CSL302: Compiler Design

Bottom Up Parsing

Vishwesh Jatala

Assistant Professor

Department of CSE

Indian Institute of Technology Bhilai

vishwesh@iitbhilai.ac.in

Acknowledgement

- Today's slides are modified from that of *Stanford University:*
  - *https://web.stanford.edu/class/archive/cs/cs143/cs143.1128/*

Bottom-Up Parsing - Example

$E \rightarrow T$

$E \rightarrow E + T$

$T \rightarrow \texttt{int}$

$T \rightarrow (E)$

```
  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.

Exercise

$E \longrightarrow T$

$T \longrightarrow T * F$

$T \longrightarrow F$

$F \longrightarrow$ `id`

`id * id`

Handles

- The basic steps of a bottom-up parser are
  - to identify a substring within a rightmost sentential form which matches the RHS of a rule.
  - when this substring is replaced by the LHS of the matching rule, it must produce the previous rightmost-sentential form.
- Such a substring is called a handle
- A left-to-right, bottom-up parse works by iteratively searching for a handle, then reducing the handle.

Finding Handles

- Where do we look for handles?
- 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 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

$E \rightarrow F$

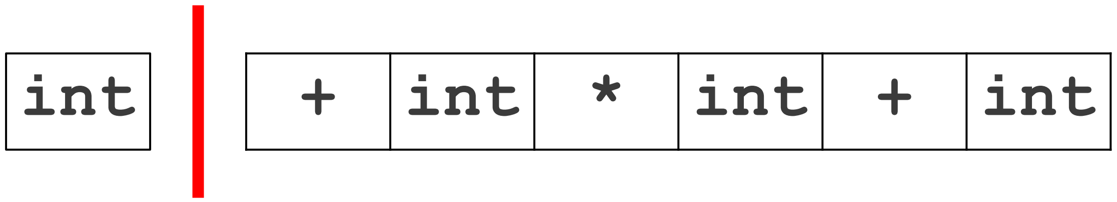
$E \rightarrow E$ `+` F

$F \rightarrow F$ `*` T

$F \rightarrow T$

$T \rightarrow$ `int`

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



A Sample Shift/Reduce Parse

$E \rightarrow F$

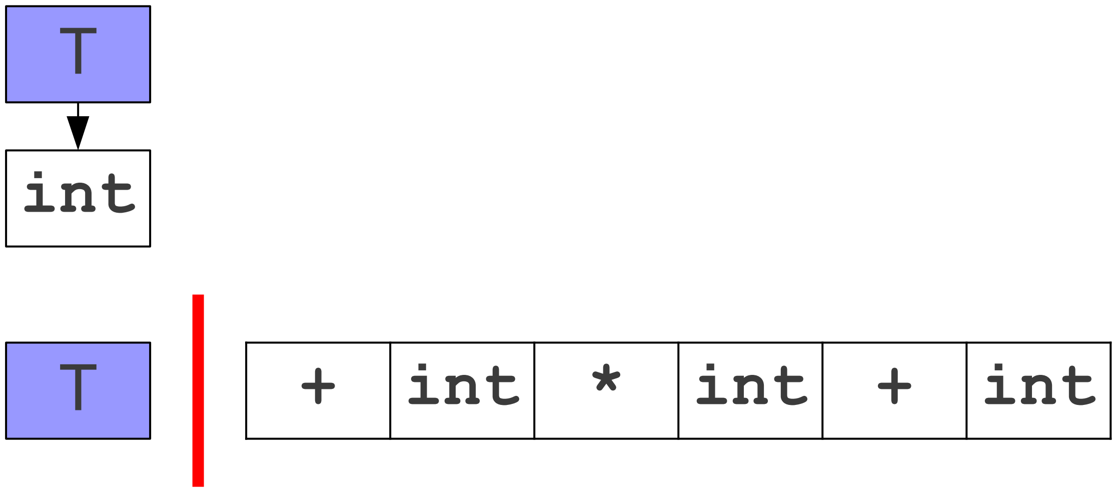
$E \rightarrow E + F$

$F \rightarrow F * T$

$F \rightarrow T$

$T \rightarrow$ **int**

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



A Sample Shift/Reduce Parse

$E \rightarrow F$

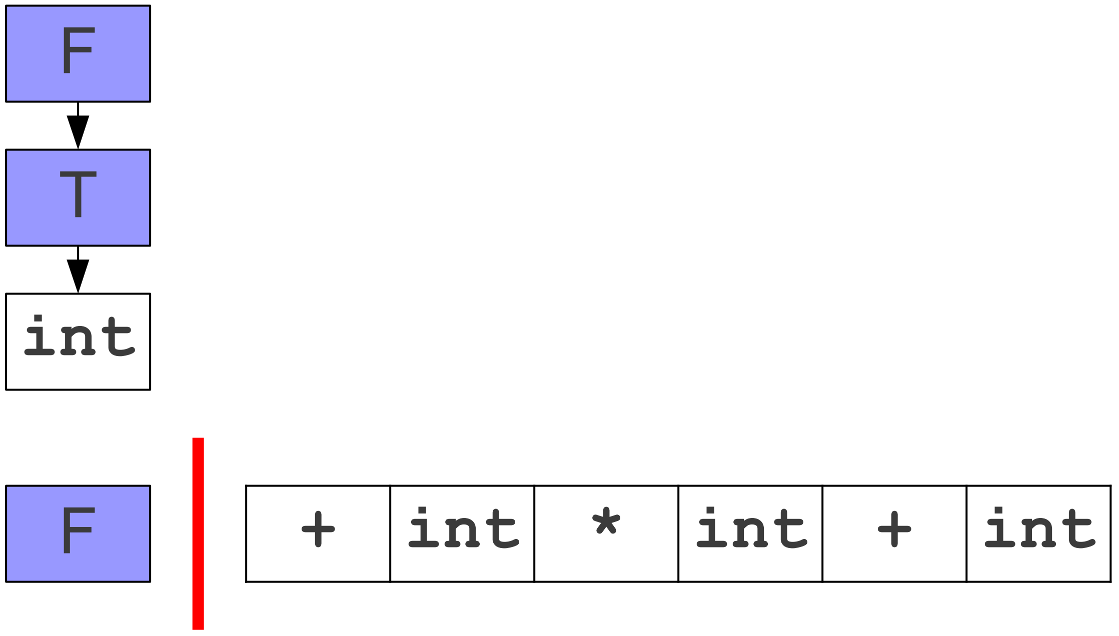
$E \rightarrow E + F$

$F \rightarrow F * T$

$F \rightarrow T$

$T \rightarrow$ int

$T \rightarrow (E)$



A Sample Shift/Reduce Parse

E → F

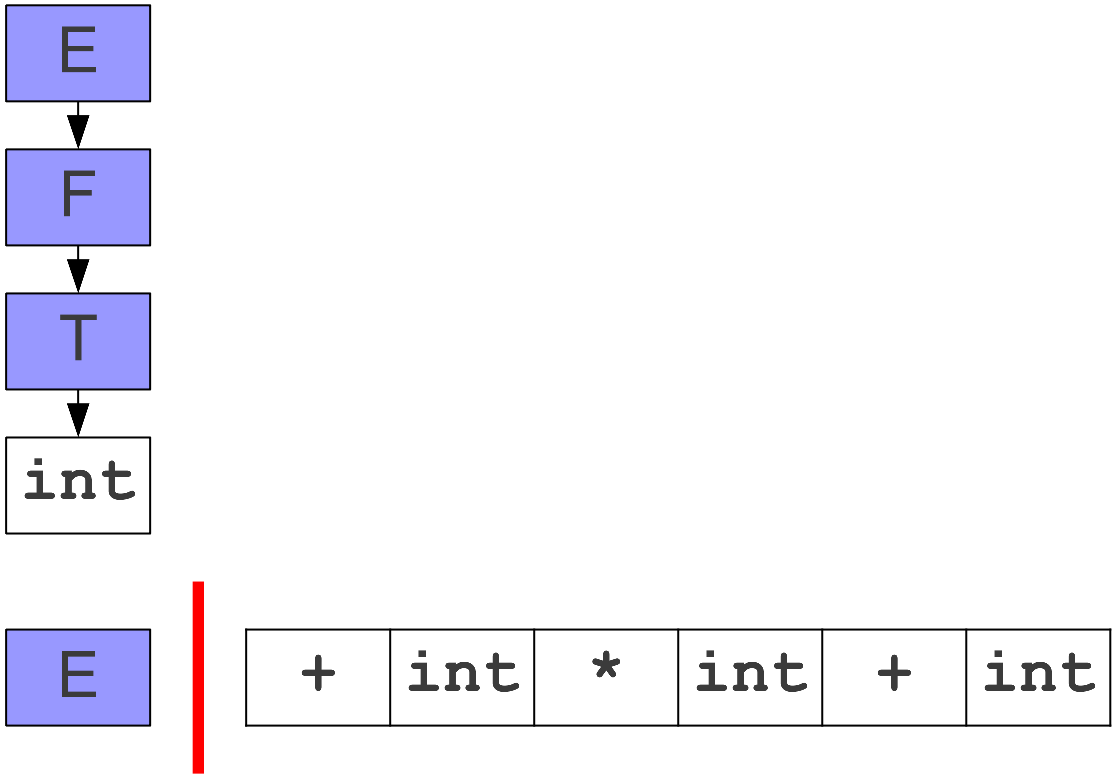
E → E + F

F → F * T

F → T

T → int

T → (E)



A Sample Shift/Reduce Parse

$E \rightarrow F$

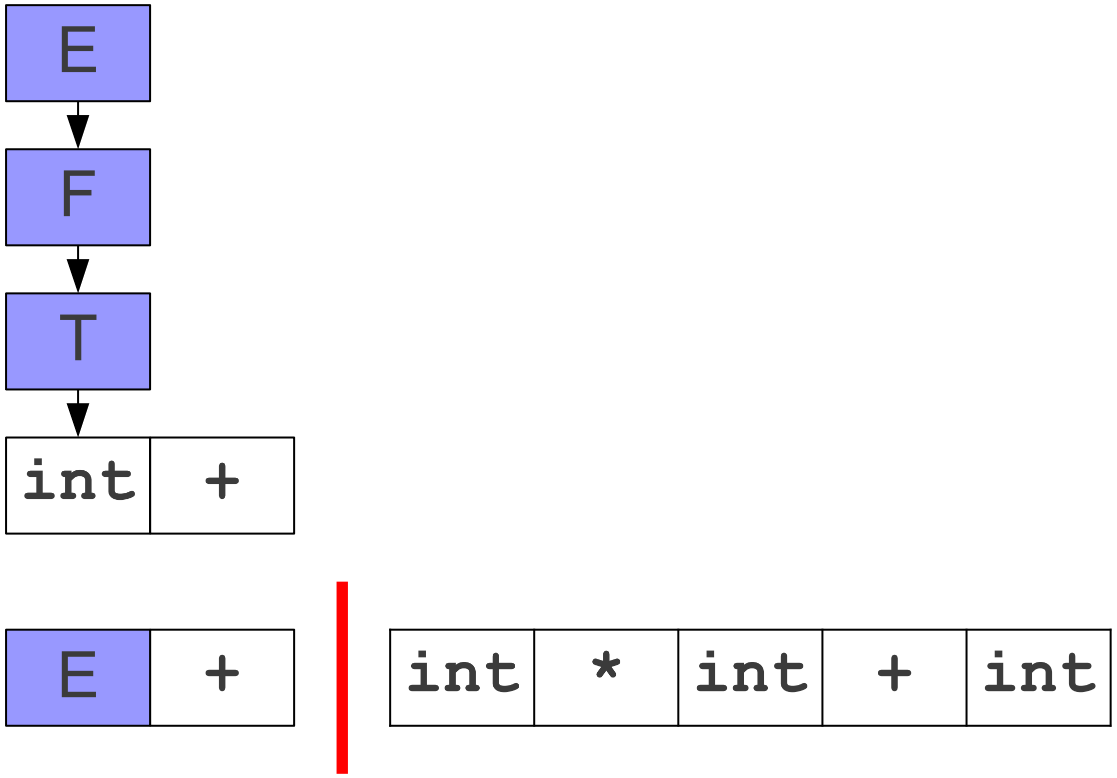
$E \rightarrow E + F$

$F \rightarrow F * T$

$F \rightarrow T$

$T \rightarrow$ `int`

$T \rightarrow (E)$



A Sample Shift/Reduce Parse

$E \rightarrow$ F

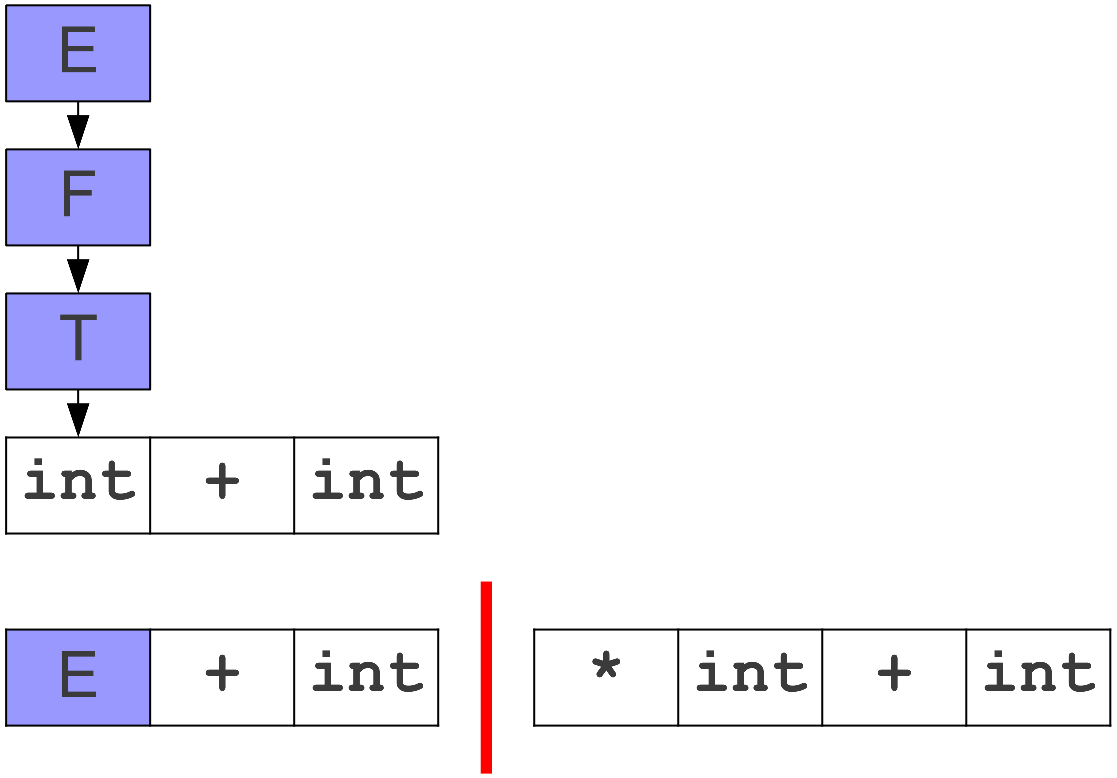
$E \rightarrow$ E + F

$F \rightarrow$ F * T

$F \rightarrow$ T

$T \rightarrow$ int

$T \rightarrow$ (E)



A Sample Shift/Reduce Parse

E → F

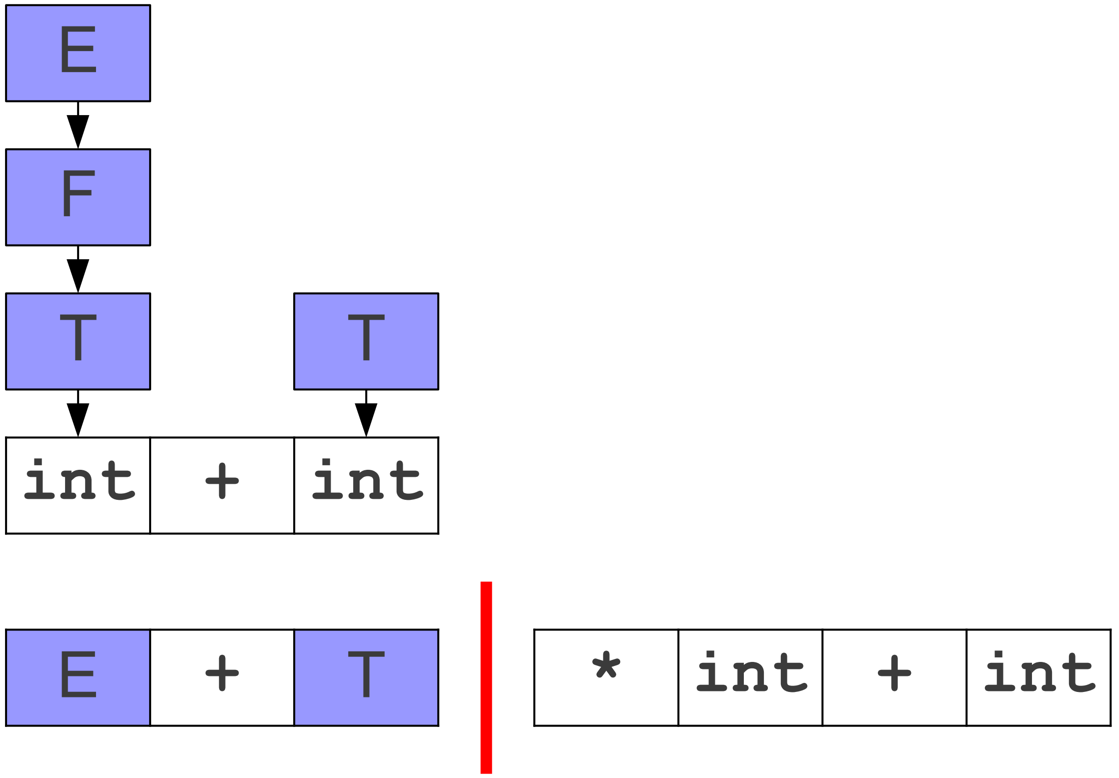
E → E + F

F → F * T

F → T

T → int

T → (E)



A Sample Shift/Reduce Parse

$\mathbf{E} \longrightarrow \mathbf{F}$

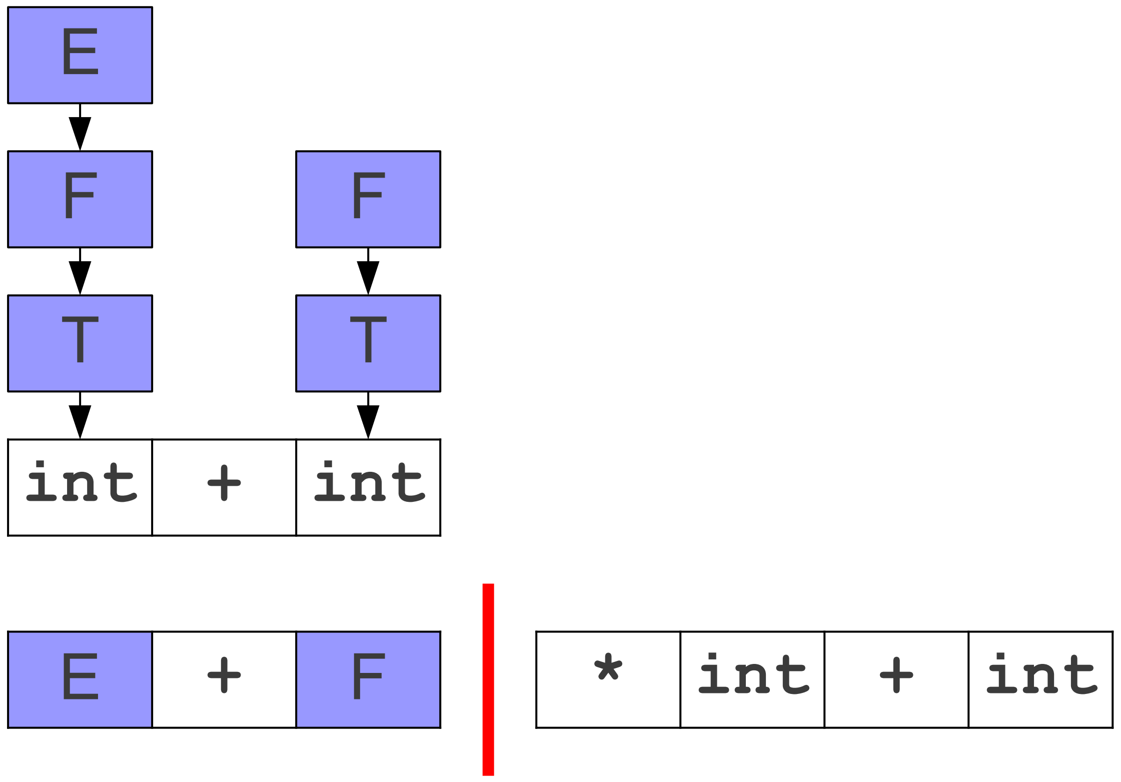
$\mathbf{E} \longrightarrow \mathbf{E} + \mathbf{F}$

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

$\mathbf{F} \longrightarrow \mathbf{T}$

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

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



A Sample Shift/Reduce Parse

E → F

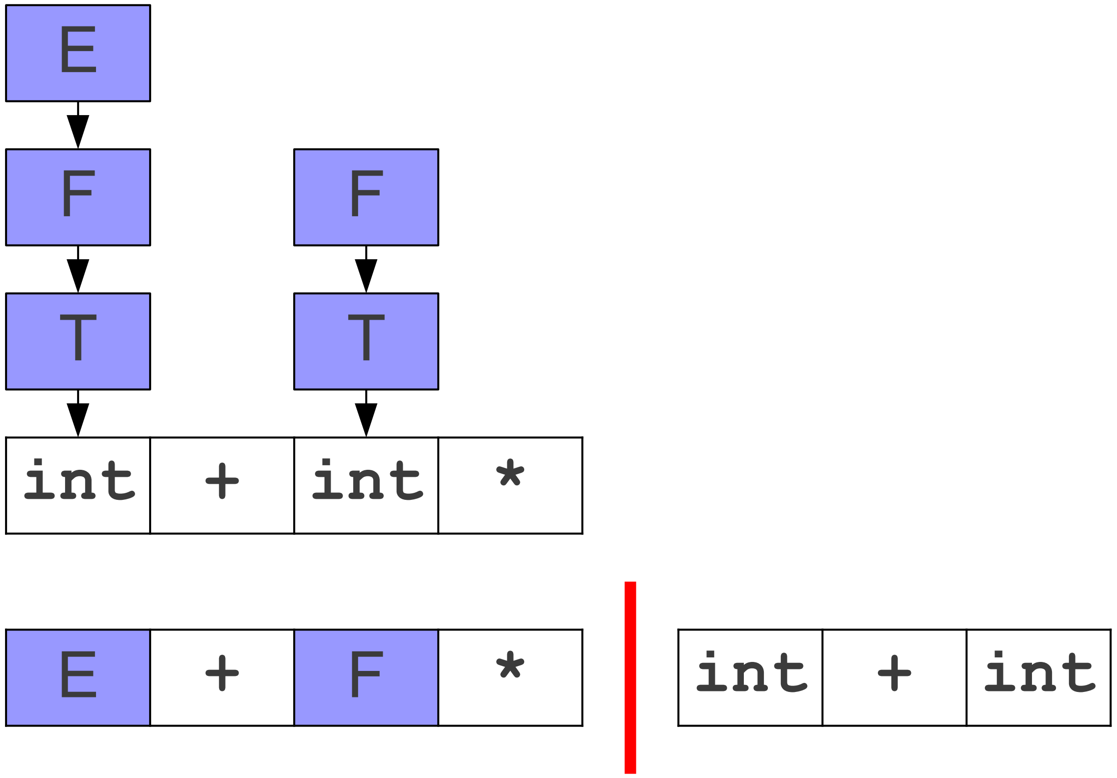
E → E + F

F → F * T

F → T

T → int

T → (E)



A Sample Shift/Reduce Parse

E → F

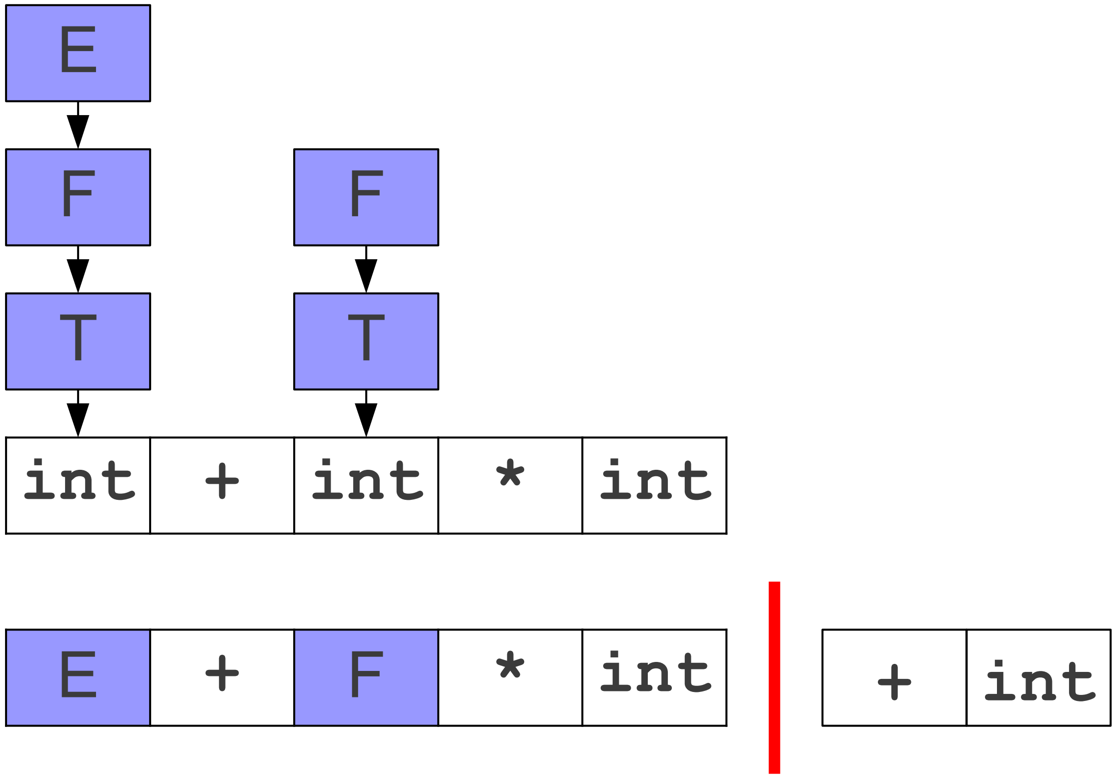
E → E + F

F → F * T

F → T

T → int

T → (E)



A Sample Shift/Reduce Parse

$E \rightarrow F$

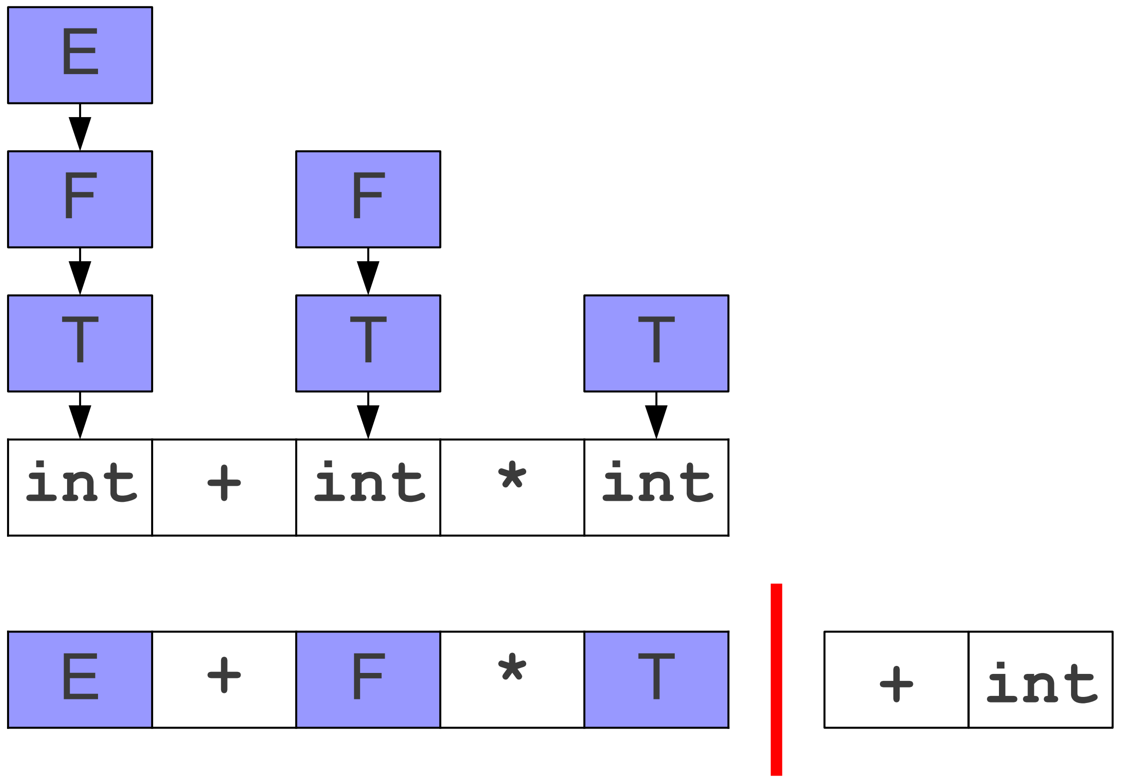
$E \rightarrow E + F$

$F \rightarrow F * T$

$F \rightarrow T$

$T \rightarrow$ `int`

$T \rightarrow (E)$



A Sample Shift/Reduce Parse

$E \rightarrow F$

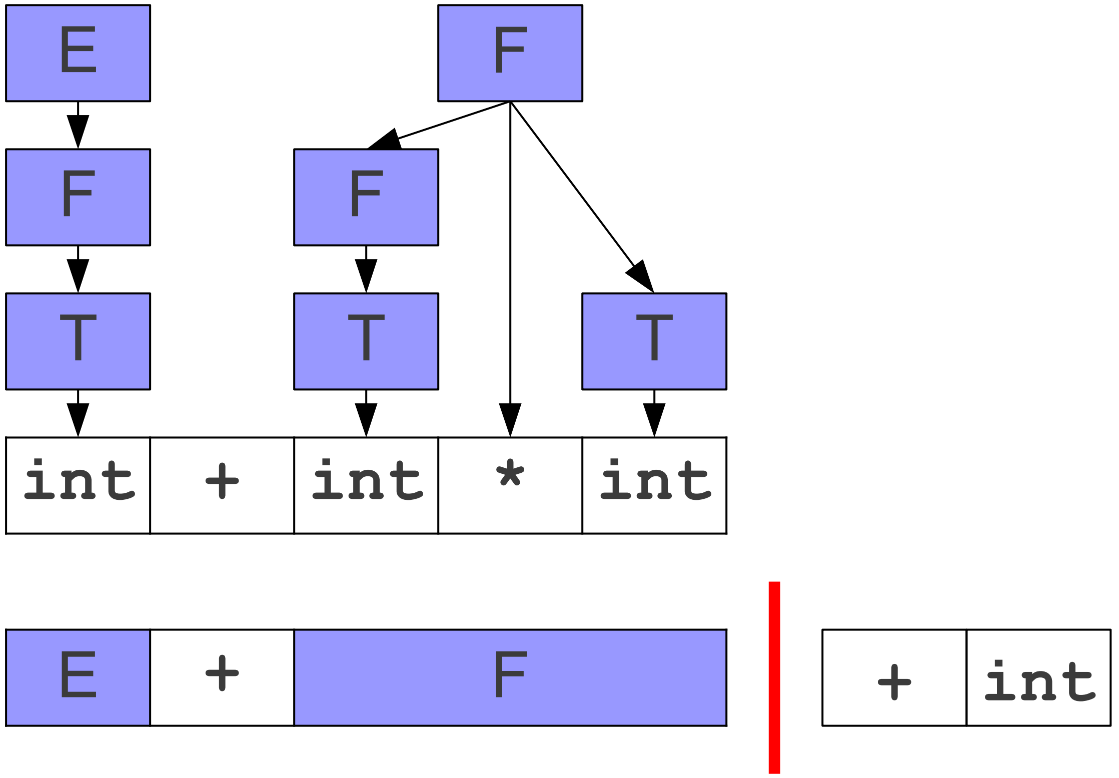
$E \rightarrow E$ + F

$F \rightarrow F$ * T

$F \rightarrow T$

$T \rightarrow$ int

$T \rightarrow$ (E)



A Sample Shift/Reduce Parse

$E \longrightarrow F$

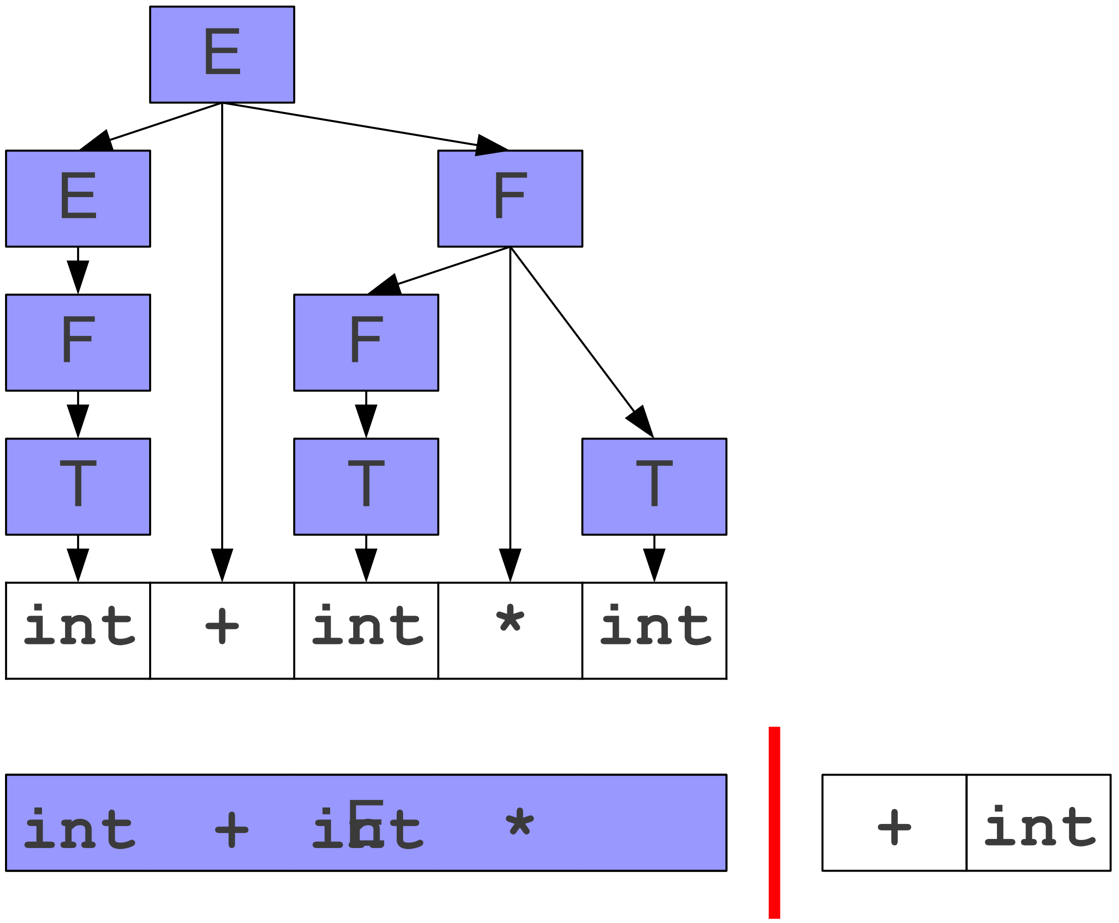
$E \longrightarrow E$ **+** F

$F \longrightarrow F$ ***** T

$F \longrightarrow T$

$T \longrightarrow$ **int**

$T \longrightarrow$ **(**E**)**



A Sample Shift/Reduce Parse

$E \rightarrow F$

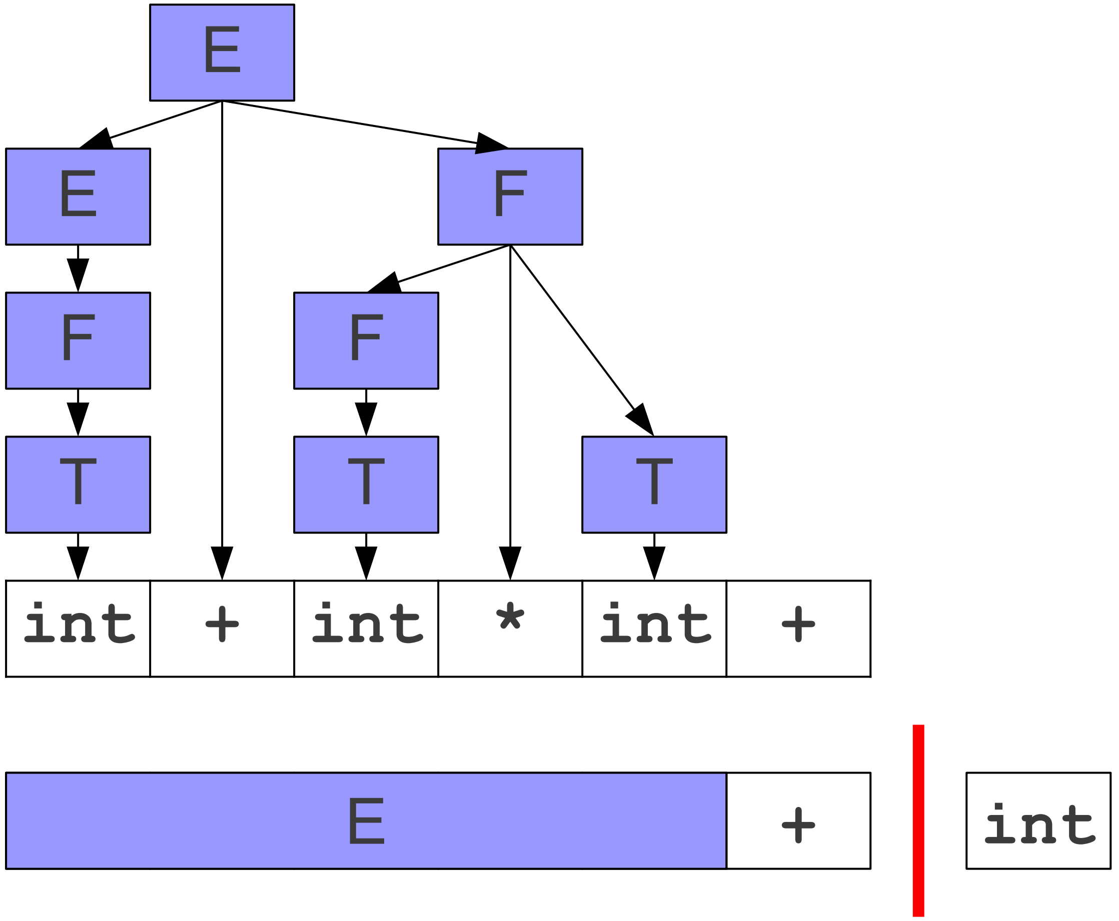
$E \rightarrow E + F$

$F \rightarrow F * T$

$F \rightarrow T$

$T \rightarrow$ `int`

$T \rightarrow (E)$



A Sample Shift/Reduce Parse

$\mathbf{E} \longrightarrow \mathbf{F}$

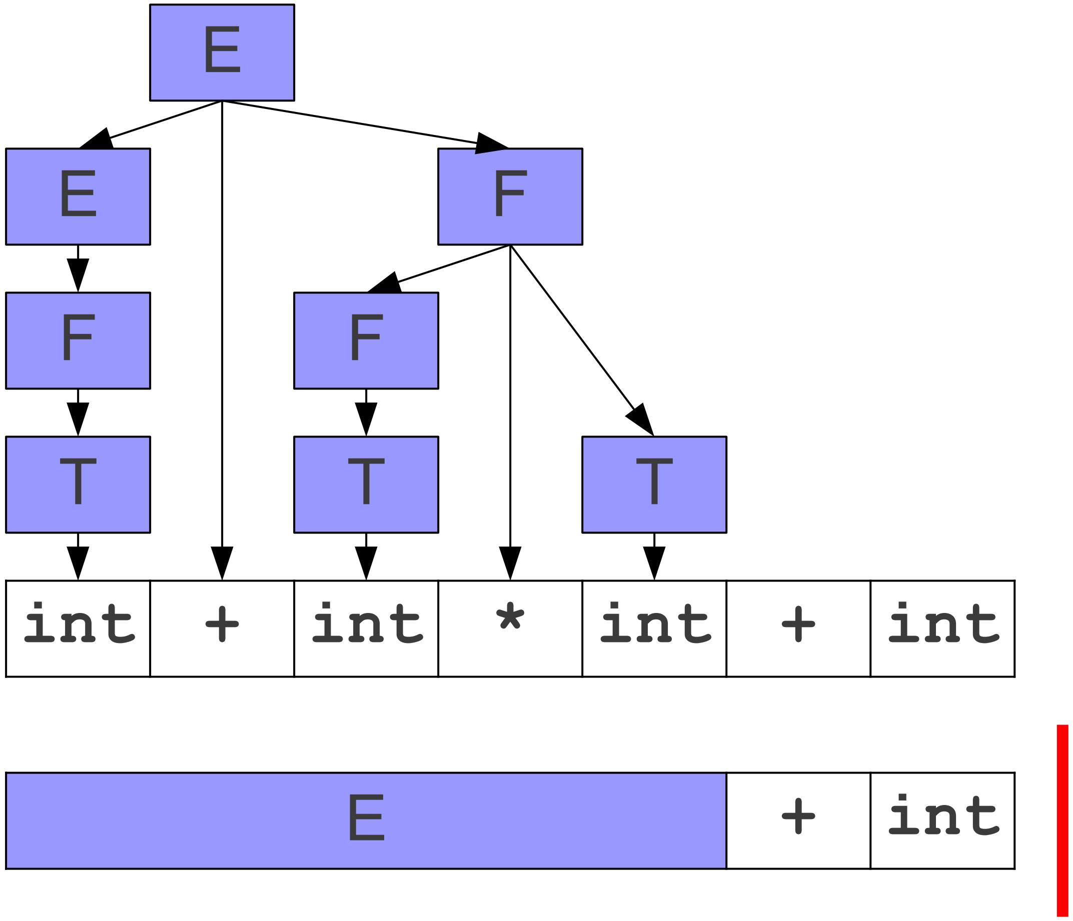
$\mathbf{E} \longrightarrow \mathbf{E} + \mathbf{F}$

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

$\mathbf{F} \longrightarrow \mathbf{T}$

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

$\mathbf{T} \longrightarrow$ **`(E)`**



A Sample Shift/Reduce Parse

E → F

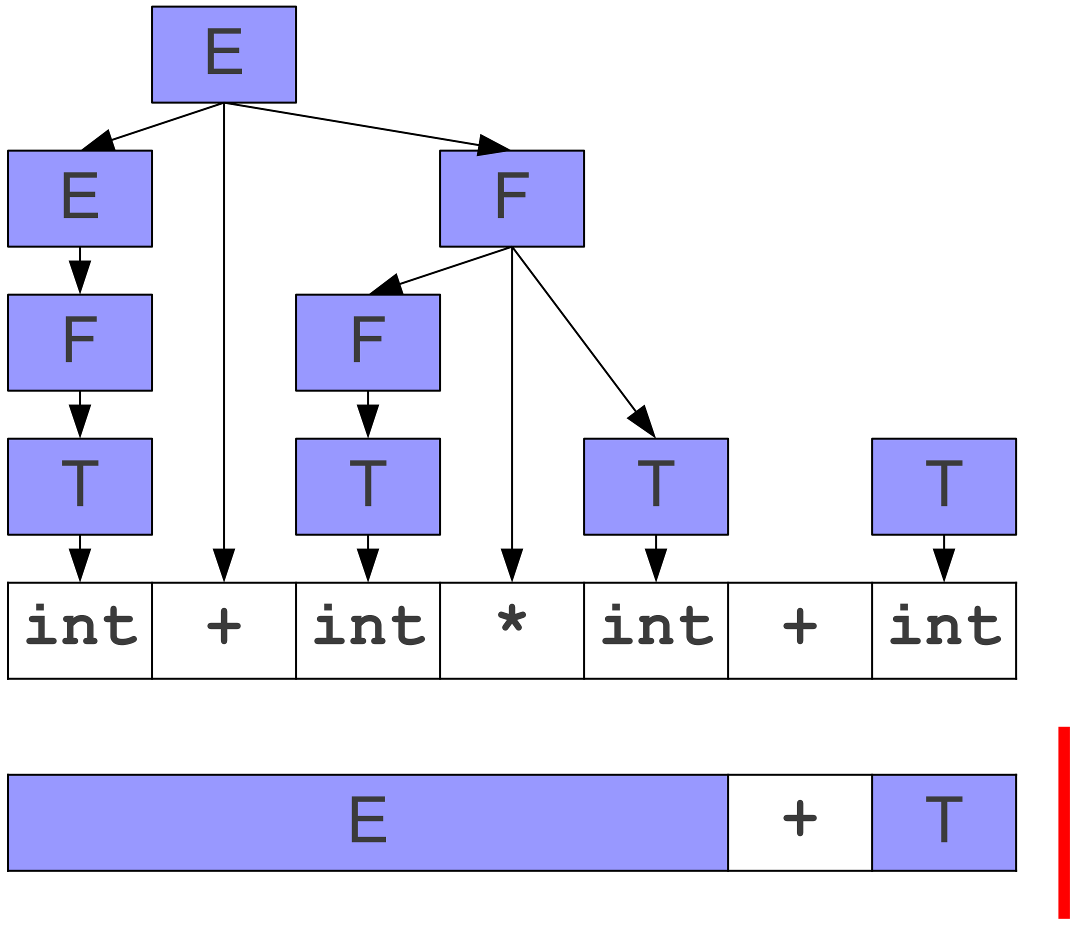
E → E + F

F → F * T

F → T

T → int

T → (E)



A Sample Shift/Reduce Parse

$E \rightarrow F$

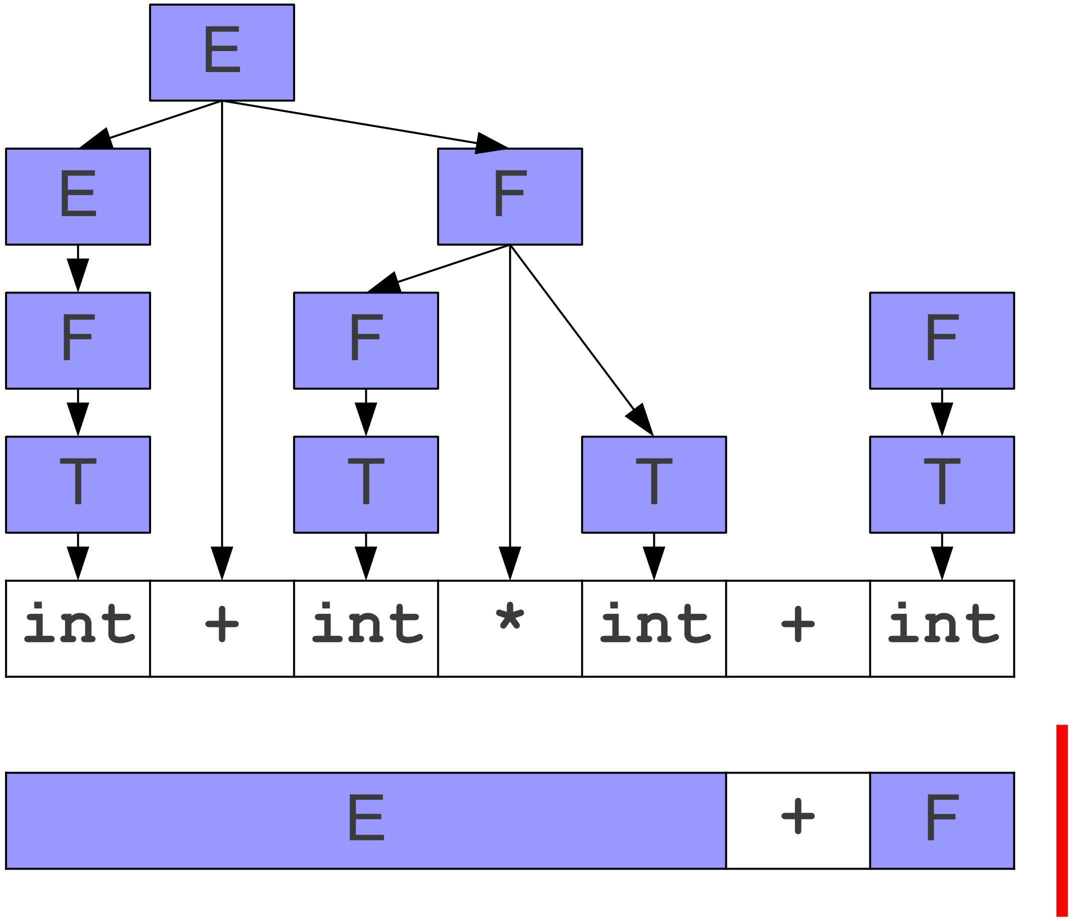
$E \rightarrow E$ + F

$F \rightarrow F$ * T

$F \rightarrow T$

$T \rightarrow$ int

$T \rightarrow$ (E)



A Sample Shift/Reduce Parse

$E \rightarrow F$

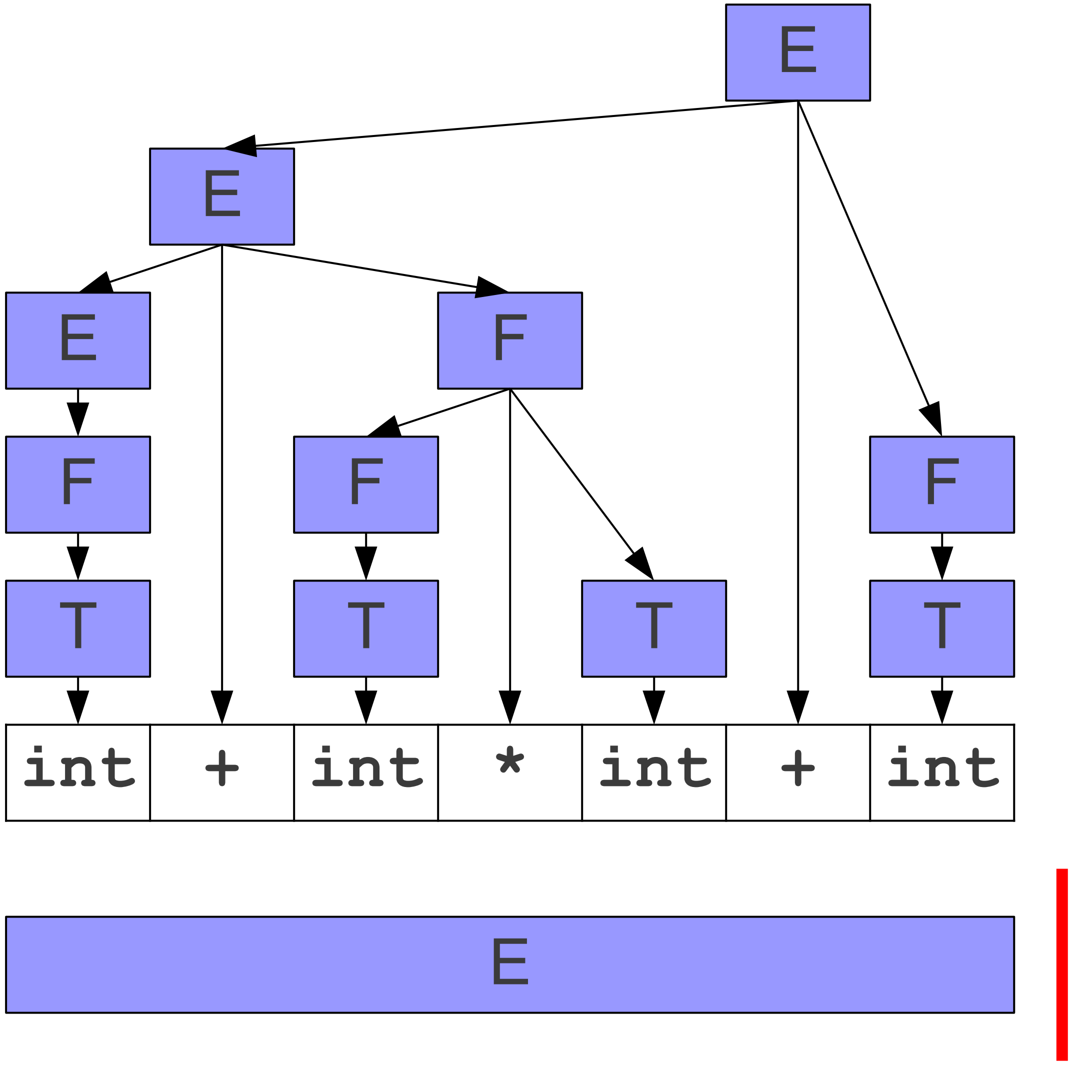
$E \rightarrow E + F$

$F \rightarrow F * T$

$F \rightarrow T$

$T \rightarrow$ `int`

$T \rightarrow (E)$



Shift/Reduce Parsing

- 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?

Exploring the Left Side

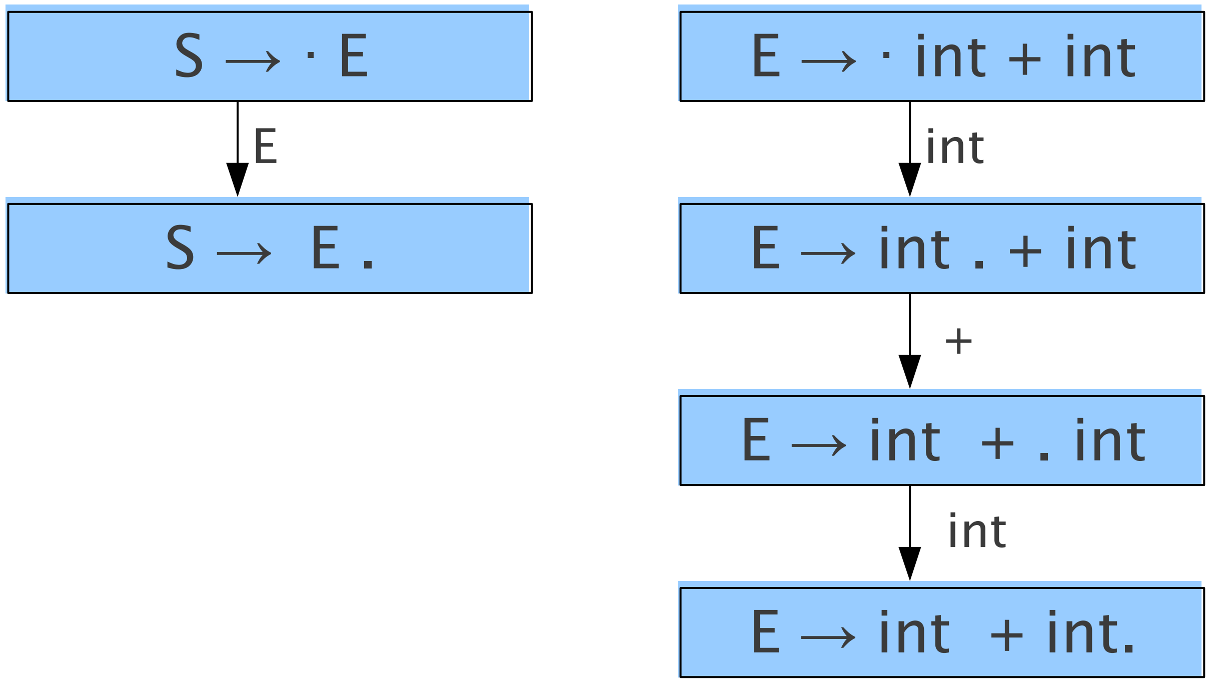
- The handle will always appear at the end of string in the left side of the parser.
- Can *any* string appear on the left side of the parser, or are there restrictions on what sorts of strings can appear there?
- If we can find a pattern to the strings that can appear on the left side, we might be able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

How to Track Handles?

$$\begin{tikzcd}
\boxed{E \rightarrow \cdot\, int + int} \arrow[d, "int"] \\
\boxed{E \rightarrow int\ .\ + int} \arrow[d, "+"] \\
\boxed{E \rightarrow int\ +\ .\ int} \arrow[d, "int"] \\
\boxed{E \rightarrow int\ + int.}
\end{tikzcd}$$

$E \rightarrow int + \ int$

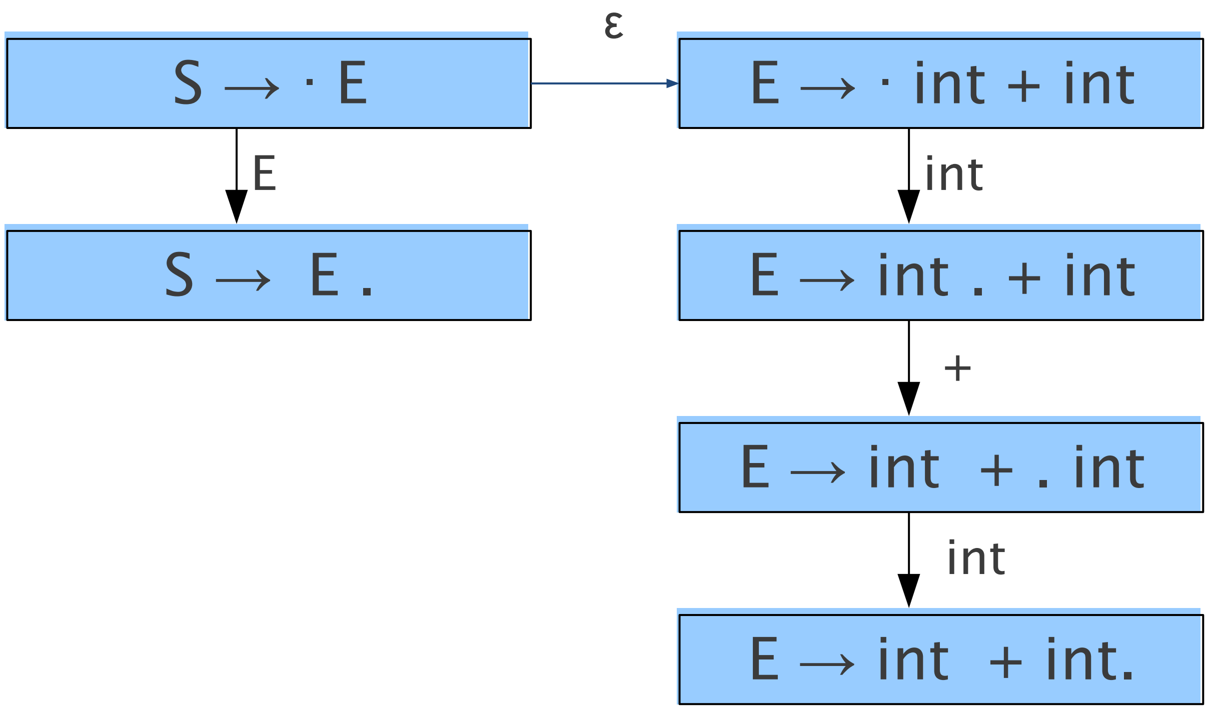
How to Track Handles?



S → E

E → int + int

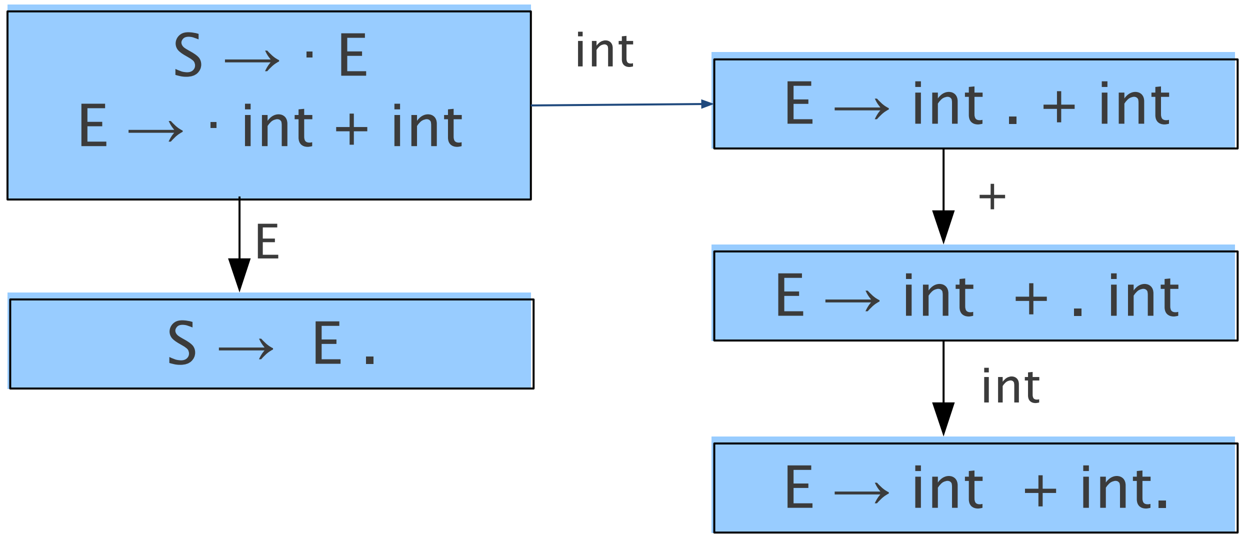
How to Track Handles?



$S \rightarrow E$

$E \rightarrow int + int$

How to Track Handles?



S → E
E → int + int

Constructing the Automaton

- Begin in a state containing **S** $\rightarrow \cdot$ **A**, where **S** is the augmented start symbol.
- Compute the **closure** of the state:
  - If **A** $\rightarrow \alpha \cdot$ **B**ω is in the state, add **B** $\rightarrow \cdot \gamma$ to the state for each production **B** $\rightarrow \gamma$.

A Deterministic Automaton

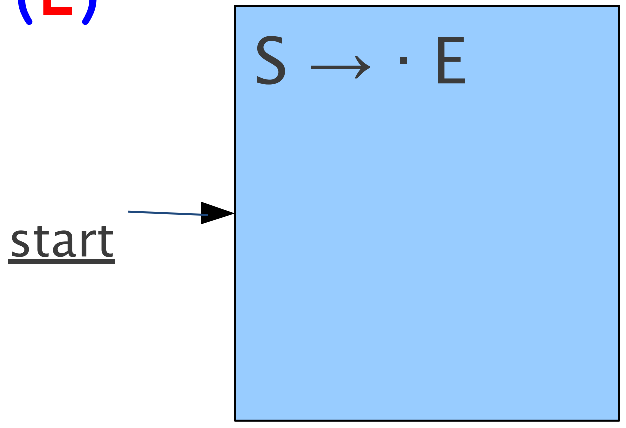
S → E

E → T;

E → T + E

T → int

T → (E)



A Deterministic Automaton

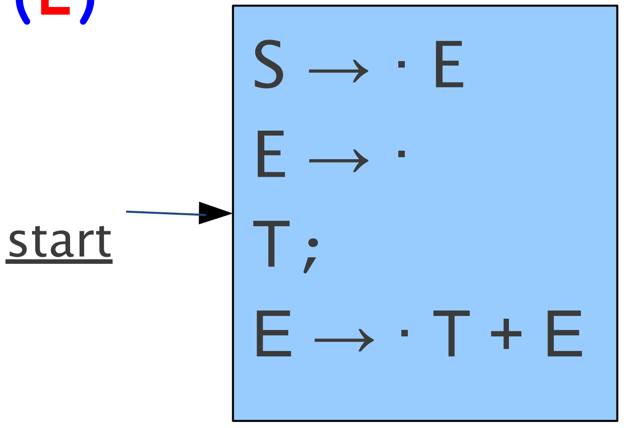
$S \to E$

$E \to T;$

$E \to T + E$

$T \to$ int

$T \to (E)$



A Deterministic Automaton

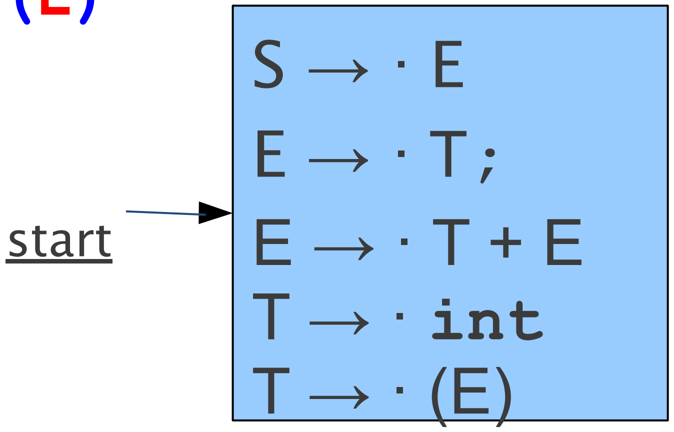
$S \rightarrow E$

$E \rightarrow T$ `;`

$E \rightarrow T$ `+` E

$T \rightarrow$ `int`

$T \rightarrow$ `(E)`

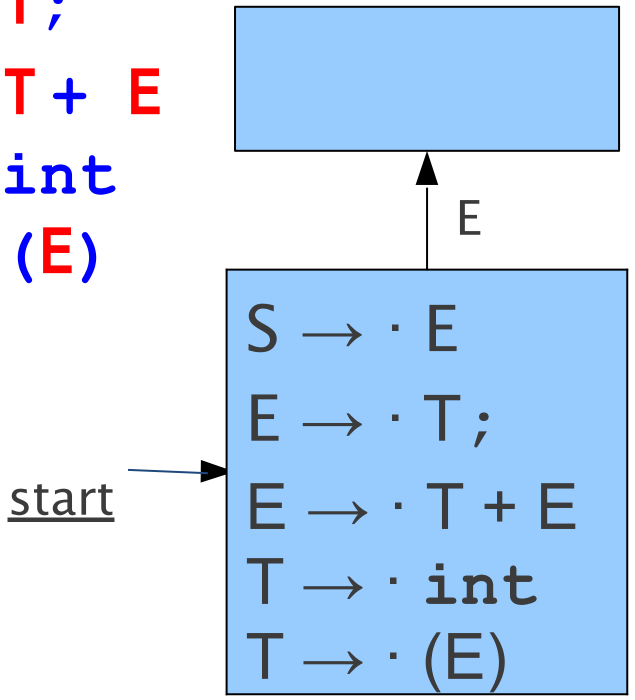


Constructing the Automaton

- Begin in a state containing **S** $\rightarrow \cdot$ **A**, where **S** is the augmented start symbol.
- Compute the **closure** of the state:
  - If **A** $\rightarrow \boldsymbol{\alpha} \cdot$ **B**$\boldsymbol{\omega}$ is in the state, add **B** $\rightarrow \cdot\, \boldsymbol{\gamma}$ to the state for each production **B** $\rightarrow \boldsymbol{\gamma}$.

A Deterministic Automaton

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



A Deterministic Automaton

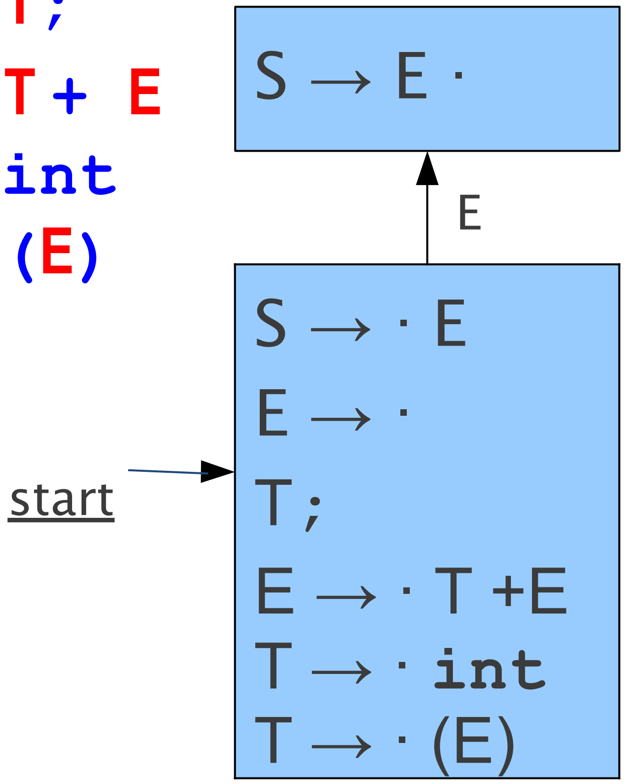
S → E

E → T;

E → T + E

T → int

T → (E)



A Deterministic Automaton

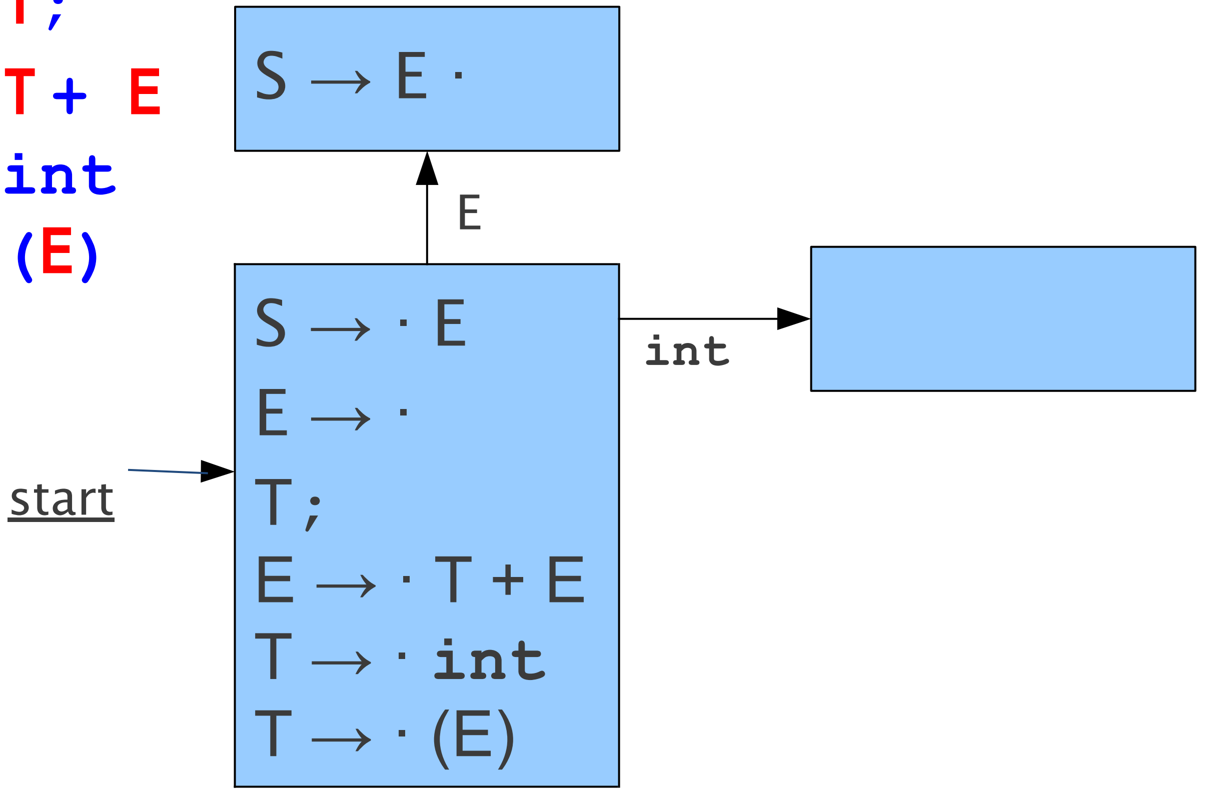
S → E

E → T;

E → T + E

T → **int**

T → (E)



A Deterministic Automaton

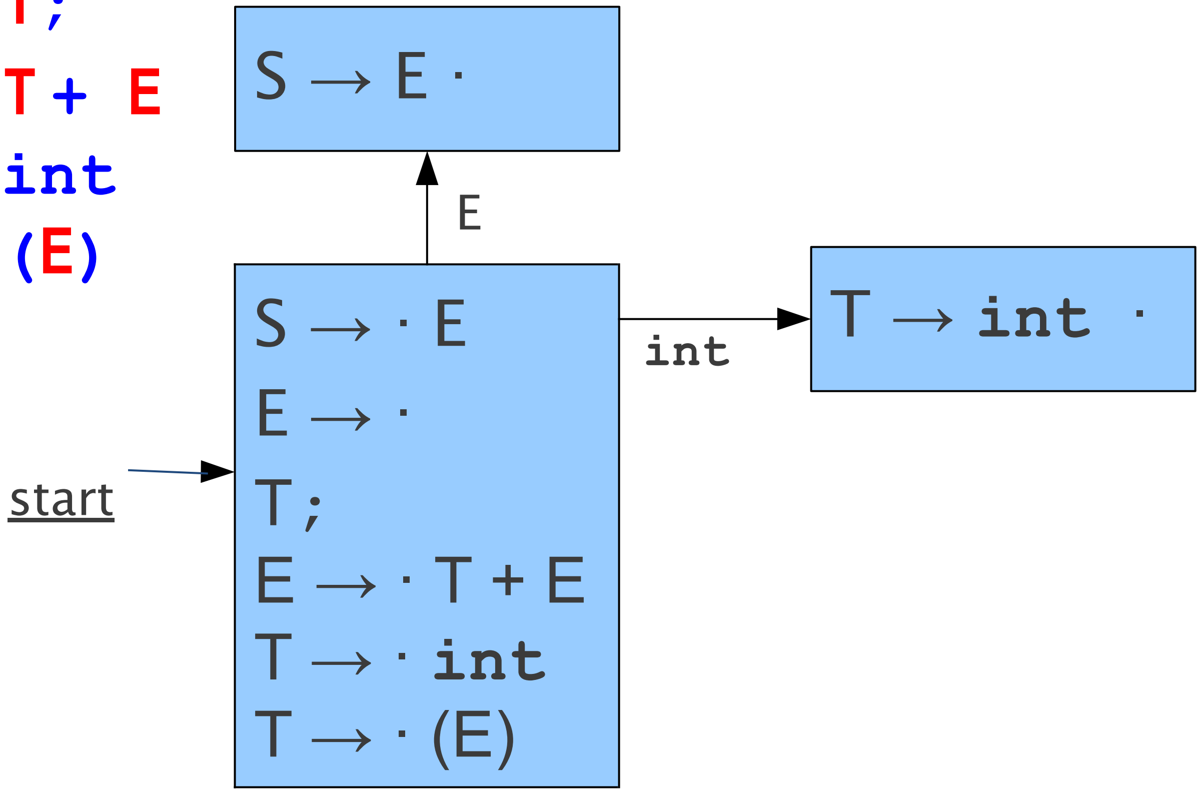
S → **E**

E → **T;**

E → **T + E**

T → `int`

T → **(E)**



A Deterministic Automaton

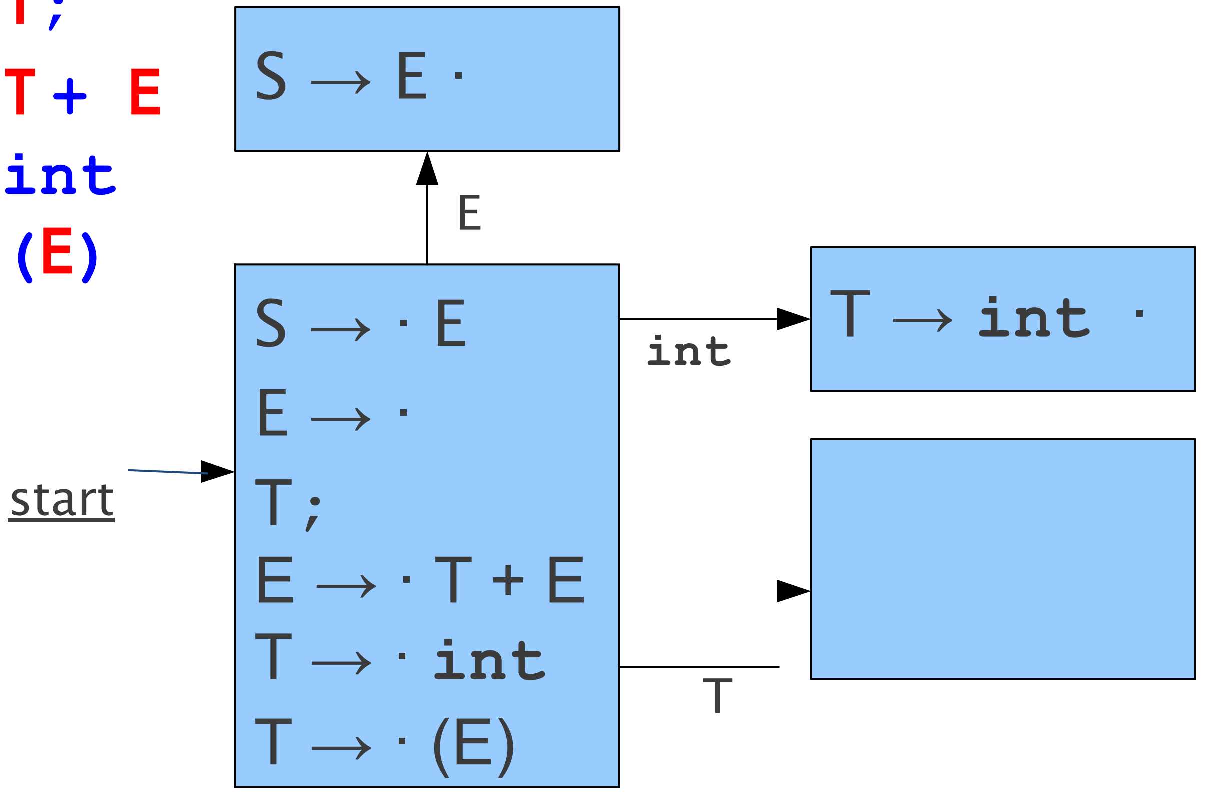
$S \to E$

$E \to T;$

$E \to T + E$

$T \to$ **int**

$T \to (E)$



A Deterministic Automaton

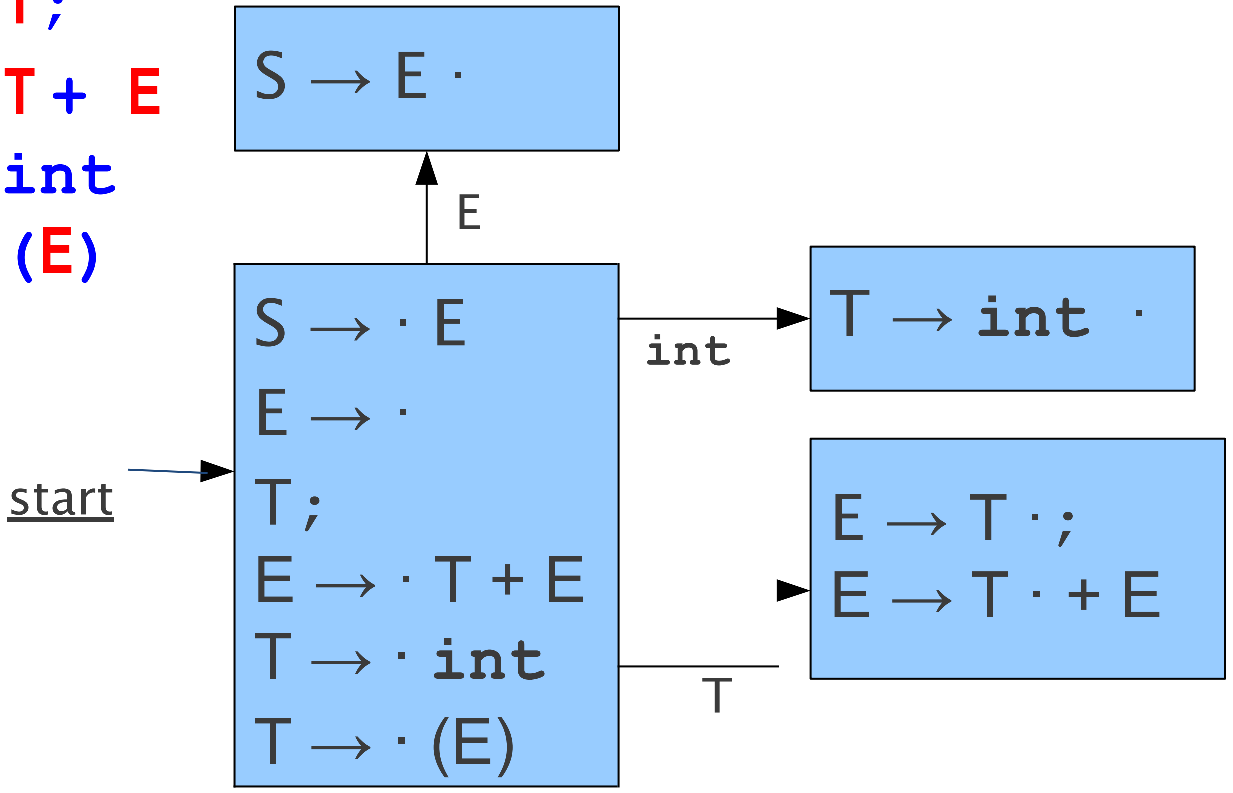
$S \rightarrow E$

$E \rightarrow T;$

$E \rightarrow T + E$

$T \rightarrow$ `int`

$T \rightarrow (E)$



A Deterministic Automaton

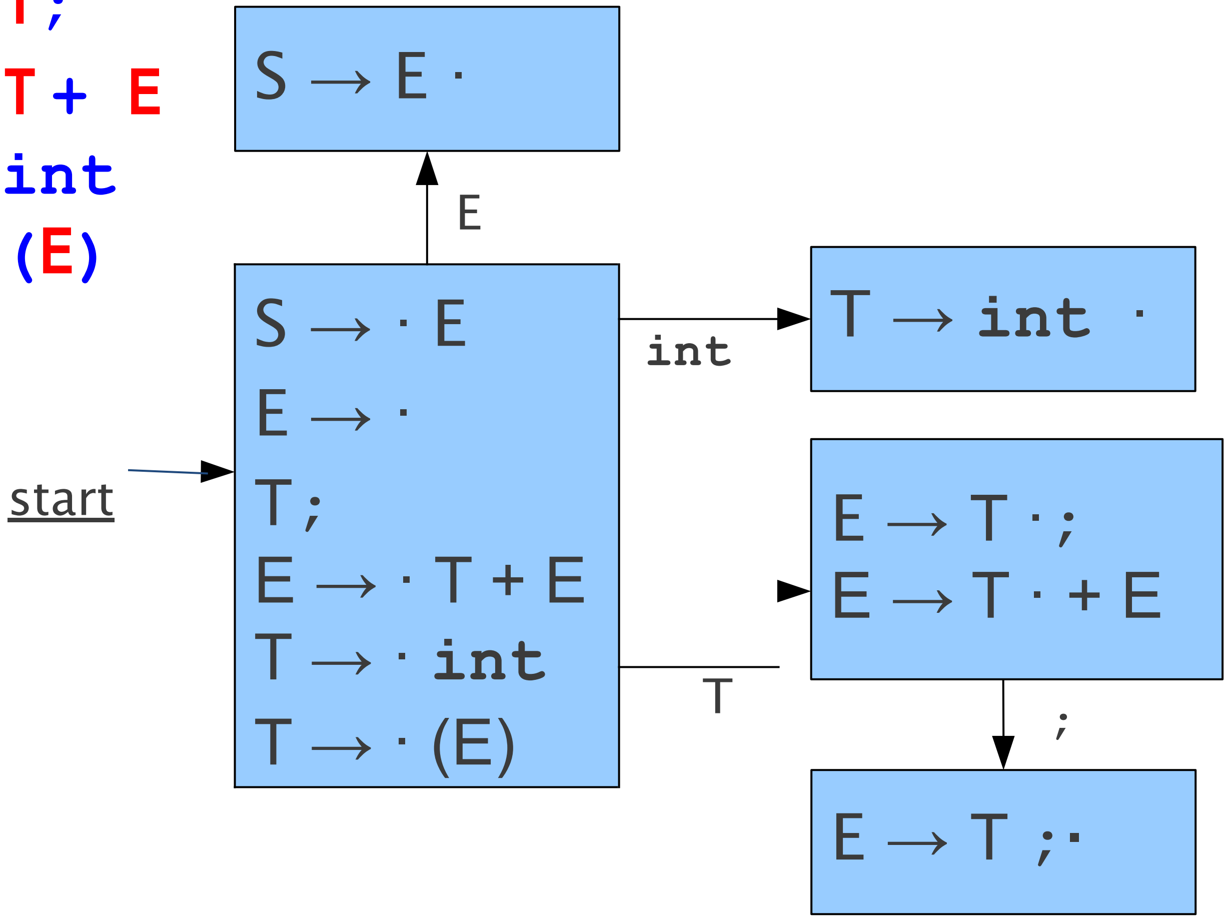
$S \rightarrow E$

$E \rightarrow T$;

$E \rightarrow T + E$

$T \rightarrow$ `int`

$T \rightarrow (E)$



Constructing the Automaton

- Repeat until no new states are added:
  - If a state contains a production $A \to \alpha \cdot x\omega$ for symbol x, add a transition on x from that state to the state containing the closure of $A \to \alpha x \cdot \omega$

A Deterministic Automaton

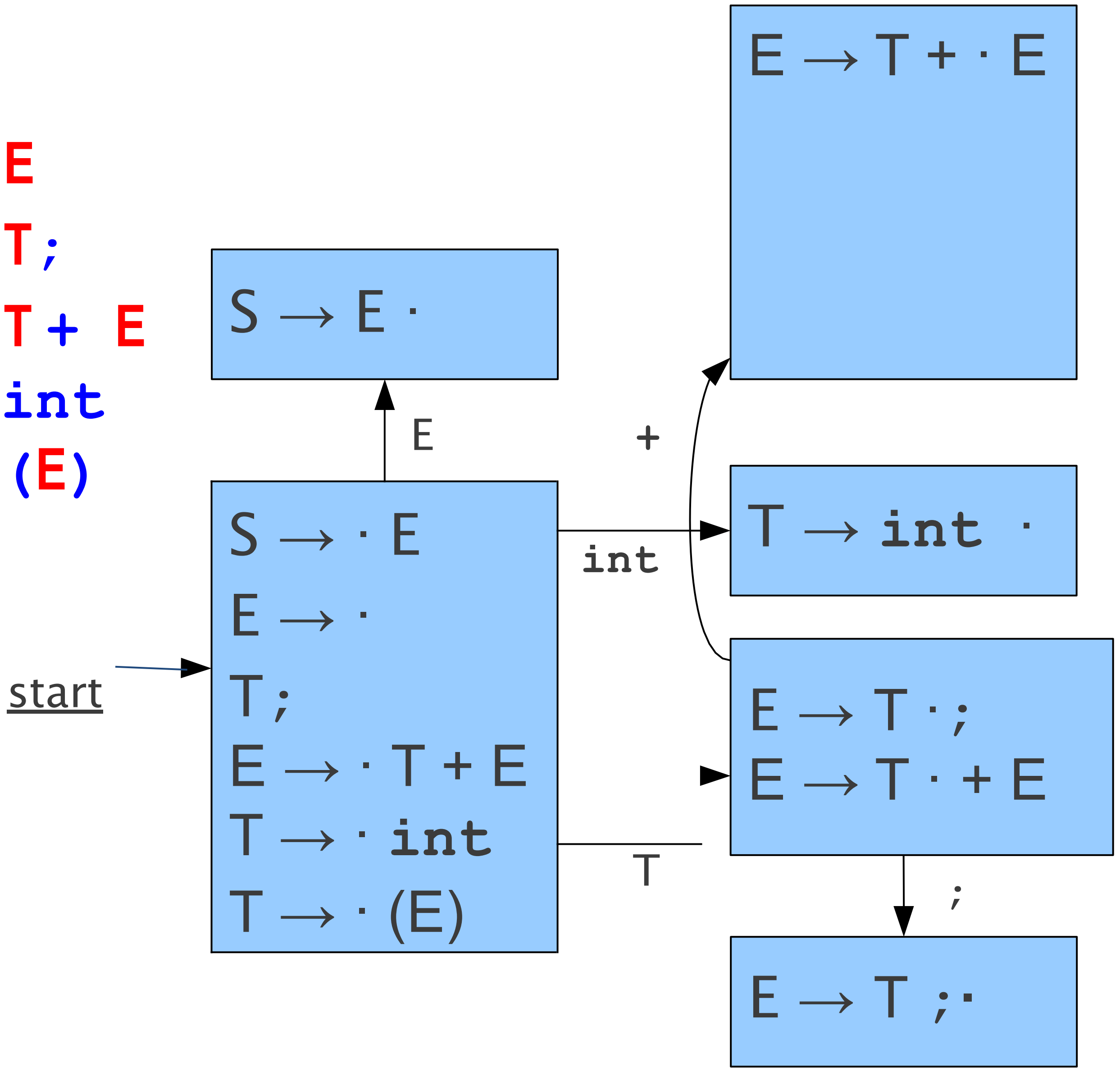
$S \rightarrow E$

$E \rightarrow T$;

$E \rightarrow T + E$

$T \rightarrow$ **int**

$T \rightarrow (E)$



A Deterministic Automaton

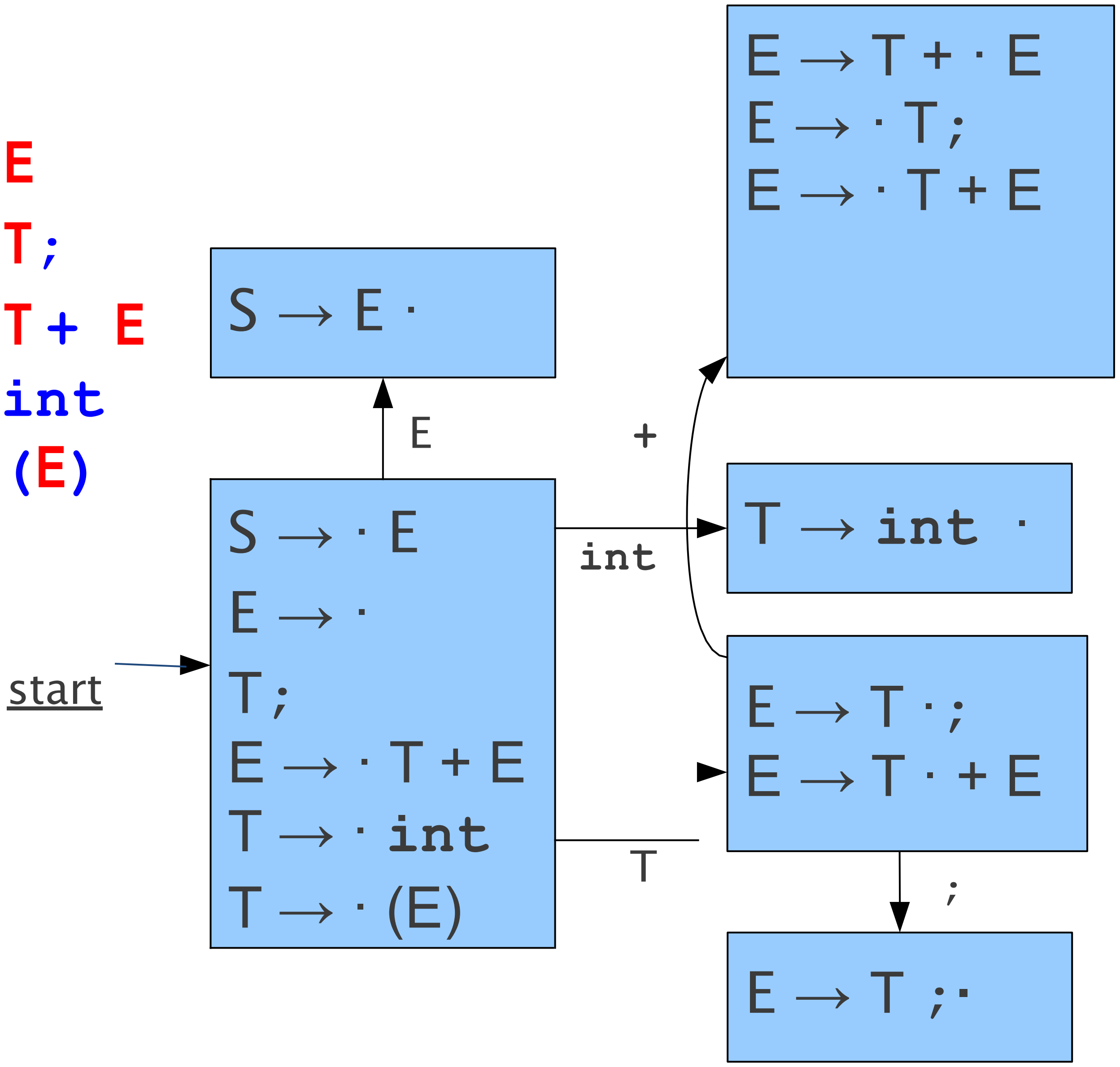
S → E

E → T;

E → T + E

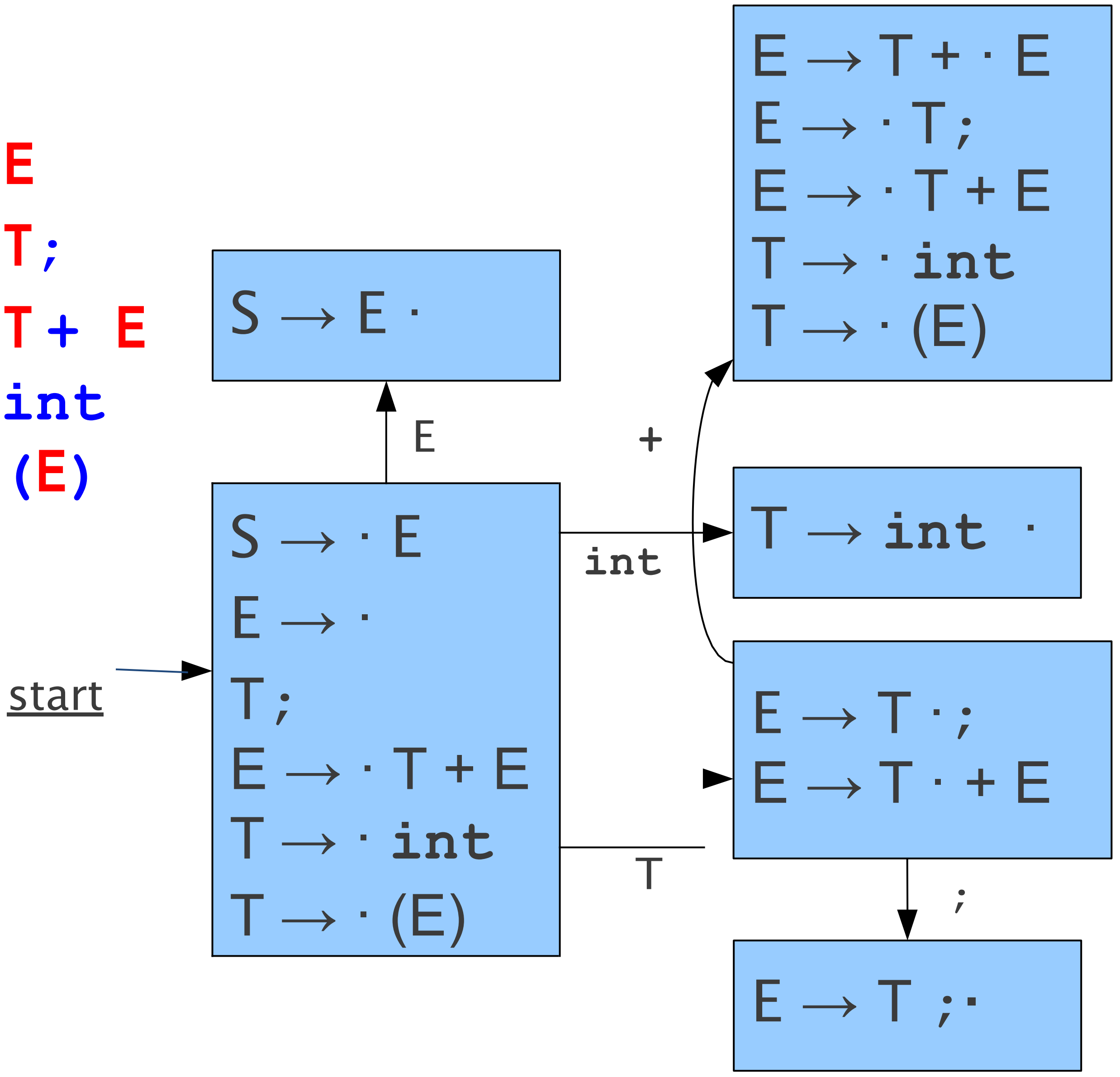
T → int

T → (E)



A Deterministic Automaton

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



A Deterministic Automaton

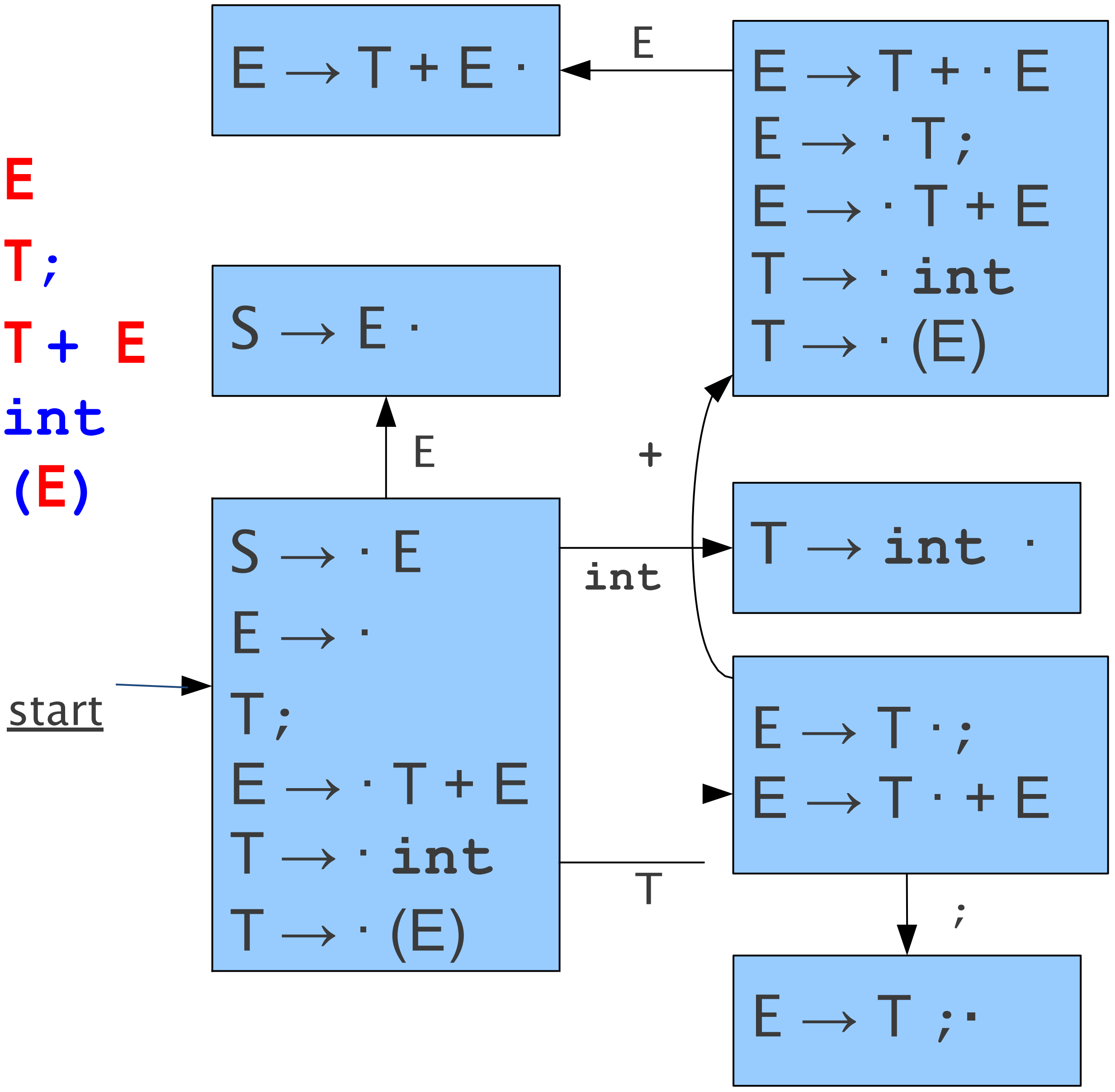
$S \rightarrow E$

$E \rightarrow T;$

$E \rightarrow T + E$

$T \rightarrow$ `int`

$T \rightarrow (E)$



A Deterministic Automaton

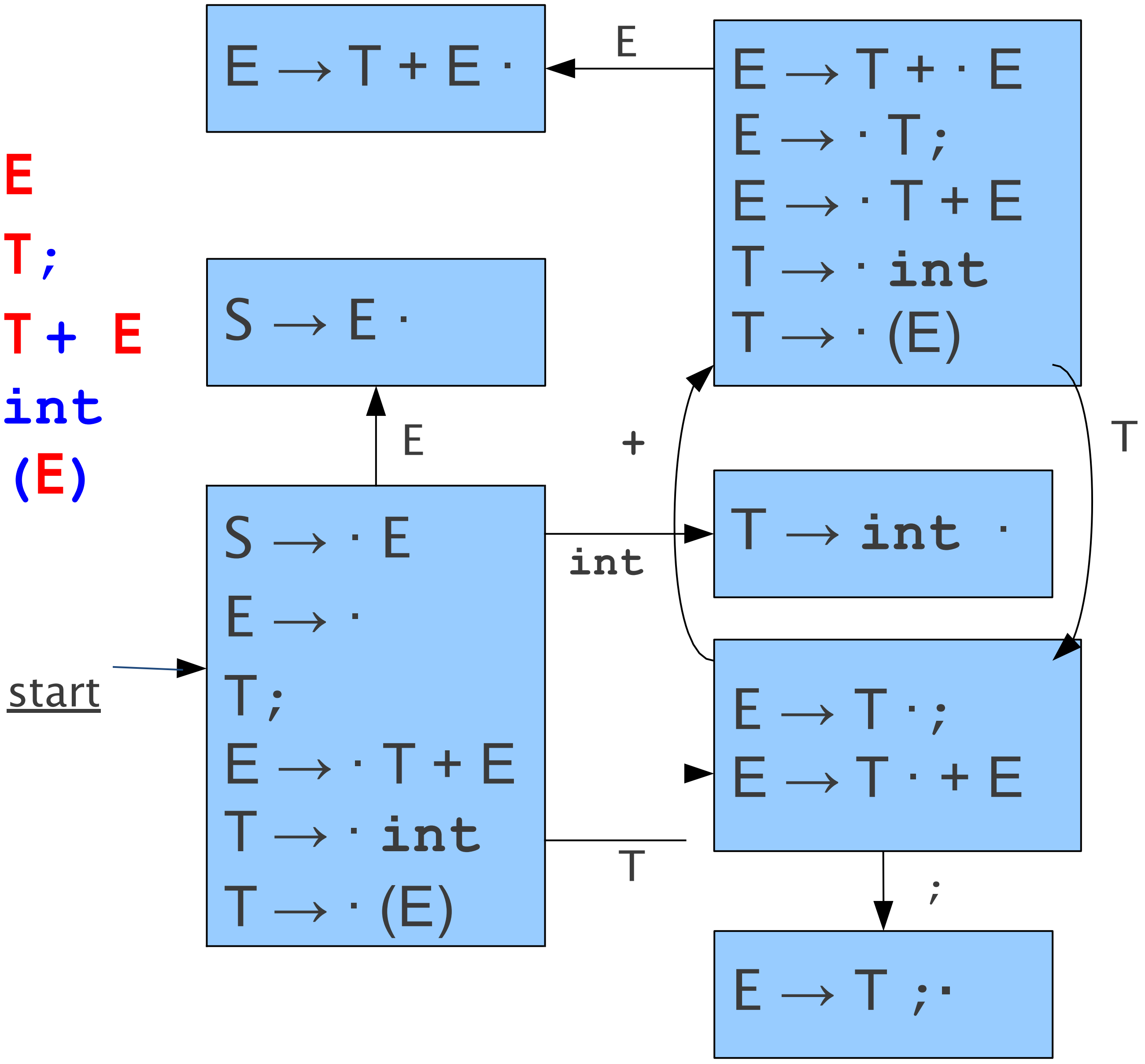
S → E

E → T;

E → T + E

T → **int**

T → (E)



A Deterministic Automaton

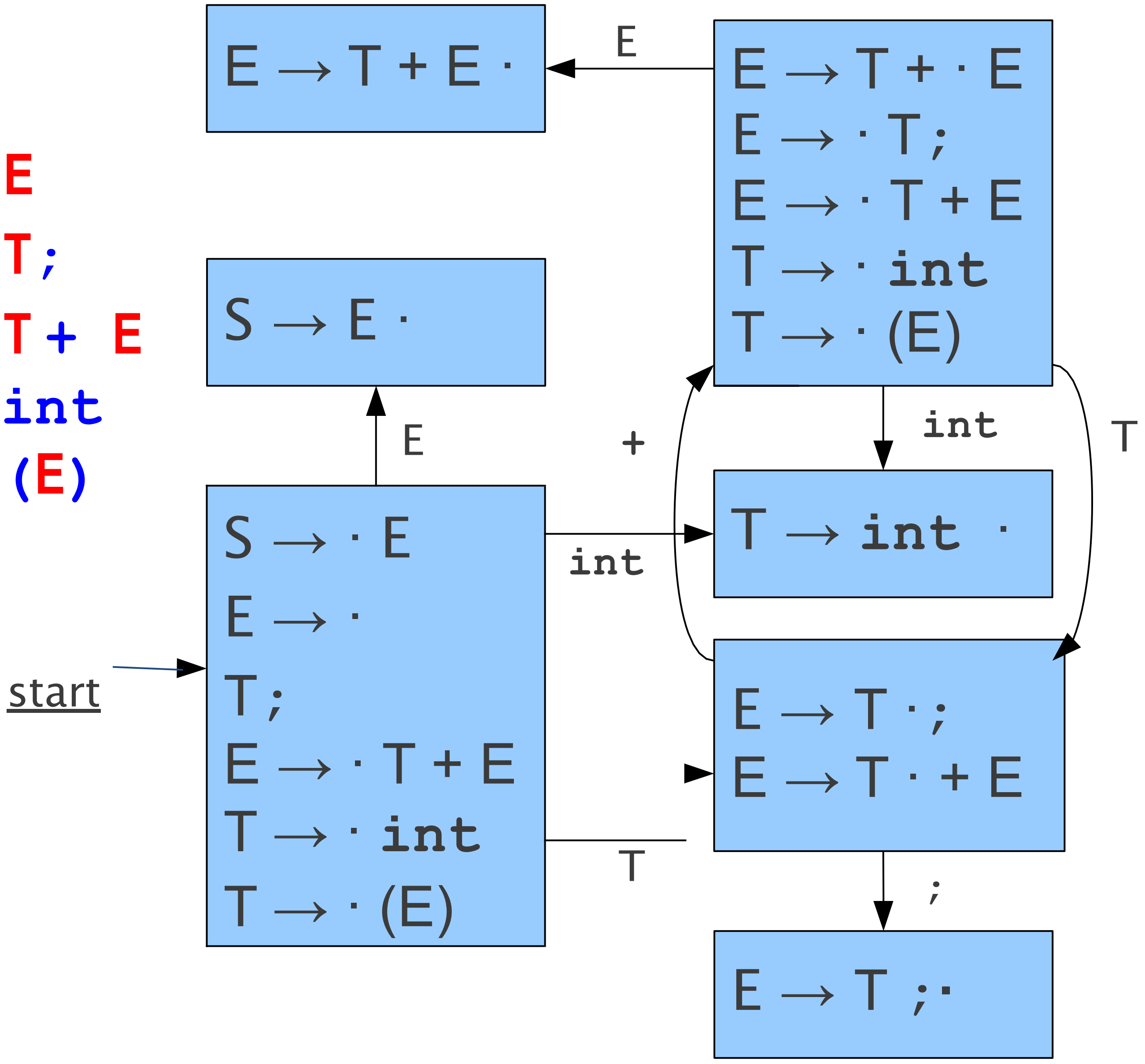
S → E

E → T;

E → T + E

T → int

T → (E)



A Deterministic Automaton

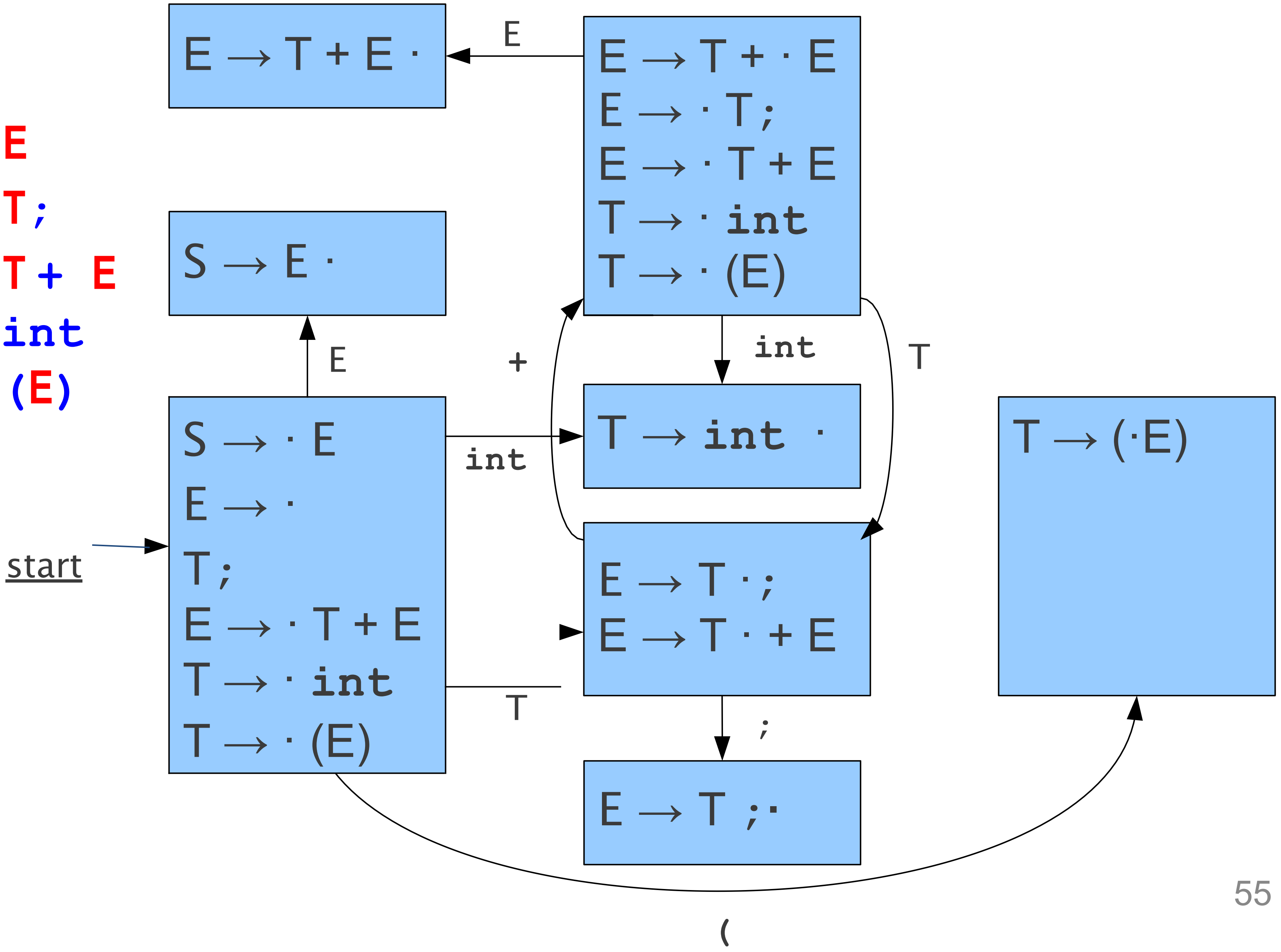
$S \rightarrow E$

$E \rightarrow T$ `;`

$E \rightarrow T$ `+` E

$T \rightarrow$ `int`

$T \rightarrow$ `(E)`



A Deterministic Automaton

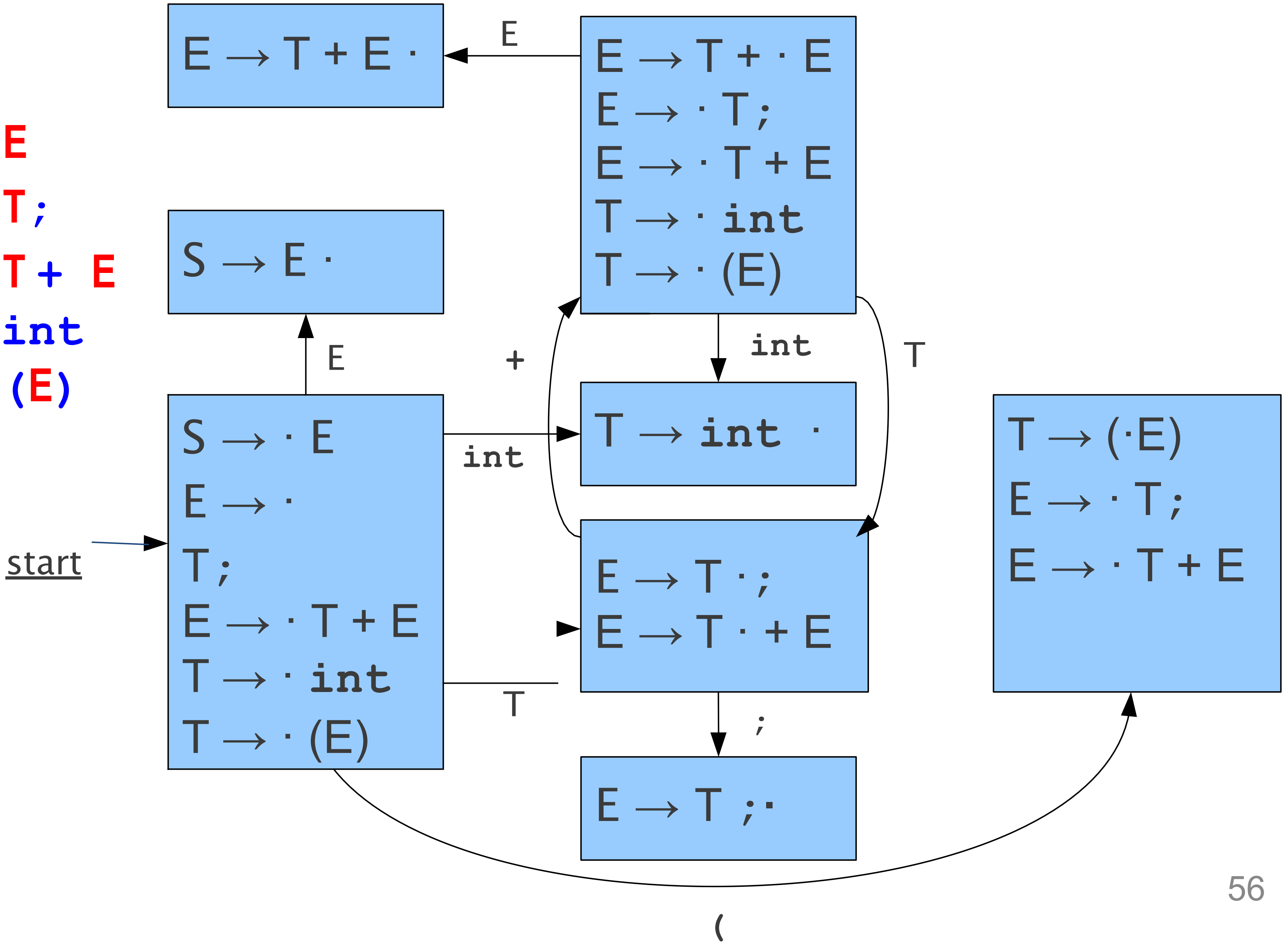
$S \rightarrow E$

$E \rightarrow T$ `;`

$E \rightarrow T$ `+` E

$T \rightarrow$ `int`

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



A Deterministic Automaton

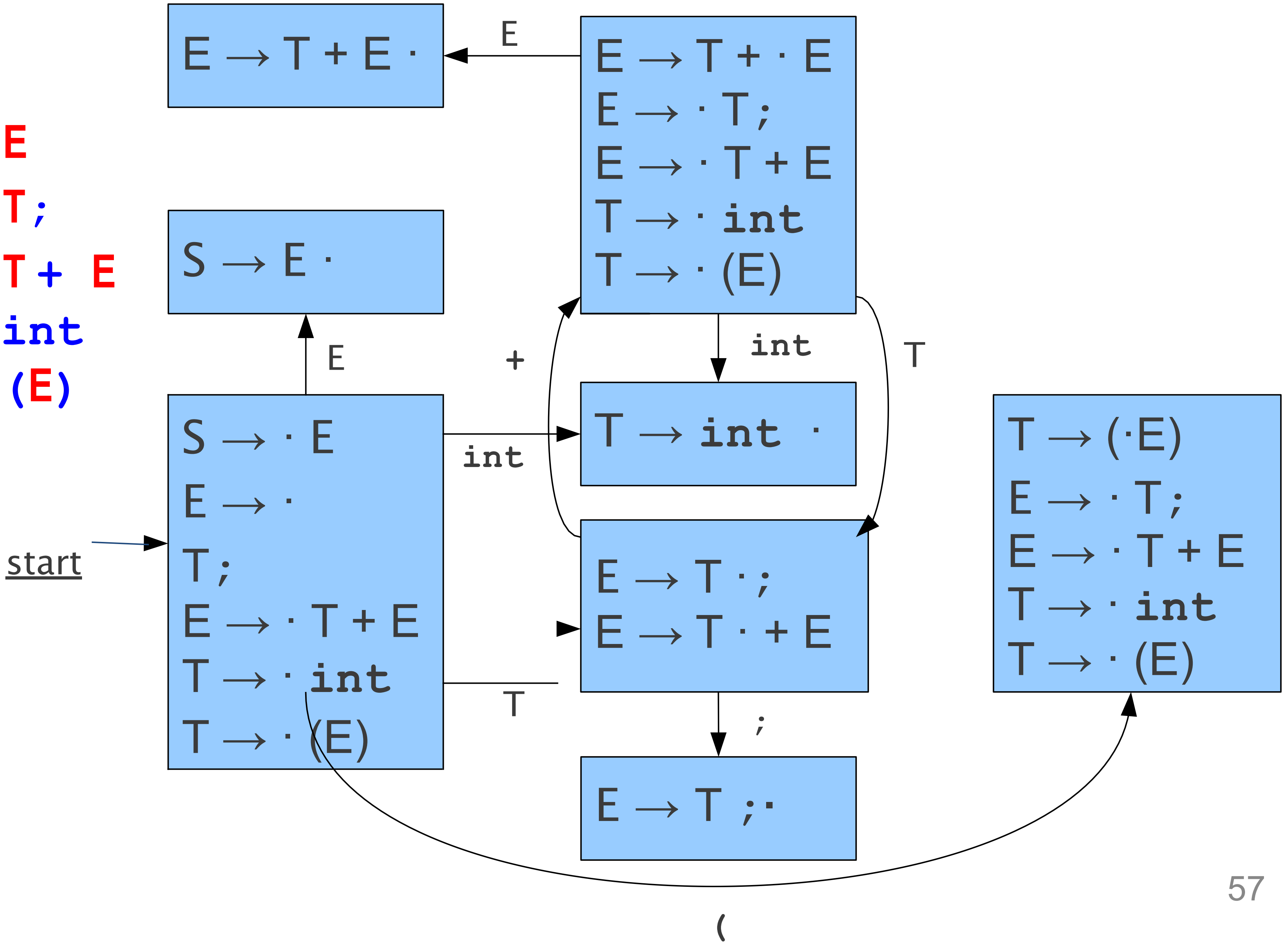
$S \rightarrow E$

$E \rightarrow T;$

$E \rightarrow T + E$

$T \rightarrow$ int

$T \rightarrow (E)$



A Deterministic Automaton

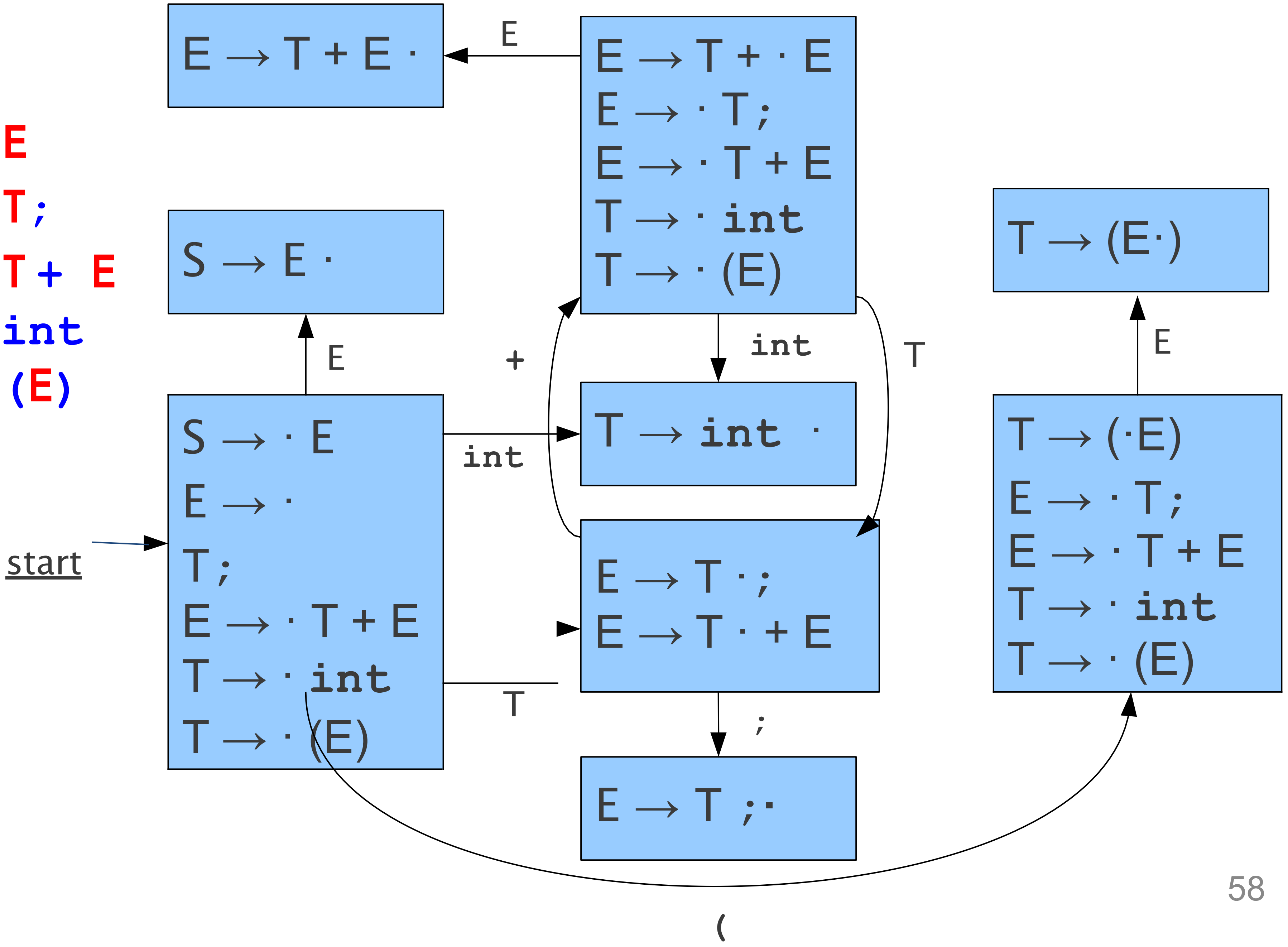
$S \rightarrow E$

$E \rightarrow T$ `;`

$E \rightarrow T + E$

$T \rightarrow$ `int`

$T \rightarrow$ `(E)`



A Deterministic Automaton

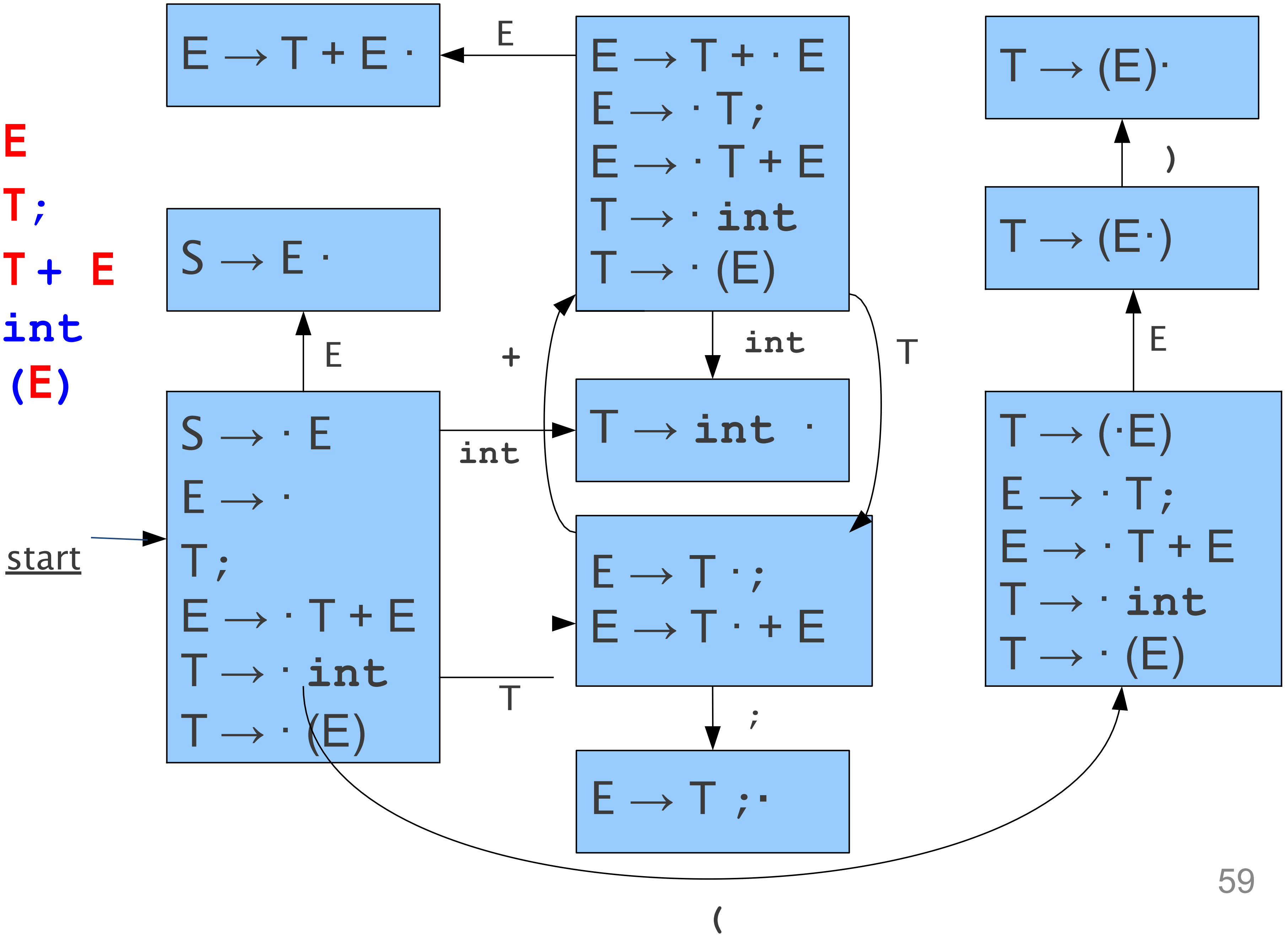
S → E

E → T;

E → T + E

T → int

T → (E)



A Deterministic Automaton

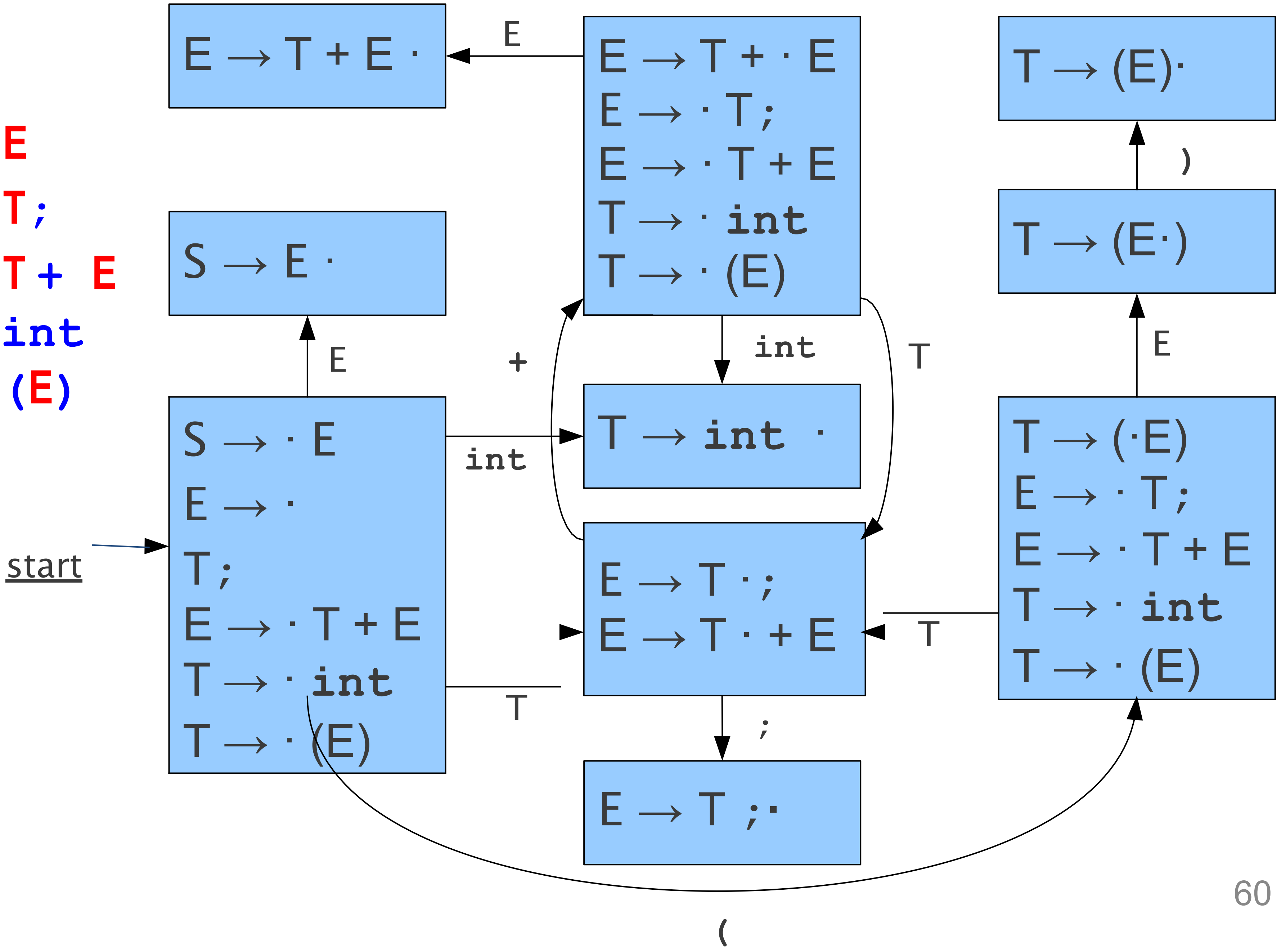
S → E

E → T;

E → T + E

T → int

T → (E)



A Deterministic Automaton

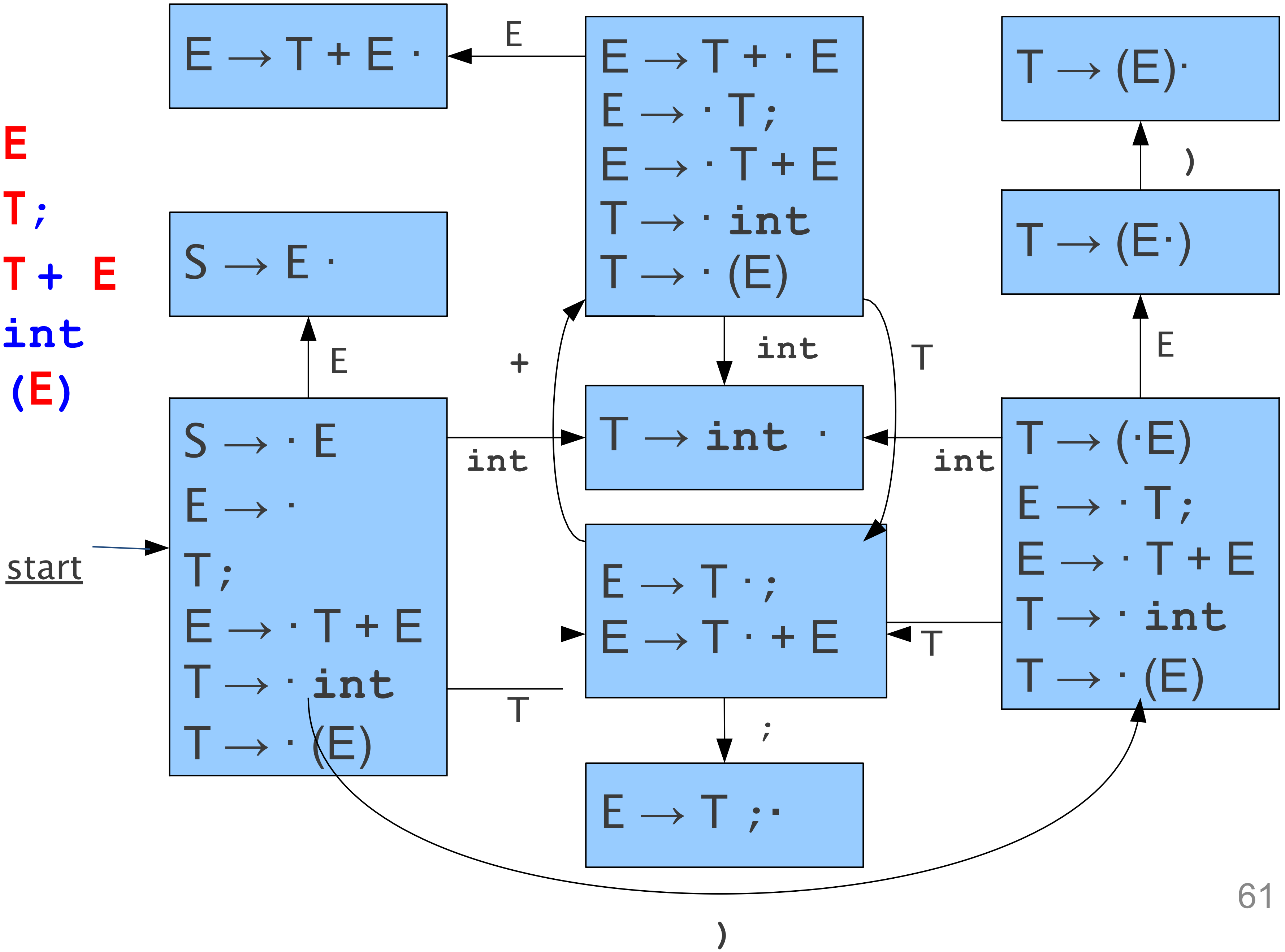
$S \rightarrow E$

$E \rightarrow T$;

$E \rightarrow T + E$

$T \rightarrow$ `int`

$T \rightarrow (E)$



A Deterministic Automaton

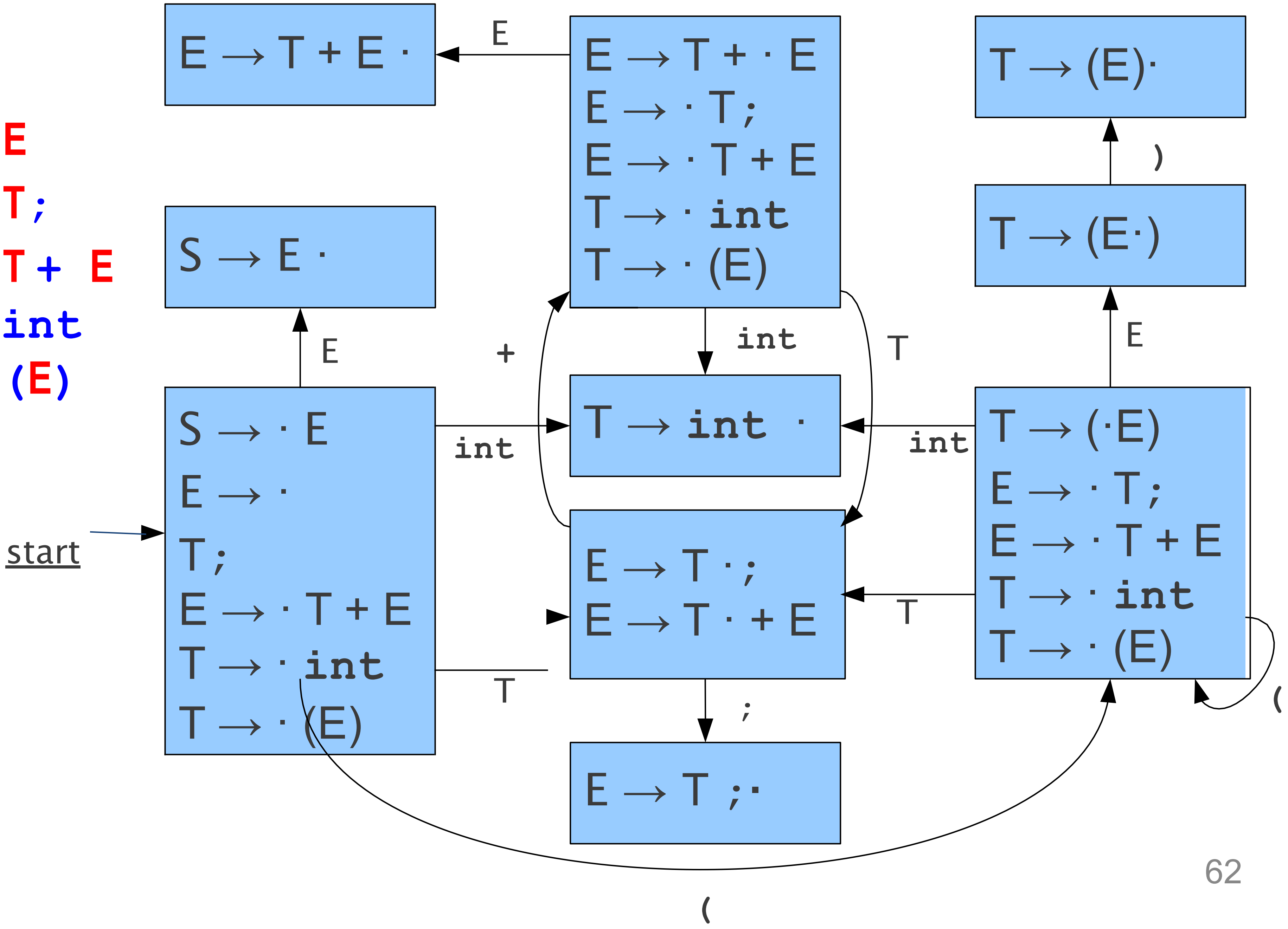
$S \rightarrow E$

$E \rightarrow T;$

$E \rightarrow T + E$

$T \rightarrow$ `int`

$T \rightarrow (E)$



A Deterministic Automaton

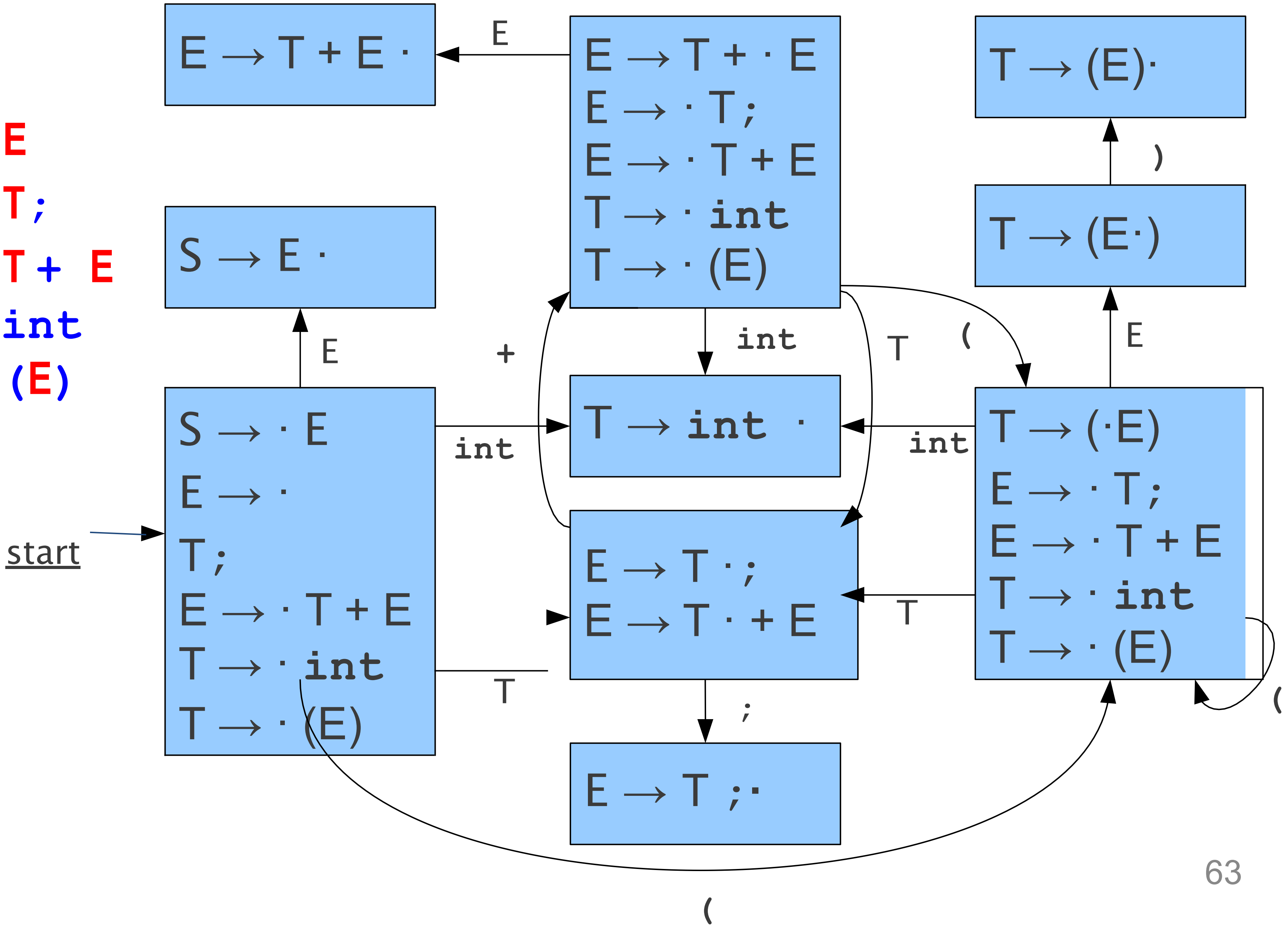
$S \rightarrow E$

$E \rightarrow T;$

$E \rightarrow T + E$

$T \rightarrow$ **int**

$T \rightarrow (E)$



A Deterministic Automaton

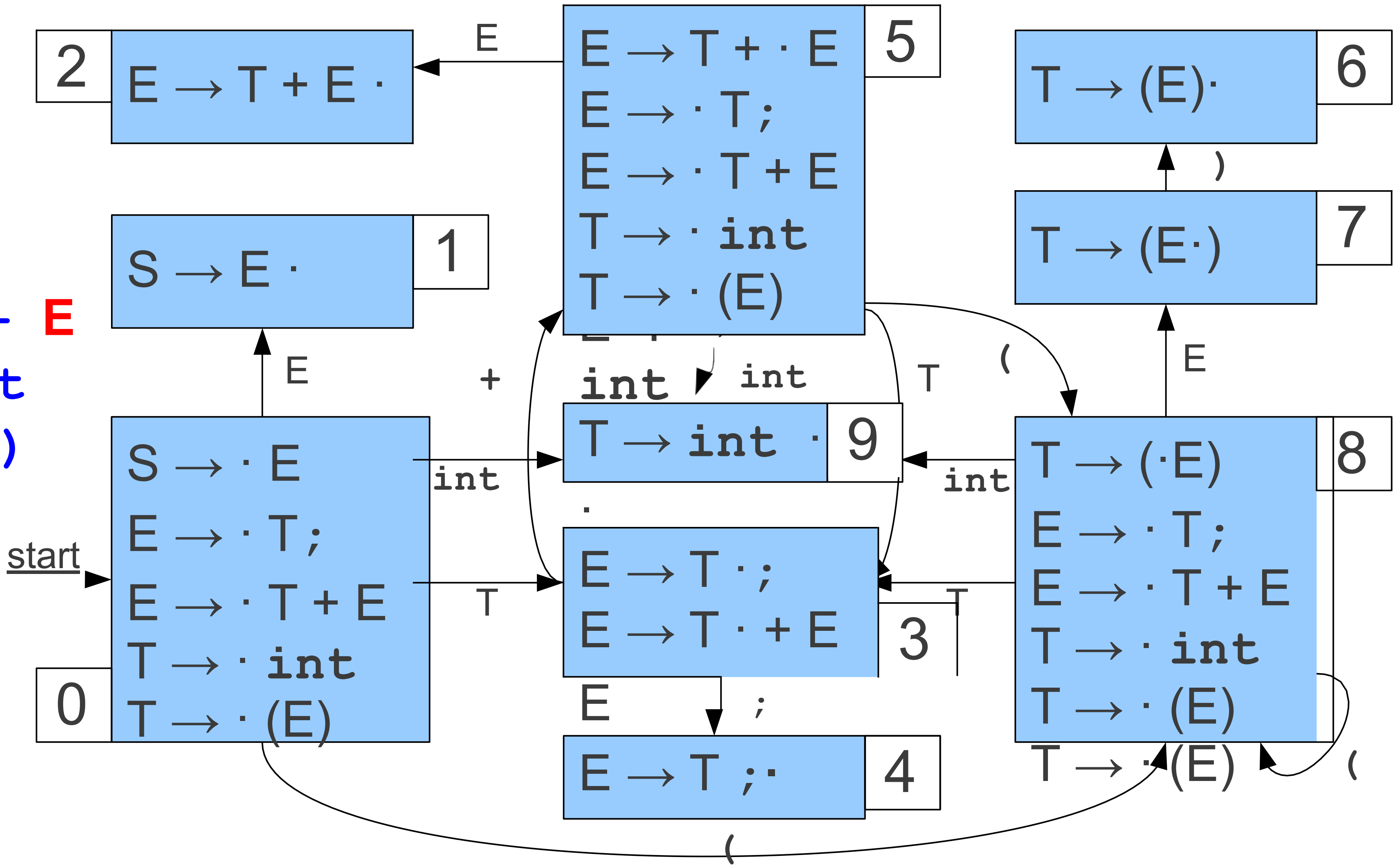
S → E

E → T;

E → T + E

T → int

T → (E)



A Deterministic Automaton

	`int`	`+`	`;`	`(`	`)`	E	T
0	S9			S8		S1	S3
1							
2							
3		S5	S4				
4							
5	S9			S8		S2	S3
6							
7					S6		
8	S9			S8		S7	S3
9							

A Deterministic Automaton

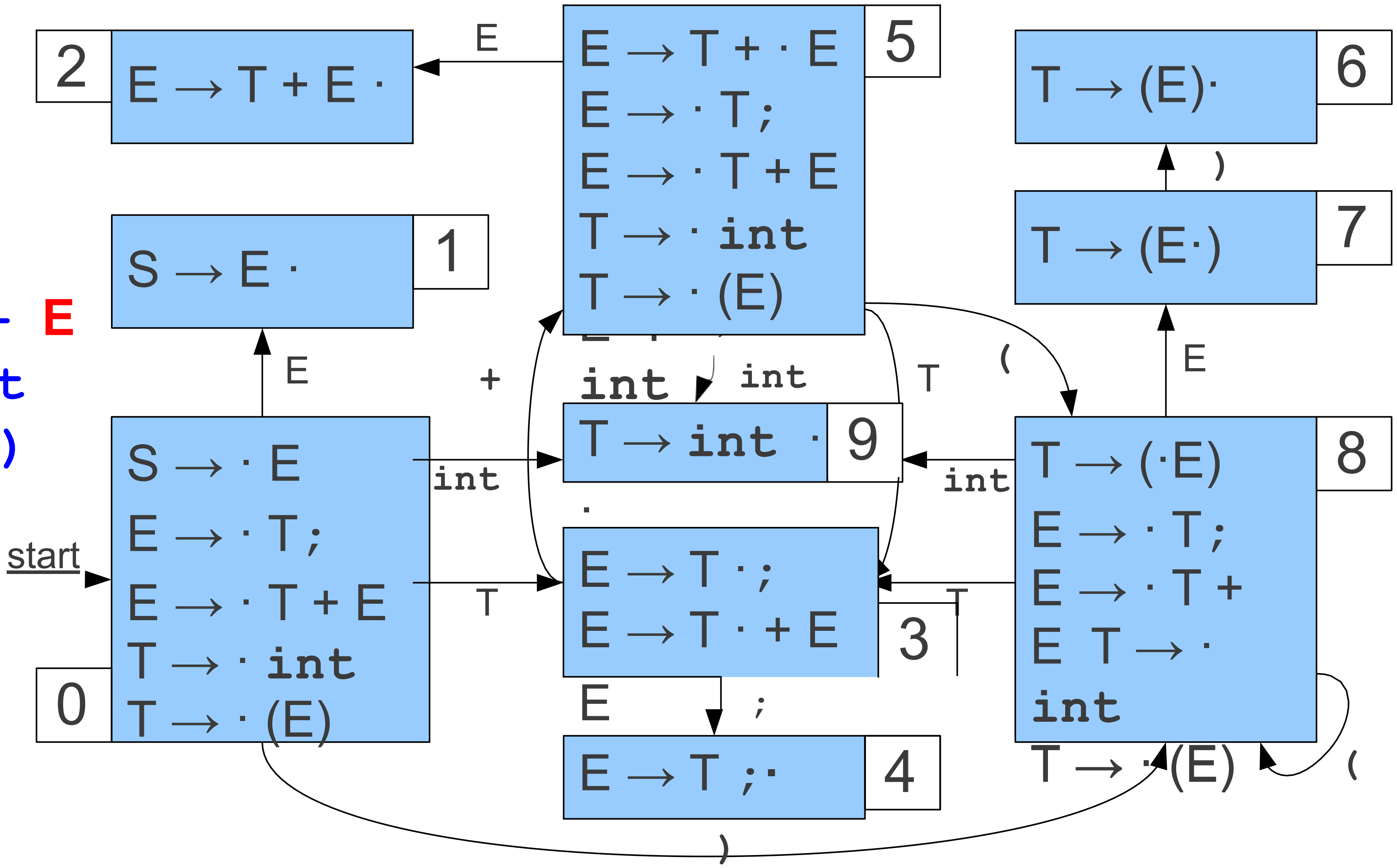
S → E

E → T;

E → T + E

T → **int**

T → (E)



LR(0) Tables

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

	Action					Goto	
	int	+	;	(	)	E	T
0	S9			S8		S1	S3
1	r1	r1	r1	r1	r1		
2	r3	r3	r3	r3	r3		
3		S5	S4				
4	r2	r2	r2	r2	r2		
5	S9			S8		S2	S3
6	r5	r5	r5	r5	r5		
7					s6		
8	S9			S8		S7	S3
9	r4	r4	r4	r4	r4		

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

$$A \rightarrow \omega \cdot$$

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

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

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