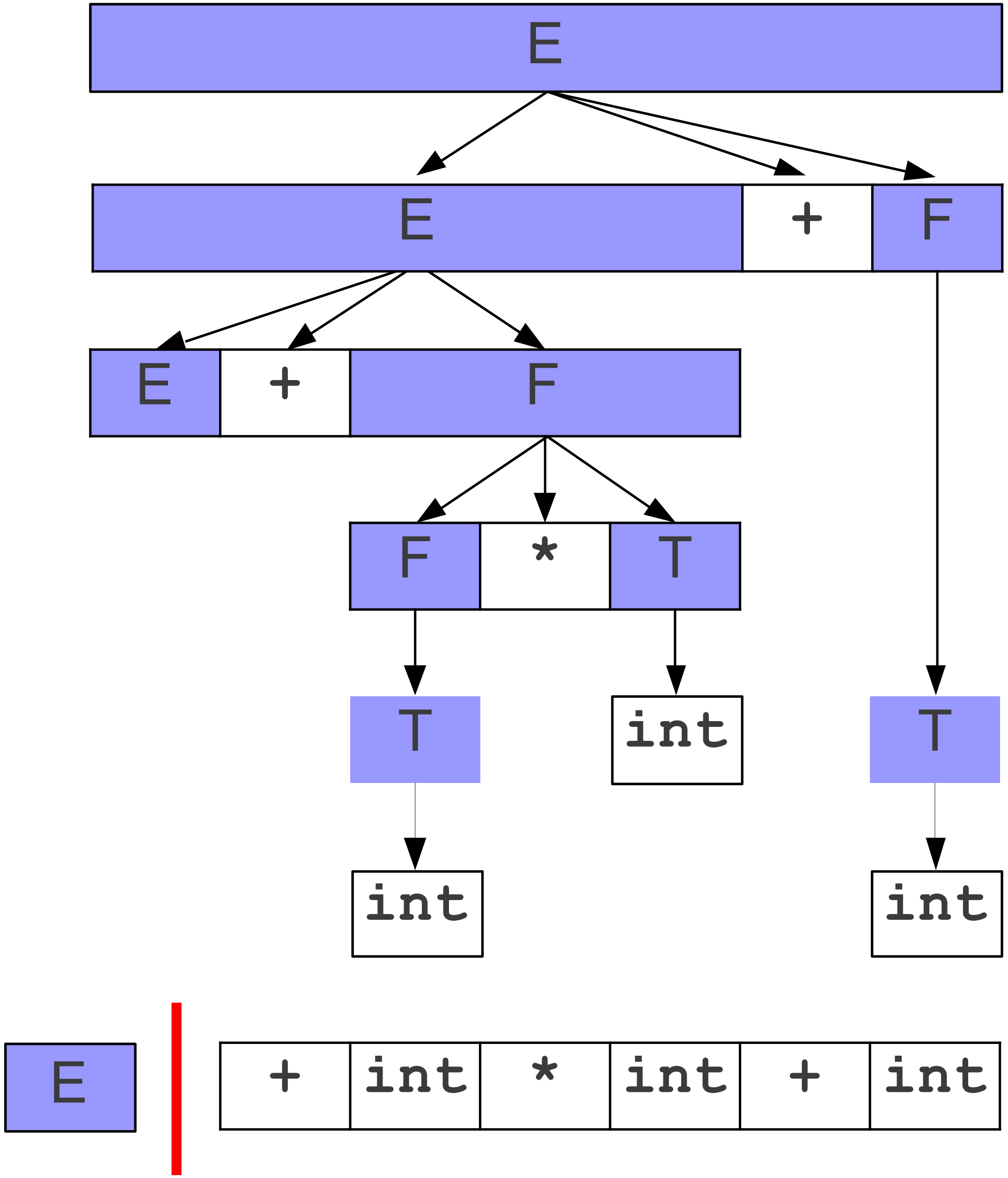
Another Look at Handles

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



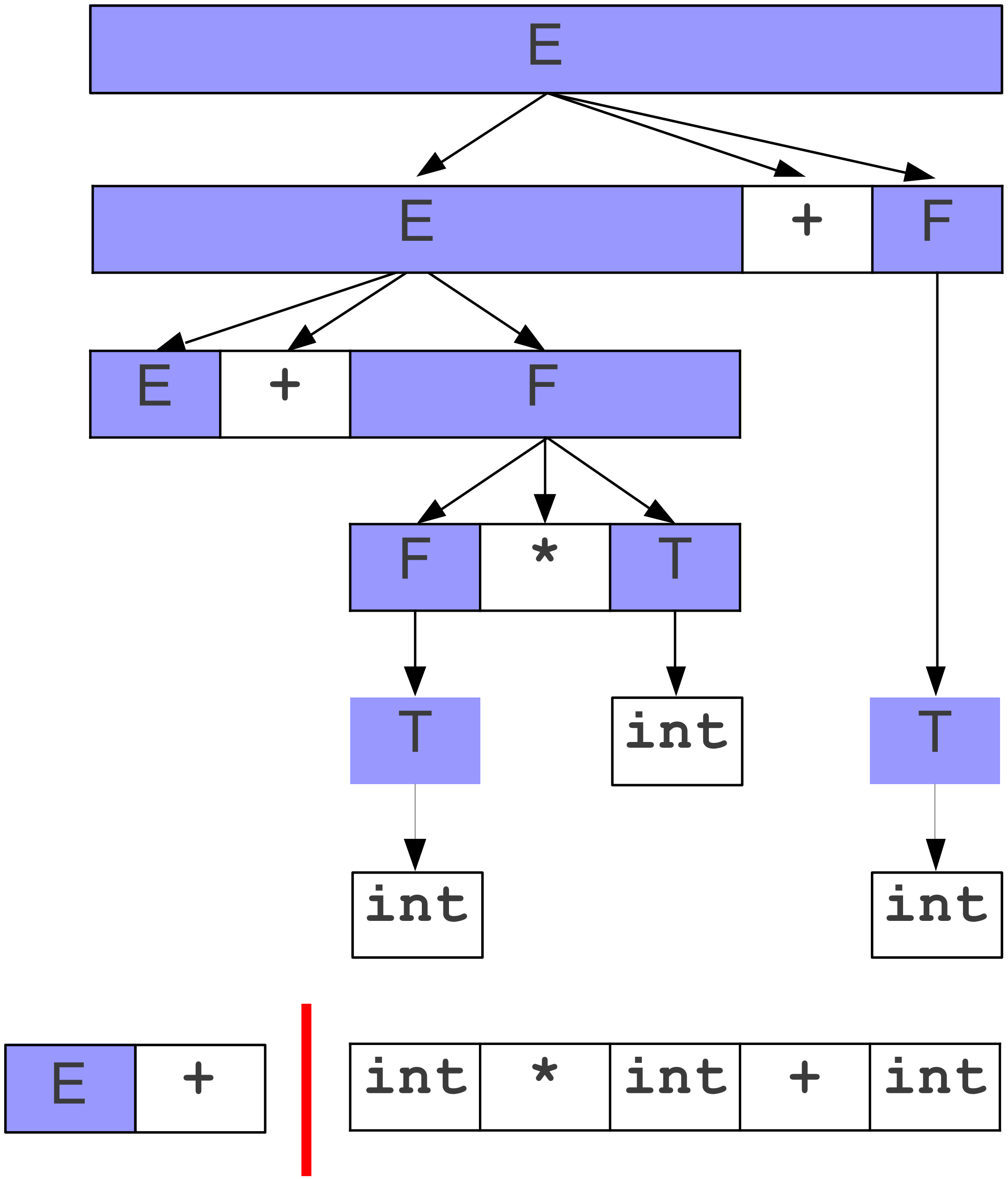
Another Look at Handles

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



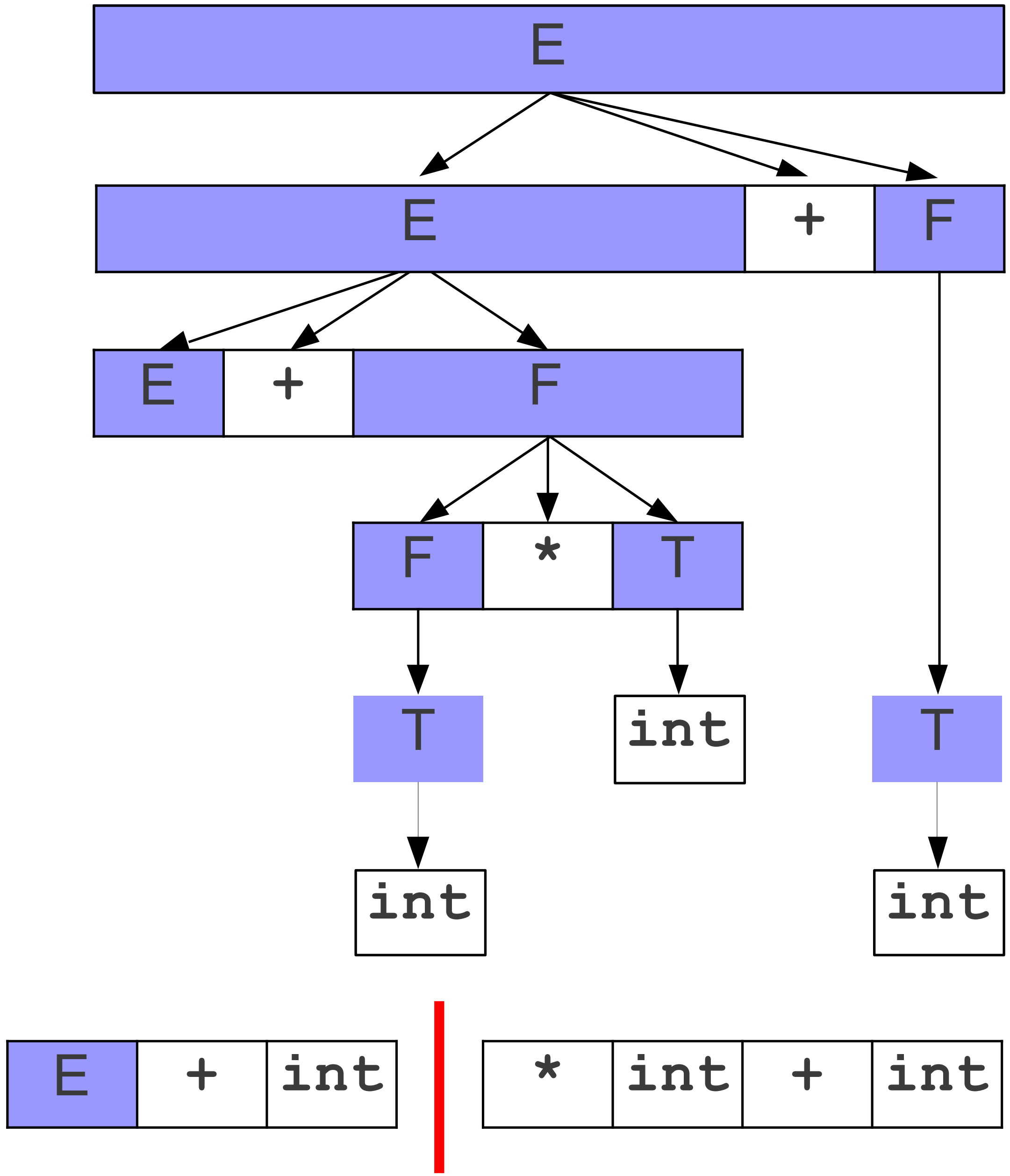
Another Look at Handles

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



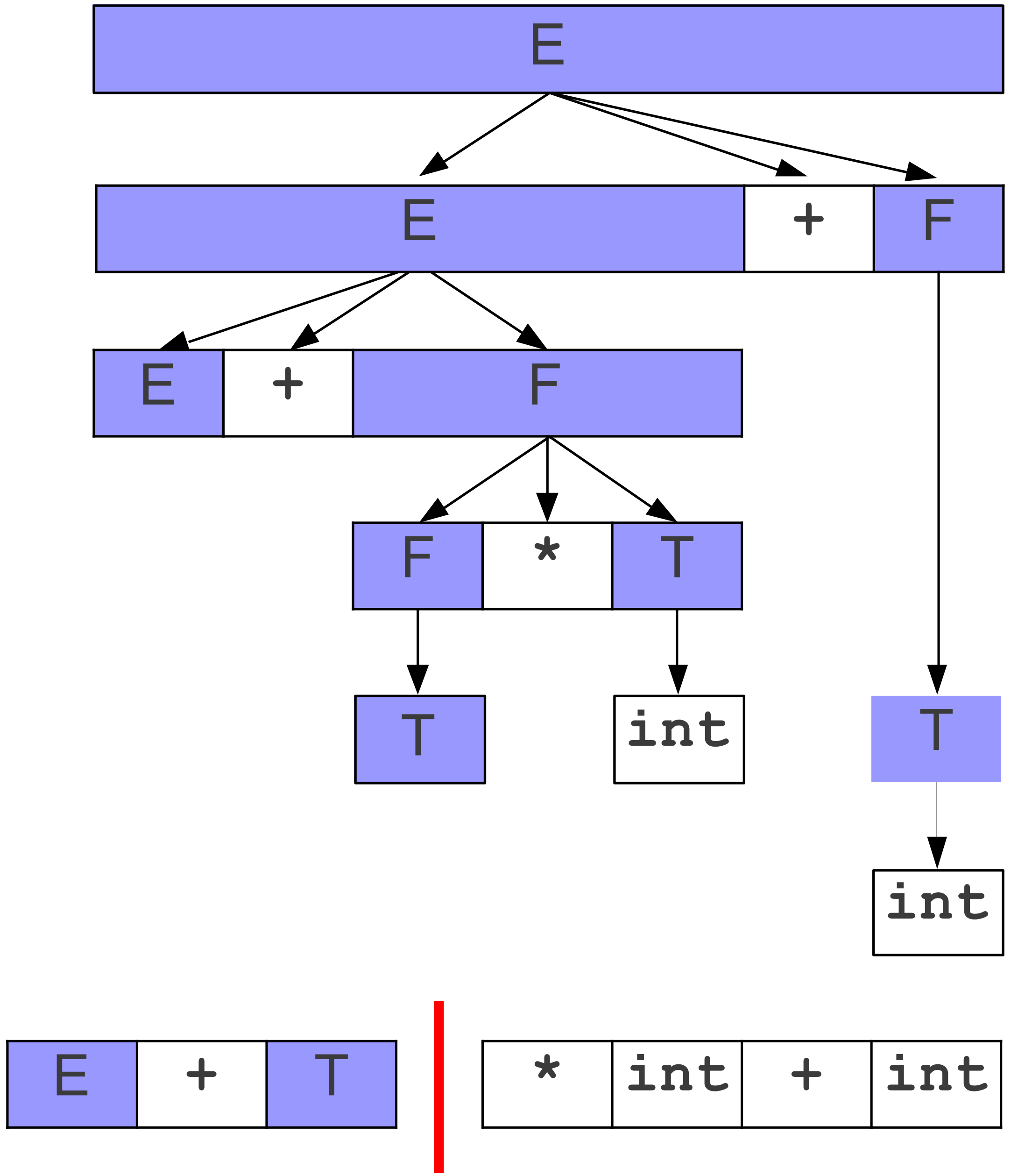
Another Look at Handles

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



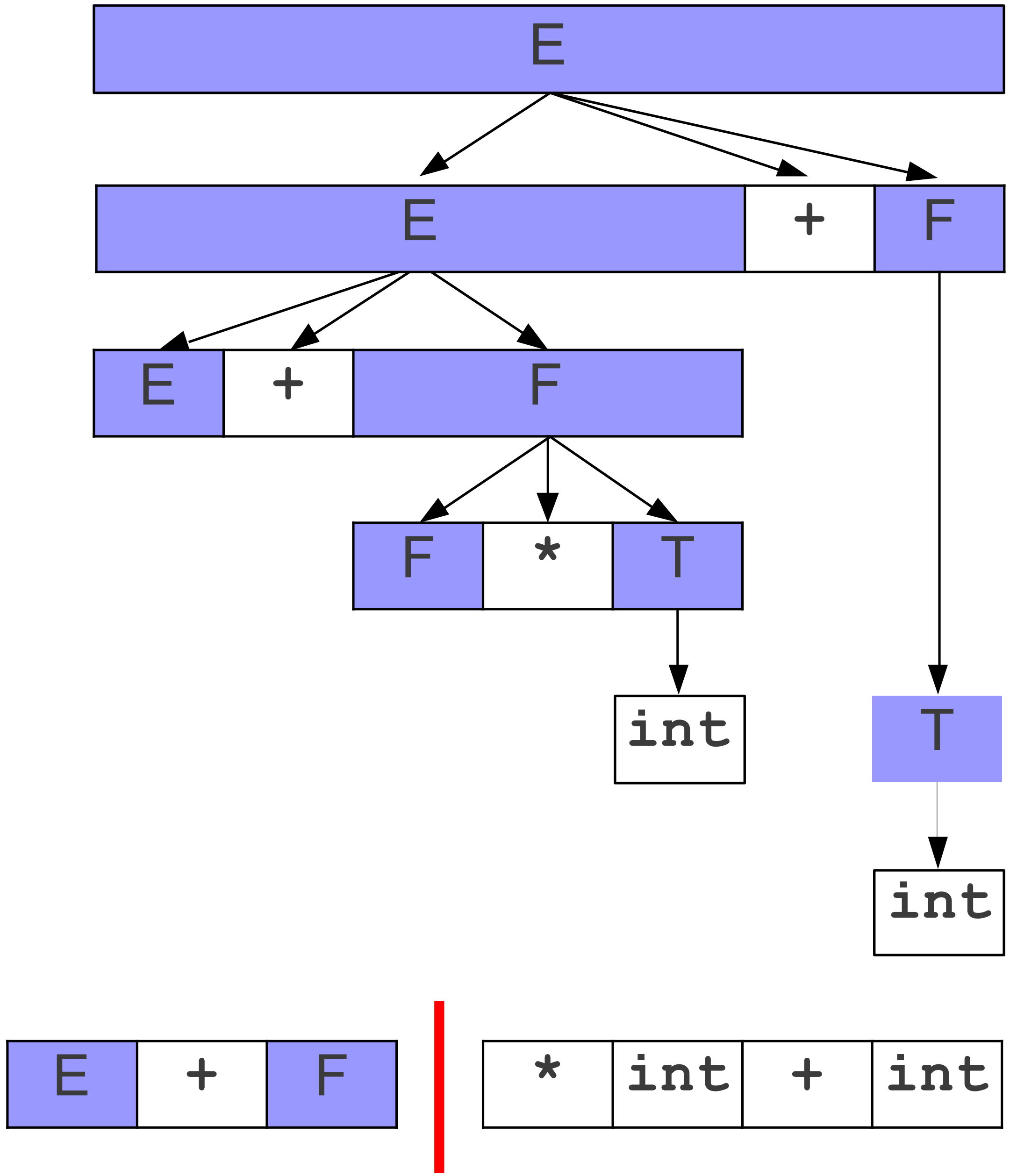
Another Look at Handles

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



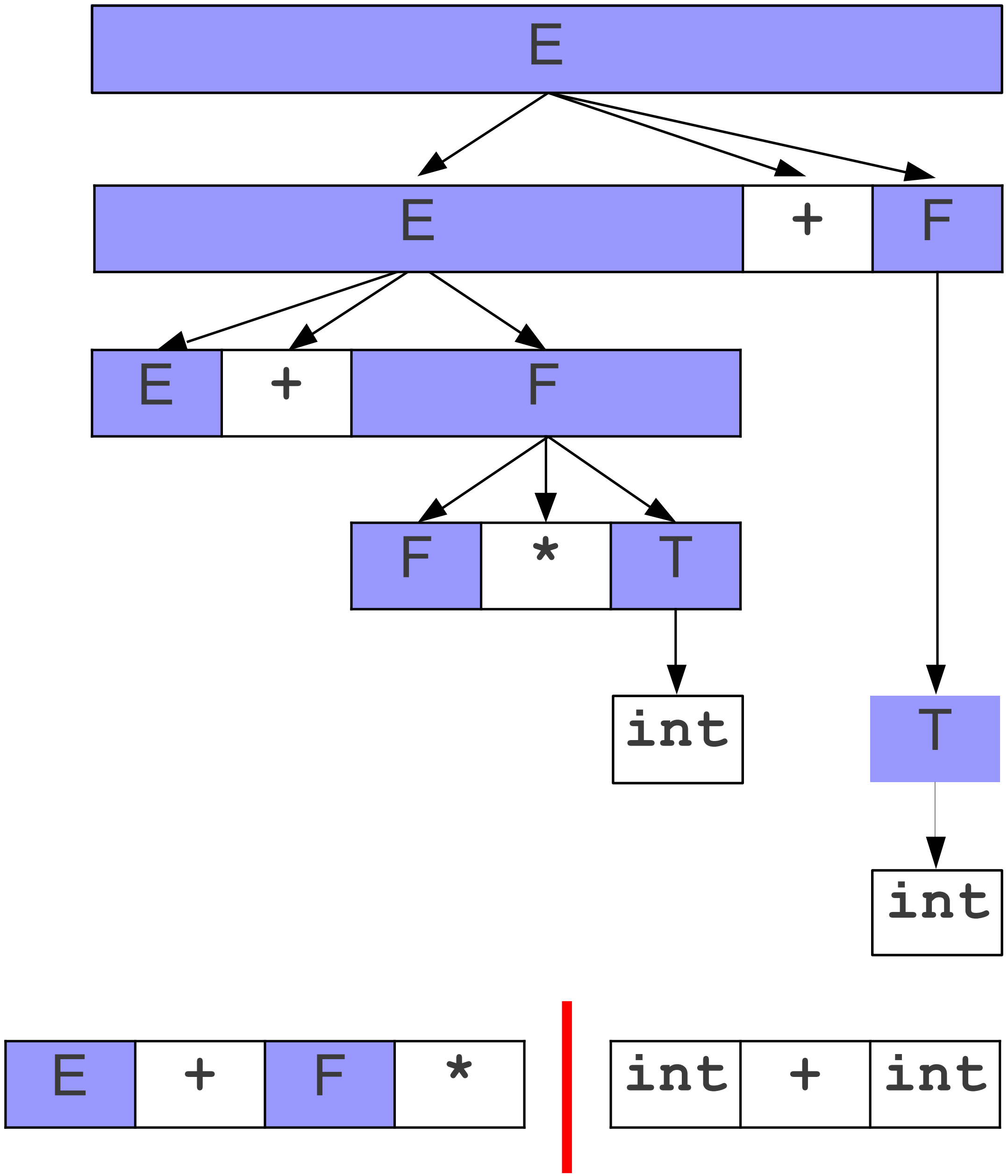
Another Look at Handles

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



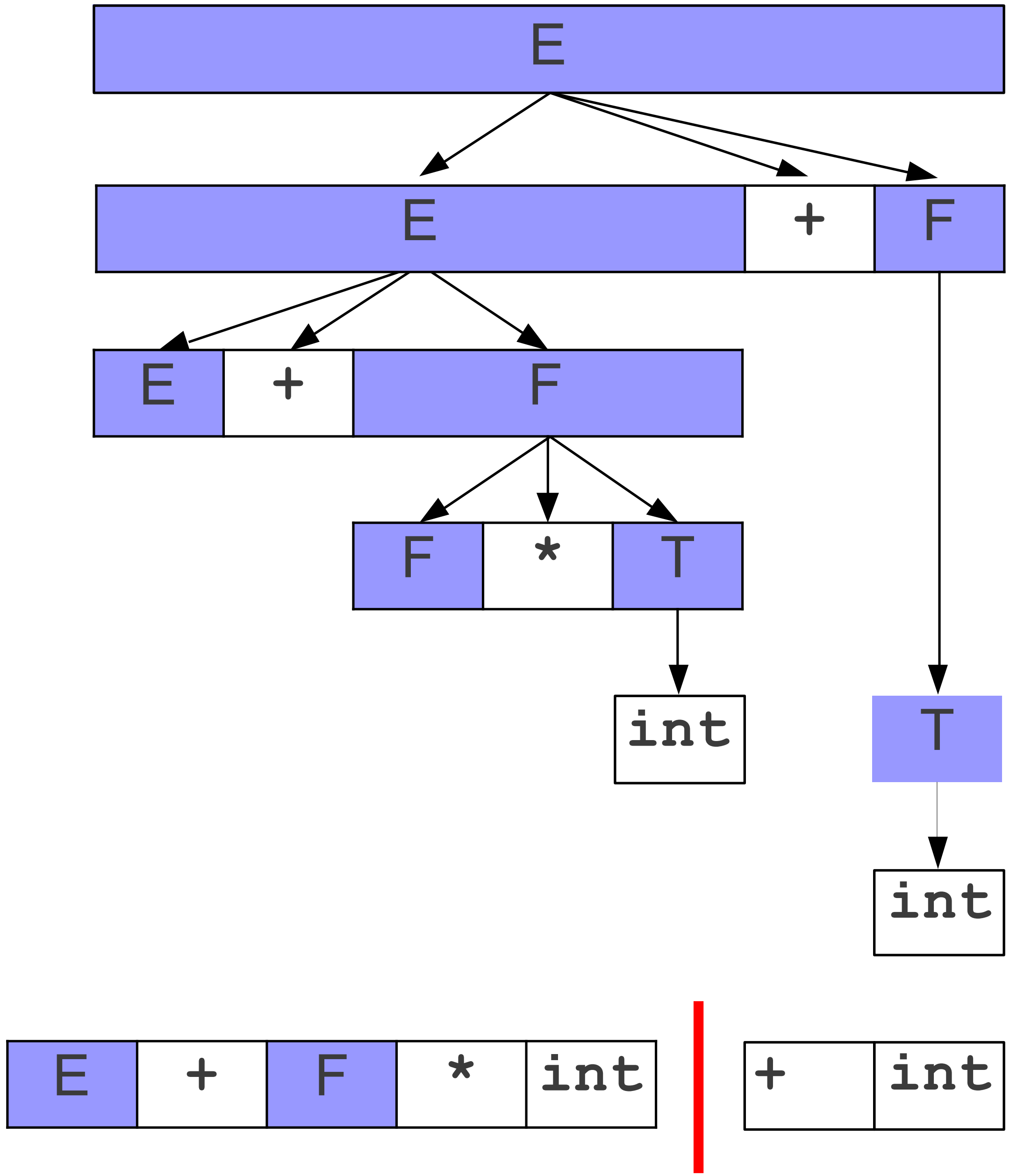
Another Look at Handles

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



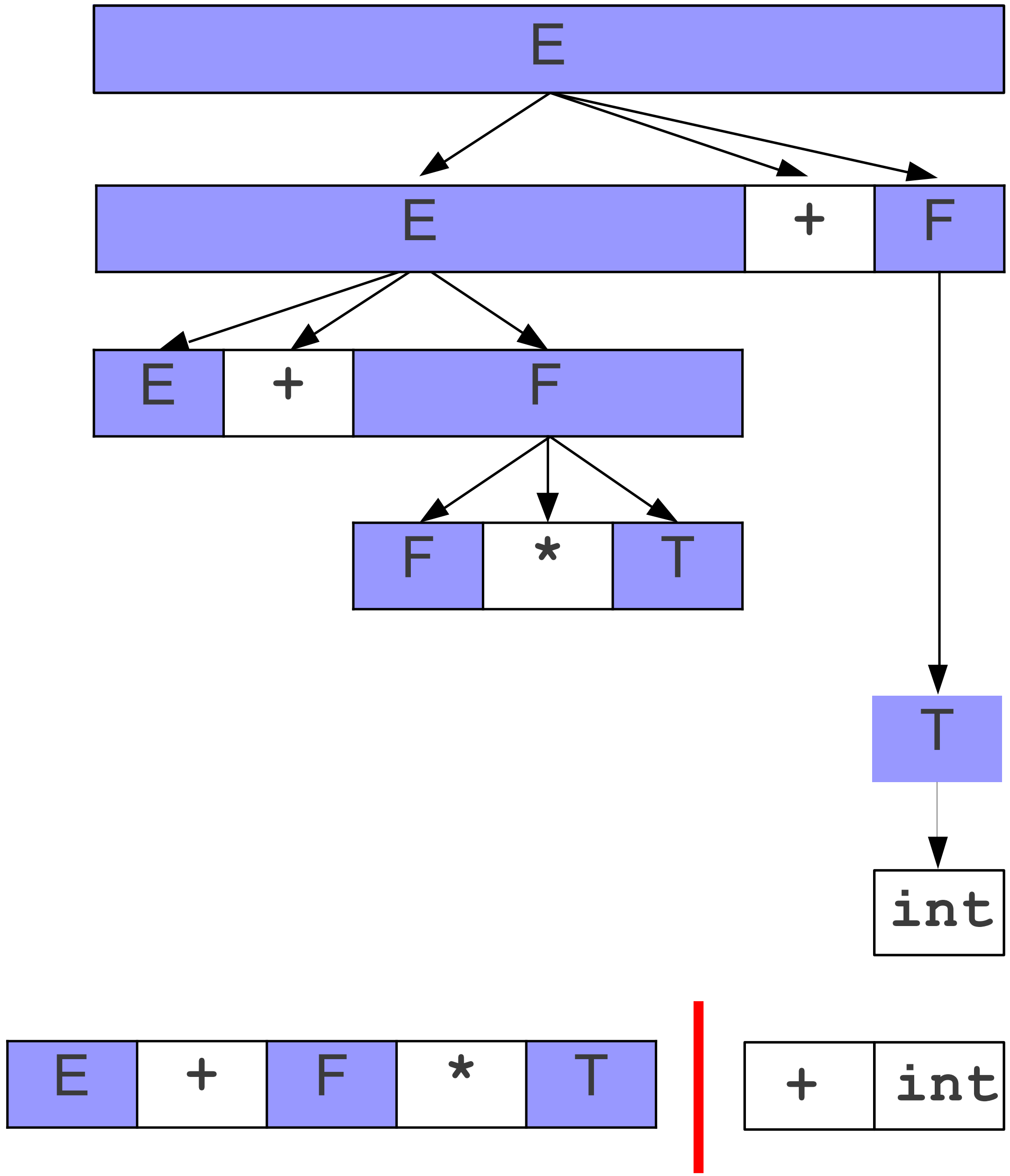
Another Look at Handles

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



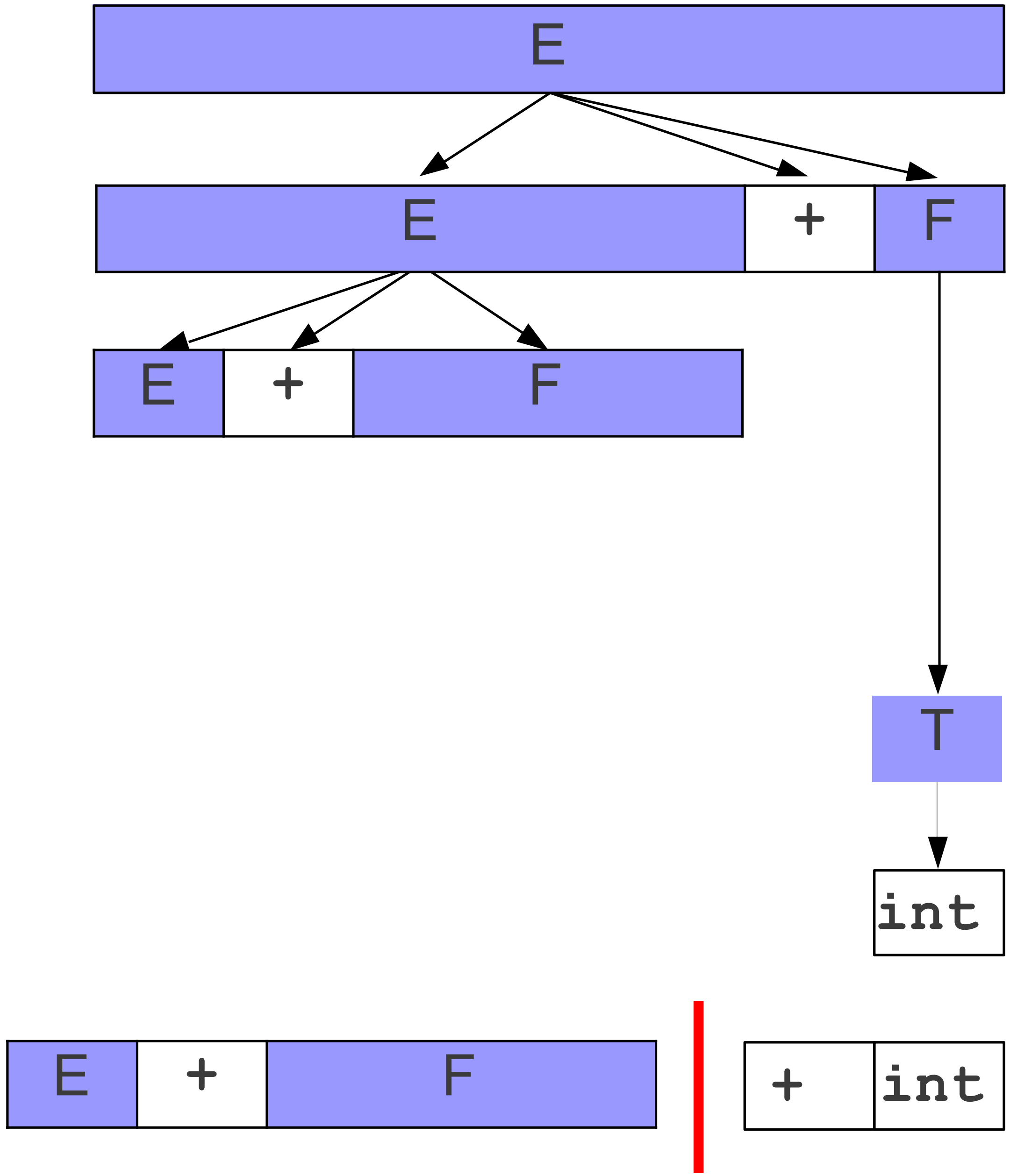
Another Look at Handles

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



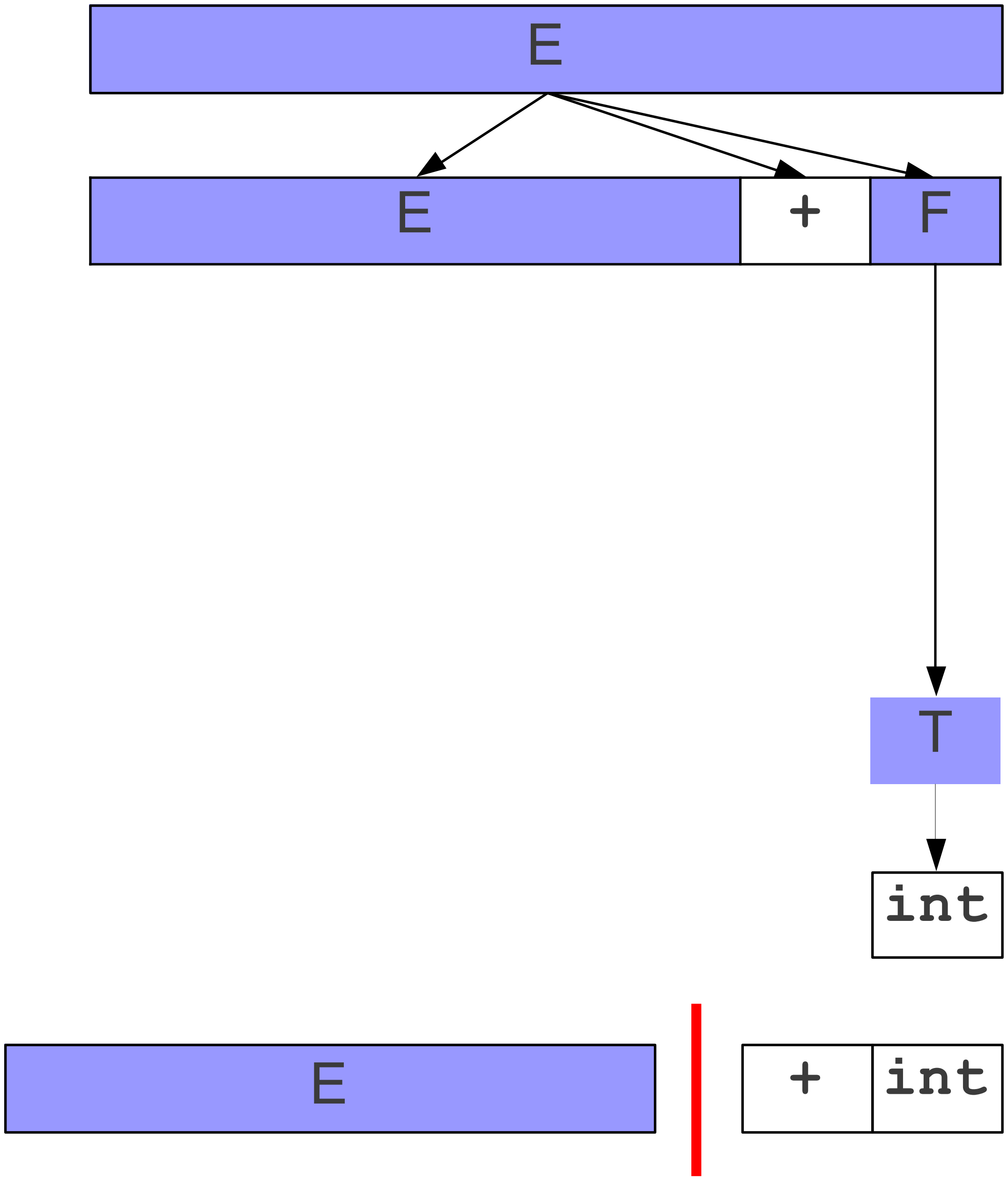
Another Look at Handles

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



Another Look at Handles

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



Another Look at Handles

$E \rightarrow F$

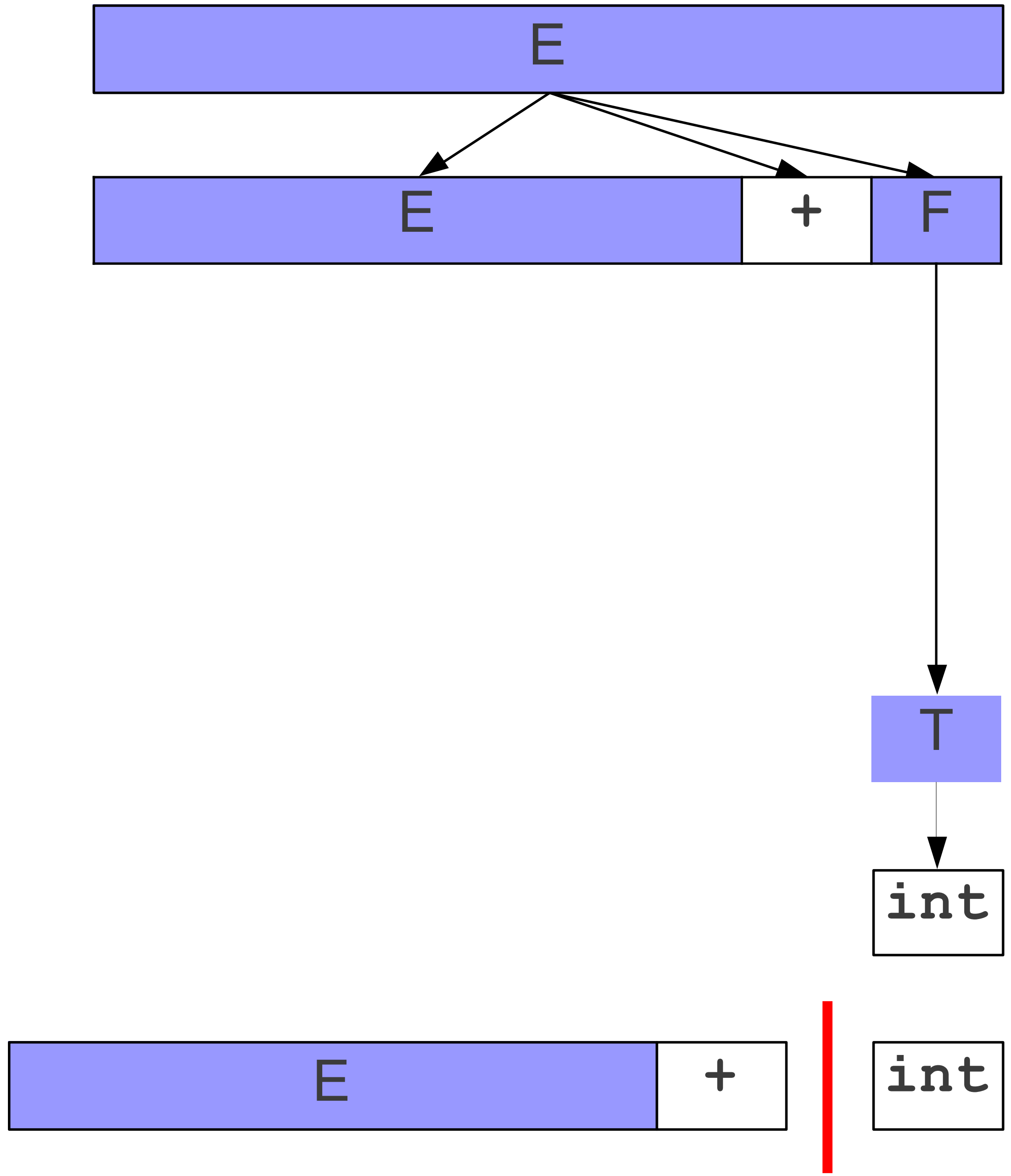
$E \rightarrow E + F$

$F \rightarrow F * T$

$F \rightarrow T$

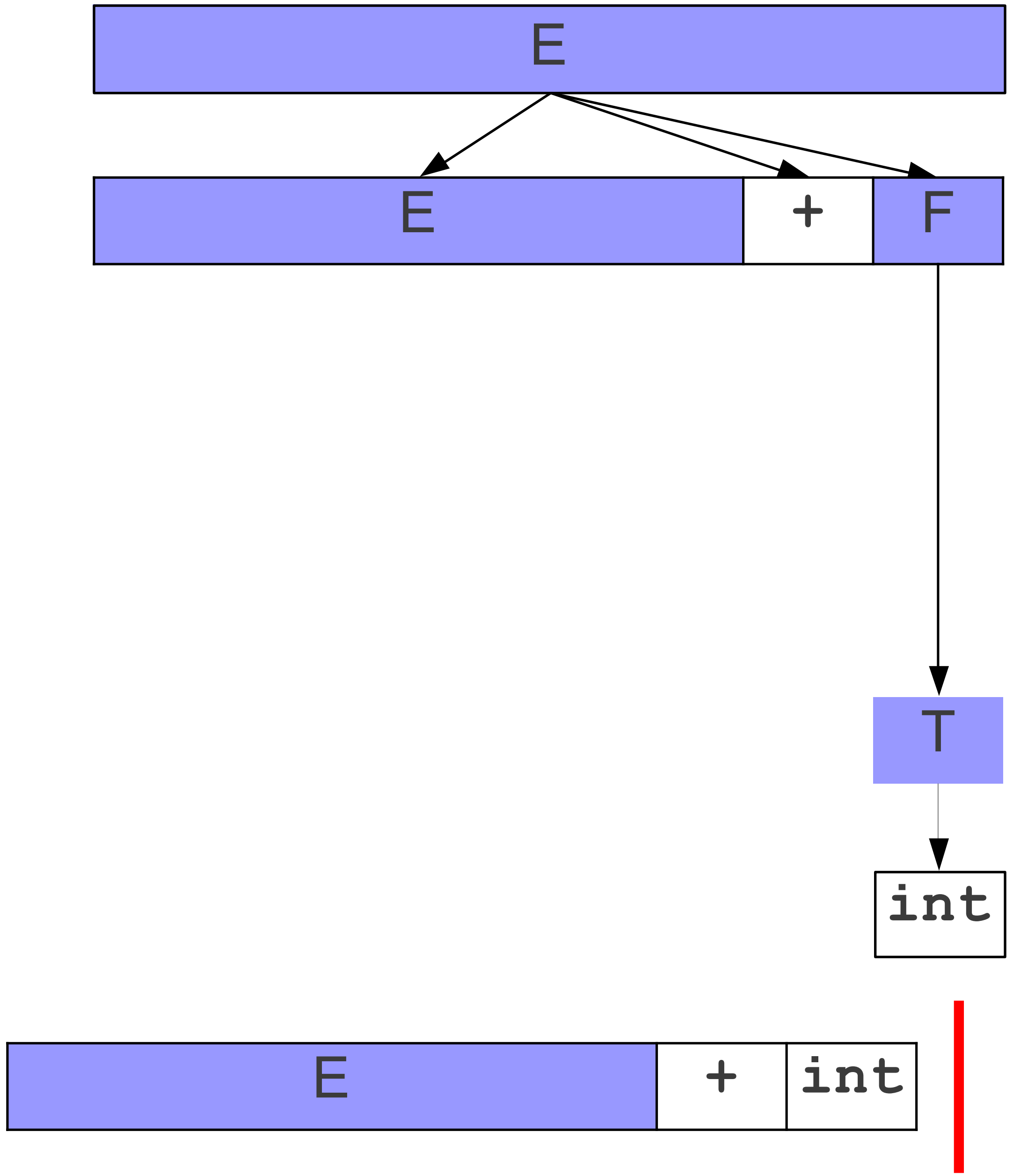
$T \rightarrow$ `int`

$T \rightarrow$ `(E)`



Another Look at Handles

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



Another Look at Handles

$E \rightarrow F$

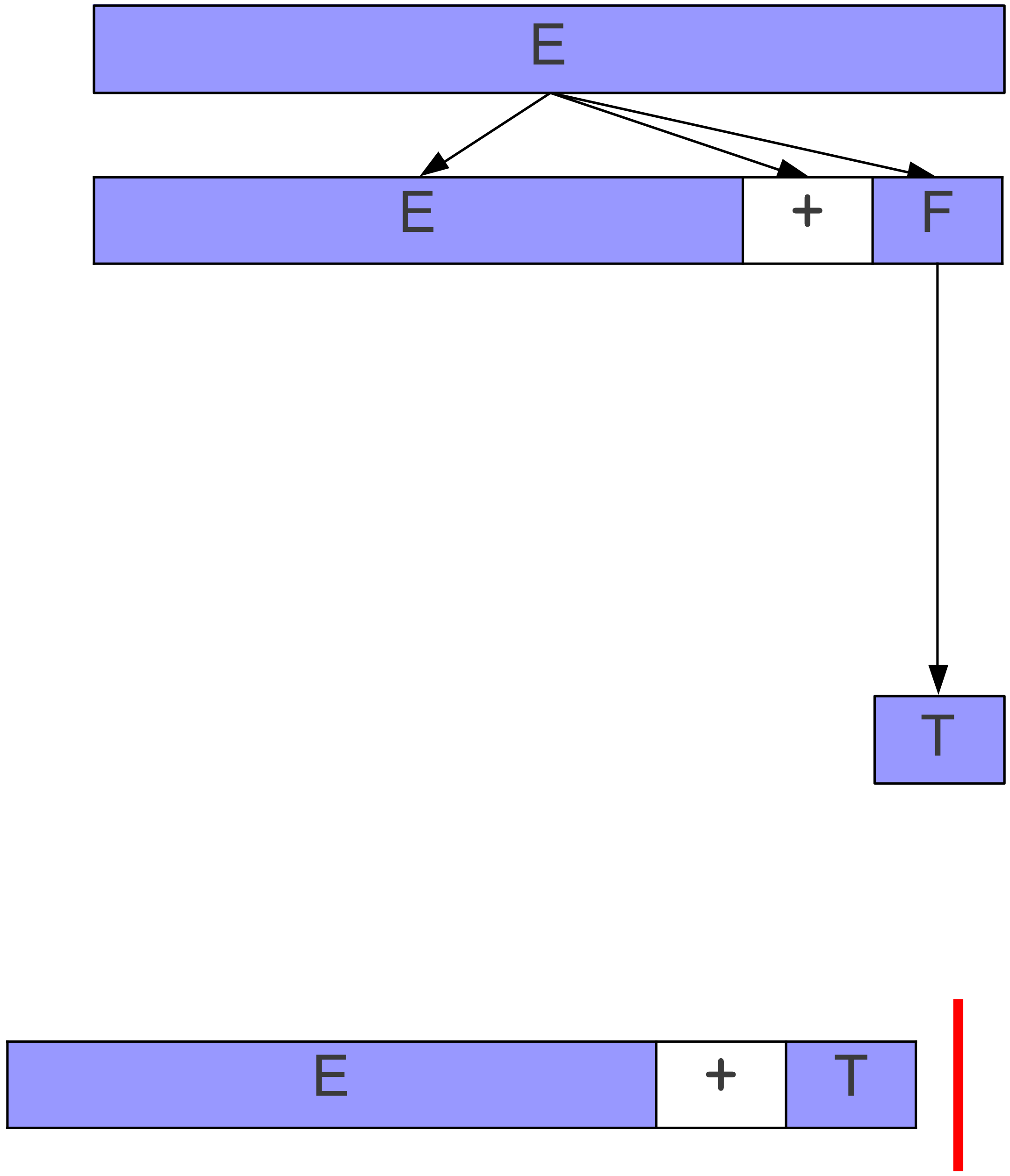
$E \rightarrow E + F$

$F \rightarrow F * T$

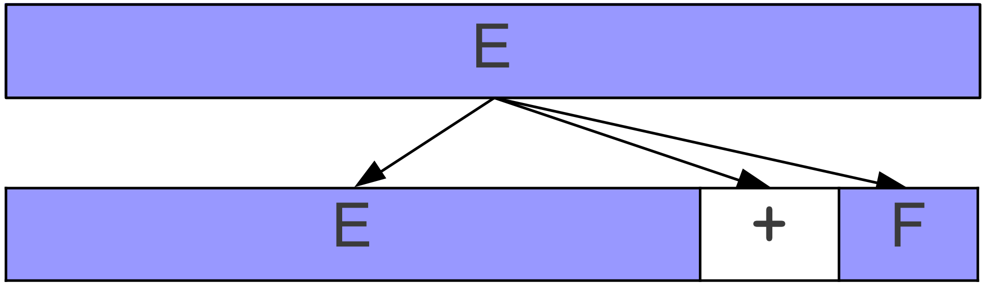
$F \rightarrow T$

$T \rightarrow \texttt{int}$

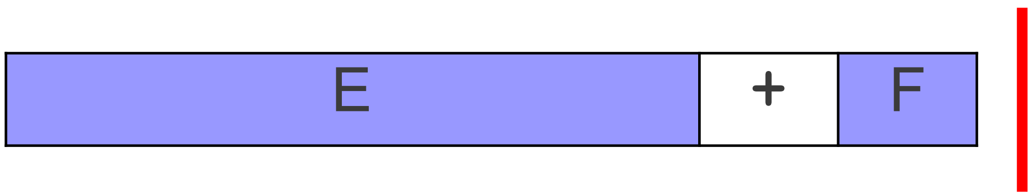
$T \rightarrow (E)$



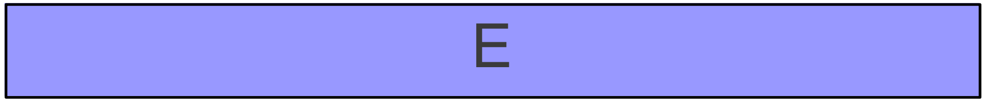
Another Look at Handles



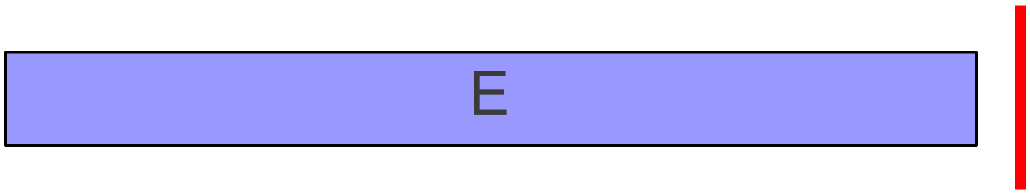
$E \rightarrow F$
$E \rightarrow E + F$
$F \rightarrow F * T$
$F \rightarrow T$
$T \rightarrow \texttt{int}$
$T \rightarrow (E)$



Another Look at Handles



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



Tracking Our Position

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

int	+	int	*	int	+	int

Tracking Our Position

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

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)

$S \to \cdot E$

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**)**

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

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

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

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

int	+	int	*	int	+	int

Tracking Our Position

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

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

int	+	int	*	int	+	int

Tracking Our Position

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

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

int	+	int	*	int	+	int

Tracking Our Position

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

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

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

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

Tracking Our Position

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

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

F	|	+	int	*	int	+	int

Tracking Our Position

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

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

F	+	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		+	int	*	int	+	int

Tracking Our Position

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

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

E	|	+	int	*	int	+	int

Tracking Our Position

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

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

E	+

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**)**

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

E	+

int	*	int	+	int

Tracking Our Position

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

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

E	+	|	int	*	int	+	int

Tracking Our Position

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

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

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

E	+	int	|	*	int	+	int

Tracking Our Position

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

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

E	+	T	|	*	int	+	int

Tracking Our Position

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

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

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

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

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

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

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

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

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	*	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
$T \rightarrow (E)$

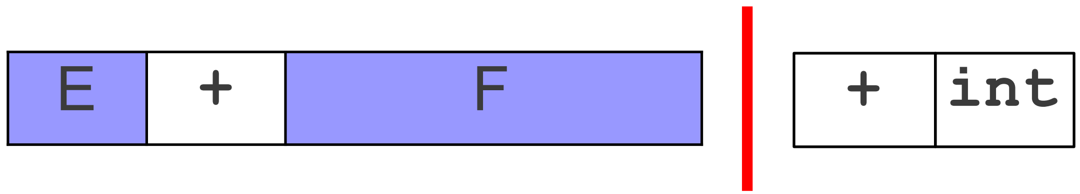
$S \rightarrow \cdot E$
$E \rightarrow \cdot E + F$
$E \rightarrow E + \cdot F$
$F \rightarrow F * T \cdot$

E	+	F	*	T	|	+	int

Tracking Our Position

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

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



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

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

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

E	|	+	`int`

Tracking Our Position

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

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

E	|	+	`int`

Tracking Our Position

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

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

E	+	|	int

Tracking Our Position

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

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

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

E	+	|	`int`

Tracking Our Position

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

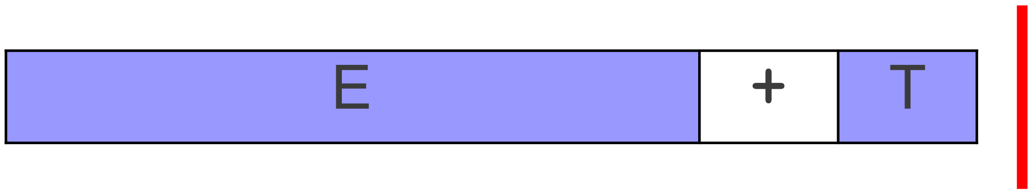
S → · E
E → E + · F
F → · T
T → `int` ·

E	+	`int`

Tracking Our Position

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

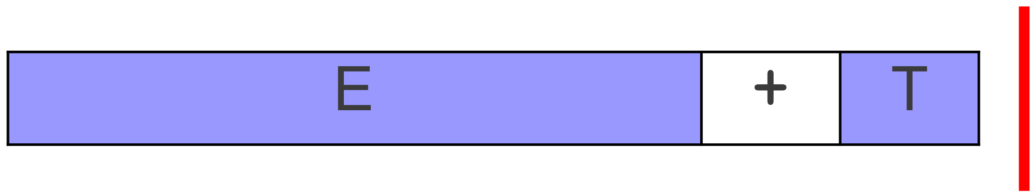
$S \to \cdot E$
$E \to E + \cdot F$
$F \to \cdot T$



Tracking Our Position

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

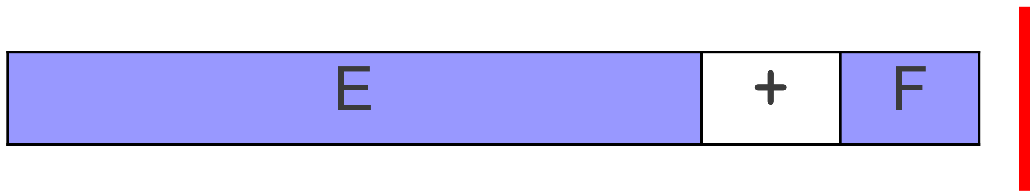
S → · E
E → E + · F
F → T ·



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`)`

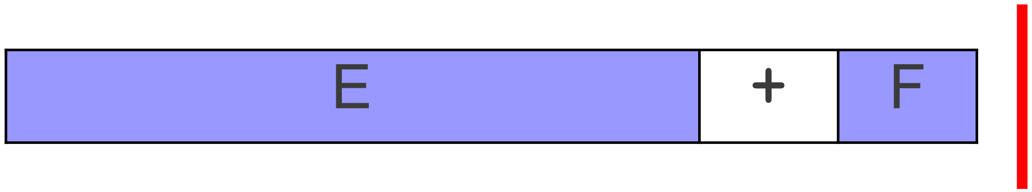
$S \rightarrow \cdot E$
$E \rightarrow E + \cdot F$



Tracking Our Position

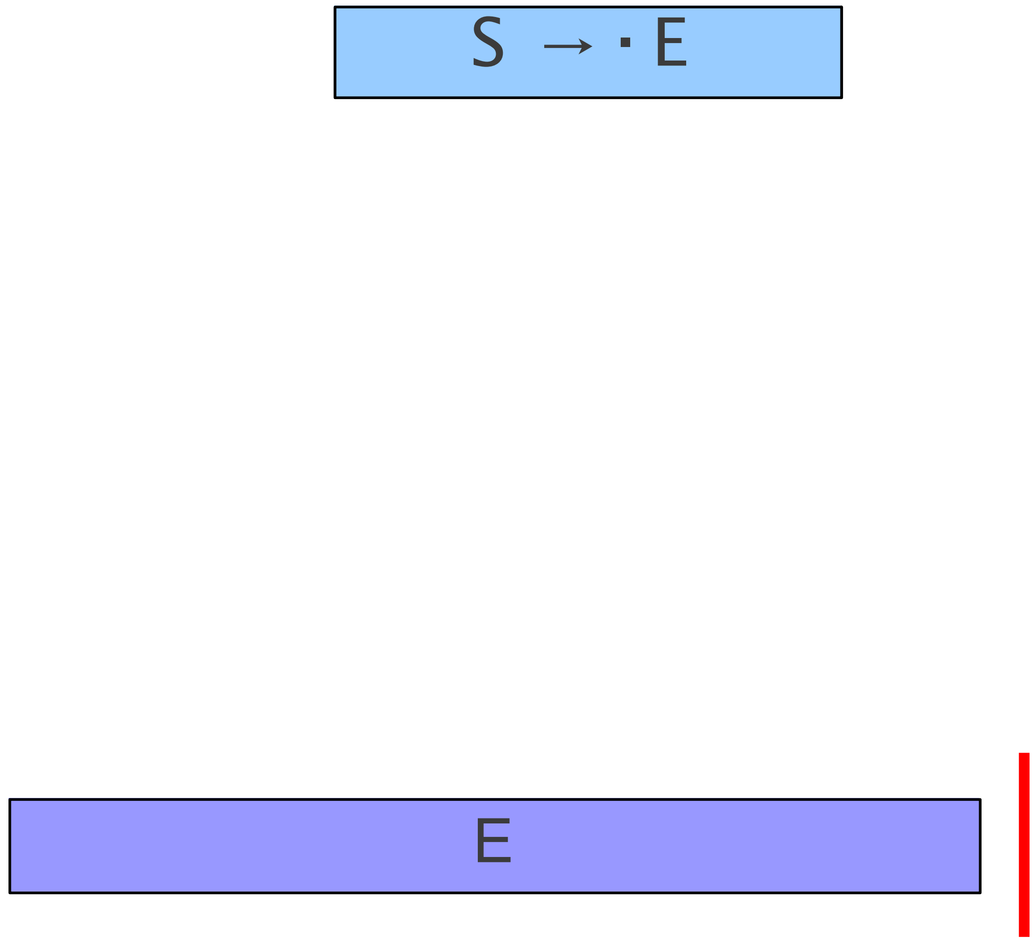
$S \rightarrow E$
$E \rightarrow F$
$E \rightarrow E$ + F
$F \rightarrow F$ * T
$F \rightarrow T$
$T \rightarrow$ `int`
$T \rightarrow$ (E)

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



Tracking Our Position

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



Tracking Our Position

S → E ·

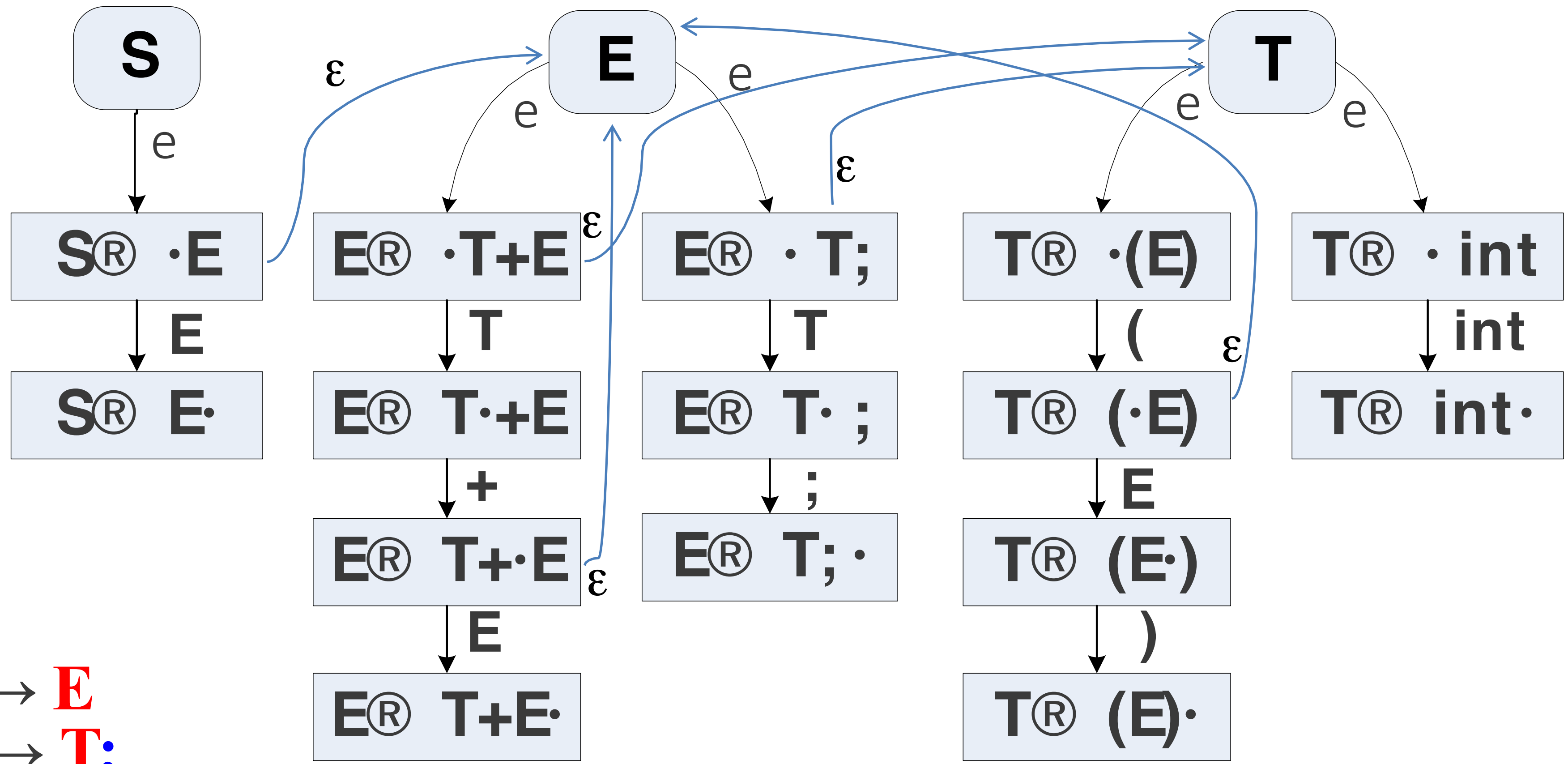
S → **E**
E → **F**
E → **E** `+` **F**
F → **F** `*` **T**
F → **T**
T → `int`
T → `(`**E**`)`

E

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



$S \longrightarrow E$

$E \longrightarrow T;$

$E \longrightarrow T + E$

$T \longrightarrow \mathbf{int}$

$T \longrightarrow (E)$

Constructing the Automaton

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

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

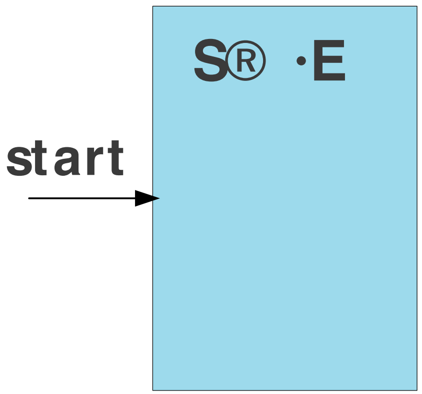
- Then we might be looking at a handle.
- This automaton can be used to discover possible handle locations!

Adding Determinism

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

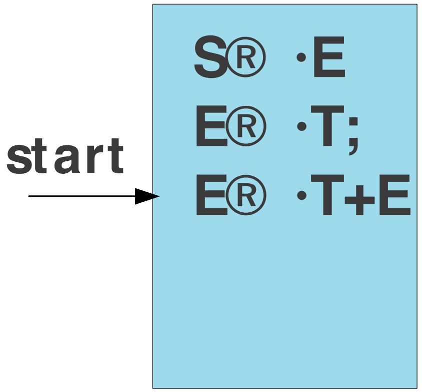
A Deterministic Automaton

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



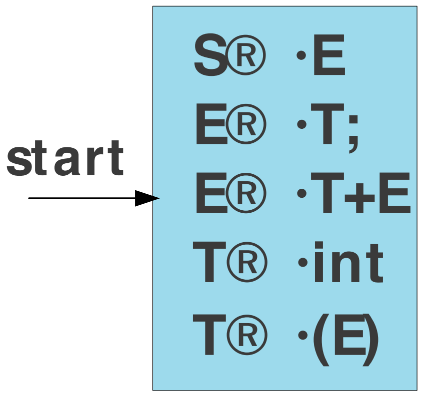
A Deterministic Automaton

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



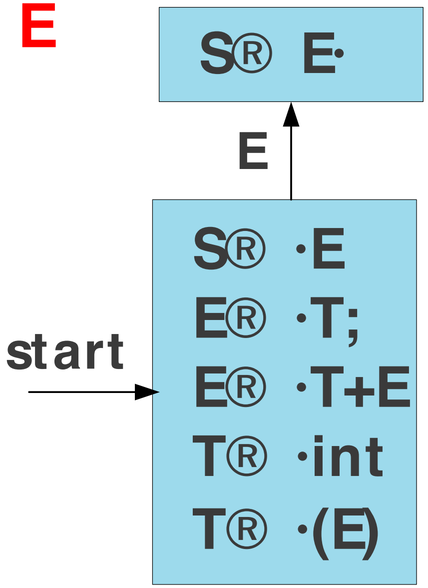
A Deterministic Automaton

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



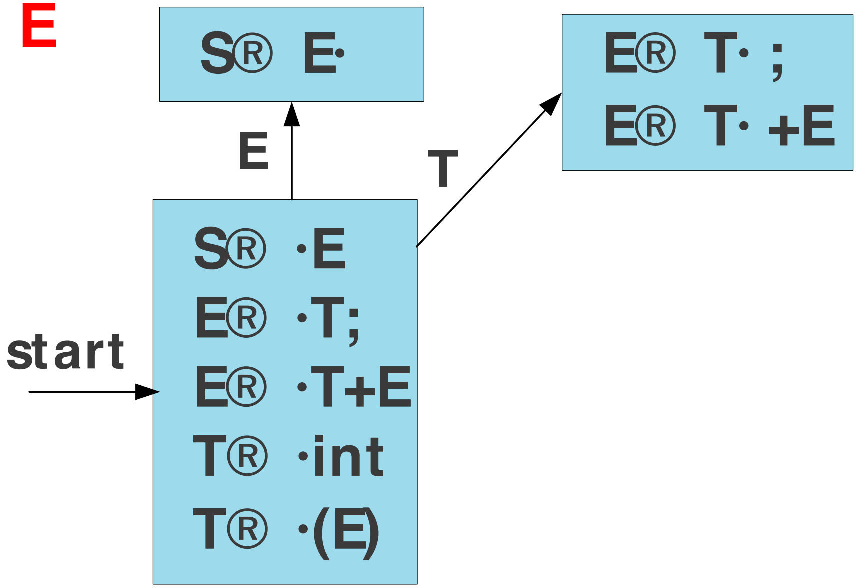
A Deterministic Automaton

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



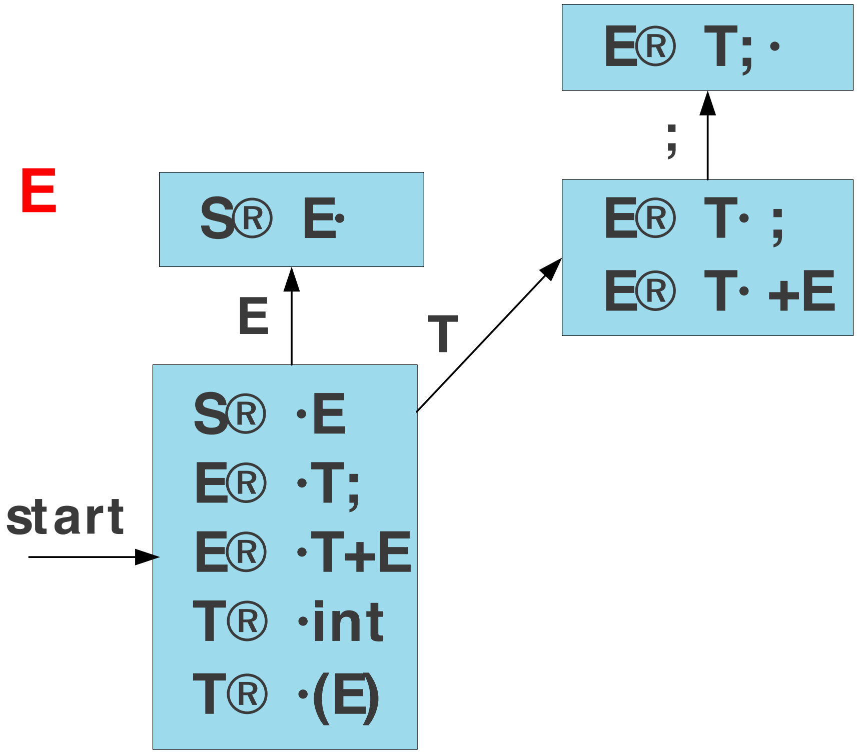
A Deterministic Automaton

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



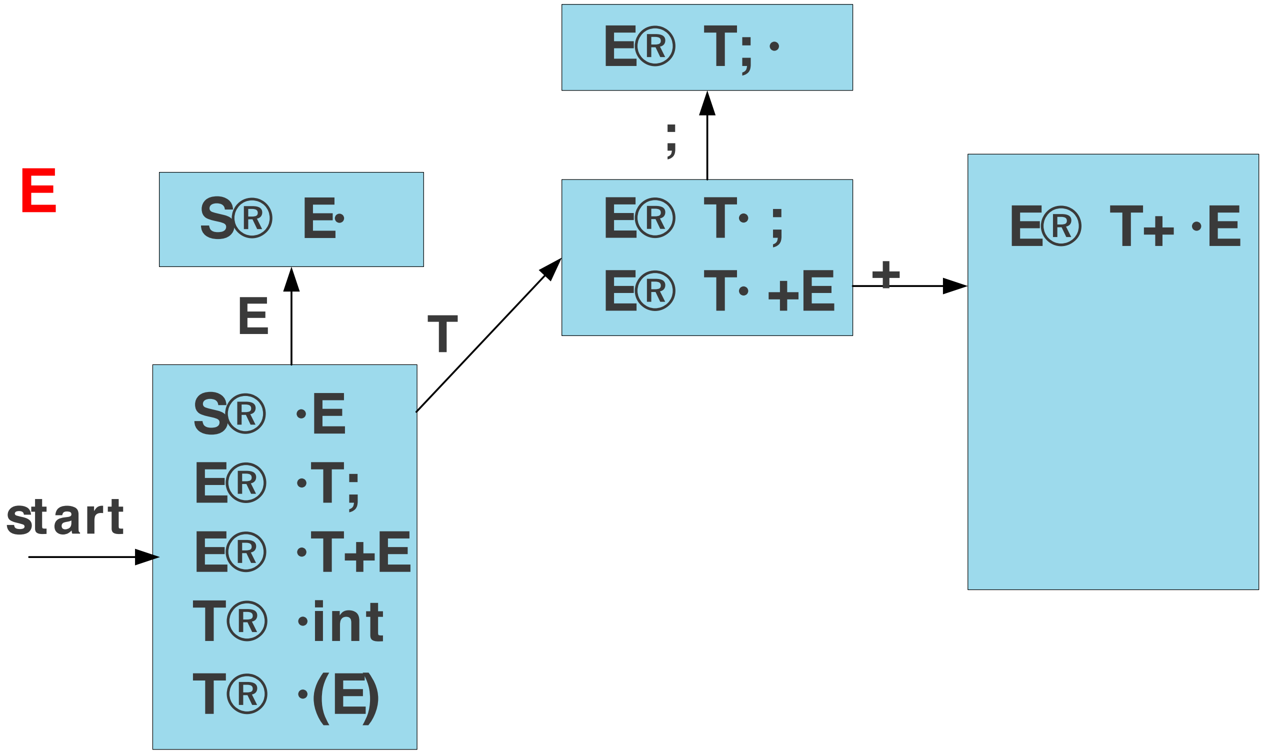
A Deterministic Automaton

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



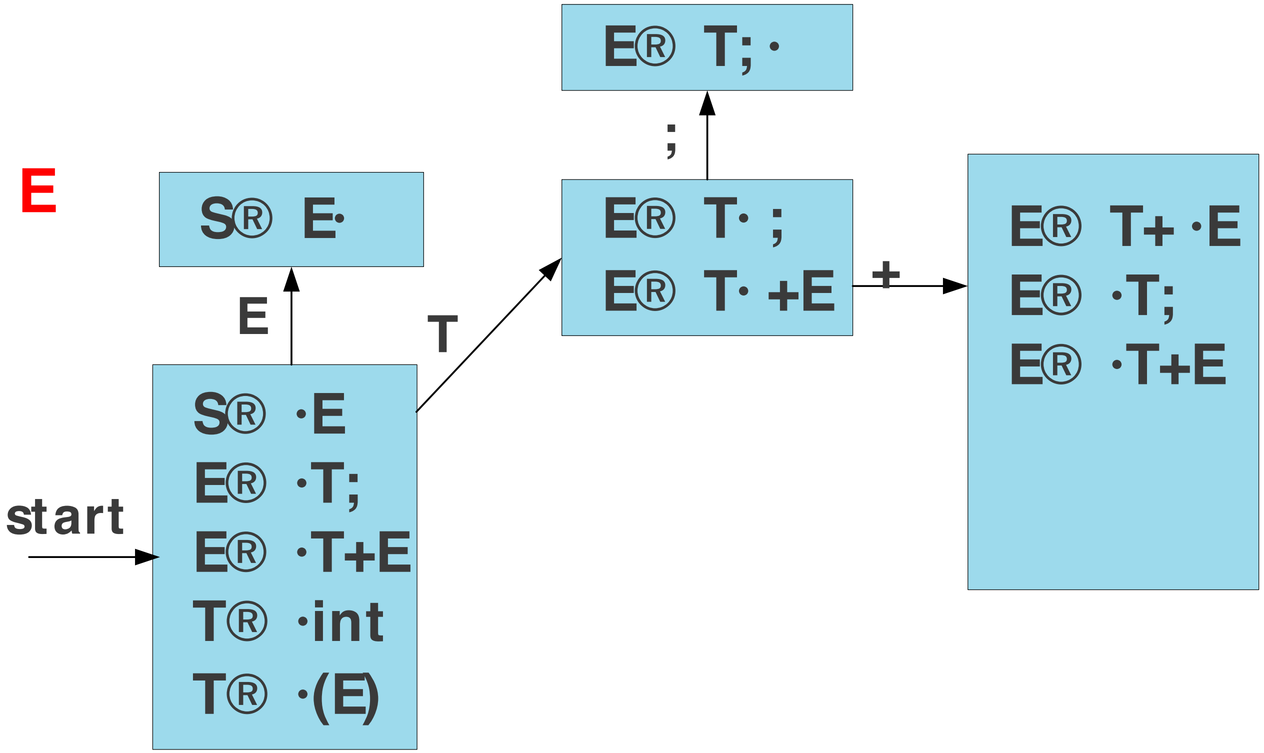
A Deterministic Automaton

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



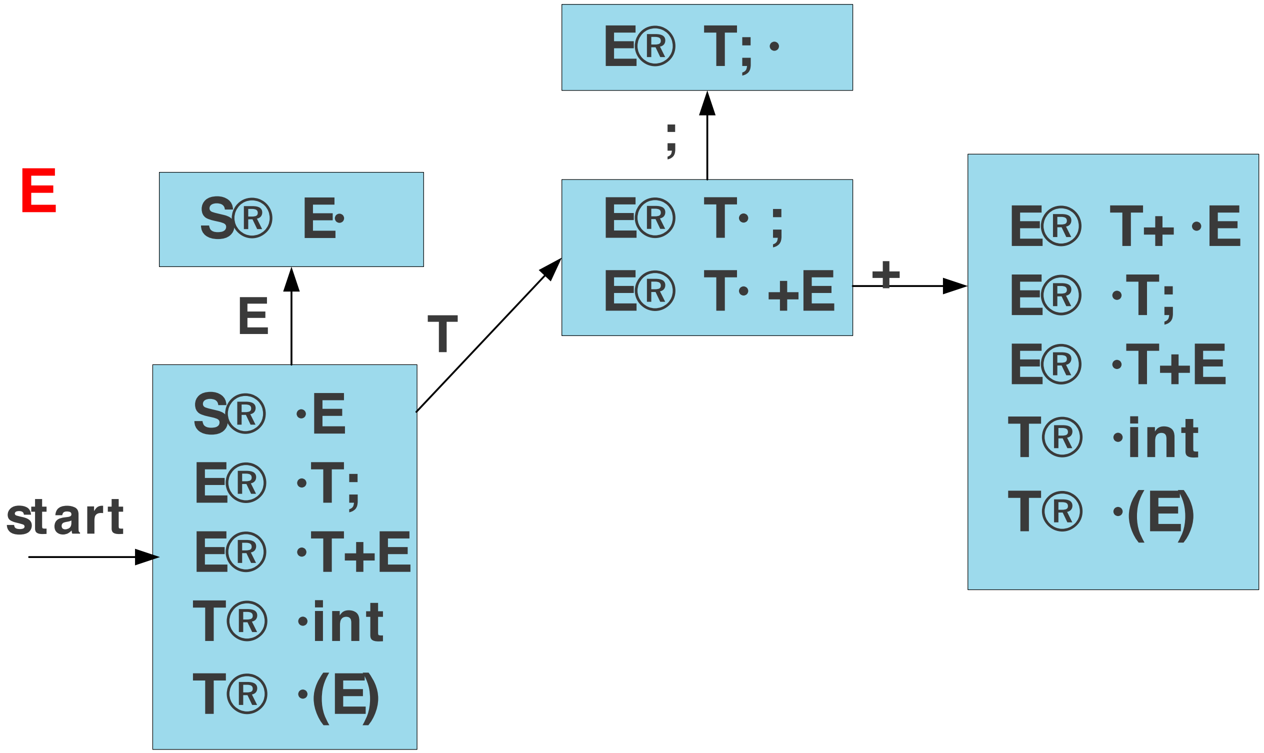
A Deterministic Automaton

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



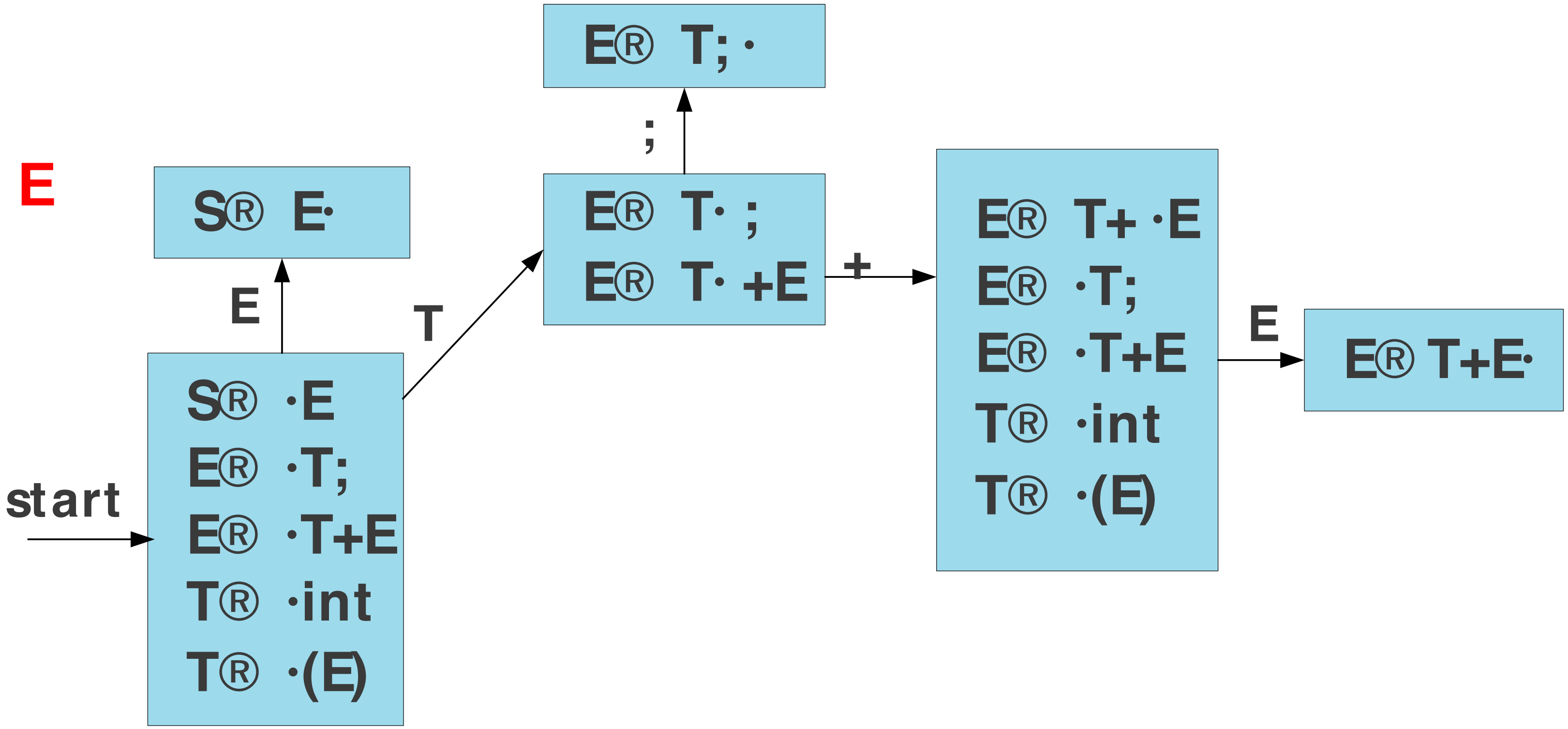
A Deterministic Automaton

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



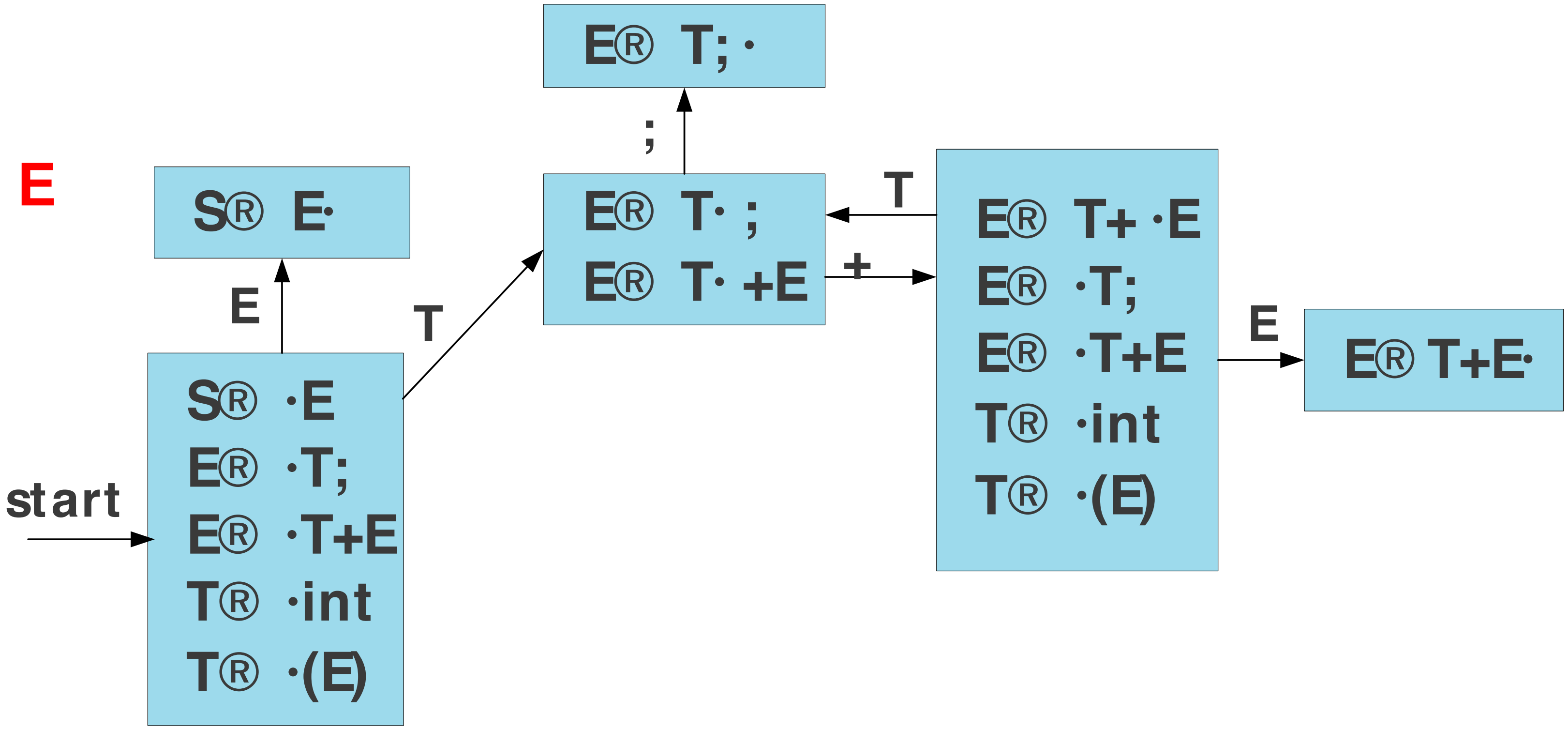
A Deterministic Automaton

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



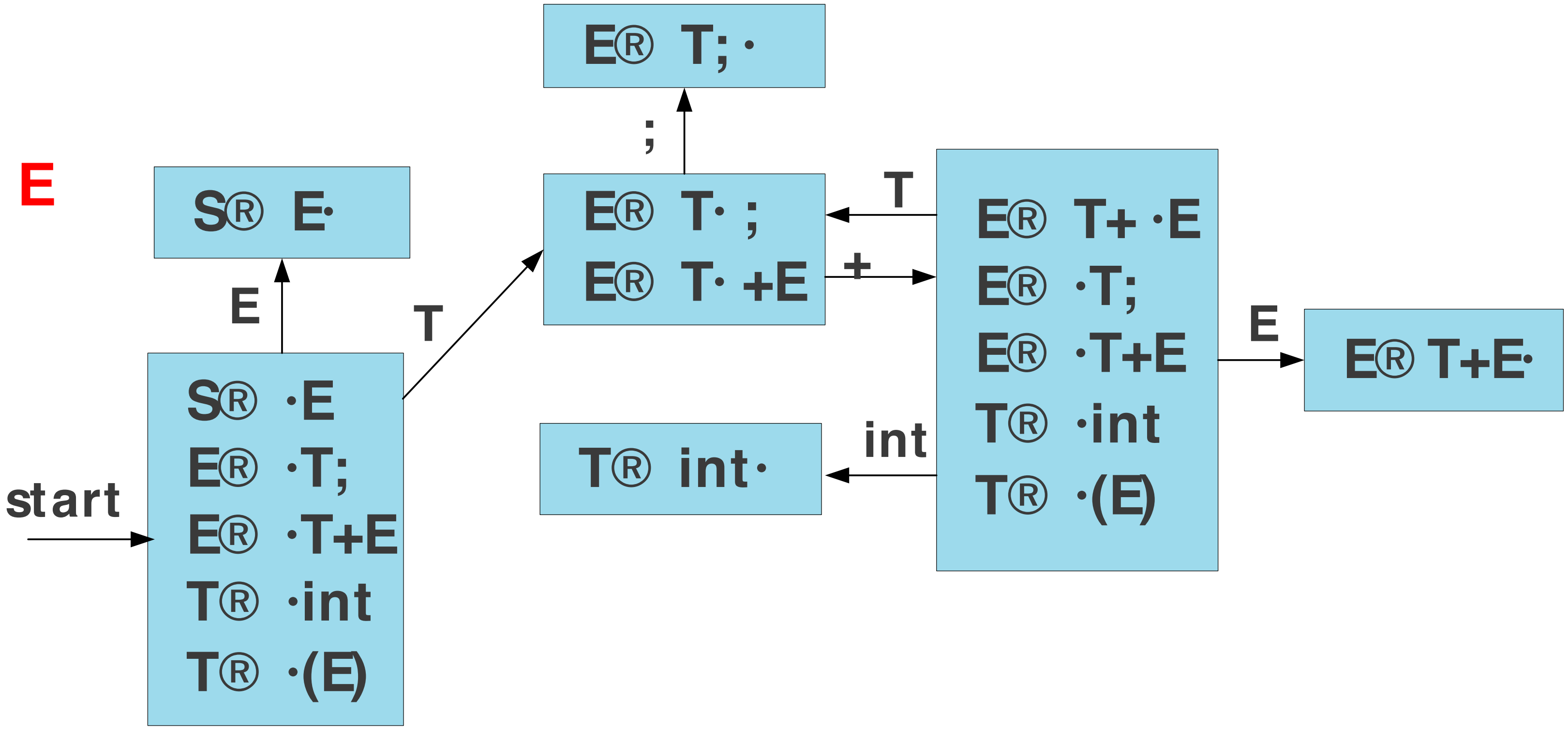
A Deterministic Automaton

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



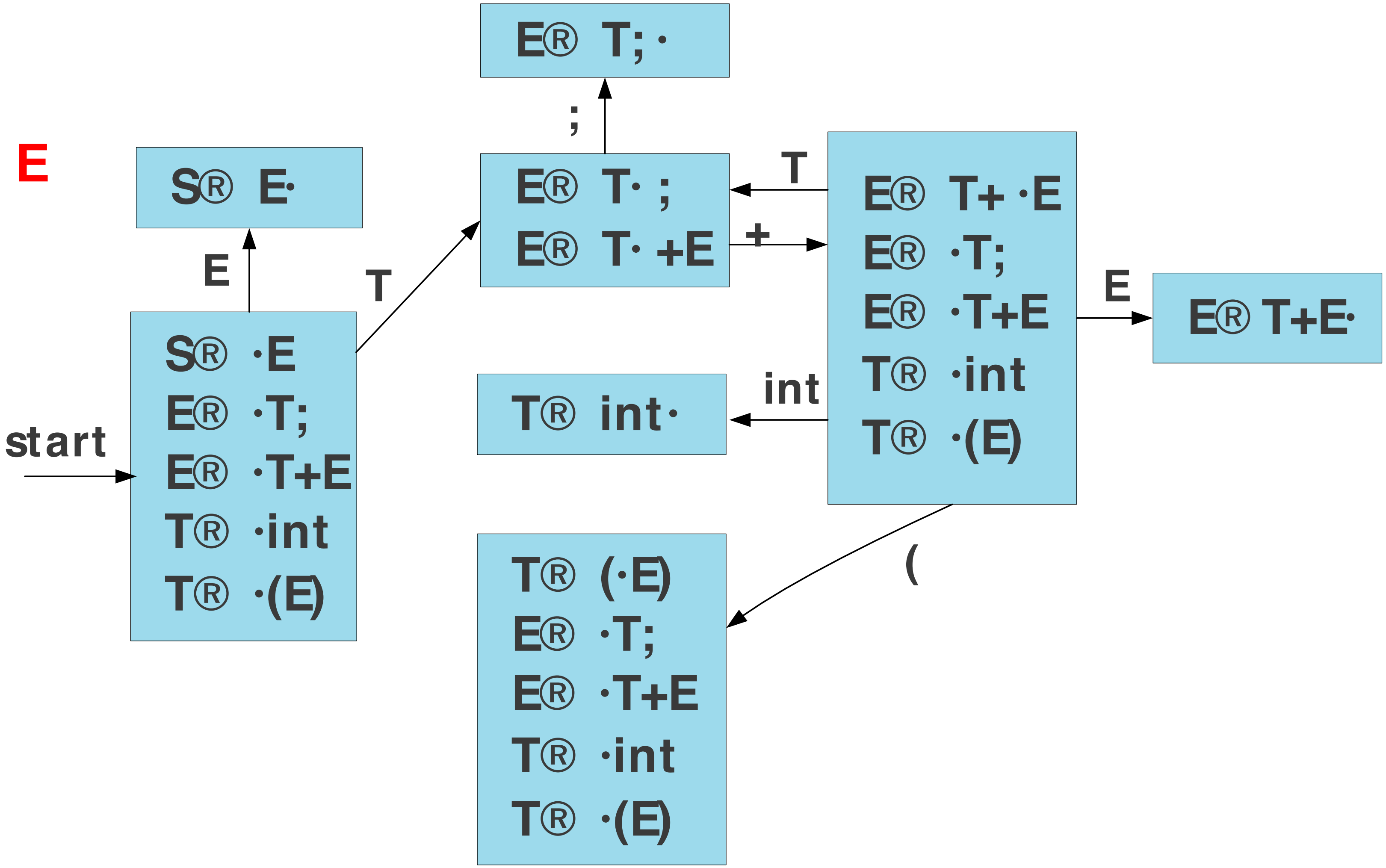
A Deterministic Automaton

S → E
E → T;
E → T+ E
T → int
T → (E)



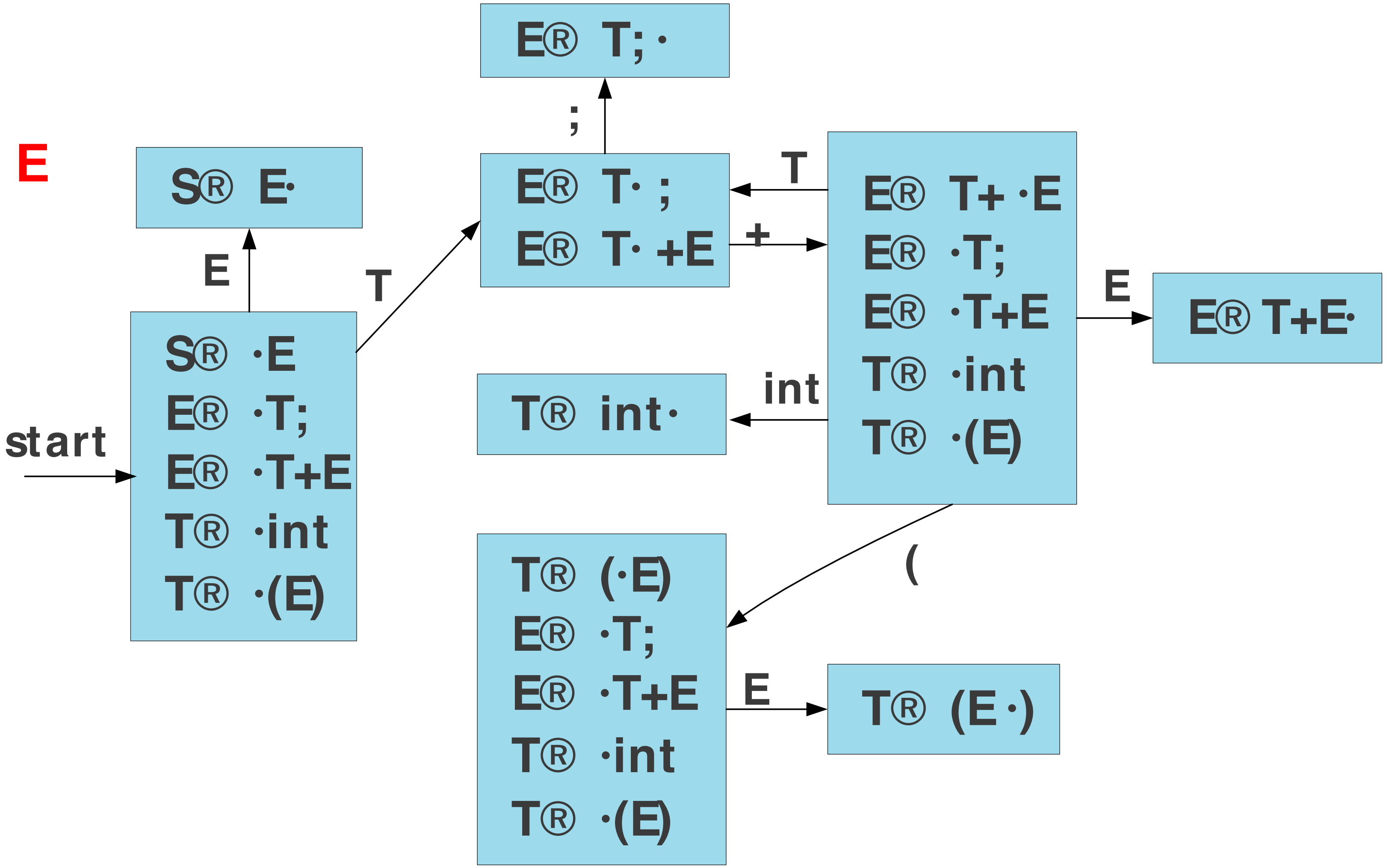
A Deterministic Automaton

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



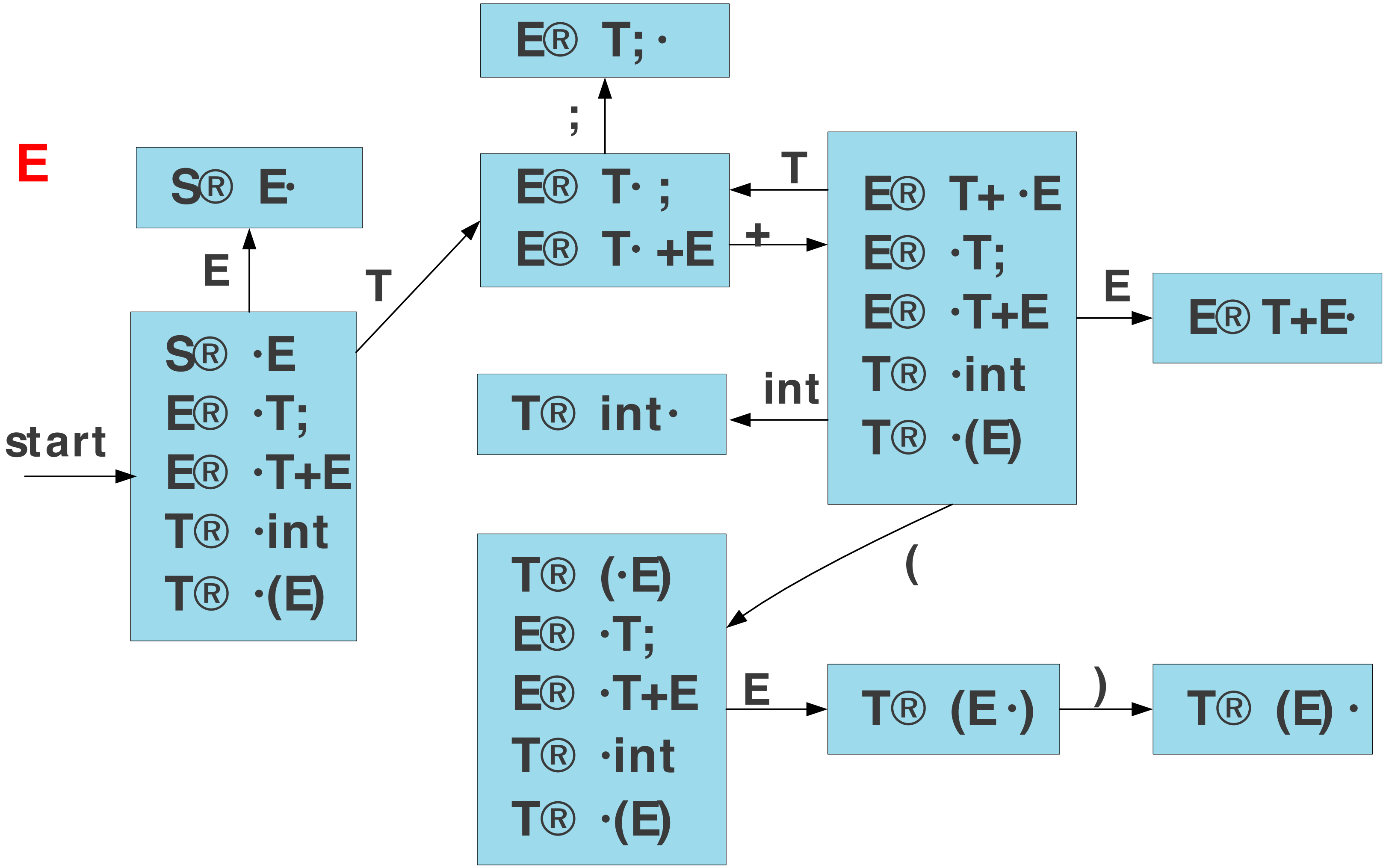
A Deterministic Automaton

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



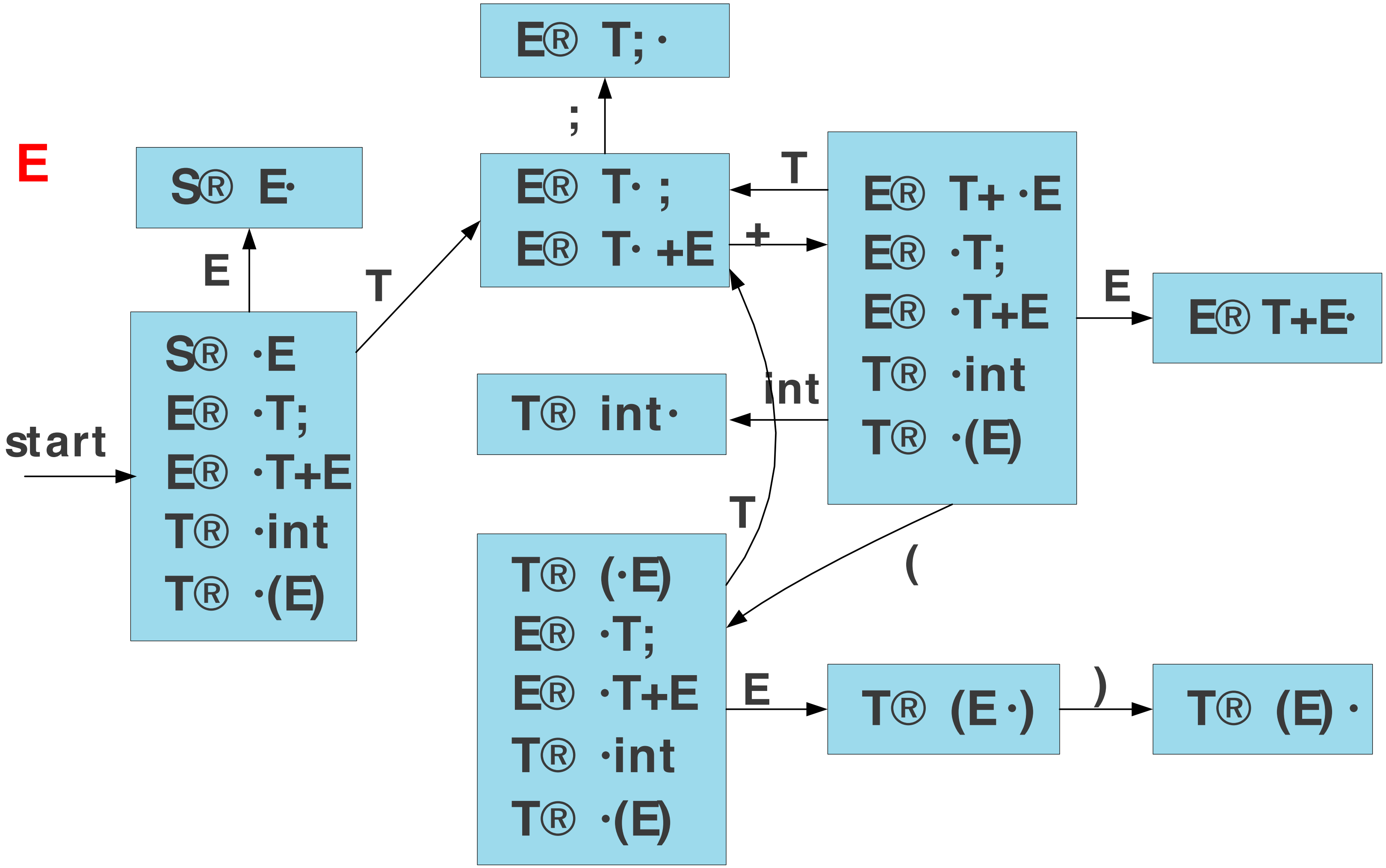
A Deterministic Automaton

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



A Deterministic Automaton

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



A Deterministic Automaton

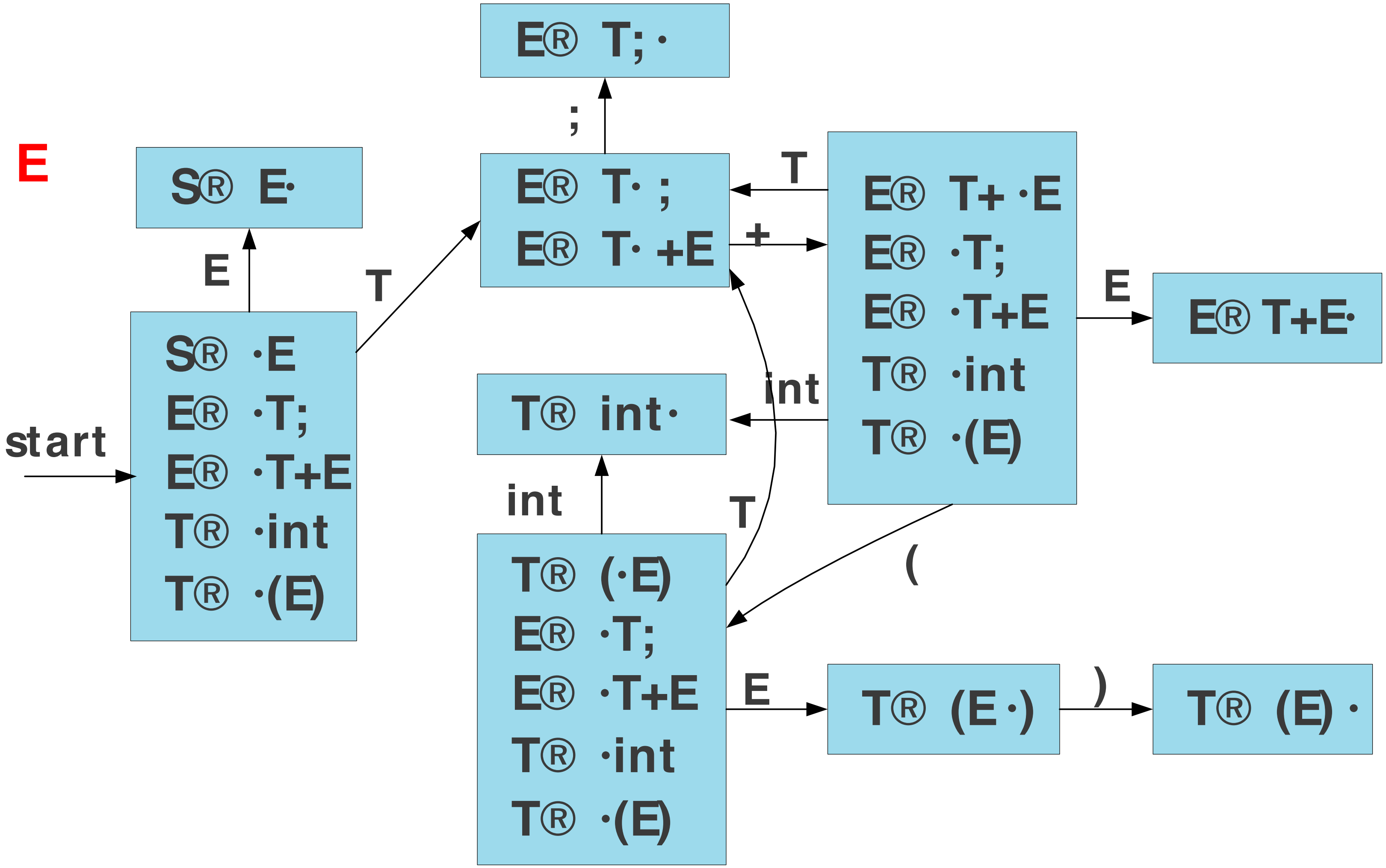
$S \rightarrow E$

$E \rightarrow T$;

$E \rightarrow T + E$

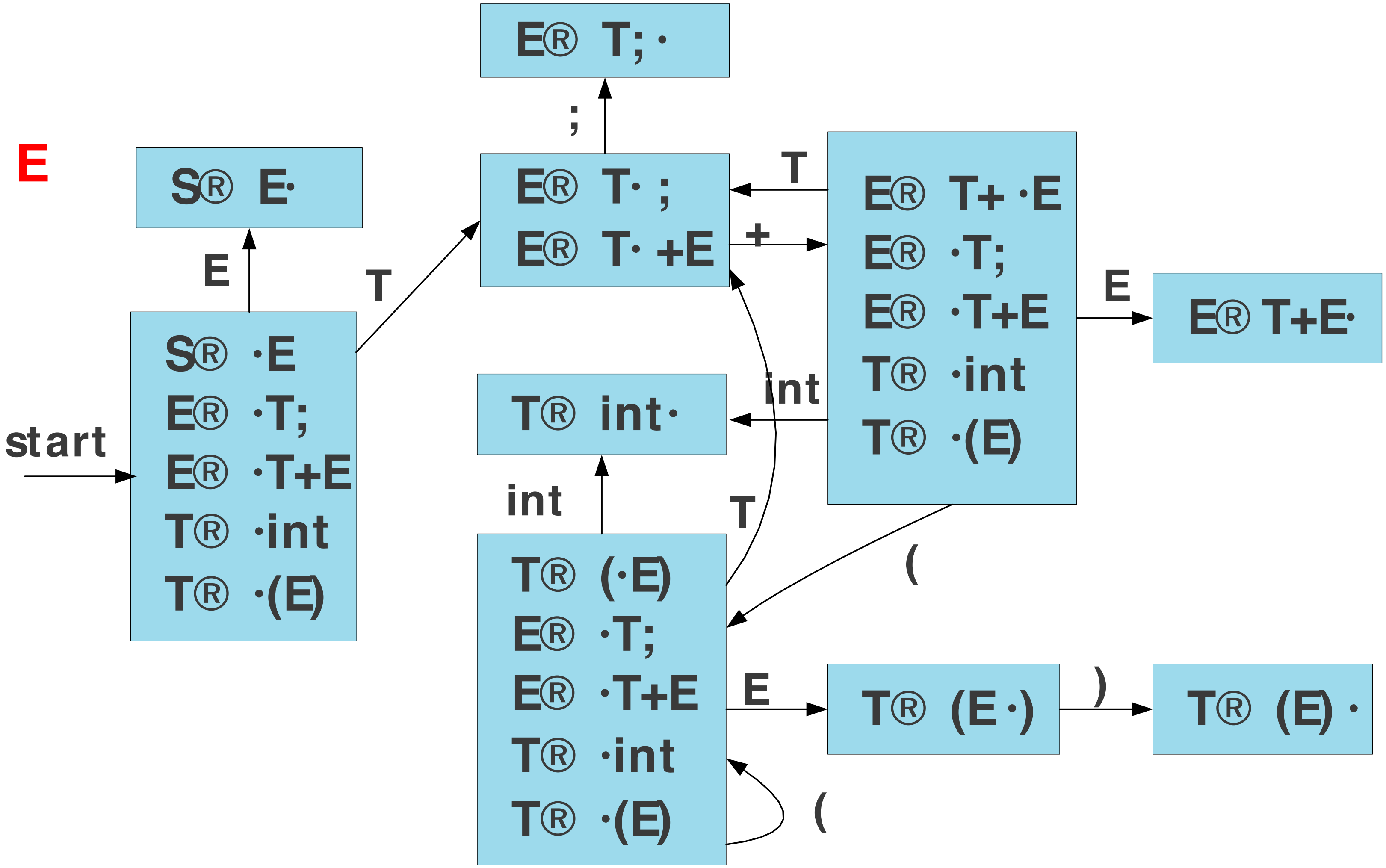
$T \rightarrow$ int

$T \rightarrow$ (E)



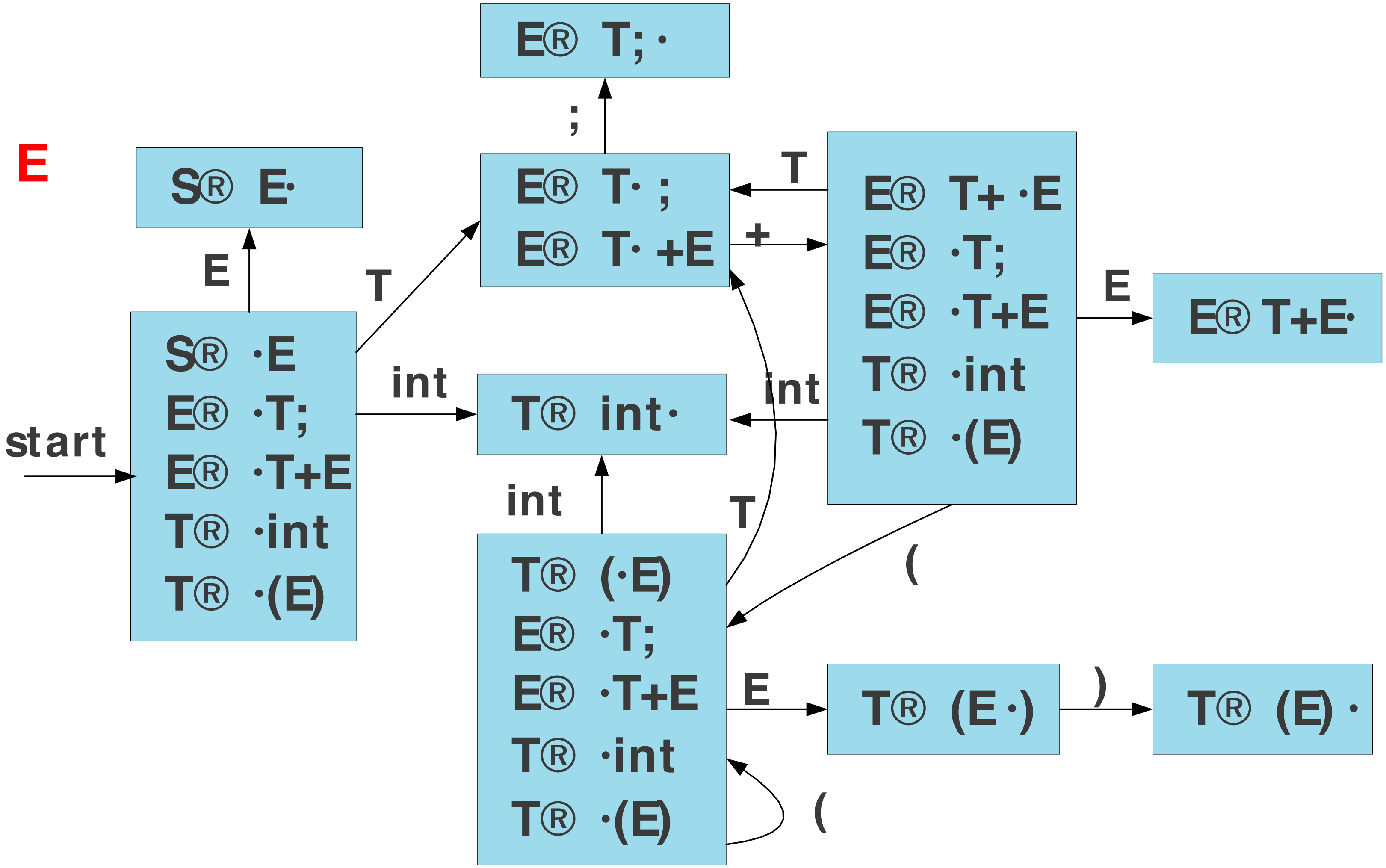
A Deterministic Automaton

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



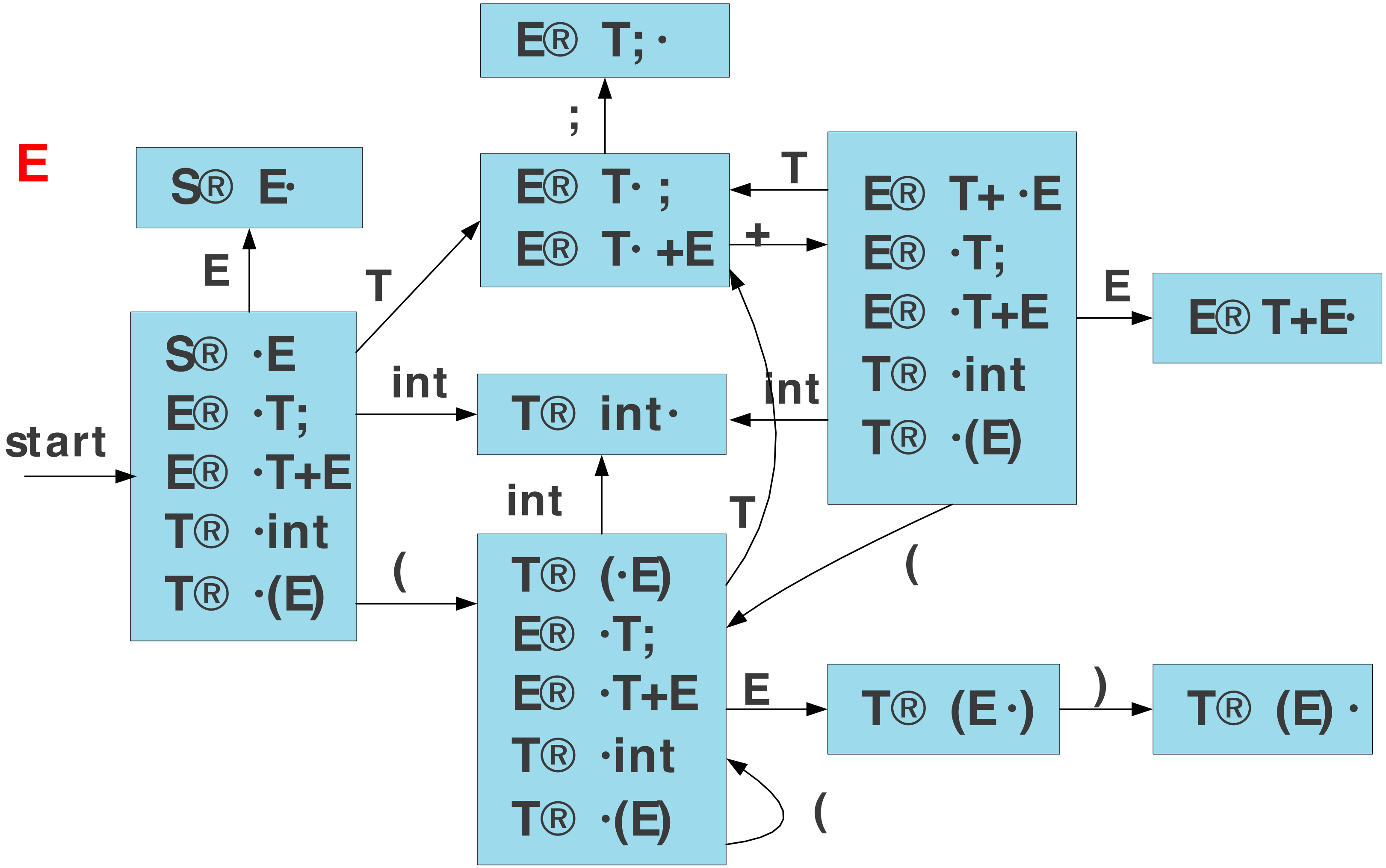
A Deterministic Automaton

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



A Deterministic Automaton

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



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 $\mathbf{A} \rightarrow \alpha \cdot \mathbf{B}\omega$ is in the state, add $\mathbf{B} \rightarrow \cdot\, \gamma$ to the state for each production $\mathbf{B} \rightarrow \gamma$.
  - Yet another fixed-point iteration!
- Repeat until no new states are added:
  - If a state contains a production $\mathbf{A} \rightarrow \alpha \cdot \boldsymbol{x}\omega$ for symbol $\boldsymbol{x}$, add a transition on $\boldsymbol{x}$ from that state to the state containing the closure of $\mathbf{A} \rightarrow \alpha\boldsymbol{x} \cdot \omega$
- 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 possible handles?
  - **Build a handle-finding automaton.**
- How do we recognize handles?
  - Once we've found a candidate handle, how do we check that it really is the handle?

Question Two:

How do we recognize handles?

Handle Recognition

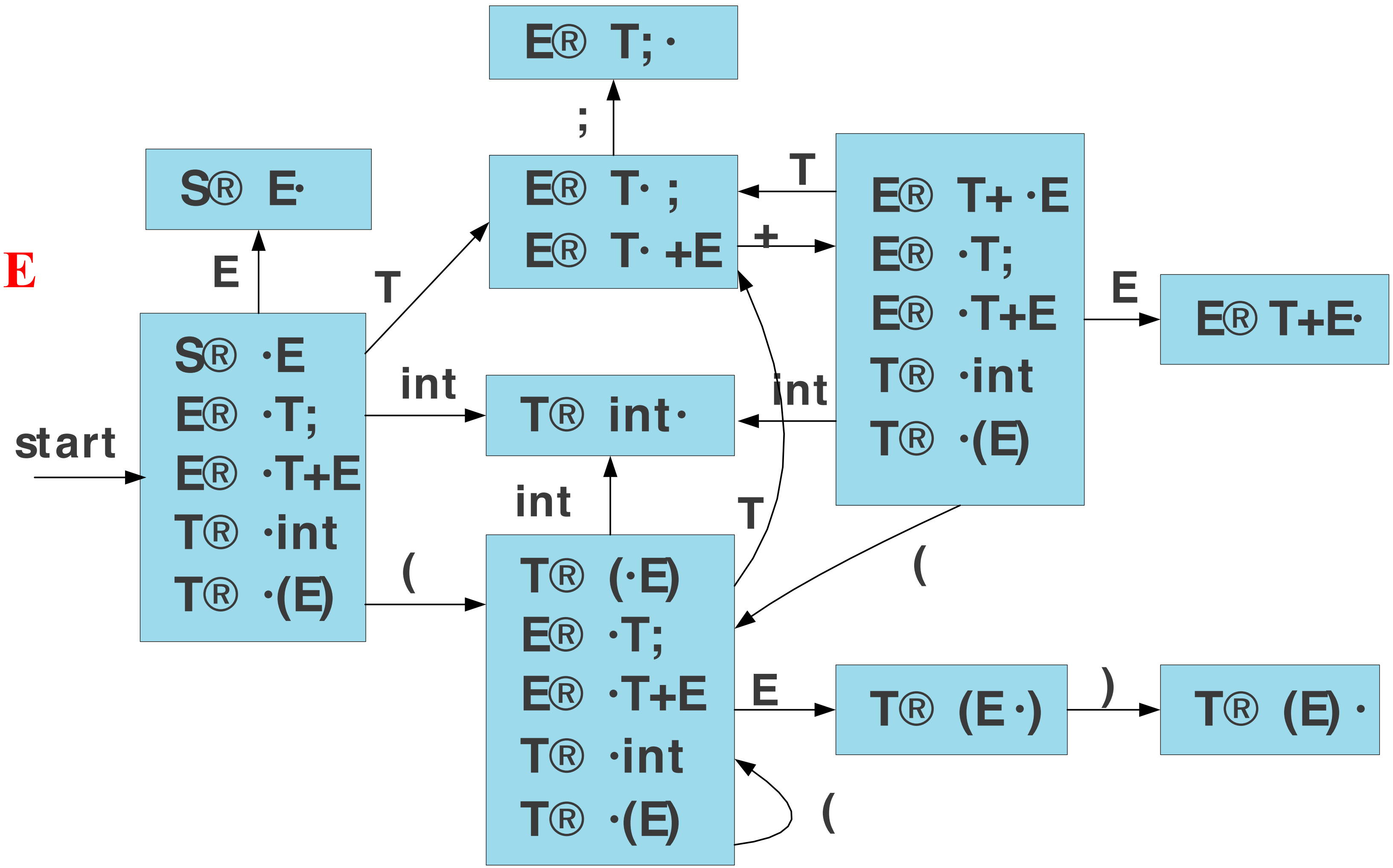
- Our automaton will tell us all places where a handle might be.
- However, if the automaton says that there might be a handle at a given point, we need a way to confirm this.
- We'll thus use **predictive bottom-up parsing**:
  - Have a deterministic procedure for guessing where handles are.
- There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

The 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

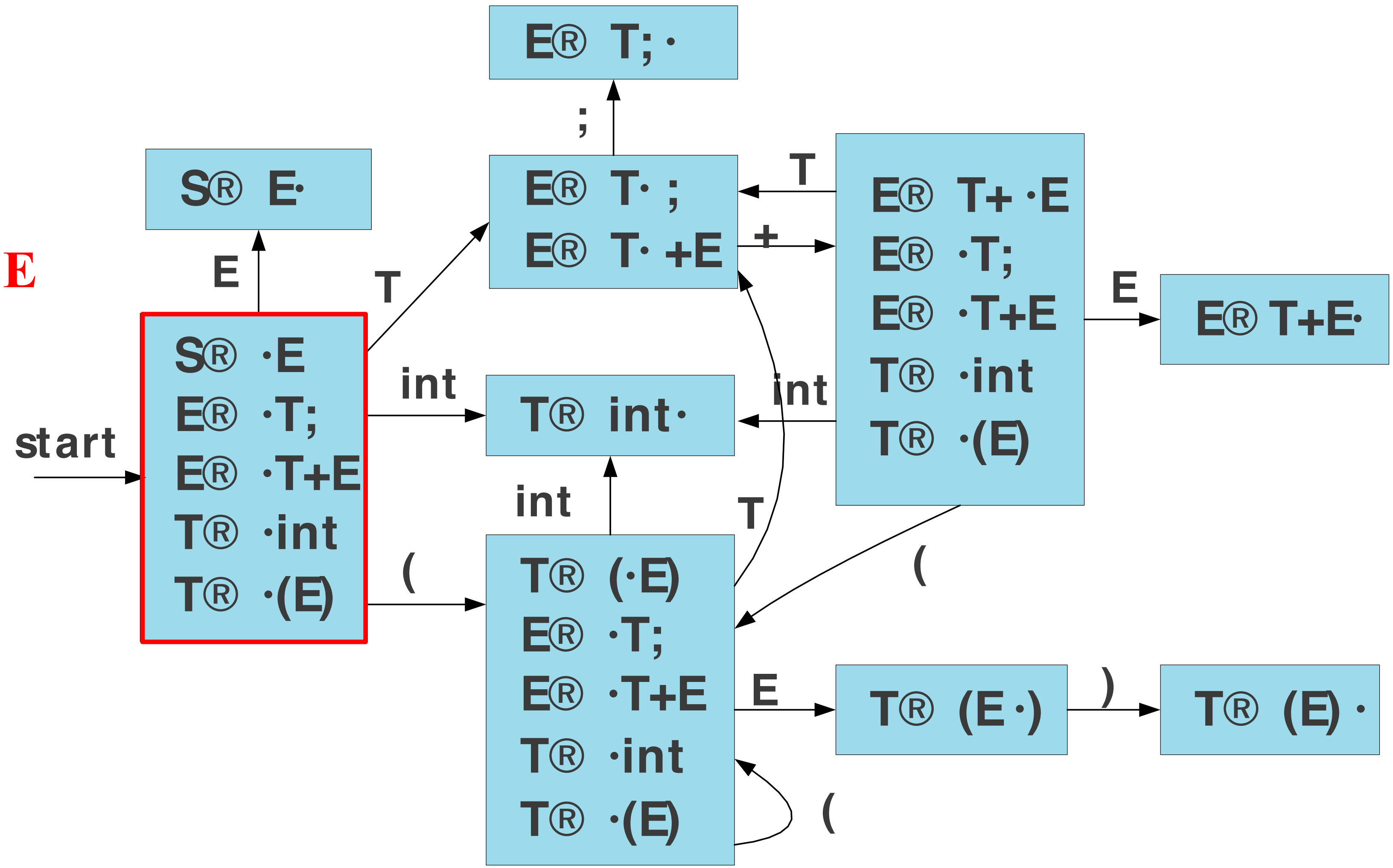
$S \to E$
$E \to T;$
$E \to T + E$
$T \to \mathrm{int}$
$T \to (E)$



int	+	(	int	+	int	;	)	;

LR(0) Parsing

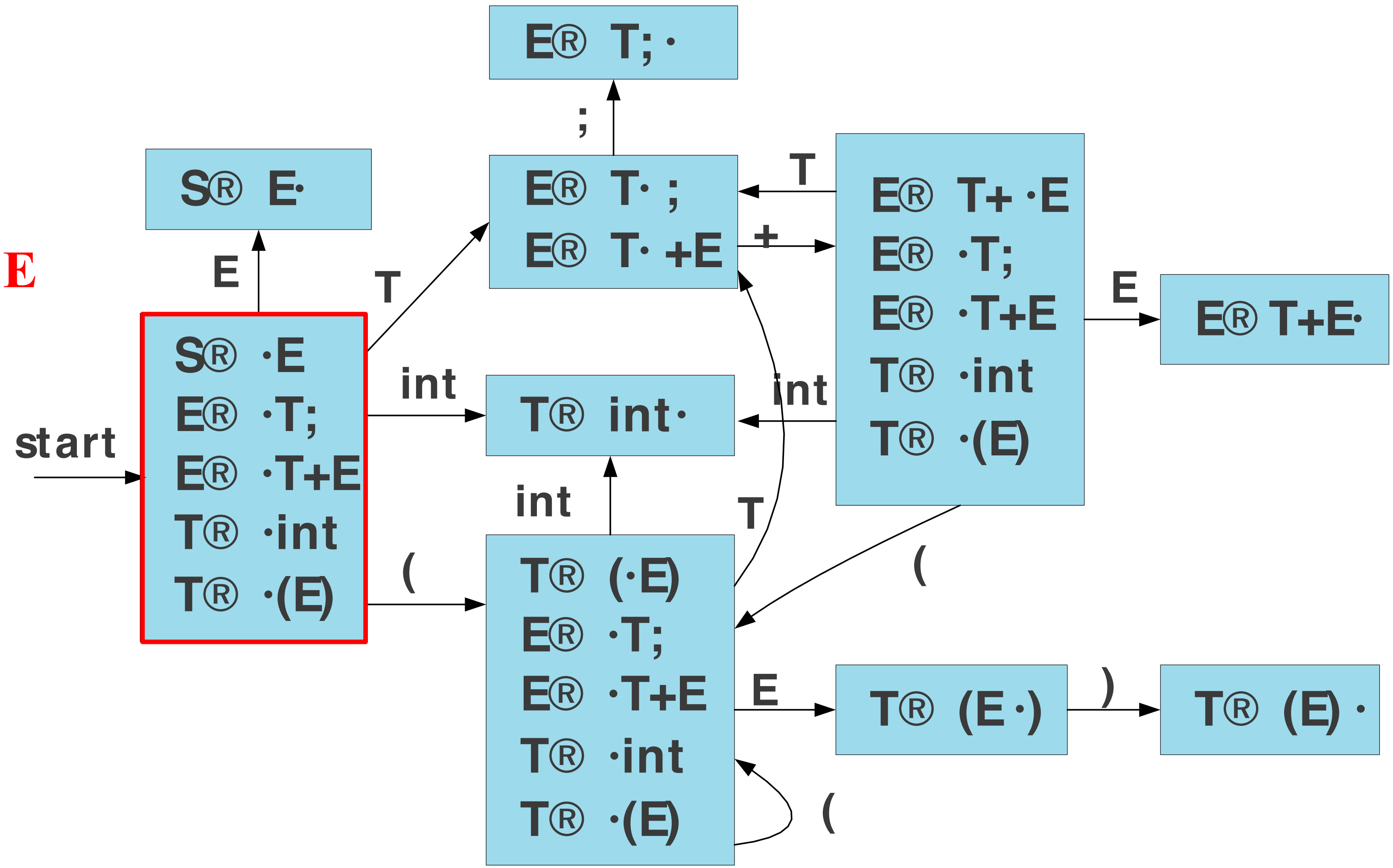
S → E
E → T;
E → T + E
T → int
T → (E)



int	+	(	int	+	int	;	)	;

LR(0) Parsing

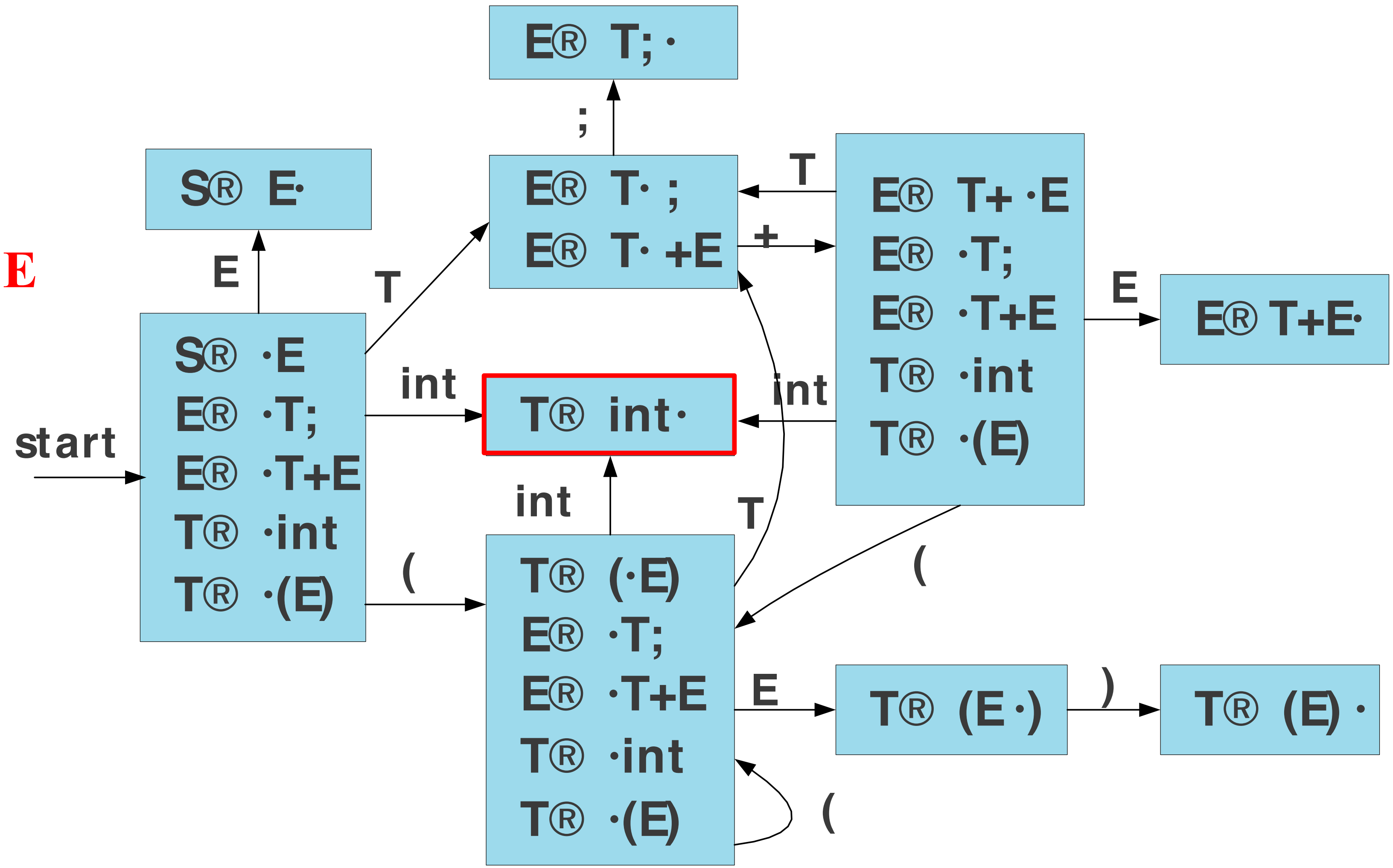
$S \rightarrow E$
$E \rightarrow T;$
$E \rightarrow T + E$
$T \rightarrow \text{int}$
$T \rightarrow (E)$



int	+	(	int	+	int	;	)	;	

LR(0) Parsing

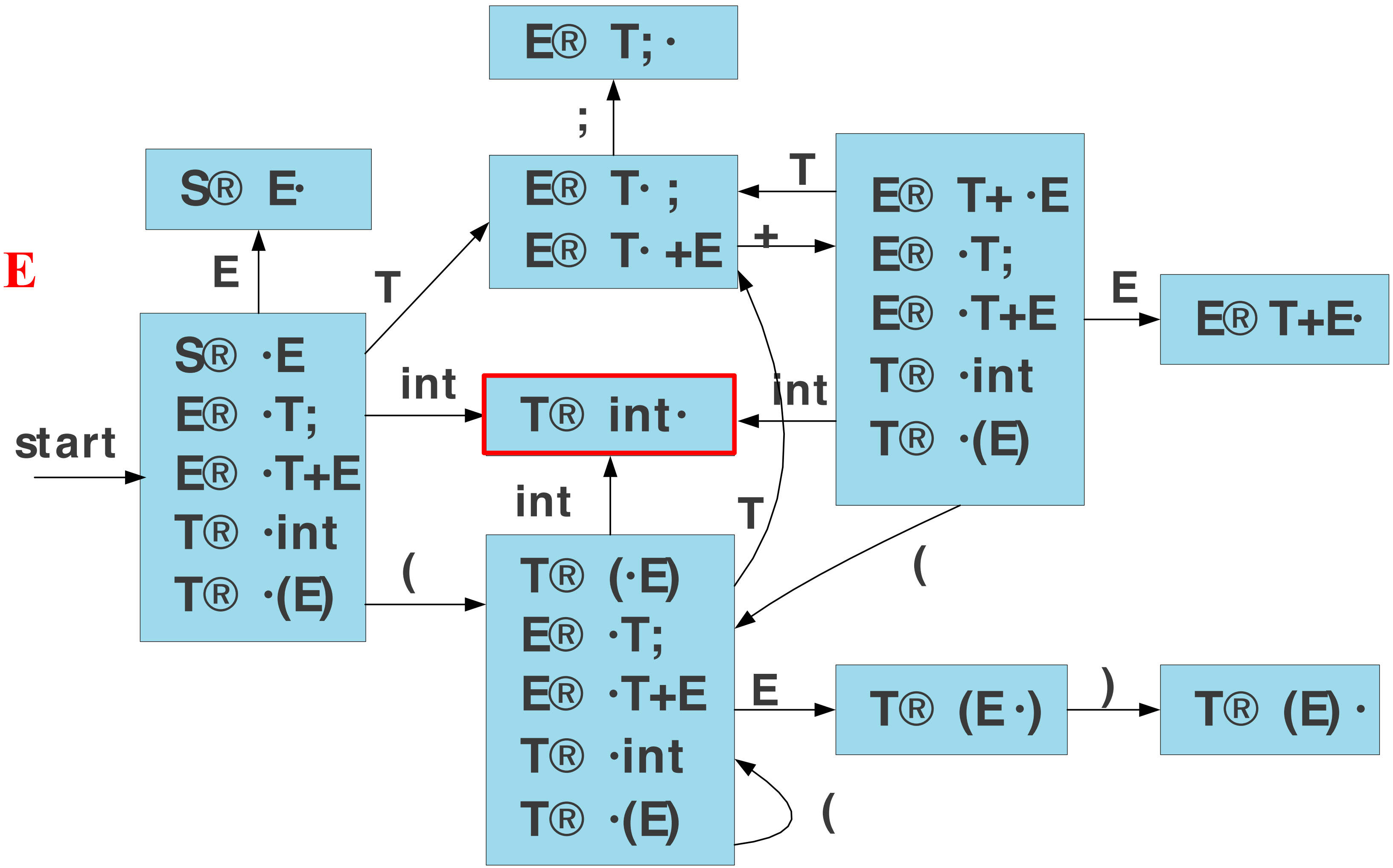
$S \rightarrow E$
$E \rightarrow T;$
$E \rightarrow T + E$
$T \rightarrow \text{int}$
$T \rightarrow (E)$



int	+	(	int	+	int	;	)	;

LR(0) Parsing

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



+	(	int	+	int	;	)	;

LR(0) Parsing

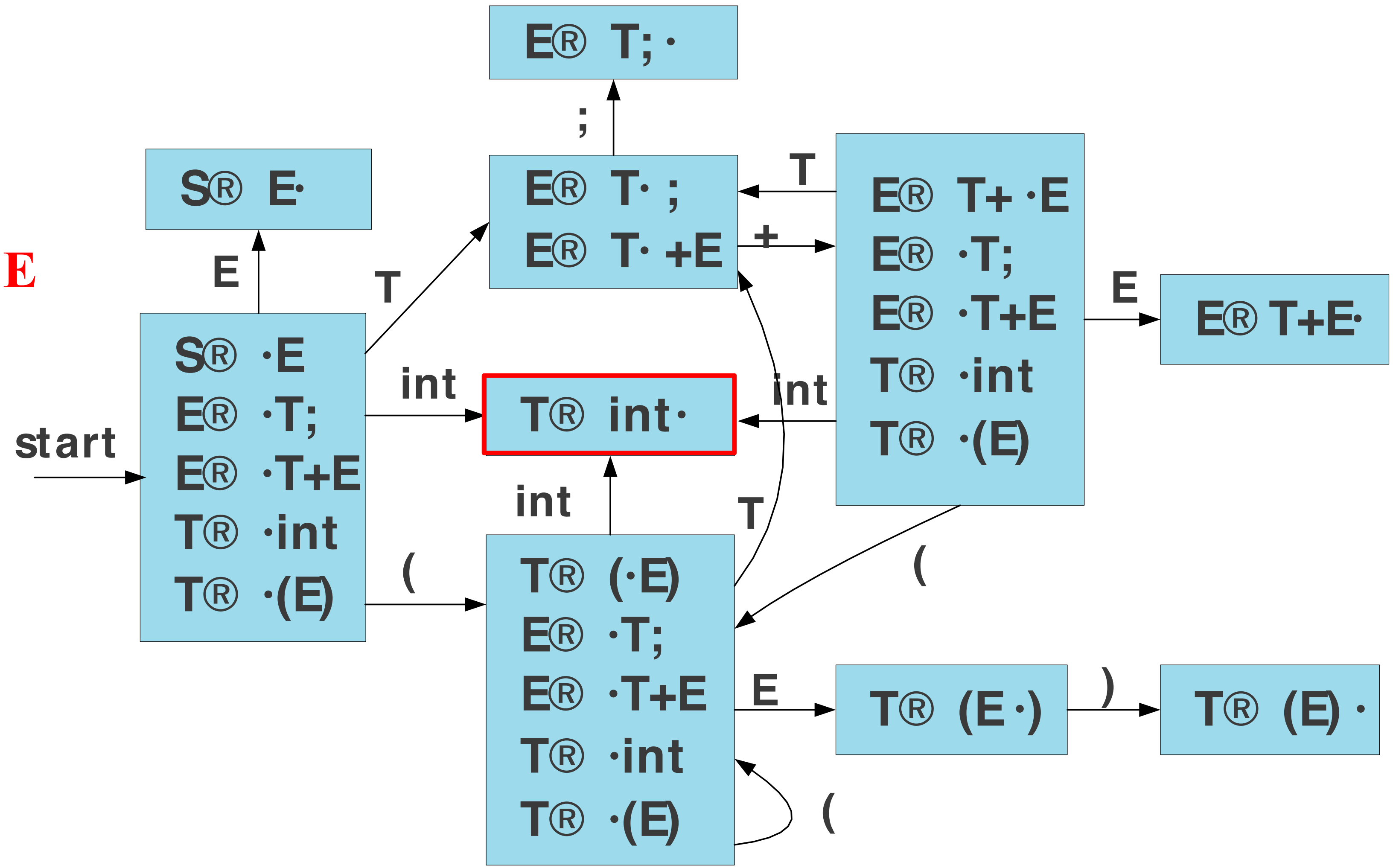
$S \rightarrow E$

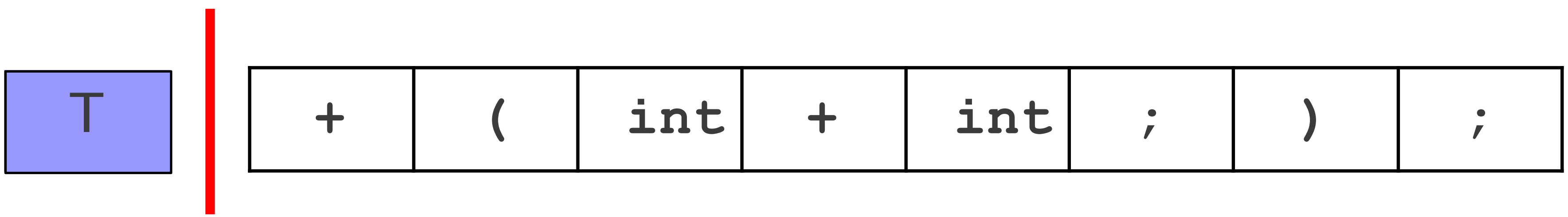
$E \rightarrow T;$

$E \rightarrow T + E$

$T \rightarrow \text{int}$

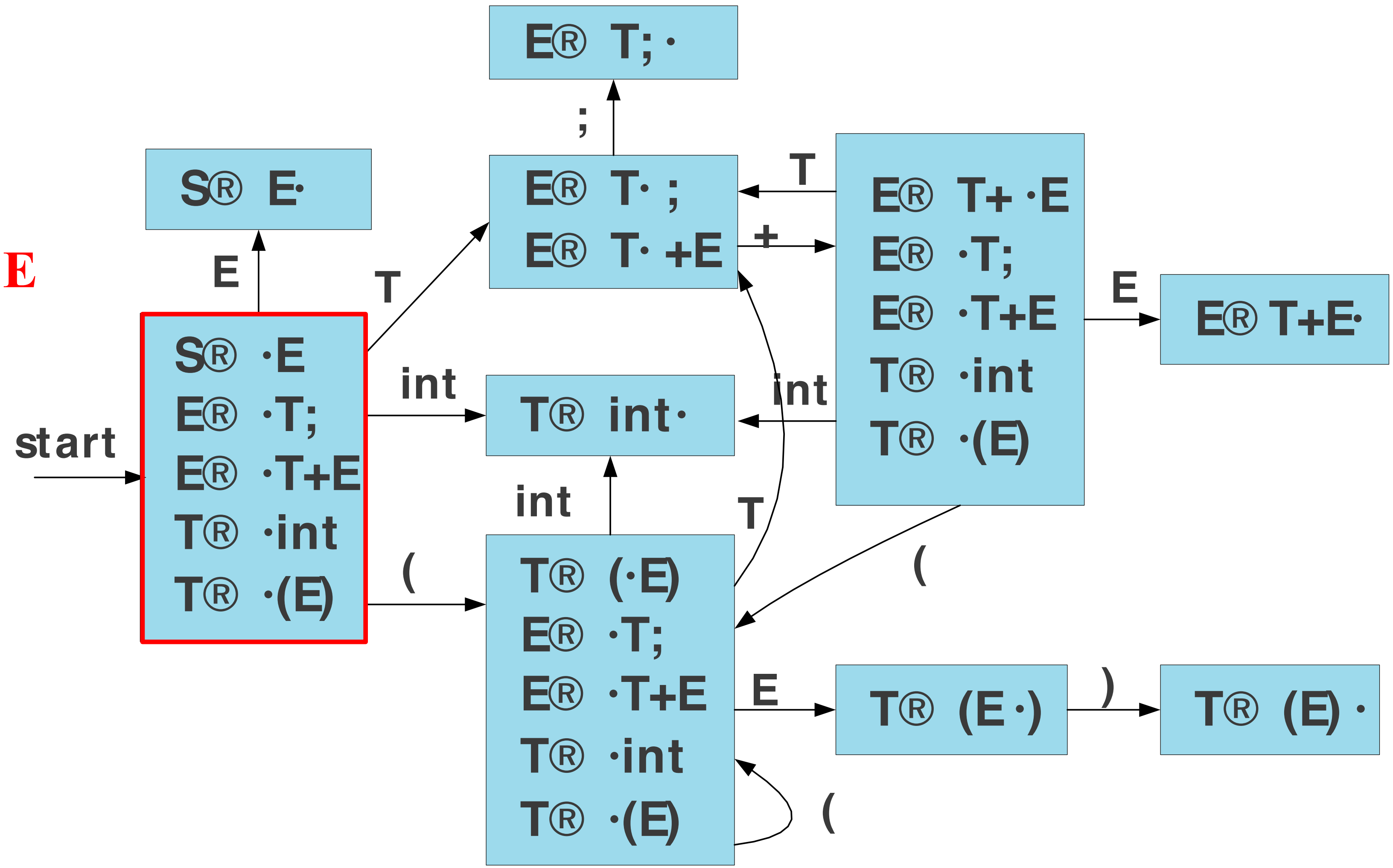
$T \rightarrow (E)$





LR(0) Parsing

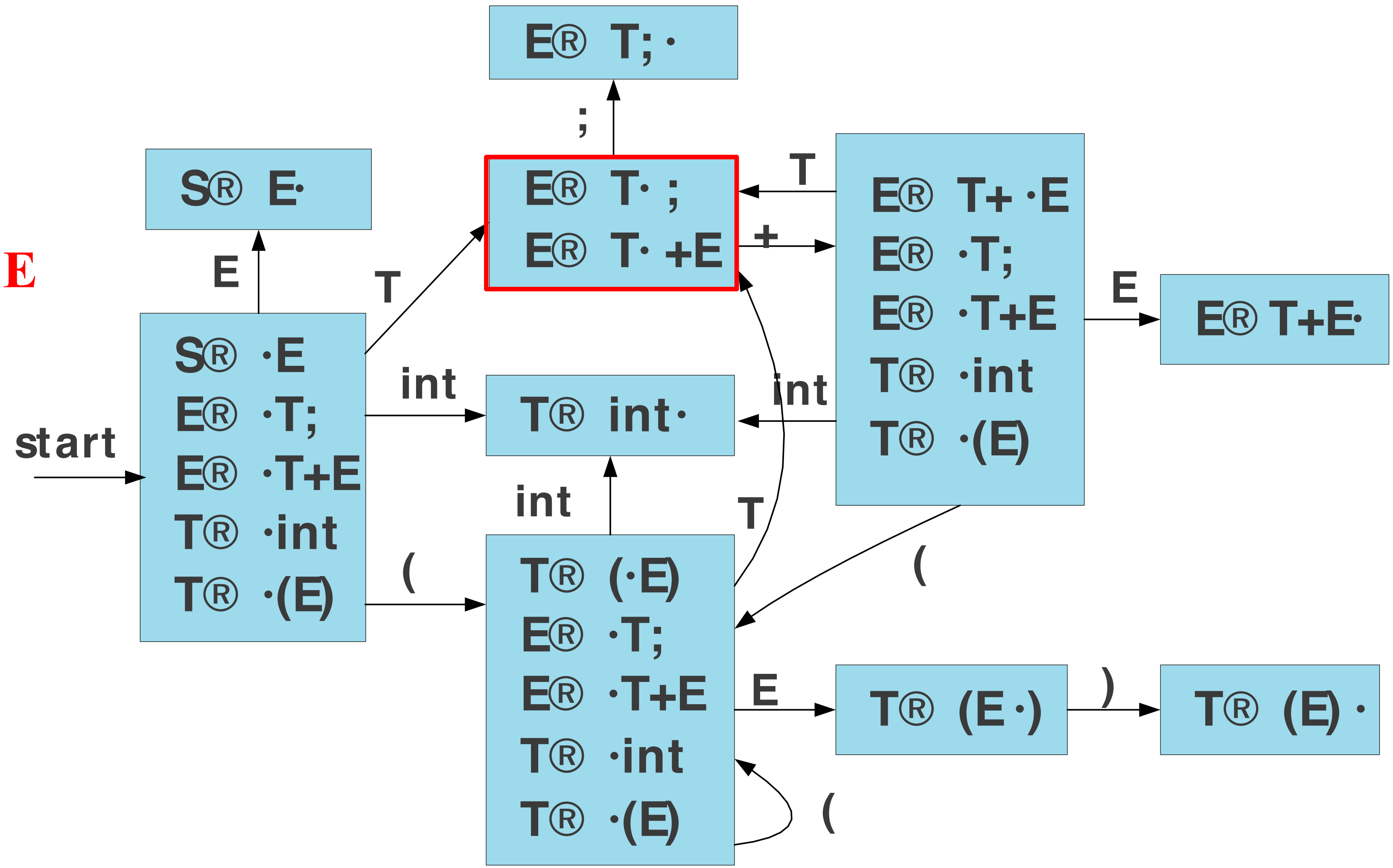
$S \rightarrow E$
$E \rightarrow T;$
$E \rightarrow T + E$
$T \rightarrow \text{int}$
$T \rightarrow (E)$



T	+	(	int	+	int	;	)	;

LR(0) Parsing

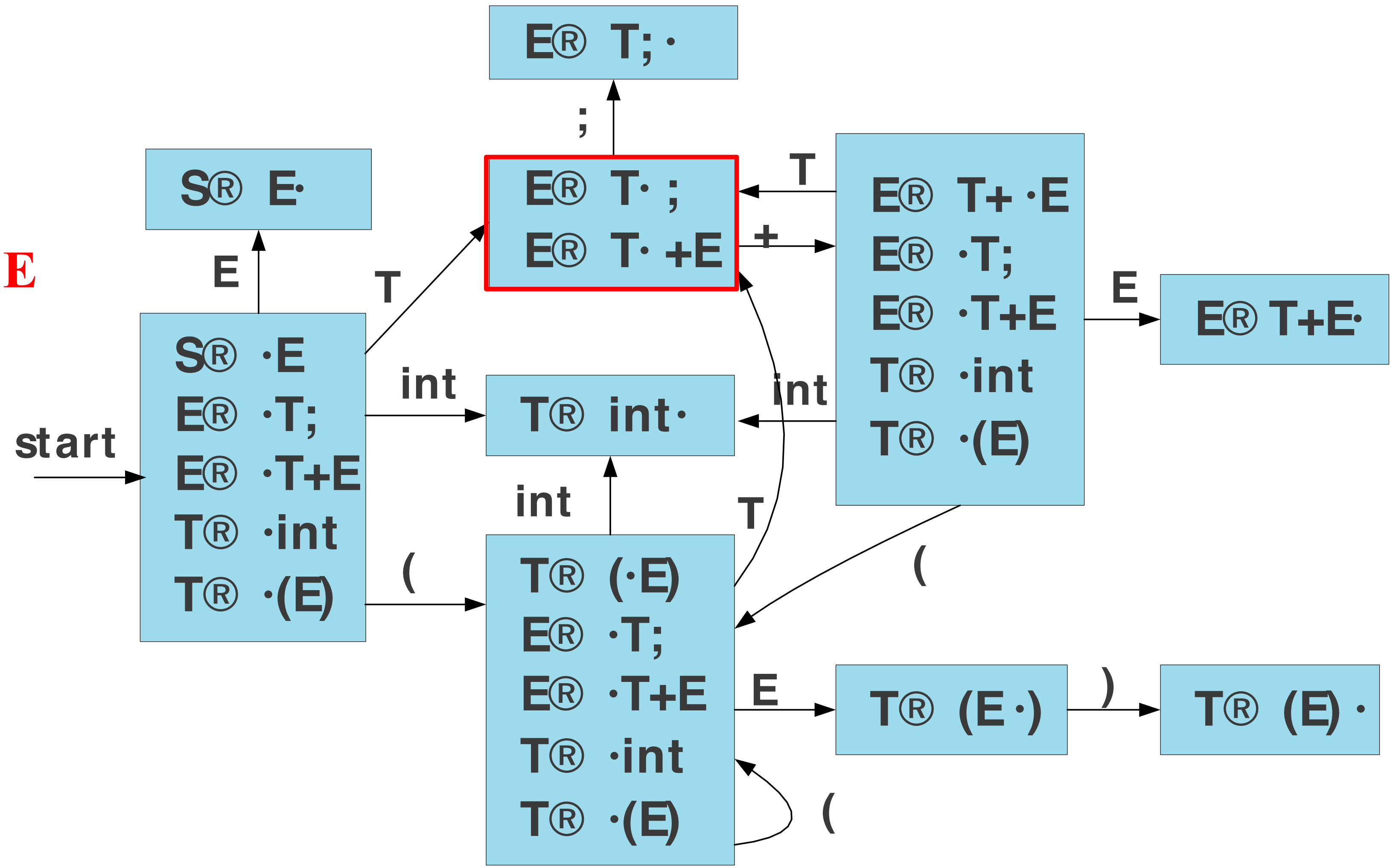
$S \rightarrow E$
$E \rightarrow T;$
$E \rightarrow T + E$
$T \rightarrow \text{int}$
$T \rightarrow (E)$



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

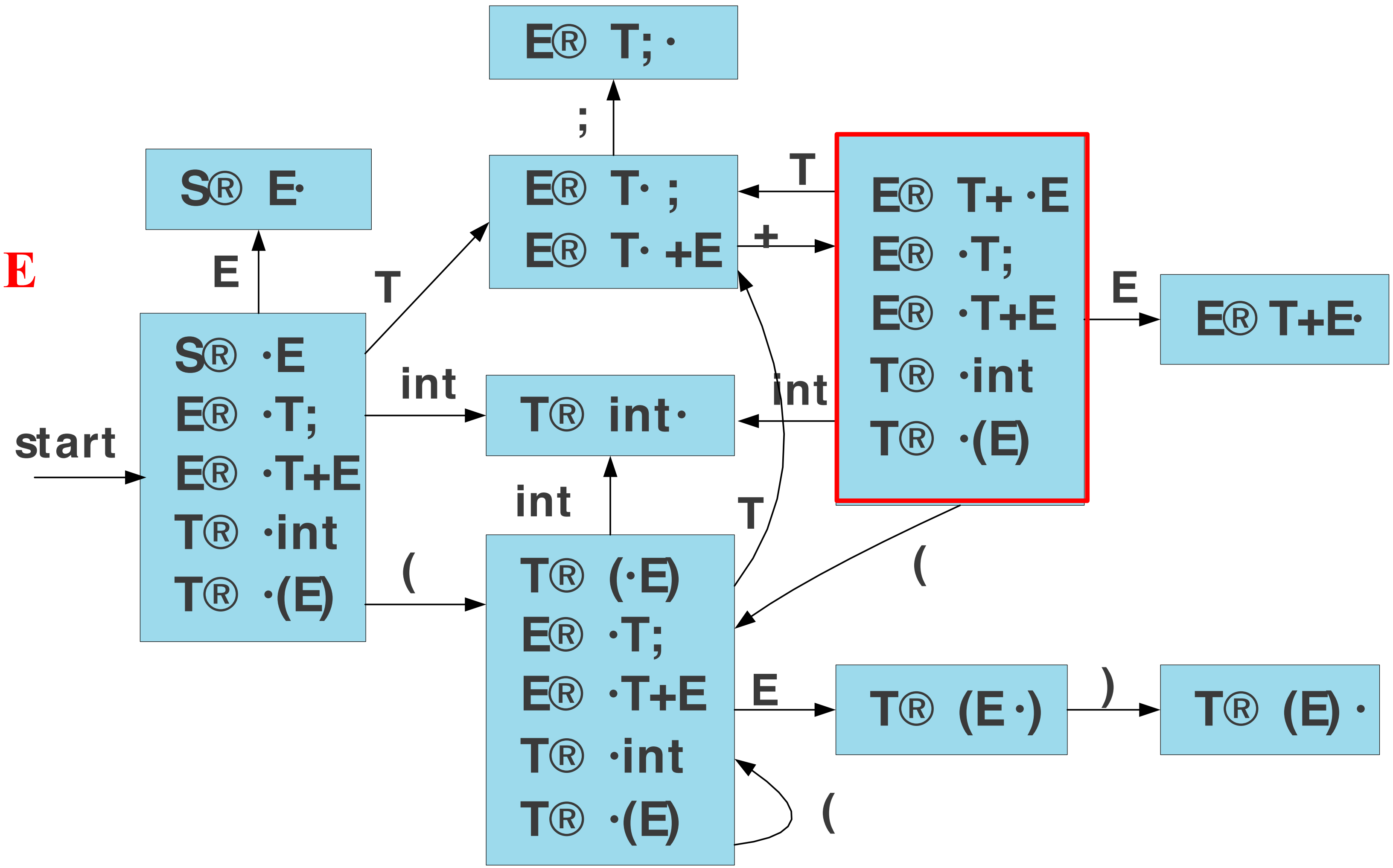
$S \rightarrow E$

$E \rightarrow T;$

$E \rightarrow T + E$

$T \rightarrow \text{int}$

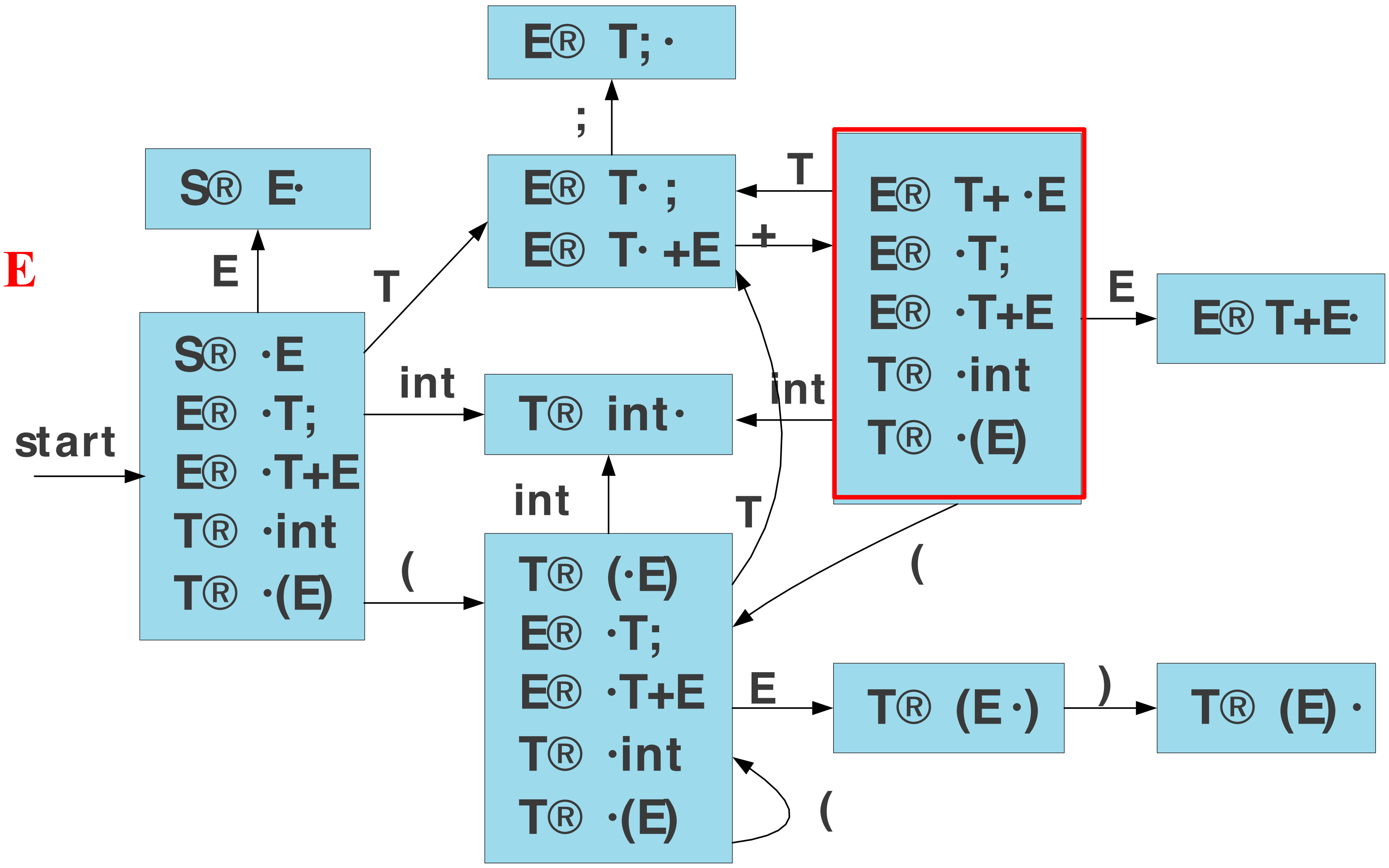
$T \rightarrow (E)$

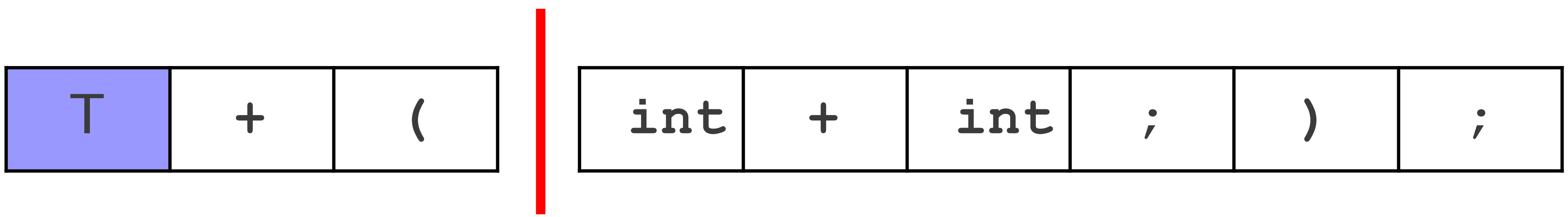


T	+	(	int	+	int	;	)	;

LR(0) Parsing

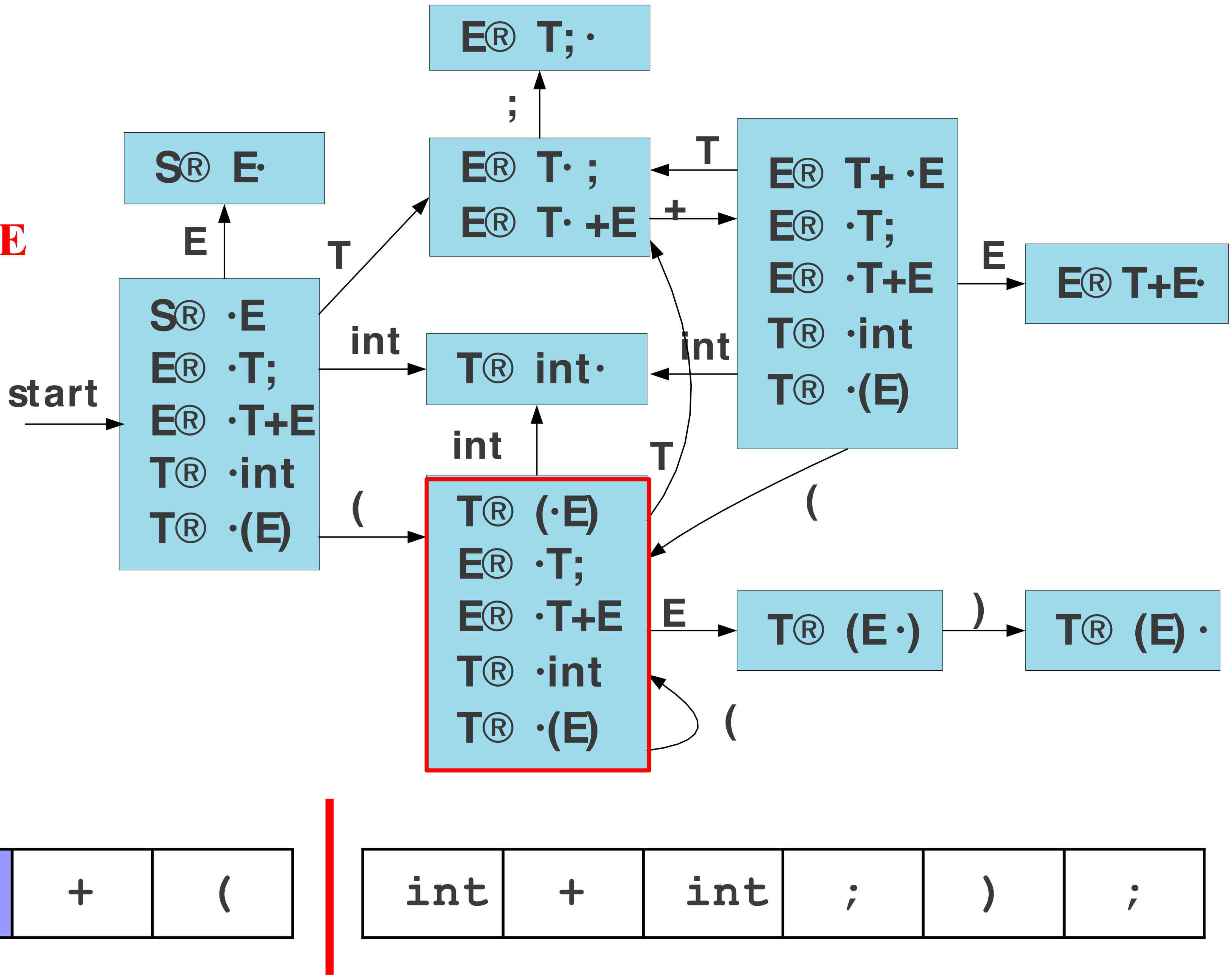
$S \rightarrow E$
$E \rightarrow T;$
$E \rightarrow T + E$
$T \rightarrow int$
$T \rightarrow (E)$





LR(0) Parsing

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



LR(0) Parsing

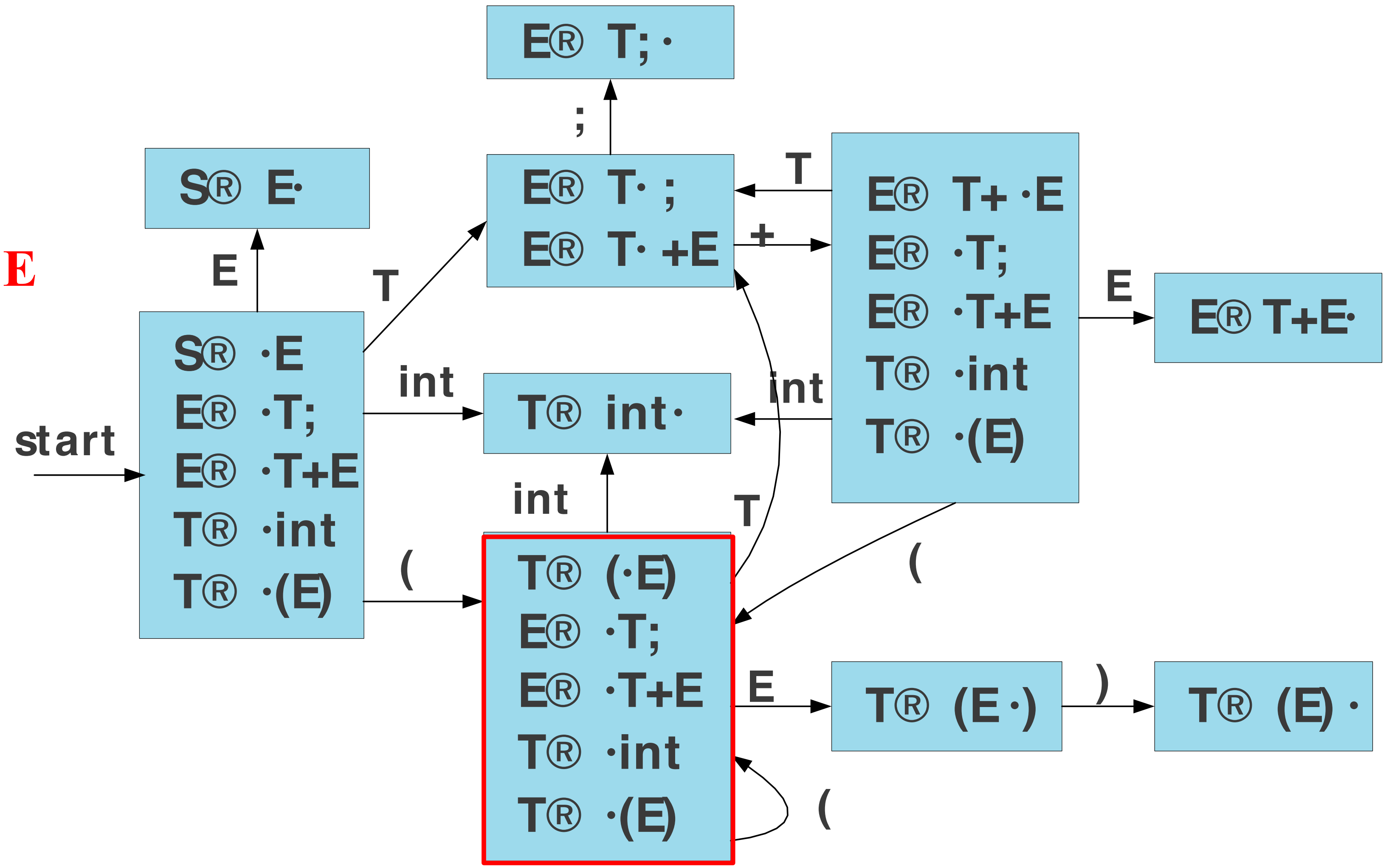
$S \rightarrow E$

$E \rightarrow T;$

$E \rightarrow T + E$

$T \rightarrow \texttt{int}$

$T \rightarrow (E)$



T	+	(	int

+	int	;	)	;

LR(0) Parsing

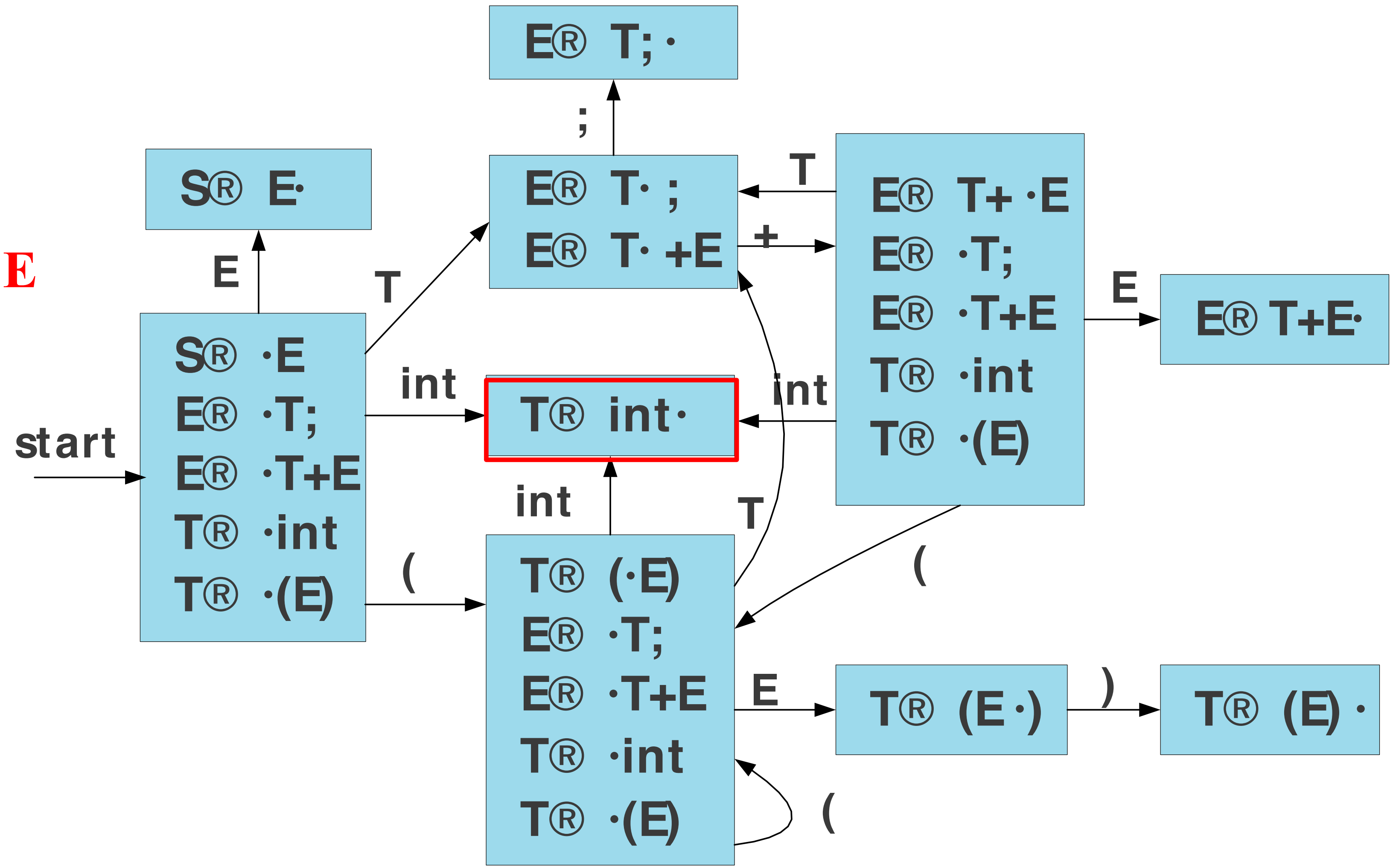
$S \rightarrow E$

$E \rightarrow T;$

$E \rightarrow T + E$

$T \rightarrow \text{int}$

$T \rightarrow (E)$

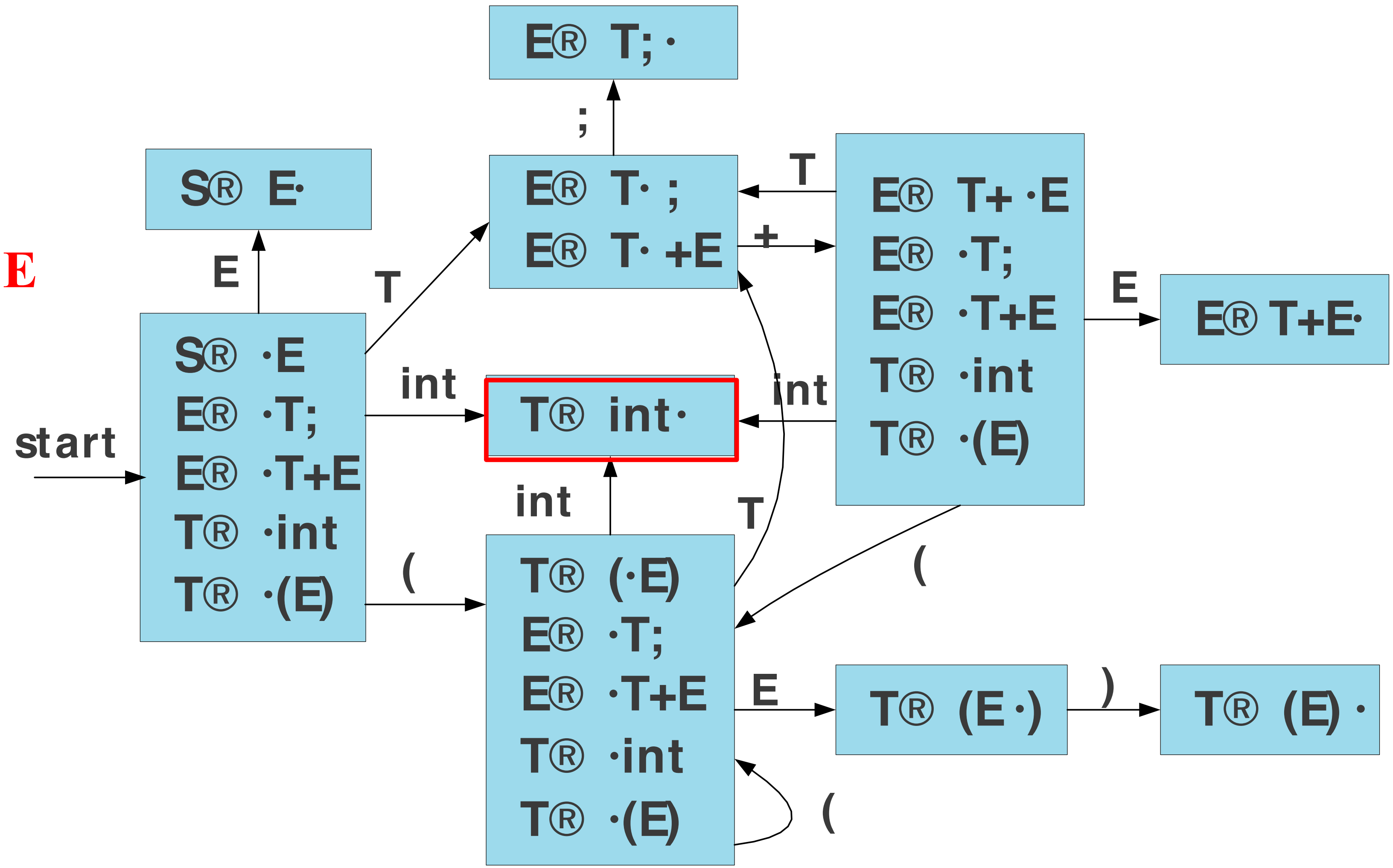


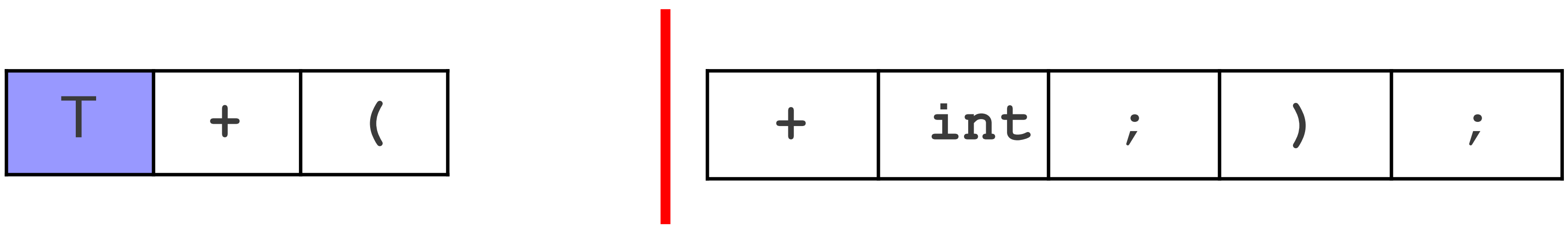
T	+	(	int

+	int	;	)	;

LR(0) Parsing

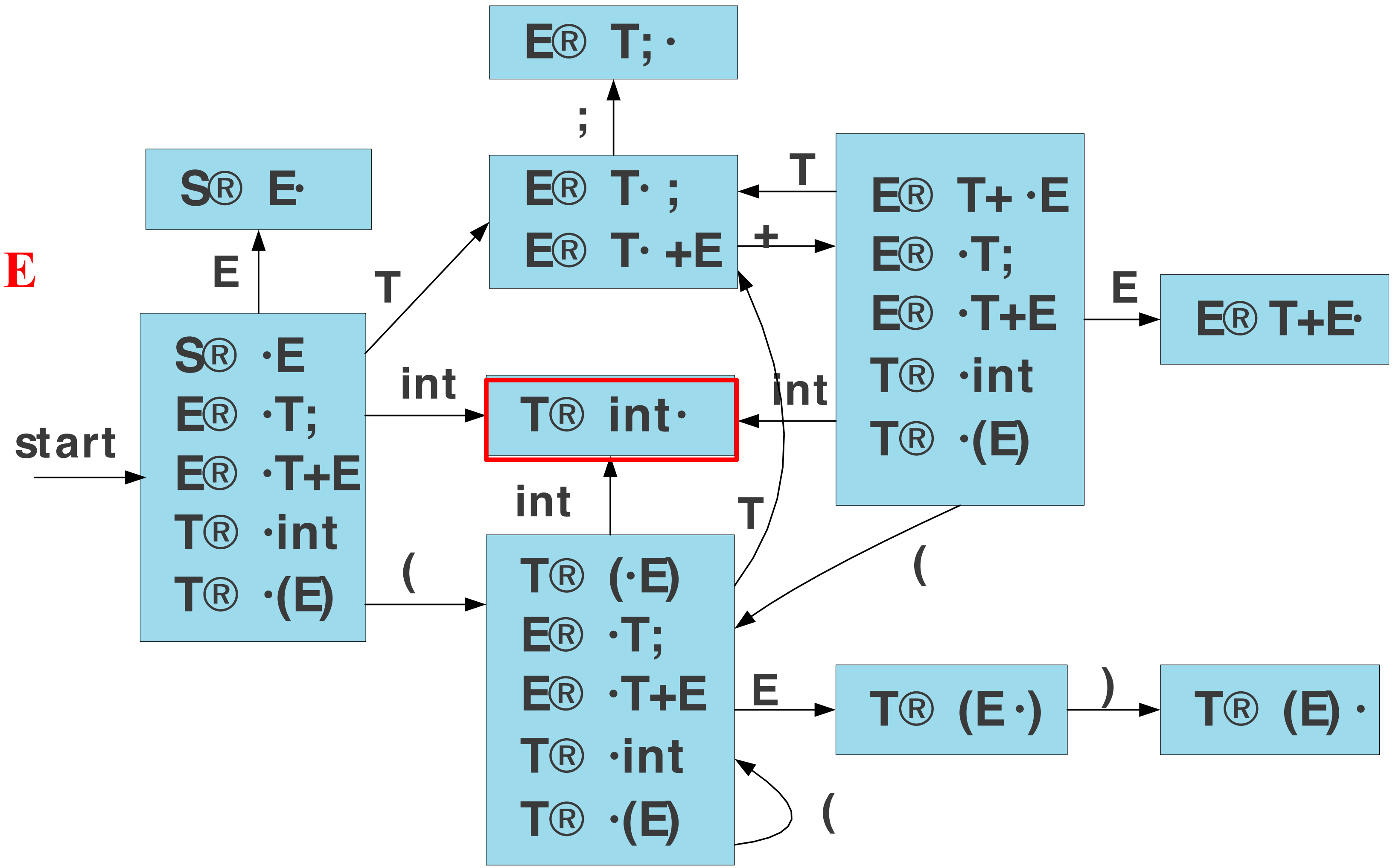
$S \rightarrow E$
$E \rightarrow T;$
$E \rightarrow T + E$
$T \rightarrow \text{int}$
$T \rightarrow (E)$





LR(0) Parsing

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

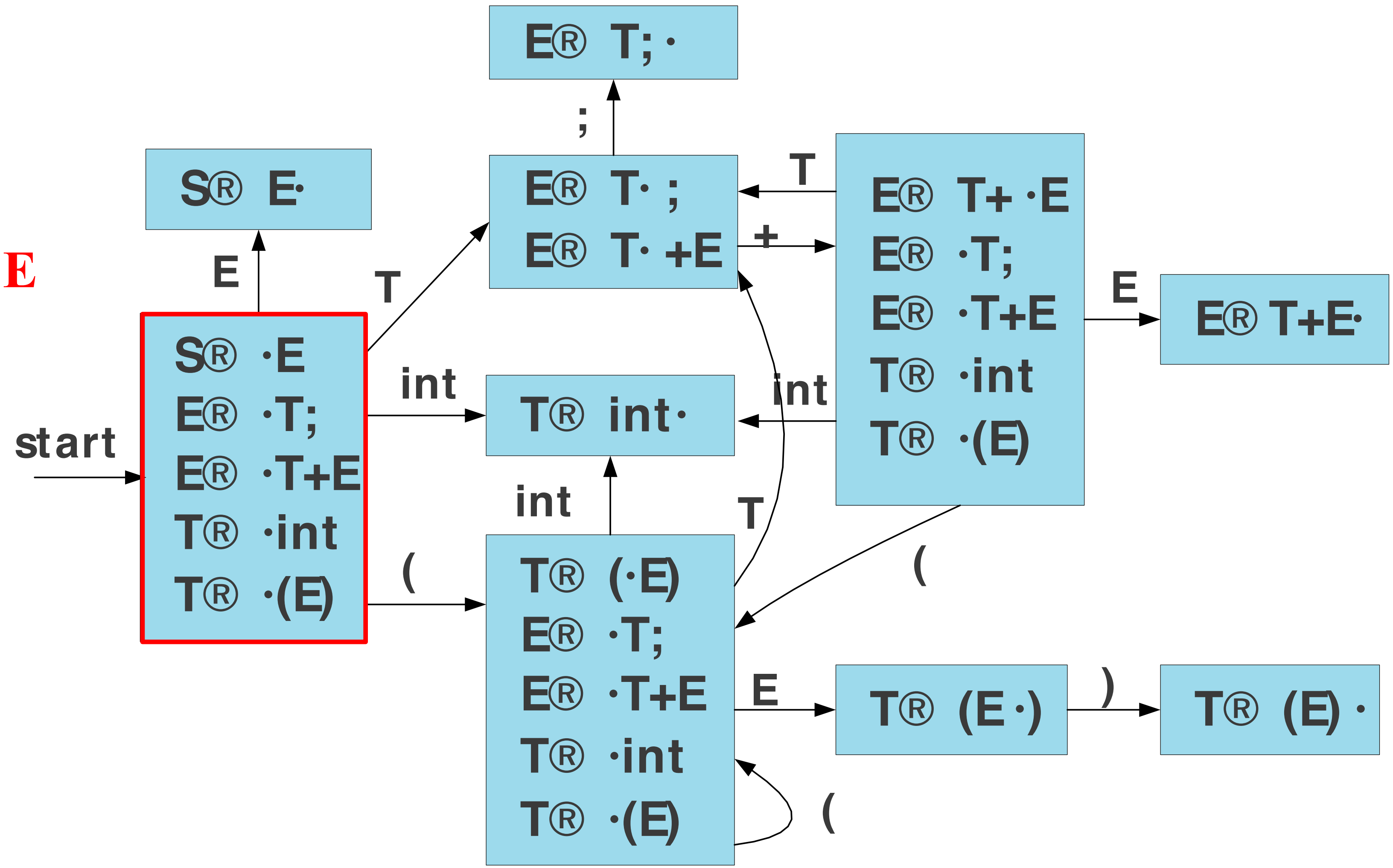


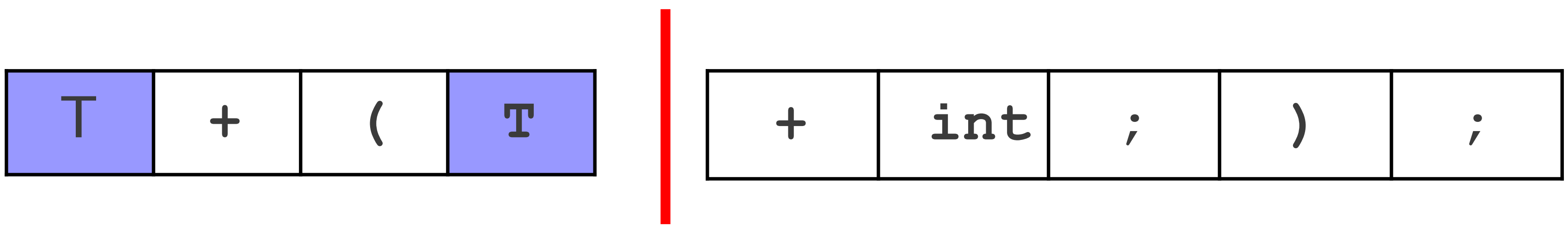
T	+	(	T

+	int	;	)	;

LR(0) Parsing

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





LR(0) Parsing

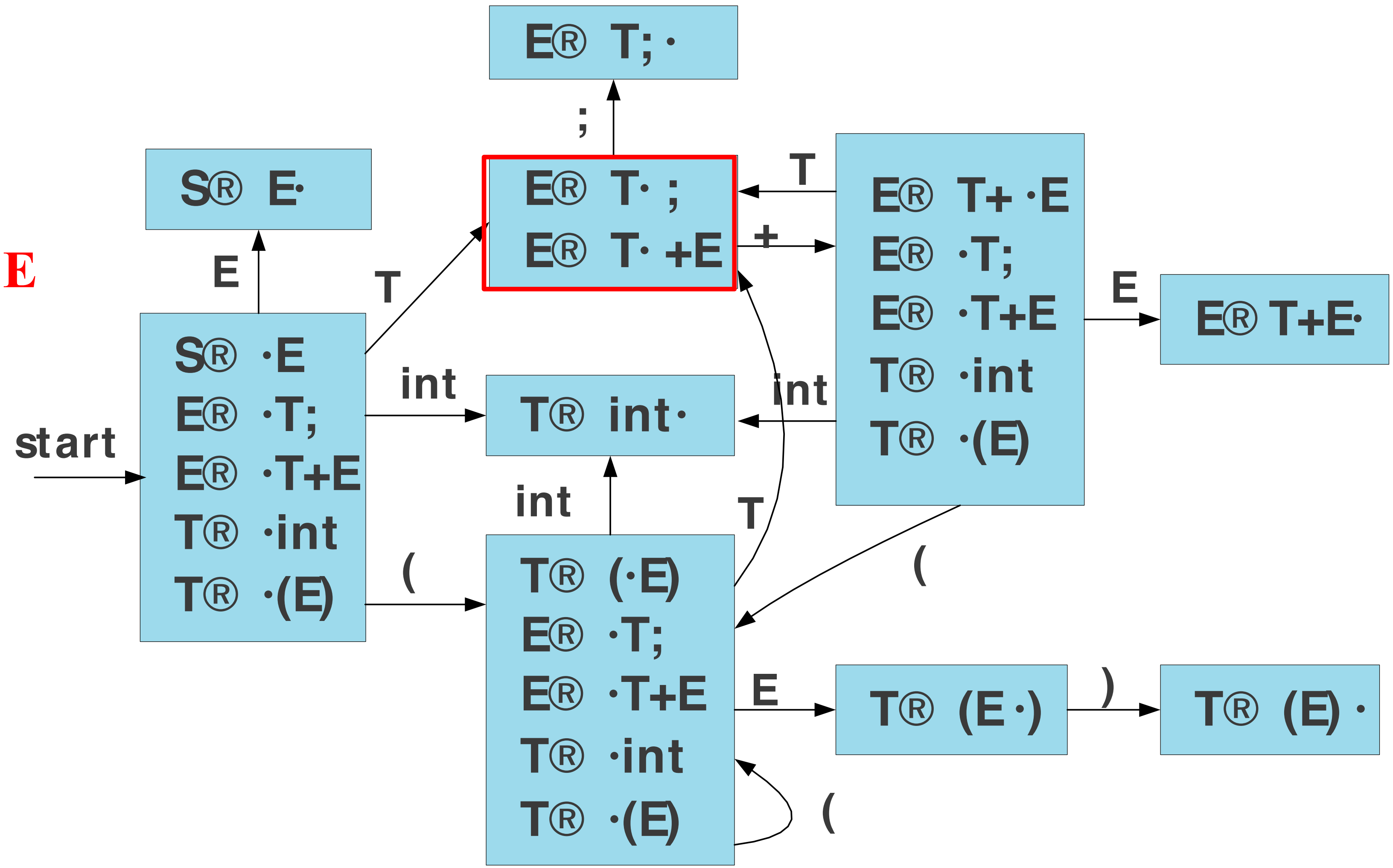
$S \rightarrow E$

$E \rightarrow T;$

$E \rightarrow T + E$

$T \rightarrow \mathrm{int}$

$T \rightarrow (E)$



T	+	(	T

+	int	;	)	;

LR(0) Parsing

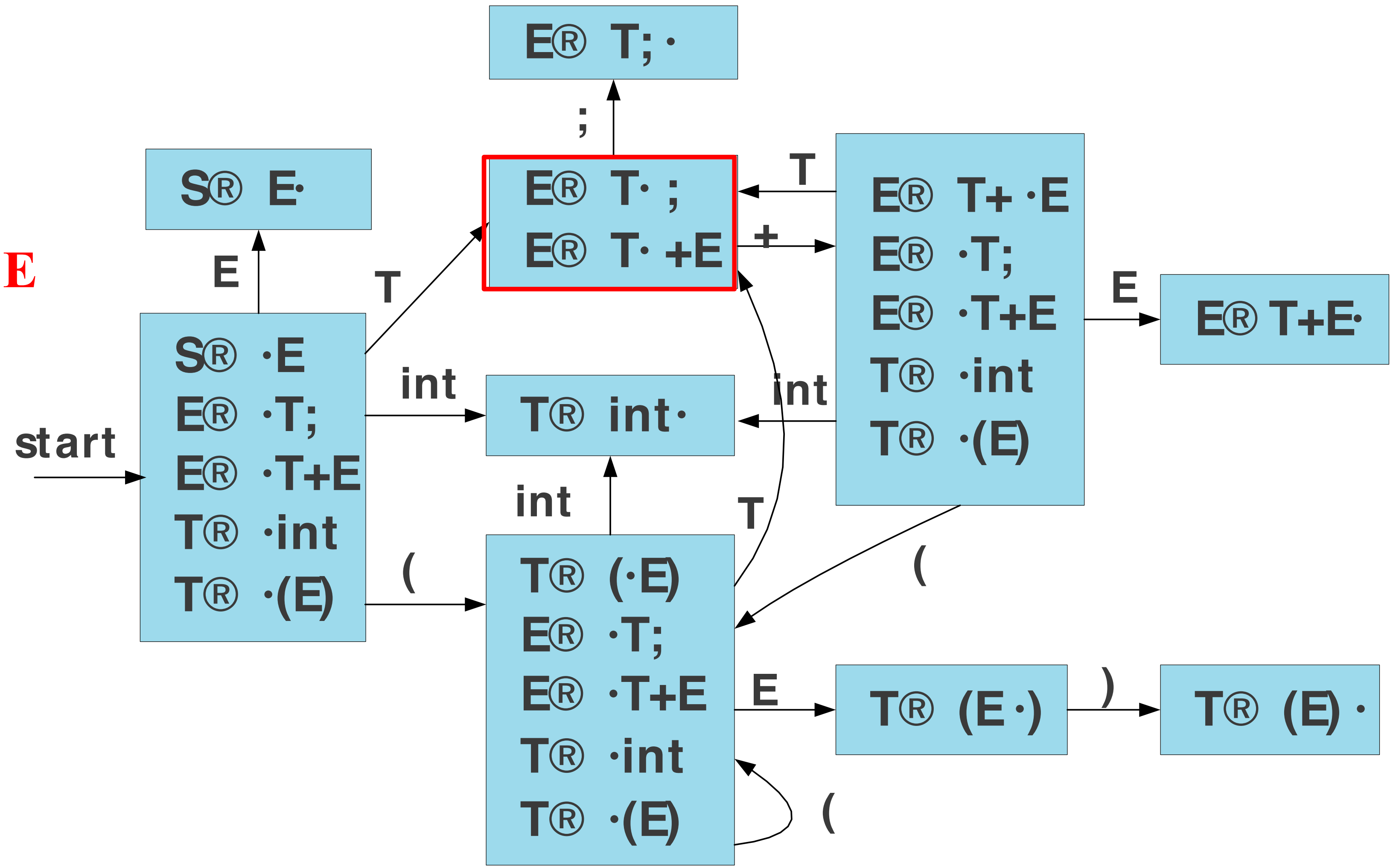
$S \to E$

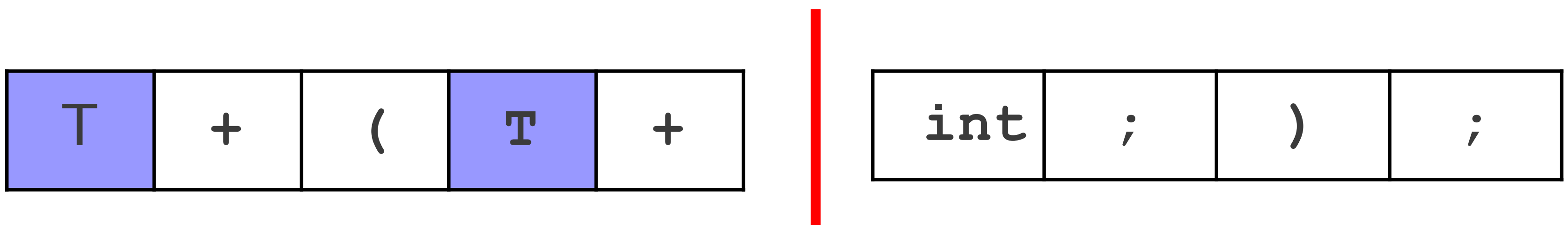
$E \to T;$

$E \to T + E$

$T \to \text{int}$

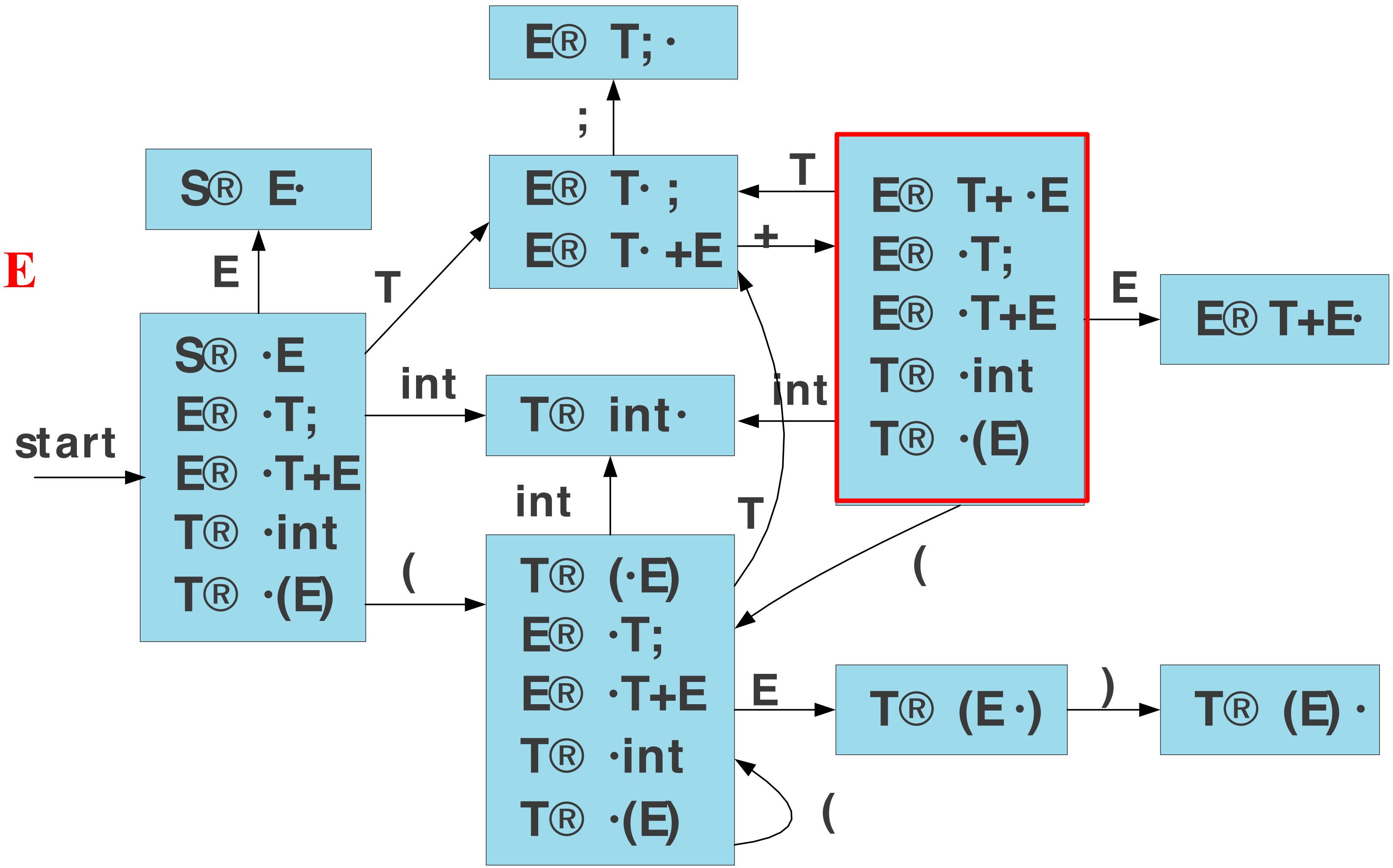
$T \to (E)$





LR(0) Parsing

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

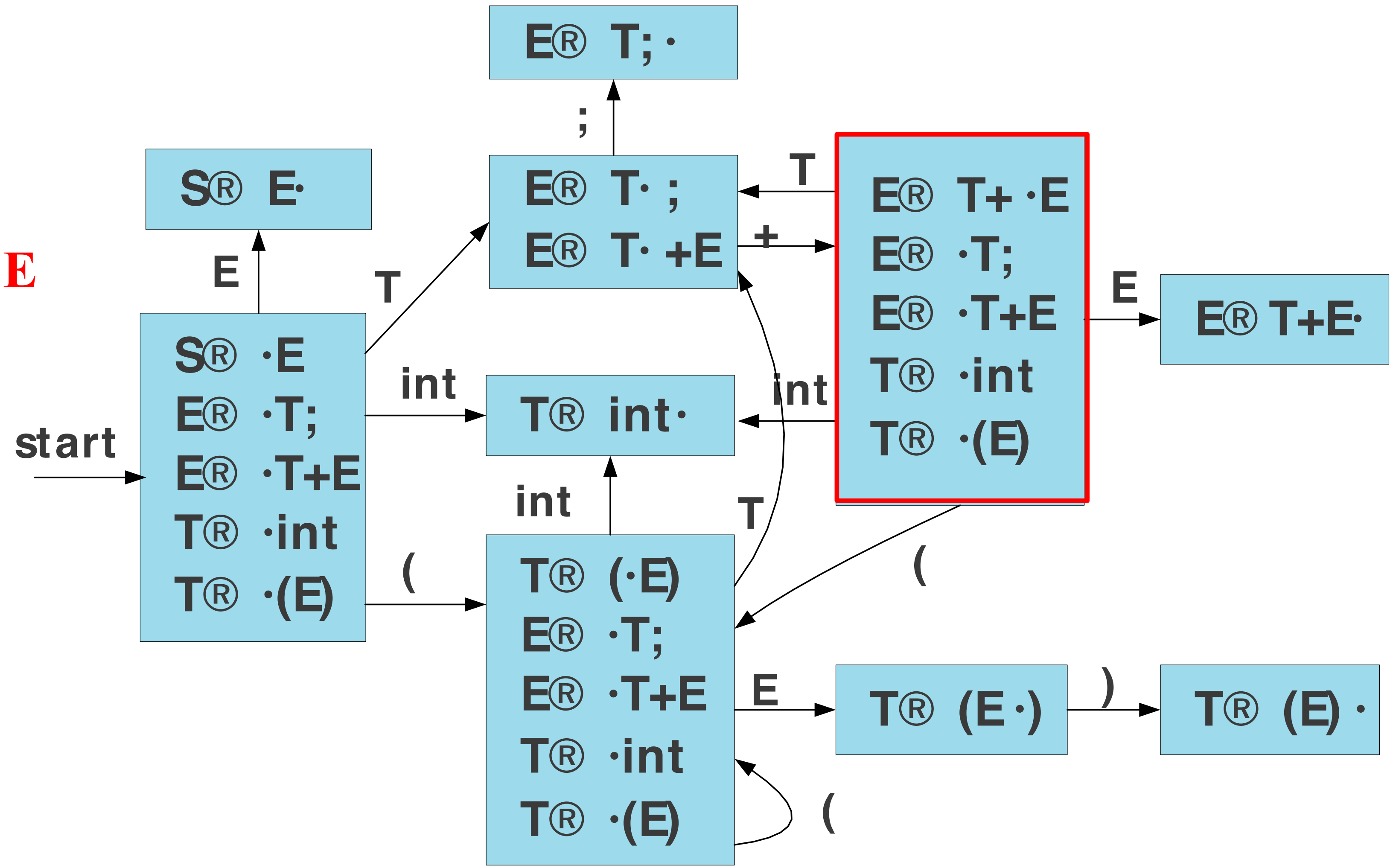


T	+	(	T	+

int	;	)	;

LR(0) Parsing

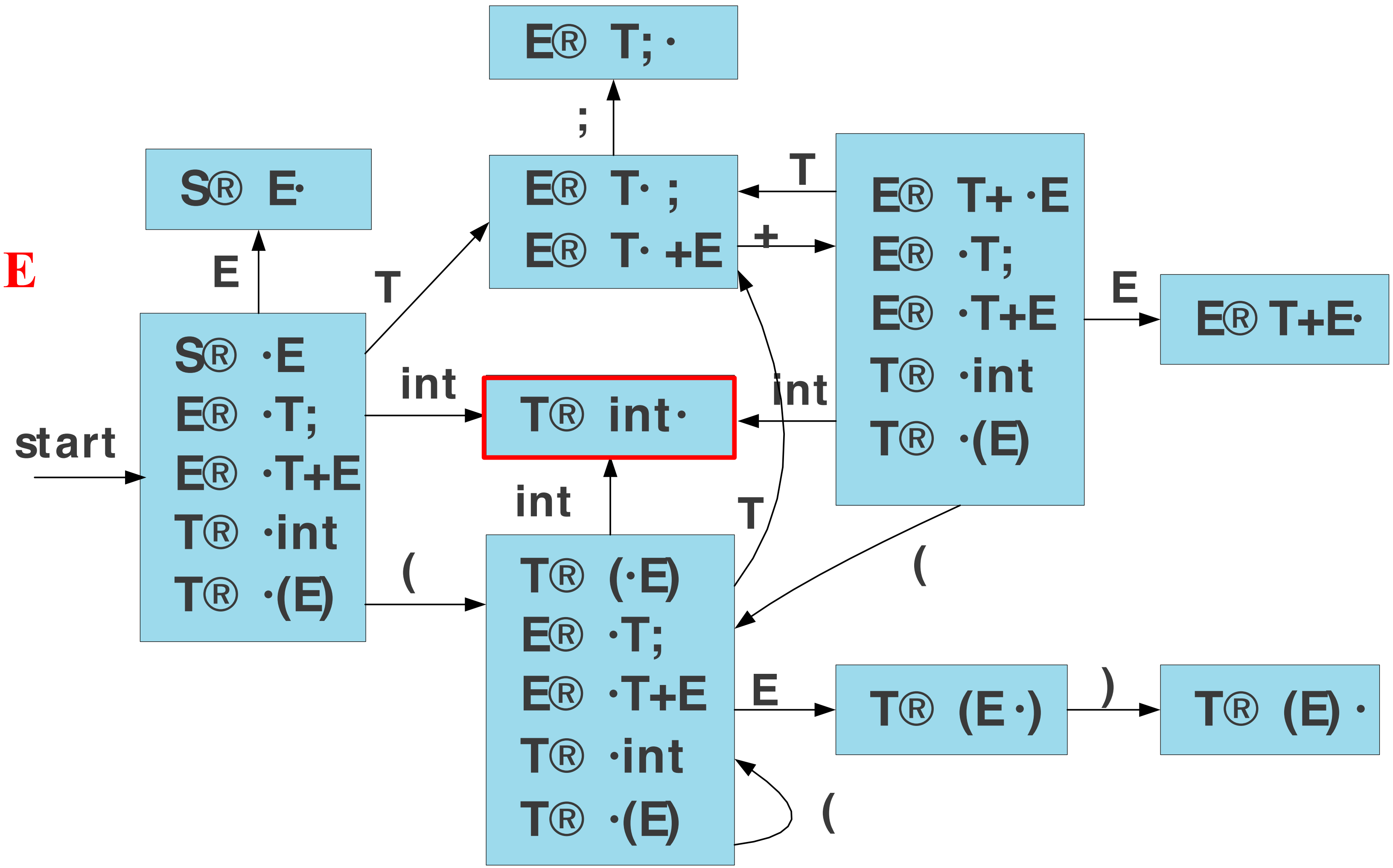
$S \rightarrow E$
$E \rightarrow T;$
$E \rightarrow T + E$
$T \rightarrow \text{int}$
$T \rightarrow (E)$

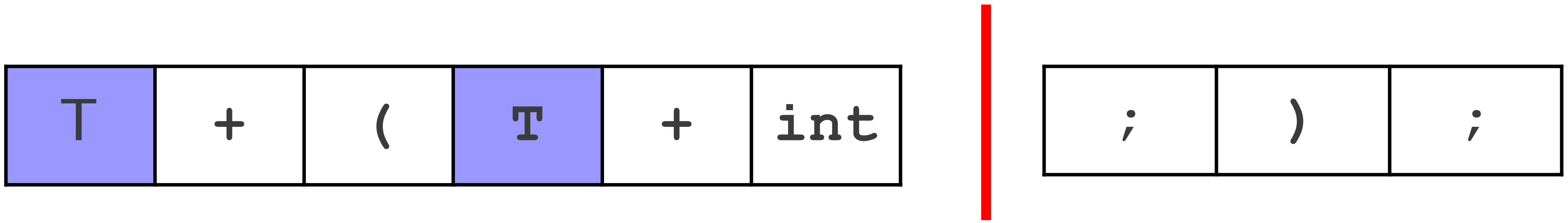


T	+	(	T	+	int		;	)	;

LR(0) Parsing

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





LR(0) Parsing

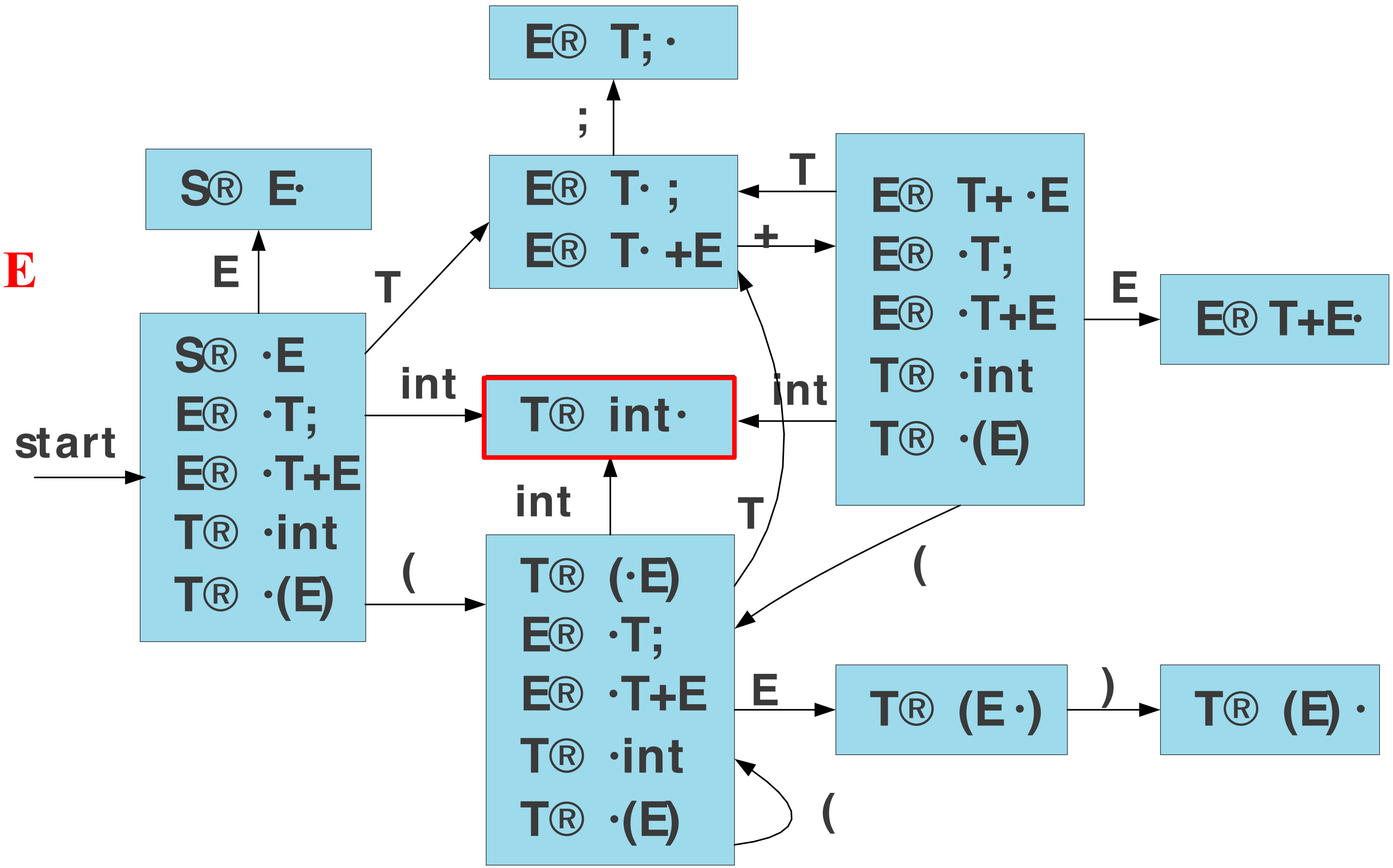
$S \rightarrow E$

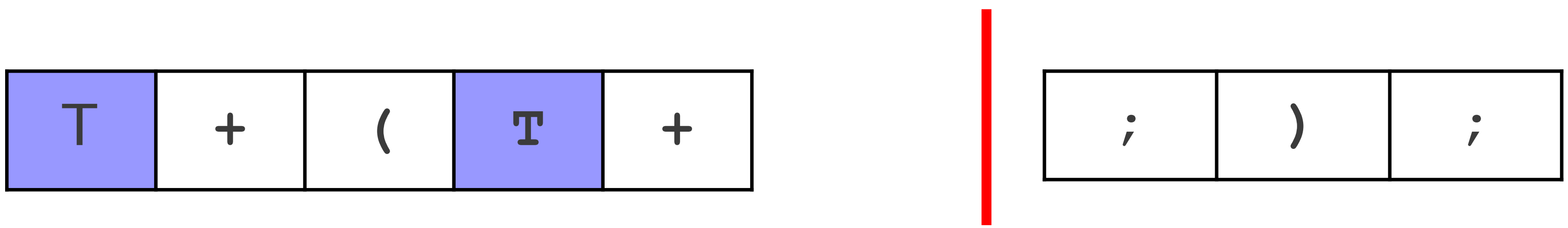
$E \rightarrow T;$

$E \rightarrow T + E$

$T \rightarrow \text{int}$

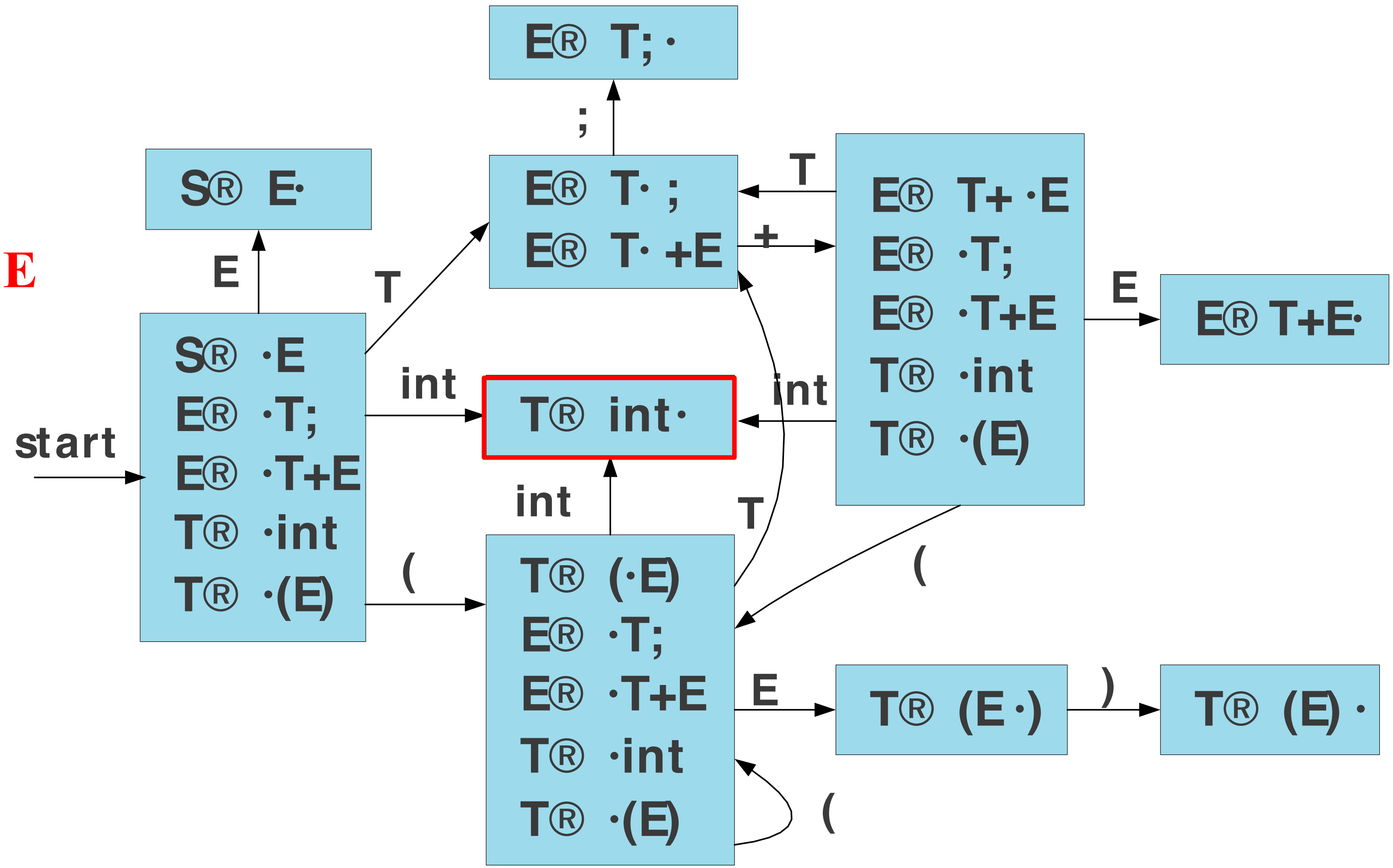
$T \rightarrow (E)$





LR(0) Parsing

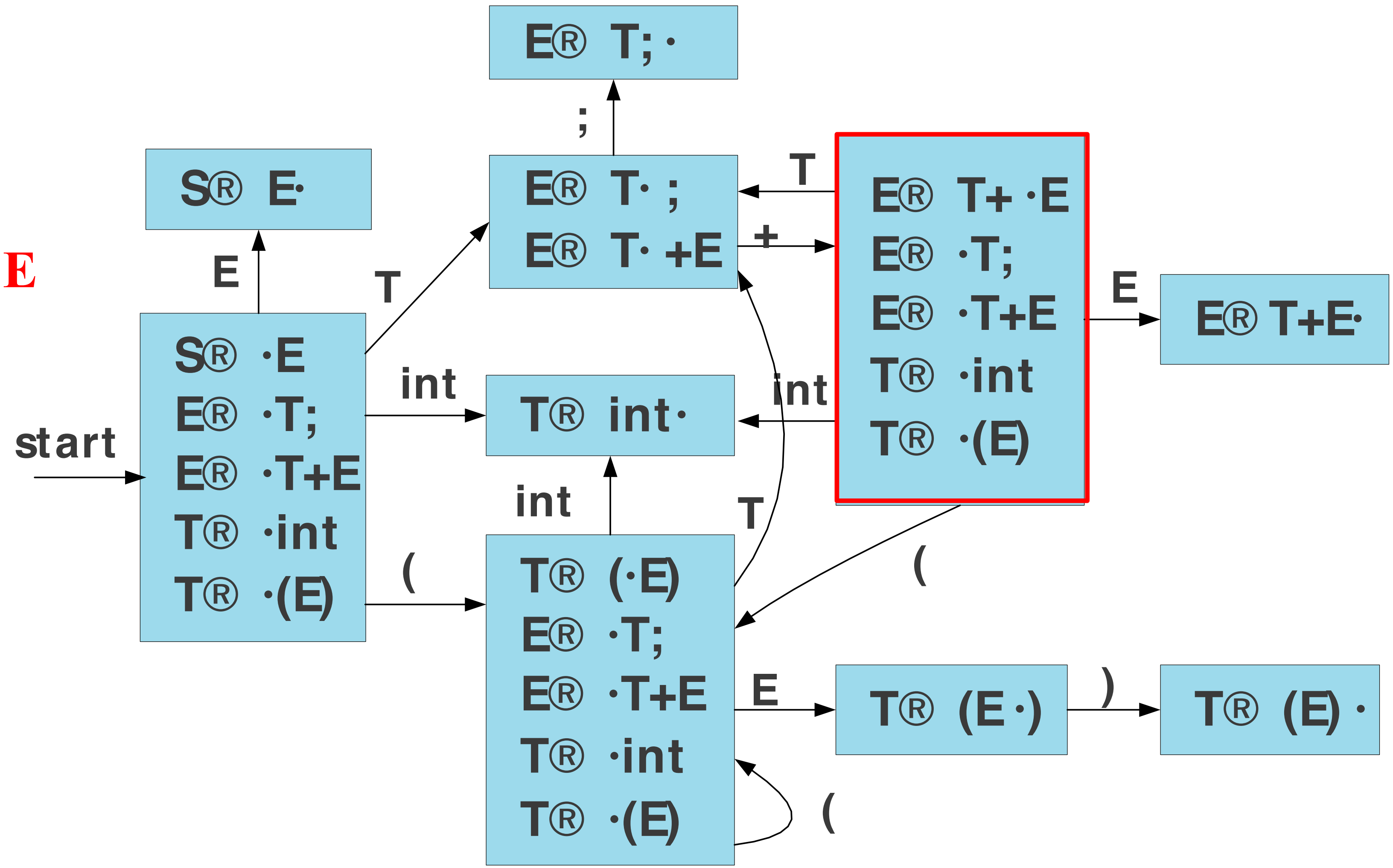
$S \longrightarrow E$
$E \longrightarrow T;$
$E \longrightarrow T + E$
$T \longrightarrow \text{int}$
$T \longrightarrow (E)$

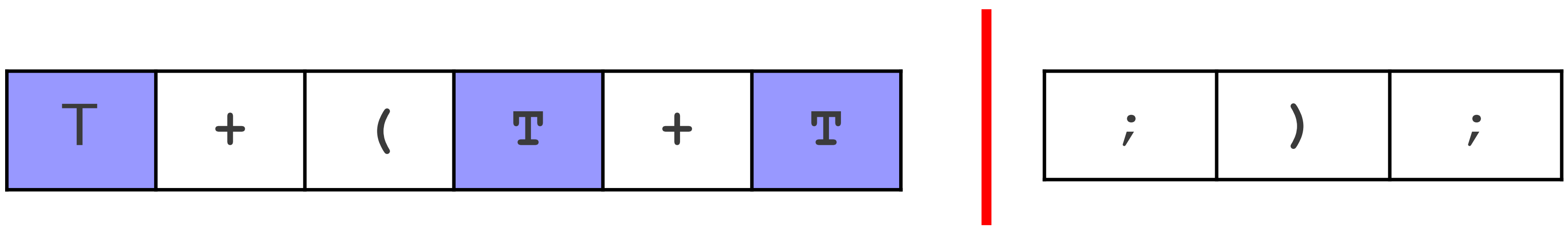


T	+	(	T	+	T	|	;	)	;

LR(0) Parsing

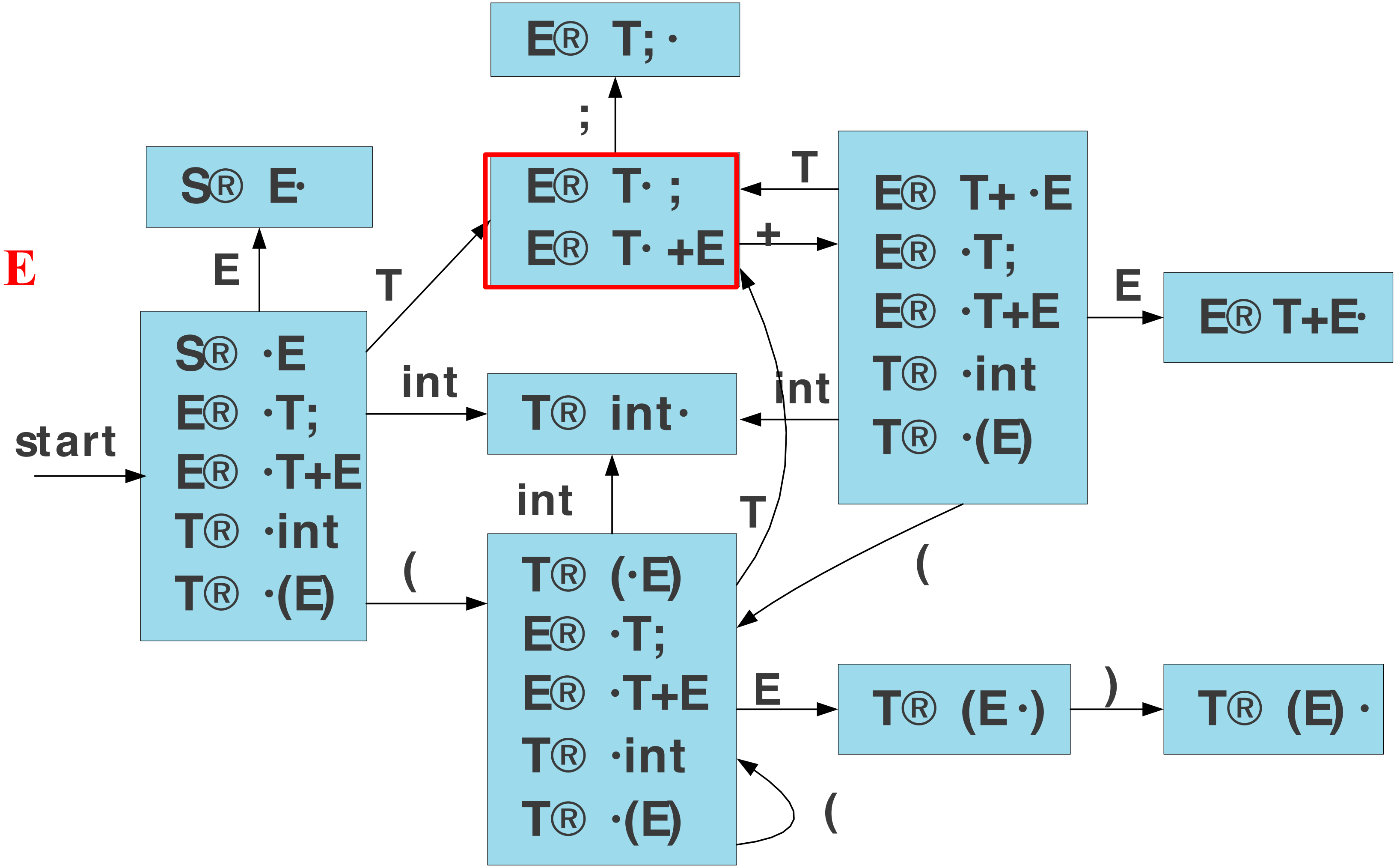
$S \to E$
$E \to T;$
$E \to T + E$
$T \to \text{int}$
$T \to (E)$

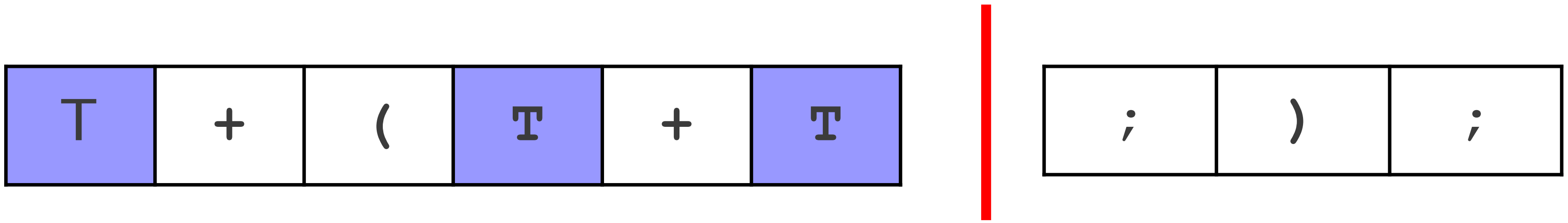




LR(0) Parsing

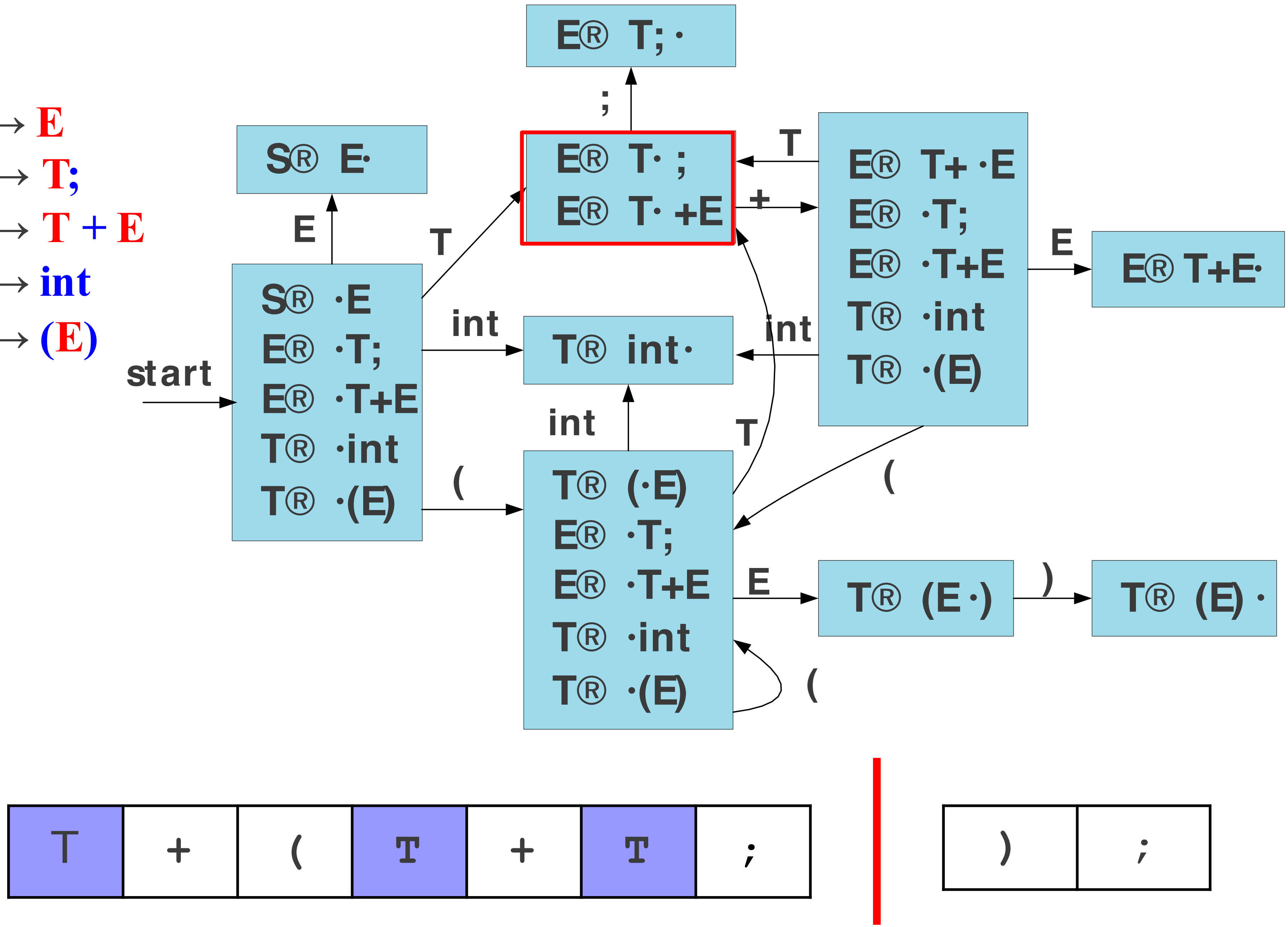
$S \rightarrow E$
$E \rightarrow T;$
$E \rightarrow T + E$
$T \rightarrow \text{int}$
$T \rightarrow (E)$





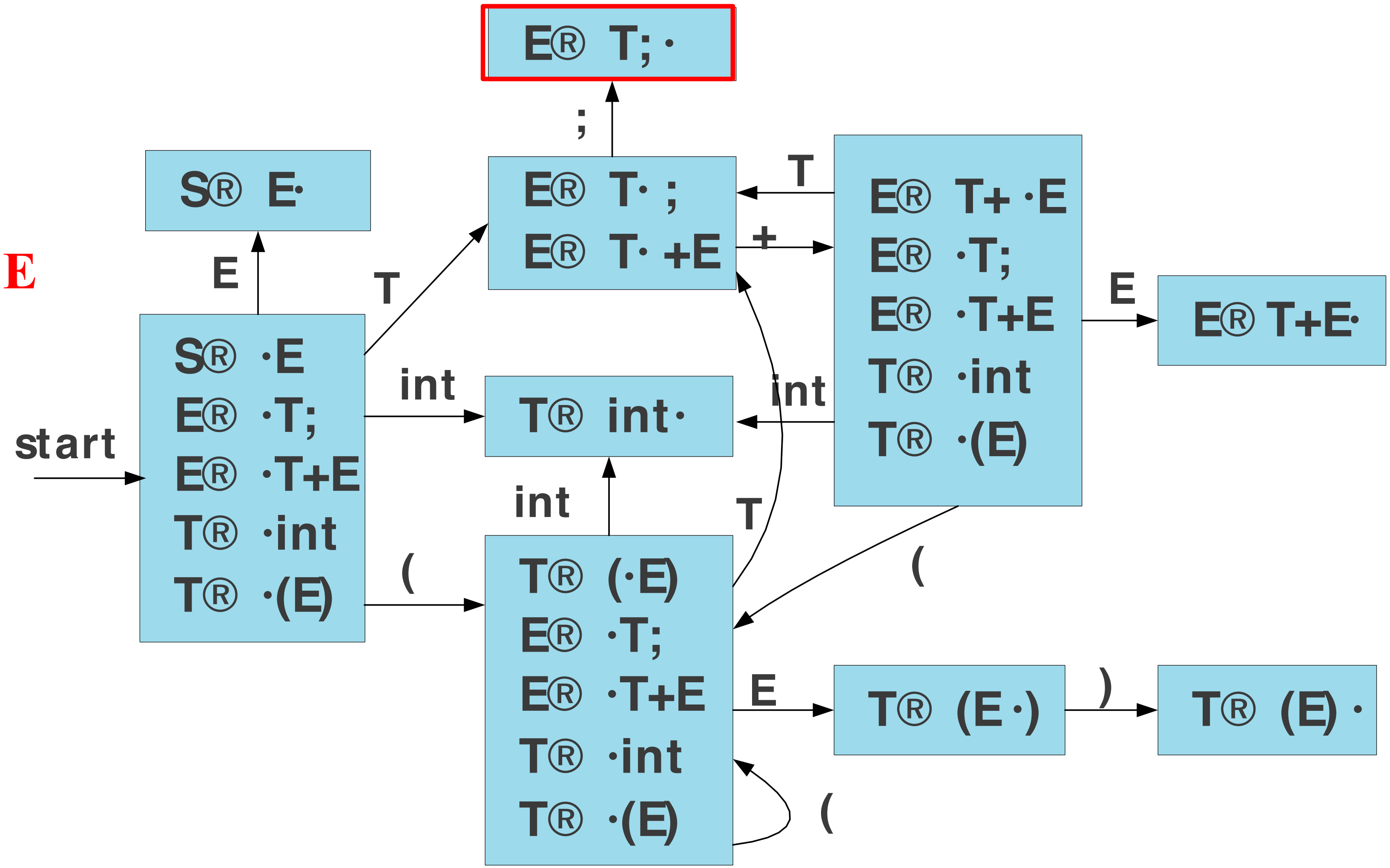
LR(0) Parsing

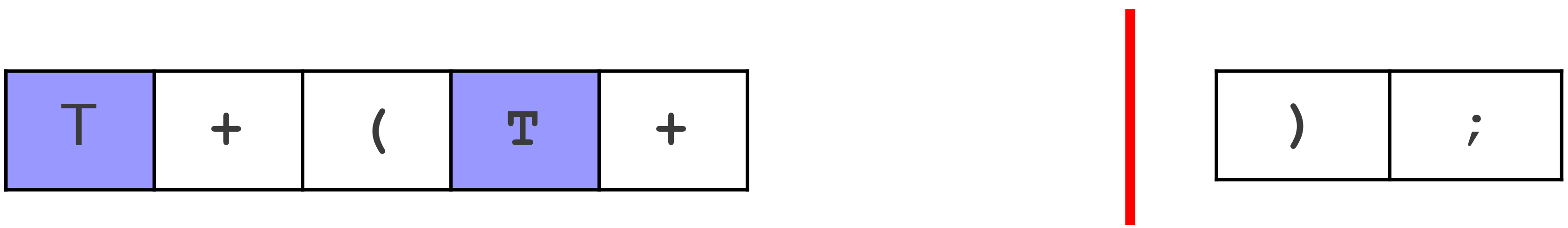
$S \to E$
$E \to T;$
$E \to T + E$
$T \to \text{int}$
$T \to (E)$



LR(0) Parsing

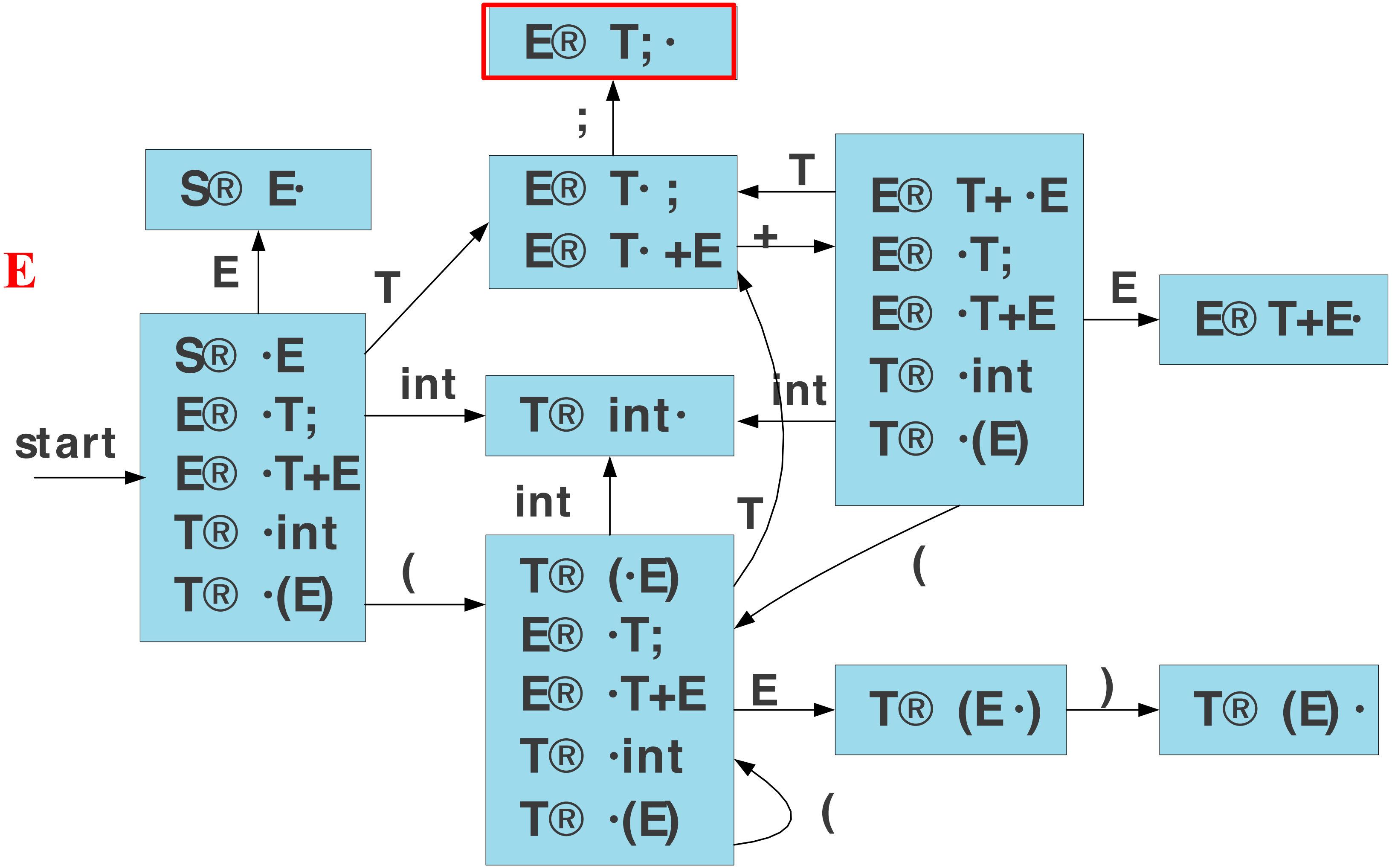
$S \rightarrow E$
$E \rightarrow T;$
$E \rightarrow T + E$
$T \rightarrow int$
$T \rightarrow (E)$





LR(0) Parsing

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



T	+	(	T	+	E	|	)	;

LR(0) Parsing

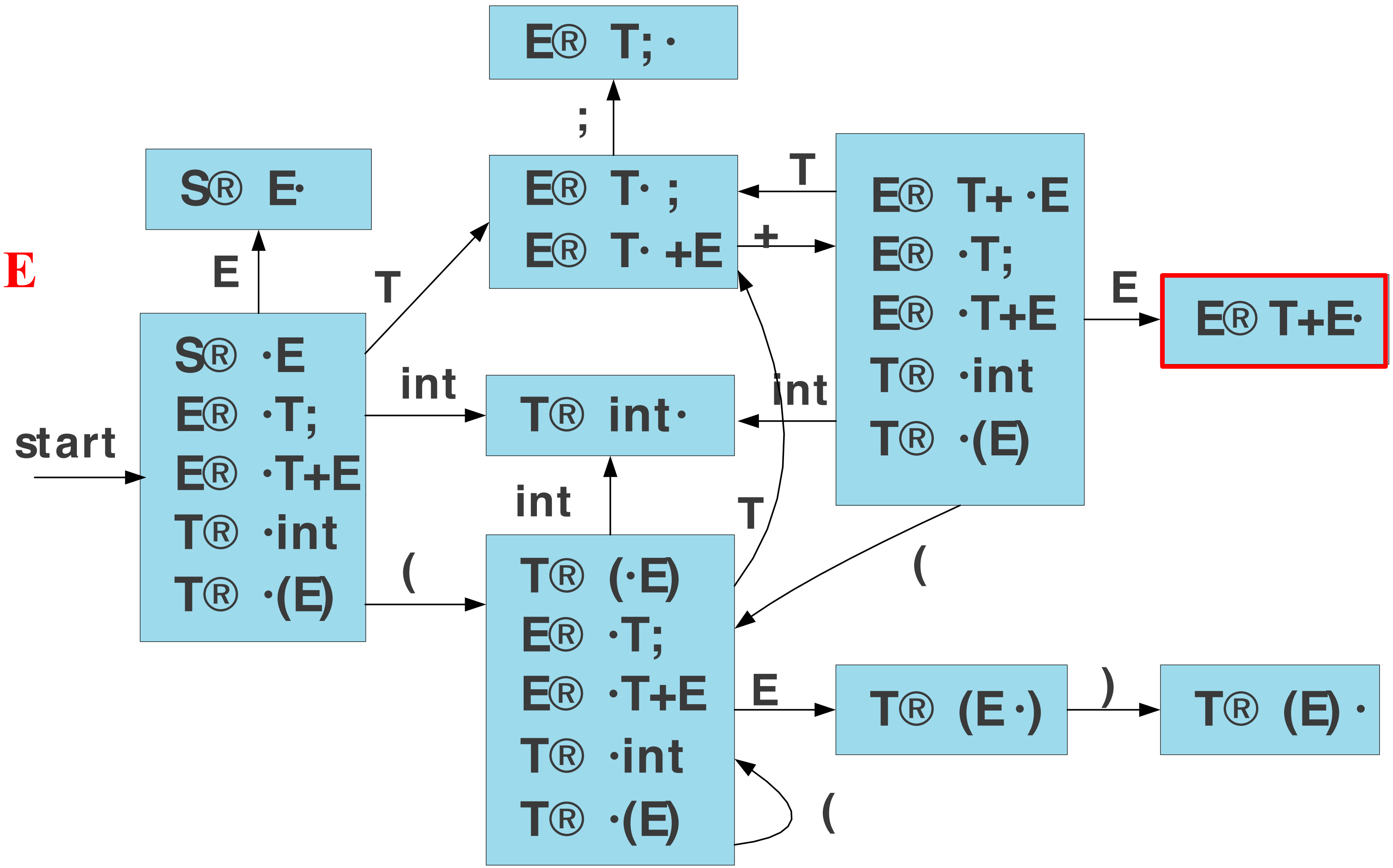
$S \longrightarrow E$

$E \longrightarrow T;$

$E \longrightarrow T + E$

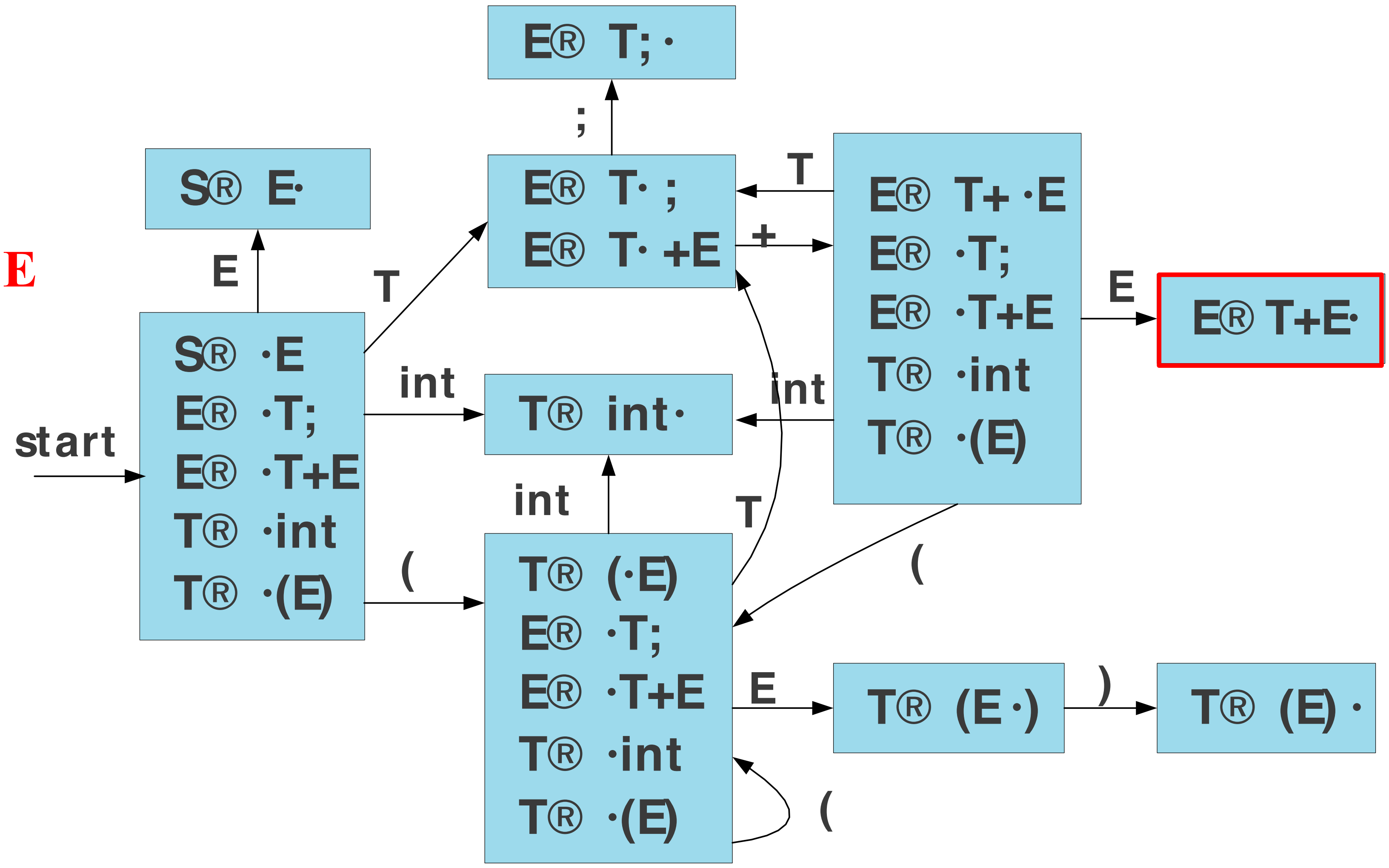
$T \longrightarrow \mathbf{int}$

$T \longrightarrow (E)$



LR(0) Parsing

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



T	+	(	E

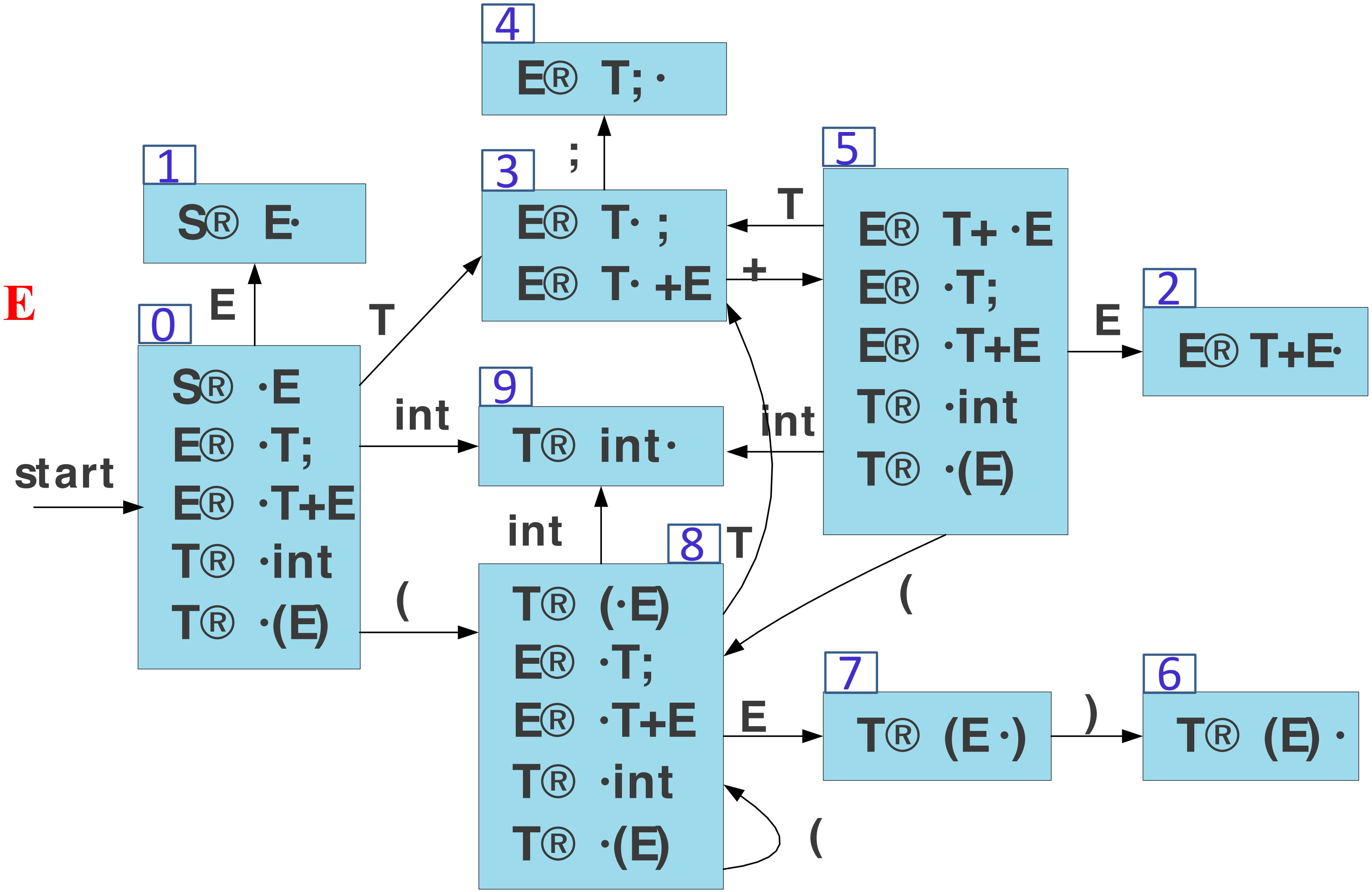
)	;

An optimization

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

LR(0) Parsing

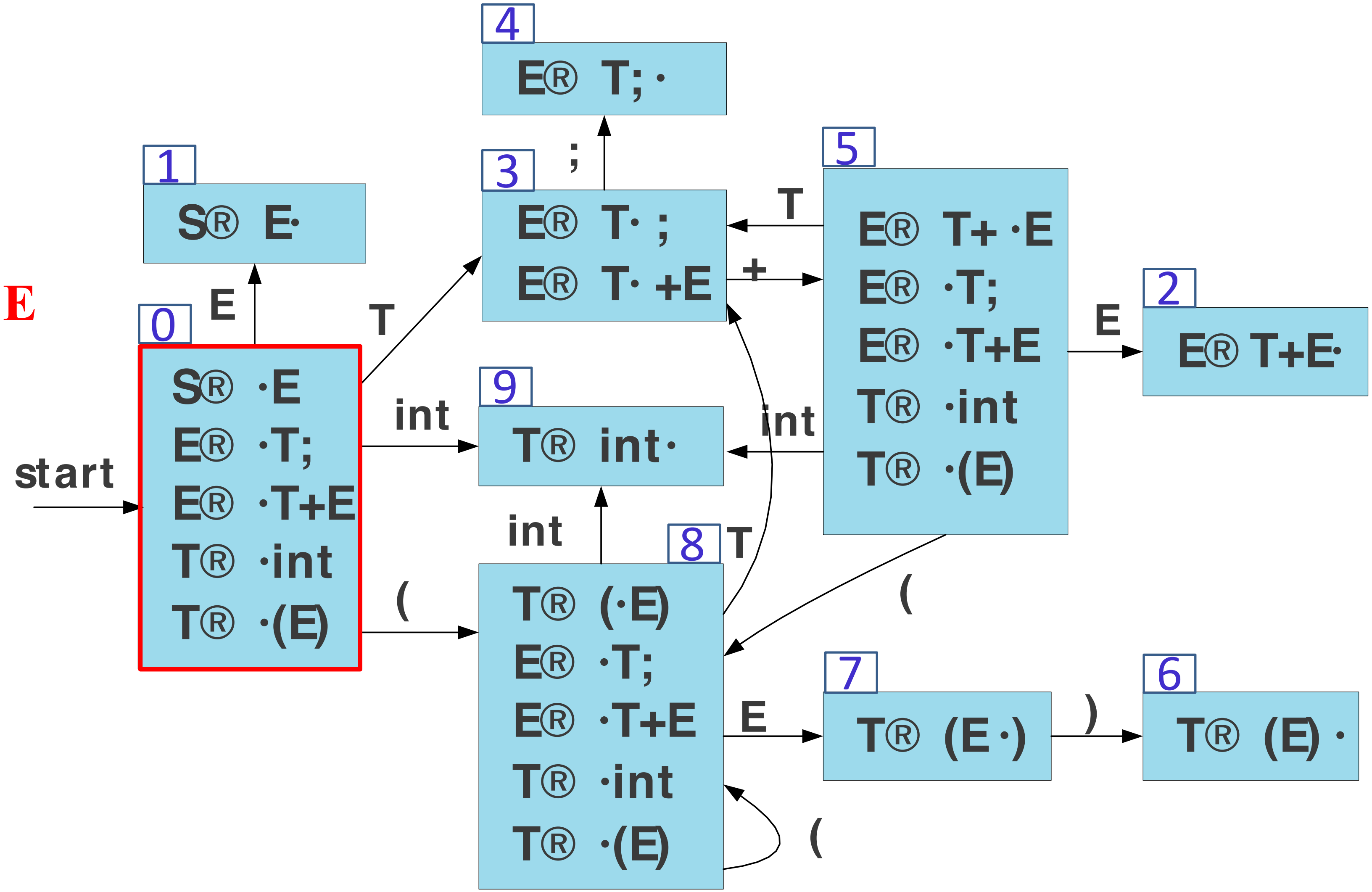
$S \rightarrow E$
$E \rightarrow T;$
$E \rightarrow T + E$
$T \rightarrow \textbf{int}$
$T \rightarrow (E)$



int	+	(	int	+	int	;	)	;

LR(0) Parsing

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



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

LR(0) Parsing

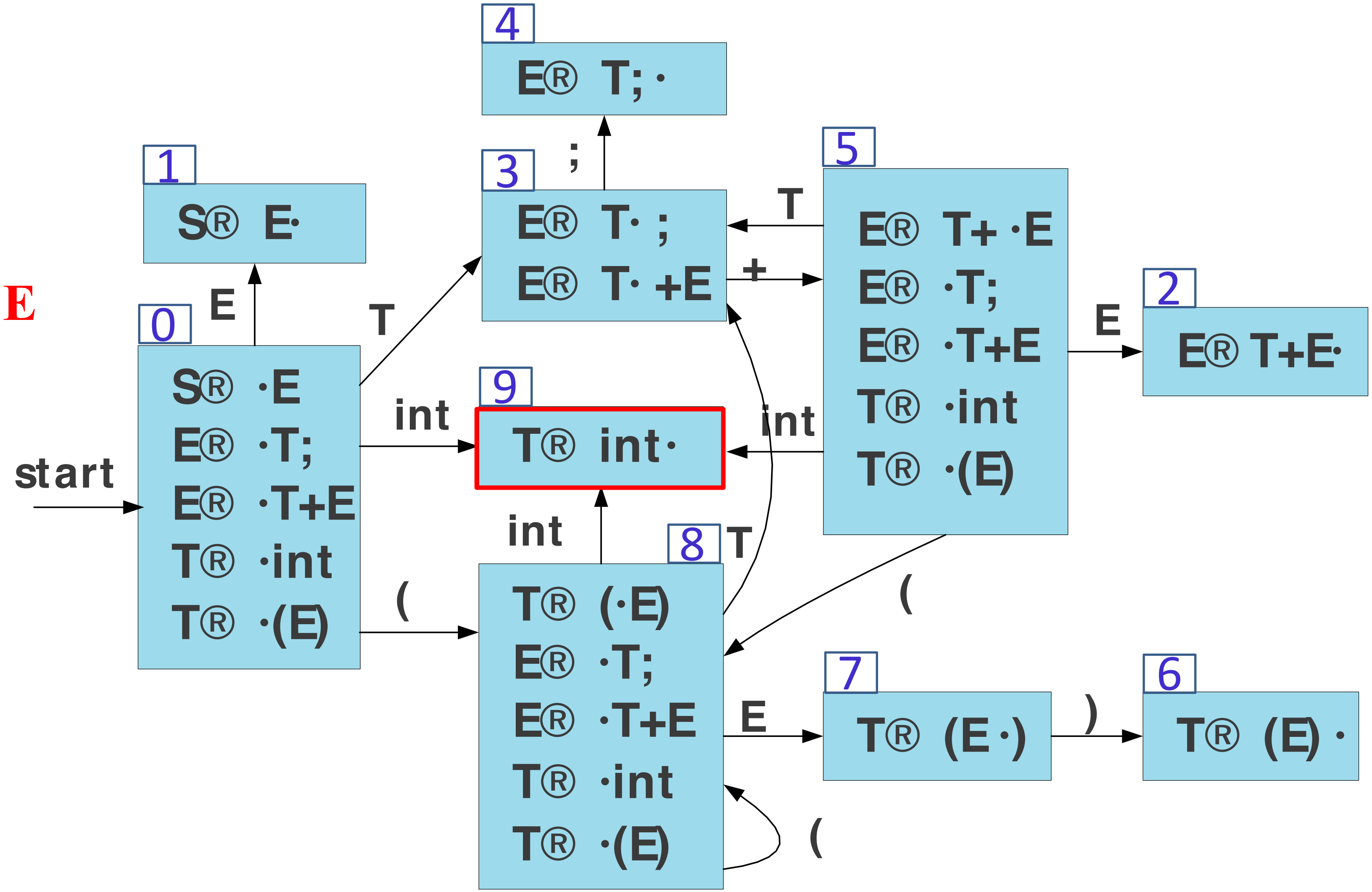
$S \rightarrow E$

$E \rightarrow T;$

$E \rightarrow T + E$

$T \rightarrow \text{int}$

$T \rightarrow (E)$

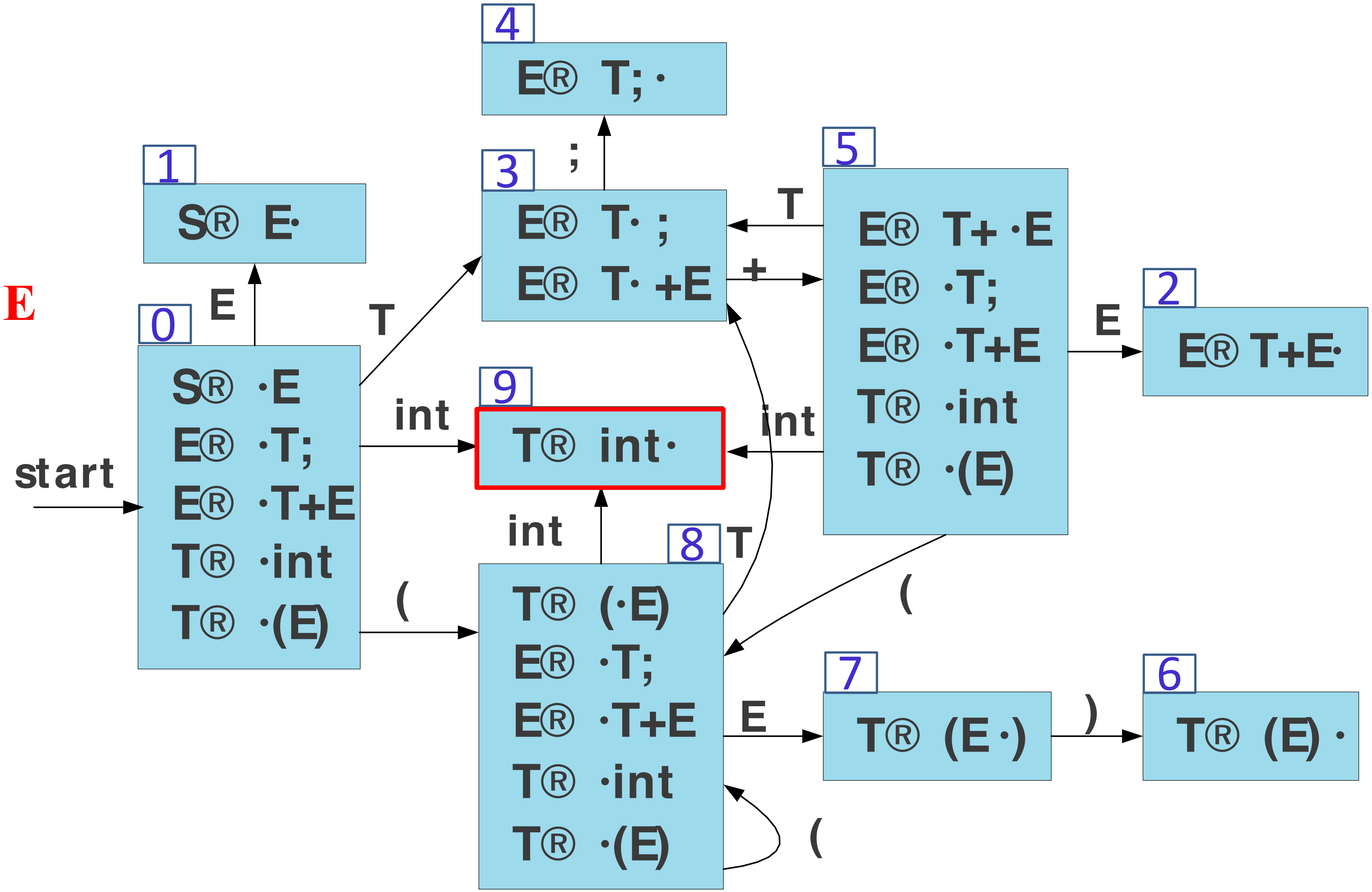


$	int
0	9

+	(	int	+	int	;	)	;	$

LR(0) Parsing

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



$

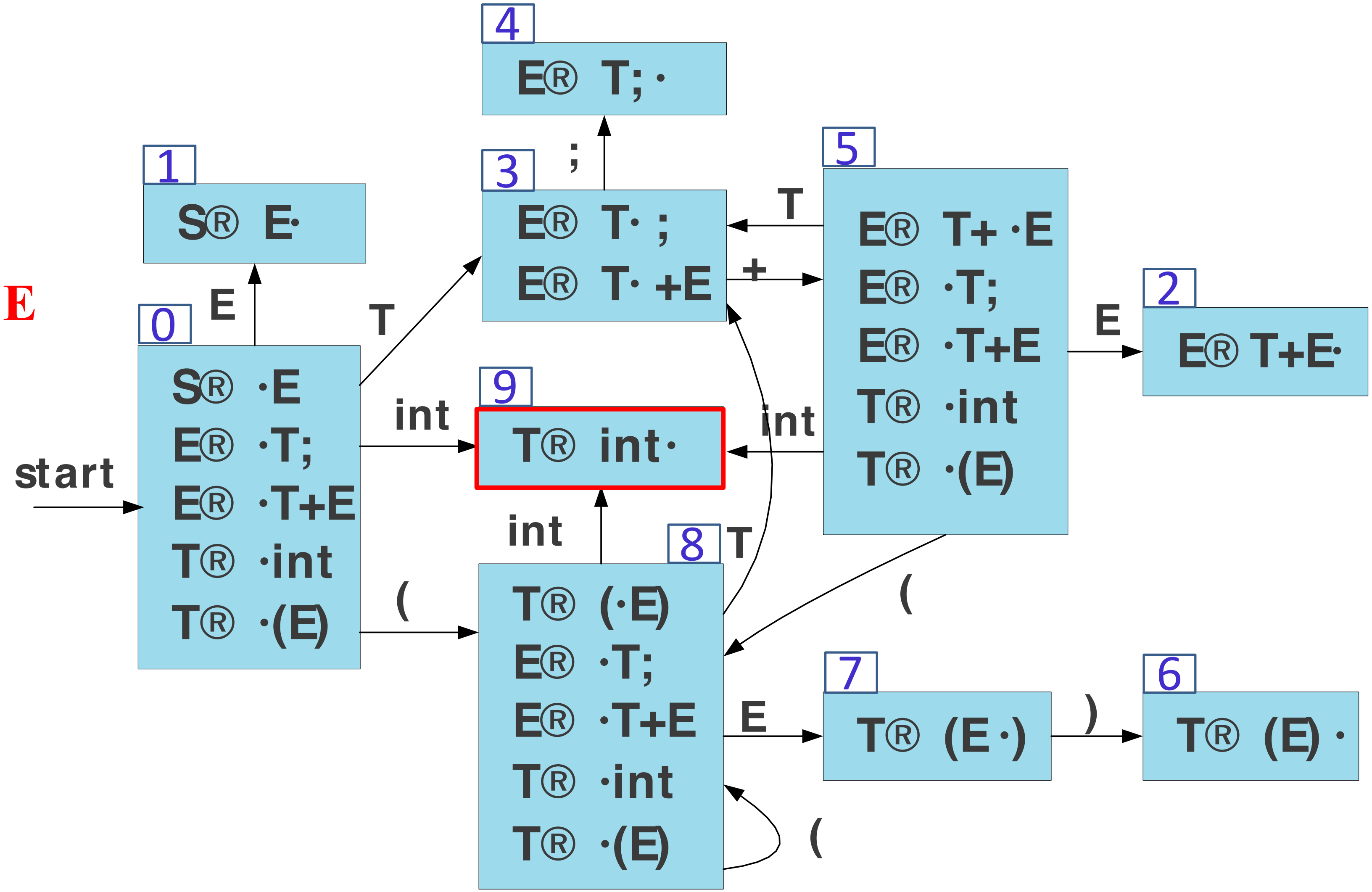
+	(	int	+	int	;	)	;

$

0

LR(0) Parsing

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

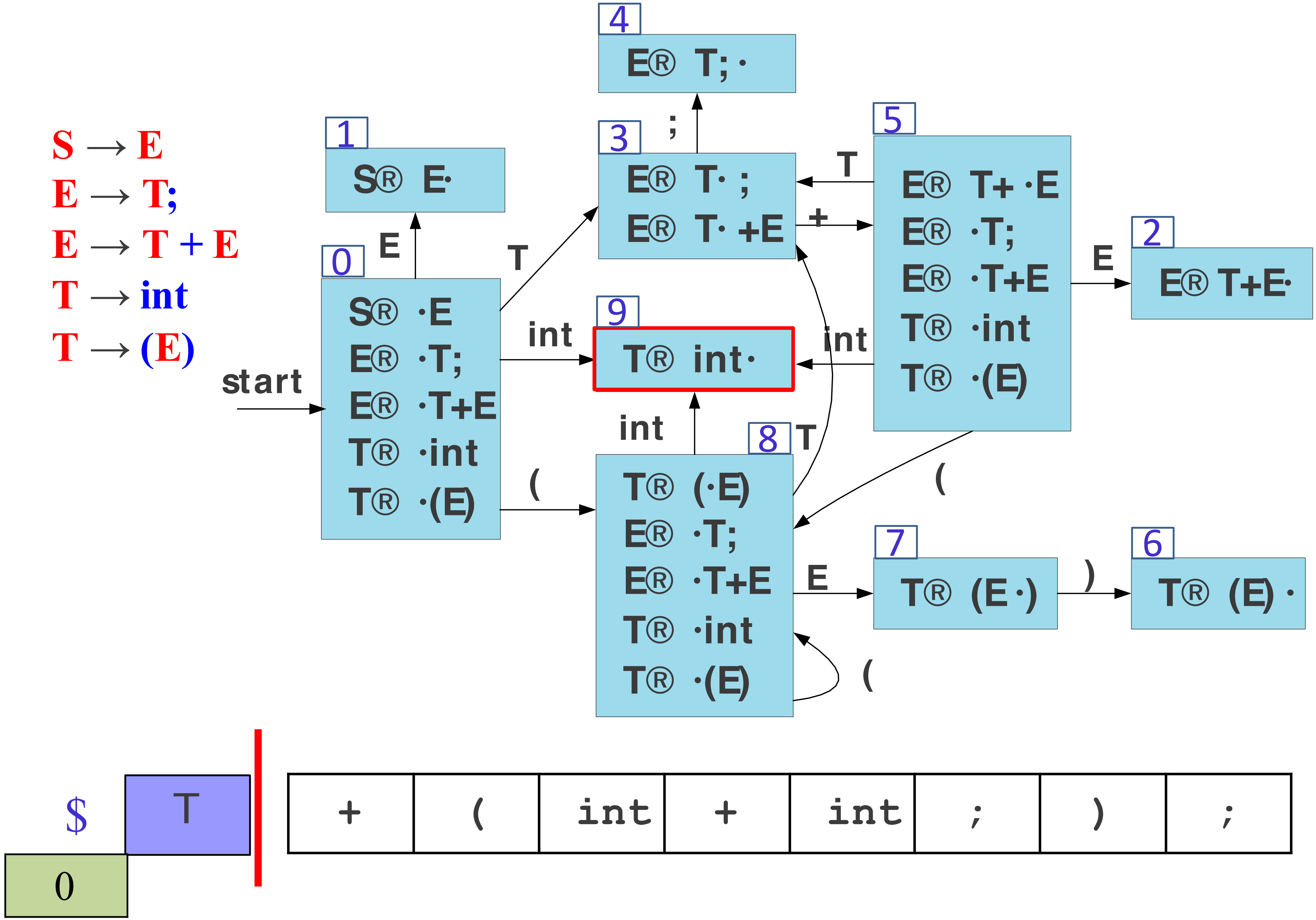


$
0

+	(	int	+	int	;	)	;	$

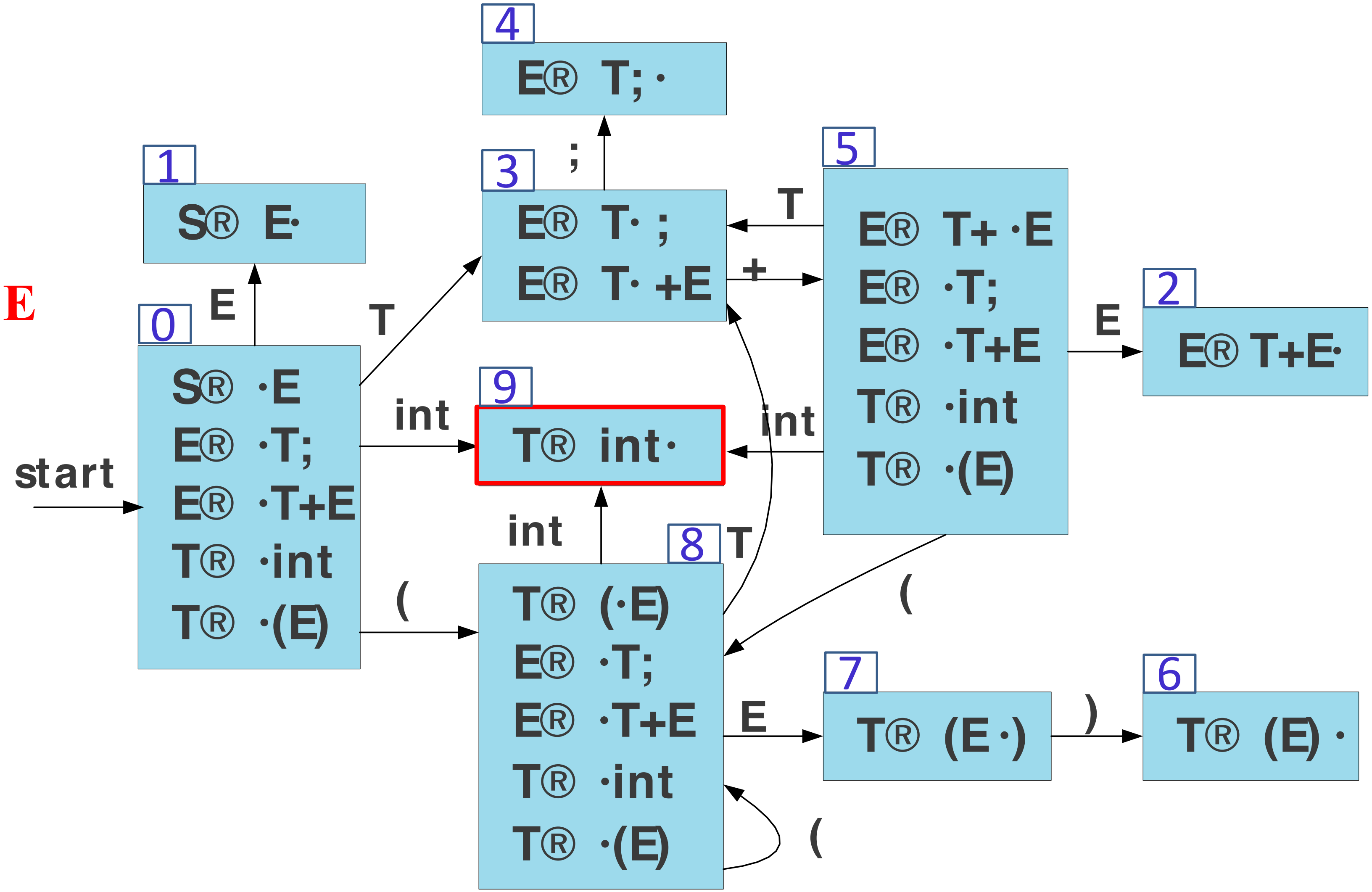
LR(0) Parsing

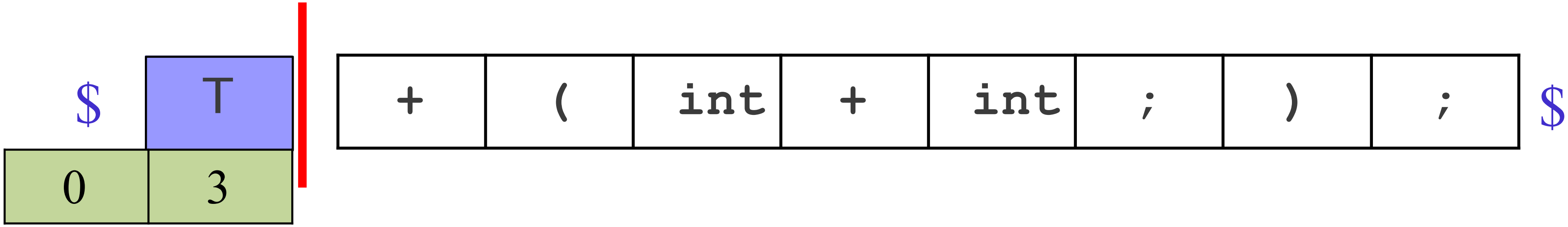
S → E
E → T;
E → T + E
T → int
T → (E)



LR(0) Parsing

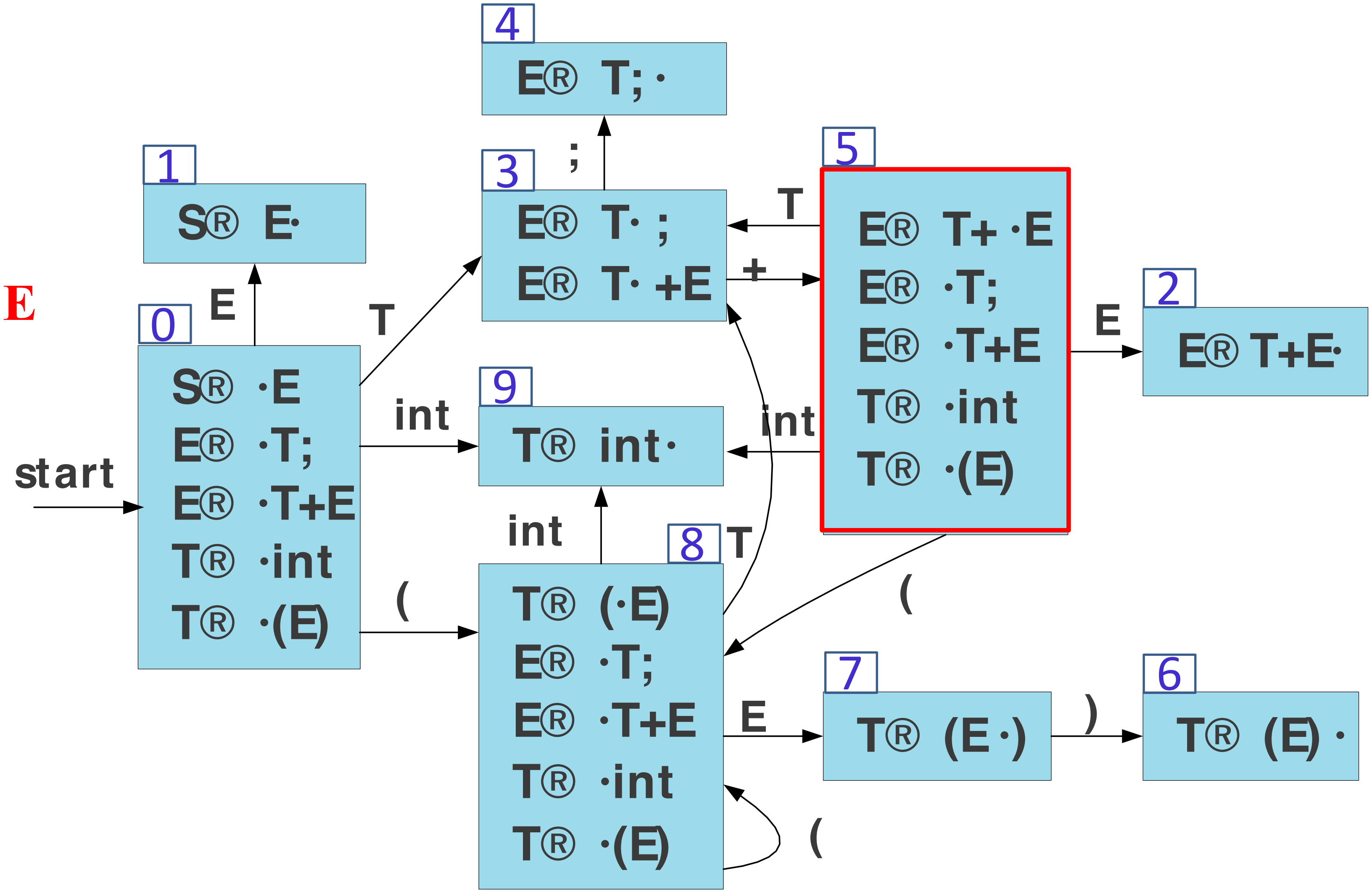
$S \rightarrow E$
$E \rightarrow T;$
$E \rightarrow T + E$
$T \rightarrow \text{int}$
$T \rightarrow (E)$





LR(0) Parsing

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

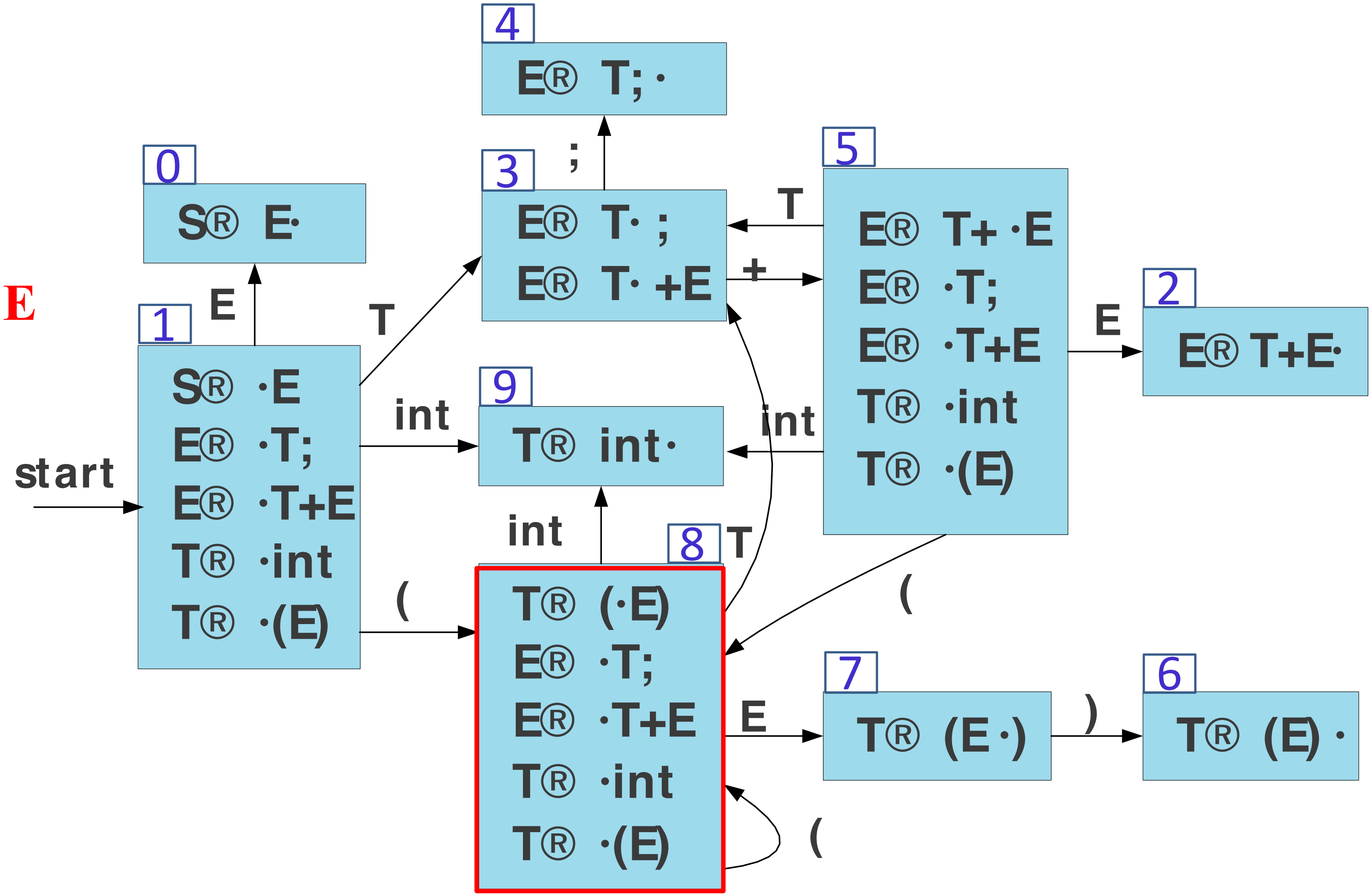


$	T	+
0	3	5

(	int	+	int	;	)	;	$

LR(0) Parsing

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

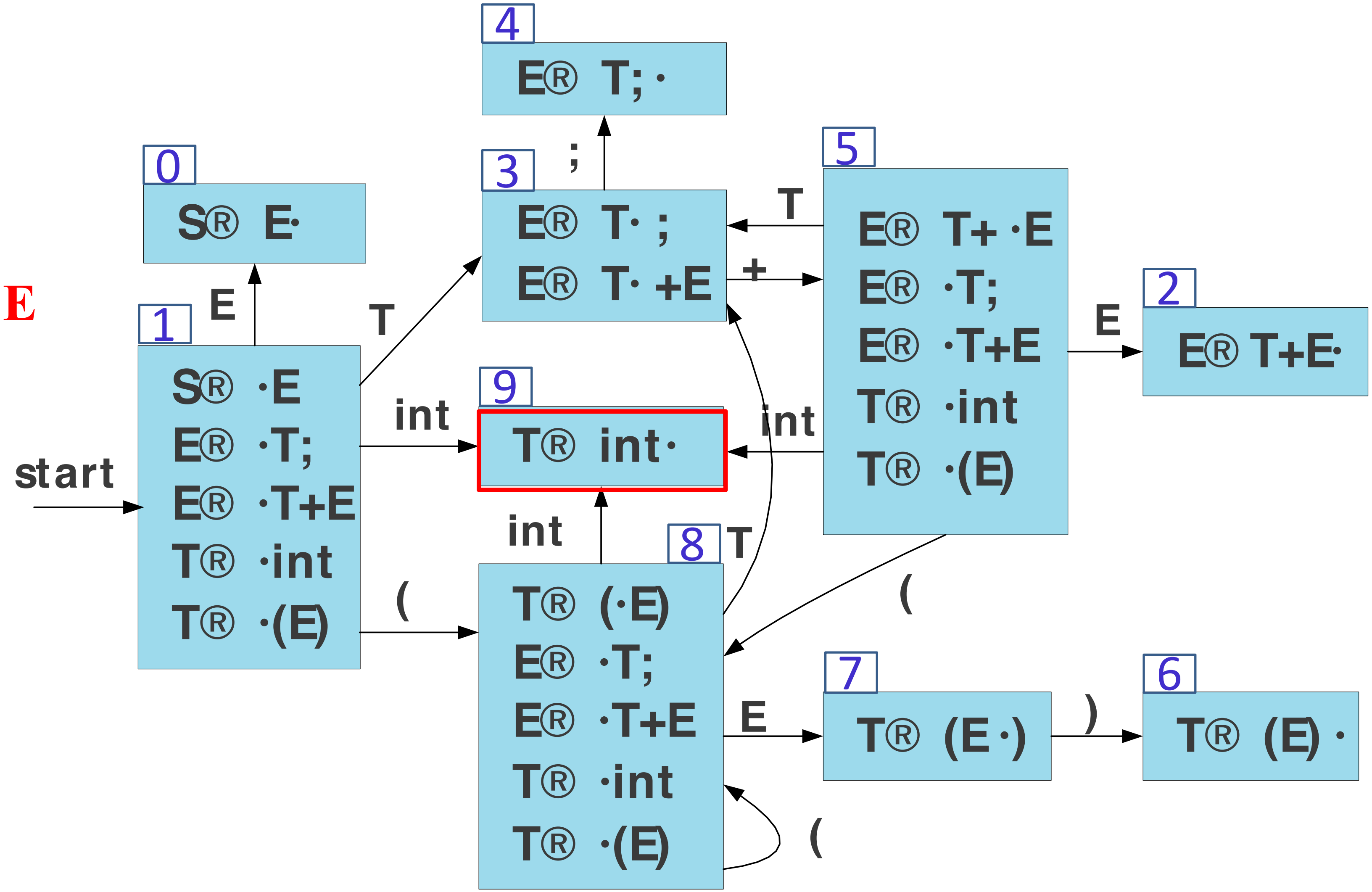


$	T	+	(
0	3	5	8

int	+	int	;	)	;	$

LR(0) Parsing

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

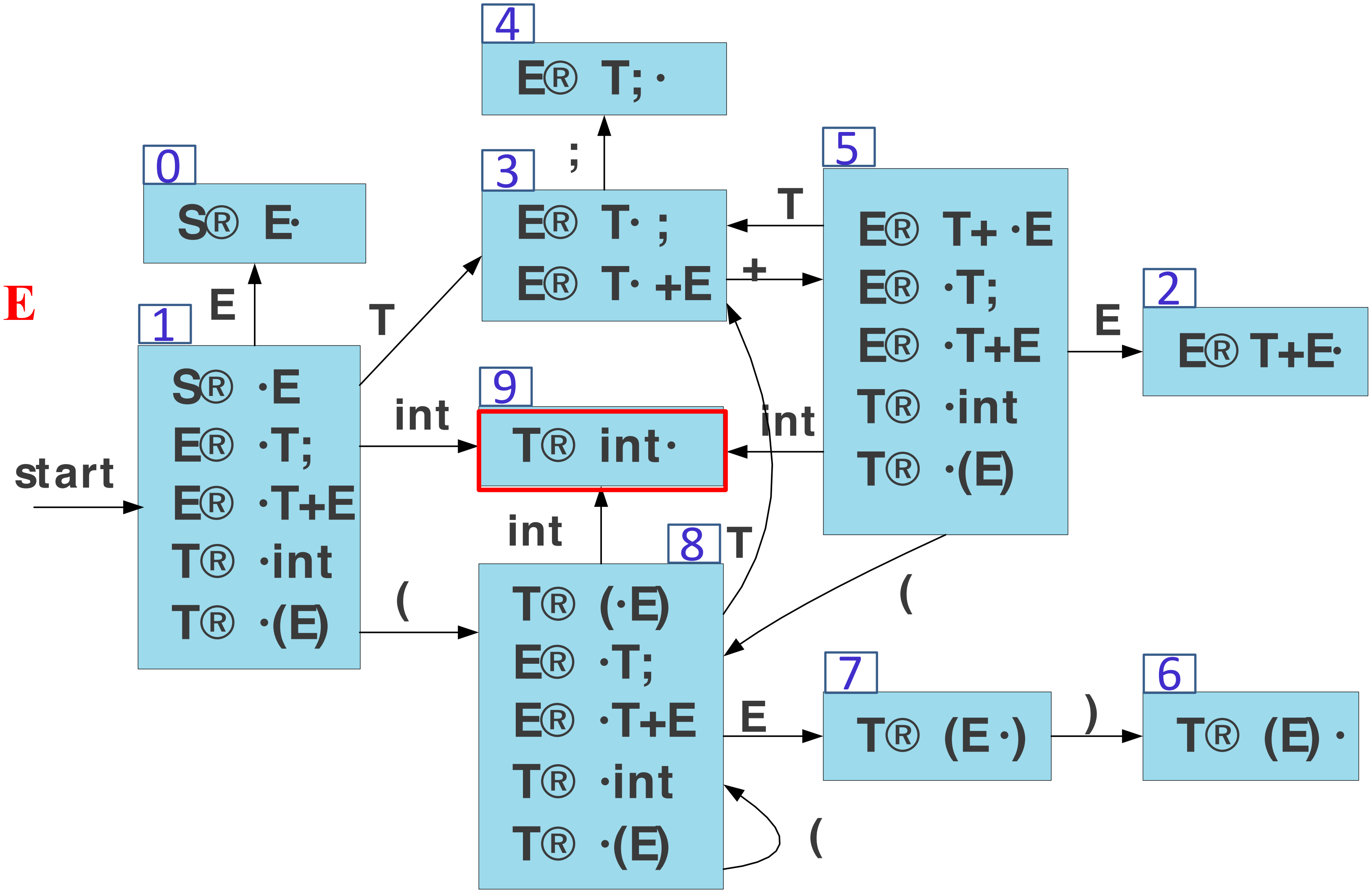


$	T	+	(	int
	0	3	5	8	9

+	int	;	)	;	$

LR(0) Parsing

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

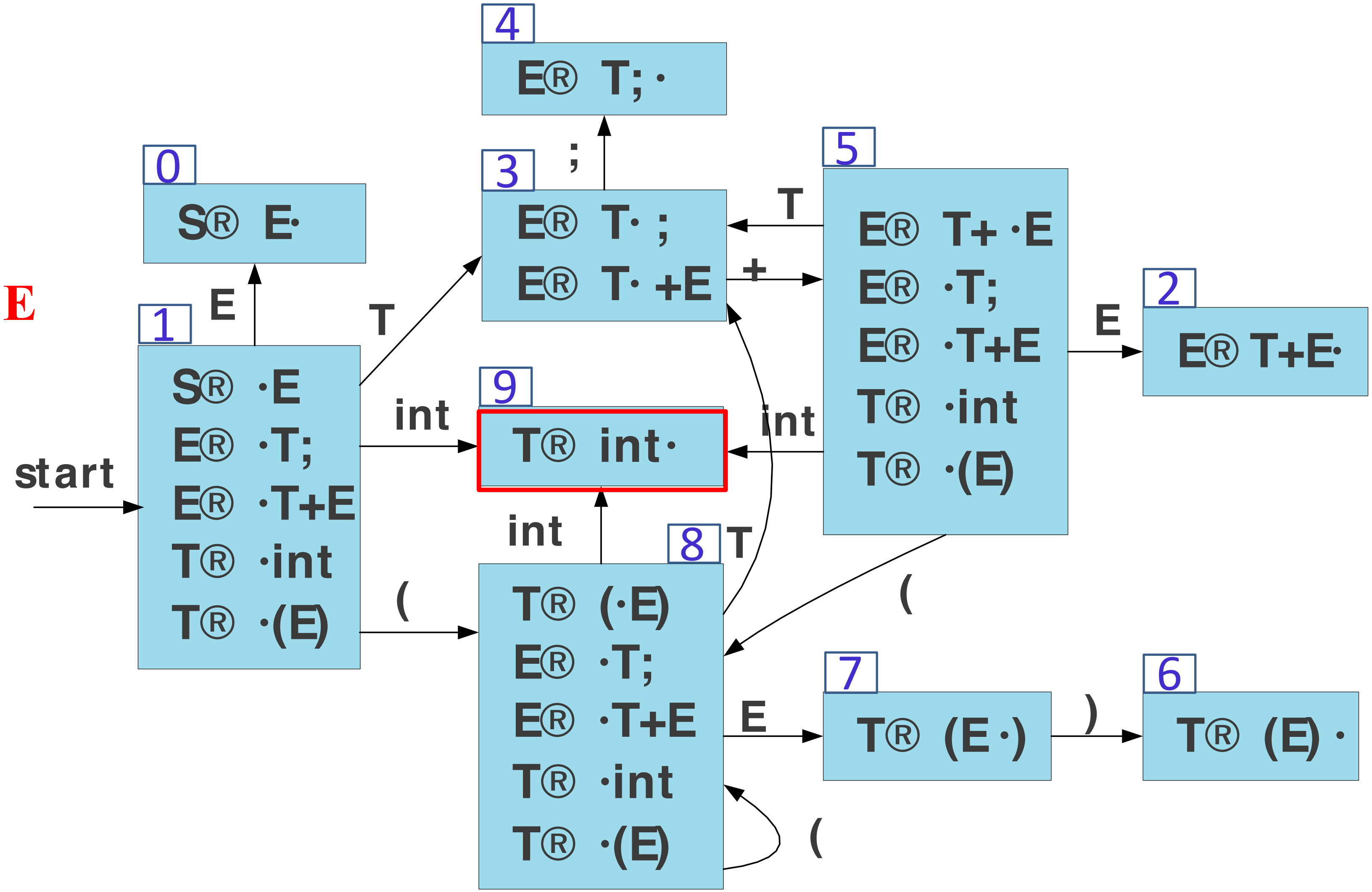


$	T	+	(	
	0	3	5	8

+	int	;	)	;	$

LR(0) Parsing

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

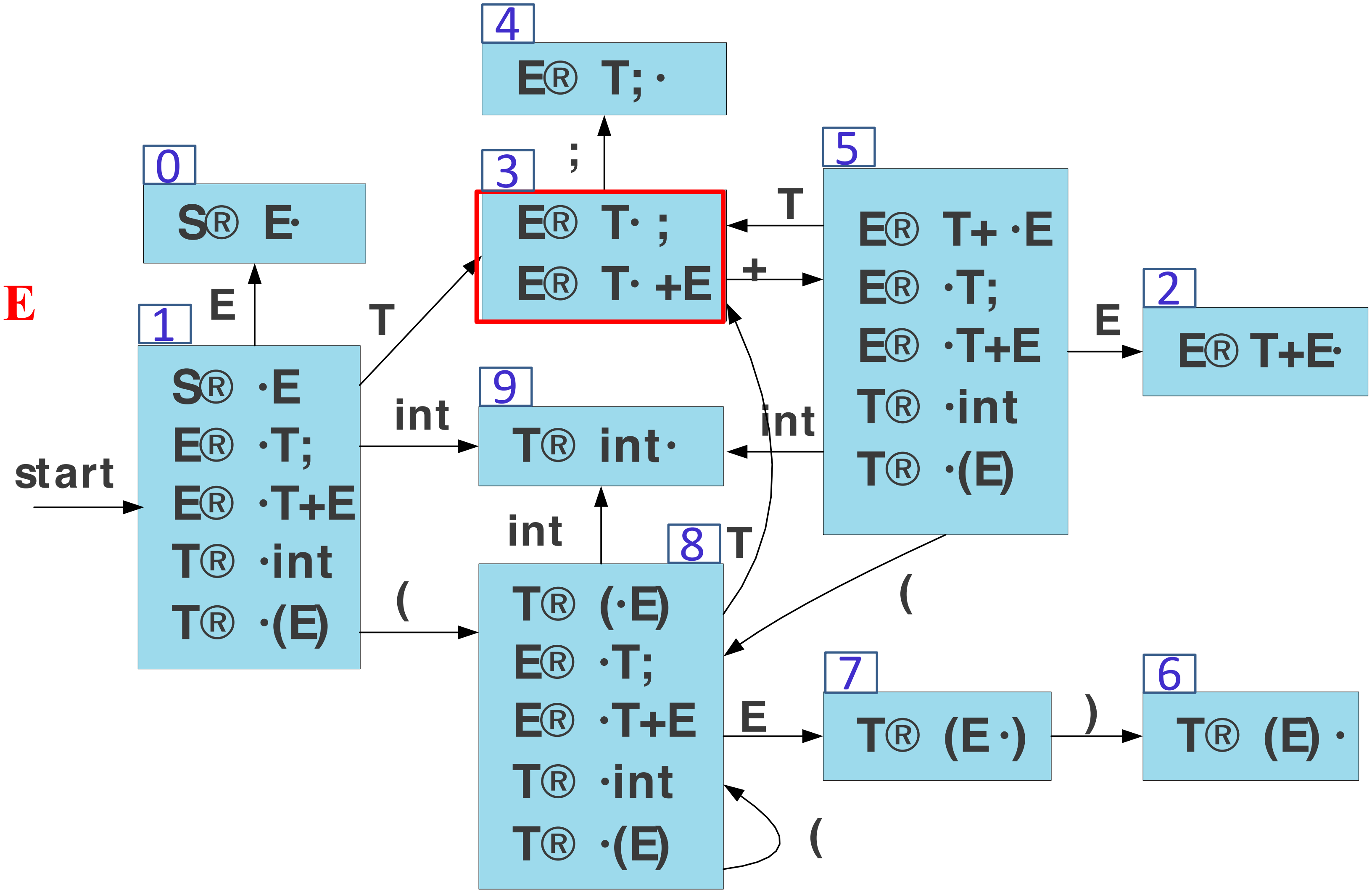


\$	T	+	(	T
0	3	5	8	3

+	int	;	)	;	\$

LR(0) Parsing

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



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

int	;	)	;	$

LR(0) Parsing

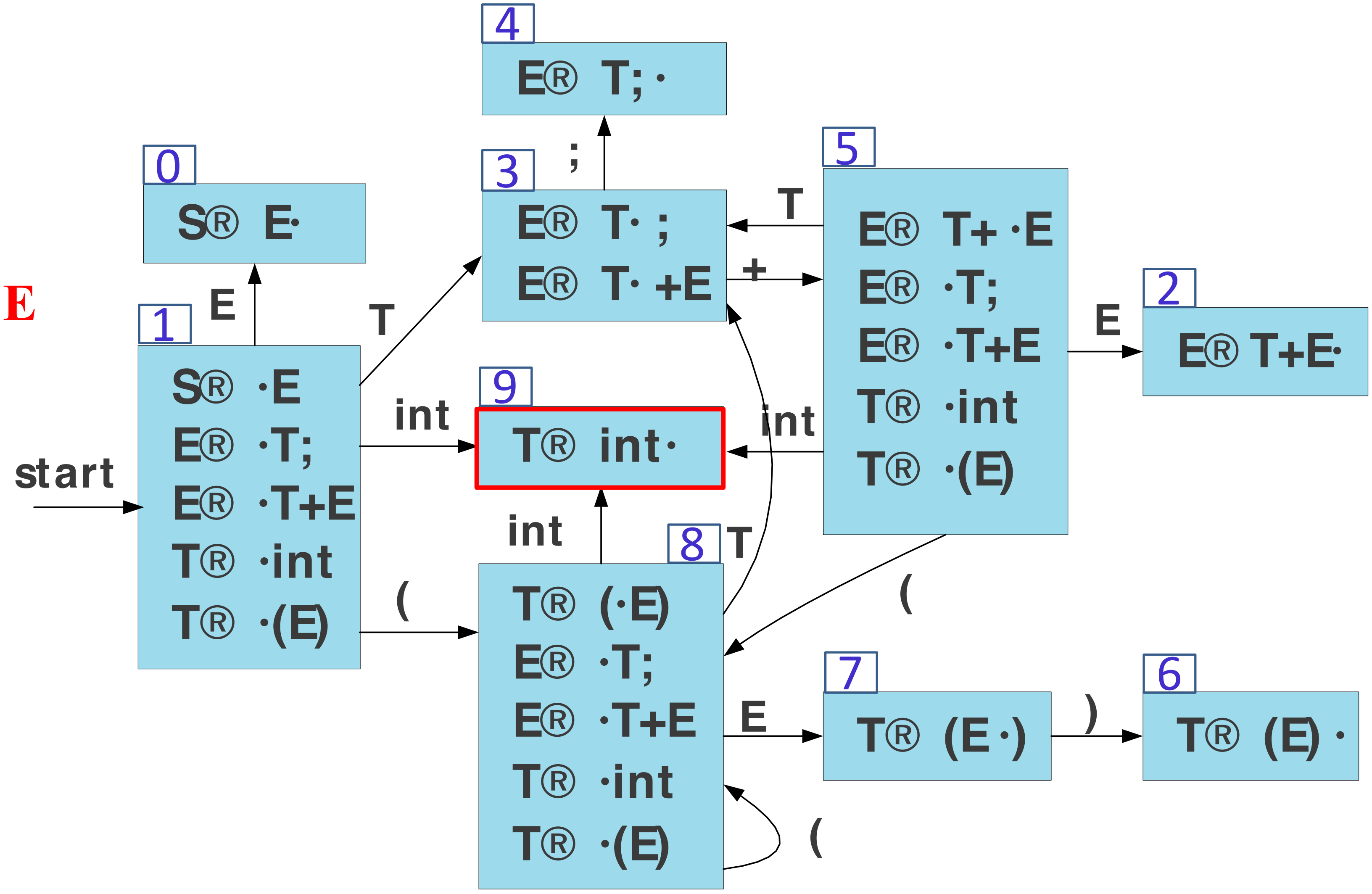
$S \rightarrow E$

$E \rightarrow T;$

$E \rightarrow T + E$

$T \rightarrow$ int

$T \rightarrow (E)$

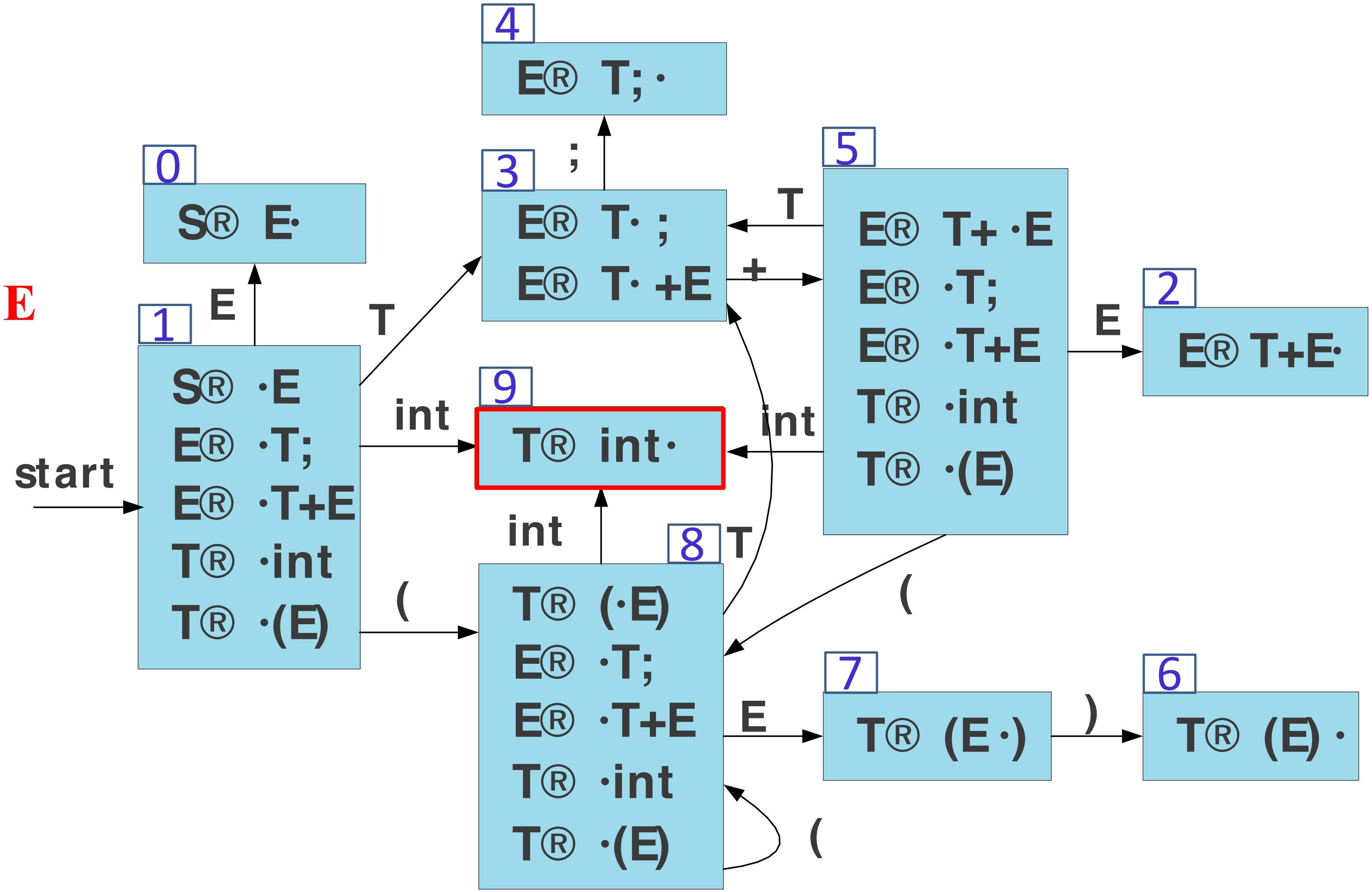


$	T	+	(	T	+	int
0	3	5	8	3	5	9

;	)	;	$

LR(0) Parsing

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



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

;	)	;	$

LR(0) Parsing

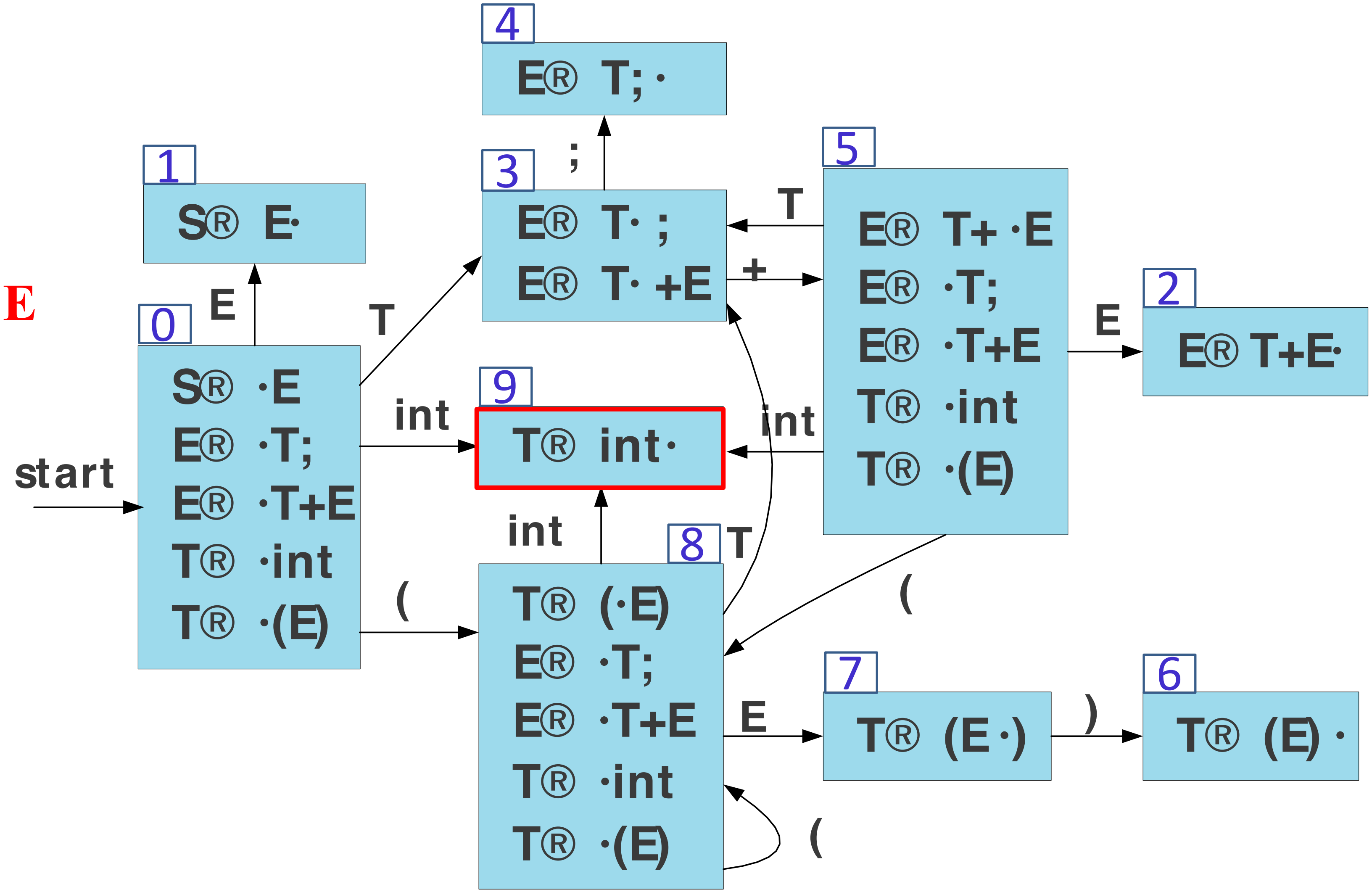
$S \rightarrow E$

$E \rightarrow T;$

$E \rightarrow T + E$

$T \rightarrow$ int

$T \rightarrow (E)$



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

;	)	;	$

LR(0) Parsing

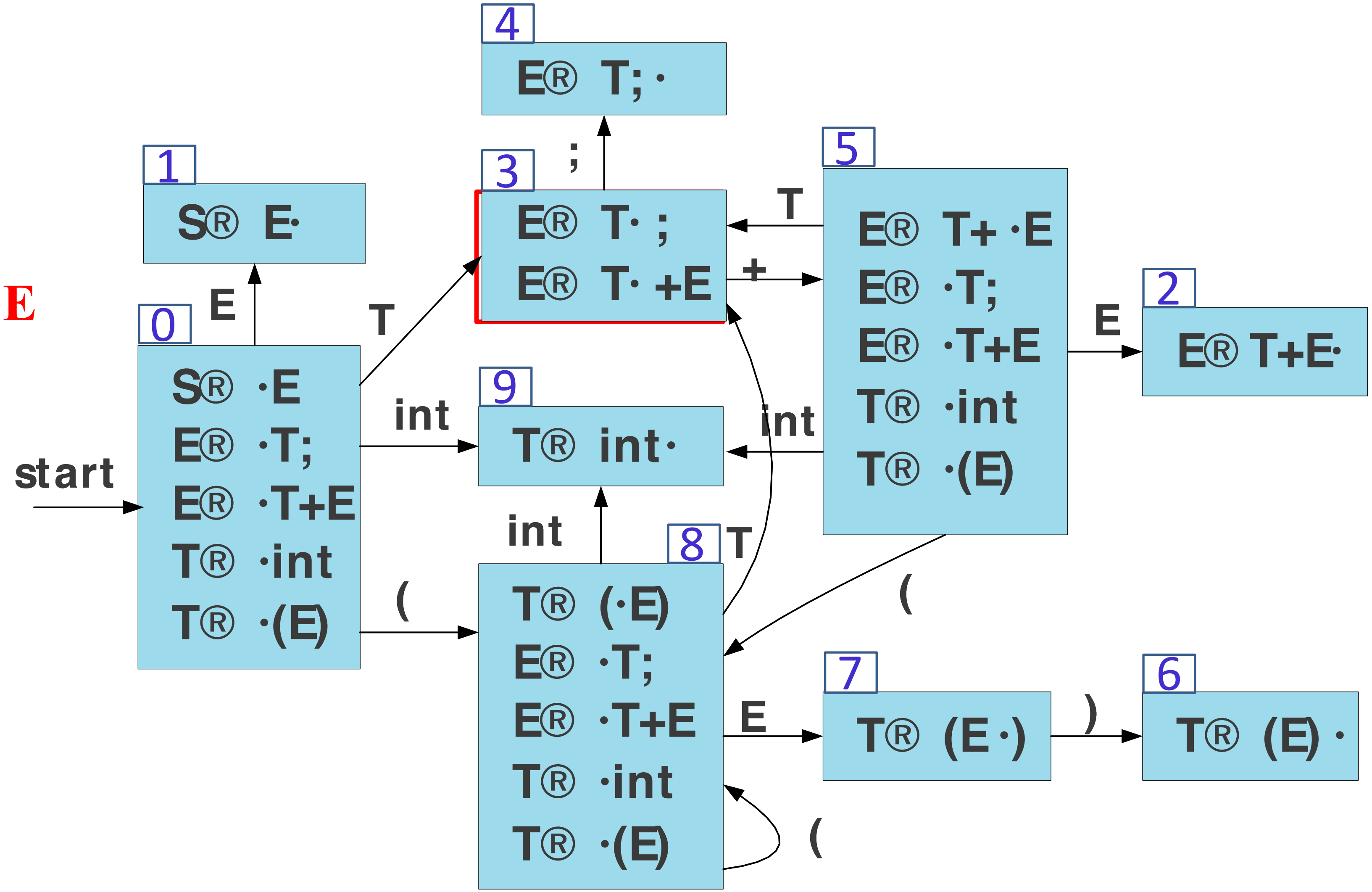
$S \rightarrow E$

$E \rightarrow T;$

$E \rightarrow T + E$

$T \rightarrow \textbf{int}$

$T \rightarrow (E)$



$	T	+	(	T	+	T	;
0	3	5	8	3	5	3	4

)	;	$

LR(0) Parsing

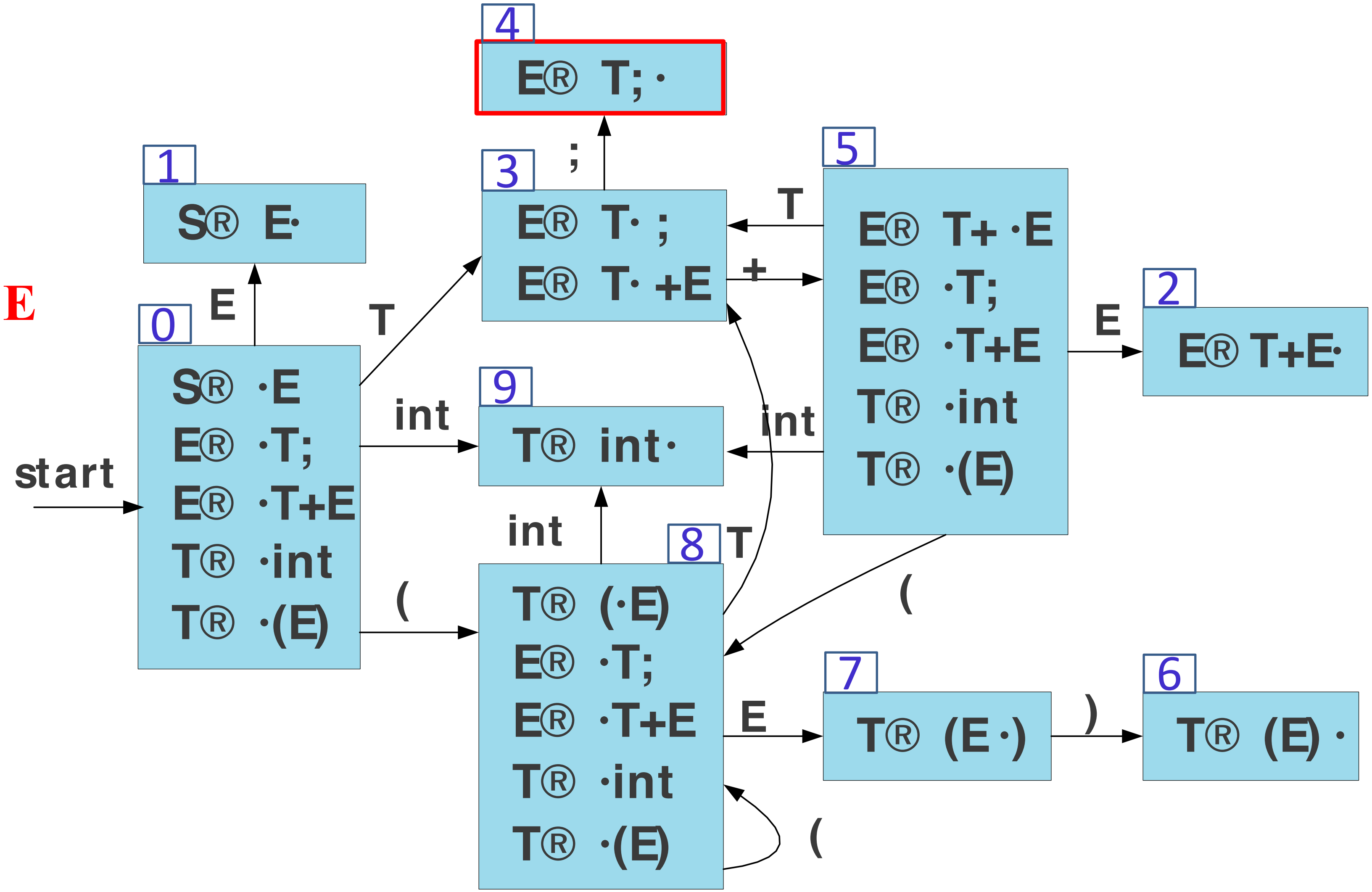
$S \rightarrow E$

$E \rightarrow T;$

$E \rightarrow T + E$

$T \rightarrow \text{int}$

$T \rightarrow (E)$

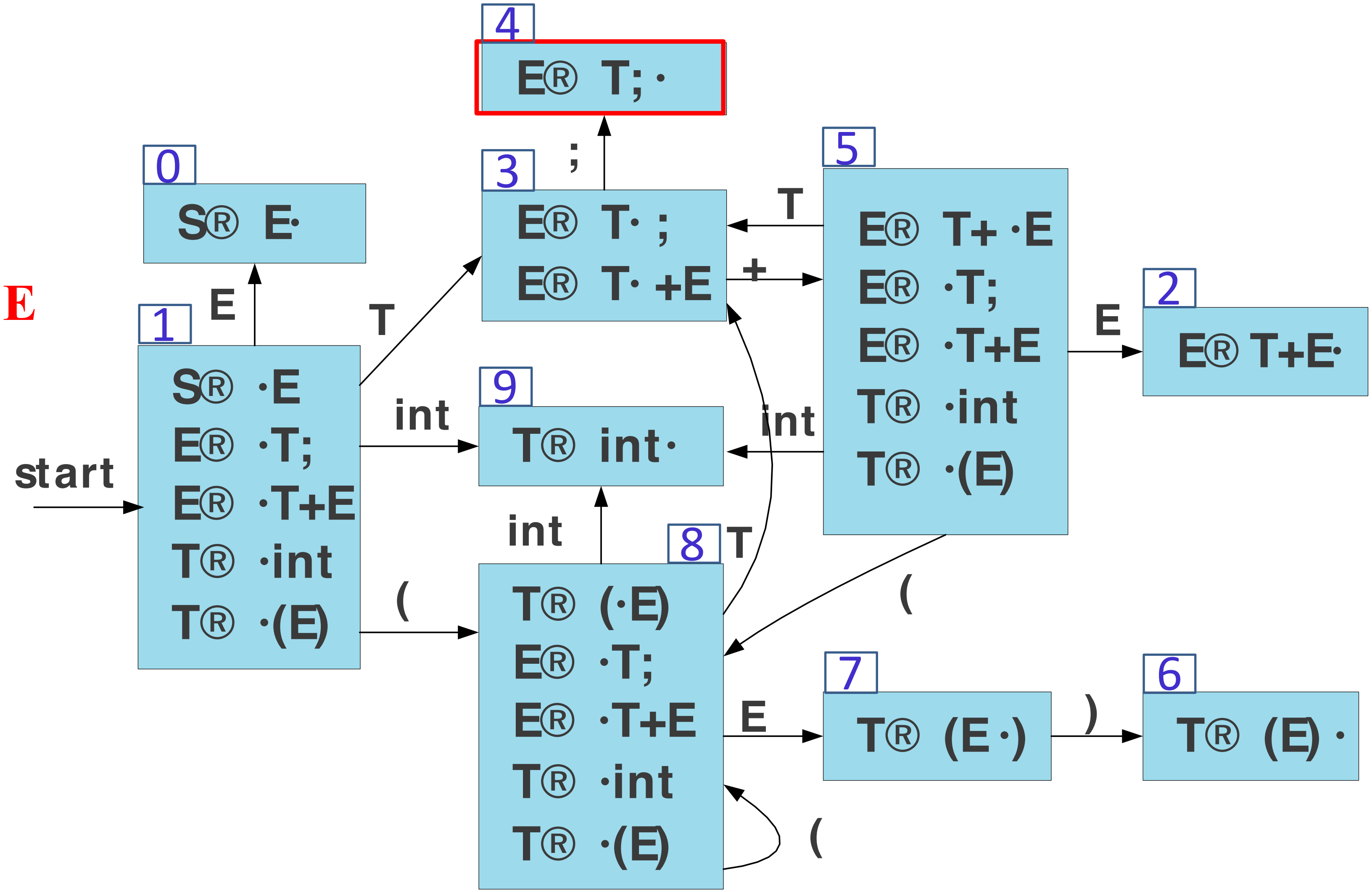


$	T	+	(	T	+
0	3	5	8	3	5

)	;	$

LR(0) Parsing

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

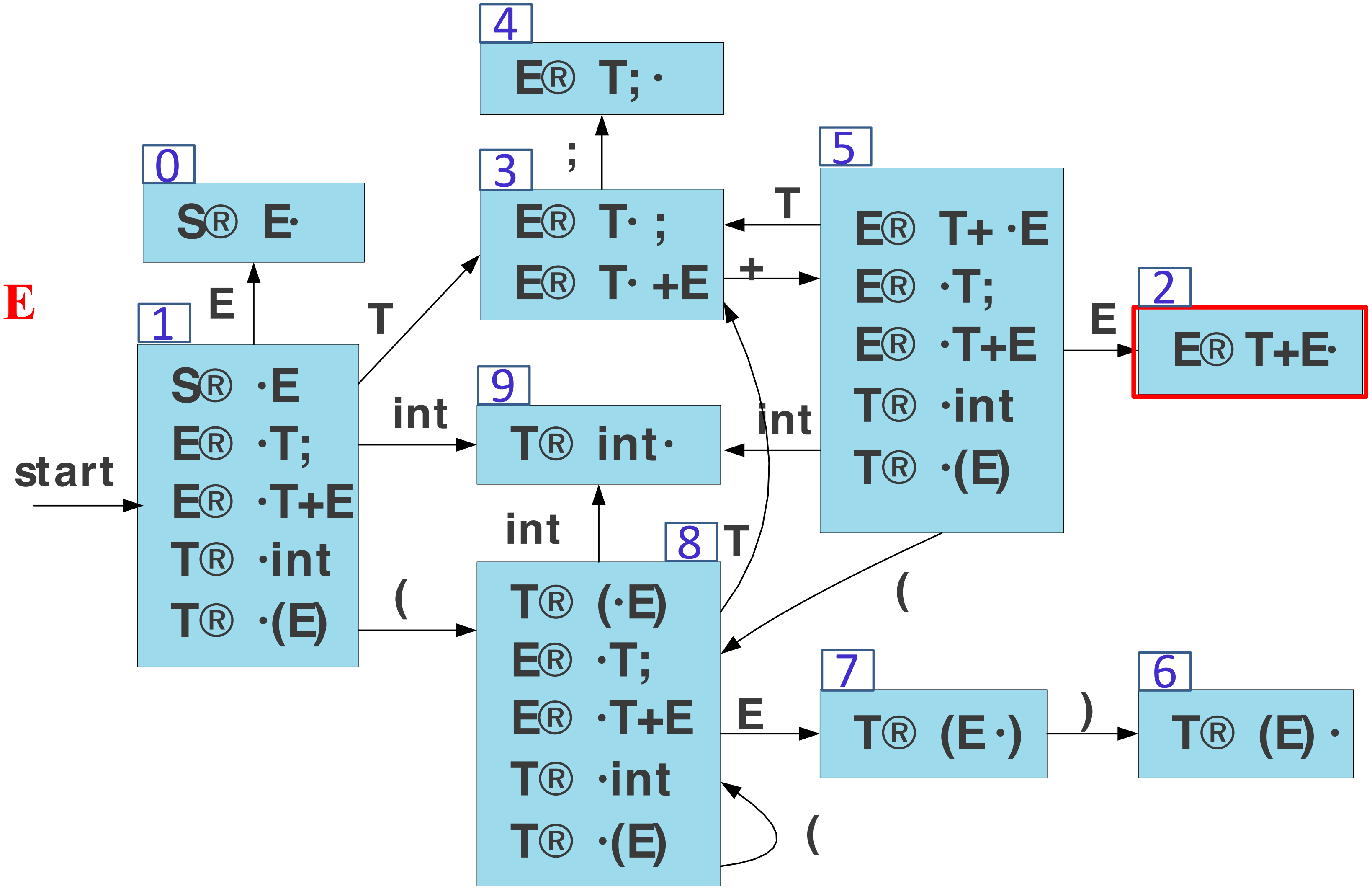


\$	T	+	(	T	+	E
0	3	5	8	3	5	2

)	;	\$

LR(0) Parsing

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

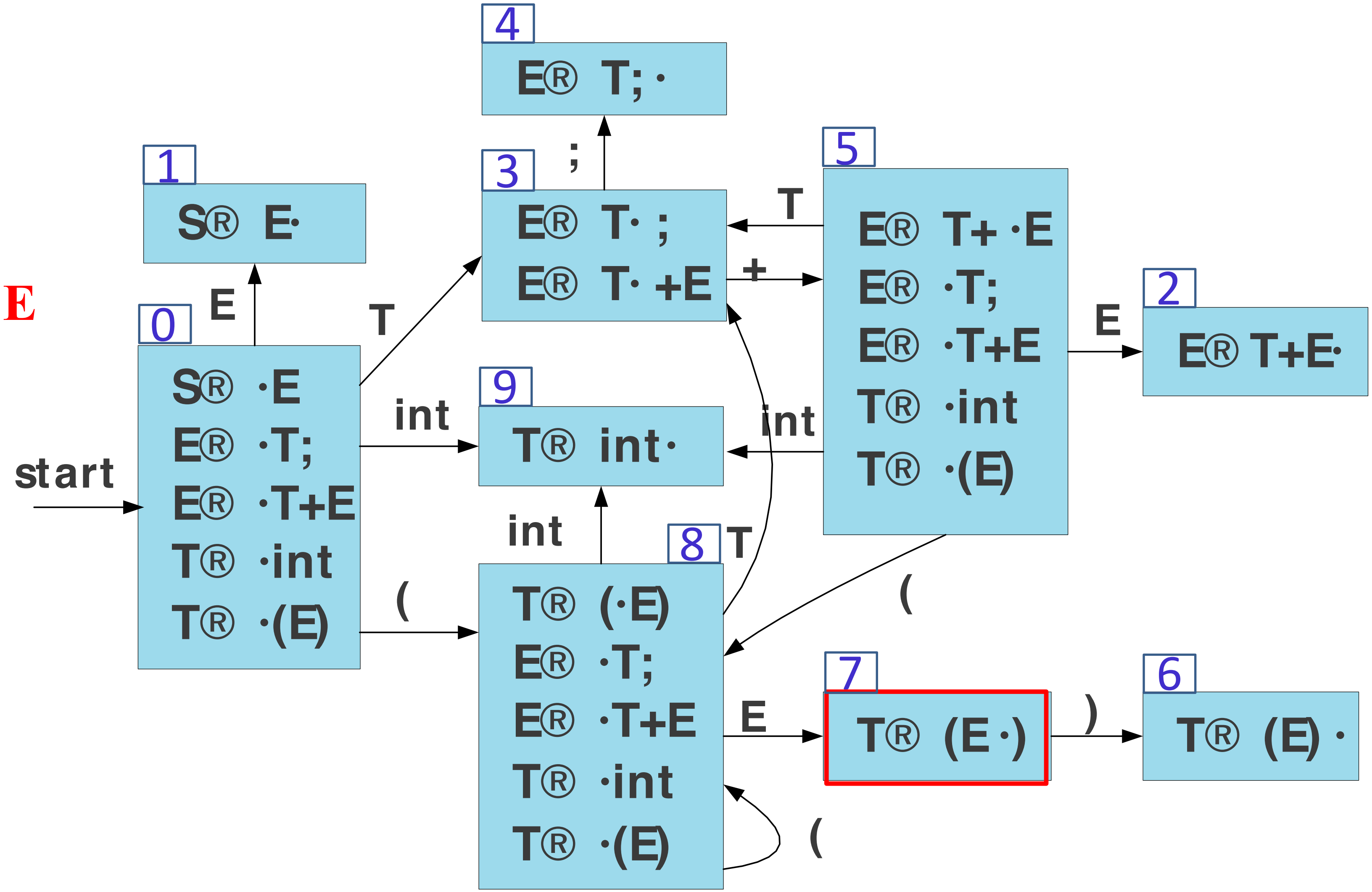


$	T	+	(
0	3	5	8

)	;	$

LR(0) Parsing

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

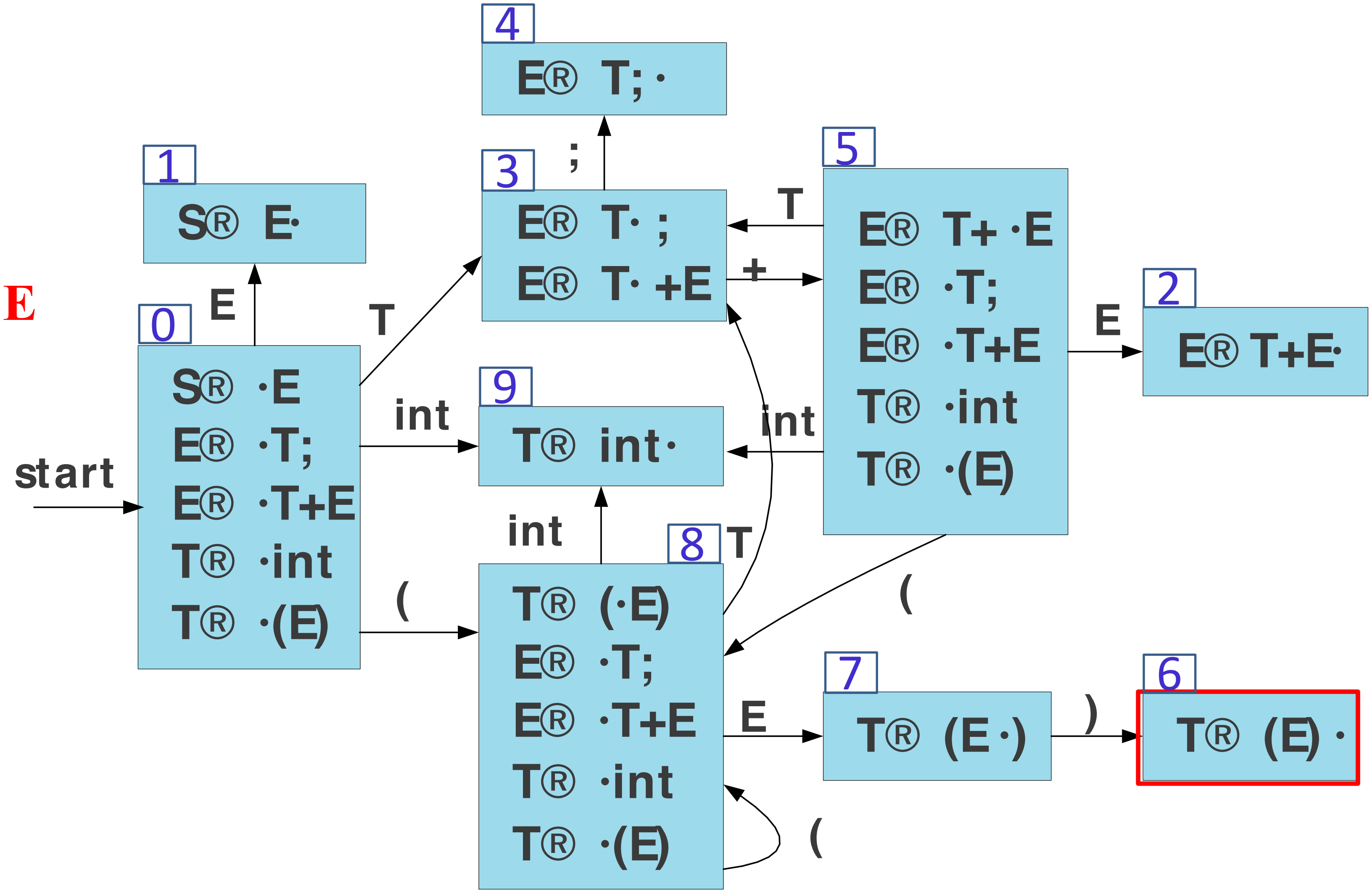


$	T	+	(	E
	0	3	5	8	7

)	;	$

LR(0) Parsing

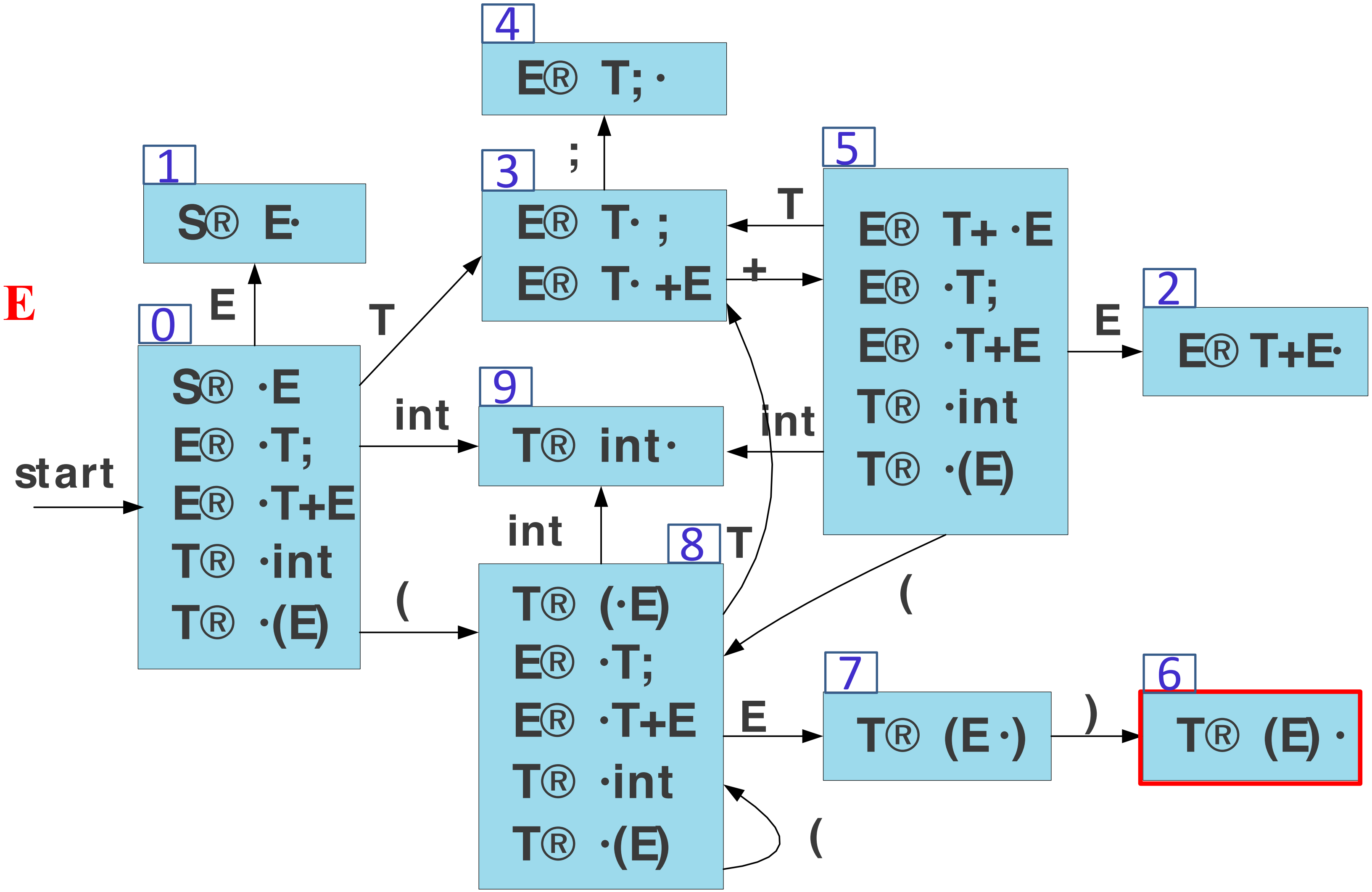
$S \rightarrow E$
$E \rightarrow T;$
$E \rightarrow T + E$
$T \rightarrow \text{int}$
$T \rightarrow (E)$



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

LR(0) Parsing

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

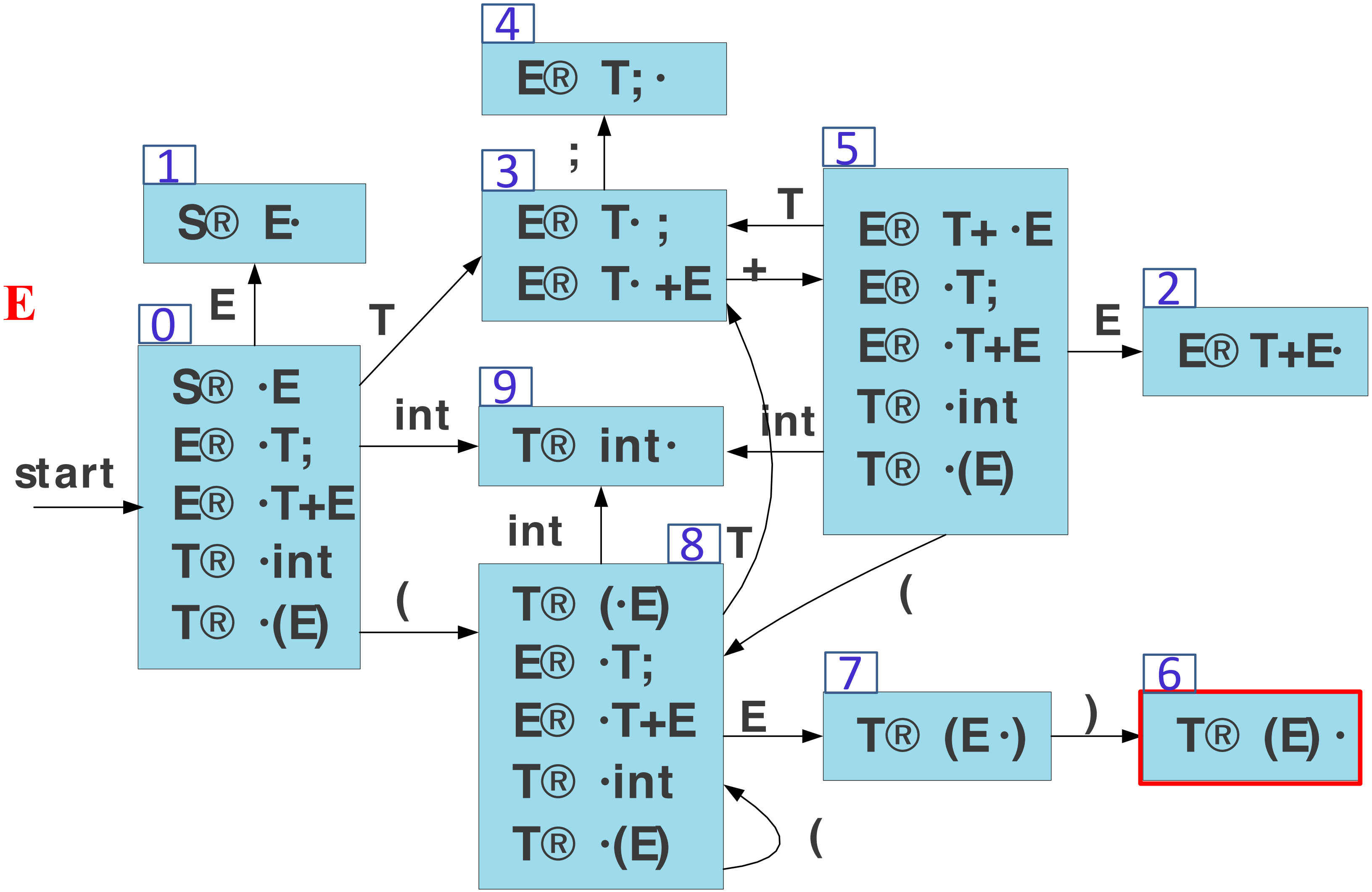


$	T	+
0	3	5

| ; $

LR(0) Parsing

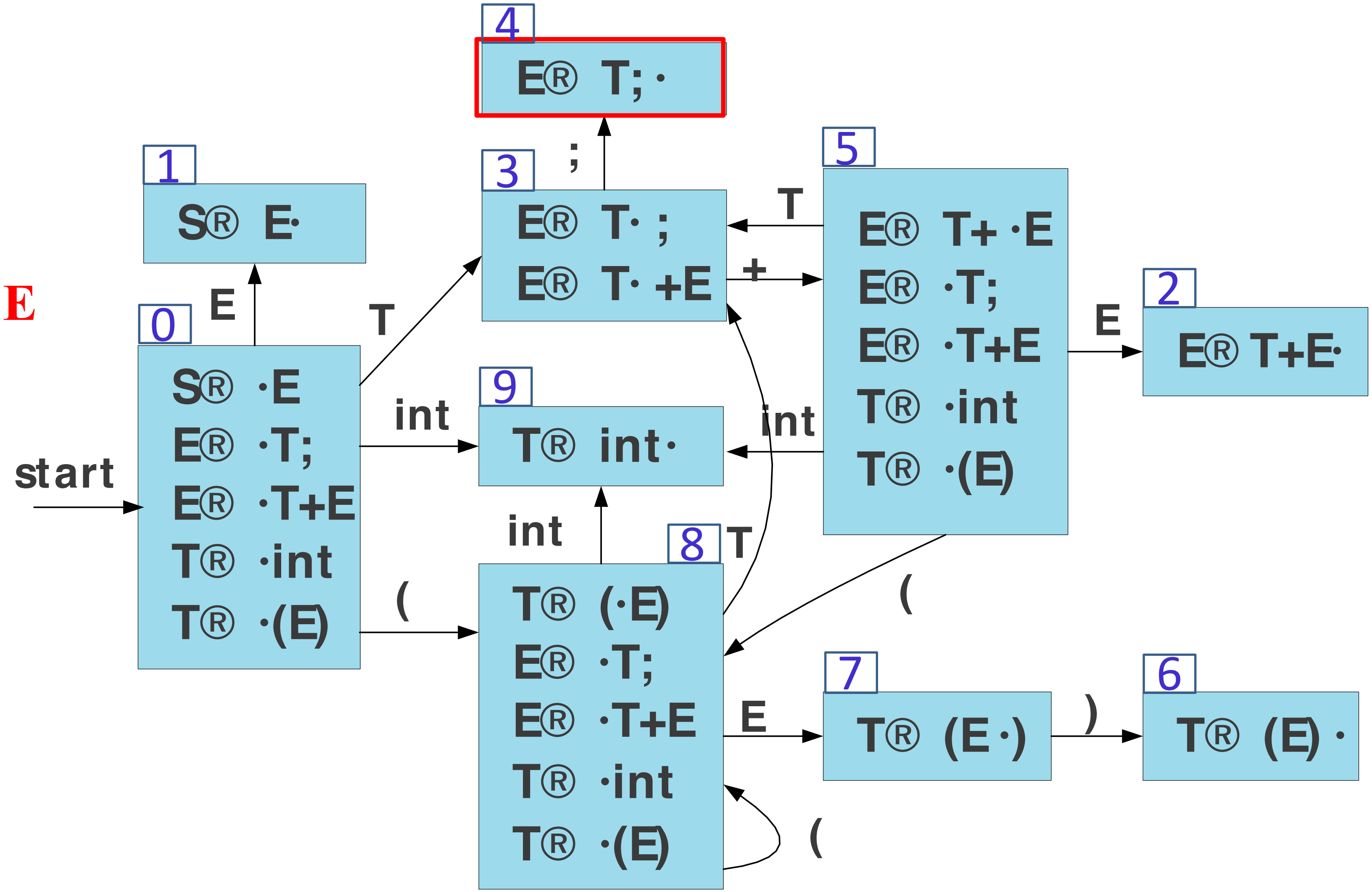
$S \rightarrow E$
$E \rightarrow T;$
$E \rightarrow T + E$
$T \rightarrow \text{int}$
$T \rightarrow (E)$



$	T	+	T		;	$
0	3	5	3			

LR(0) Parsing

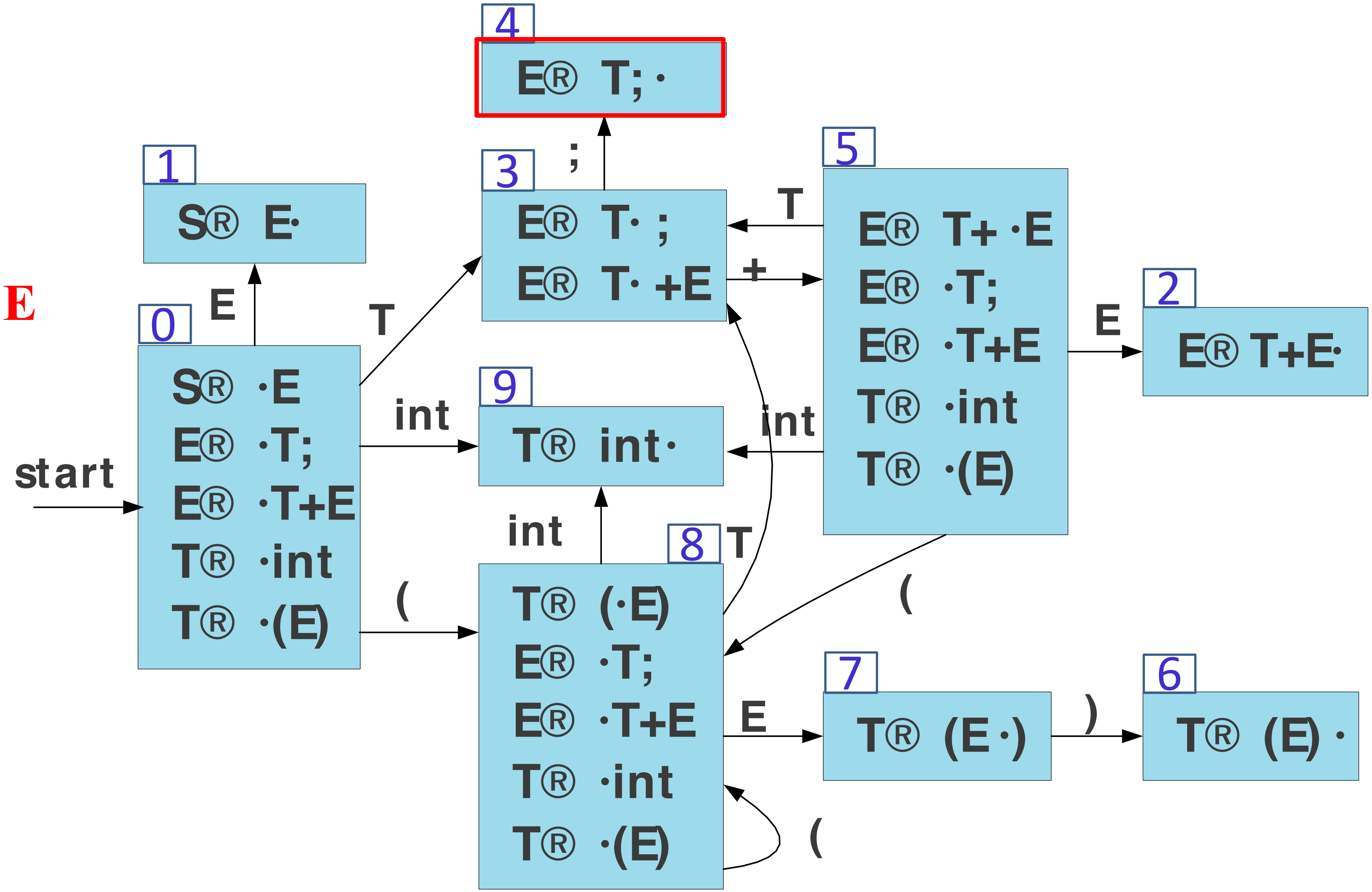
$S \rightarrow E$
$E \rightarrow T;$
$E \rightarrow T + E$
$T \rightarrow \text{int}$
$T \rightarrow (E)$



$	T	+	T	;	$
0	3	5	3	4	

LR(0) Parsing

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

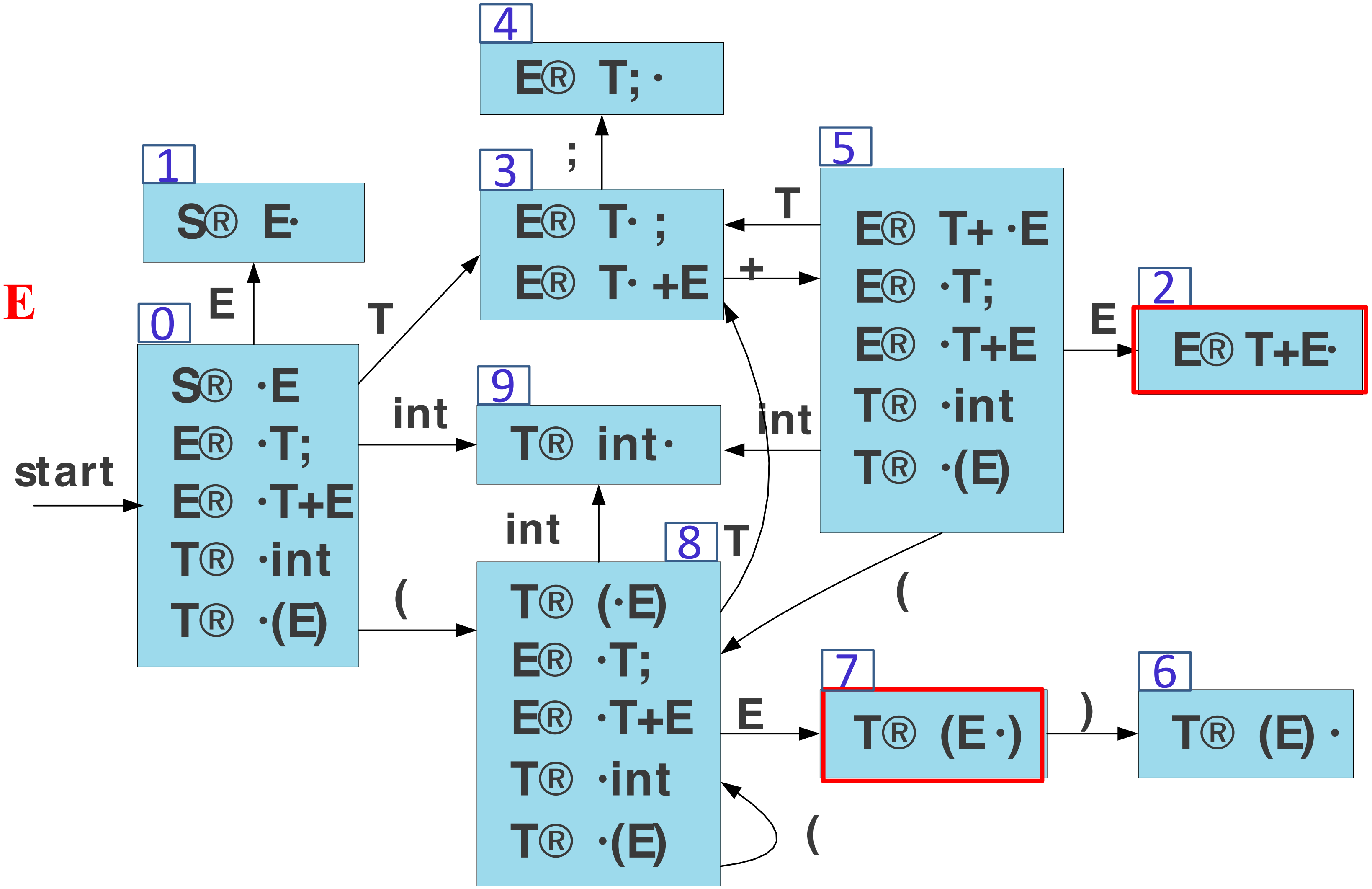


$	T	+
0	3	5

$

LR(0) Parsing

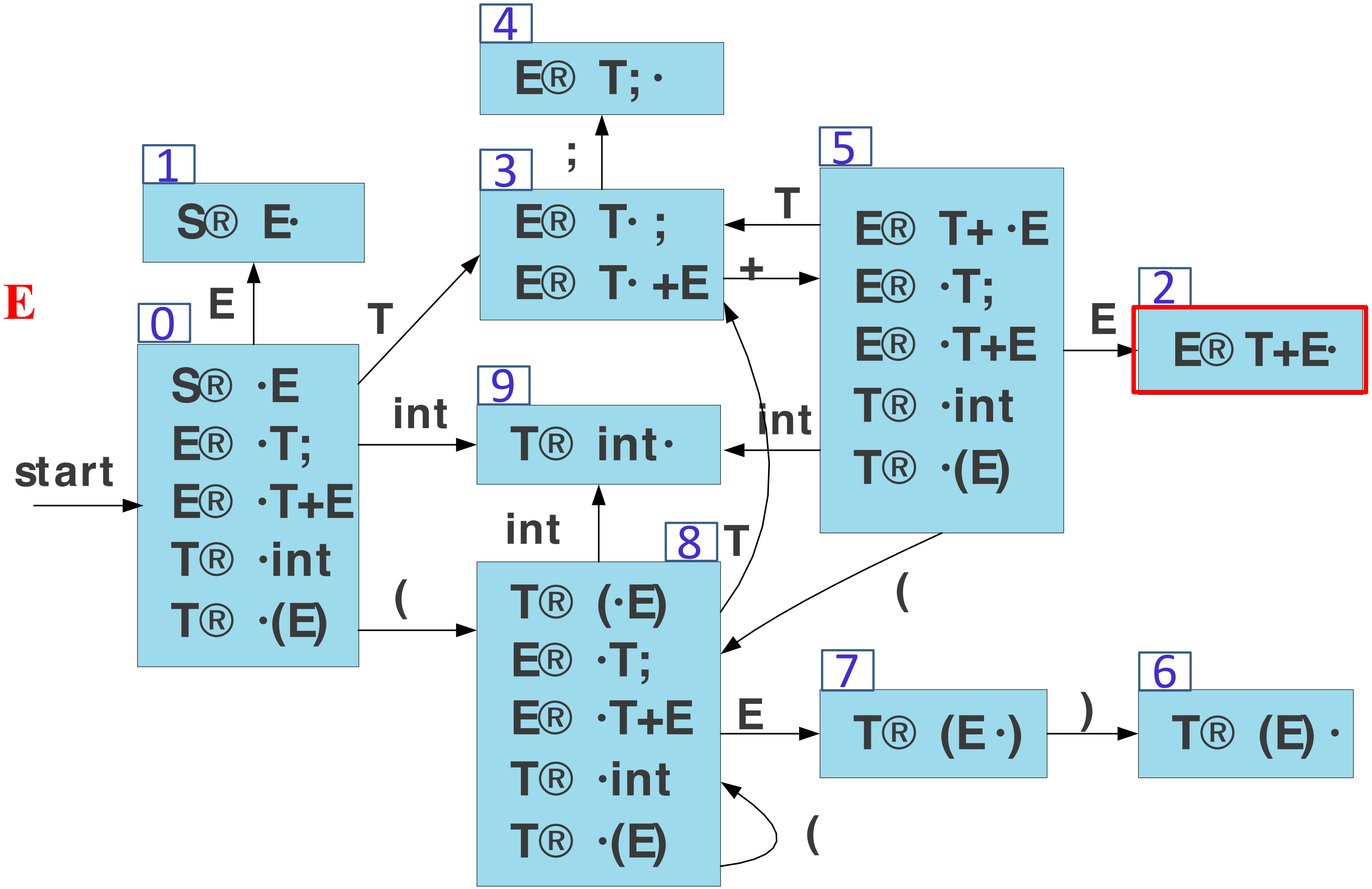
$S \rightarrow E$
$E \rightarrow T;$
$E \rightarrow T + E$
$T \rightarrow int$
$T \rightarrow (E)$



$	T	+	E		$
0	3	5	2		

LR(0) Parsing

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



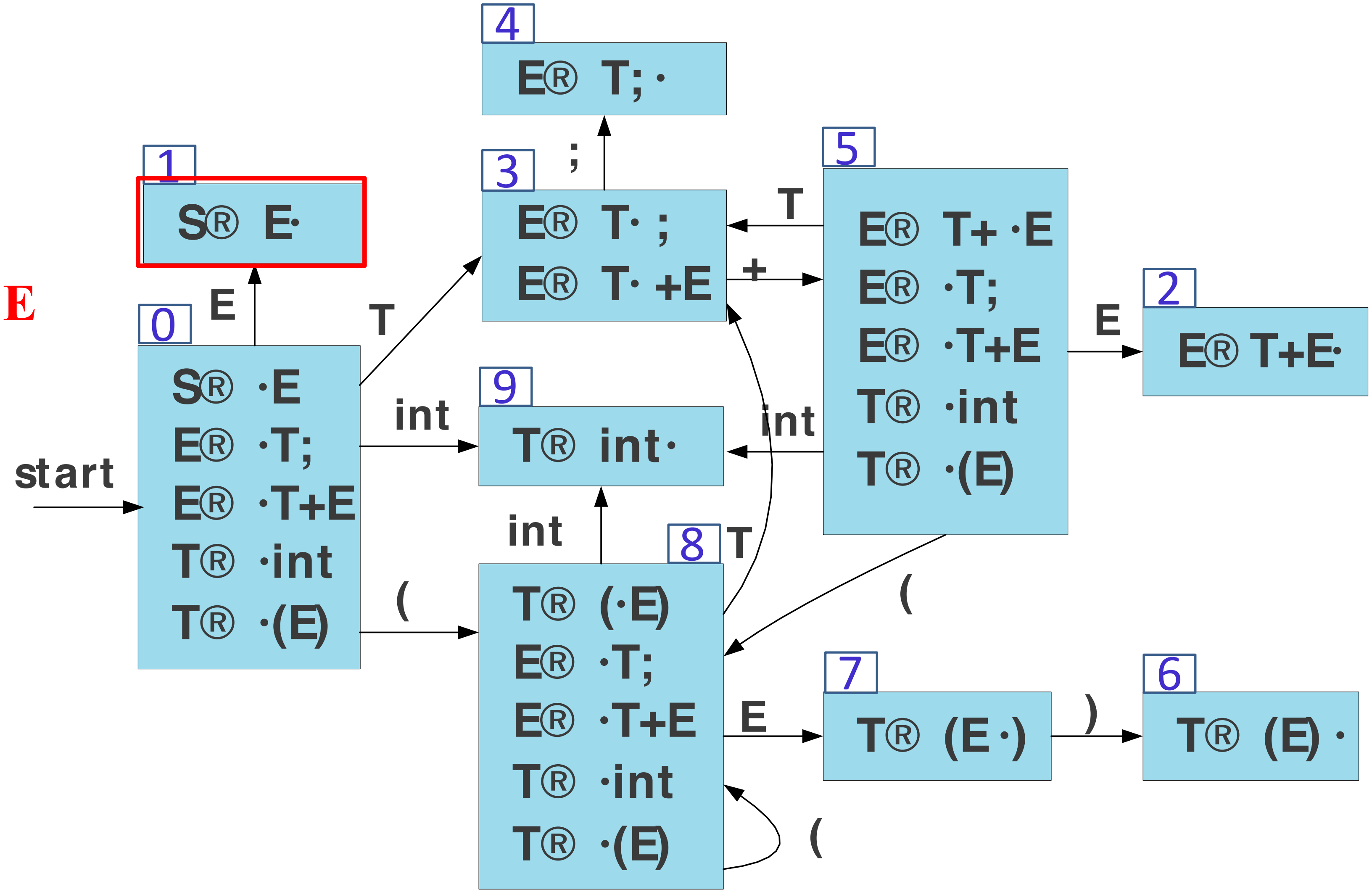
$

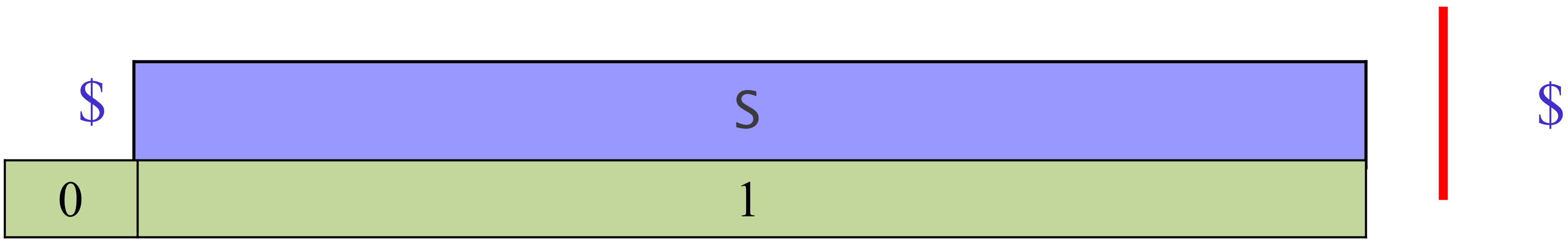
0

$

LR(0) Parsing

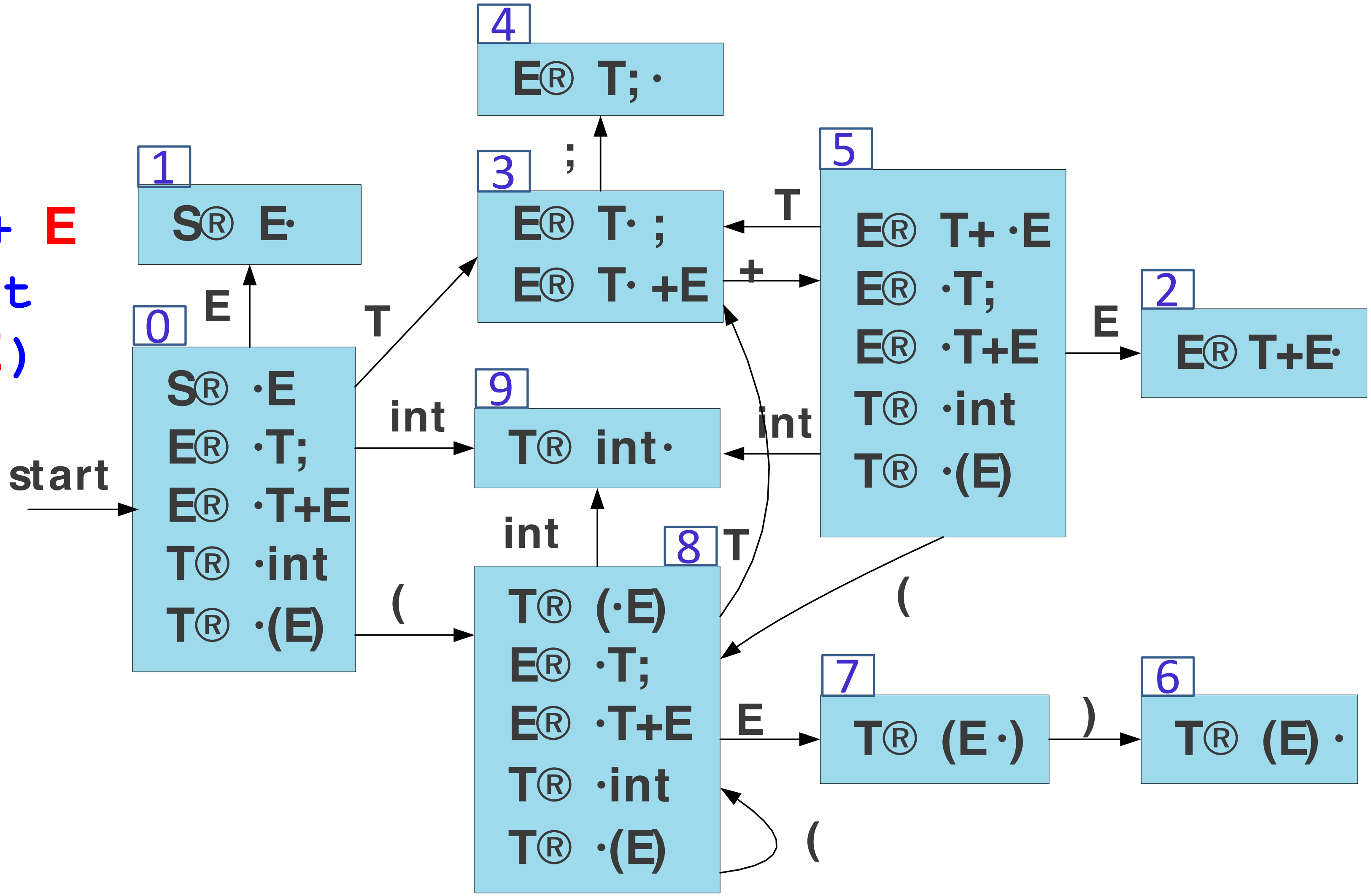
$S \rightarrow E$
$E \rightarrow T;$
$E \rightarrow T + E$
$T \rightarrow \text{int}$
$T \rightarrow (E)$





Building LR(0) Tables

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



LR Tables

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

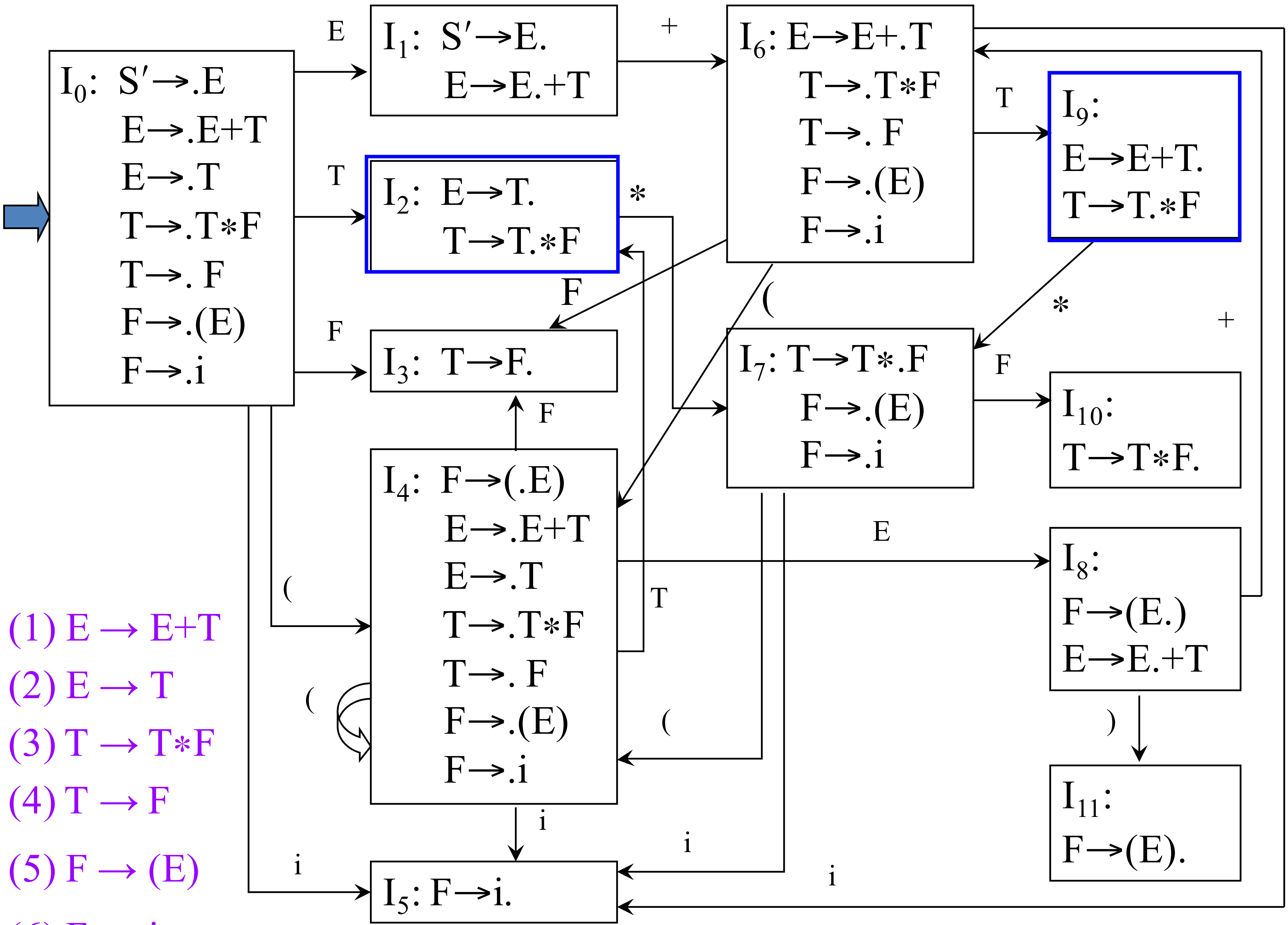
LR Tables

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

Representing the Automaton

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

The Limits of LR(0)



(1) $E \rightarrow E+T$

(2) $E \rightarrow T$

(3) $T \rightarrow T*F$

(4) $T \rightarrow F$

(5) $F \rightarrow (E)$

(6) $F \rightarrow i$

LR(0) table for the expression grammar

state	ACTION					
	i	+	*	(	)	$
2	r2	r2	r2/s7	r2	r2	r2
3	r4	r4	r4	r4	r4	r4
	……					
9	r1	r1	r1/s7	r1	r1	r1
10	r3	r3	r3	r3	r3	r3
11	r5	r5	r5	r5	r5	r5

LR Conflicts

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

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

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

(1) E → E+T

(2) E → T

(3) T → T*F

(4) T → F

(5) F → (E)

(6) F → i

state	ACTION						GOTO		
	i	+	*	(	)	$	E	T	F
0	s5			s4			1	2	3
1		s6				acc			
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4			8	2	3
5		r6	r6		r6	r6			
6	s5			s4				9	3
7	s5			s4					10
8		s6			s11				
9		r1	s7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

(1) $E \rightarrow E+T$
(2) $E \rightarrow T$
(3) $T \rightarrow T*F$
(4) $T \rightarrow F$
(5) $F \rightarrow (E)$
(6) $F \rightarrow i$

Step	state	stack	input
1)	0	\$	i*i+i \$
2)	05	\$i	*i+i \$
3)	03	\$F	*i+i \$
4)	02	\$T	*i+i \$
5)	027	\$T*	i+i \$

state	ACTION						GOTO		
	i	+	*	(	)	\$	E	T	F
0	s5			s4			1	2	3
1		s6				acc			
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4			8	2	3
5		r6	r6		r6	r6			
6	s5			s4				9	3
7	s5			s4					10
8		s6			s11				
9		r1	s7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

input : i*i+i

(1) E → E+T
(2) E → T
(3) T → T*F
(4) T → F
(5) F → (E)
(6) F → i

Step	state	stack	input
1)	0	$	i*i+i $
2)	05	$i	*i+i $
3)	03	$F	*i+i $
4)	02	$T	*i+i $
5)	027	$T*	i+i $
6)	0275	$T*i	+i $
7)	027 <u>10</u>	$T*F	+i $
8)	02	$T	+i $
9)	01	$E	+i $
10)	016	$E+	i $
11)	0165	$E+i	$
12)	0163	$E+F	$
13)	0169	$E+T	$
14)	01	$E	$

Conflicts During Shift-Reduce Parsing

- There are context-free grammars for which shift-reduce parsers cannot be used.
- Stack contents and the next input symbol may not decide action:

  shift/reduce conflict: Whether make a shift operation or a reduction.

  reduce/reduce conflict: The parser cannot decide which of several reductions to make.
- If a shift-reduce parser cannot be used for a grammar, that grammar is called as non-LR(k) grammar.

left to right scanning   right-most derivation   k lookhead

An ambiguous grammar can never be a LR grammar.

Shift-Reduce Parsers

There are two main categories of shift-reduce parsers

- **Operator-Precedence Parser**

  simple, but only a small class of grammars.

- **LR-Parsers**

  covers wide range of grammars.

  SLR – simple LR parser

  LR – most general LR parser

  LALR – intermediate LR parser (lookhead LR parser)

  SLR, LR and LALR work same, only their parsing tables are different.

LR Parsers

- The most powerful shift-reduce parsing (yet efficient) is: LR(k) parsing
- LR parsing is attractive because:
  - LR parsing is most general non-backtracking shift-reduce parsing, yet it is still efficient.
  - The class of grammars that can be parsed using LR methods is a proper superset of the class of grammars that can be parsed with predictive parsers.

    LL(1)-Grammars $\subset$ LR(1)-Grammars
  - An LR-parser can detect a syntactic error as soon as it is possible to do so a left-to-right scan of the input.

LR Parsers

LR-Parsers

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

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

G:

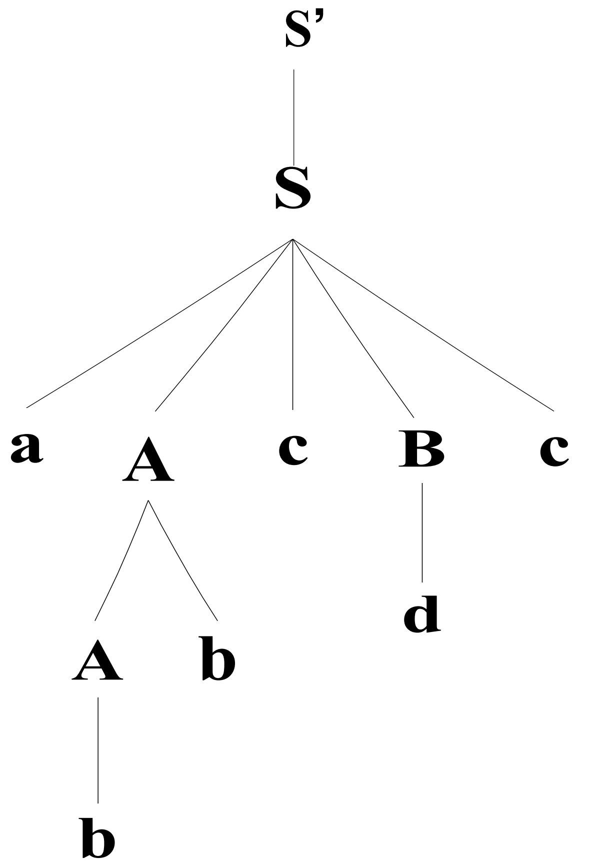
S’ → S

S → aAcBe

A → b

A → Ab

B → d



parsing :

$\text{a}\underline{\text{b}}\text{bcde} \Leftarrow \text{a}\underline{\text{Ab}}\text{cde} \Leftarrow \text{aAc}\underline{\text{d}}\text{e} \Leftarrow \underline{\text{aAcBe}} \Leftarrow \text{S} \Leftarrow \text{S'}$

G:

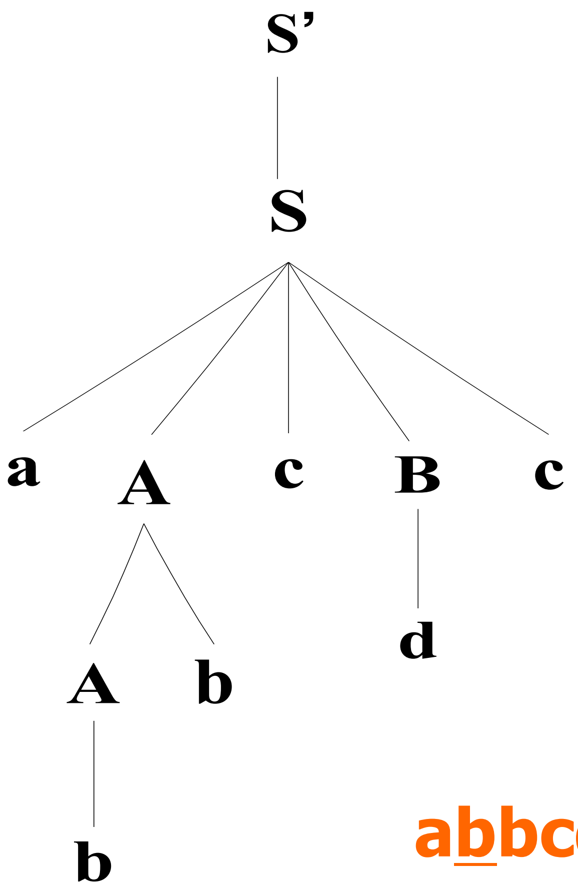
S’ → S

S → aAcBe

A → b

A → Ab

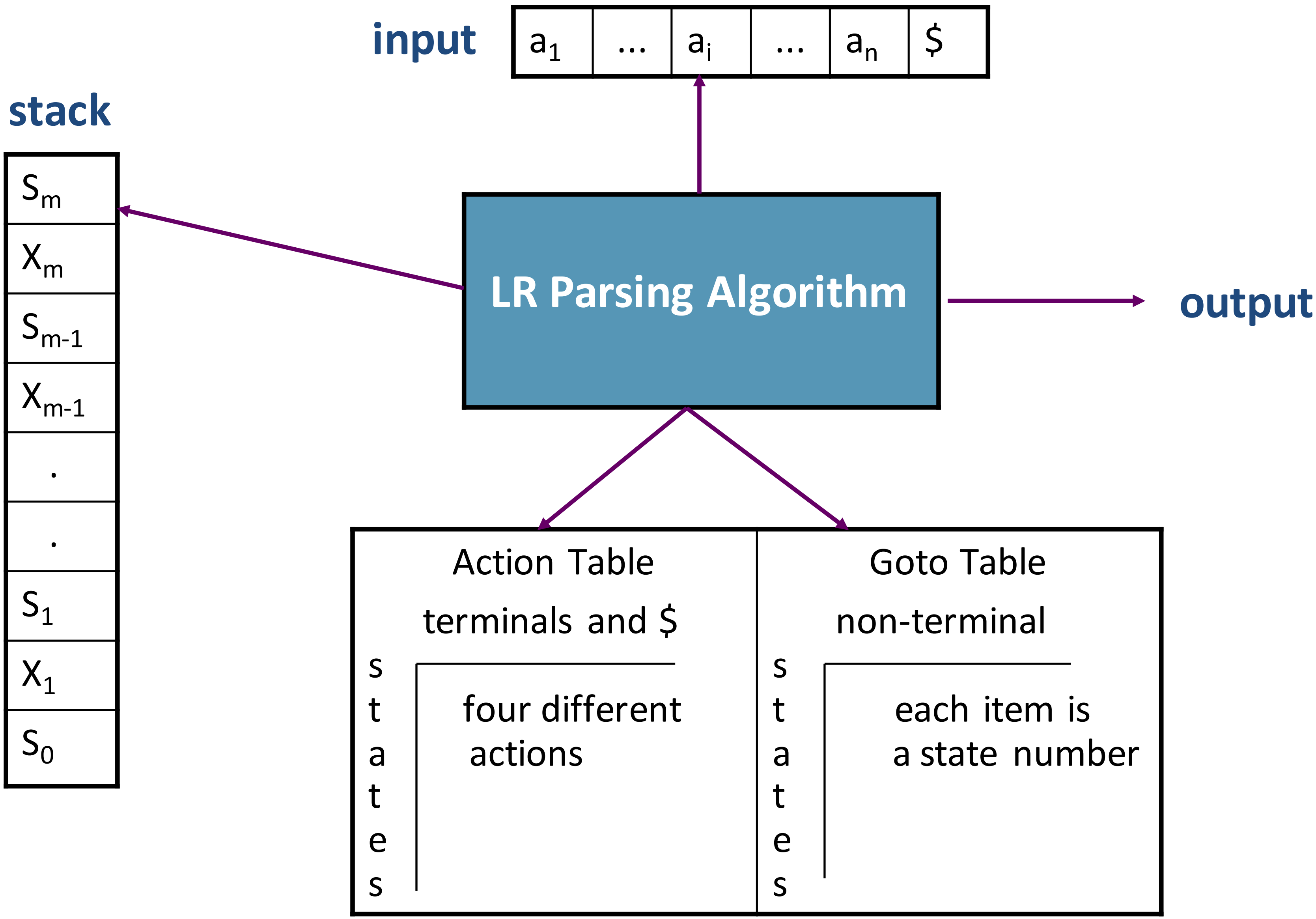
B → d



steps	stack	input	output
1)	\$	abbcde\$	
2)	\$a	bbcde\$	shift
3)	\$ab	bcde\$	shift
4)	\$aA	bcde\$	reduce(A→b)
5)	\$aAb	cde\$	shift
6)	\$aA	cde\$	reduce(A→Ab)
7)	\$aAc	de\$	shift
8)	\$aAcd	e\$	shift
9)	\$aAcB	e\$	reduce(B→d)
10)	\$aAcBe	\$	shift
11)	\$S	\$	reduce(S→aAcBe)
12)	\$S’	\$	accept

abbcde ⇐ aAbcde ⇐ aAcde ⇐ aAcBe ⇐ S ⇐ S’

LR Parsing Algorithm



(SLR) Parsing Tables for Expression Grammar

1) E → E+T
2) E → T
3) T → T*F
4) T → F
5) F → (E)
6) F → id

	Action Table						Goto Table		
state	**id**	**+**	*****	**(**	**)**	**$**	**E**	**T**	**F**
0	**s5**			**s4**			**1**	**2**	**3**
1		**s6**				**acc**			
2		**r2**	**s7**		**r2**	**r2**			
3		**r4**	**r4**		**r4**	**r4**			
4	**s5**			**s4**			**8**	**2**	**3**
5		**r6**	**r6**		**r6**	**r6**			
6	**s5**			**s4**				**9**	**3**
7	**s5**			**s4**					**10**
8		**s6**			**s11**				
9		**r1**	**s7**		**r1**	**r1**			
10		**r3**	**r3**		**r3**	**r3**			
11		**r5**	**r5**		**r5**	**r5**			

Actions of A (S)LR-Parser – Example

stack	input	action	output
0	id*id+id$	shift 5	
0id5	*id+id$	reduce by F→id	F→id
0F3	*id+id$	reduce by T→F	T→F
0T2	*id+id$	shift 7	
0T2*7	id+id$	shift 5	
0T2*7id5	+id$	reduce by F→id	F→id
0T2*7F10	+id$	reduce by T→T*F	T→T*F
0T2	+id$	reduce by E→T	E→T
0E1	+id$	shift 6	
0E1+6	id$	shift 5	
0E1+6id5	$	reduce by F→id	F→id
0E1+6F3	$	reduce by T→F	T→F
0E1+6T9	$	reduce by E→E+T	E→E+T
0E1	$	accept	

Reference

- Compilers Principles, Techniques & Tools (Second Edition) Alfred Aho., Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman, Addison-Wesley, 2007
- Coursera Course – Compiler, http://www. Coursera.org
- Stanford Course CS143 by Keith Schwarz, http://cs143.stanford.edu