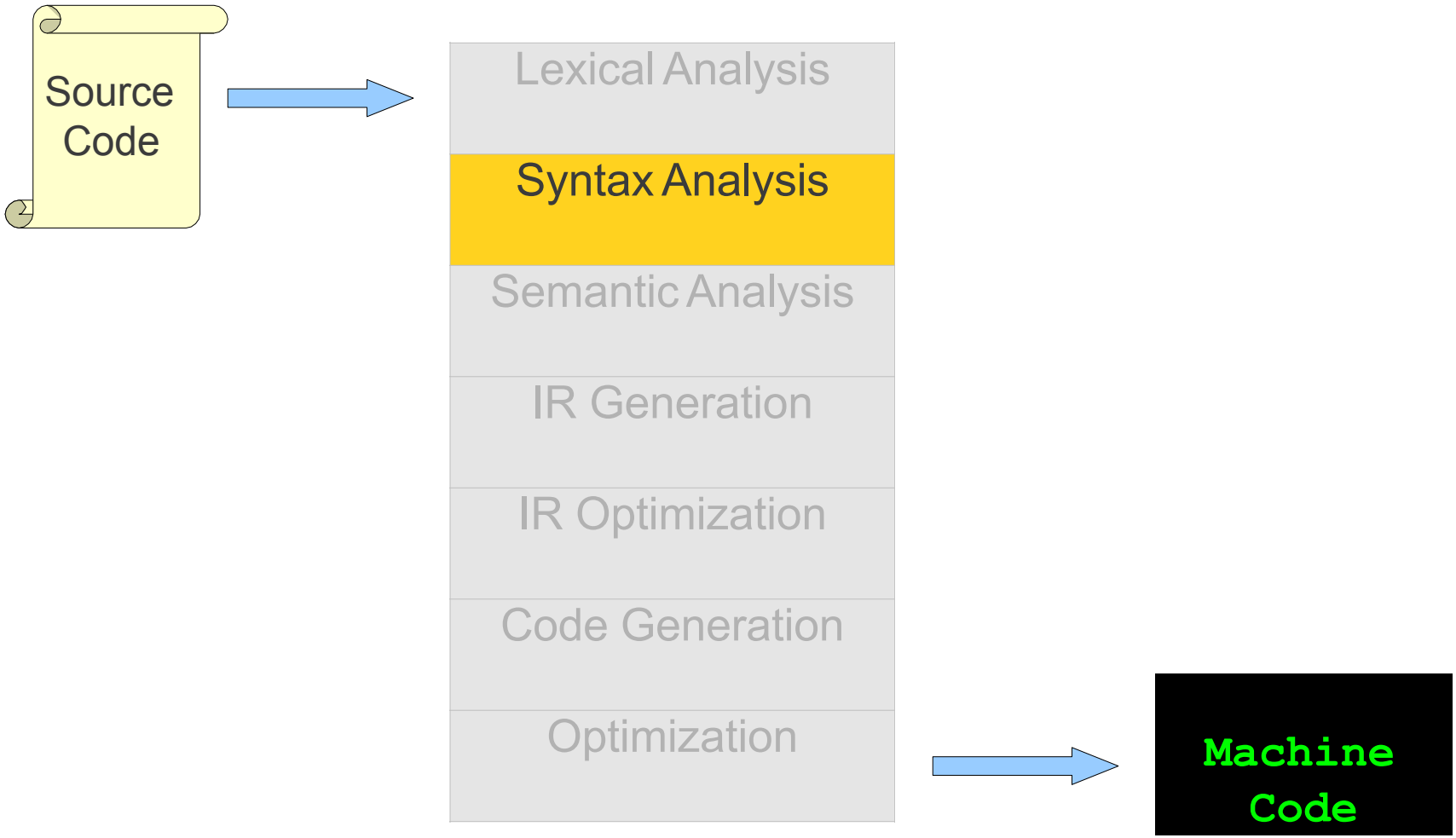
بسم الله الرحمن الرحيم

Parsing:
Top-Down Parsing, Recursive Descent & Predictive Parser & LL(1)

Next Time


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

Bottom-Up Parsing

Bottom-Up Parsing

- Idea: Apply productions **in reverse** to convert the user's program to the start symbol.
- Bottom-up parsing is more general than top-down, but with the same efficiency.
- **Directional**: Scan the input from left-to-right.
- **Predictive**: Guess which production should be inverted.

Bottom-Up Parsing?

- It is preferred method.
- No need to left-factored grammars.
- In Nutshell: ***Reduce*** string to S.

A Second View of a Bottom-Up Parse

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

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

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
```

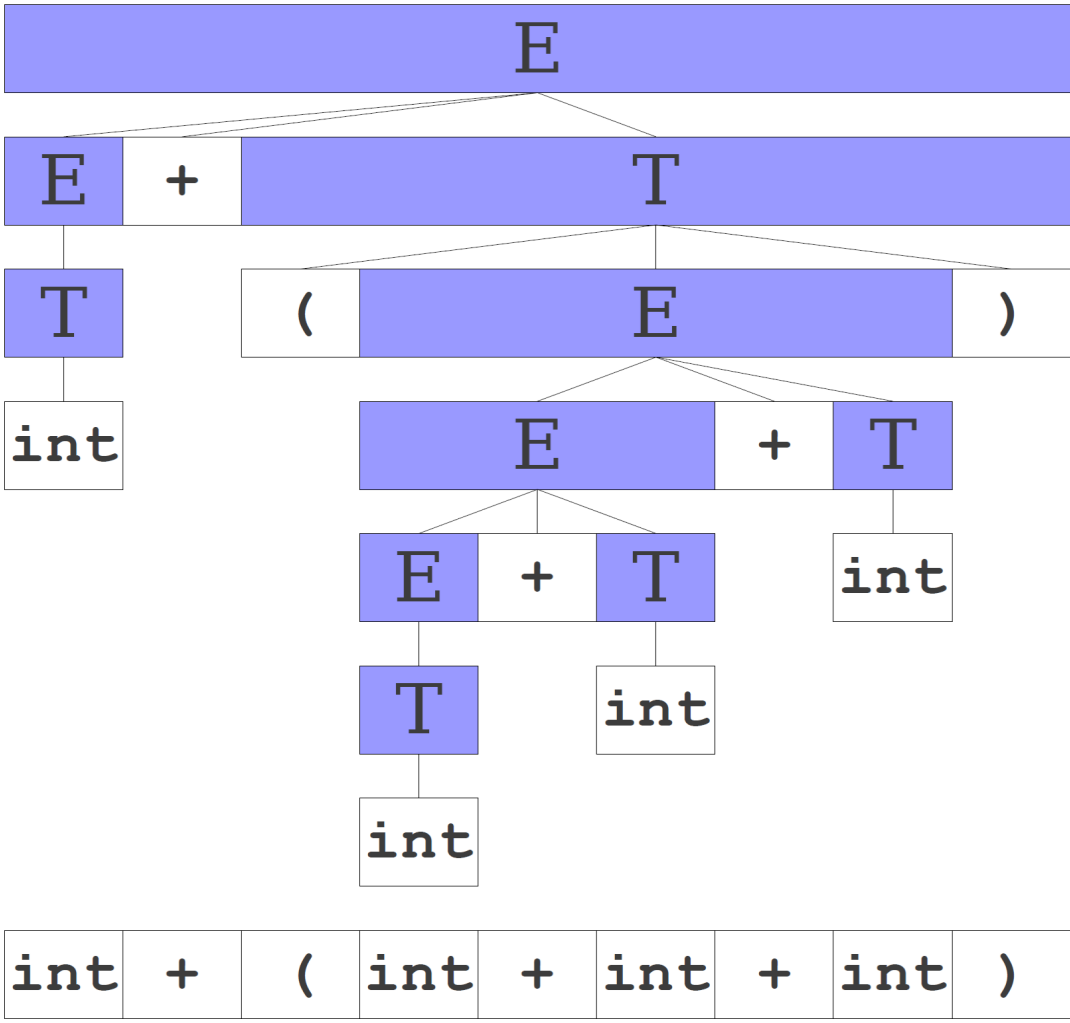
Each step in this bottom-up parse is called a **reduction**.

We **reduce** a substring of the sentential form back to a nonterminal.

A Third View of a Bottom-Up Parse

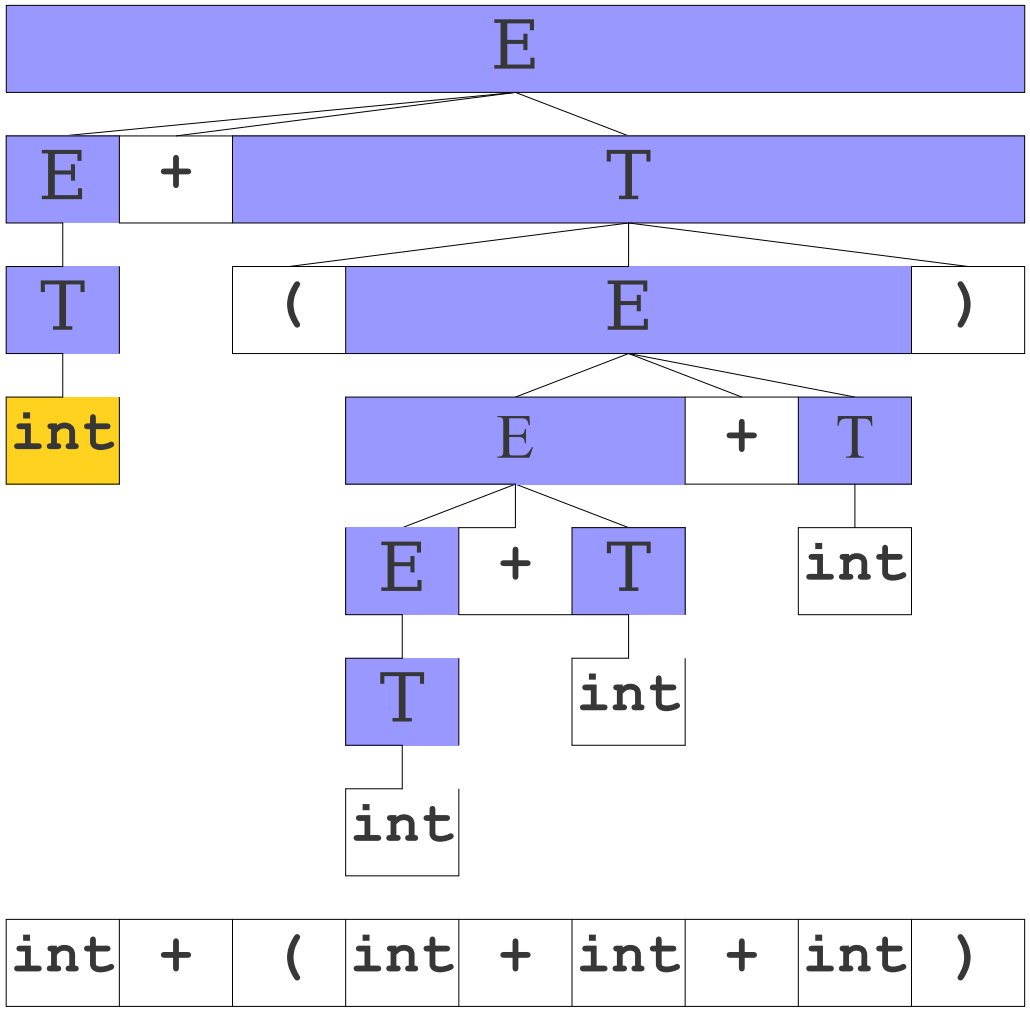
```
int    + (int  + int + int)
⇒ T  + (int   + int   + int)
⇒ E  + (int   + int   + int)
⇒ E  +  (T  + int   + int)
⇒ E  +  (E  + int   + int)
⇒ E  +  (E  + T  + int)
⇒ E  +  (E  +  int)
⇒ E  +  (E  + T)
⇒ E  +  (E
⇒ E  +  T

⇒ E
```



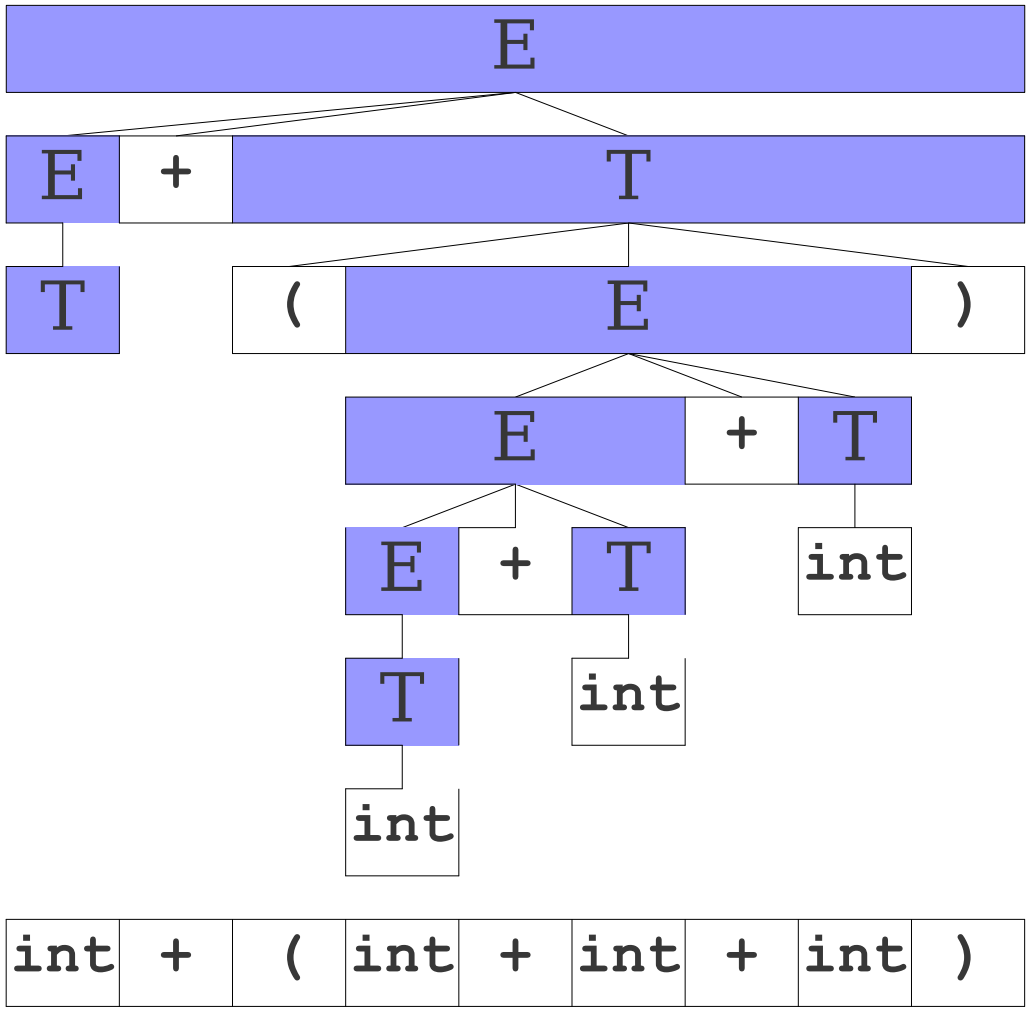
A Third View of a Bottom-Up Parse

```
  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
```



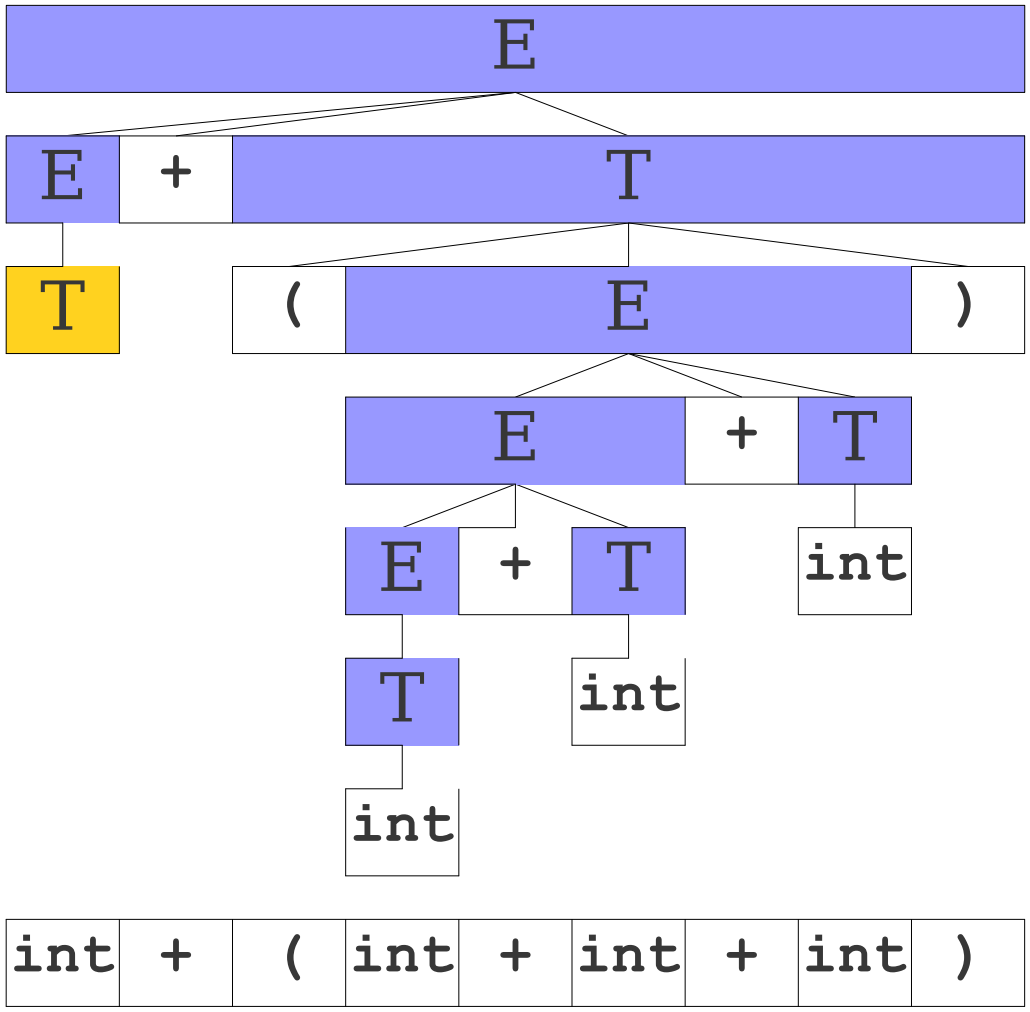
A Third View of a Bottom-Up Parse

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



A Third View of a Bottom-Up Parse

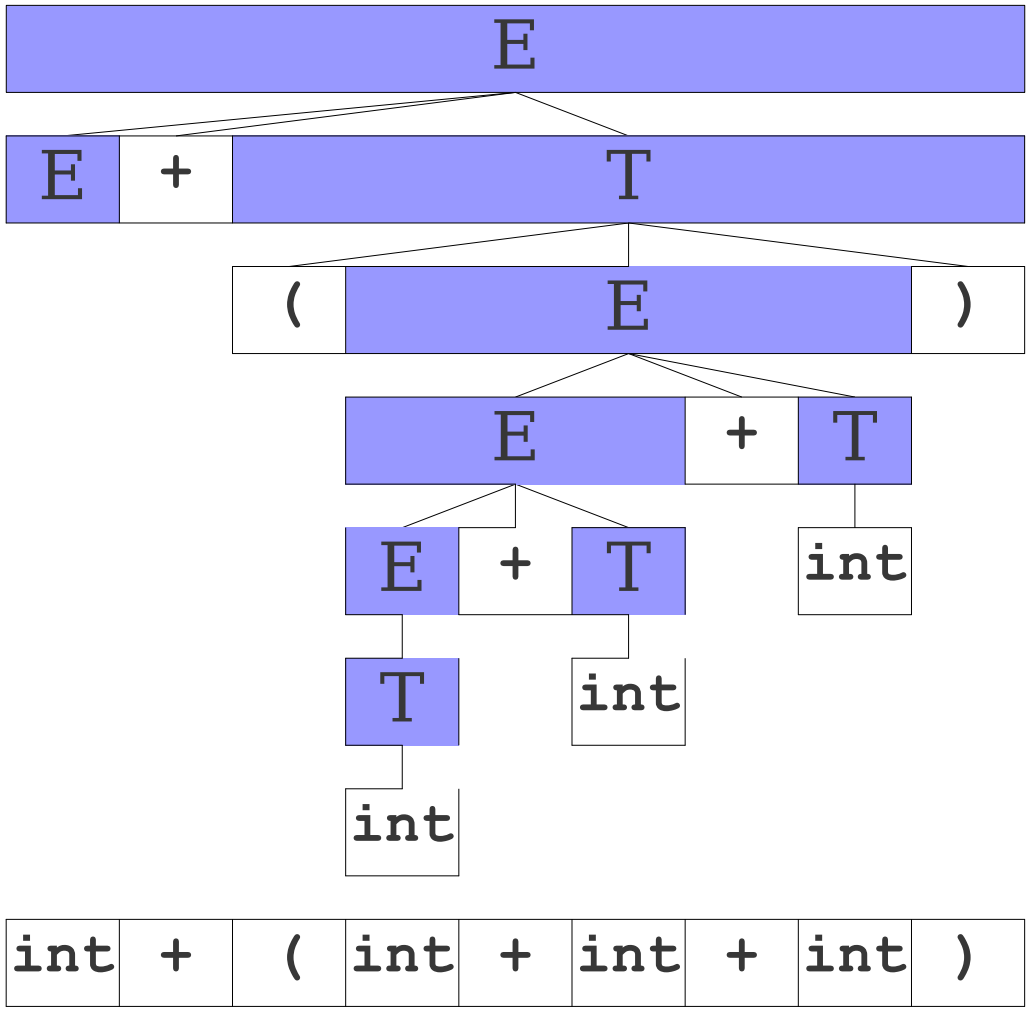
$\Rightarrow$ **T** + (int + int + int)
$\Rightarrow$ **E** + (int + int + int)
$\Rightarrow$ **E** + (**T** + int + int)
$\Rightarrow$ **E** + (**E** + int + int)
$\Rightarrow$ **E** + (**E** + **T** + int)
$\Rightarrow$ **E** + (**E** + int)
$\Rightarrow$ **E** + (**E** + **T**)
$\Rightarrow$ **E**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



A Third View of a Bottom-Up Parse

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

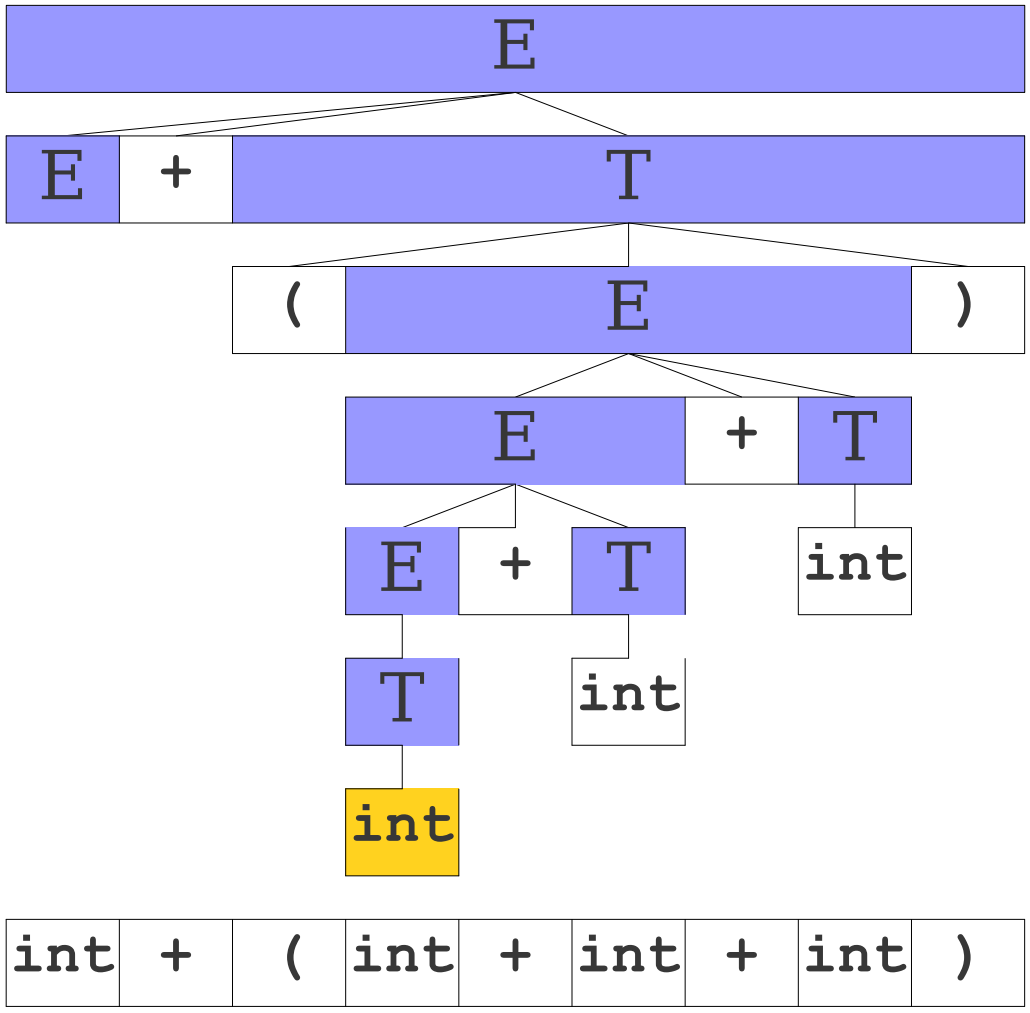
⇒ E



A Third View of a Bottom-Up Parse

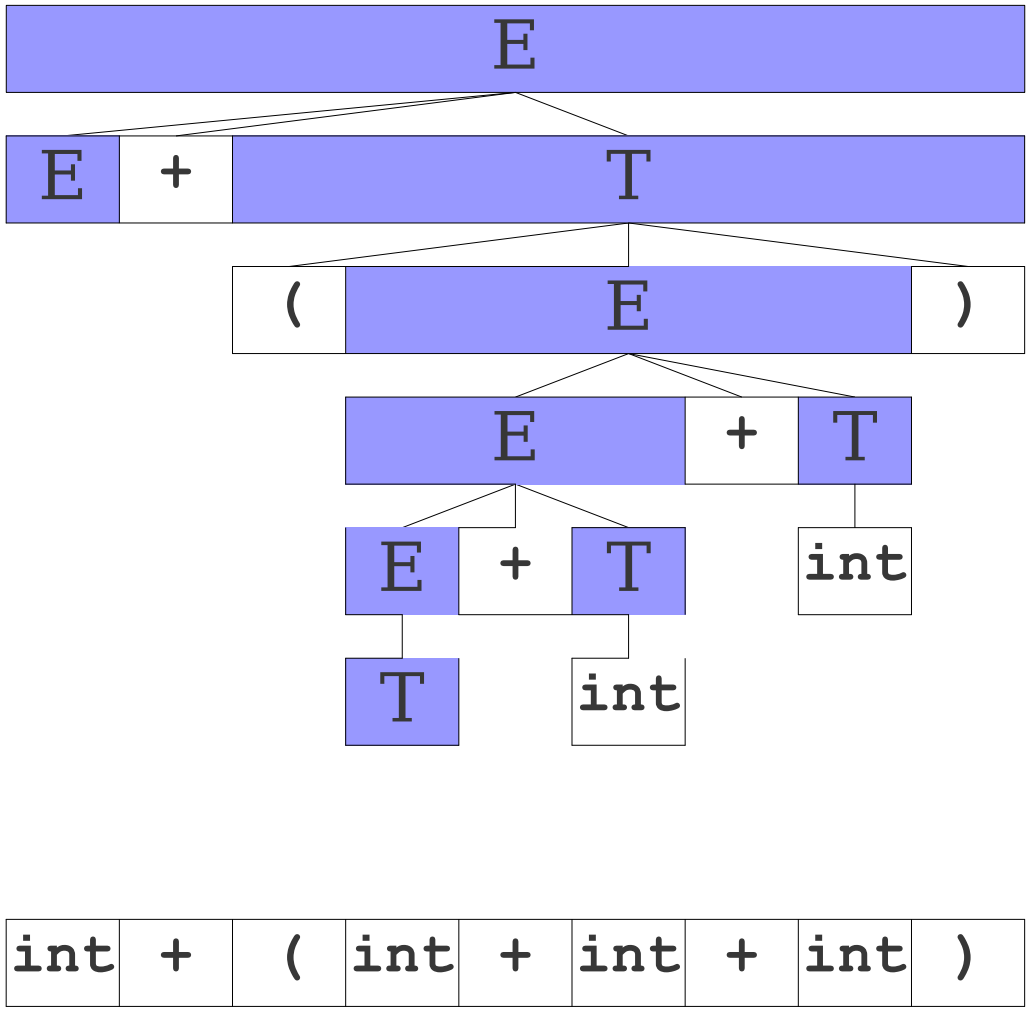
$\Rightarrow$ E + (int + int + int)
$\Rightarrow$ E + (T + int + int)
$\Rightarrow$ E + (E + int + int)
$\Rightarrow$ E + (E + T + int)
$\Rightarrow$ E + (E + int)
$\Rightarrow$ E + (E + T)
$\Rightarrow$ E + (E)
$\Rightarrow$ E + T

$\Rightarrow$ E



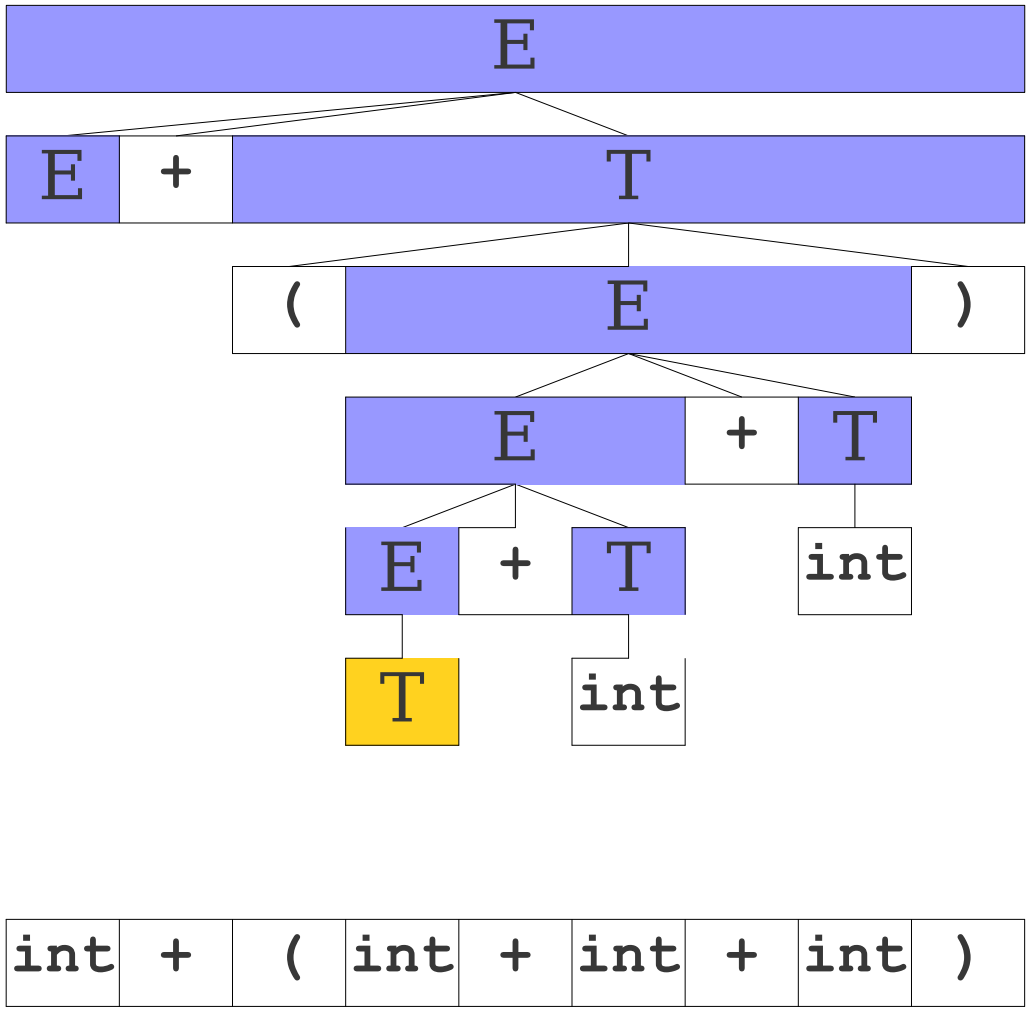
A Third View of a Bottom-Up Parse

$\Rightarrow$ 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



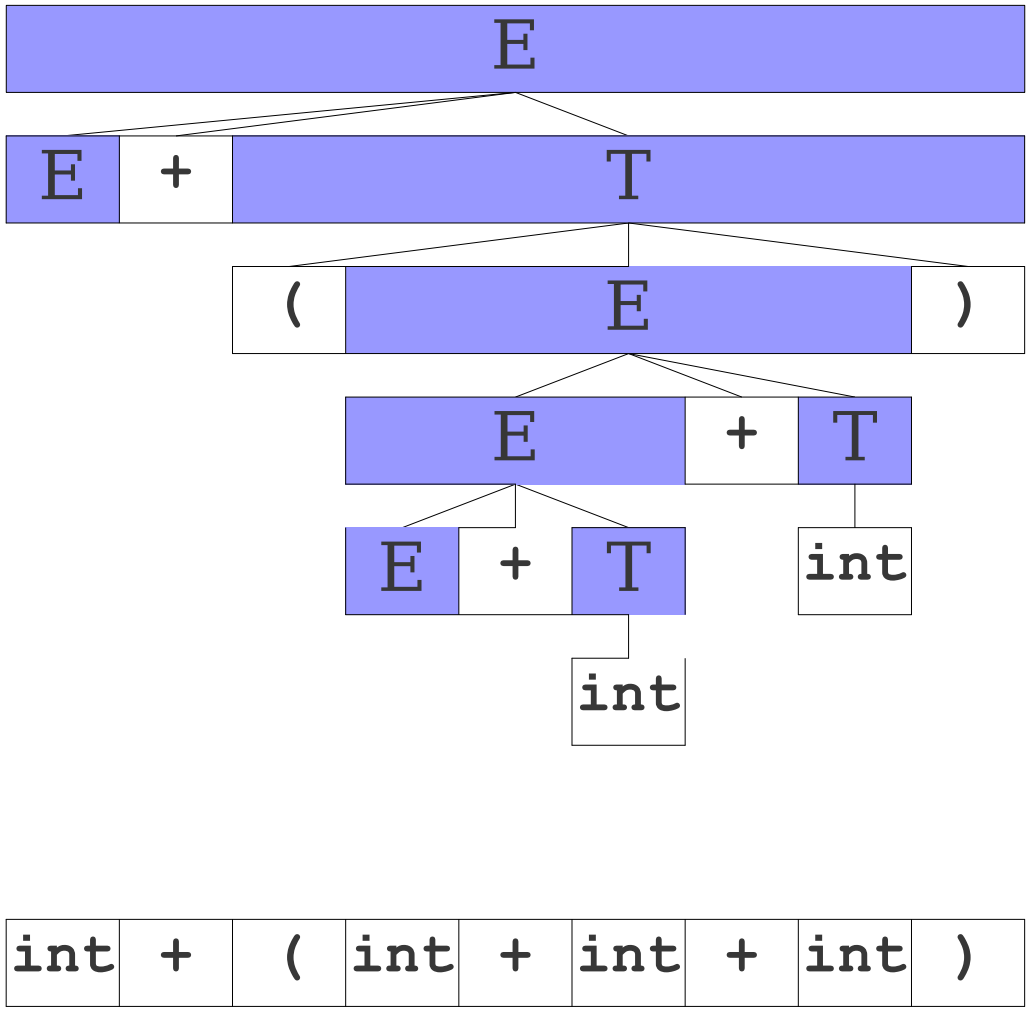
A Third View of a Bottom-Up Parse

$\Rightarrow$ 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



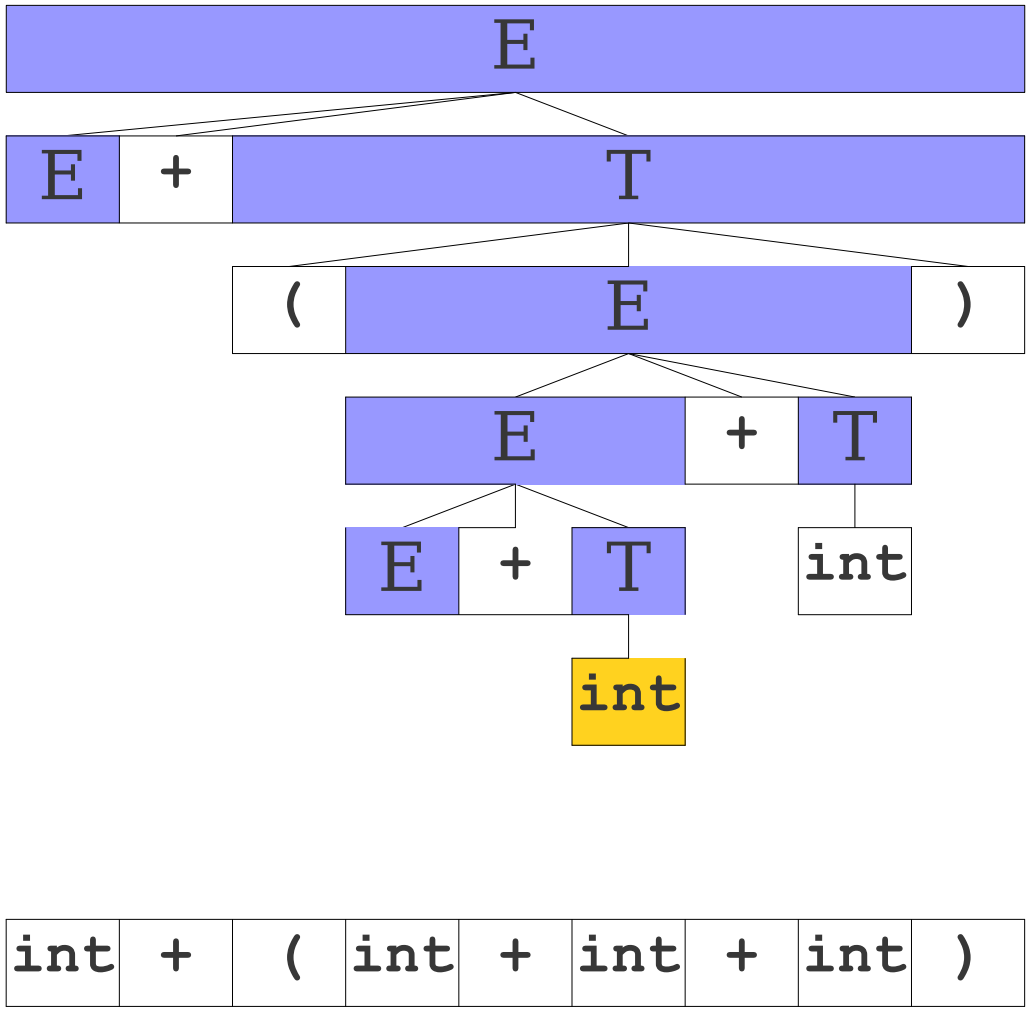
A Third View of a Bottom-Up Parse

$\Rightarrow$ 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



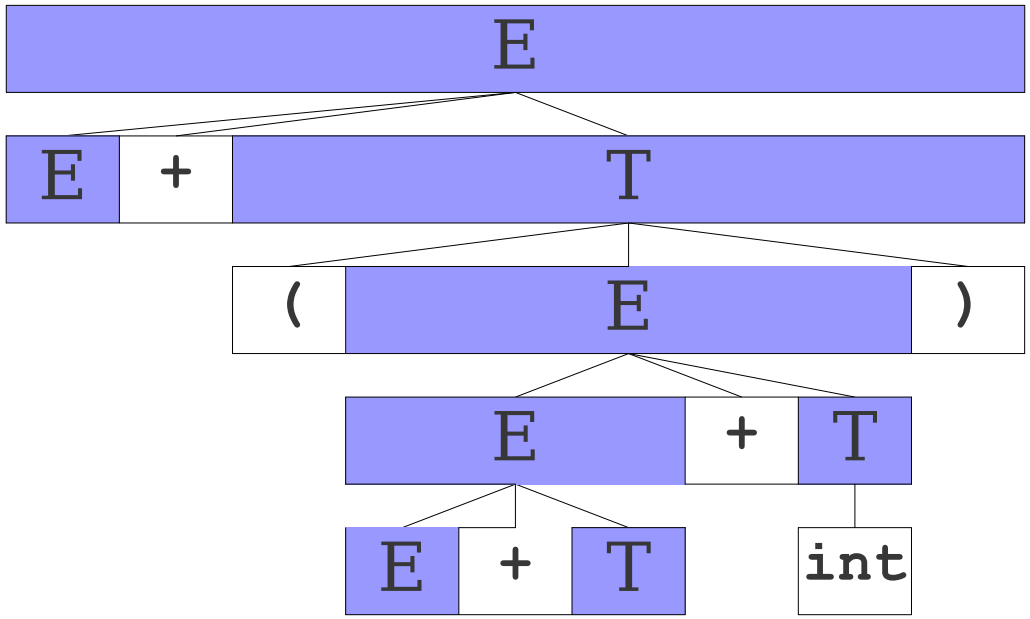
A Third View of a Bottom-Up Parse

$\Rightarrow$ 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

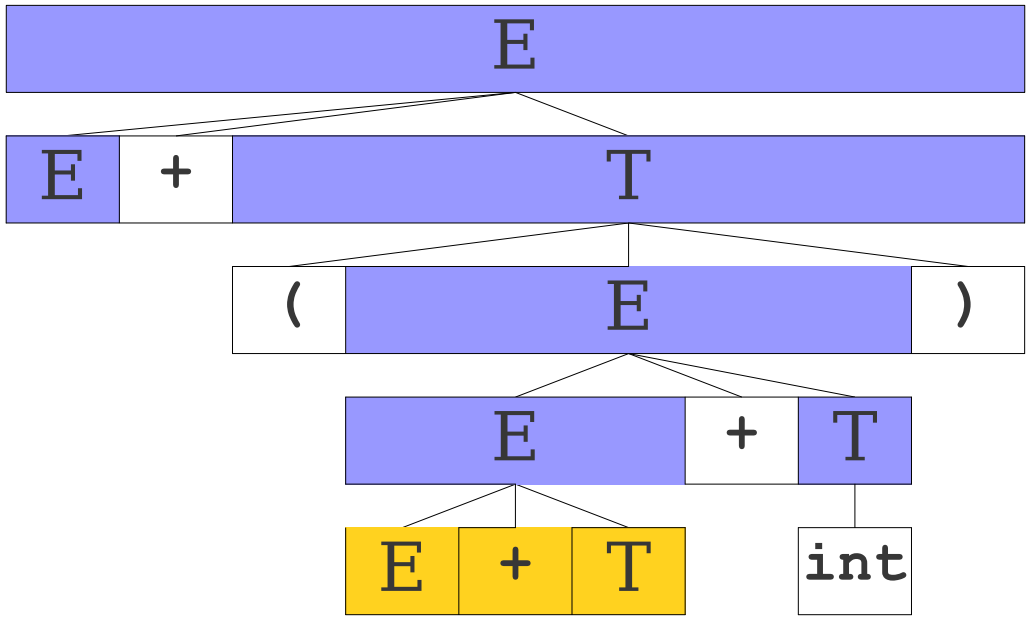
$\Rightarrow$ **E + (E + T + int)**
$\Rightarrow$ **E + (E + int)**
$\Rightarrow$ **E + (E + T)**
$\Rightarrow$ **E + (E)**
$\Rightarrow$ **E + T**
$\Rightarrow$ **E**



int	+	(	int	+	int	+	int	)

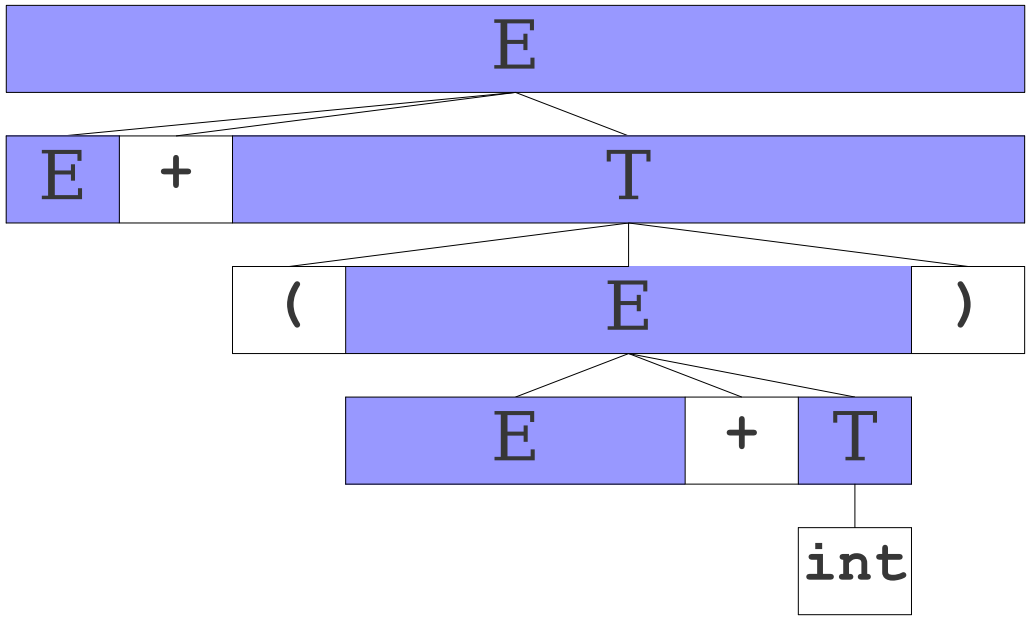
A Third View of a Bottom-Up Parse

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



int	+	(	int	+	int	+	int	)

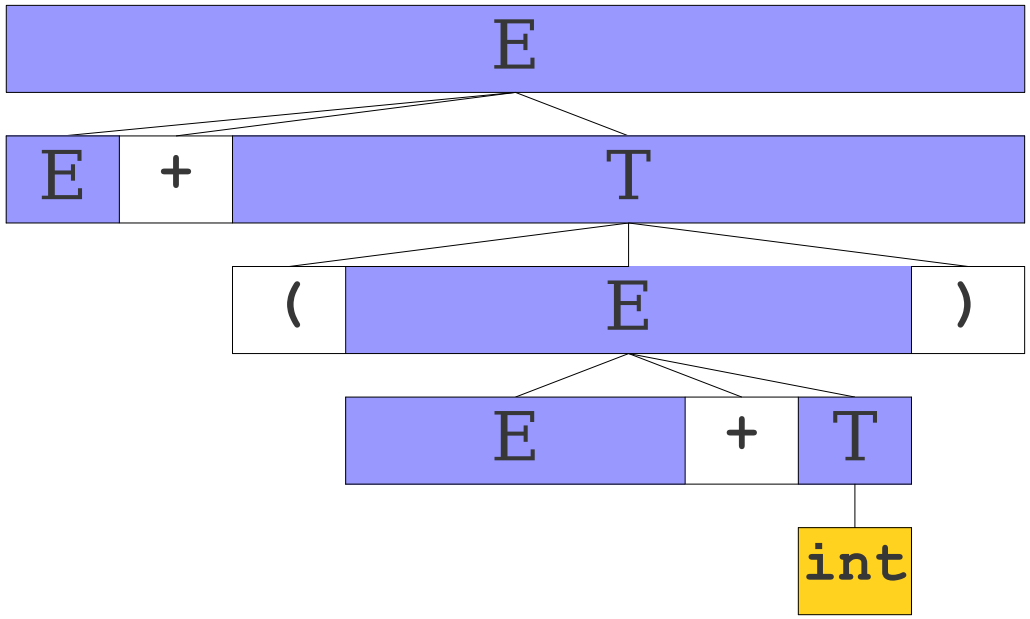
A Third View of a Bottom-Up Parse



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

int	+	(	int	+	int	+	int	)

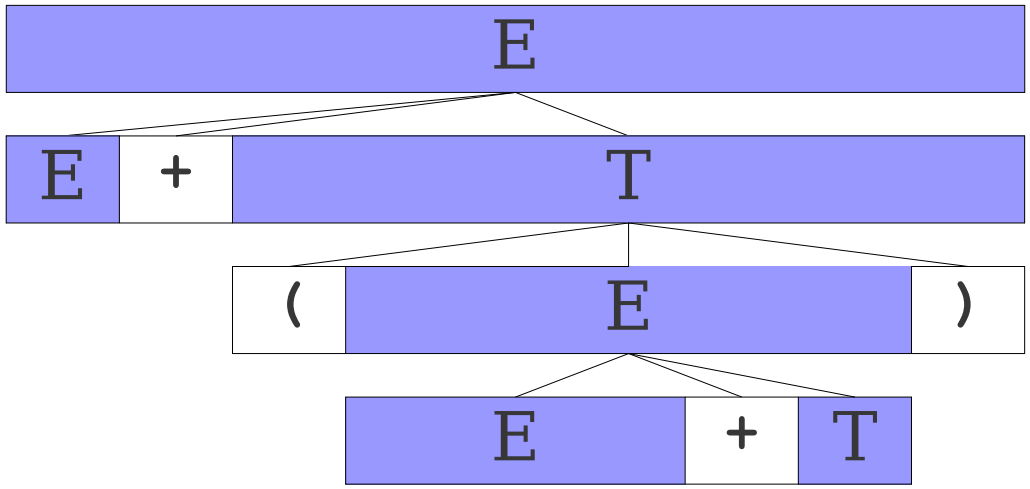
A Third View of a Bottom-Up Parse



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

int	+	(	int	+	int	+	int	)

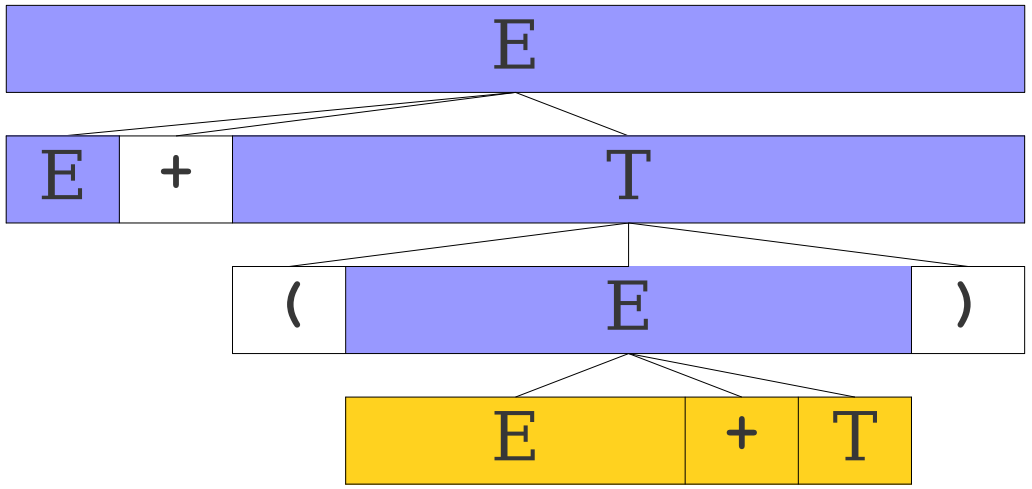
A Third View of a Bottom-Up Parse



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

int	+	(	int	+	int	+	int	)

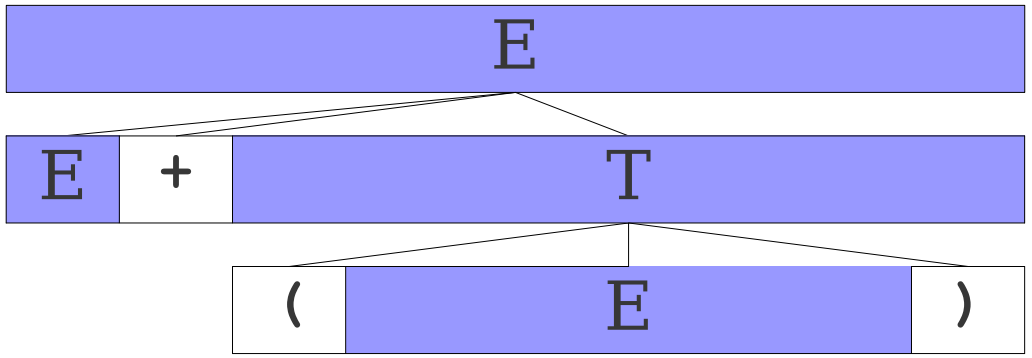
A Third View of a Bottom-Up Parse



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

int	+	(	int	+	int	+	int	)

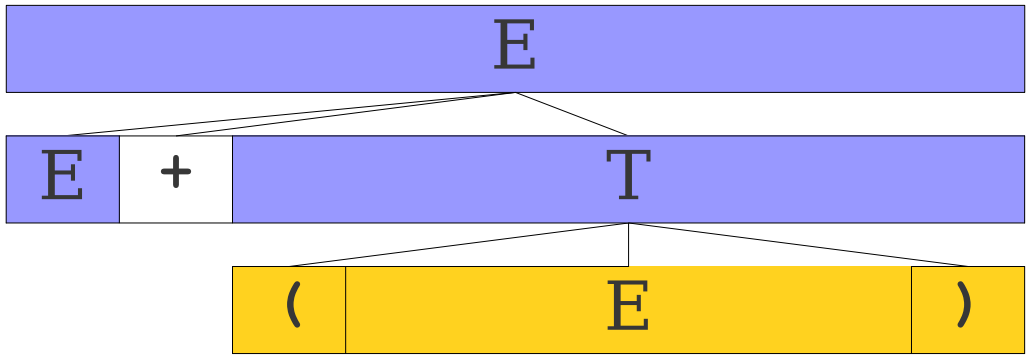
A Third View of a Bottom-Up Parse



⇒ **E + (E)**
⇒ **E + T**
⇒ **E**

int	+	(	int	+	int	+	int	)

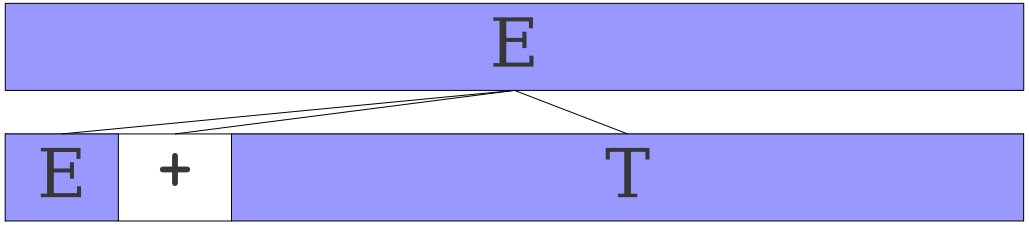
A Third View of a Bottom-Up Parse



⇒ **E + (E)**
⇒ **E + T**
⇒ **E**

int	+	(	int	+	int	+	int	)

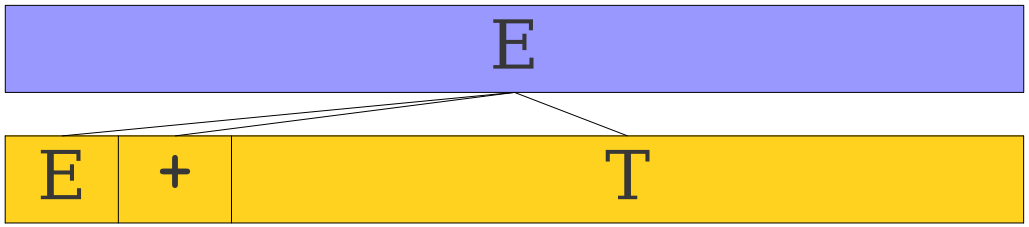
A Third View of a Bottom-Up Parse



⇒ **E + T**
⇒ **E**

int	+	(	int	+	int	+	int	)

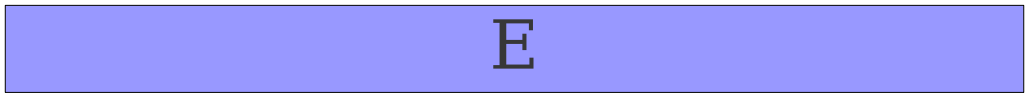
A Third View of a Bottom-Up Parse



⇒ **E + T**
⇒ **E**

int	+	(	int	+	int	+	int	)

A Third View of a Bottom-Up Parse

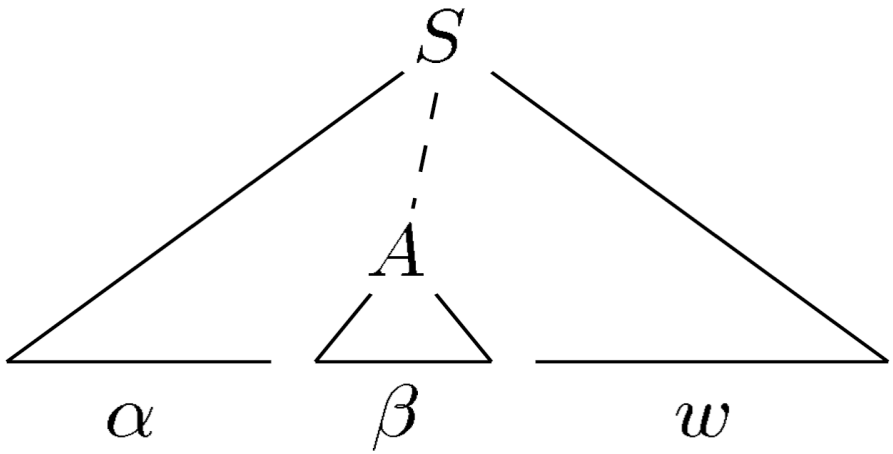


$\Rightarrow$ **E**

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

Handles

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



Handles

- Mostly 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.

The leftmost reduction
isn't always the handle.

A Detail about Handles

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

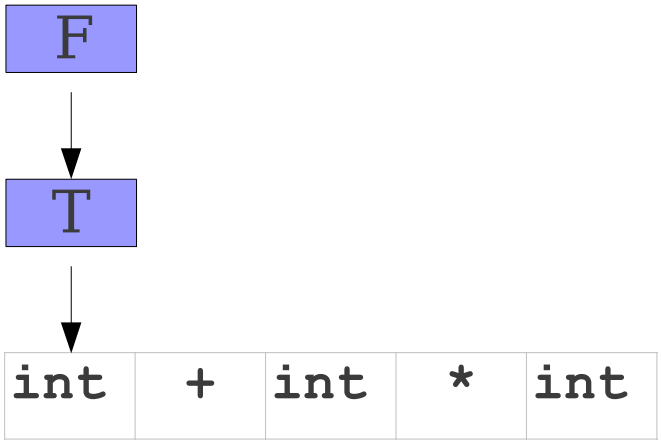
int	+	int	*	int

A Detail about Handles

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

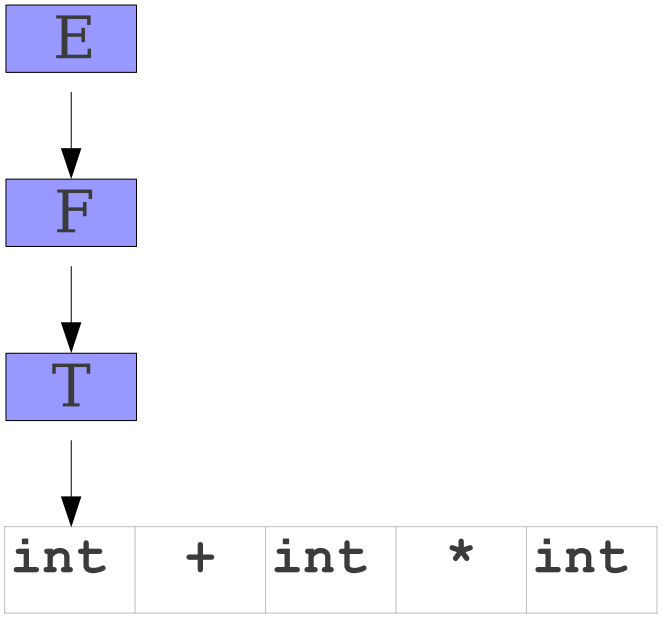
A Detail about Handles

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



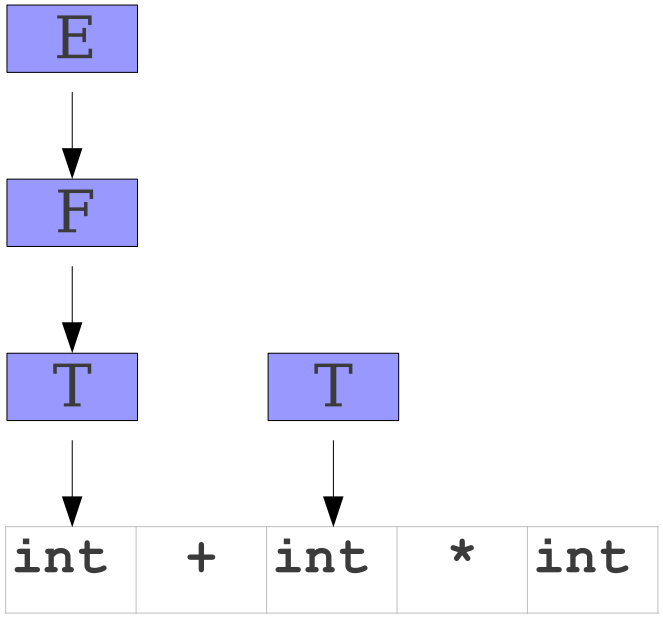
A Detail about Handles

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



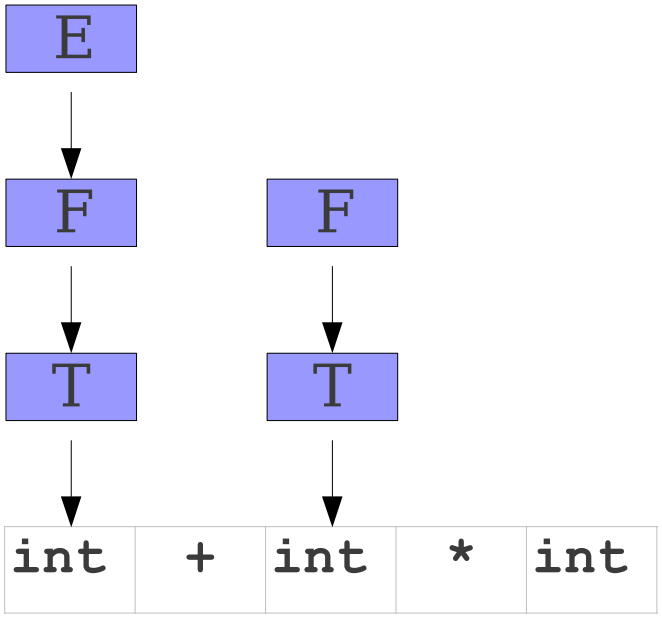
A Detail about Handles

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



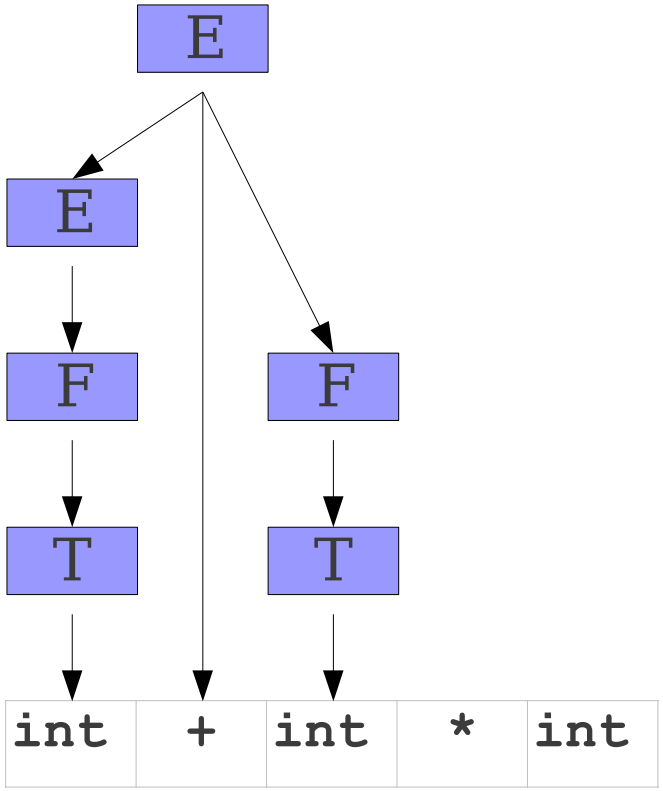
A Detail about Handles

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



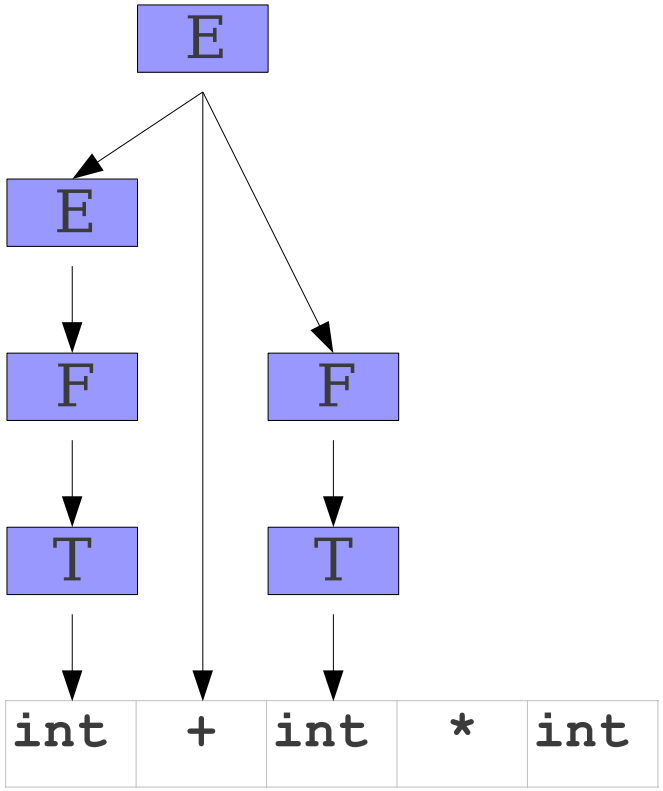
A Detail about Handles

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



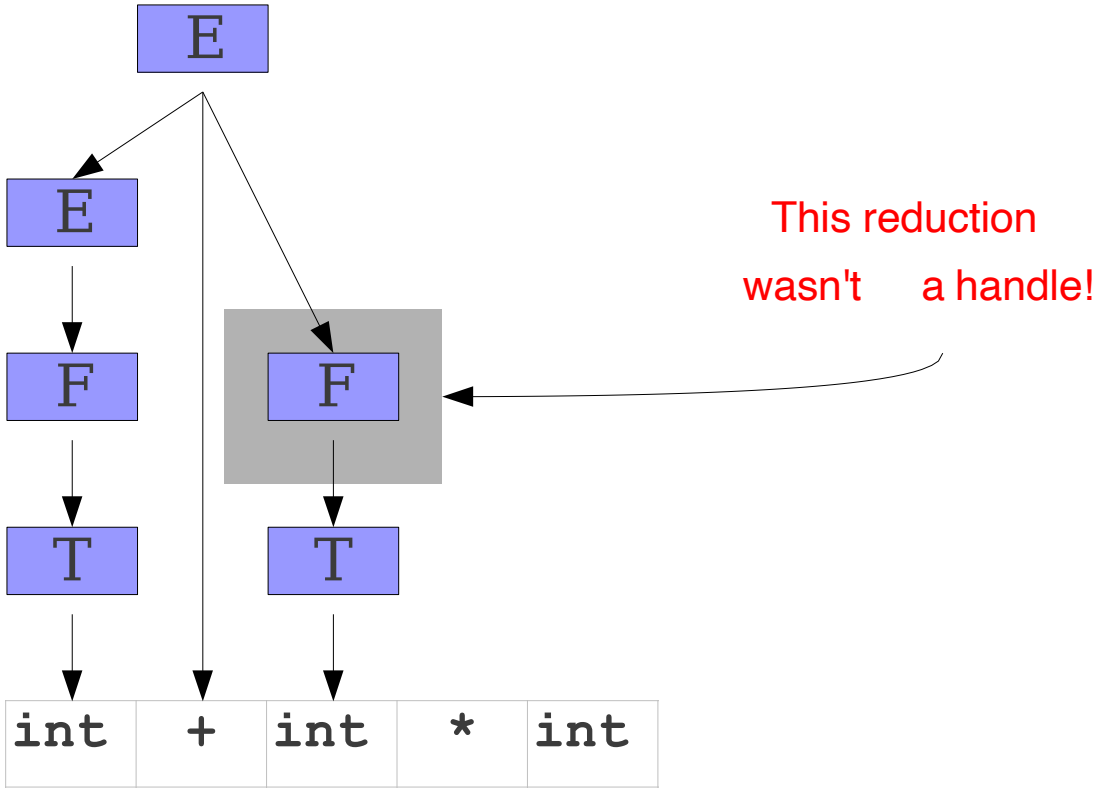
A Detail about Handles

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



A Detail about Handles

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



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.

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 $\boldsymbol{\alpha\gamma\omega}$, where $\boldsymbol{\gamma}$ is the handle and $\mathbf{A} \rightarrow \boldsymbol{\gamma}$ is a production rule.
- After reducing $\boldsymbol{\gamma}$ back to **A**, we have the string $\boldsymbol{\alpha A\omega}$.
- *Thus* $\boldsymbol{\omega}$ *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.

Shift/Reduce Parsing

- The bottom-up parsers we will consider are called **shift/reduce** parsers.
  - Contrast with the LL(1) **predict/match** parser.
- Idea: Split the input into two parts:
  - Left substring is our **work area**; all handles must be here.
  - Right substring is input we have **not yet processed**; consists purely of terminals.

Shift/Reduce Parsing

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

A Sample Shift/Reduce Parse

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

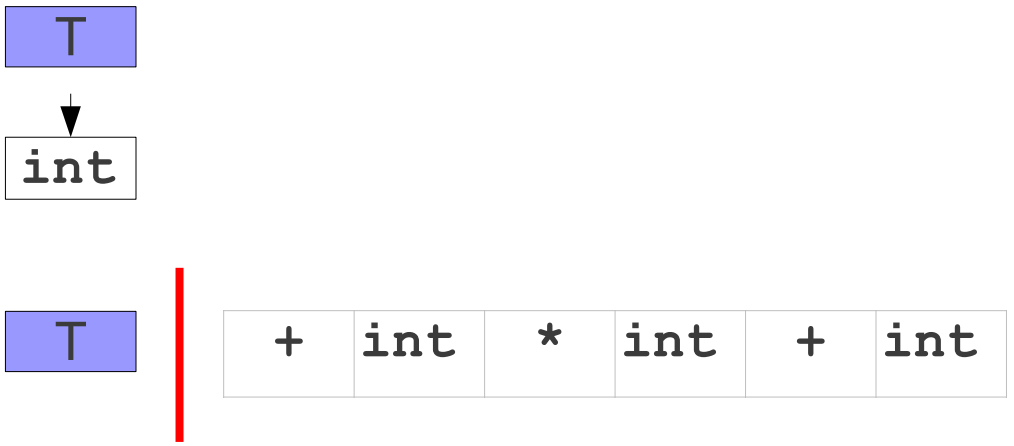
int	+	int	*	int	+	int

A Sample Shift/Reduce Parse

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

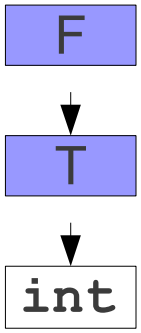
A Sample Shift/Reduce Parse

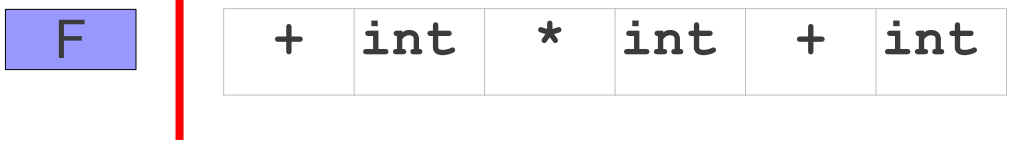
E → F
E → E + F
F → F * T
F → T
T → int
T → (E)



A Sample Shift/Reduce Parse

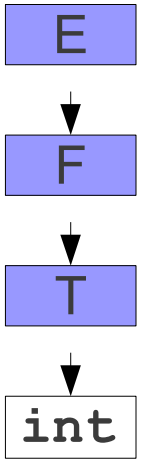
E → F
E → E + F
F → F * T
F → T
T → int
T → (E)





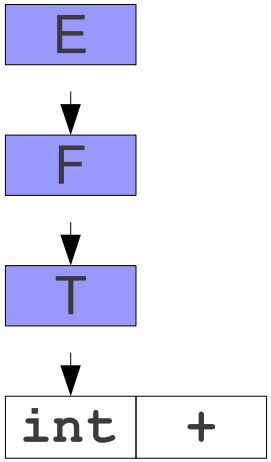
A Sample Shift/Reduce Parse

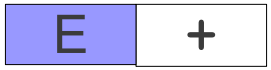
$E \rightarrow F$
$E \rightarrow E + F$
$F \rightarrow F * T$
$F \rightarrow T$
$T \rightarrow \texttt{int}$
$T \rightarrow (E)$



A Sample Shift/Reduce Parse

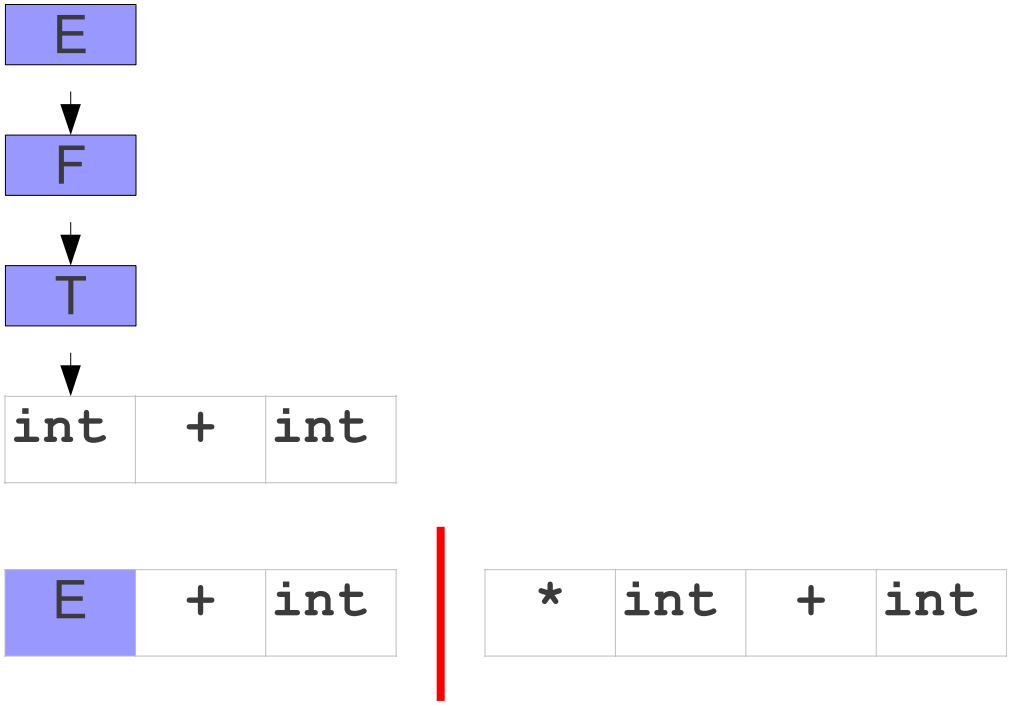
E → F
E → E + F
F → F * T
F → T
T → int
T → (E)





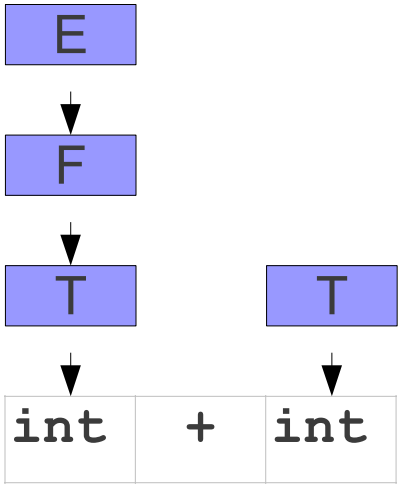
A Sample Shift/Reduce Parse

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



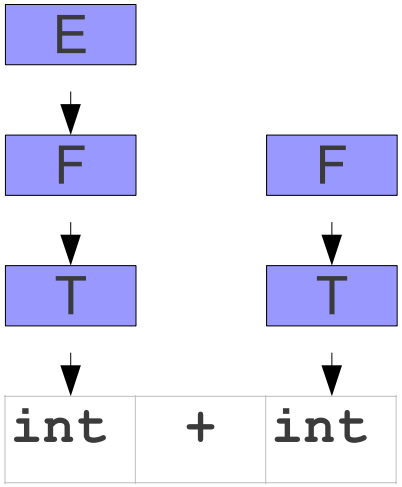
A Sample Shift/Reduce Parse

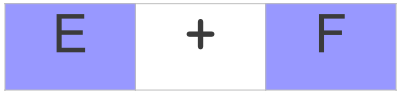
E → F
E → E + F
F → F * T
F → T
T → int
T → (E)



A Sample Shift/Reduce Parse

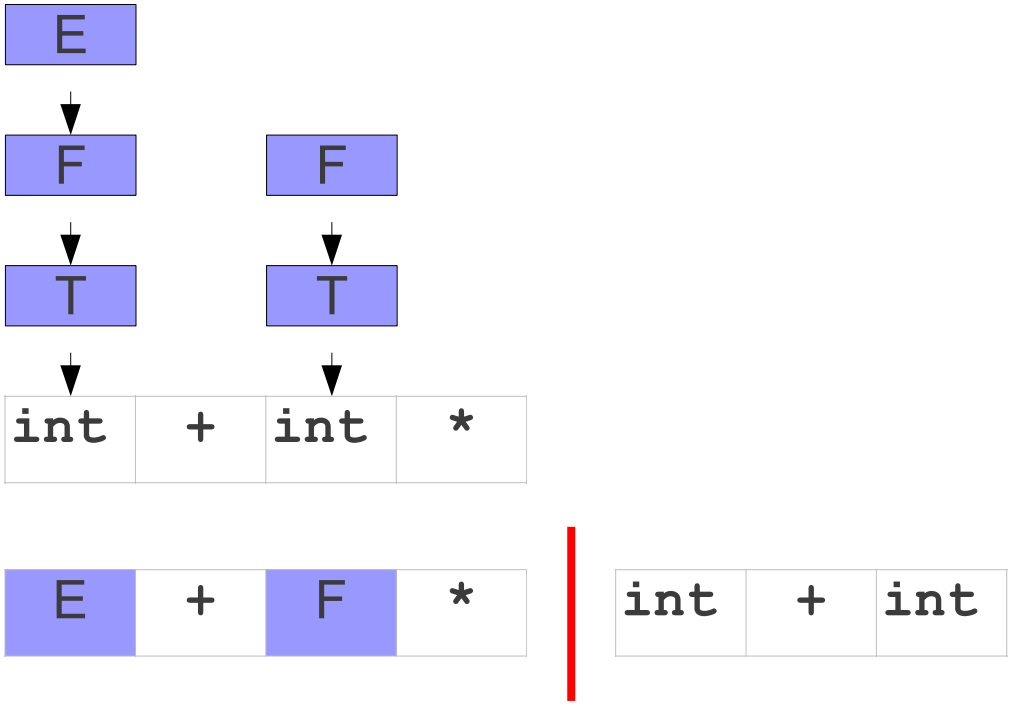
E → F
E → E + F
F → F * T
F → T
T → int
T → (E)





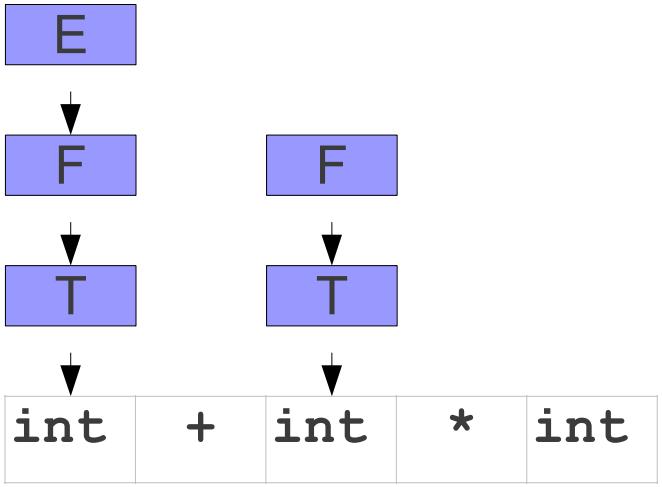
A Sample Shift/Reduce Parse

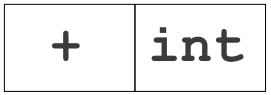
E → F
E → E + F
F → F * T
F → T
T → int
T → (E)



A Sample Shift/Reduce Parse

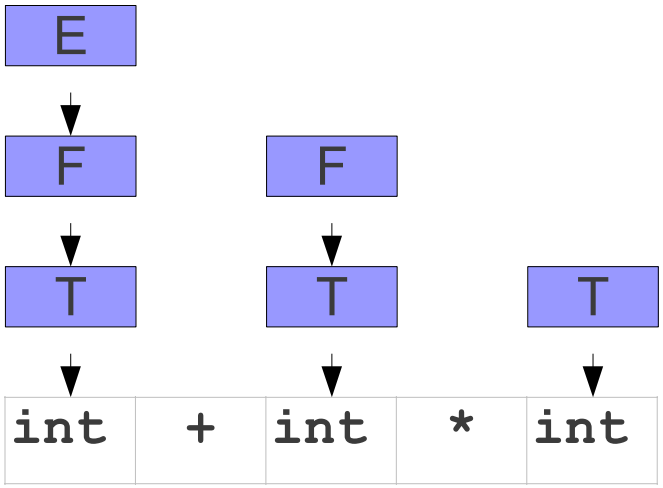
E → F
E → E + F
F → F * T
F → T
T → int
T → (E)





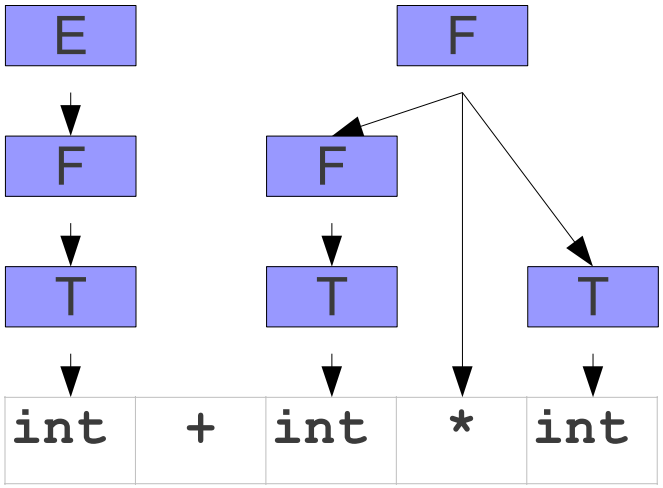
A Sample Shift/Reduce Parse

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



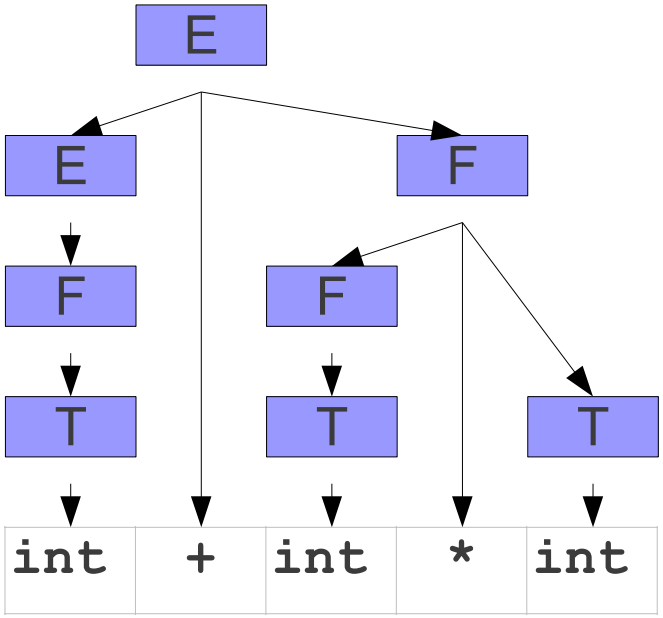
A Sample Shift/Reduce Parse

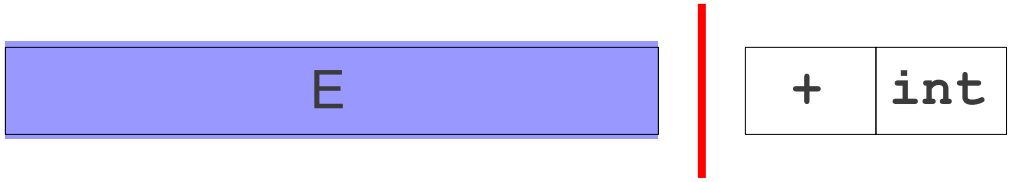
E → F
E → E + F
F → F * T
F → T
T → int
T → (E)



A Sample Shift/Reduce Parse

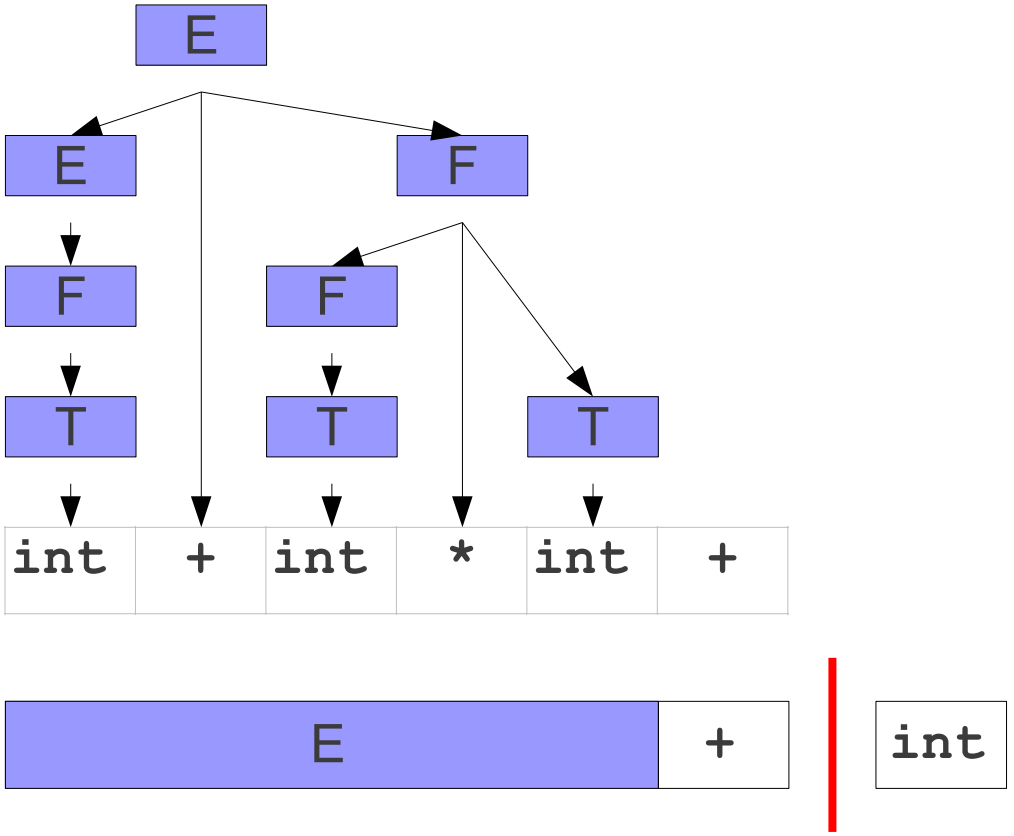
$E \rightarrow F$
$E \rightarrow E + F$
$F \rightarrow F * T$
$F \rightarrow T$
$T \rightarrow \texttt{int}$
$T \rightarrow (E)$





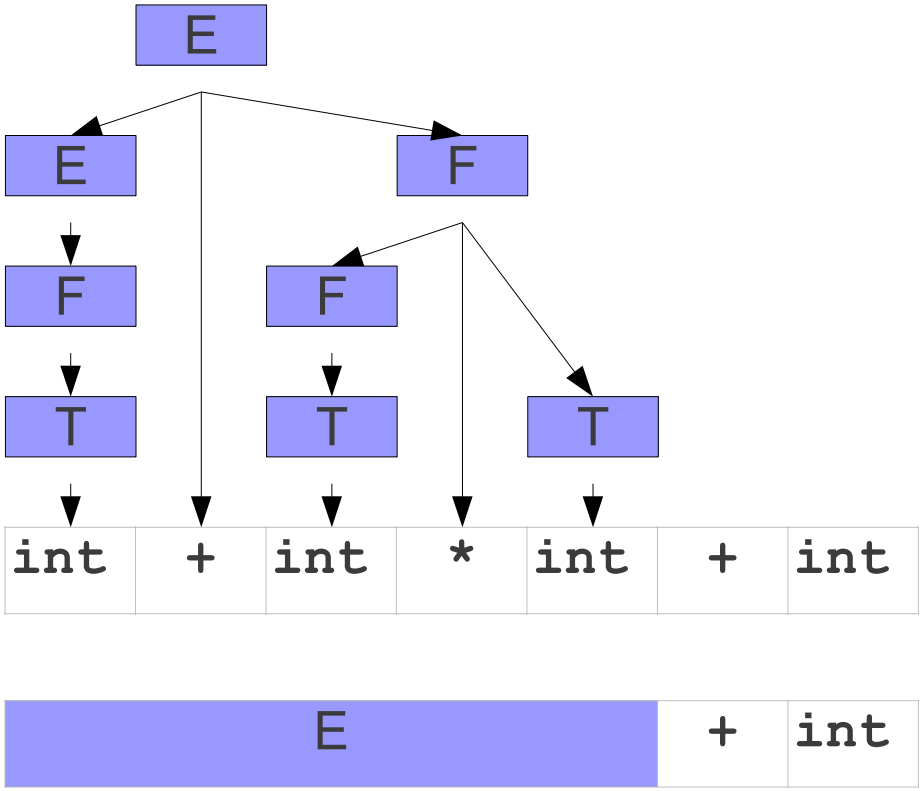
A Sample Shift/Reduce Parse

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



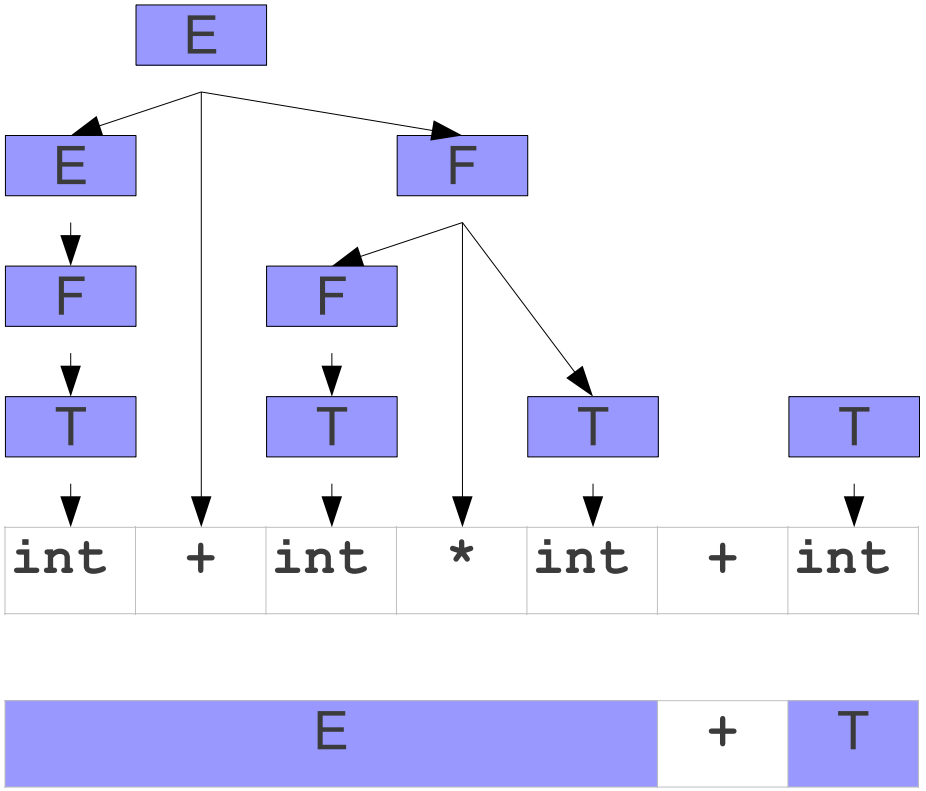
A Sample Shift/Reduce Parse

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



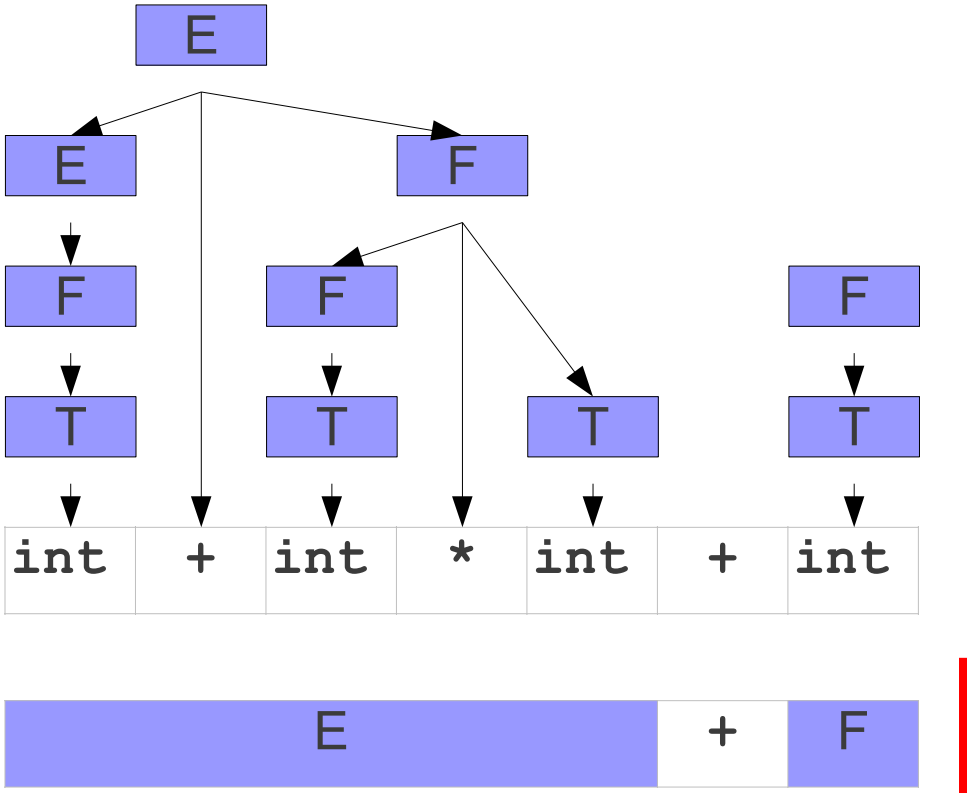
A Sample Shift/Reduce Parse

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



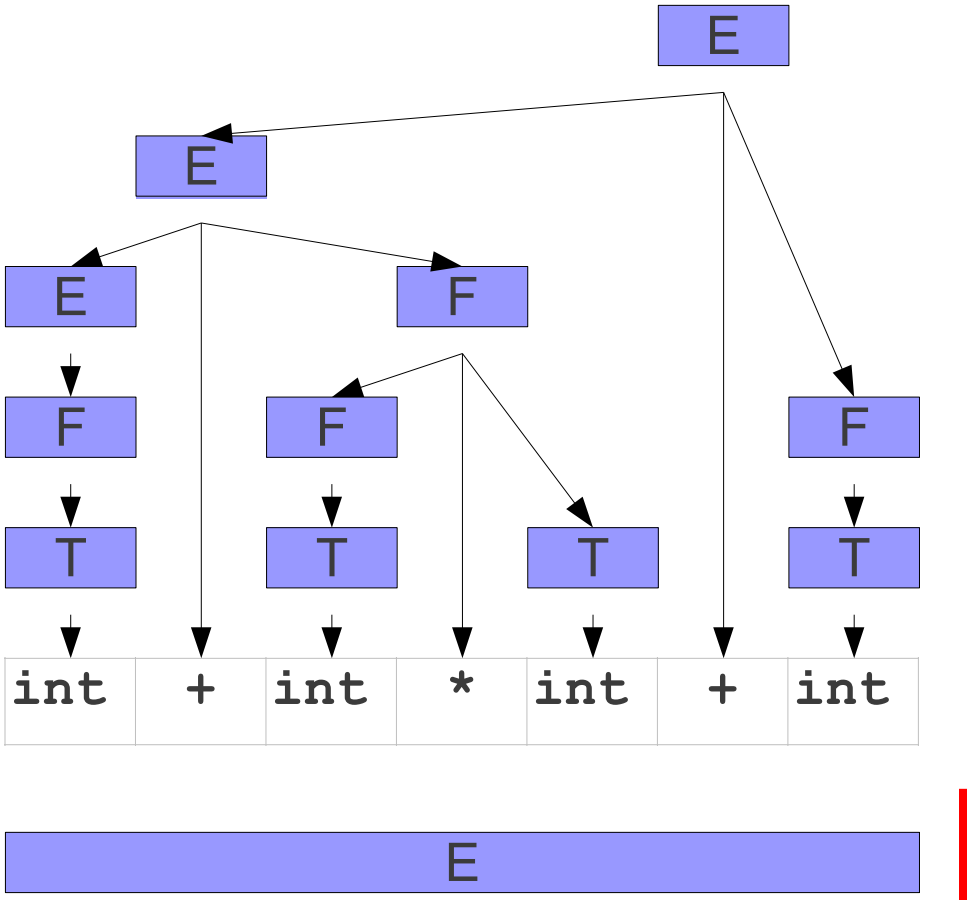
A Sample Shift/Reduce Parse

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



A Sample Shift/Reduce Parse

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



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.

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.
- i.e. No need to “uncover” something to do a reduction.

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.
- i.e. 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 on to 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?

Where are handles?

Look Back to the Previous Example

A Sample Shift/Reduce Parse

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

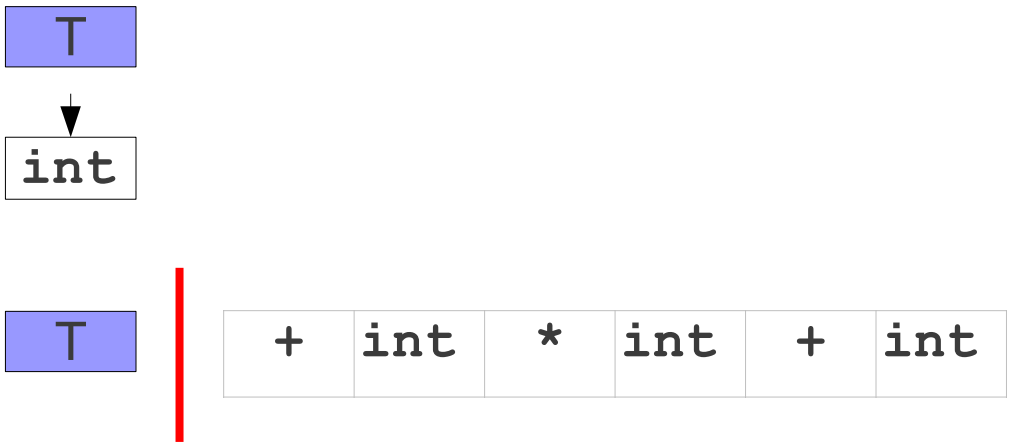
int	+	int	*	int	+	int

A Sample Shift/Reduce Parse

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

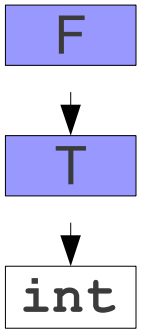
A Sample Shift/Reduce Parse

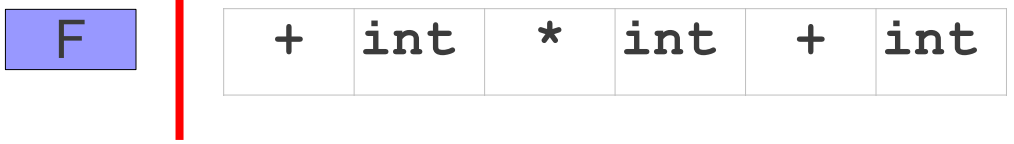
E → F
E → E + F
F → F * T
F → T
T → int
T → (E)



A Sample Shift/Reduce Parse

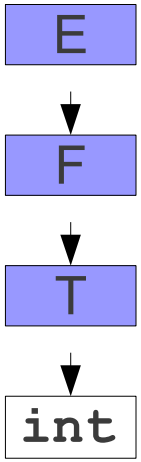
E → F
E → E + F
F → F * T
F → T
T → int
T → (E)





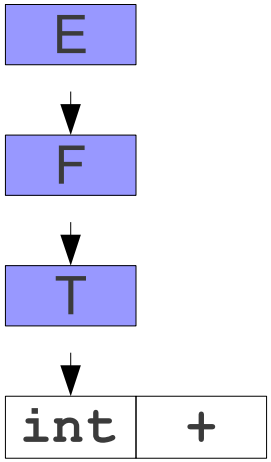
A Sample Shift/Reduce Parse

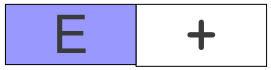
$E \rightarrow F$
$E \rightarrow E + F$
$F \rightarrow F * T$
$F \rightarrow T$
$T \rightarrow \texttt{int}$
$T \rightarrow (E)$



A Sample Shift/Reduce Parse

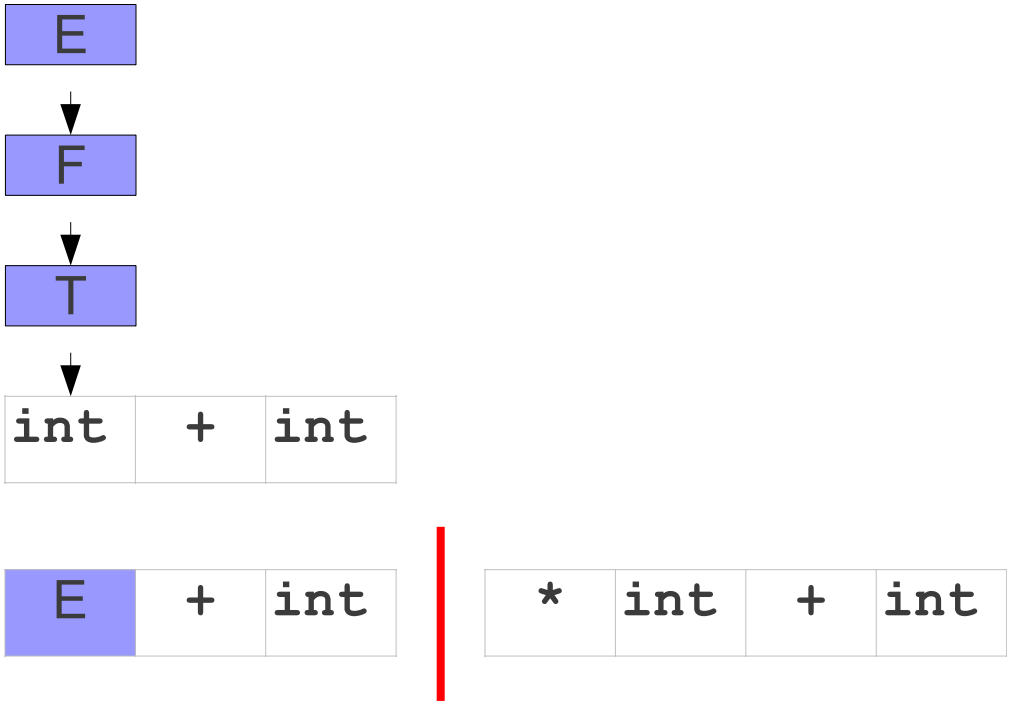
E → F
E → E + F
F → F * T
F → T
T → int
T → (E)





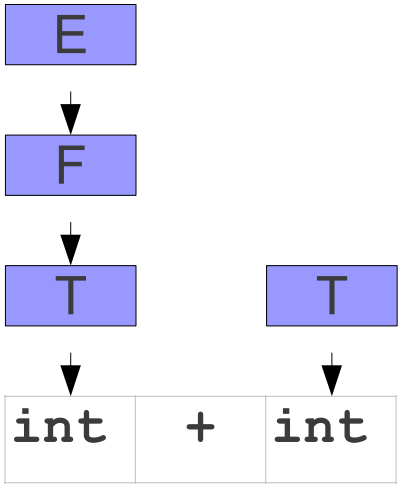
A Sample Shift/Reduce Parse

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



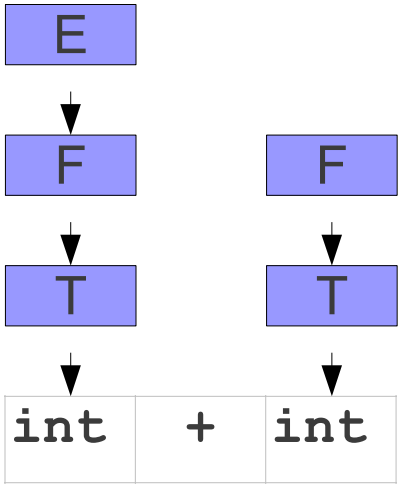
A Sample Shift/Reduce Parse

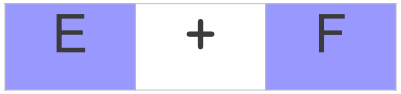
E → F
E → E + F
F → F * T
F → T
T → int
T → (E)



A Sample Shift/Reduce Parse

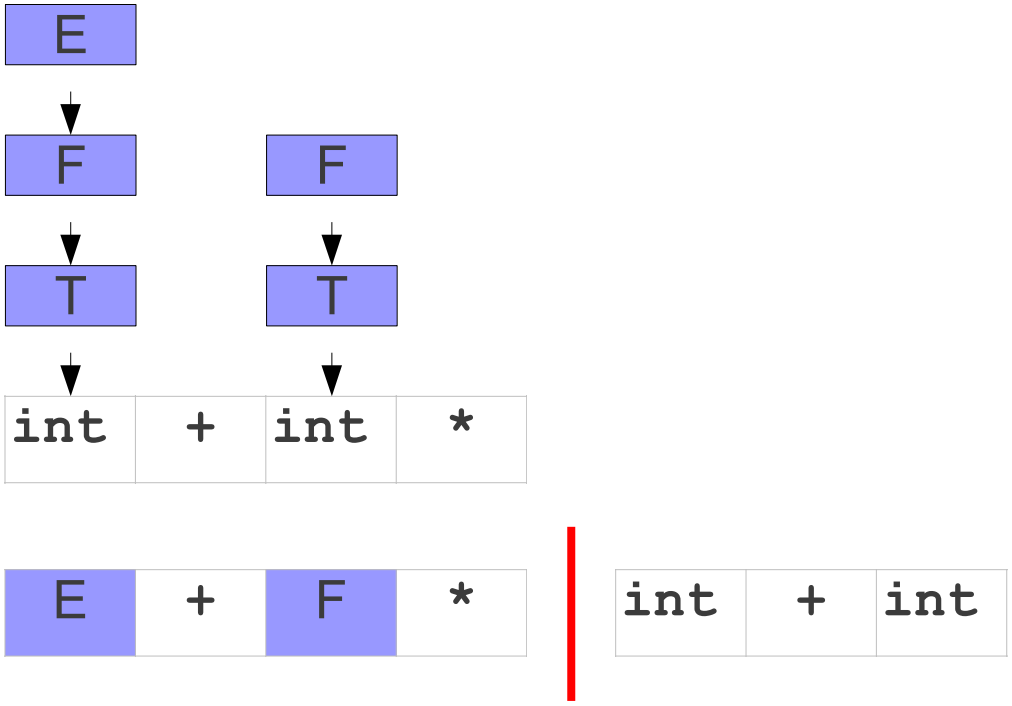
E → F
E → E + F
F → F * T
F → T
T → int
T → (E)





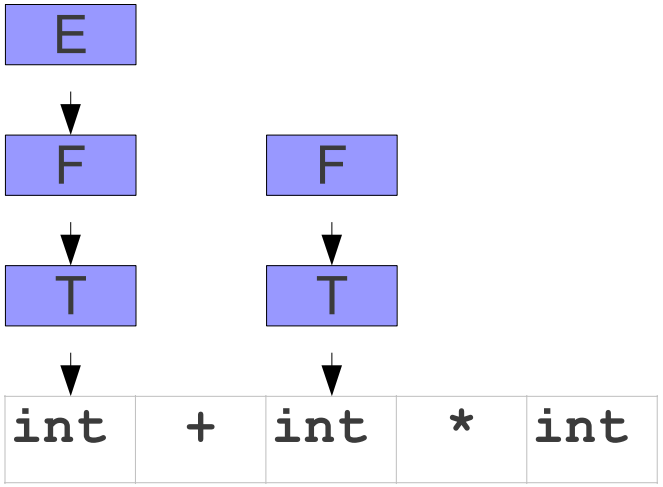
A Sample Shift/Reduce Parse

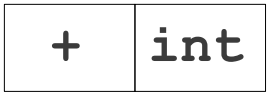
E → F
E → E + F
F → F * T
F → T
T → int
T → (E)



A Sample Shift/Reduce Parse

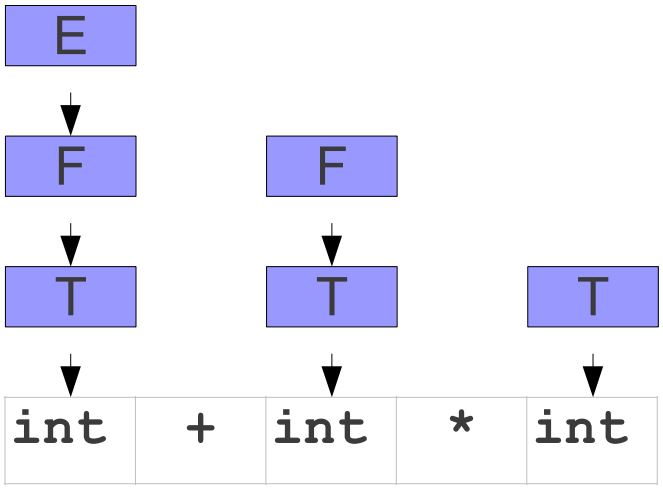
E → F
E → E + F
F → F * T
F → T
T → int
T → (E)





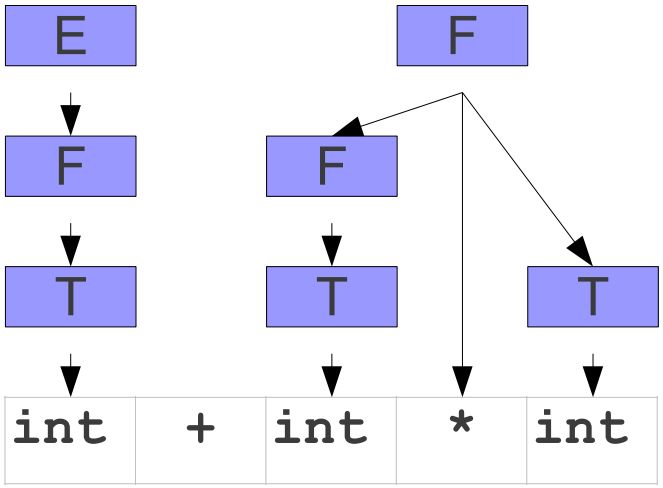
A Sample Shift/Reduce Parse

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



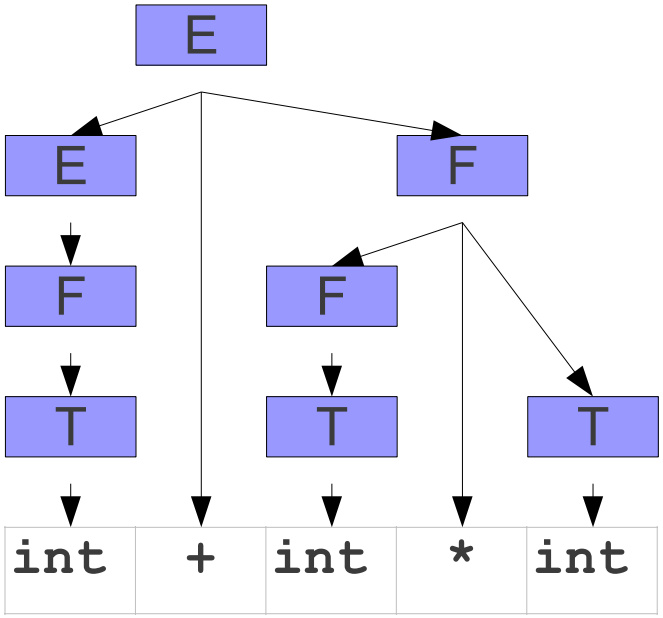
A Sample Shift/Reduce Parse

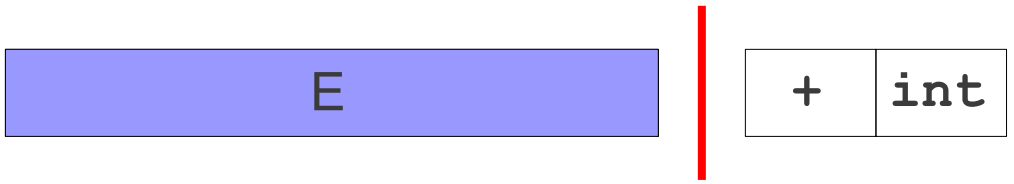
E → F
E → E + F
F → F * T
F → T
T → int
T → (E)



A Sample Shift/Reduce Parse

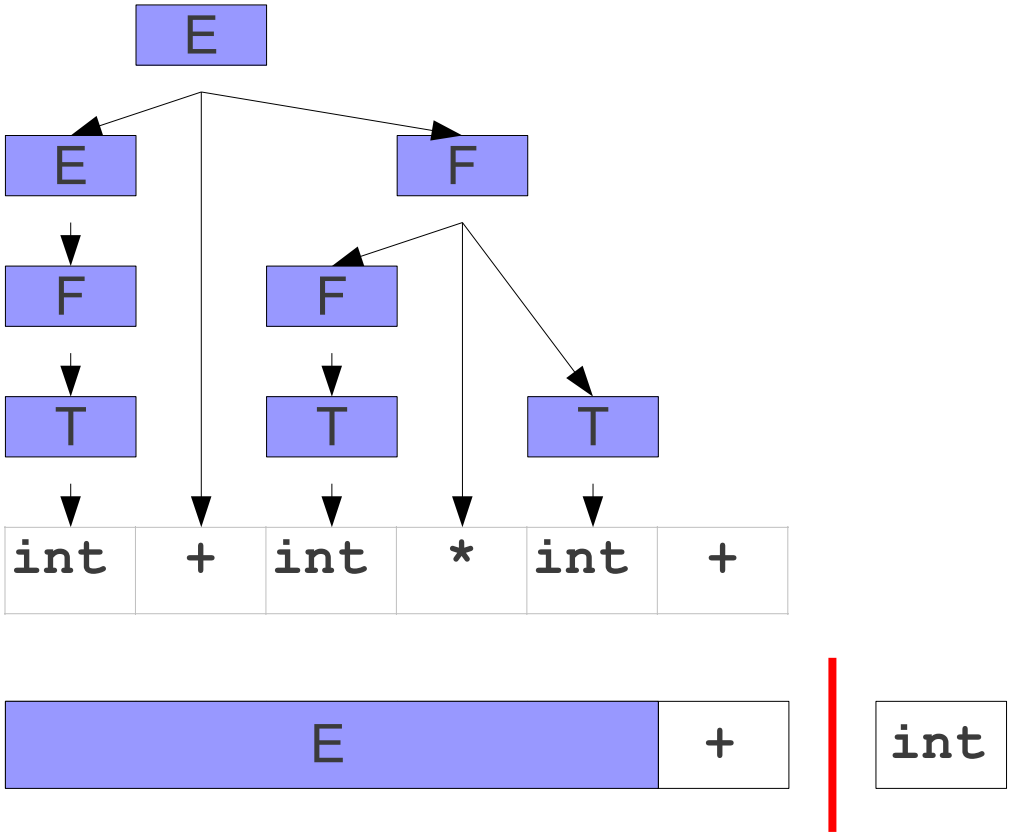
E → F
E → E + F
F → F * T
F → T
T → int
T → (E)





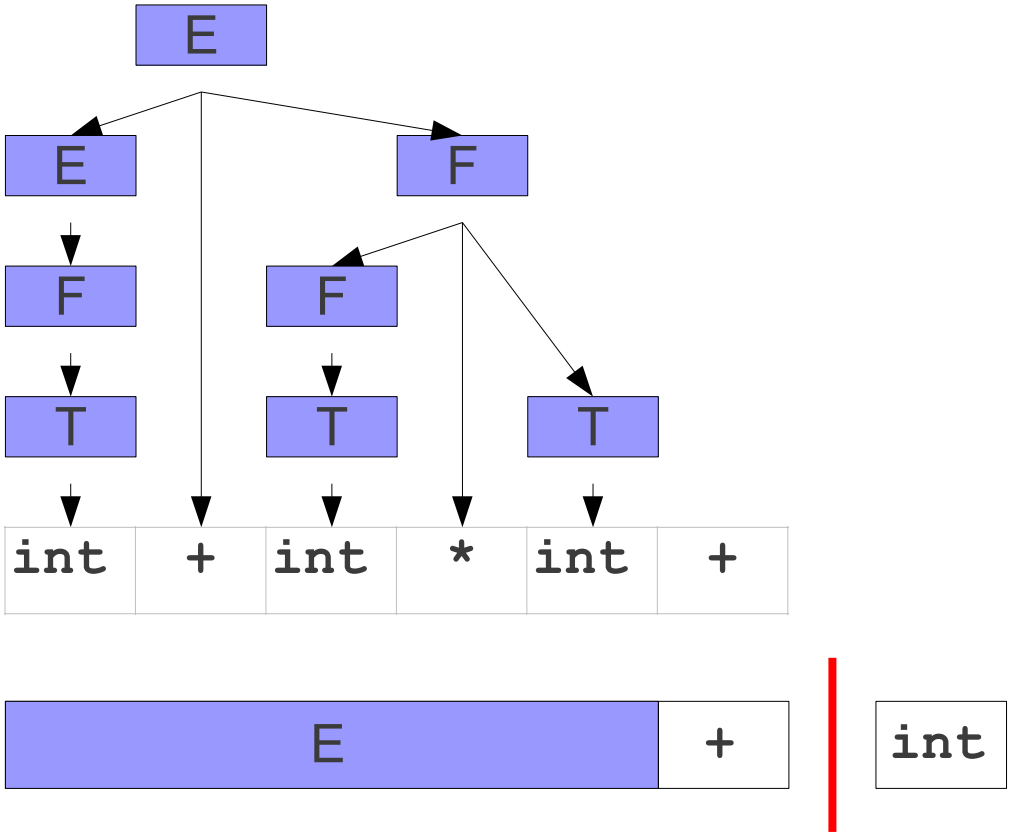
A Sample Shift/Reduce Parse

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



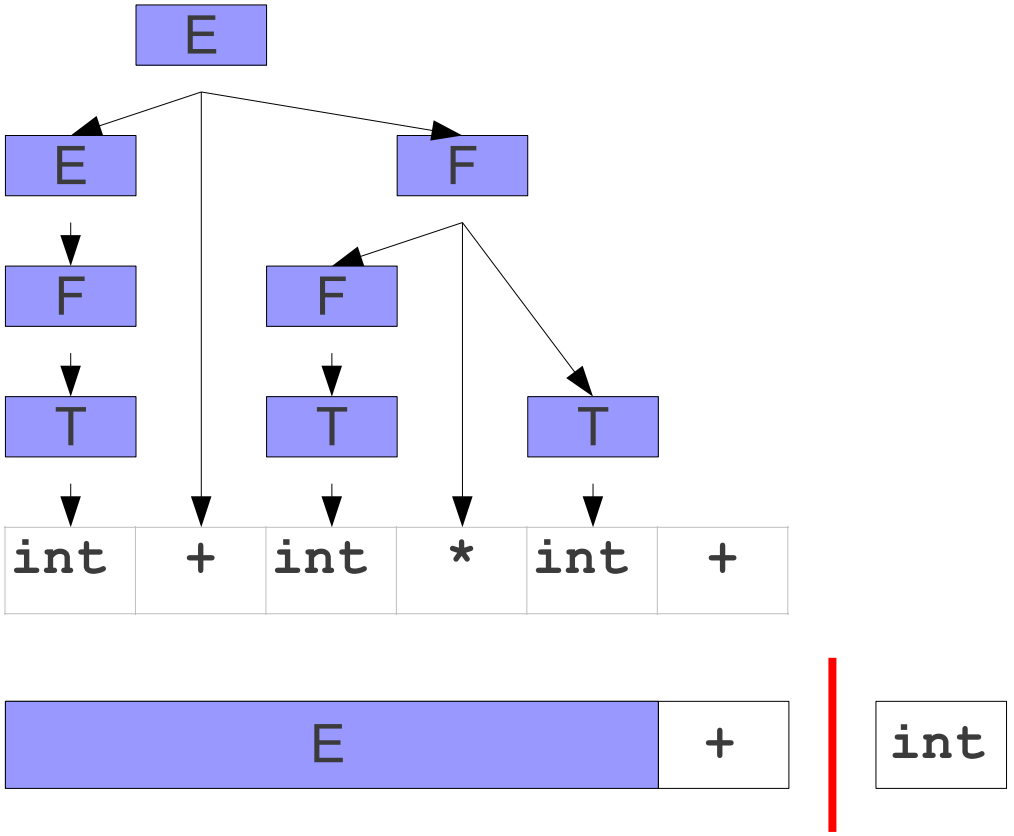
A Sample Shift/Reduce Parse

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



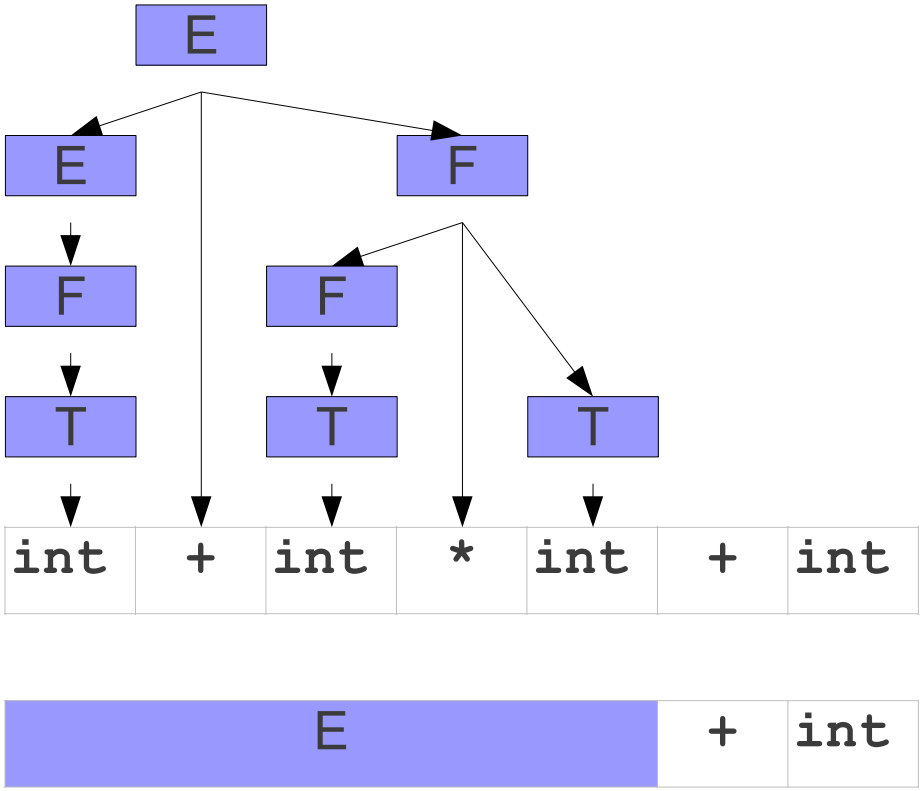
A Sample Shift/Reduce Parse

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



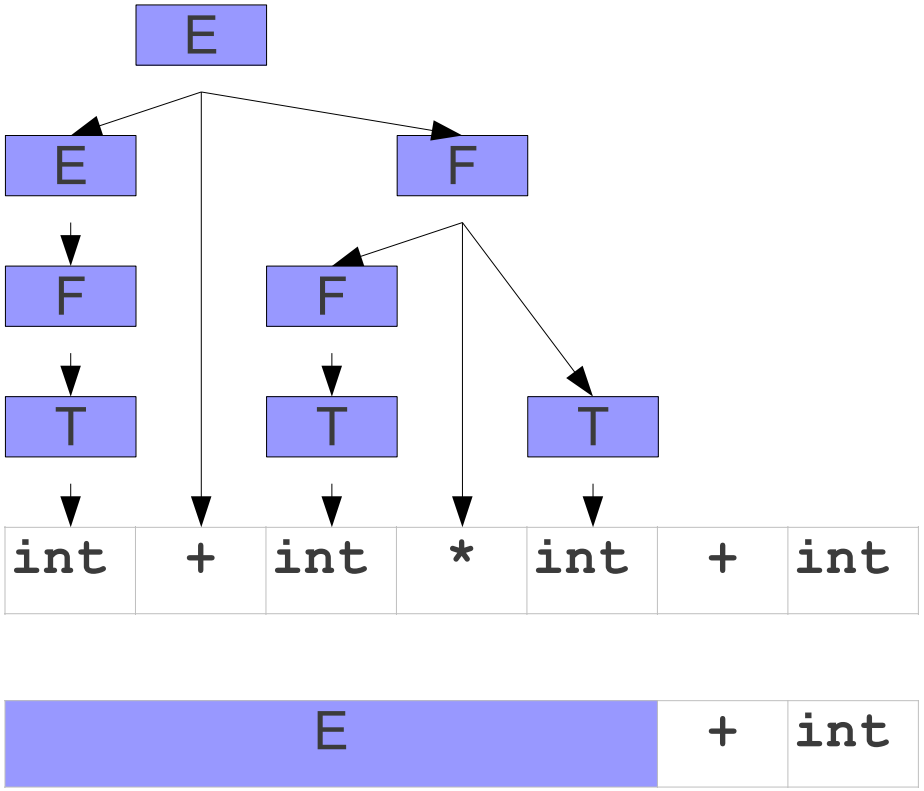
A Sample Shift/Reduce Parse

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



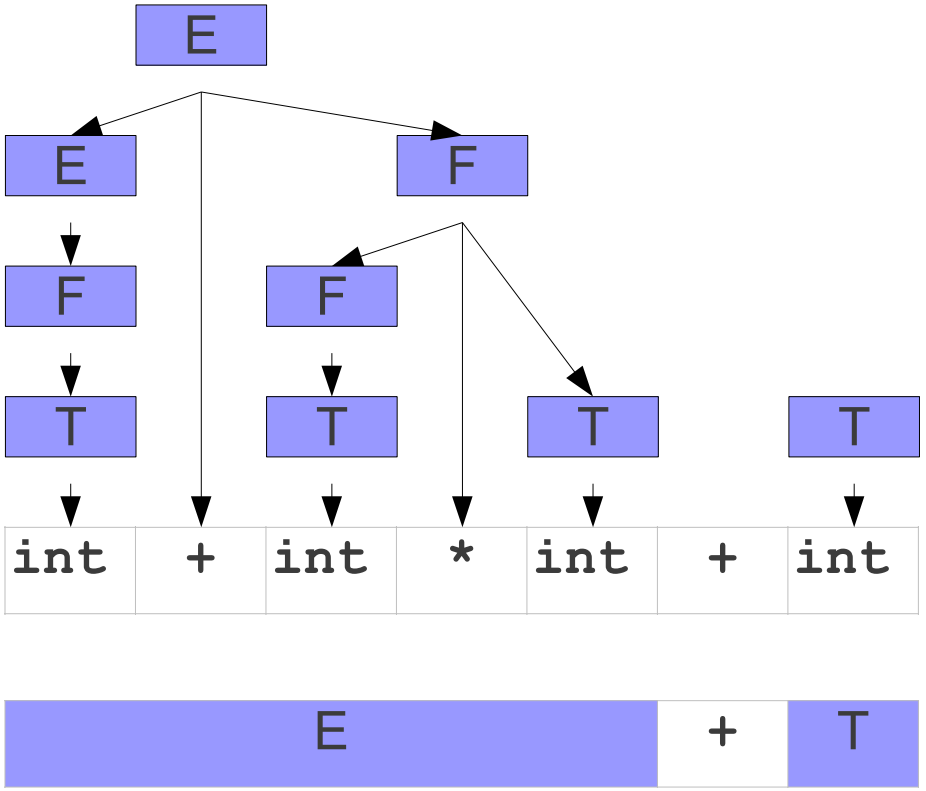
A Sample Shift/Reduce Parse

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



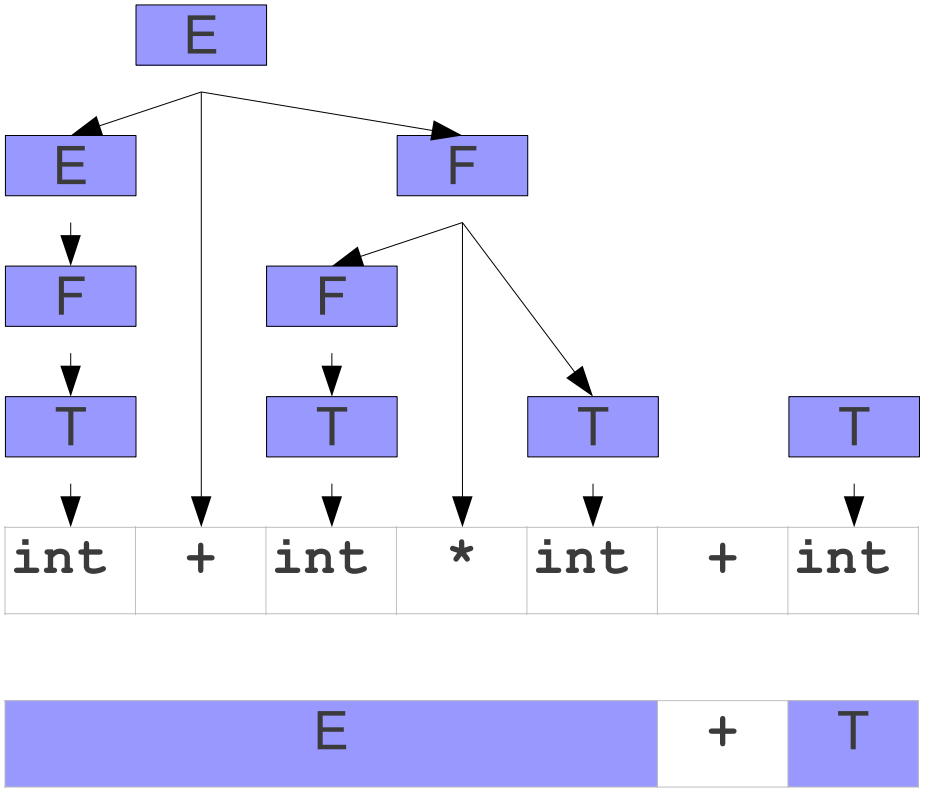
A Sample Shift/Reduce Parse

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



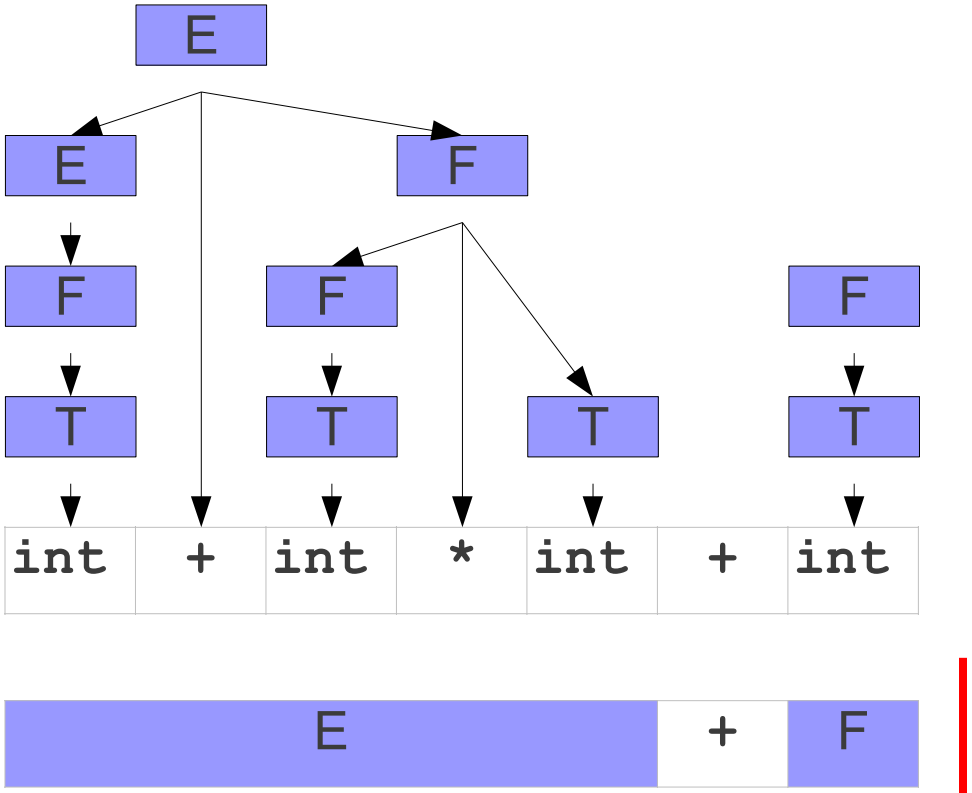
A Sample Shift/Reduce Parse

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



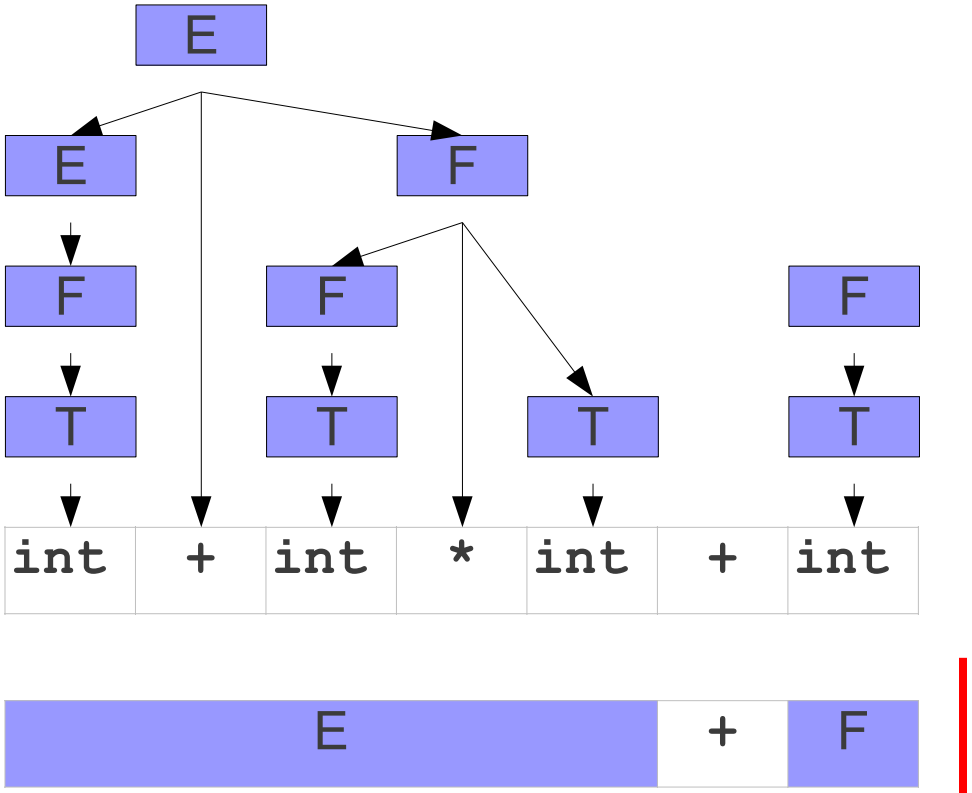
A Sample Shift/Reduce Parse

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



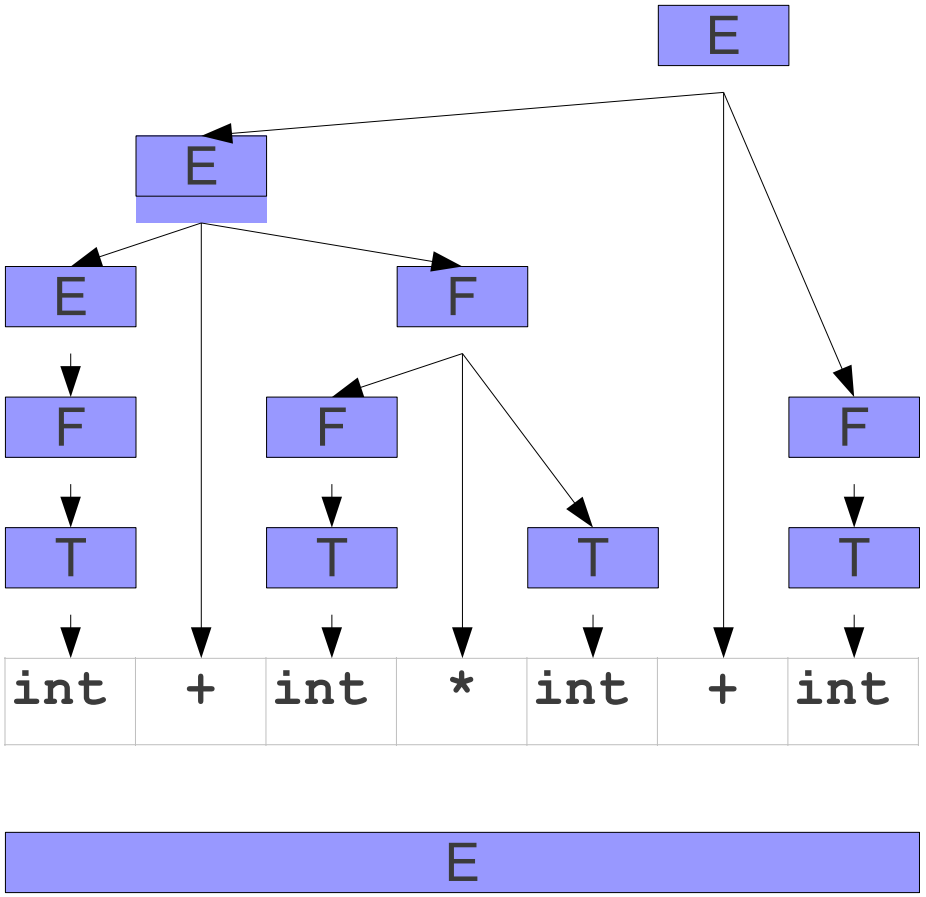
A Sample Shift/Reduce Parse

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



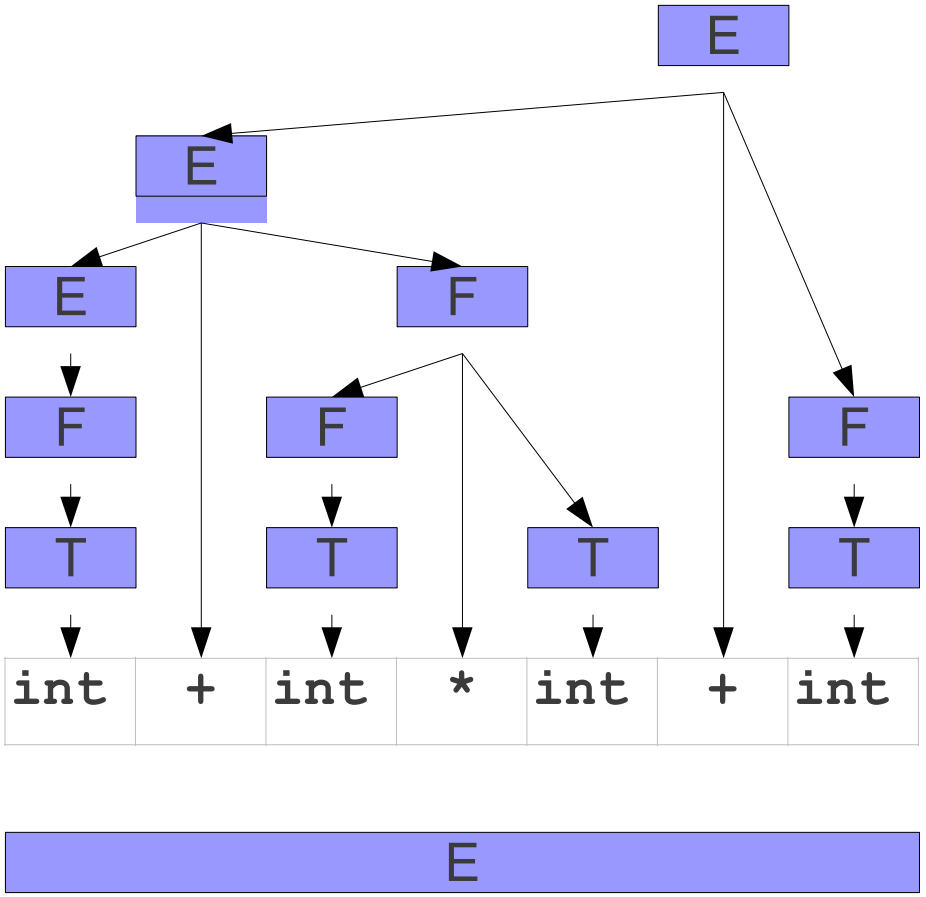
A Sample Shift/Reduce Parse

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



A Sample Shift/Reduce Parse

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



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**.

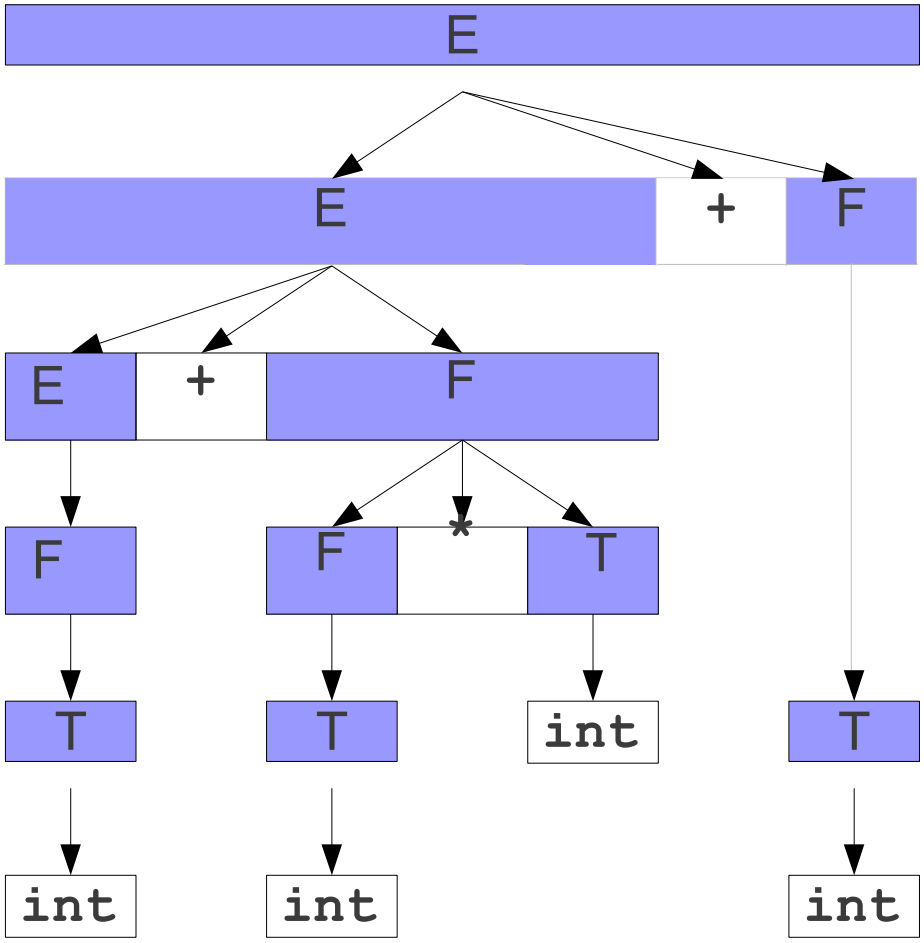
Exploring the Left Side

- Can *any* string appear on the left side of the parser?
- Are there any 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

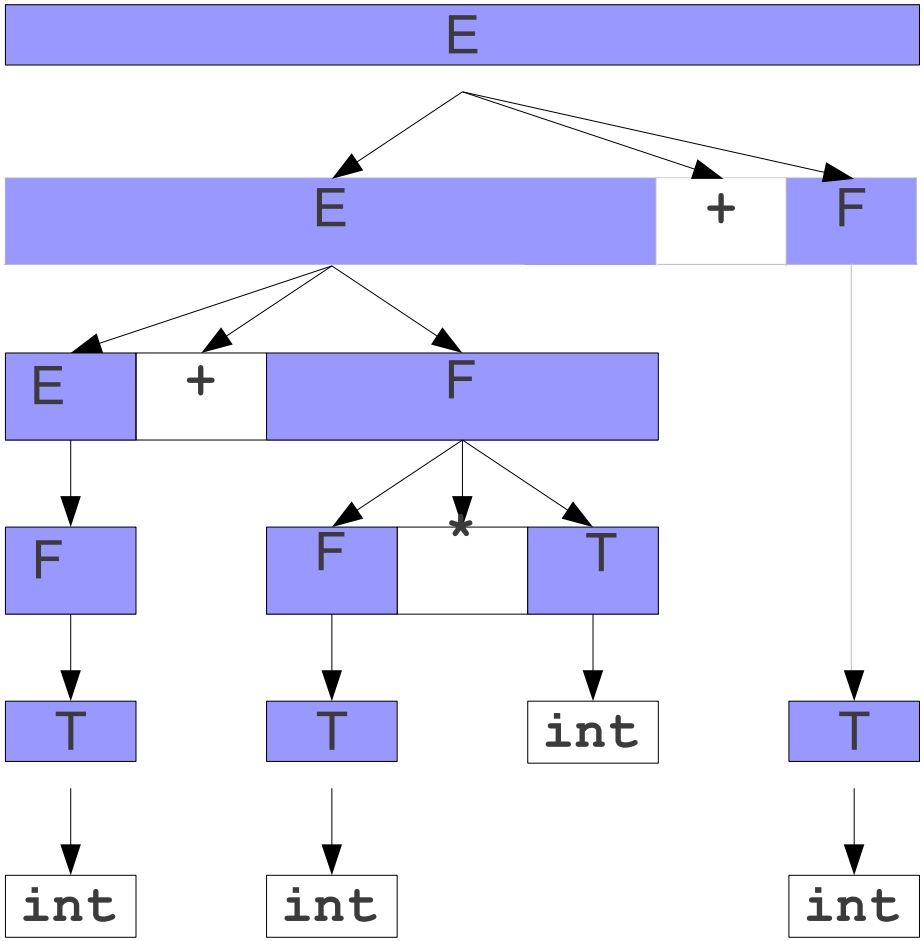
E → F
E → E + F
F → F * T
F → T
T → int
T → (E)



int	+	int	*	int	+	int

Another Look at Handles

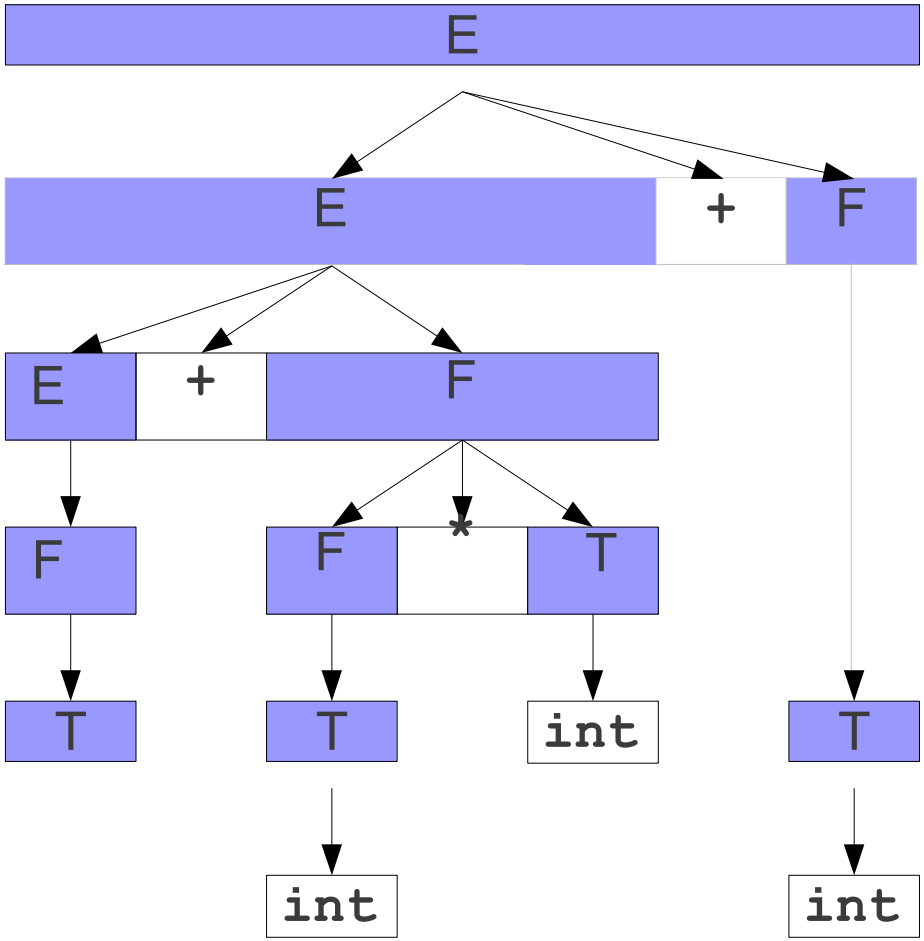
E → F
E → E + F
F → F * T
F → T
T → int
T → (E)



int	+	int	*	int	+	int

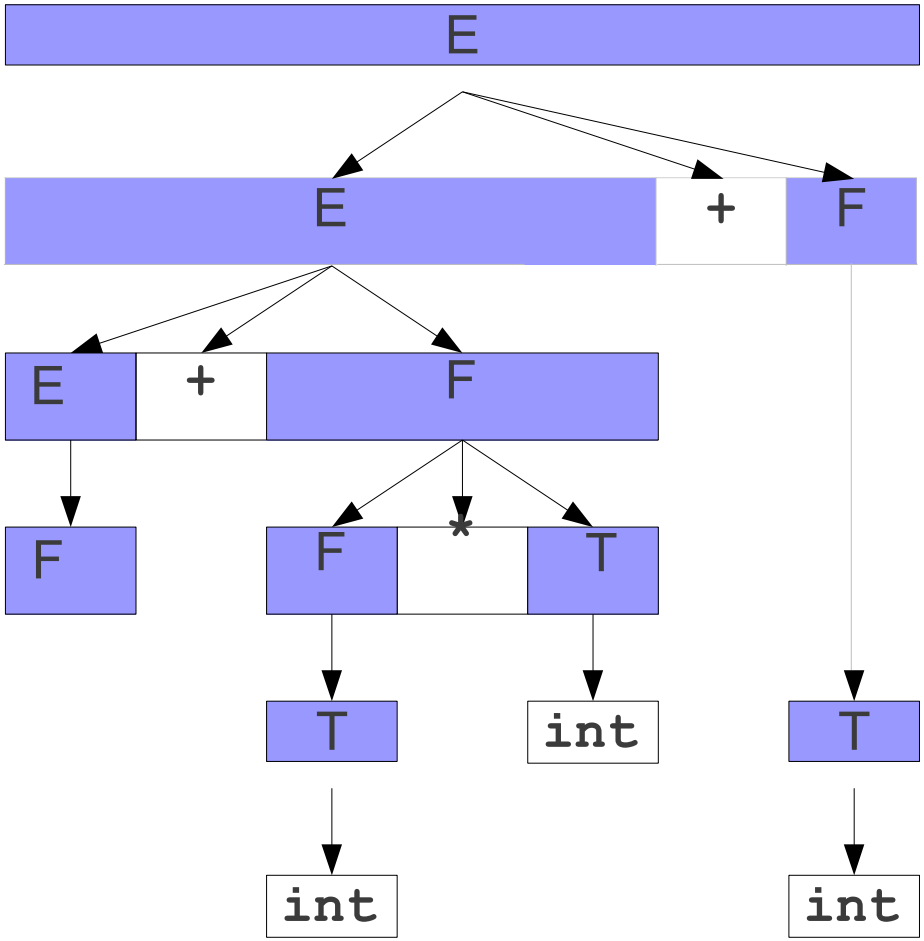
Another Look at Handles

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



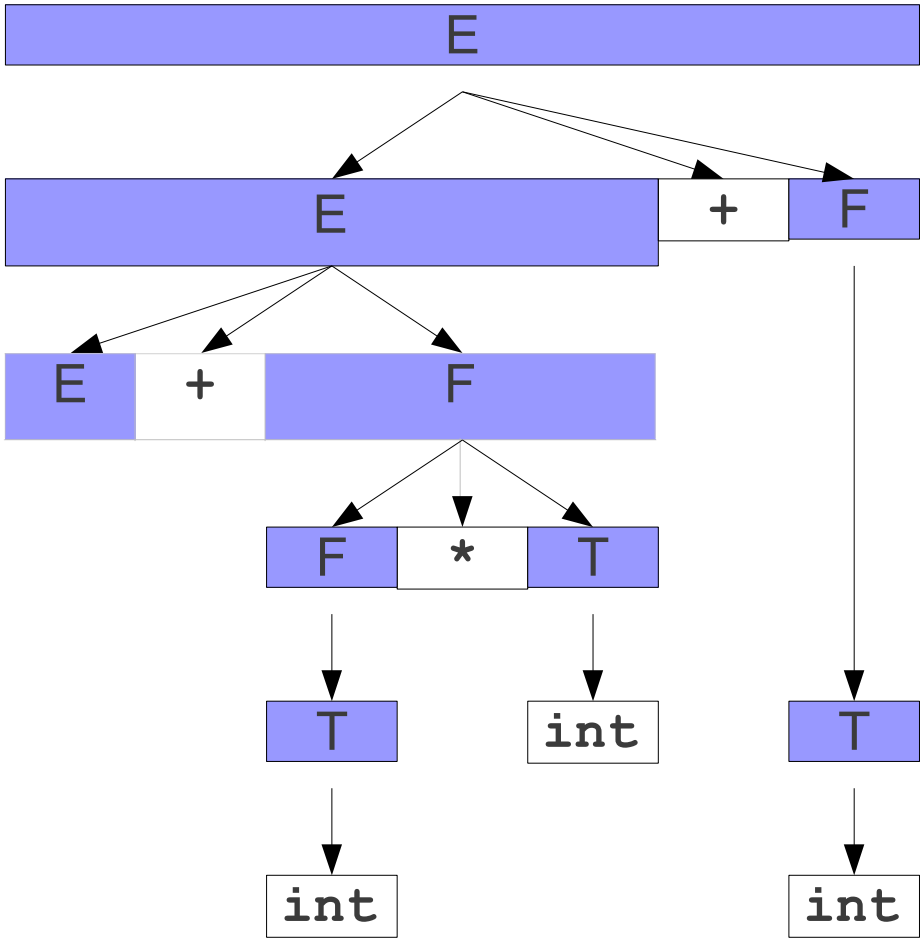
Another Look at Handles

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



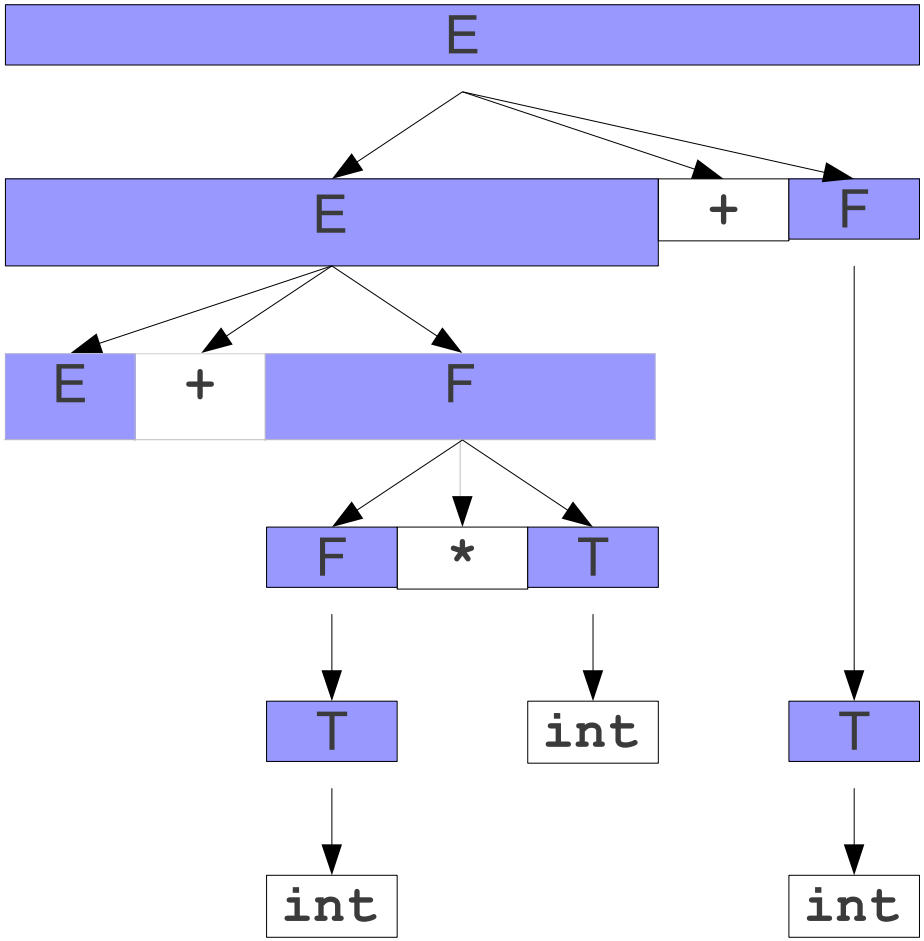
Another Look at Handles

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



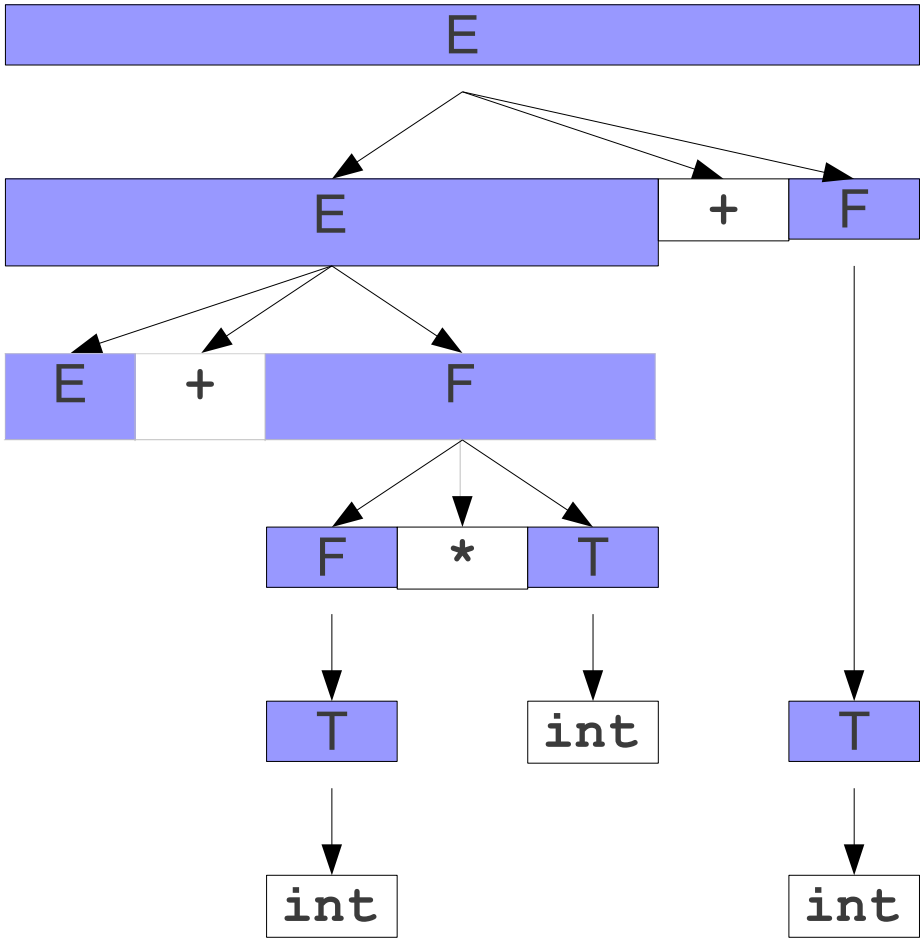
Another Look at Handles

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



Another Look at Handles

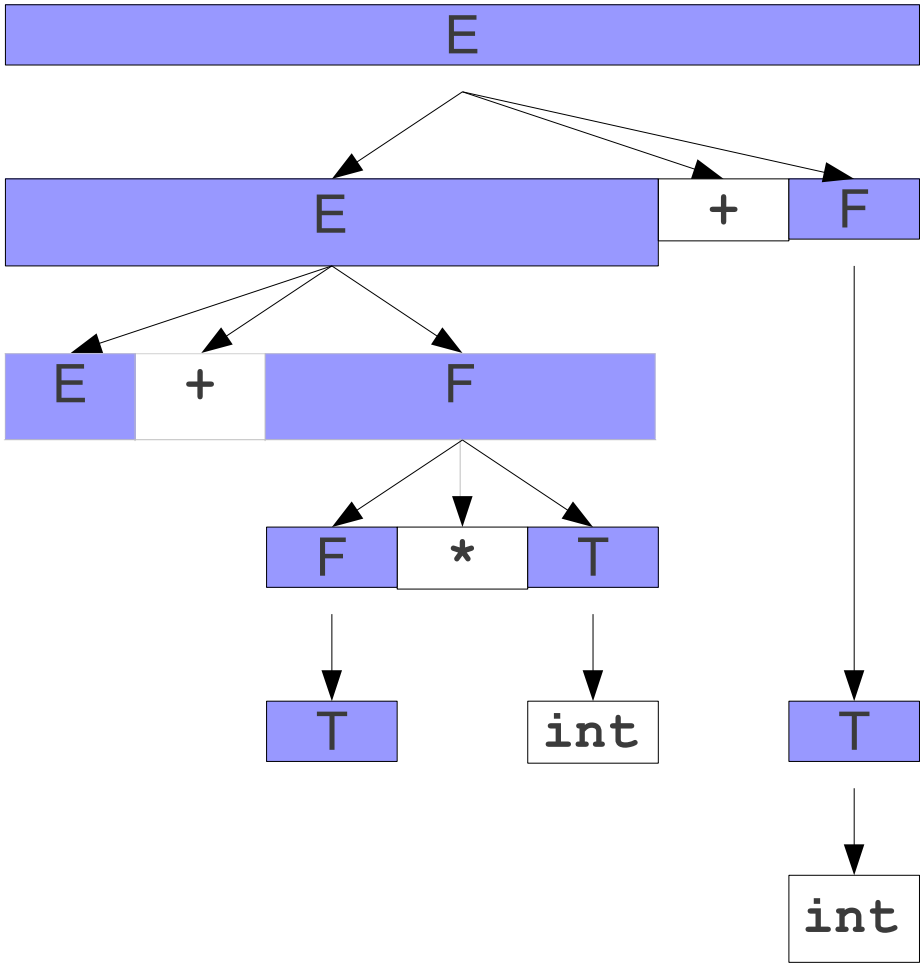
$E \rightarrow F$
$E \rightarrow E + F$
$F \rightarrow F * T$
$F \rightarrow T$
$T \rightarrow \texttt{int}$
$T \rightarrow (E)$



E	+	int	*	int	+	int

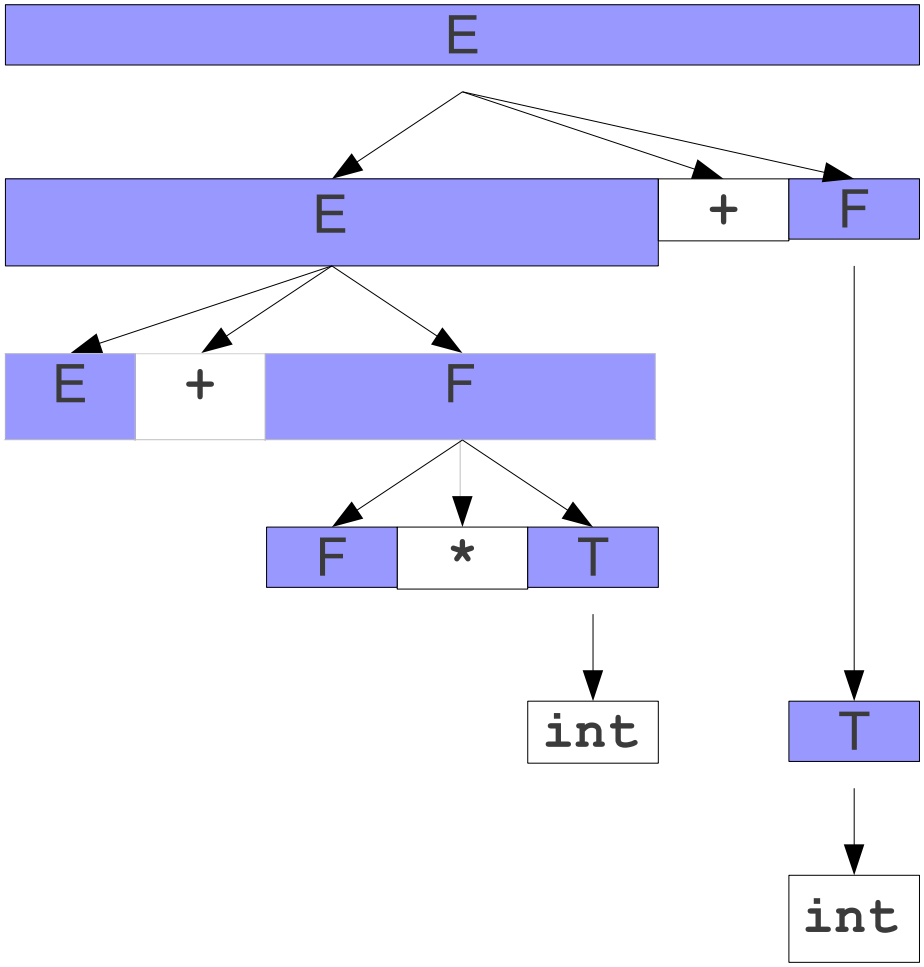
Another Look at Handles

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



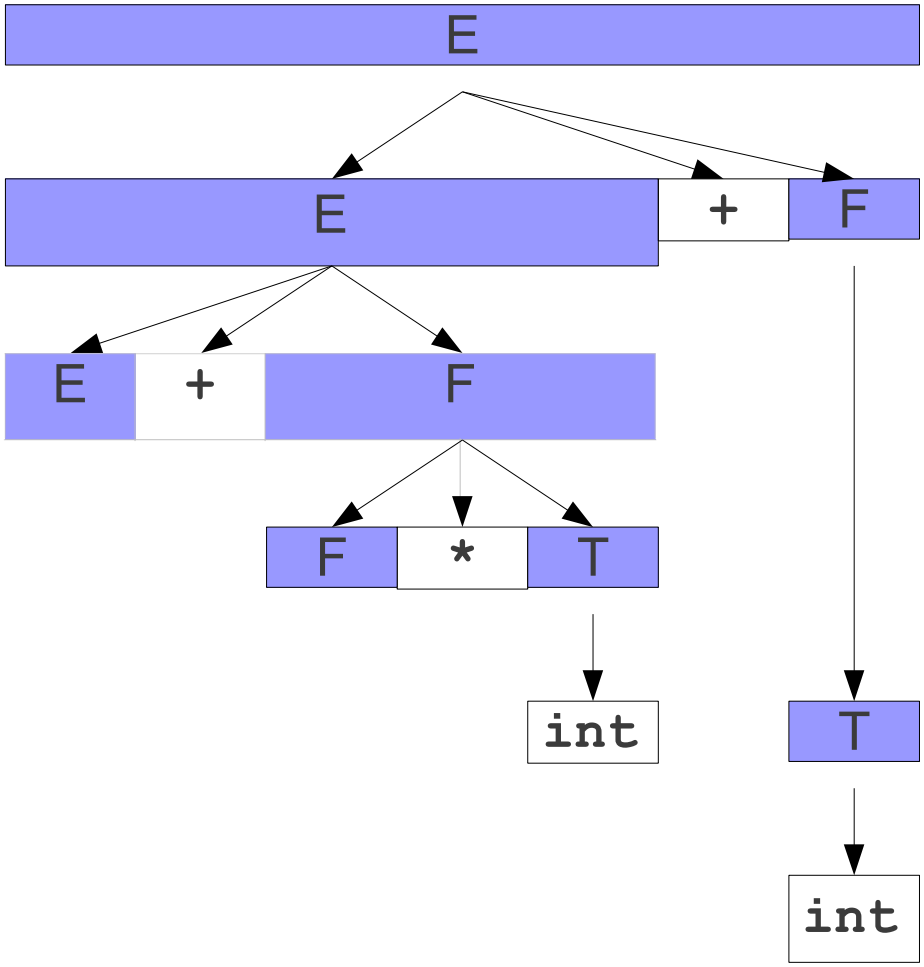
Another Look at Handles

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



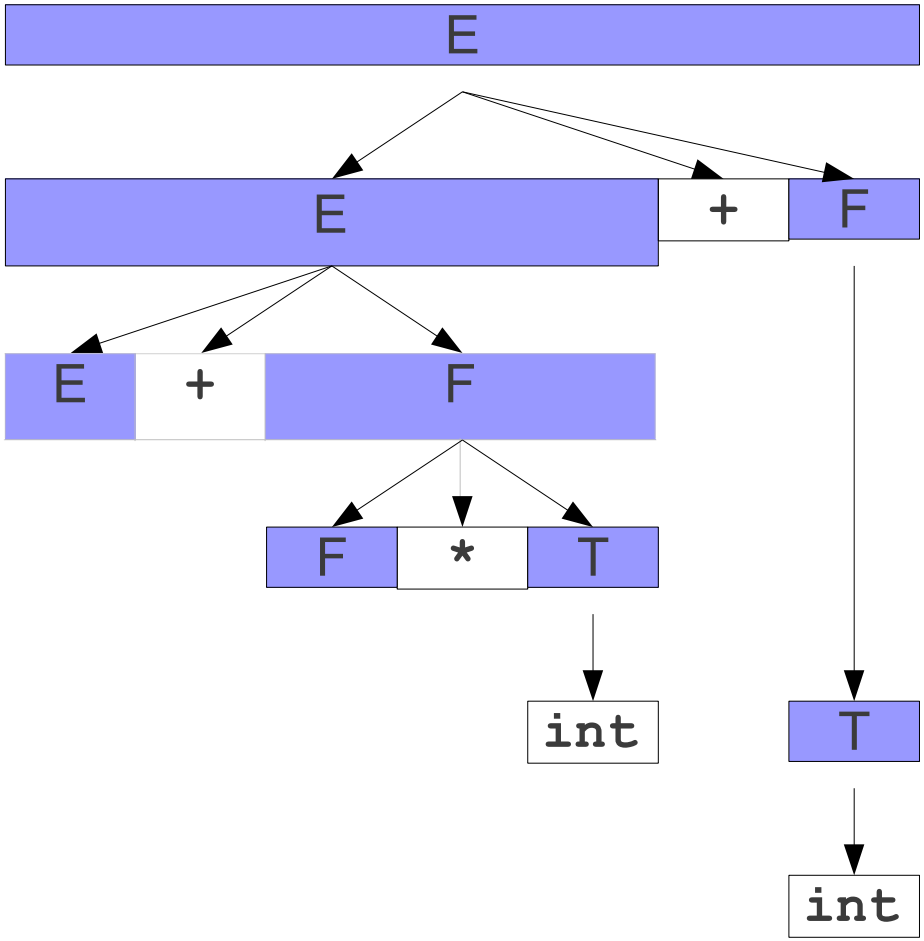
Another Look at Handles

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



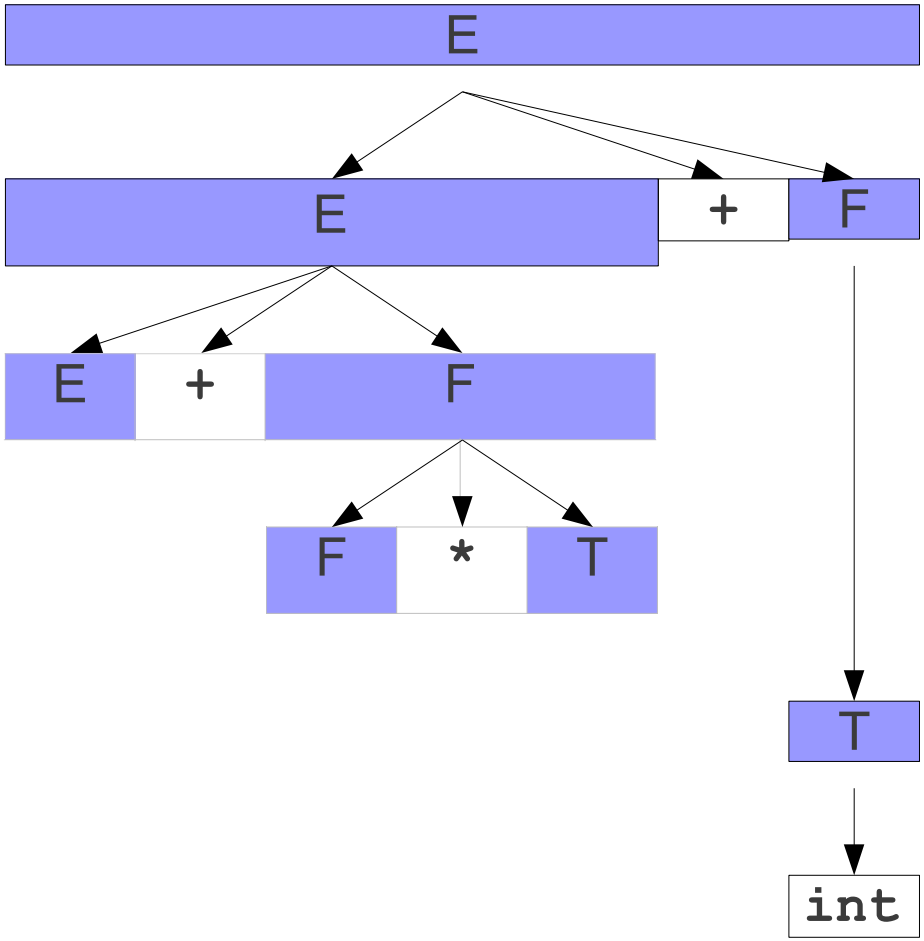
Another Look at Handles

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



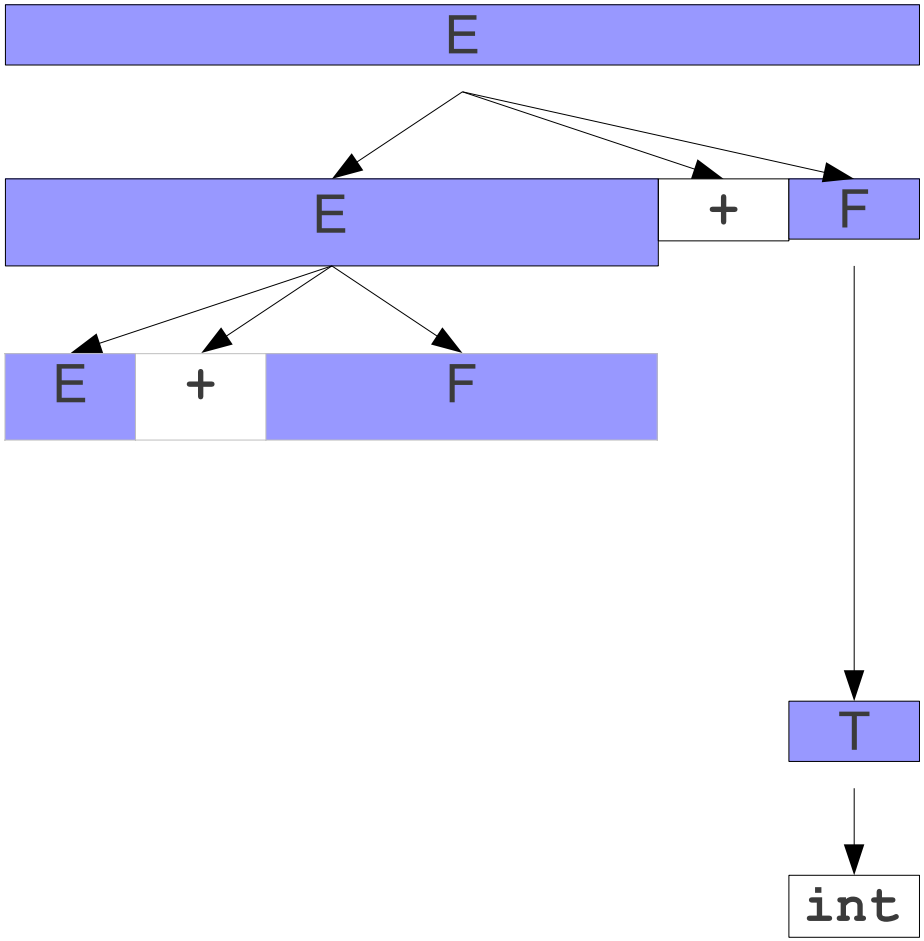
Another Look at Handles

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

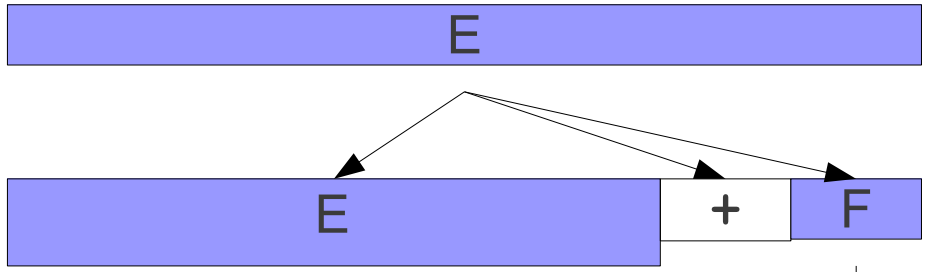


Another Look at Handles

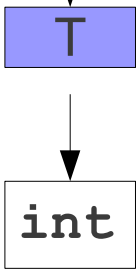
E → F
E → E + F
F → F * T
F → T
T → int
T → (E)

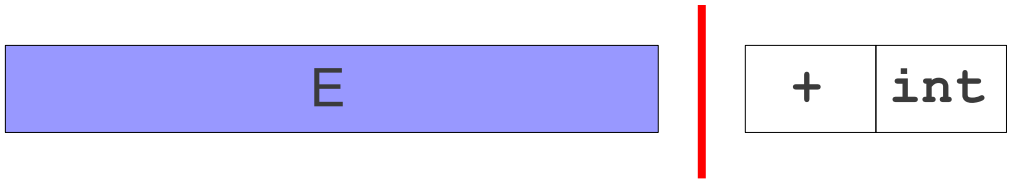


Another Look at Handles

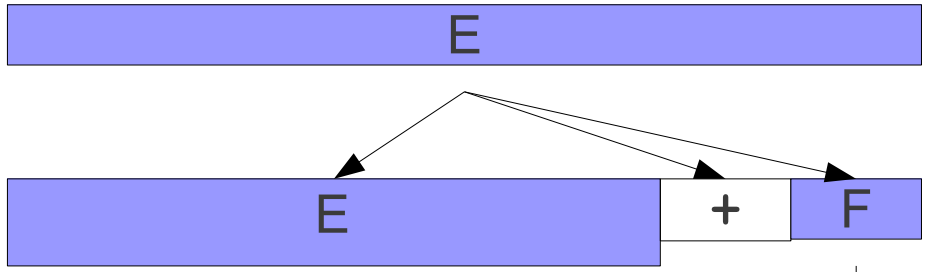


E → F
E → E + F
F → F * T
F → T
T → int
T → (E)

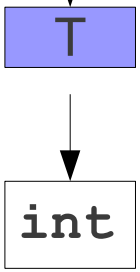


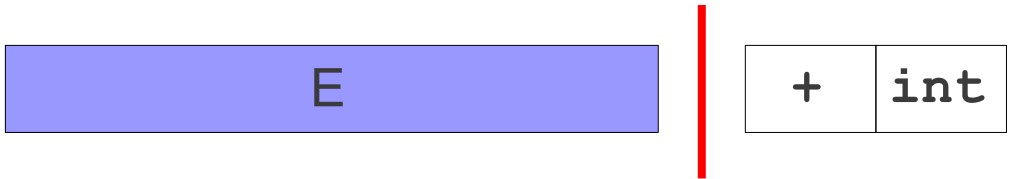


Another Look at Handles

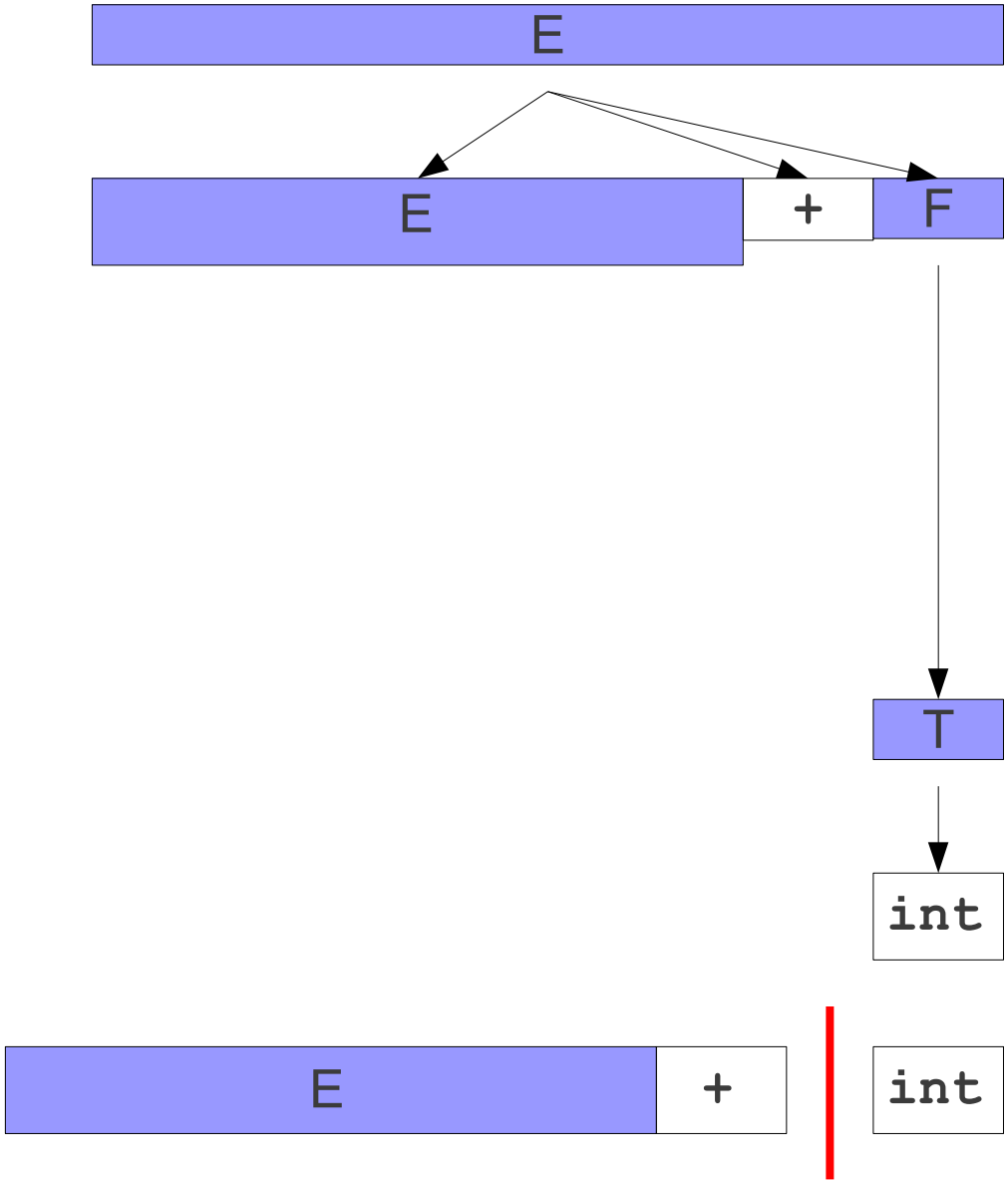


E → F
E → E + F
F → F * T
F → T
T → int
T → (E)





Another Look at Handles



$E \rightarrow F$

$E \rightarrow E + F$

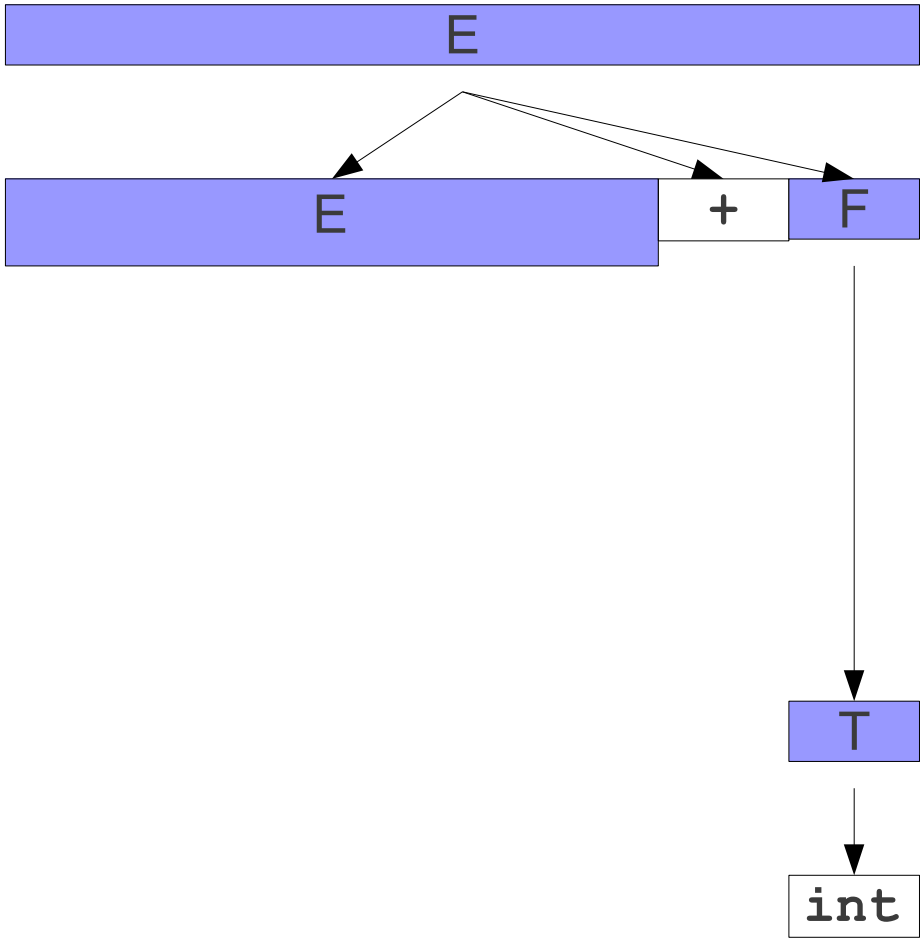
$F \rightarrow F * T$

$F \rightarrow T$

$T \rightarrow \texttt{int}$

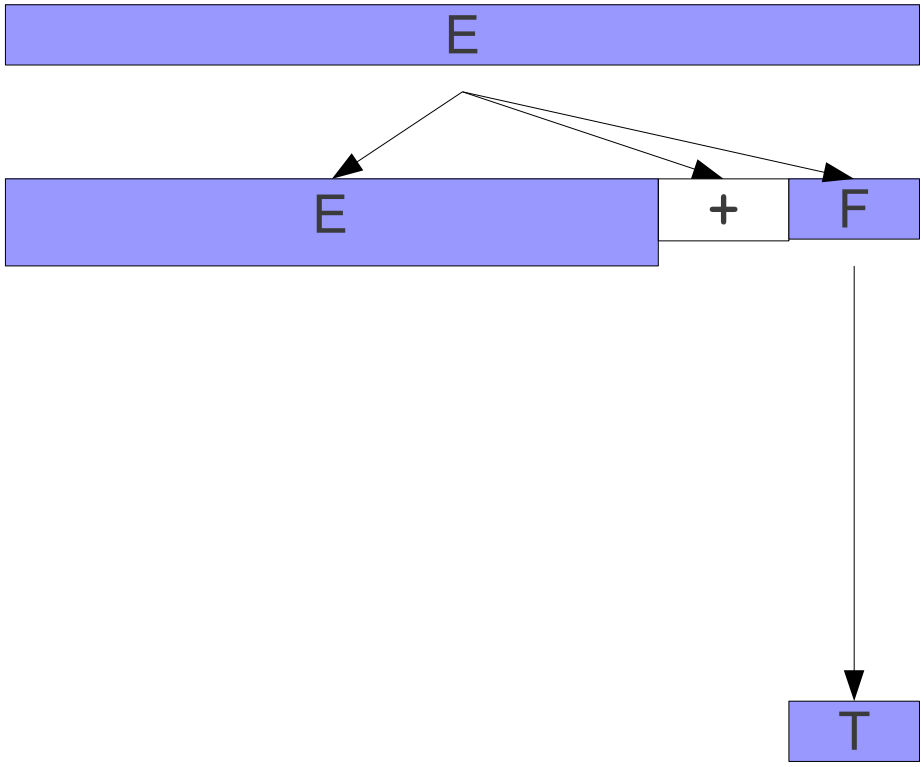
$T \rightarrow (E)$

Another Look at Handles



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

Another Look at Handles



$E \rightarrow F$

$E \rightarrow E + F$

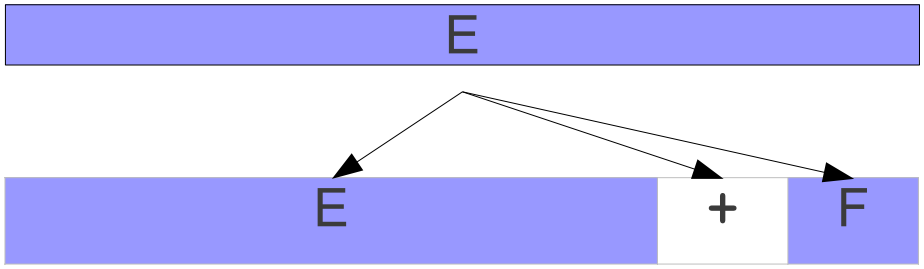
$F \rightarrow F * T$

$F \rightarrow T$

$T \rightarrow \texttt{int}$

$T \rightarrow (E)$

Another Look at Handles



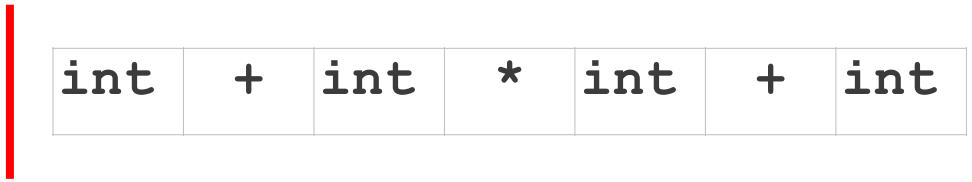
E → F
E → E + F
F → F * T
F → T
T → int
T → (E)

Another Look at Handles

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

Tracking Our Position

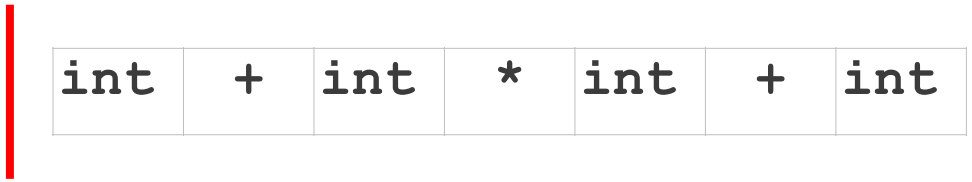
E → F
E → E + F
F → F * T
F → T
T → int
T → (E)



Tracking Our Position

(A bidirectional look!)

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



Tracking Our Position

(A bidirectional look!)

S → · E

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

int	+	int	*	int	+	int

Tracking Our Position

(A bidirectional look!)

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

S → · E
E → · E + F

int	+	int	*	int	+	int

Tracking Our Position

(A bidirectional look!)

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

S → · E
E → · E + F
E → · E + F

int	+	int	*	int	+	int

Tracking Our Position

(A bidirectional look!)

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	+	int	*	int	+	int

Tracking Our Position

(A bidirectional look!)

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

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

int	+	int	*	int	+	int

Tracking Our Position

(A bidirectional look!)

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

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

int	+	int	*	int	+	int

Tracking Our Position

(A bidirectional look!)

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

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

int	+	int	*	int	+	int

Tracking Our Position

(A bidirectional look!)

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

Tracking Our Position

(A bidirectional look!)

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 ·

Tracking Our Position

(A bidirectional look!)

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

Tracking Our Position

(A bidirectional look!)

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 ·

Tracking Our Position

(A bidirectional look!)

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

S → · E
E → · E + F
E → · E + F

Tracking Our Position

(A bidirectional look!)

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

S → · E
E → · E + F
E → E · + F

Tracking Our Position

(A bidirectional look!)

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

S → · E
E → · E + F
E → E + · F

Tracking Our Position

(A bidirectional look!)

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

Tracking Our Position

(A bidirectional look!)

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

Tracking Our Position

(A bidirectional look!)

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

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

Tracking Our Position

(A bidirectional look!)

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

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

E	+	int

*	int	+	int

Tracking Our Position

(A bidirectional look!)

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

(A bidirectional look!)

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

(A bidirectional look!)

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

(A bidirectional look!)

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

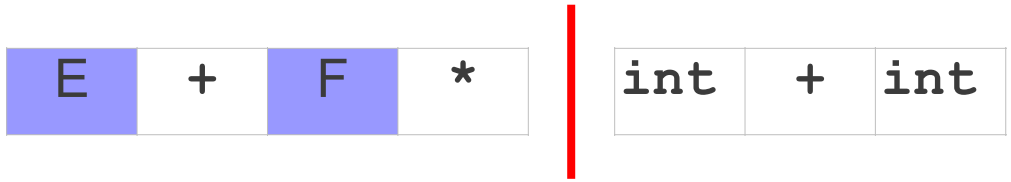
*	int	+	int

Tracking Our Position

(A bidirectional look!)

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



Tracking Our Position

(A bidirectional look!)

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

(A bidirectional look!)

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 ·

Tracking Our Position

(A bidirectional look!)

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

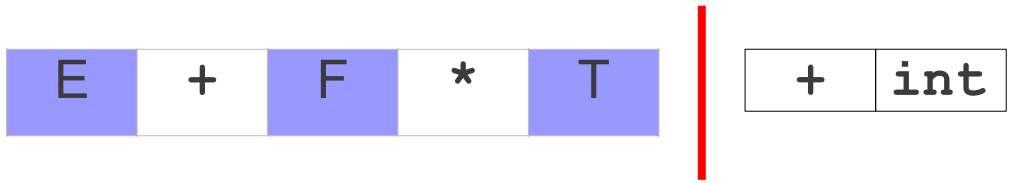
S → · E
E → · E + F
E → E + · F
F → F * · T

Tracking Our Position

(A bidirectional look!)

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 ·



Tracking Our Position

(A bidirectional look!)

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

S → · E
E → · E + F
E → E + · F

Tracking Our Position

(A bidirectional look!)

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

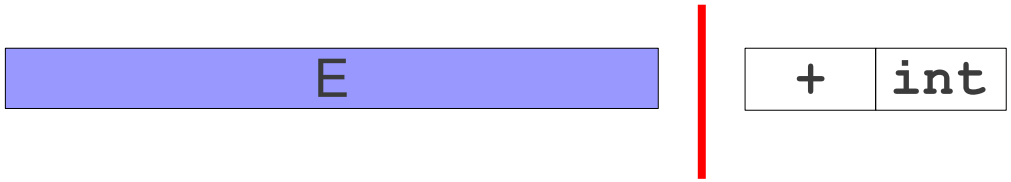
S → · E
E → · E + F
E → E + F ·

Tracking Our Position

(A bidirectional look!)

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

S → · E
E → · E + F

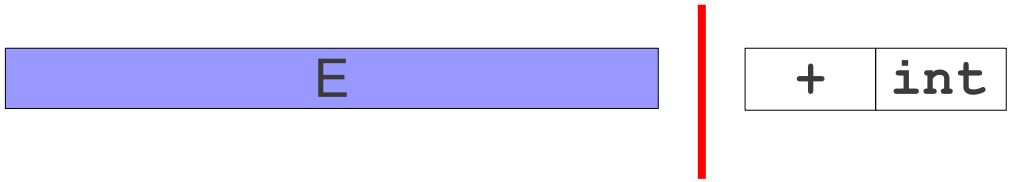


Tracking Our Position

(A bidirectional look!)

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

S → · E
E → E · + F



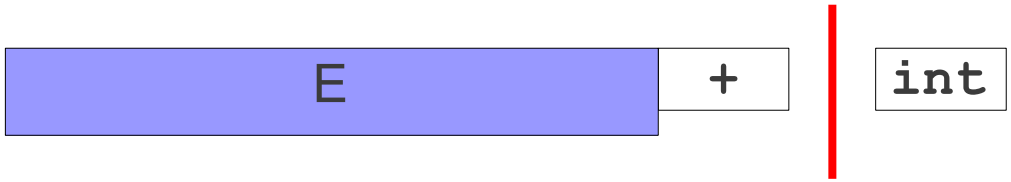
Tracking Our Position

(A bidirectional look!)

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

S → · E

E → E + · F

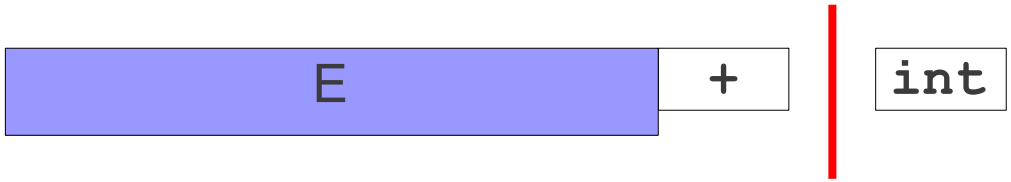


Tracking Our Position

(A bidirectional look!)

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

S → · E
E → E + · F
F → · T

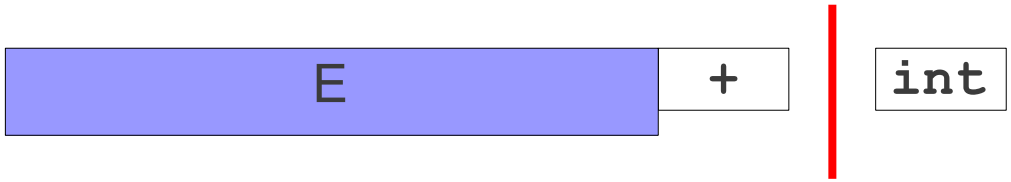


Tracking Our Position

(A bidirectional look!)

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



Tracking Our Position

(A bidirectional look!)

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** ·

Tracking Our Position

(A bidirectional look!)

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

(A bidirectional look!)

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

(A bidirectional look!)

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

S → · E
E → E + · F

Tracking Our Position

(A bidirectional look!)

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

S → · E
E → E + F ·

Tracking Our Position

(A bidirectional look!)

S → · E

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

E

Tracking Our Position

(A bidirectional look!)

S → E ·

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

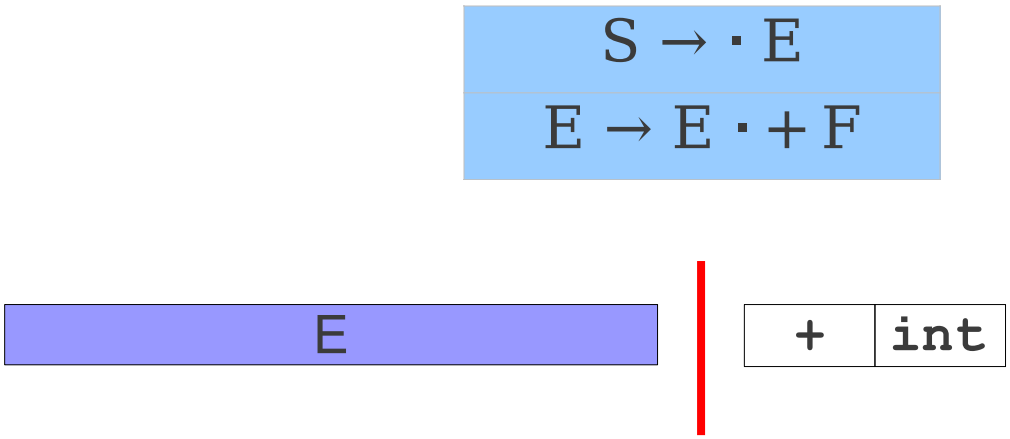
E

Generating Left-Hand Sides

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

Recognizing Left-Hand Sides

- Idea: At each point, track
  - Which production we are in, and
  - Where we are in that production.



Recognizing Left-Hand Sides

- Idea: At each point, track
  - Which production we are in, and
  - Where we are in that production.
- At each point, we can do one of two things:
  - **Match** the next symbol
  - (Just for now) **Guess** which production used.
  - (More precisely the production chooses non-deterministically)

Recognizing Left-Hand Sides

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

Recognizing Left-Hand Sides

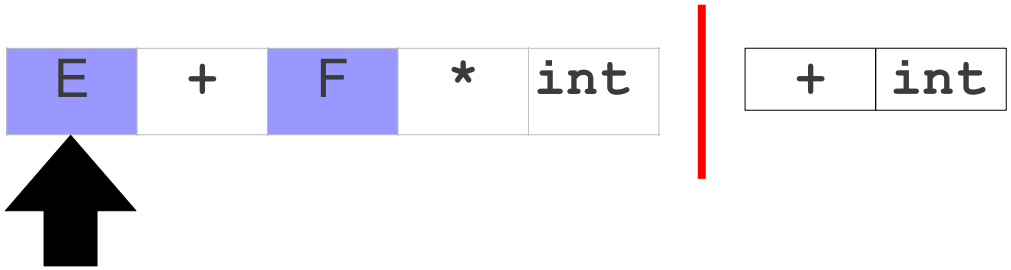
$S \rightarrow \cdot E$

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

Recognizing Left-Hand Sides

$S \rightarrow \cdot E$

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



Recognizing Left-Hand Sides

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

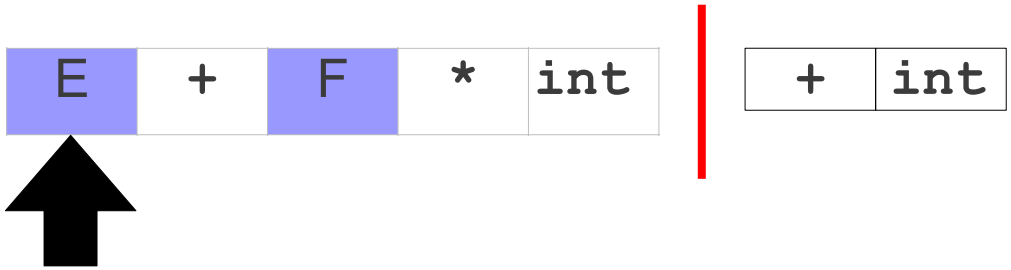
S → · E

E → · E + F

Recognizing Left-Hand Sides

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

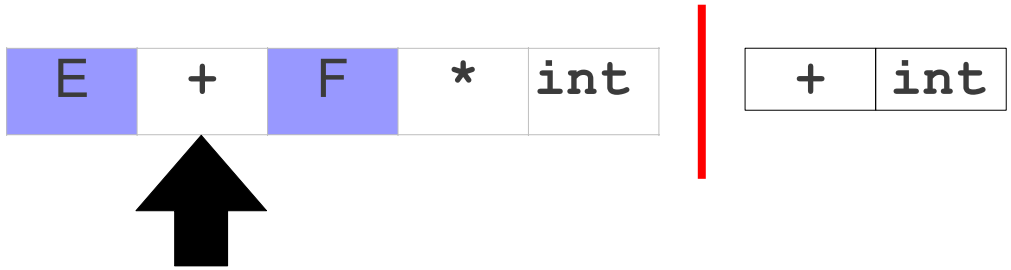
S → · E
E → · E + F
E → · E + F



Recognizing Left-Hand Sides

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

S → · E
E → · E + F
E → E · + F



Recognizing Left-Hand Sides

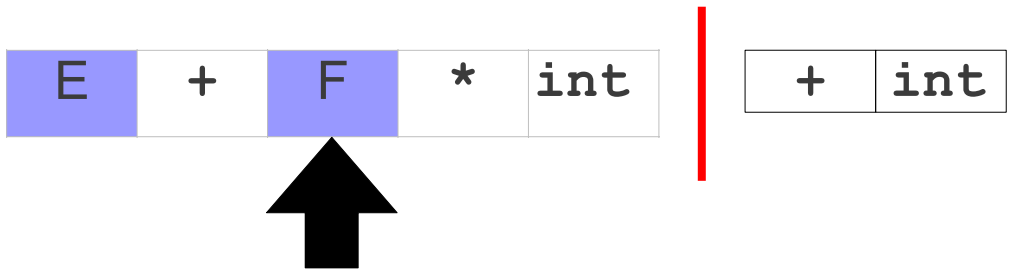
S → E
E → F
E → E + F
F → F * T
F → T
T → int
T → (E)

S → · E
E → · E + F
E → E + · F

Recognizing Left-Hand Sides

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

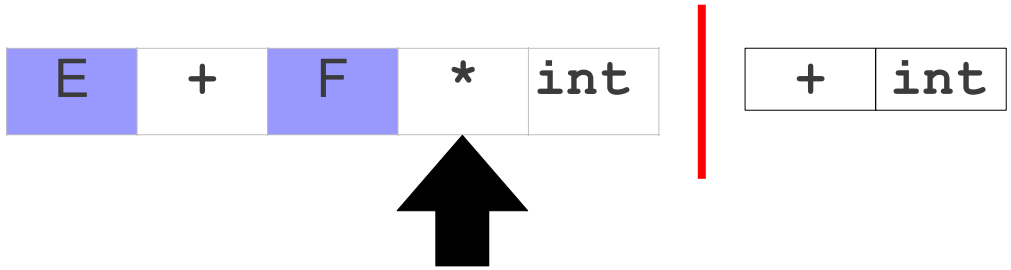
S → · E
E → · E + F
E → E + · F
F → · F * T



Recognizing Left-Hand Sides

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

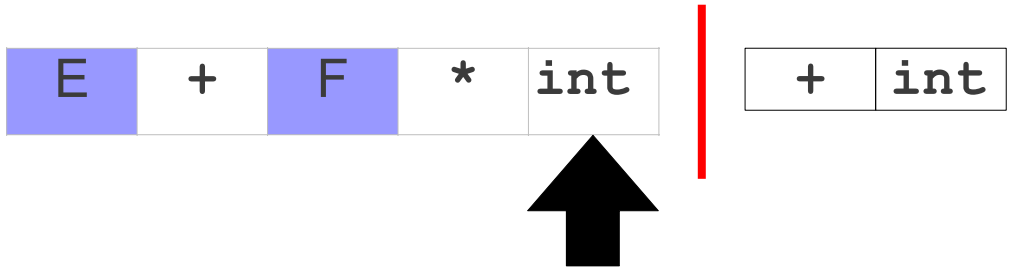
S → · E
E → · E + F
E → E + · F
F → F · * T



Recognizing Left-Hand Sides

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

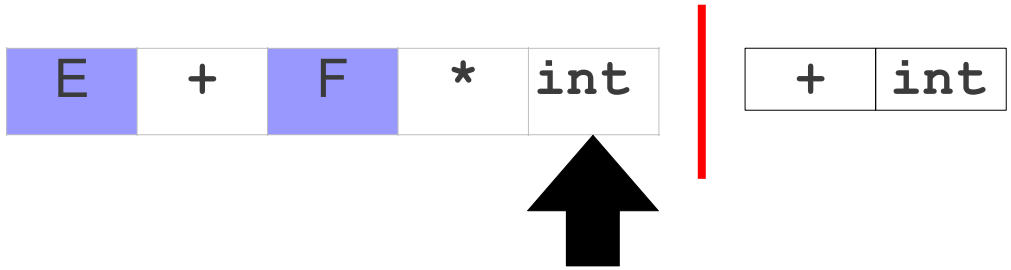
S → · E
E → · E + F
E → E + · F
F → F * · T



Recognizing Left-Hand Sides

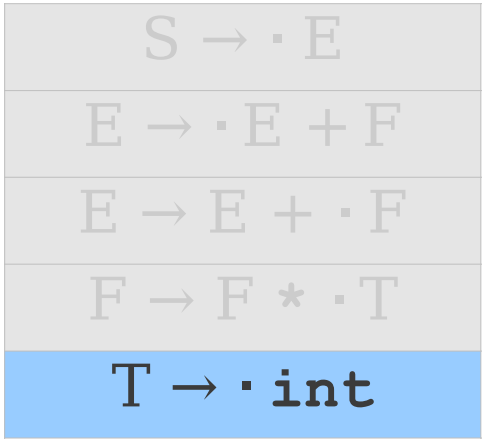
S → E
E → F
E → E + F
F → F * T
F → T
T → int
T → (E)

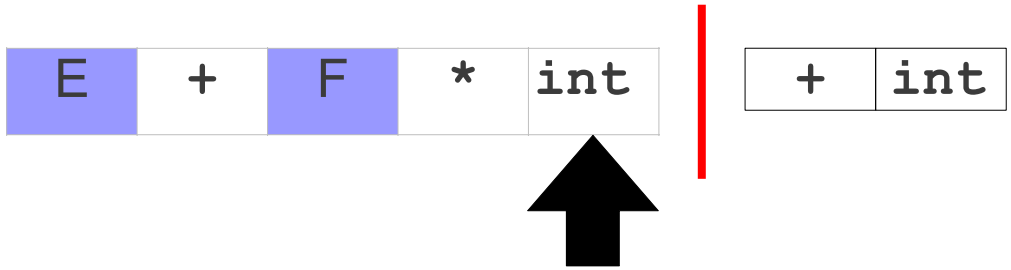
S → · E
E → · E + F
E → E + · F
F → F * · T



Recognizing Left-Hand Sides

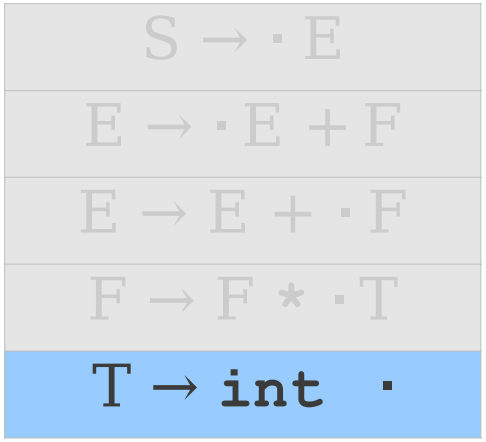
S → E
E → F
E → E + F
F → F * T
F → T
T → int
T → (E)





Recognizing Left-Hand Sides

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



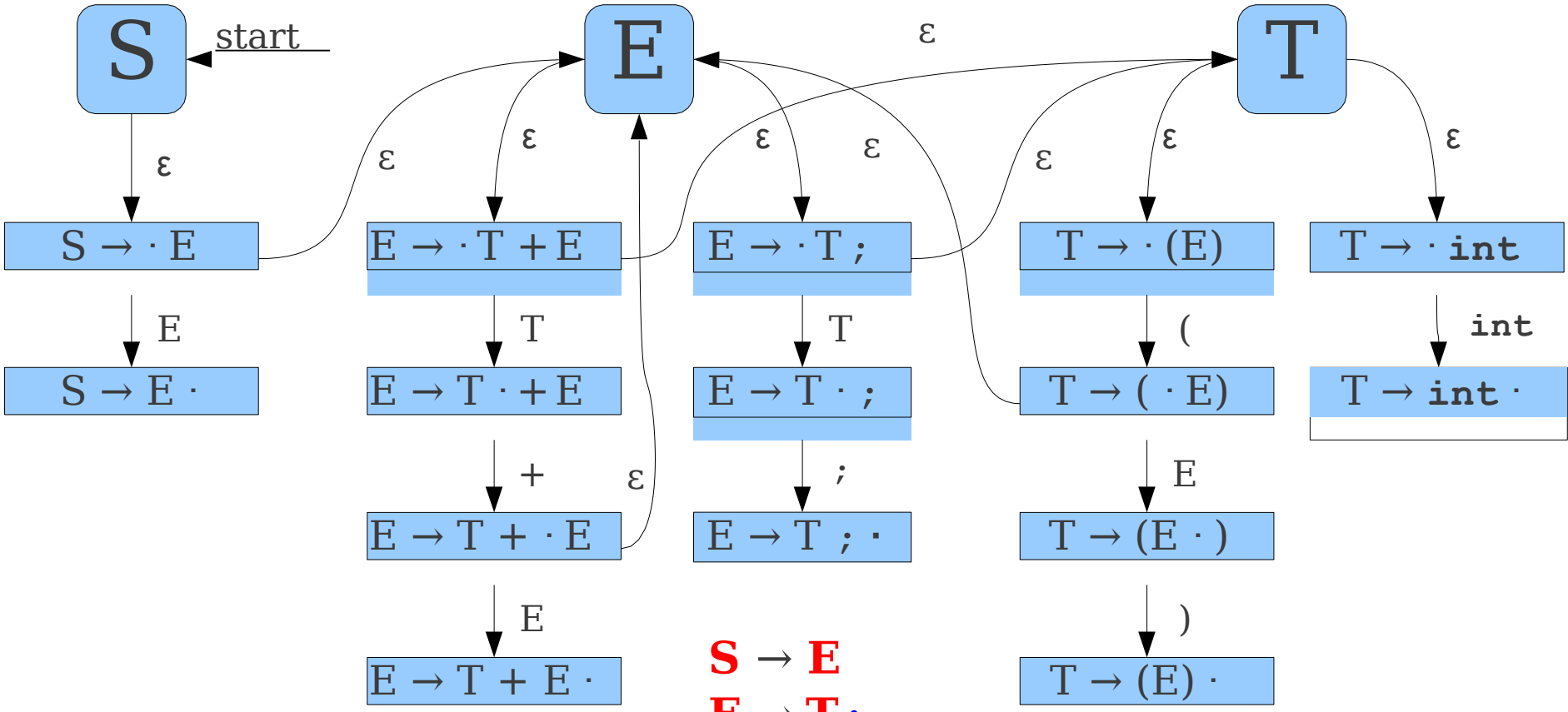
An Important Result

- At any point in time, we only need to track where we are in **one production** (In one computation path)**.**
- (We’ll correct it later!)
- We can use a finite automaton as our recognizer.

Constructing the Automaton

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

An Automaton for Left Areas



Why This Matters

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

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

- Then we might be looking at a handle.

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.

Constructing the Automaton II

- Begin in a state containing **S** → · **A**, where **S** is the augmented start symbol.

Constructing the Automaton II

- Begin in a state containing **S** → · **A**, where **S** is the augmented start symbol.
- Compute the **closure** of the state:
  - If **A** → α · **B**ω is in the state, add **B** → · γ to the state for each production **B** → γ.
  - Yet another fixed-point iteration!

Constructing the Automaton II

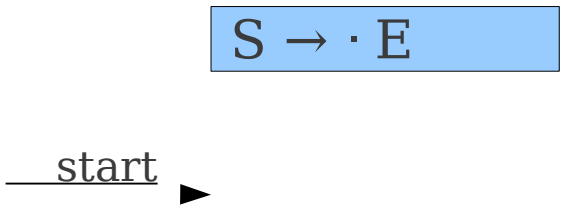
- Begin in a state containing **S** → · **A**, where **S** is the augmented start symbol.
- Compute the **closure** of the state:
  - If **A** → ***α*** · **B*ω*** is in the state, add **B** → · ***γ*** to the state for each production **B** → ***γ***.
  - Yet another fixed-point iteration!
- Repeat until no new states are added:
  - If a state contains a production **A** → ***α*** · ***x*ω** for symbol ***x***, add a transition on ***x*** from that state to the state containing the closure of **A** → ***α x*** · ***ω***

Constructing the Automaton II

- Begin in a state containing **S** → · **A**, where **S** is the augmented start symbol.
- Compute the **closure** of the state:
  - If **A** → ***α*** · **B*ω*** is in the state, add **B** → · ***γ*** to the state for each production **B** → ***γ***.
  - Yet another fixed-point iteration!
- Repeat until no new states are added:
  - If a state contains a production **A** → ***α*** · ***x*ω** for symbol ***x***, add a transition on ***x*** from that state to the state containing the closure of **A** → ***α x*** · ***ω***
- This is equivalent to a subset construction on the NFA.

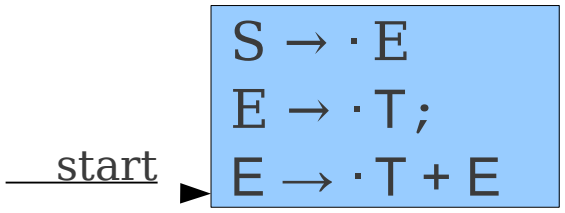
A Deterministic Automaton

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



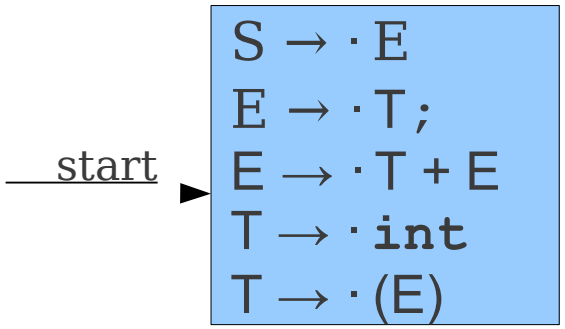
A Deterministic Automaton

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



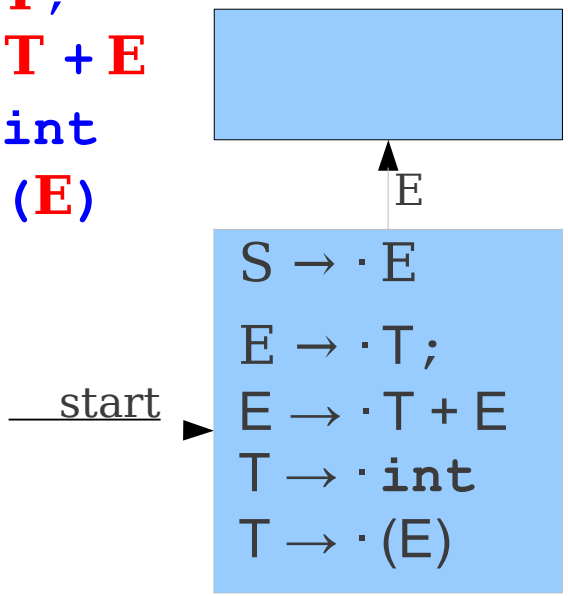
A Deterministic Automaton

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



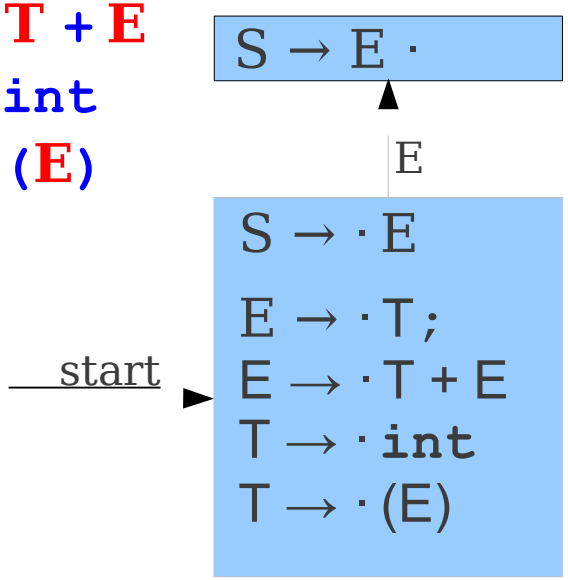
A Deterministic Automaton

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



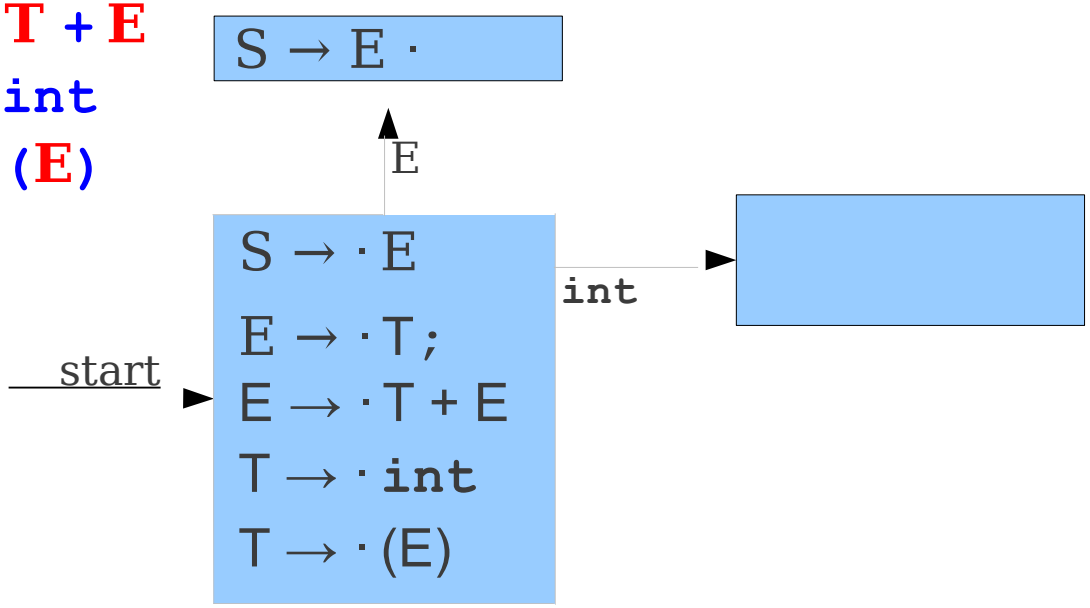
A Deterministic Automaton

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



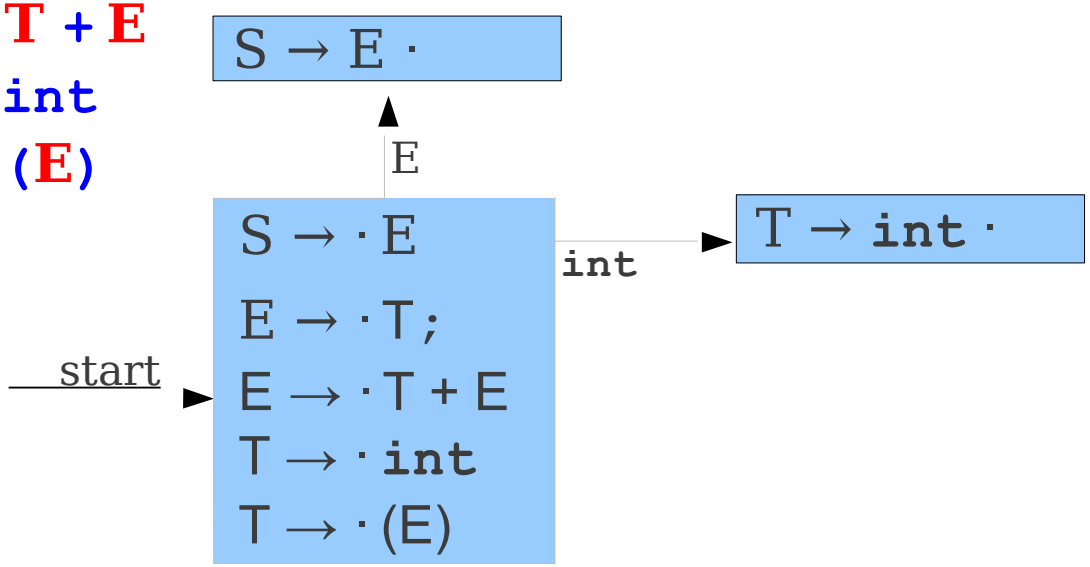
A Deterministic Automaton

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



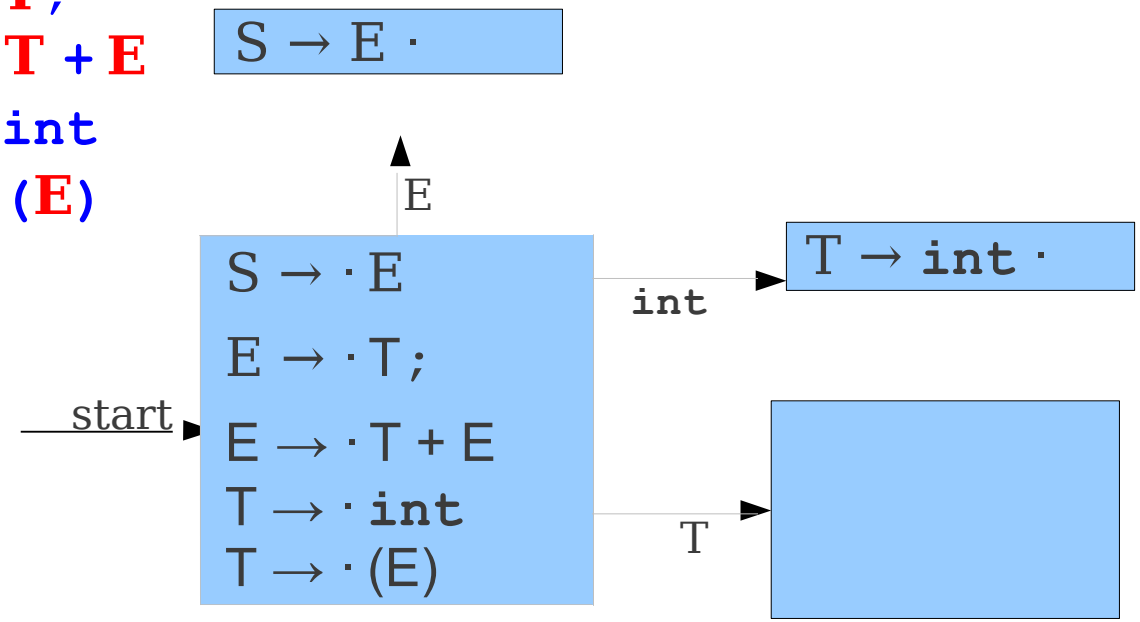
A Deterministic Automaton

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



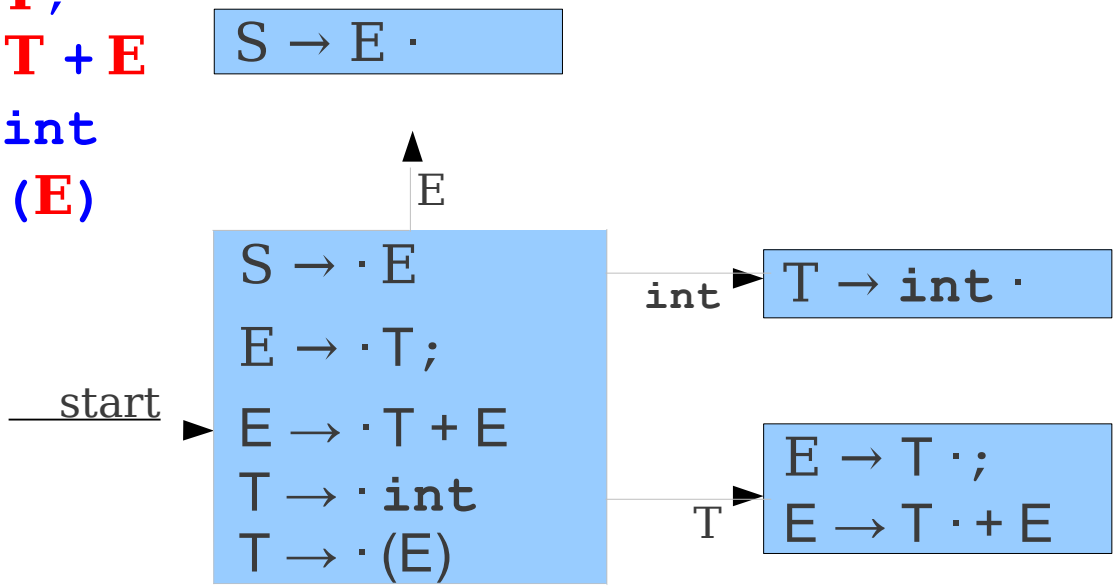
A Deterministic Automaton

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



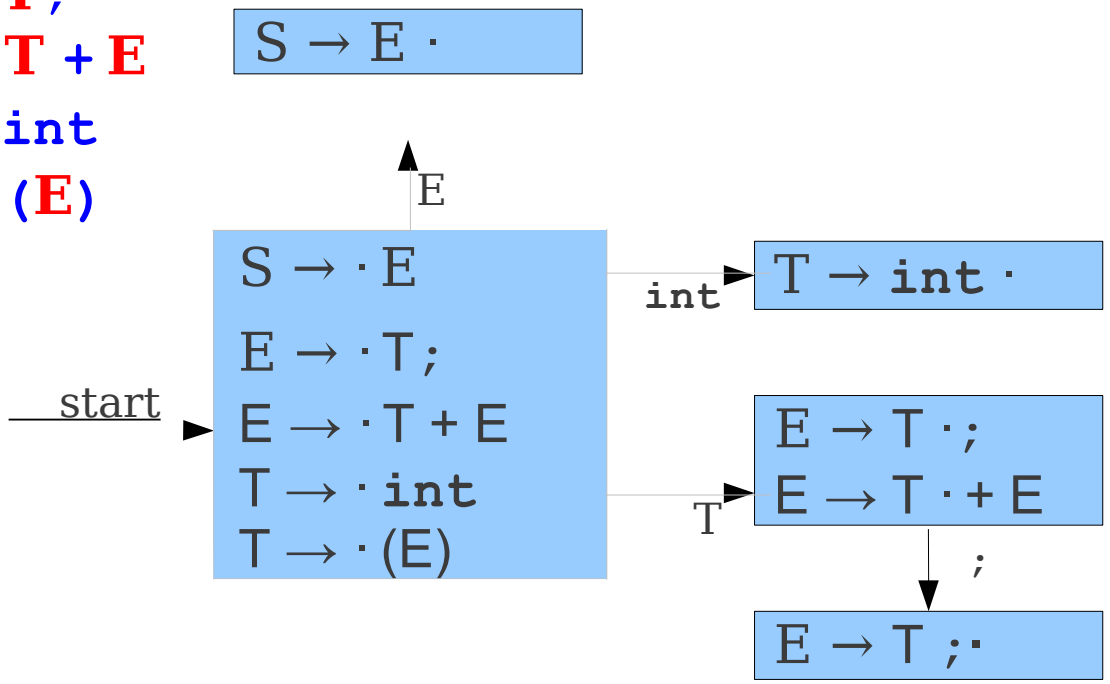
A Deterministic Automaton

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

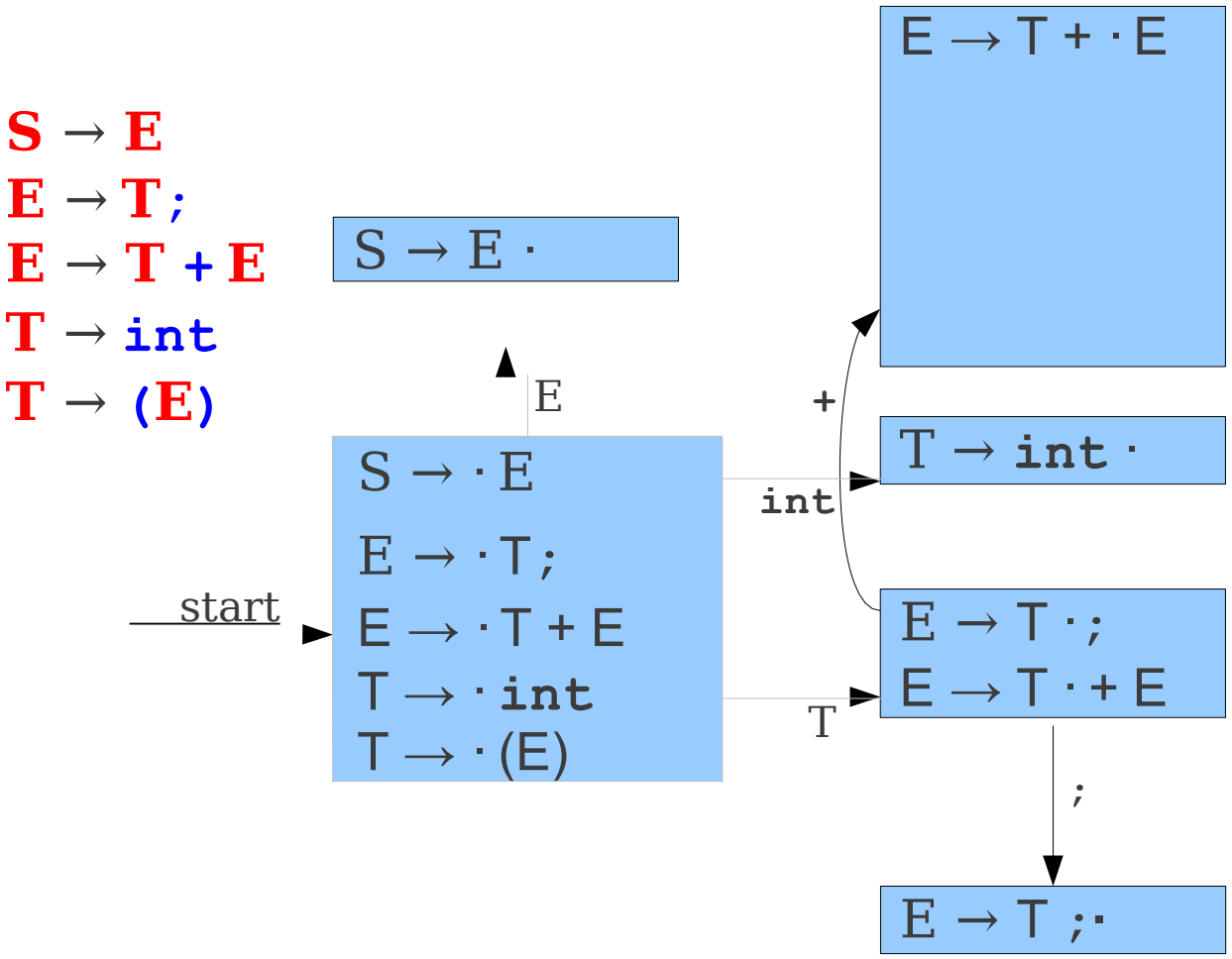


A Deterministic Automaton

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

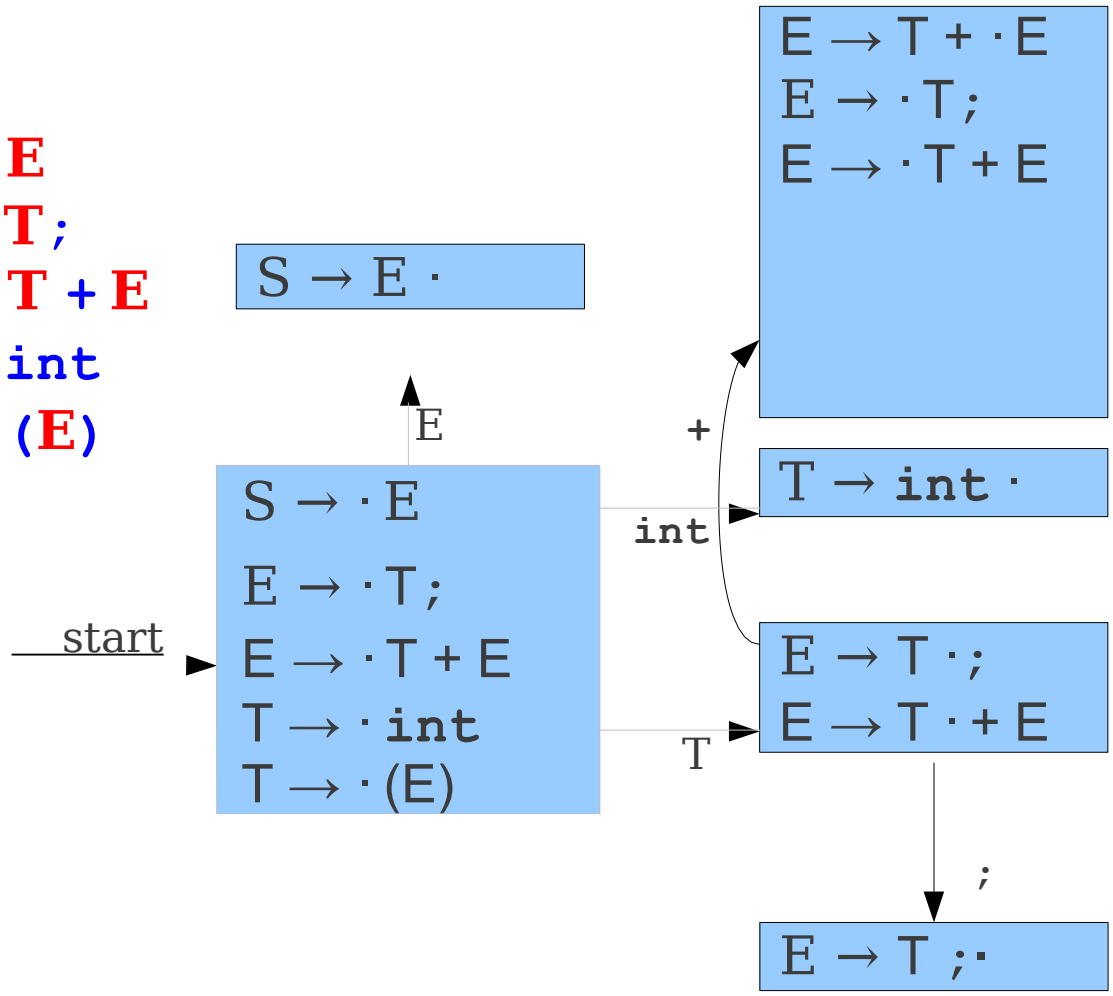


A Deterministic Automaton

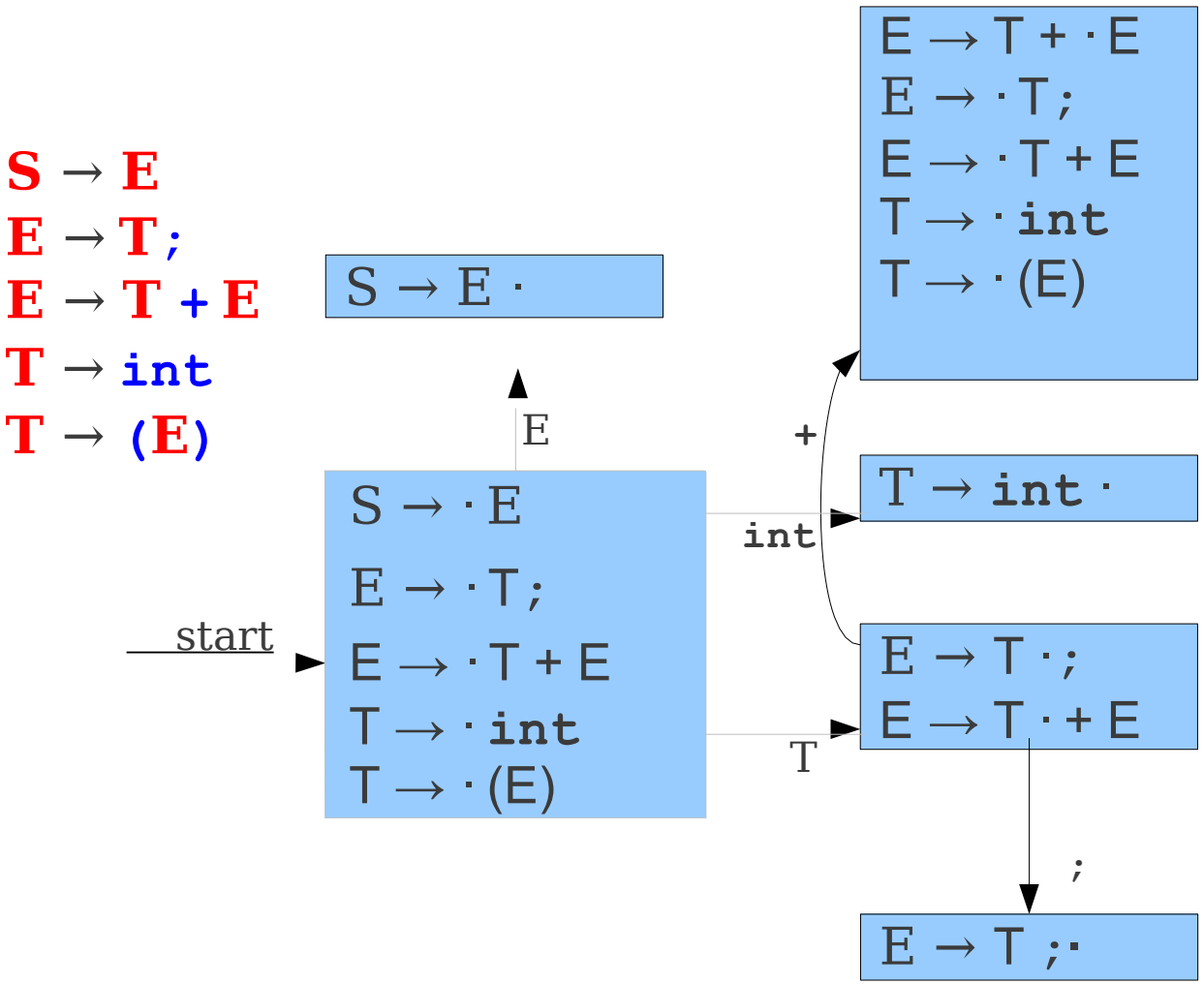


A Deterministic Automaton

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

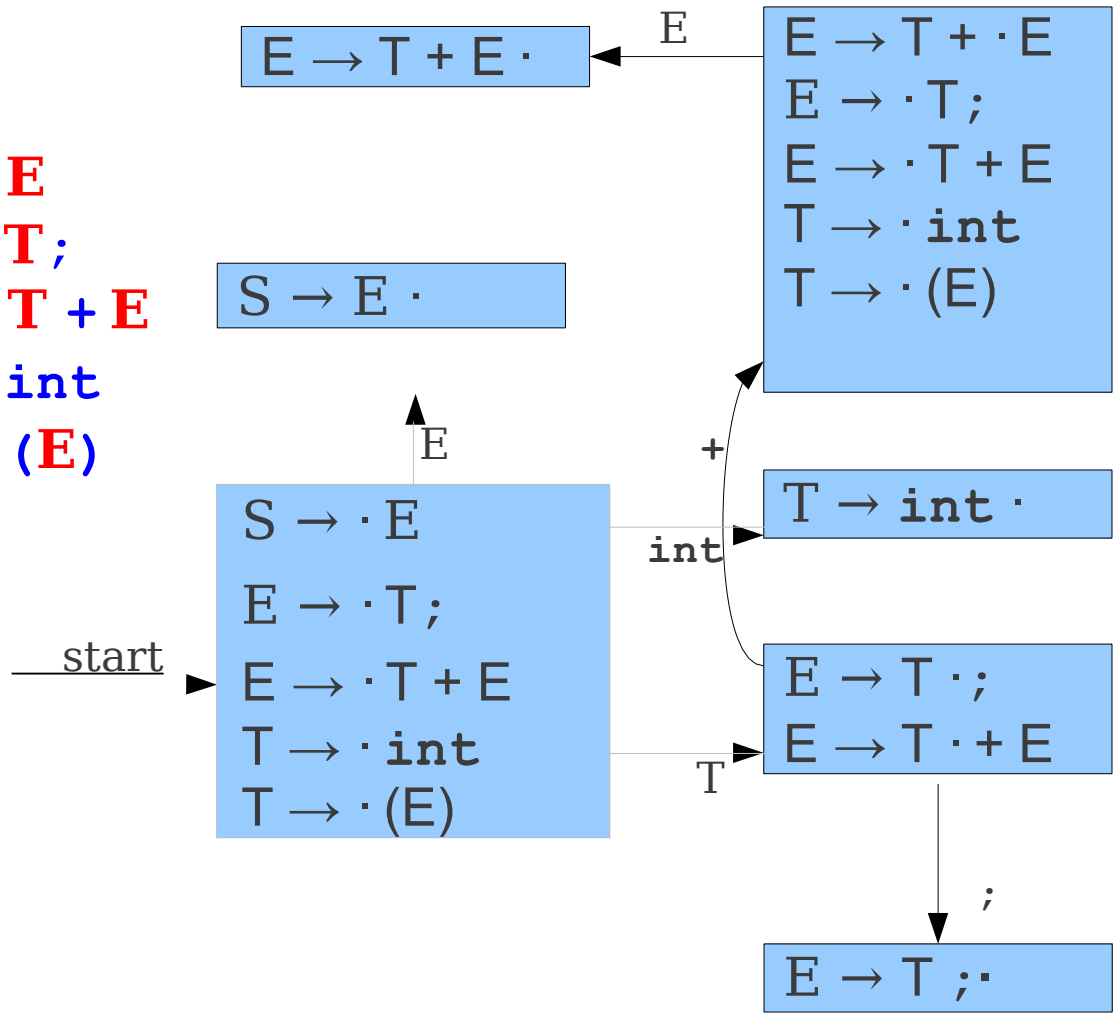


A Deterministic Automaton



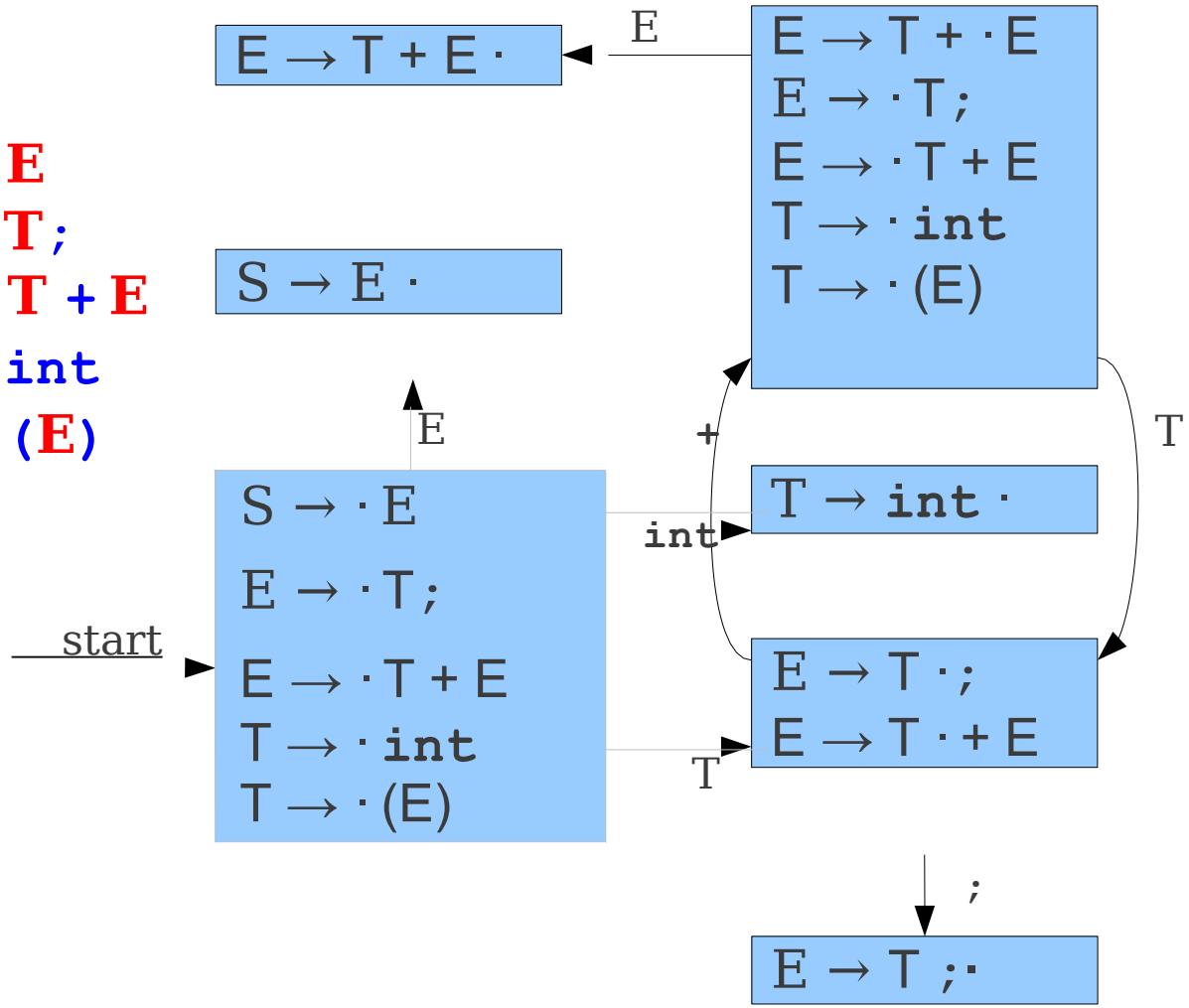
A Deterministic Automaton

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



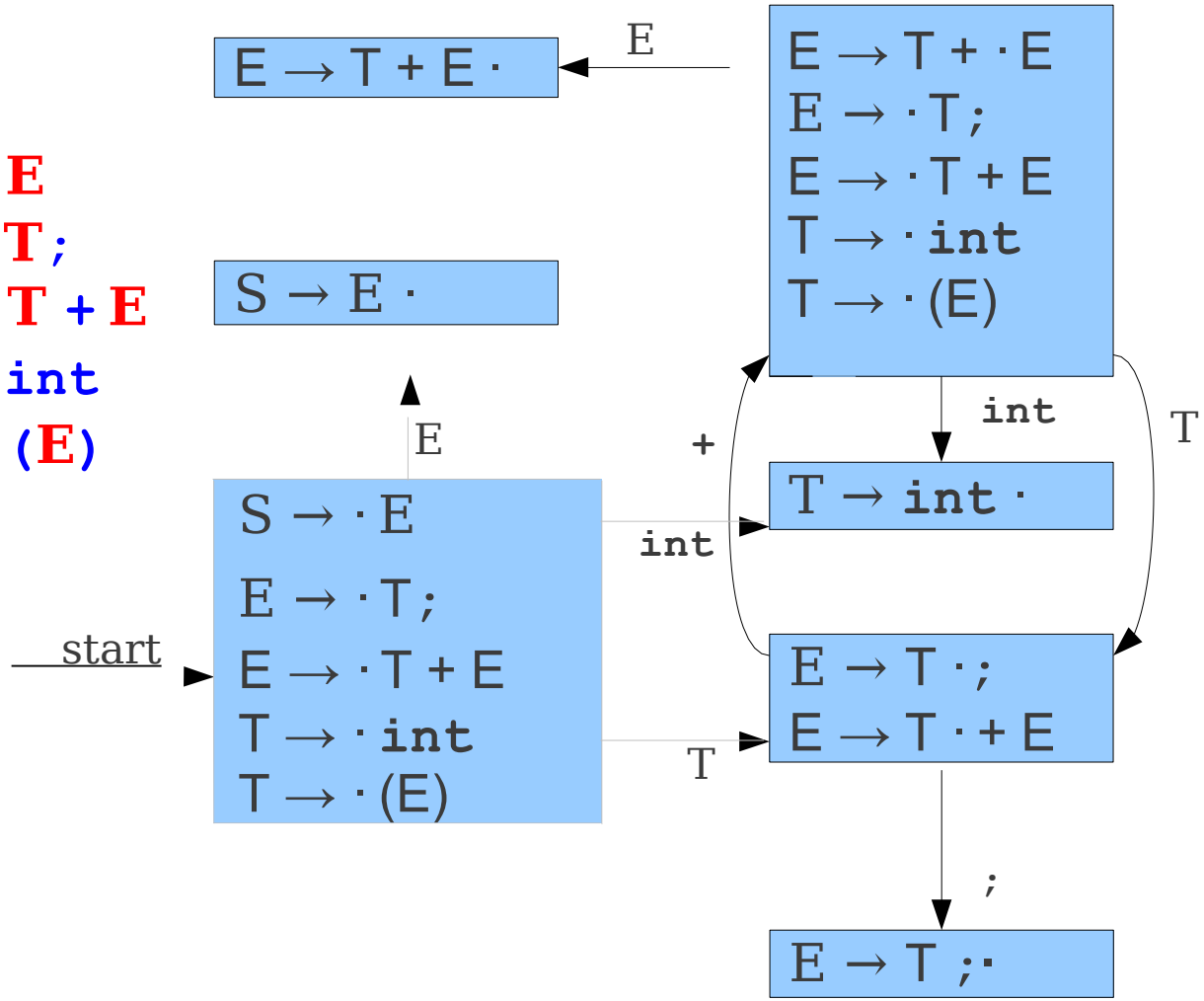
A Deterministic Automaton

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

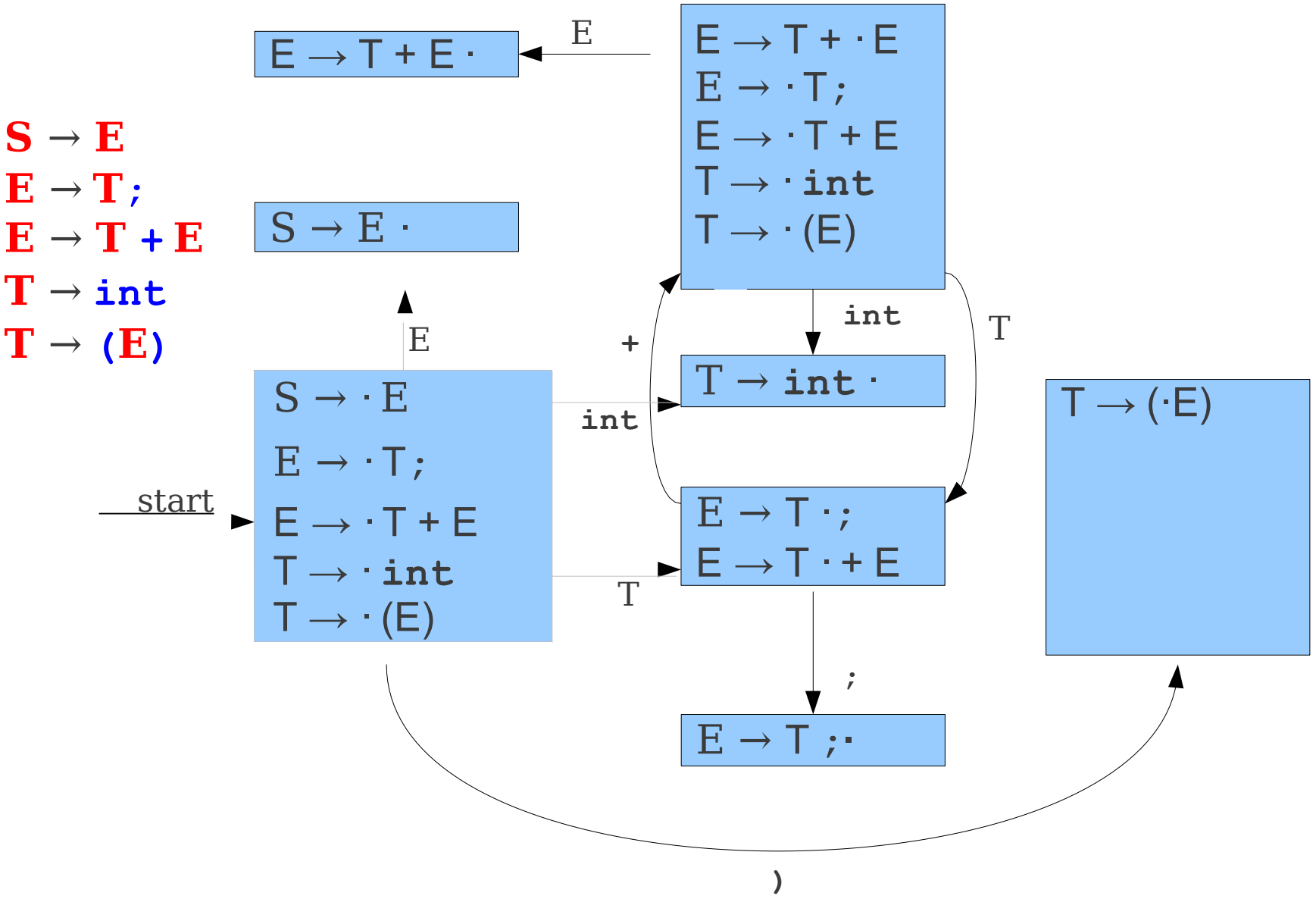


A Deterministic Automaton

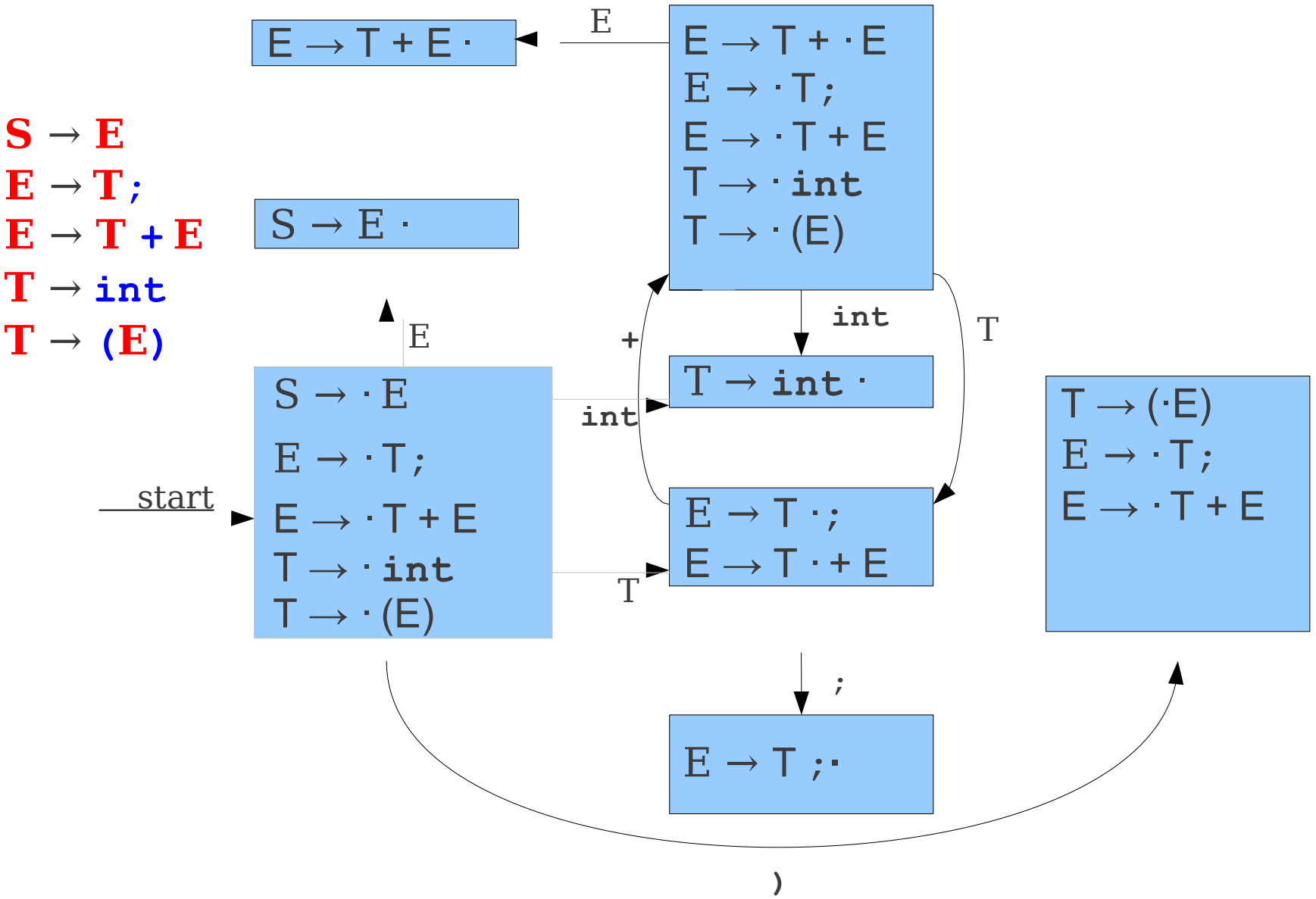
S → E
E → T ;
E → T + E
T → int
T → (E)



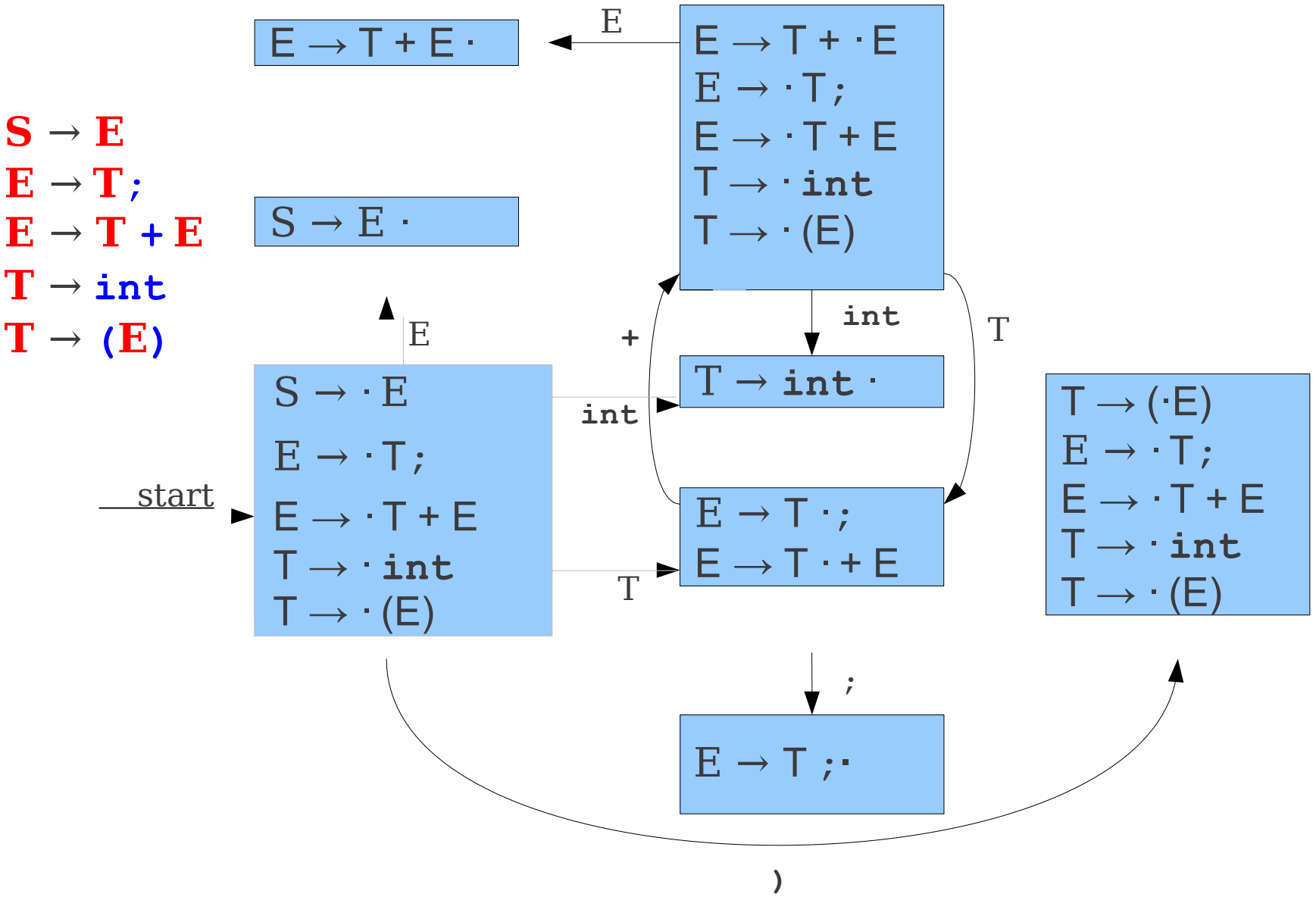
A Deterministic Automaton



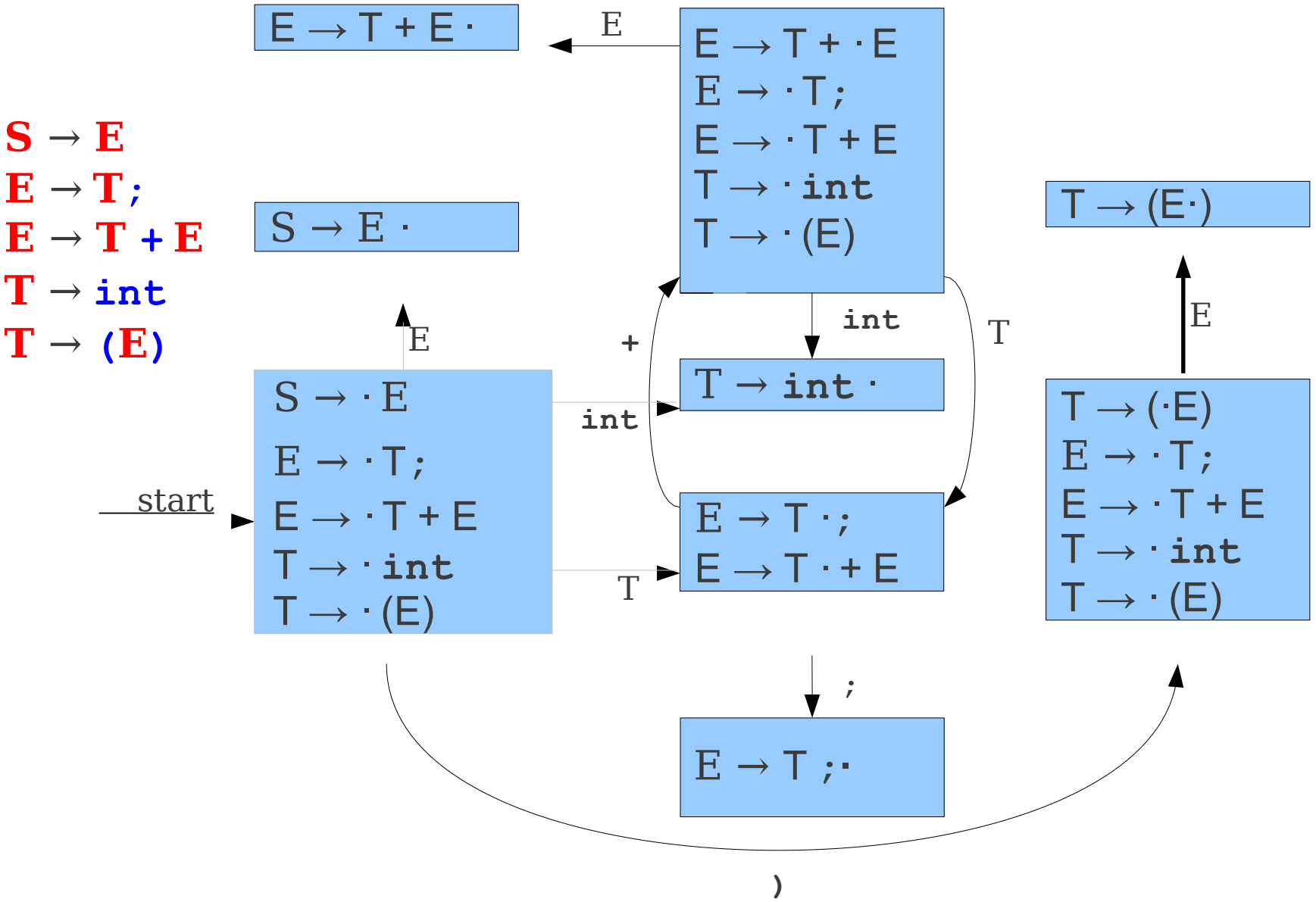
A Deterministic Automaton



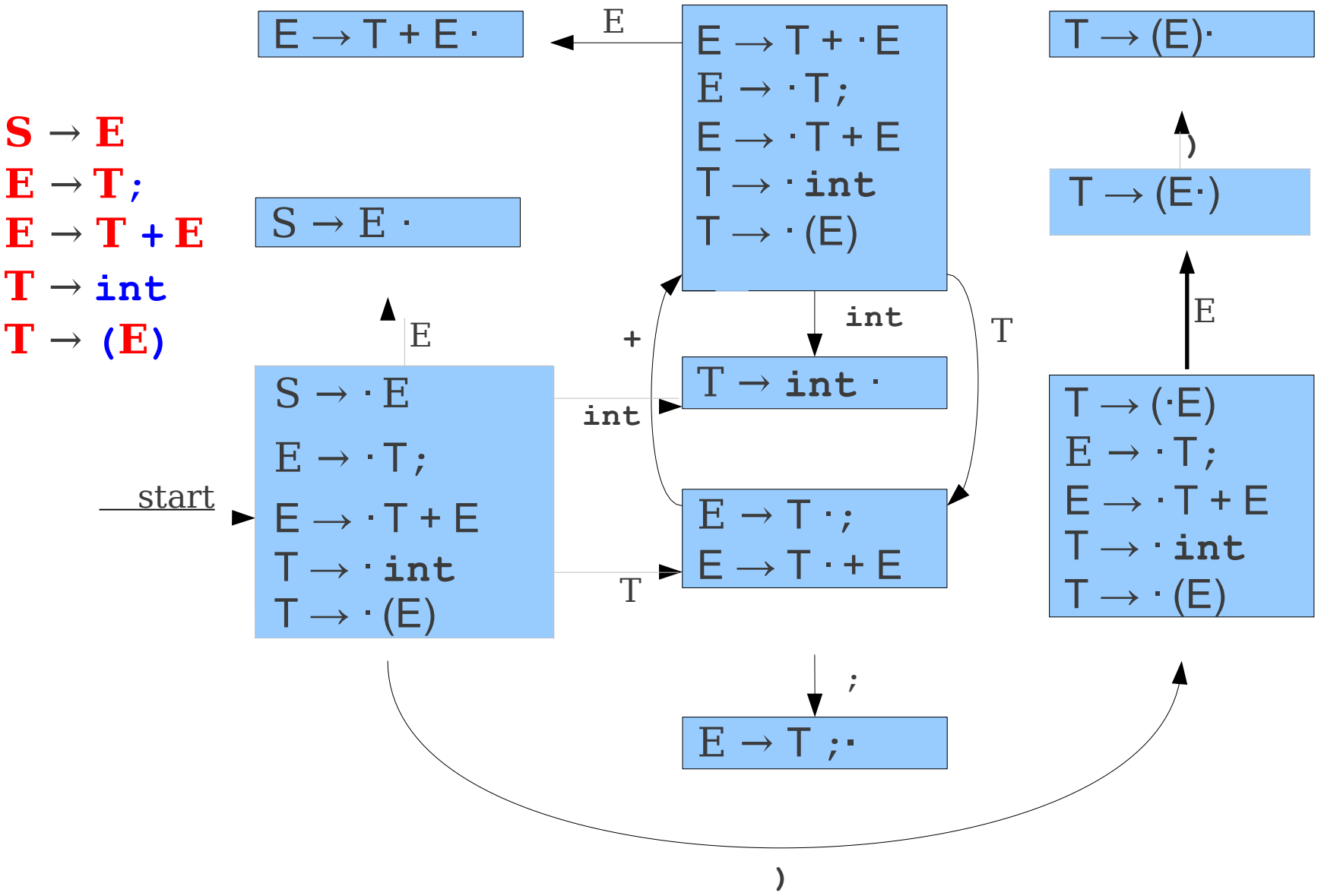
A Deterministic Automaton



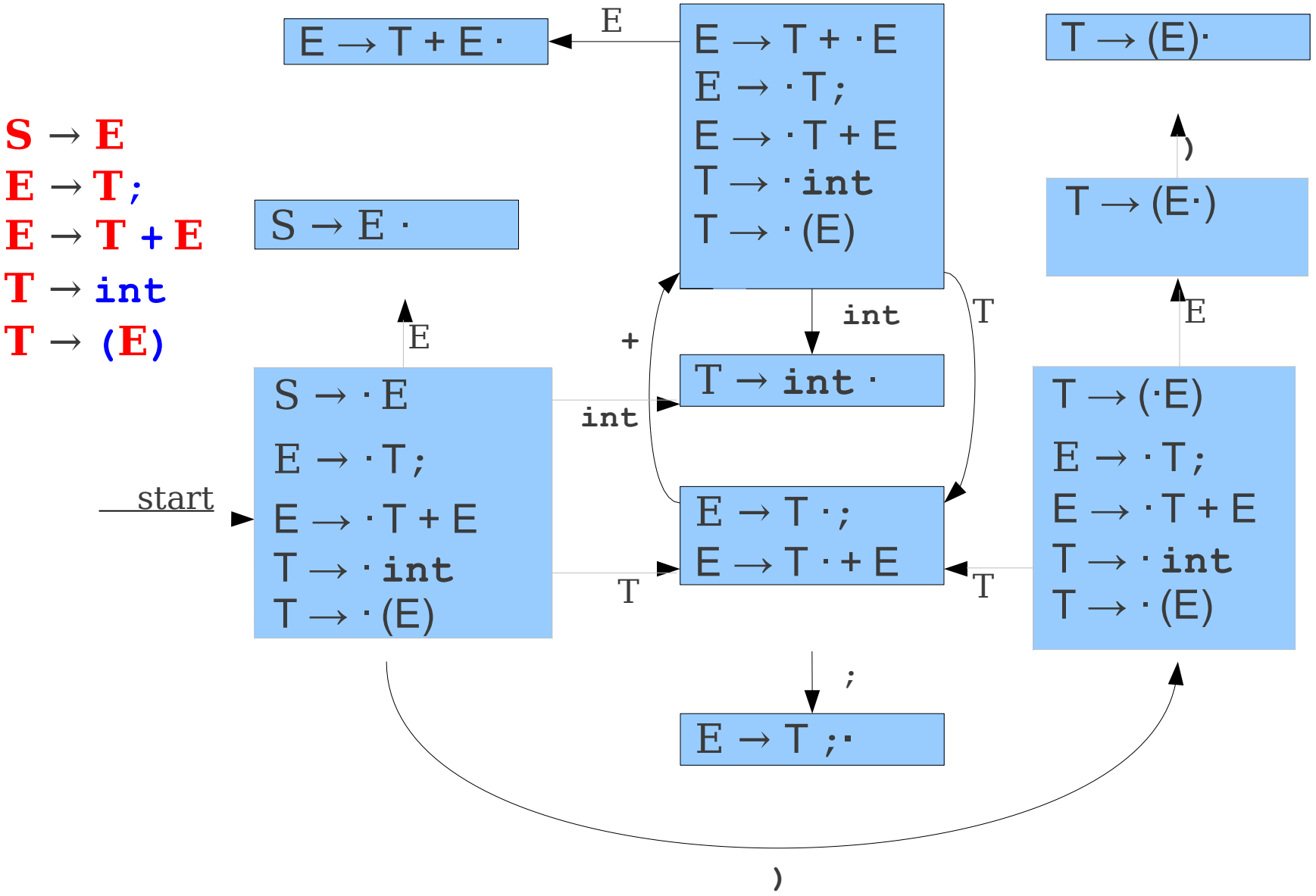
A Deterministic Automaton



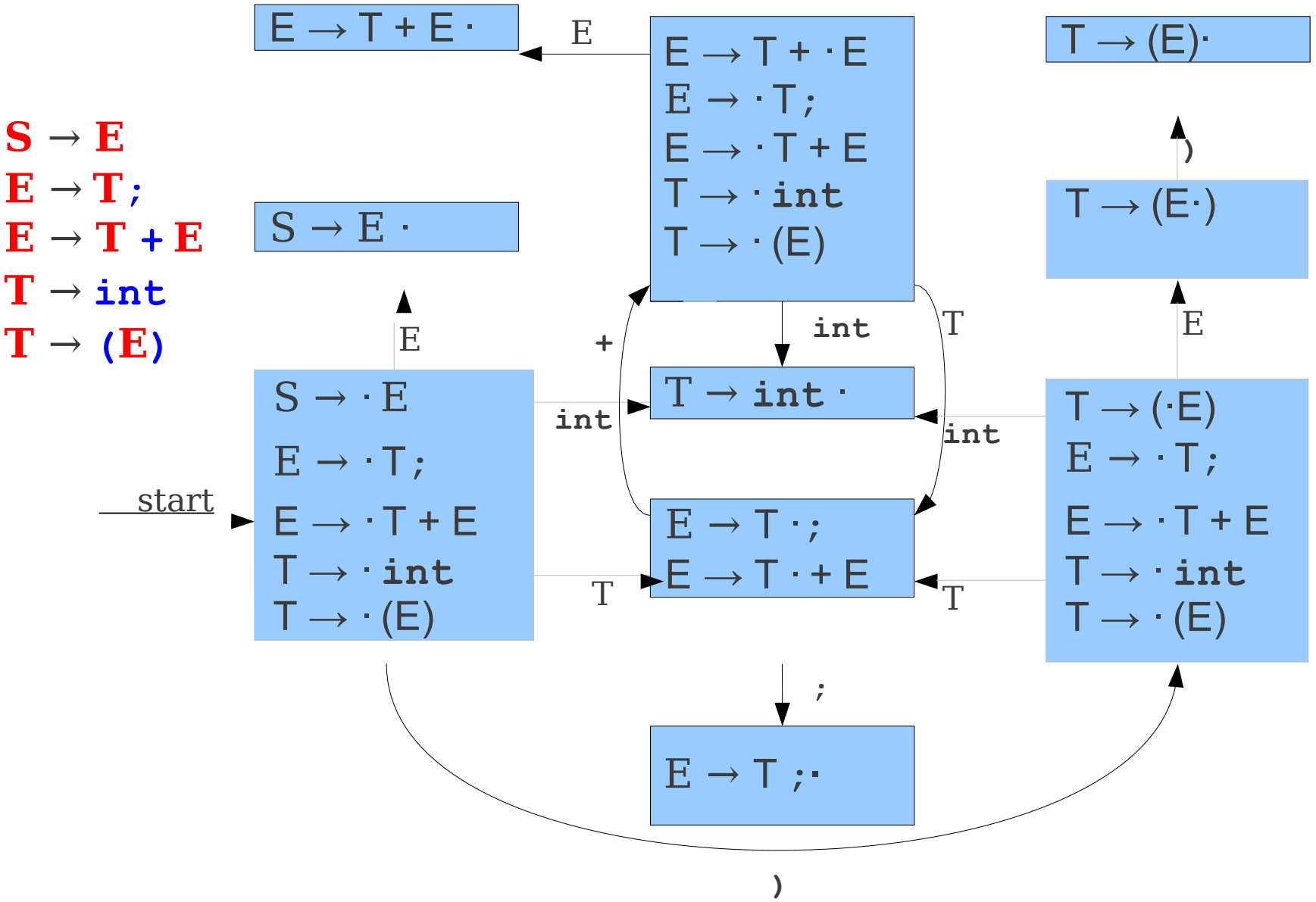
A Deterministic Automaton



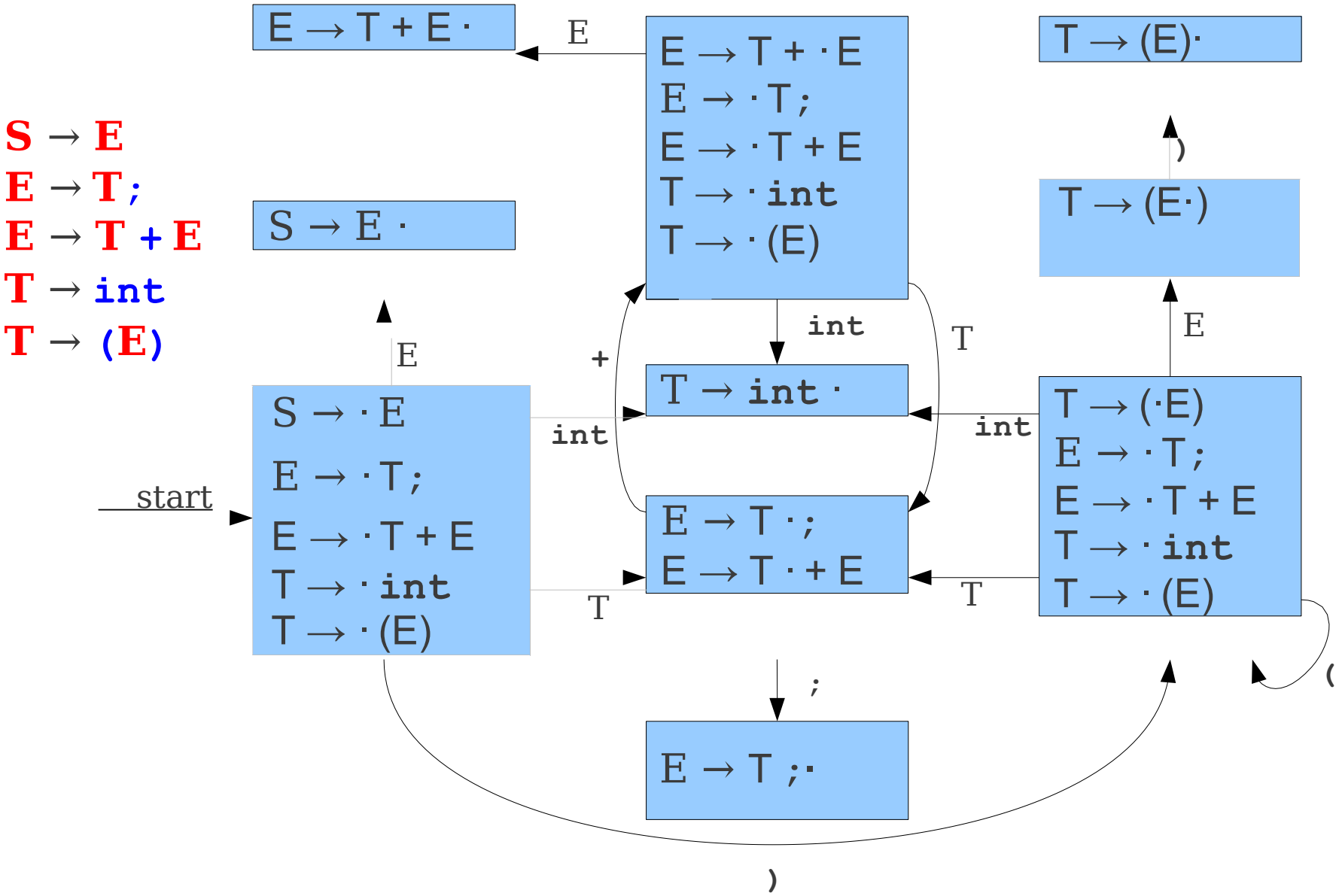
A Deterministic Automaton



A Deterministic Automaton

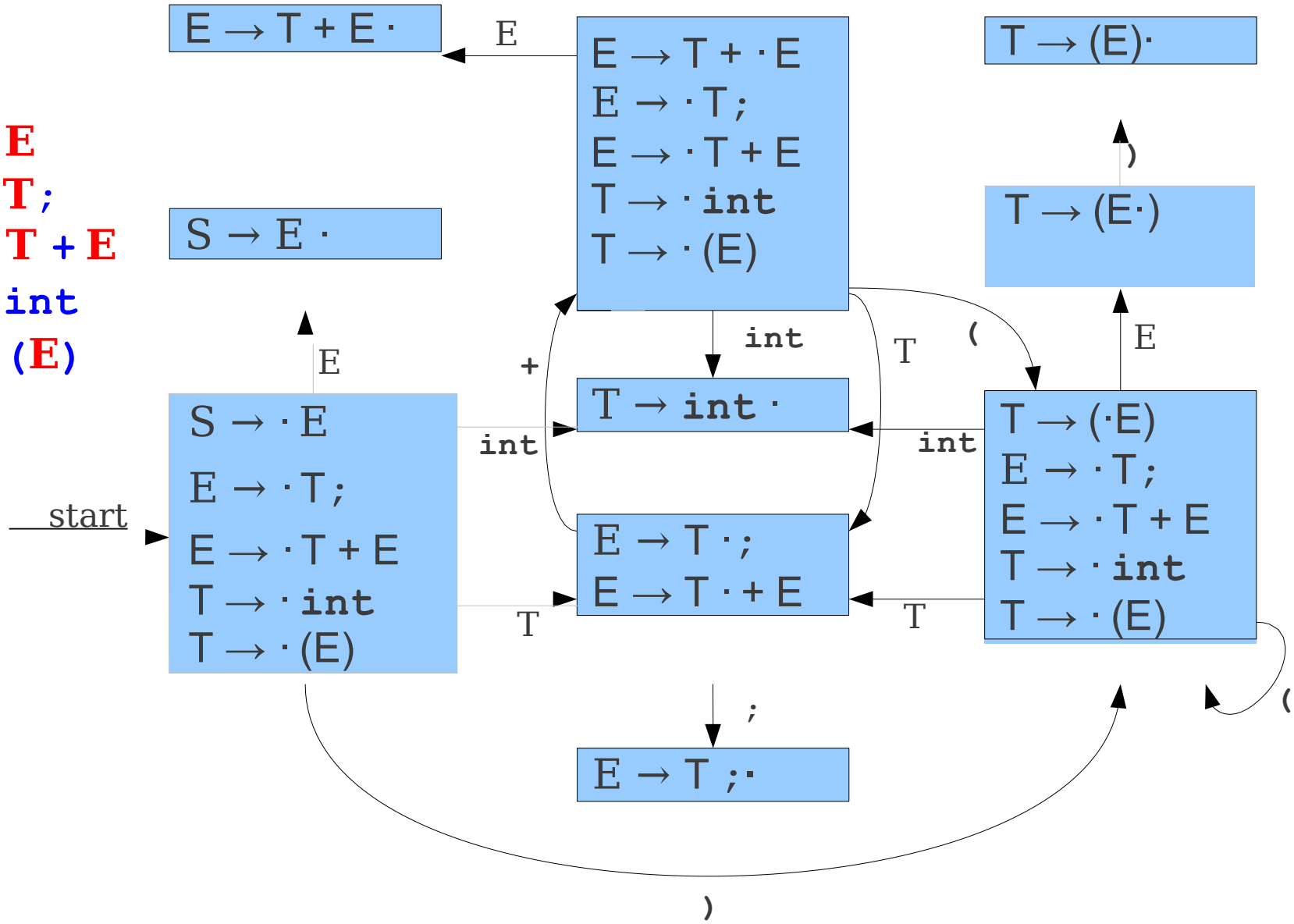


A Deterministic Automaton



A Deterministic Automaton

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



Finding Handles

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

Question Three:

How do we recognize handles?