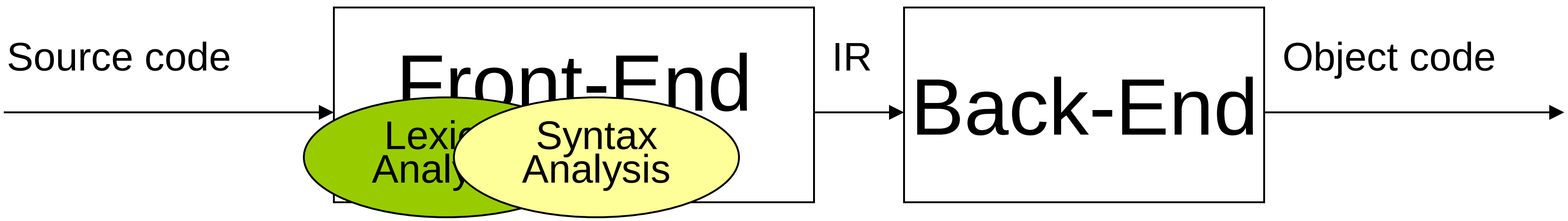
Compiler Design

Lecture 6:
Syntax Analysis
Bottom-Up Parsing (part I)

Dr. Momtazi
momtazi@aut.ac.ir

based on the slides of the course book

Parsing (Syntax Analysis)



- Syntax Analysis:
  - Derivation and parse trees ✓
  - Top-down parsing ✓
  - **Bottom-up parsing**

Outline

- **Introduction**
- Shift-reduce Parsing
- LR Parsing
- Ambiguity
- Error-handling

Top-Down Parsing (overview)

- Start at the root of the tree and grow towards leaves
- Pick a production and try to match the input
  - May need to backtrack if a bad choice is made
  - Use alternative grammars that are backtrack-free (predictive parsing).

Bottom-Up Parsing

- **<u>Goal</u>**: Given a grammar, G, construct a parse tree for a string by starting at the leaves and working to the root
  - i.e., by working from the input string back toward the start symbol S.
- **<u>Recall</u>**: the point of parsing is to construct a derivation:

$$S \Rightarrow \delta_0 \Rightarrow \delta_1 \Rightarrow \delta_2 \Rightarrow \ldots \Rightarrow \delta_{n-1} \Rightarrow sentence$$

  - To derive δ_{i-1} from δ_i, we match some *rhs* b in δ_i, then replace b with its corresponding *lhs*, A.
  - This is called a **<u>reduction</u>** (it assumes $A \rightarrow b$).
  - We reduce a substring of the sentential form back to a nonterminal.

Bottom-Up Parsing

- **Example**:

```
1.   Goal→aABe
2.        A→Abc
3.           |b
4.        B→d
```

- Input string: **abbcde**.

Sentential Form	Production	Position
abbcde	3	2
a A bcde	2	4
a A de	4	3
a A B e	1	4
Goal	-	-

Bottom-Up Parsing

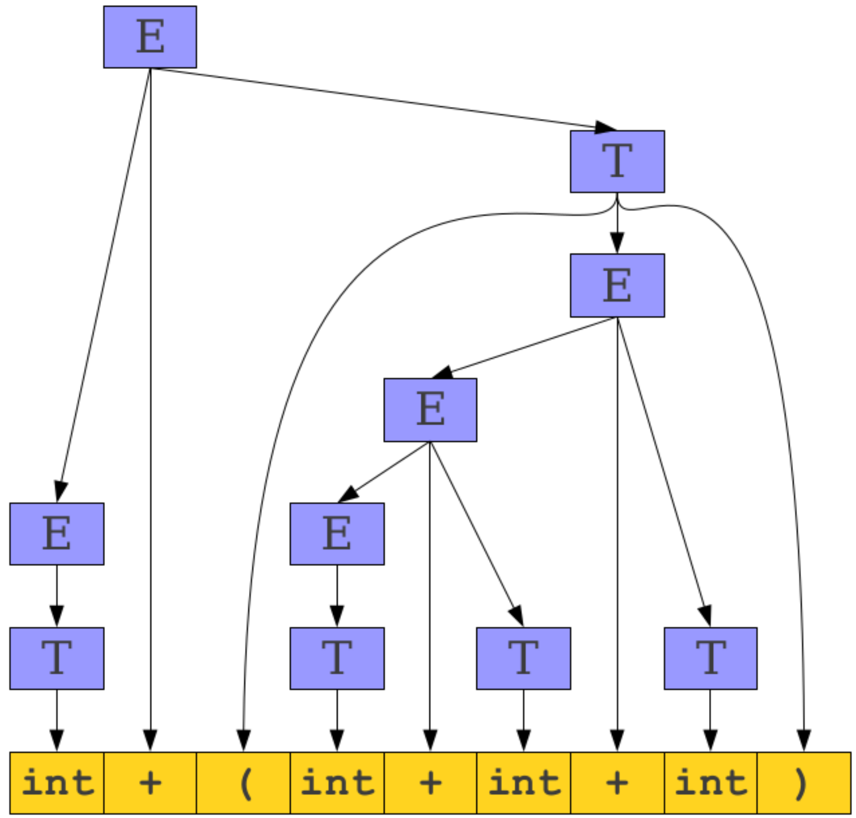
- Idea: Apply productions in reverse to convert the user's program to the start symbol.
- As with top-down, could be done with a DFS or BFS, though this is rarely done in practice.
- We'll be exploring four directional, predictive bottom-up parsing techniques:
  - Directional: Scan the input from left-to-right.
  - Predictive: Guess which production should be inverted.

$E \rightarrow T$

$E \rightarrow E + T$

$T \rightarrow$ int

$T \rightarrow (E)$



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

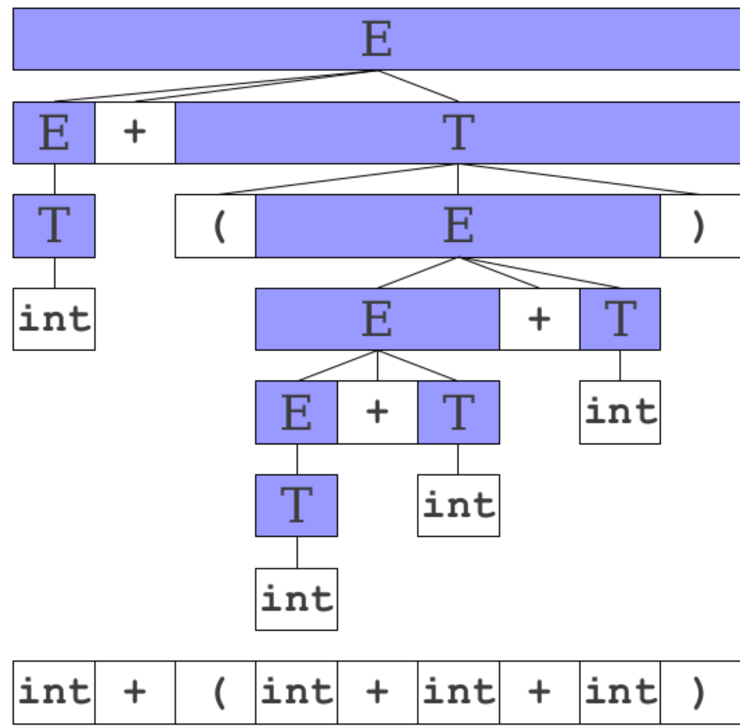
$$
\begin{aligned}
& \texttt{int + (int + int + int)} \\
\Rightarrow\ & T \texttt{ + (int + int + int)} \\
\Rightarrow\ & E \texttt{ + (int + int + int)} \\
\Rightarrow\ & E \texttt{ + (}T\texttt{ + int + int)} \\
\Rightarrow\ & E \texttt{ + (}E\texttt{ + int + int)} \\
\Rightarrow\ & E \texttt{ + (}E\texttt{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

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

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

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

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



Handles

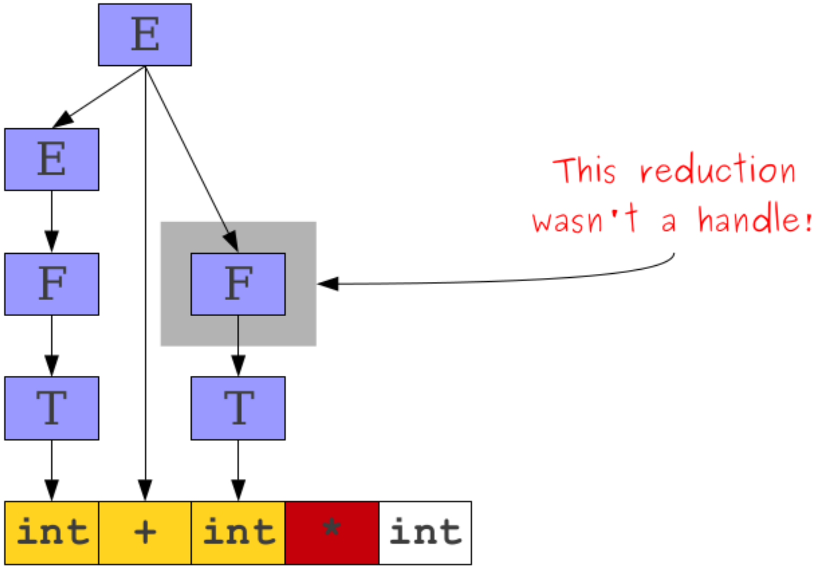
- The handle of a parse tree T is the leftmost complete cluster of leaf nodes.
- A left-to-right, bottom-up parse works by iteratively searching for a handle, then reducing the handle.

Summarizing Our Intuition

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

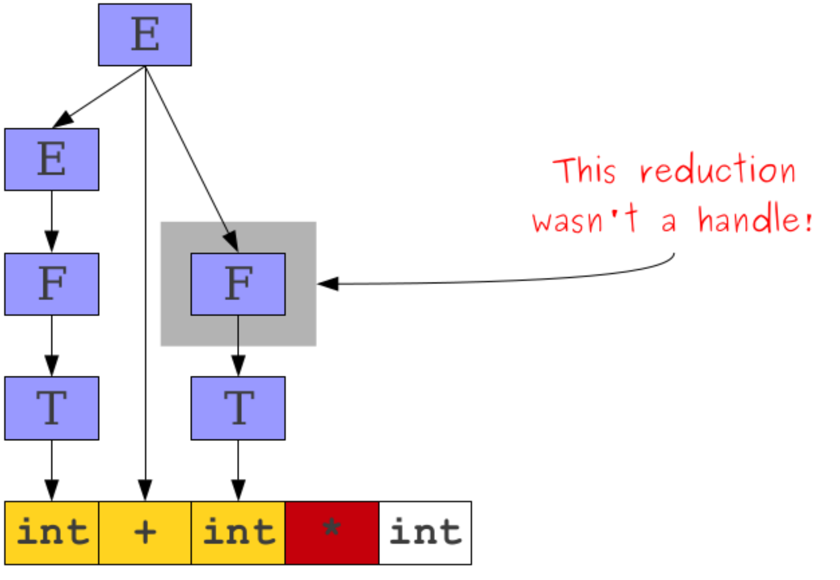
Handles

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



Handles

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



The leftmost reduction isn't always the handle.

Finding Handles: Main Questions

- 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 have found a possible handle, how do we confirm that it is correct?

Where are Handles?

- Recall: A left-to-right, bottom-up parse traces a rightmost derivation in reverse.
- Each time we do a reduction, we are reversing a production applied to the rightmost nonterminal symbol.
- Suppose that our current sentential form is $\alpha\gamma\omega$, where γ is the handle and $A \rightarrow \gamma$ is a production rule.
- After reducing γ back to A, we have the string $\alpha A\omega$.
- Thus ω must consist purely of terminals, since otherwise the reduction we just did was not for the rightmost terminal.

Why This Matters

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

Outline

- Introduction
- **Shift-reduce Parsing**
- LR Parsing
- Ambiguity
- Error-handling

Shift/Reduce Parsing

- The bottom-up parsers we will consider are called shift/reduce parsers.
  - Idea: Split the input into two parts:
- Contrast with the LL(1) predict/match parser.
  - 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

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

int	+	int	*	int	+	int

A Sample Shift/Reduce Parse

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

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

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

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

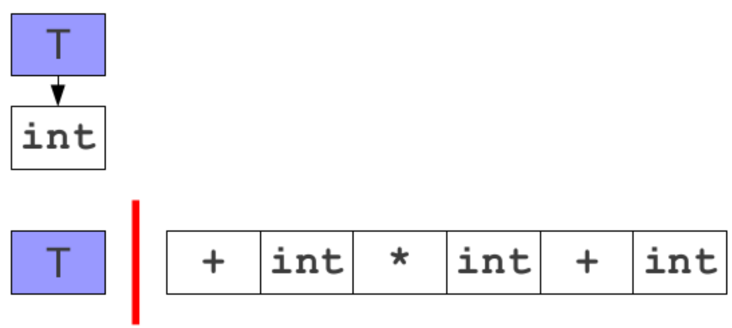
$\mathbf{T} \rightarrow$ `int`

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

int	|	+	int	*	int	+	int

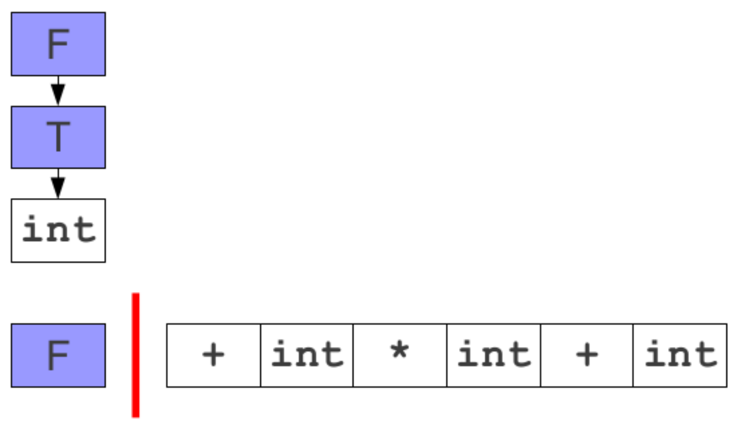
A Sample Shift/Reduce Parse

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



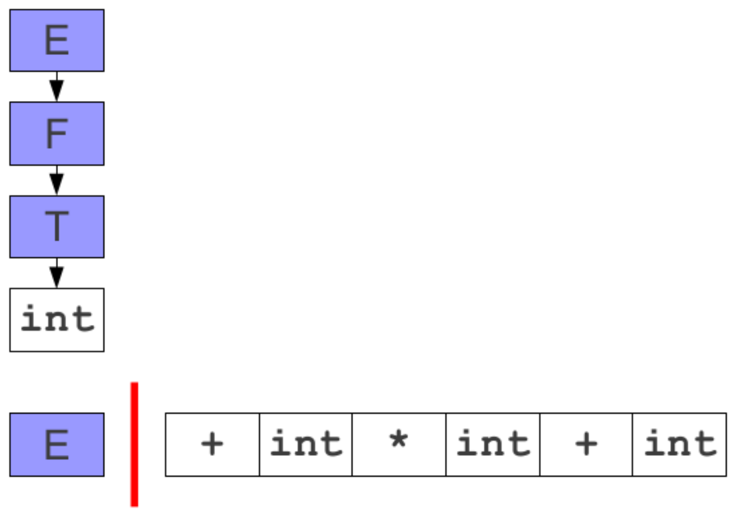
A Sample Shift/Reduce Parse

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



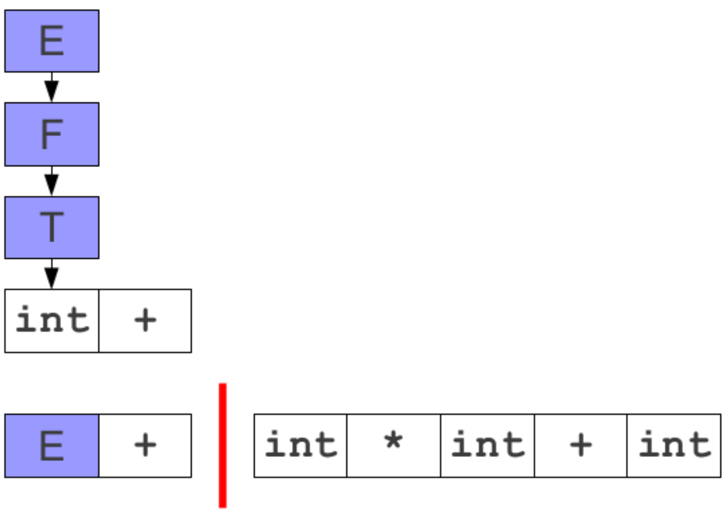
A Sample Shift/Reduce Parse

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



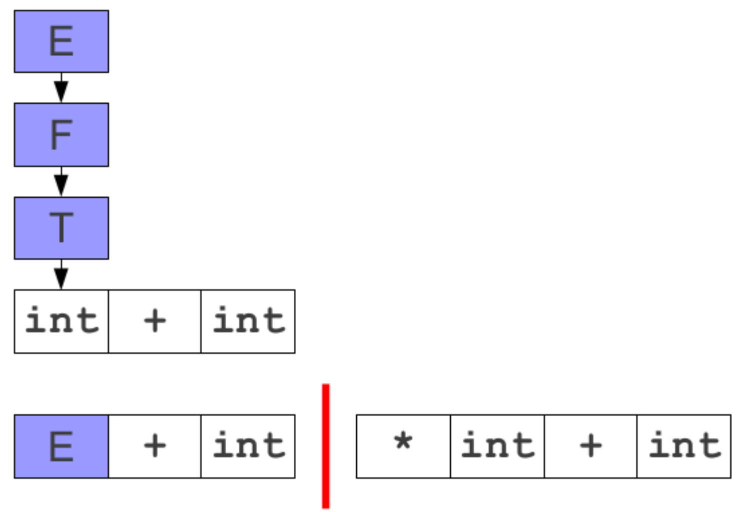
A Sample Shift/Reduce Parse

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



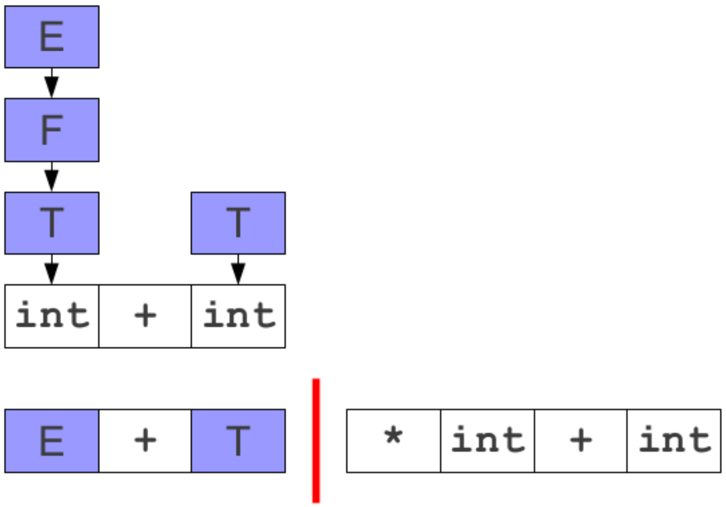
A Sample Shift/Reduce Parse

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



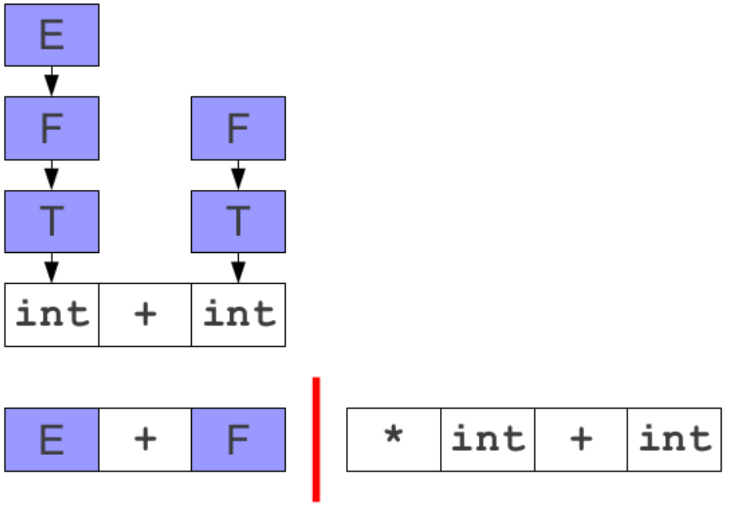
A Sample Shift/Reduce Parse

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



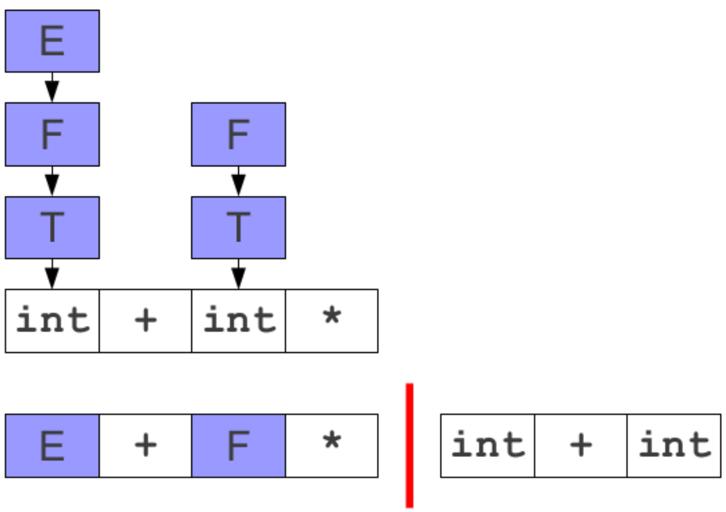
A Sample Shift/Reduce Parse

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



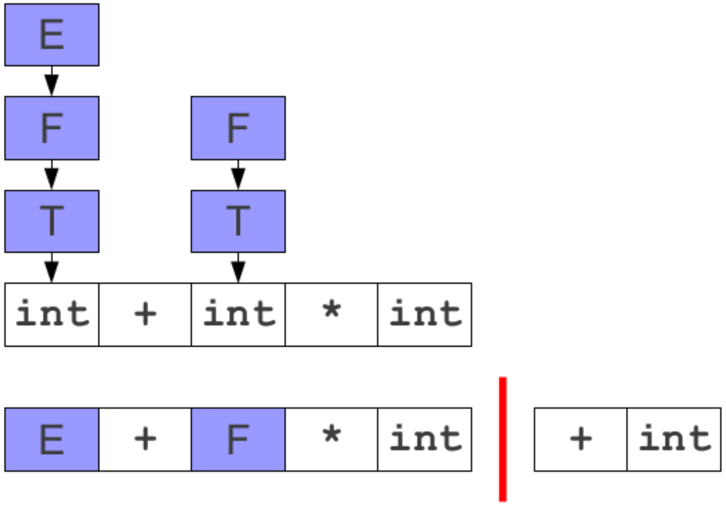
A Sample Shift/Reduce Parse

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



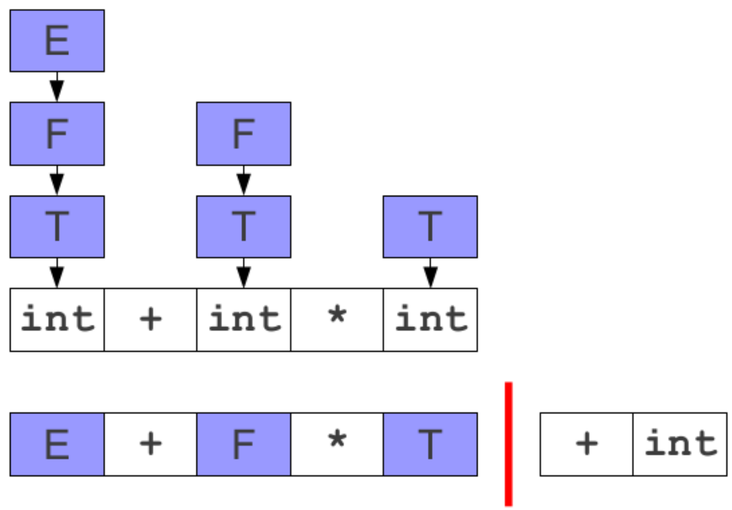
A Sample Shift/Reduce Parse

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



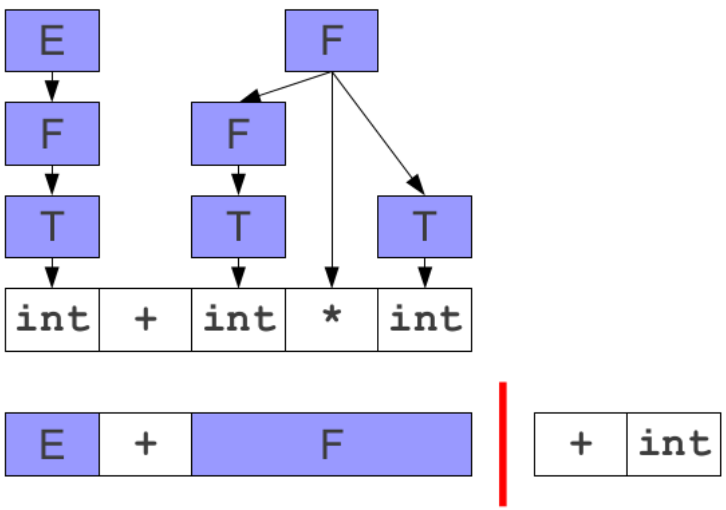
A Sample Shift/Reduce Parse

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



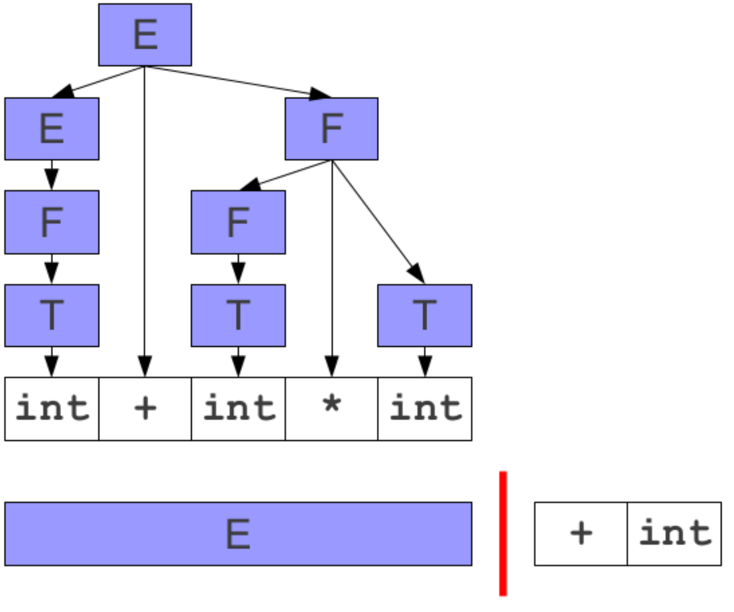
A Sample Shift/Reduce Parse

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



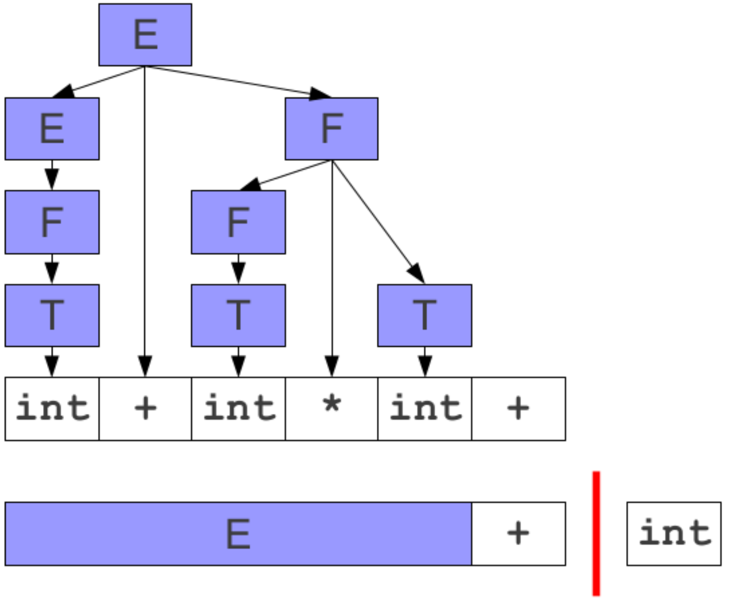
A Sample Shift/Reduce Parse

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



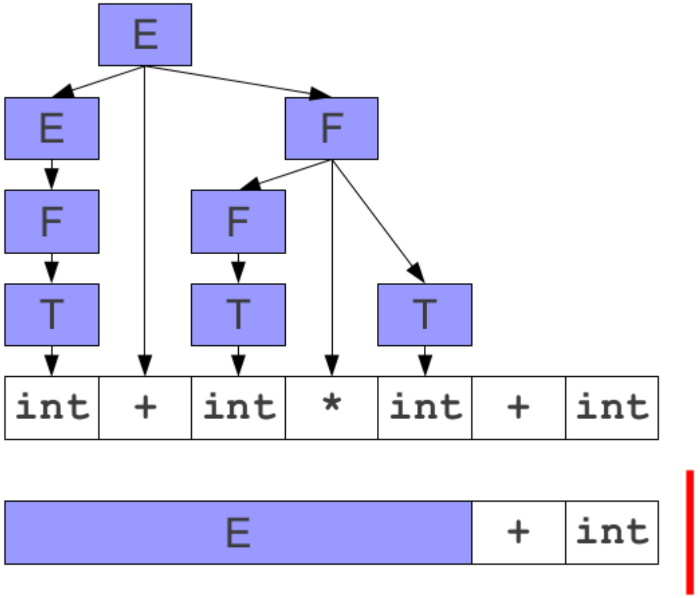
A Sample Shift/Reduce Parse

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



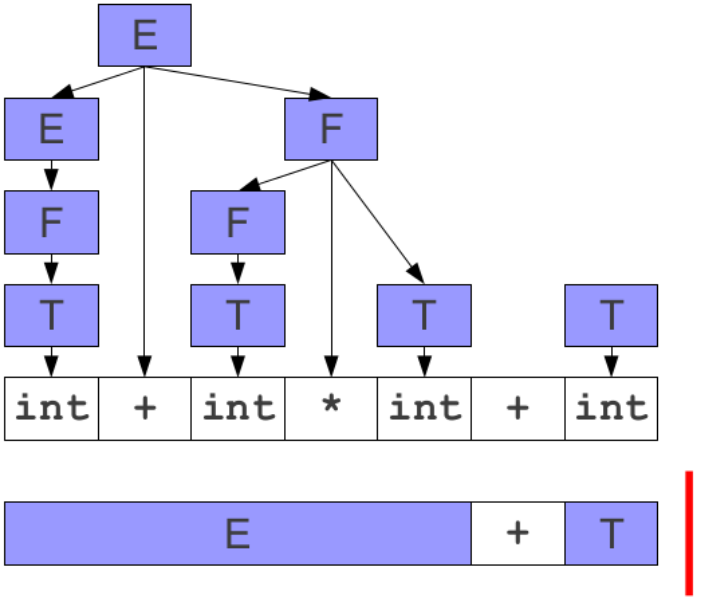
A Sample Shift/Reduce Parse

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



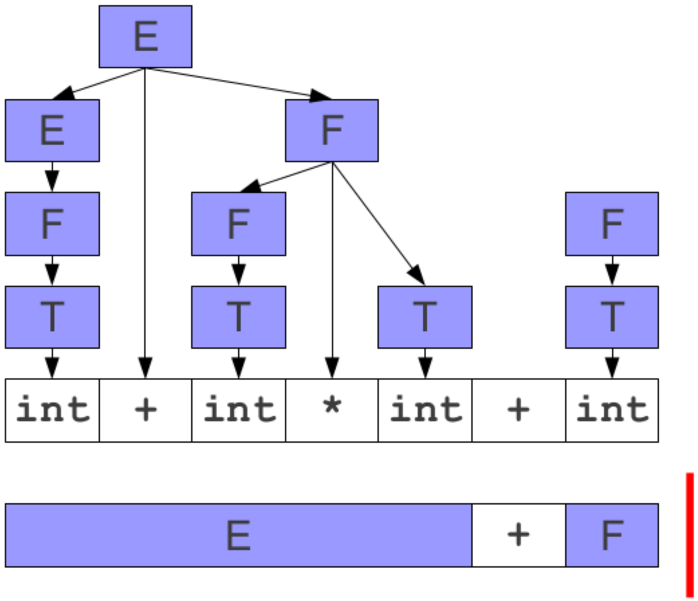
A Sample Shift/Reduce Parse

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



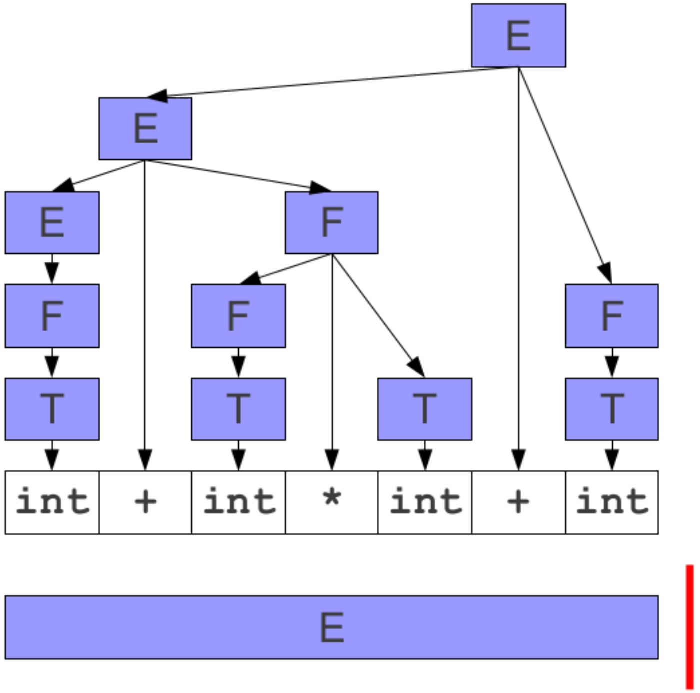
A Sample Shift/Reduce Parse

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



A Sample Shift/Reduce Parse

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



An Important Observation

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

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.

Simplifying our Terminology

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

Finding Handles: Main Questions

- 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?

Searching for Handles

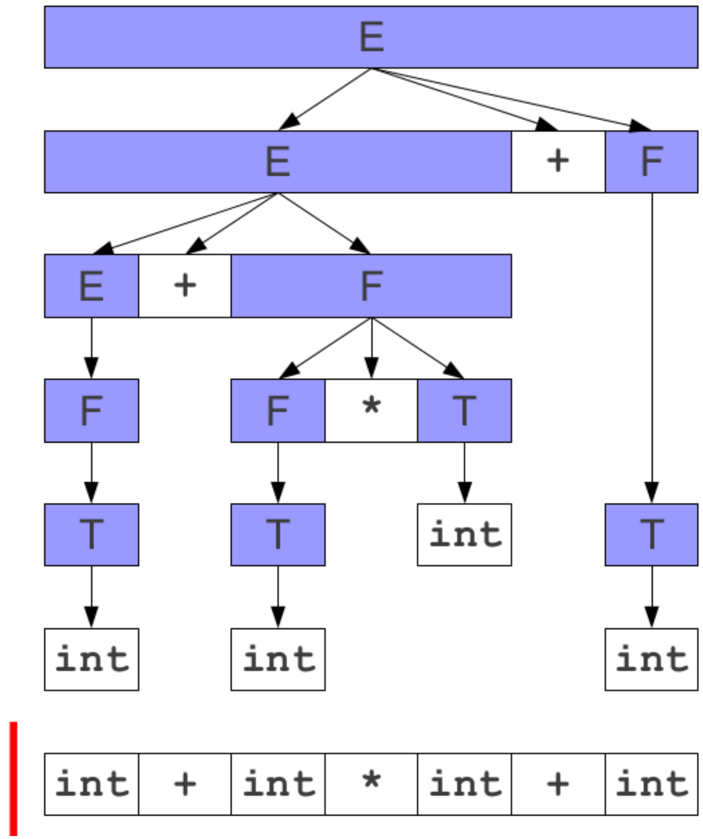
- When using a shift/reduce parser, we must decide whether to shift or reduce at each point.
- We only want to reduce when we know we have a handle.
- Question: How can we tell that we might be looking at a handle?

Exploring the Left Side

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

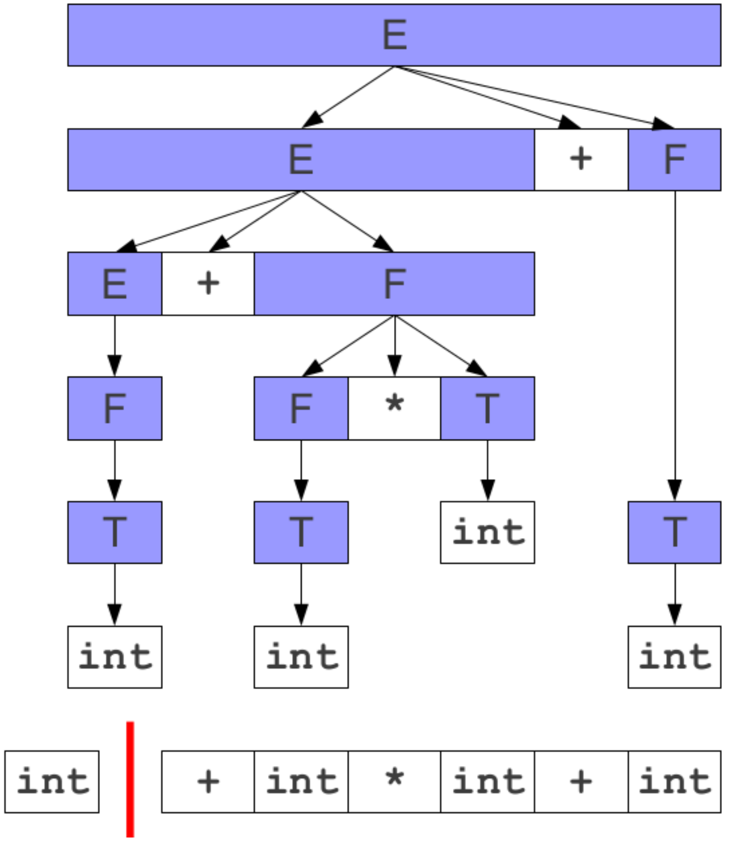
Another Look at Handles

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



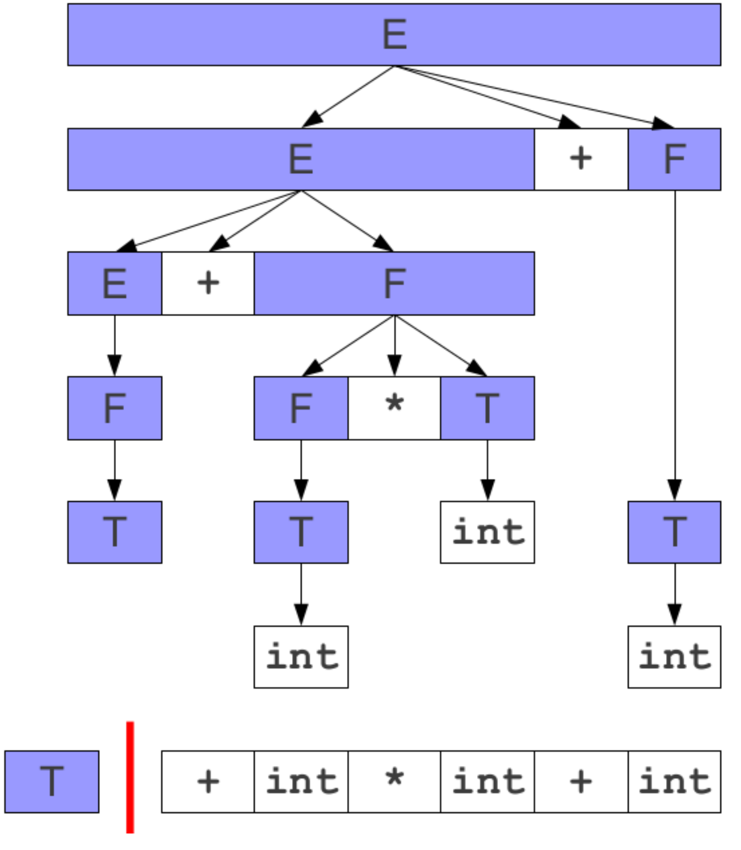
Another Look at Handles

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



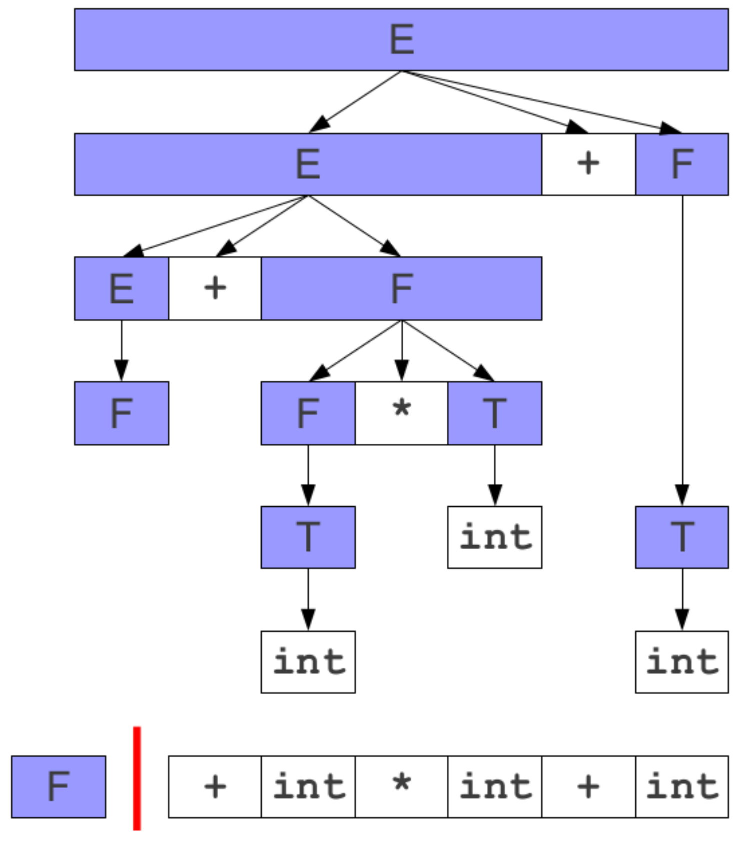
Another Look at Handles

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



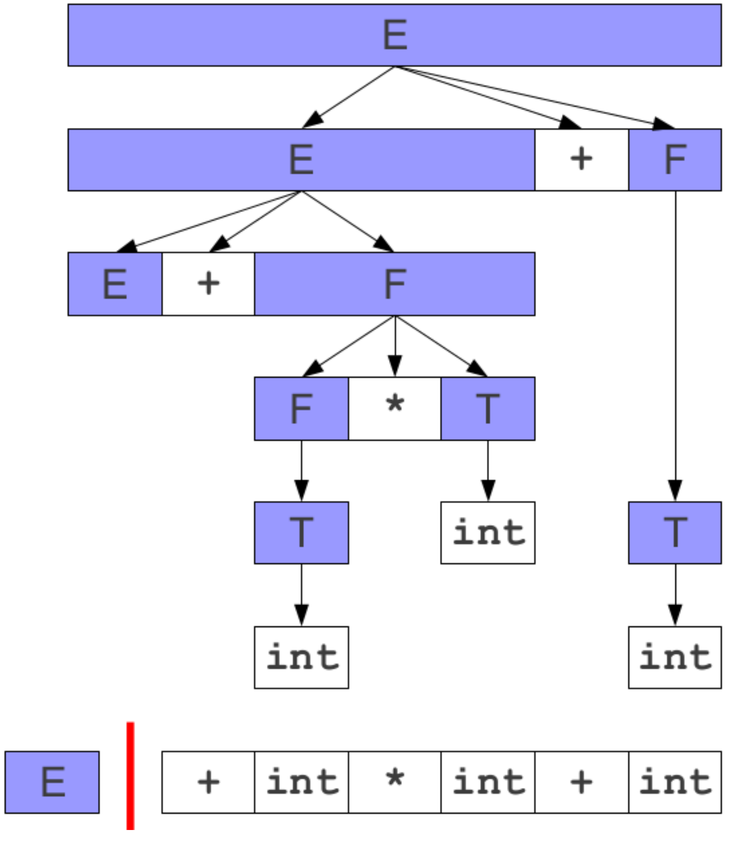
Another Look at Handles

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



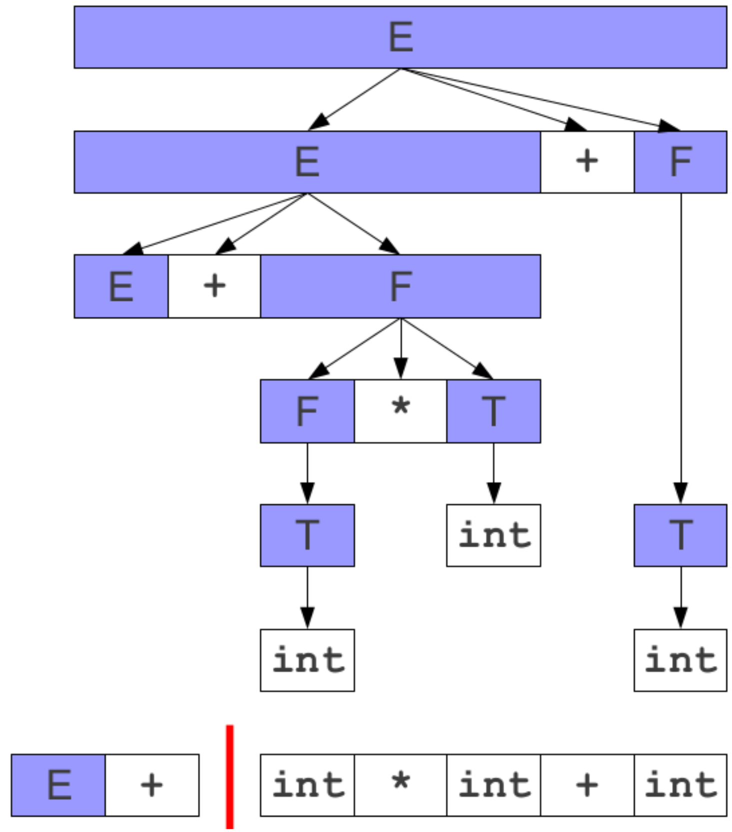
Another Look at Handles

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



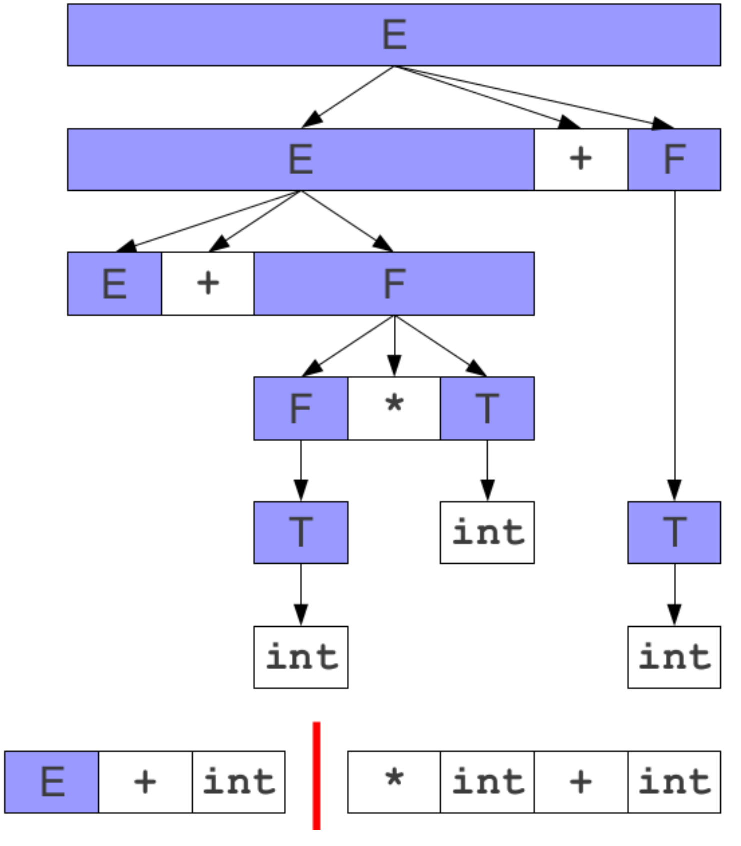
Another Look at Handles

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



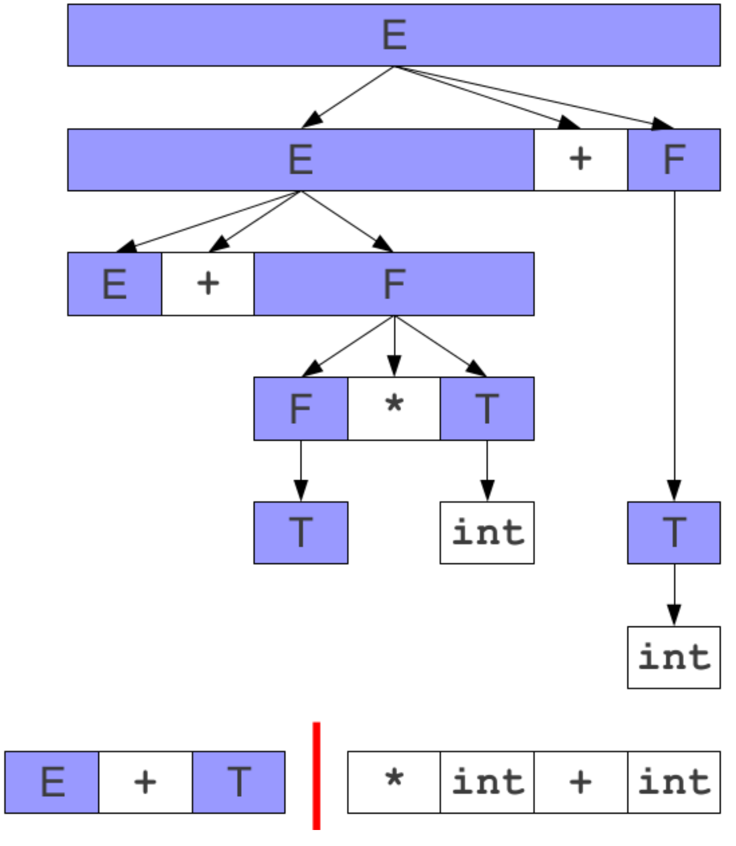
Another Look at Handles

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



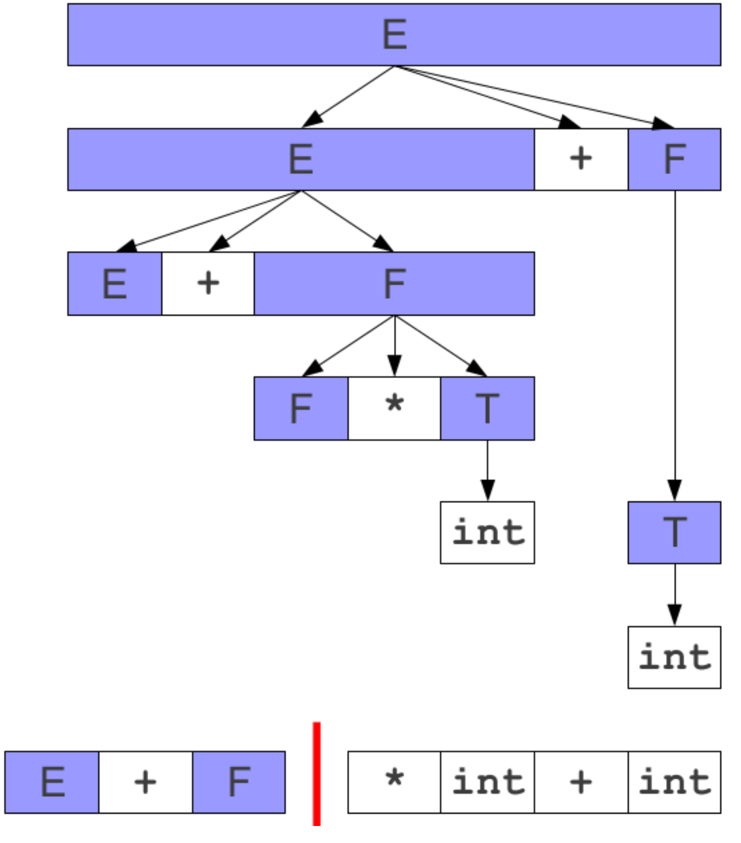
Another Look at Handles

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



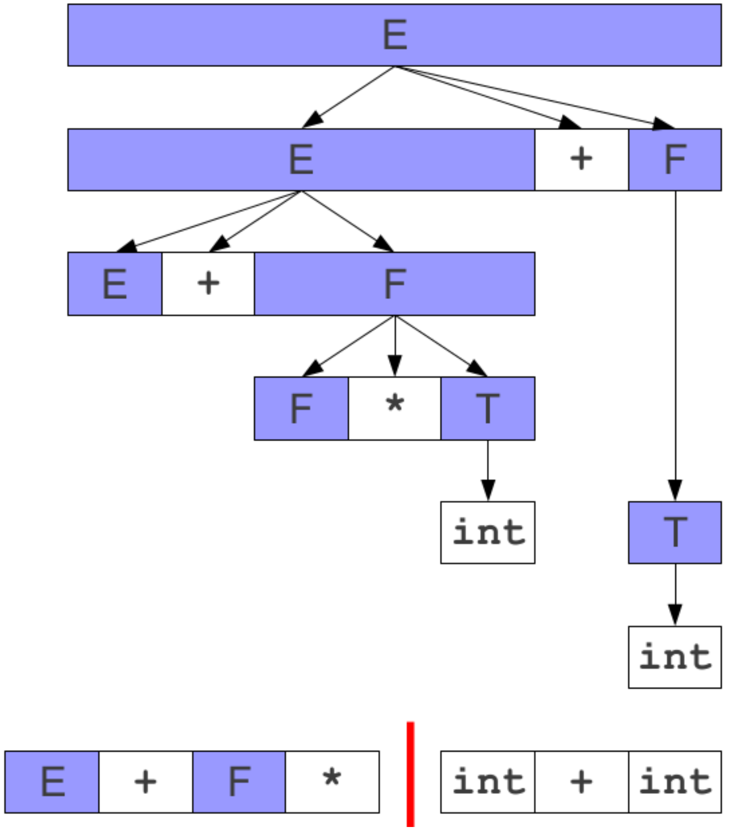
Another Look at Handles

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



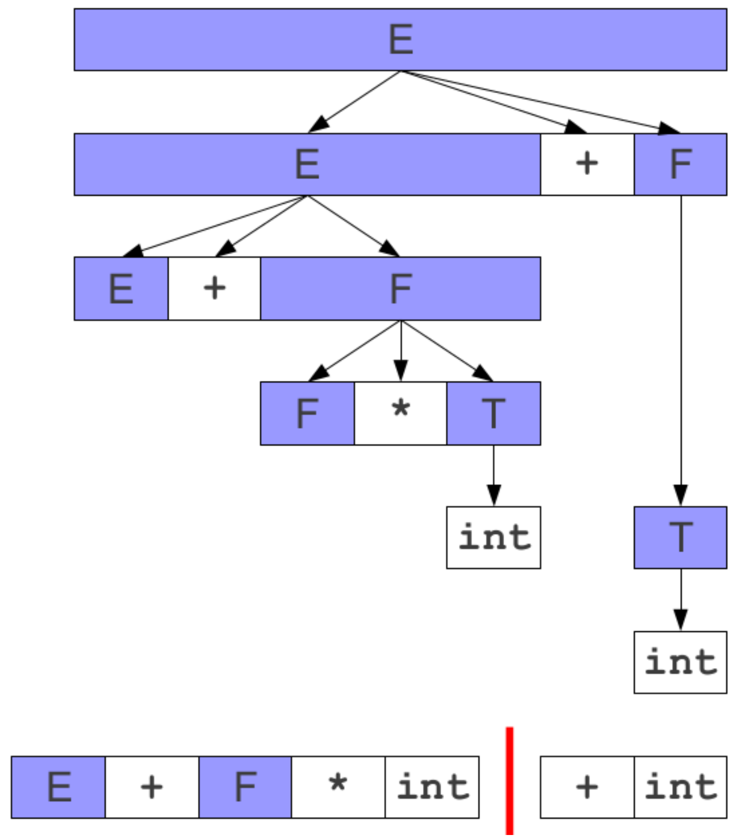
Another Look at Handles

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



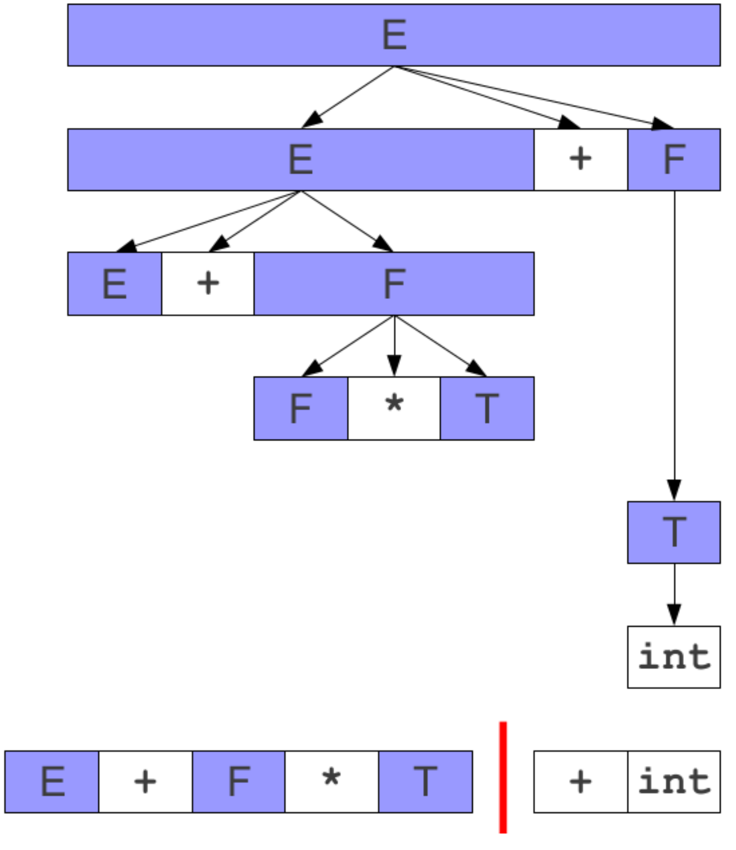
Another Look at Handles

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



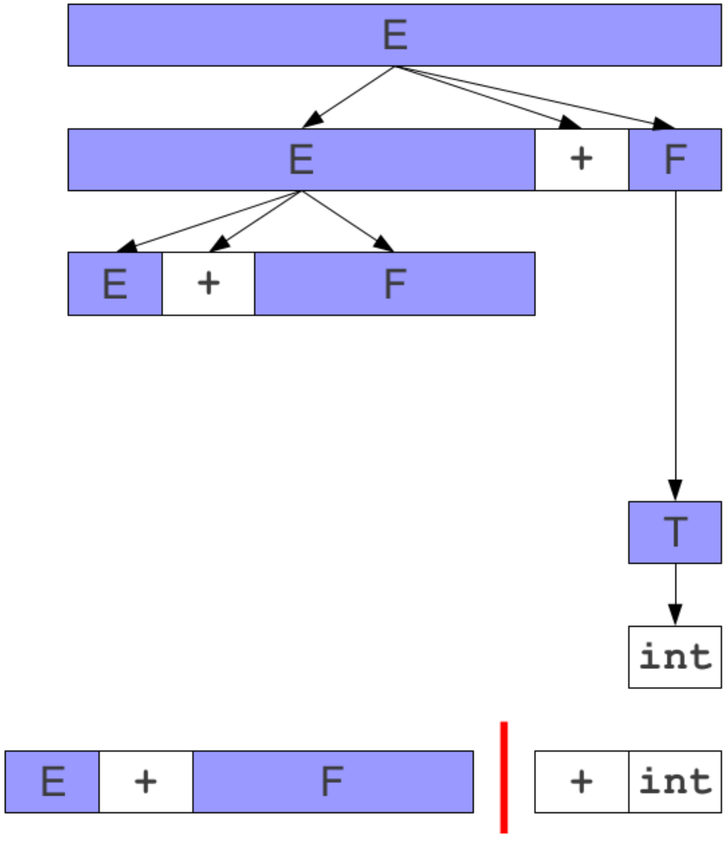
Another Look at Handles

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



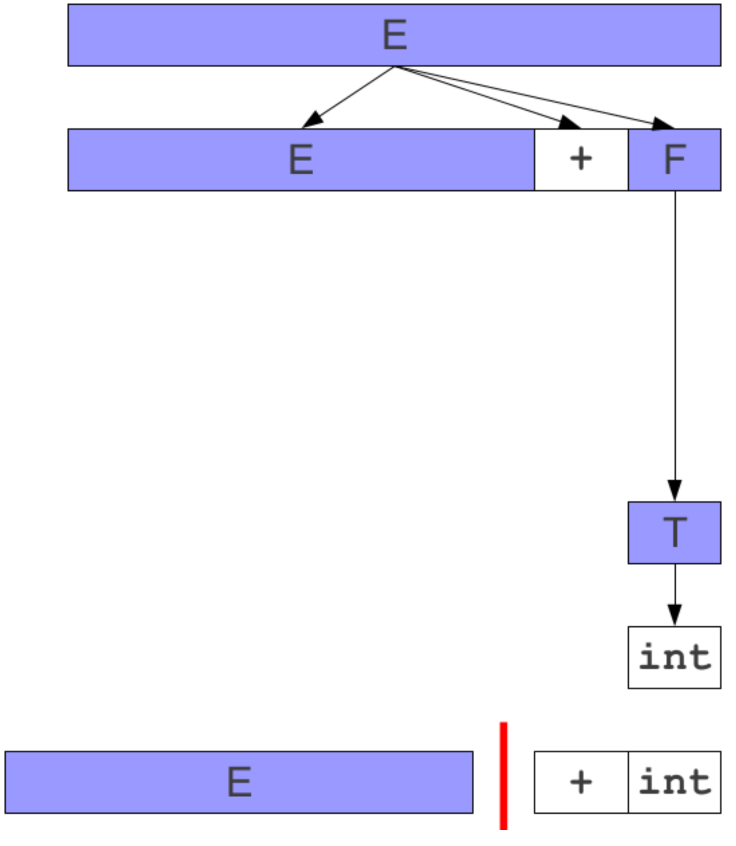
Another Look at Handles

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



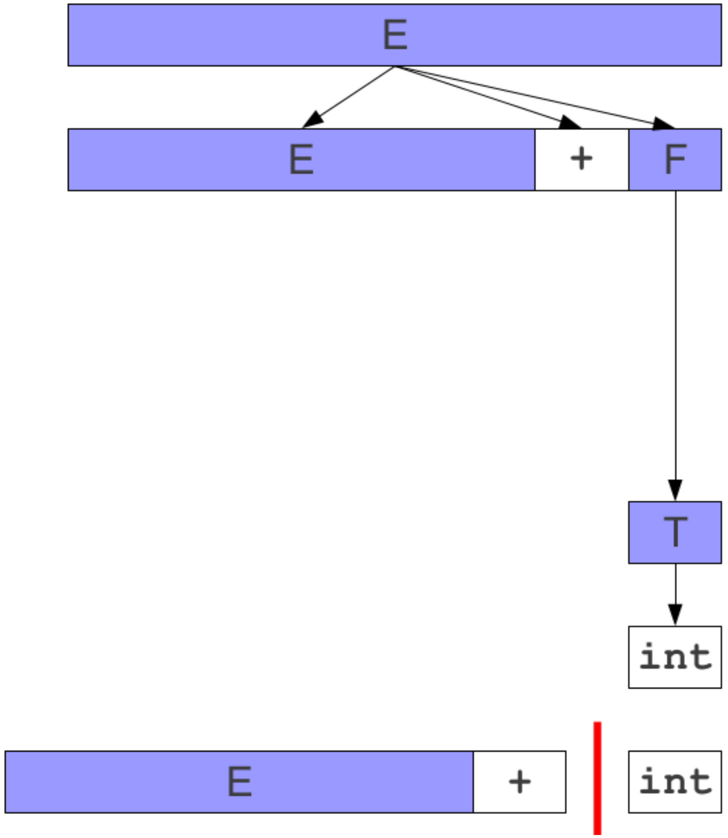
Another Look at Handles

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



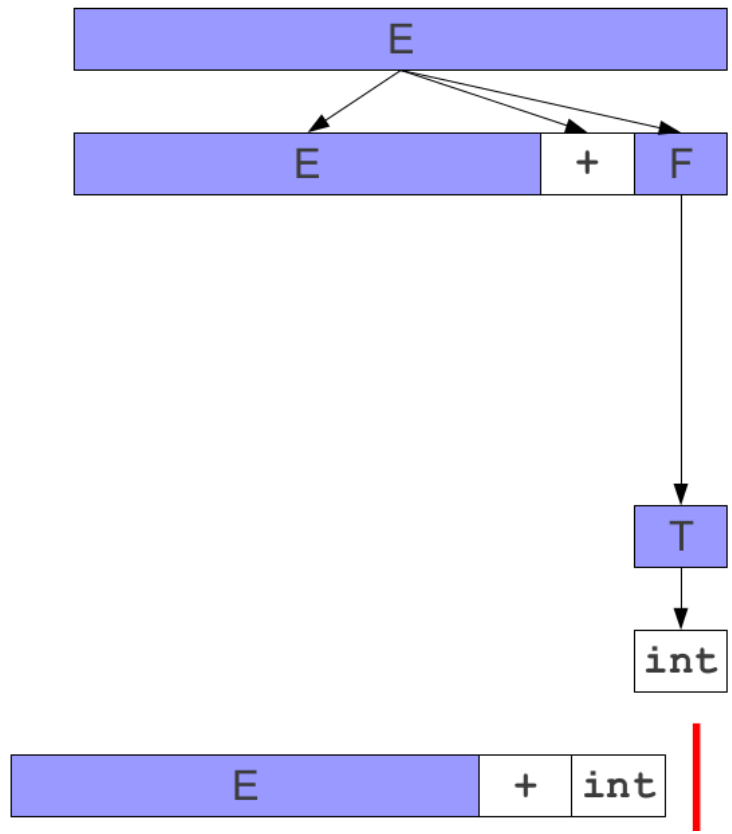
Another Look at Handles

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



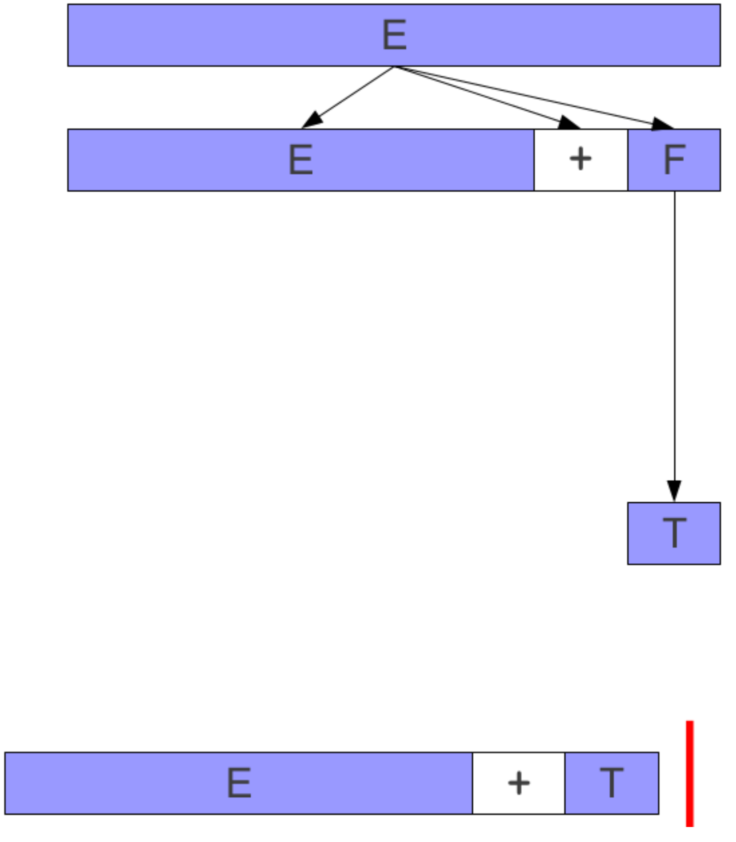
Another Look at Handles

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

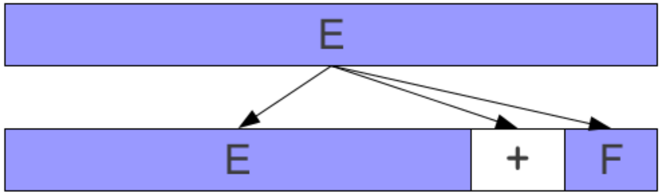


Another Look at Handles

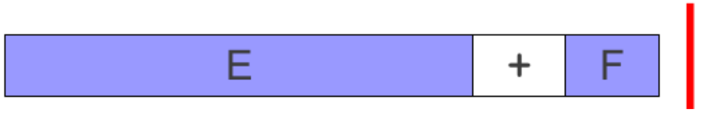
$E \rightarrow F$
$E \rightarrow E + F$
$F \rightarrow F * T$
$F \rightarrow T$
$T \rightarrow \texttt{int}$
$T \rightarrow (E)$



Another Look at Handles



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



Another Look at Handles

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

Tracking Our Position

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

int	+	int	*	int	+	int

Tracking Our Position

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

int	+	int	*	int	+	int

Tracking Our Position

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

$S \rightarrow \cdot E$

int	+	int	*	int	+	int

Tracking Our Position

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

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

int	+	int	*	int	+	int

Tracking Our Position

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

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

int	+	int	*	int	+	int

Tracking Our Position

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

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

int	+	int	*	int	+	int

Tracking Our Position

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

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

int	+	int	*	int	+	int

Tracking Our Position

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

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

int	+	int	*	int	+	int

Tracking Our Position

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

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

int	|	+	int	*	int	+	int

Tracking Our Position

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

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

T	|	+	int	*	int	+	int

Tracking Our Position

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

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

T	|	+	int	*	int	+	int

Tracking Our Position

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

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

F	|	+	int	*	int	+	int

Tracking Our Position

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

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

F	|	+	int	*	int	+	int

Tracking Our Position

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

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

E	|	+	int	*	int	+	int

Tracking Our Position

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

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

E	|	+	int	*	int	+	int

Tracking Our Position

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

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

E	+	|	int	*	int	+	int

Tracking Our Position

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

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

E	+	|	int	*	int	+	int

Tracking Our Position

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

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

E	+	|	int	*	int	+	int

Tracking Our Position

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

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

E	+	|	int	*	int	+	int

Tracking Our Position

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

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

E	+	int	*	int	+	int

Tracking Our Position

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

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

E	+	T	*	int	+	int

Tracking Our Position

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

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

E	+	T	*	int	+	int

Tracking Our Position

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

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

E	+	F	|	*	int	+	int

Tracking Our Position

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

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

E	+	F	|	*	int	+	int

Tracking Our Position

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

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

E	+	F	*	|	int	+	int

Tracking Our Position

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

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

E	+	F	*	|	int	+	int

Tracking Our Position

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

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

E	+	F	*	int	|	+	int

Tracking Our Position

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

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

E	+	F	*	T	|	+	int

Tracking Our Position

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

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

E	+	F	*	T	|	+	int

Tracking Our Position

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

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

E	+	F	+	int

Tracking Our Position

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

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

E	+	F	|	+	int

Tracking Our Position

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

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

E | + int

Tracking Our Position

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

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

E		+	int

Tracking Our Position

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

S → · E
E → E + · F

E	+	|	int

Tracking Our Position

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

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

E	+	|	int

Tracking Our Position

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

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

E	+	|	int

Tracking Our Position

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

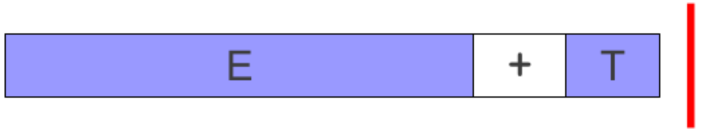
$S \rightarrow \cdot E$
$E \rightarrow E + \cdot F$
$F \rightarrow \cdot T$
$T \rightarrow \texttt{int} \cdot$

E	+	int

Tracking Our Position

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

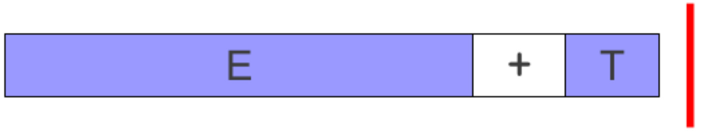
$S \rightarrow \cdot E$
$E \rightarrow E + \cdot F$
$F \rightarrow \cdot T$



Tracking Our Position

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

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



Tracking Our Position

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

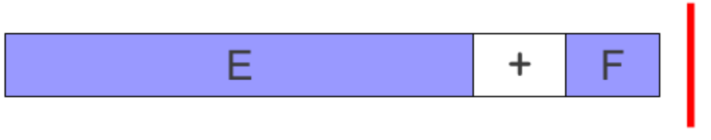
$S \rightarrow \cdot E$
$E \rightarrow E + \cdot F$

E	+	F

Tracking Our Position

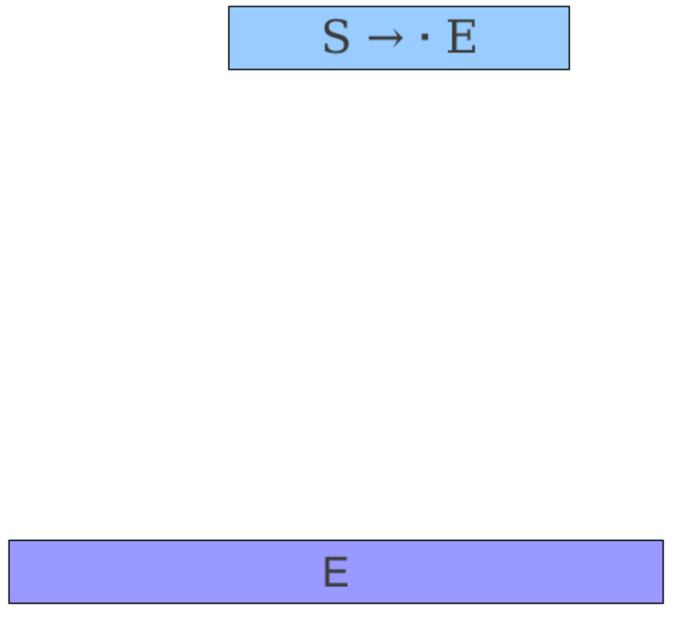
$S \rightarrow E$
$E \rightarrow F$
$E \rightarrow E + F$
$F \rightarrow F * T$
$F \rightarrow T$
$T \rightarrow \texttt{int}$
$T \rightarrow (E)$

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



Tracking Our Position

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



Tracking Our Position

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

$S \to E \cdot$

E

Generating Left-Hand Sides

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

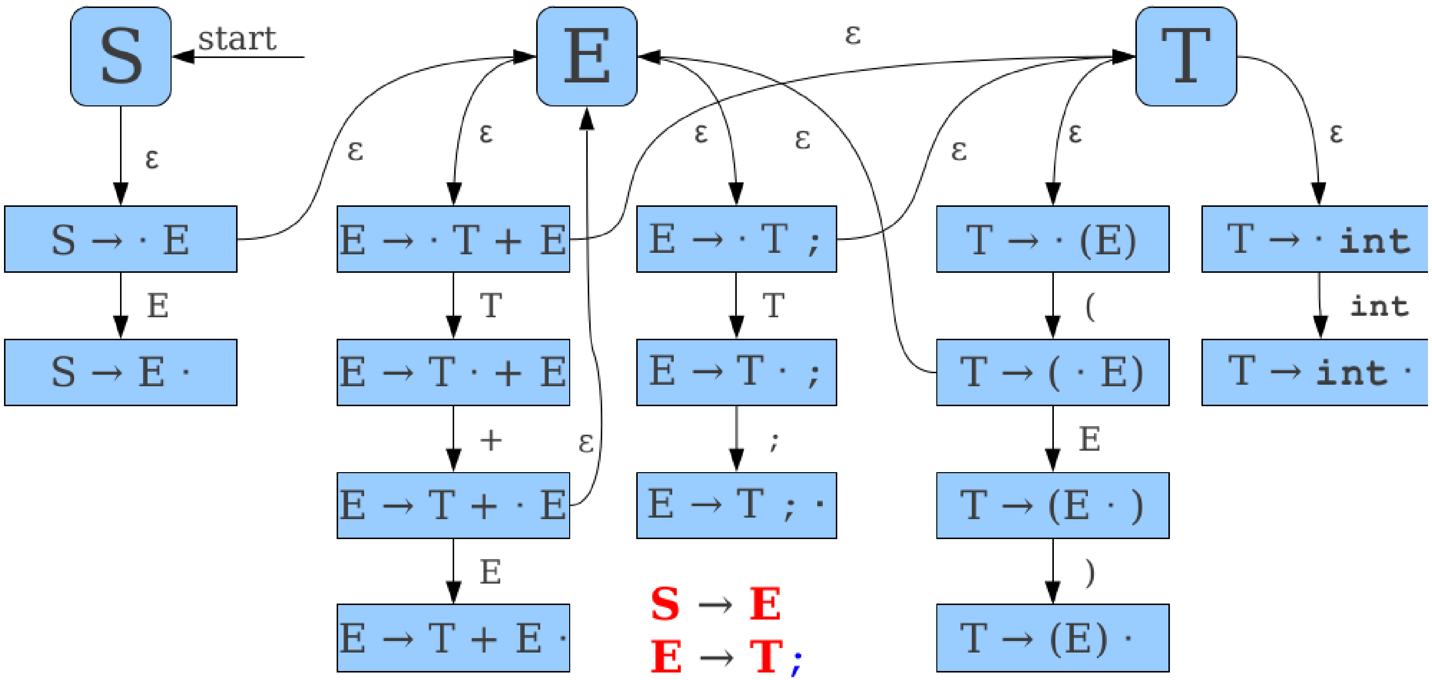
Recognizing Left-Hand Sides

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

An Important Result

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

An Automaton for Left Areas



$S \rightarrow E$

$E \rightarrow T$;

$E \rightarrow T + E$

$T \rightarrow$ `int`

$T \rightarrow$ (E)

Constructing the Automaton

- Create a state for each nonterminal.

- For each production $A \to \gamma$:
  - Construct states $A \to \alpha \cdot \omega$ for each possible way of splitting γ into two substrings α and ω.
  - Add transitions on x between $A \to \alpha \cdot x\omega$ and $A \to \alpha x \cdot \omega$.

- For each state $A \to \alpha \cdot B\omega$ for nonterminal B, add an ε-transition from $A \to \alpha \cdot B\omega$ to B.

Why This Matters

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

$$A \rightarrow \omega \cdot$$

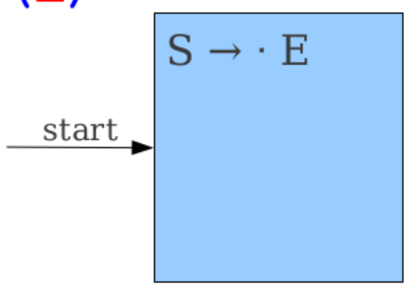
- Then we might be looking at a handle.
- This automaton can be used to discover possible handle locations!

Adding Determinism

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

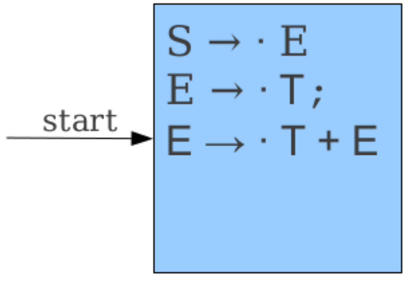
A Deterministic Automaton

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



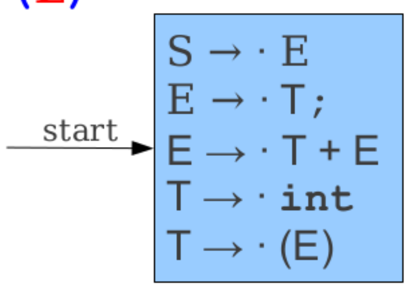
A Deterministic Automaton

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



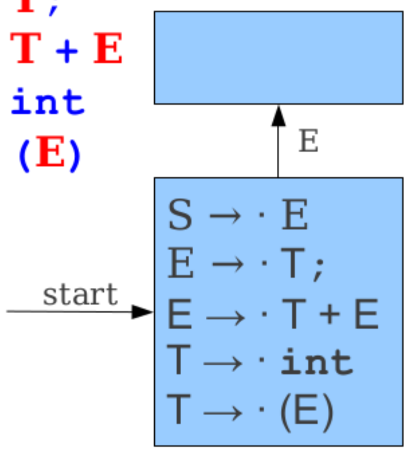
A Deterministic Automaton

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



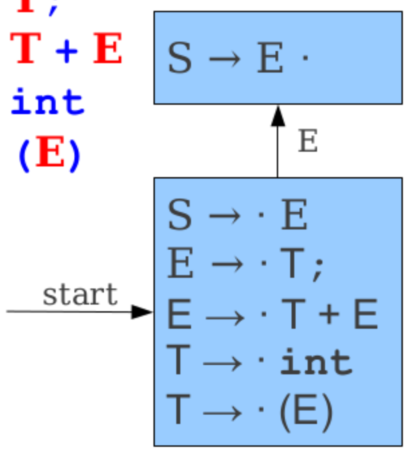
A Deterministic Automaton

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



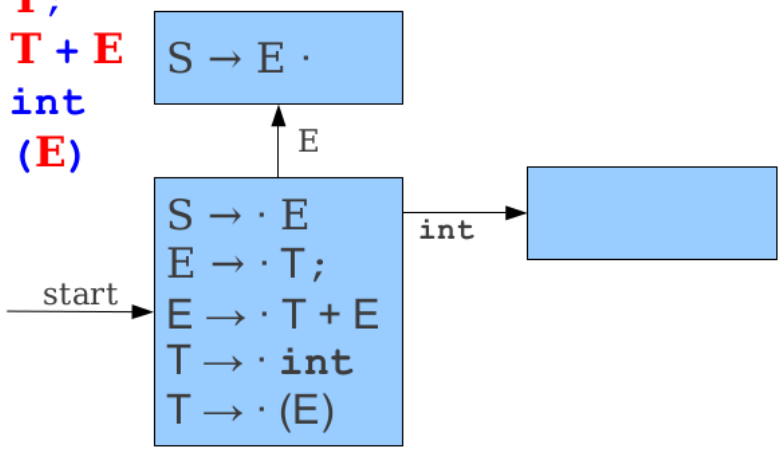
A Deterministic Automaton

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



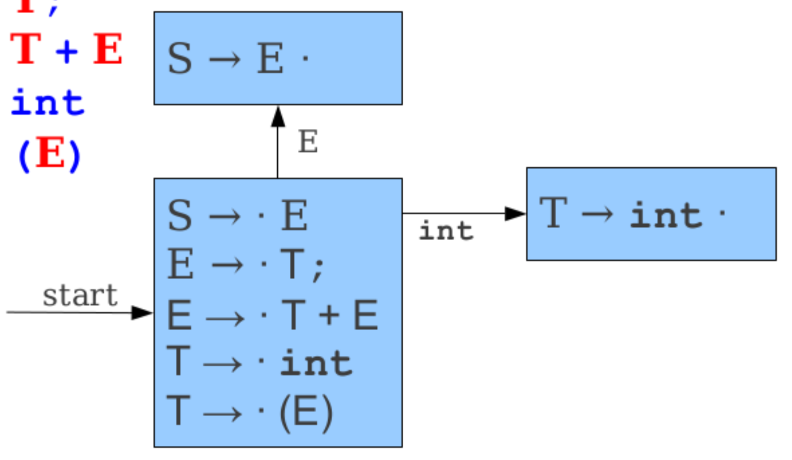
A Deterministic Automaton

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



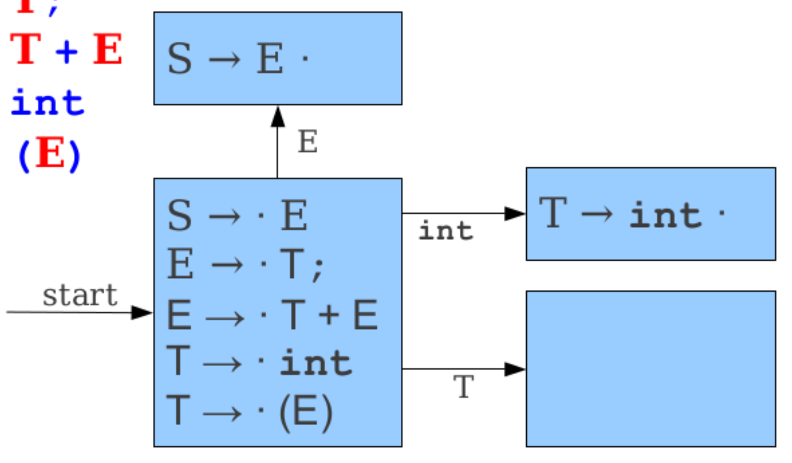
A Deterministic Automaton

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



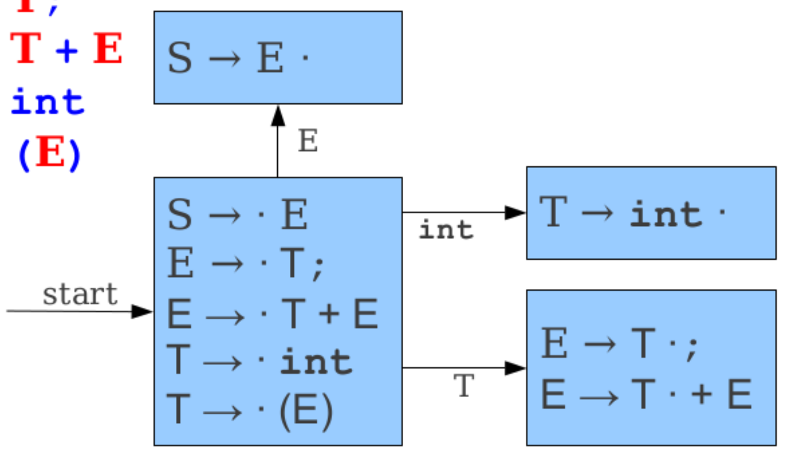
A Deterministic Automaton

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



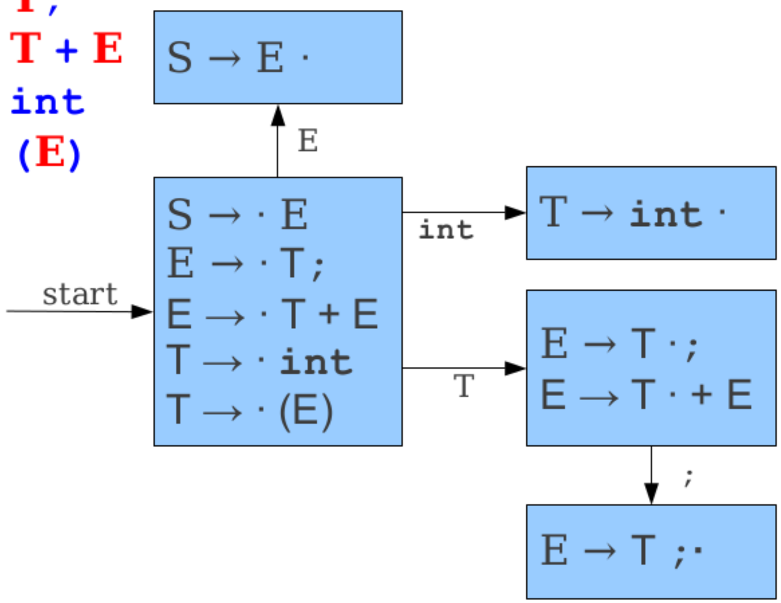
A Deterministic Automaton

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



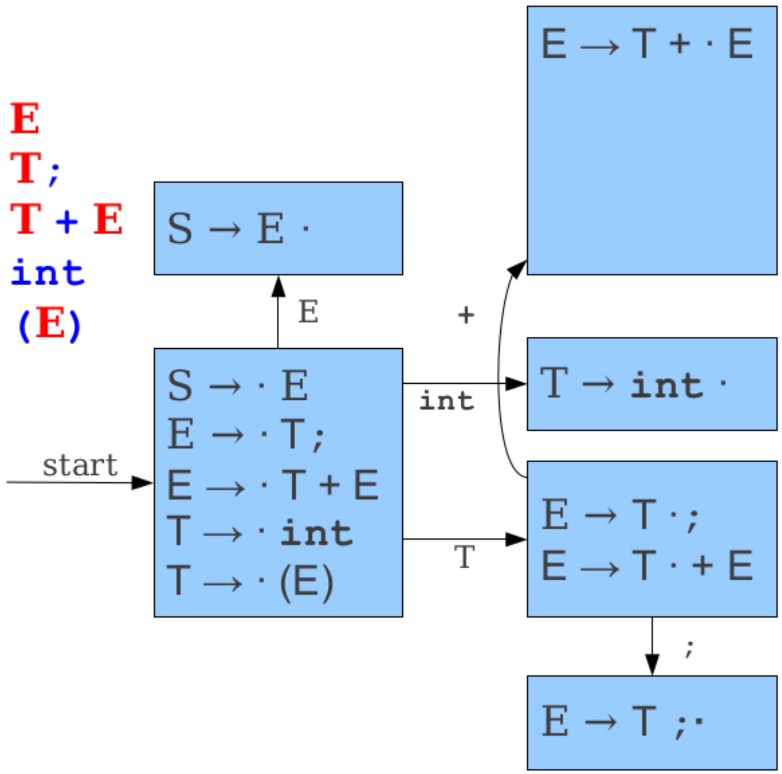
A Deterministic Automaton

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



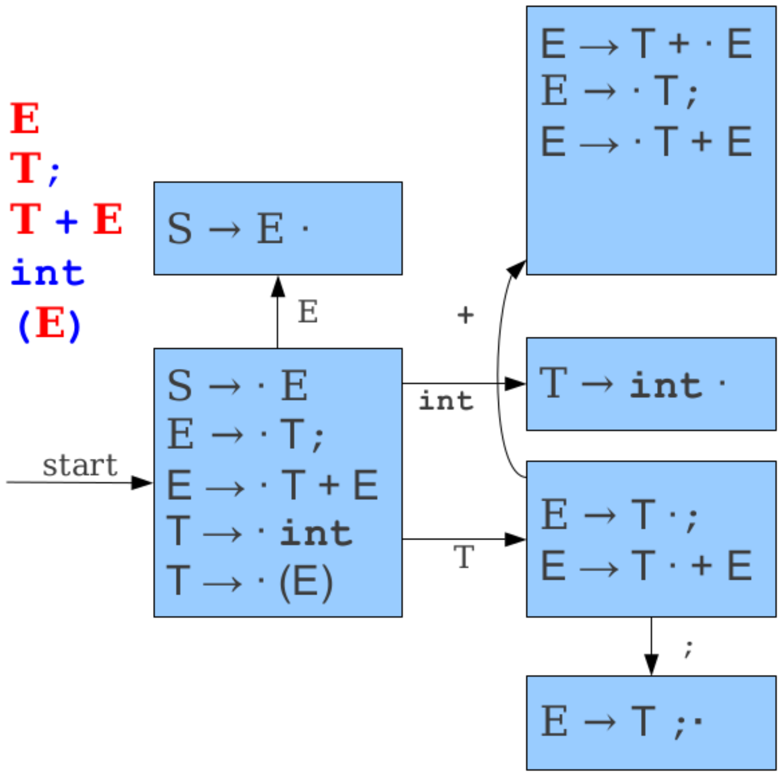
A Deterministic Automaton

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



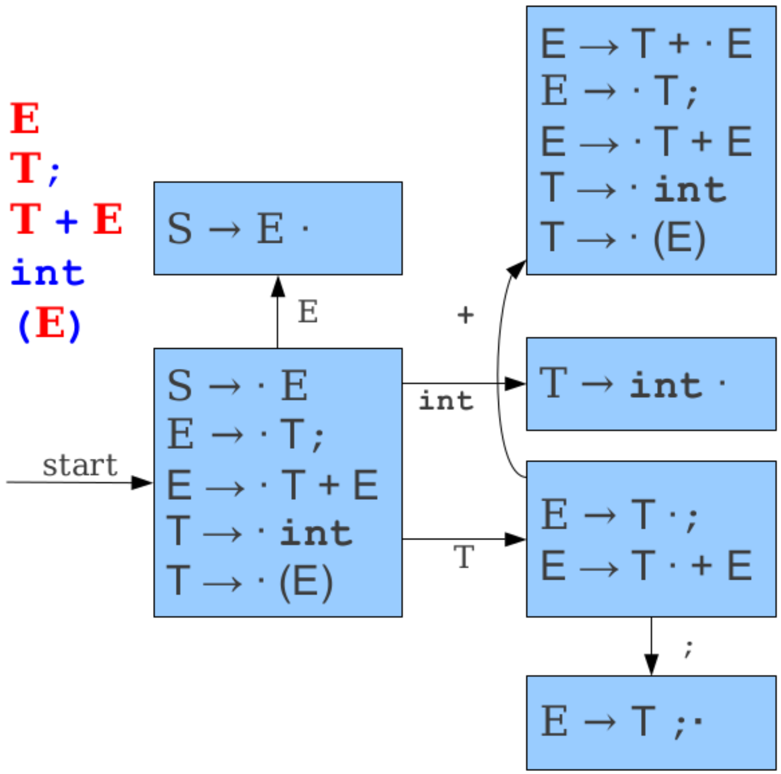
A Deterministic Automaton

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



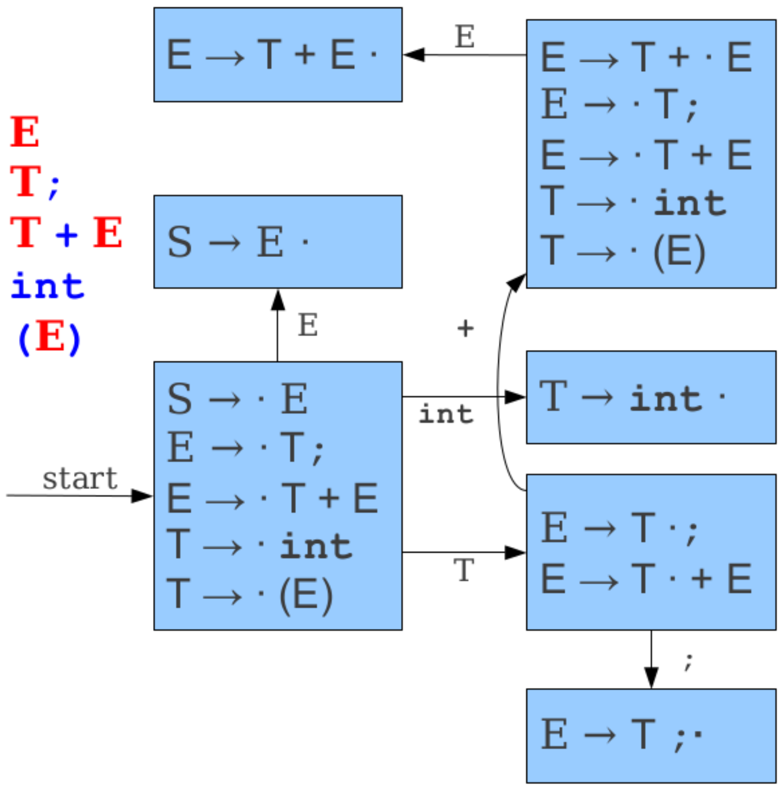
A Deterministic Automaton

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



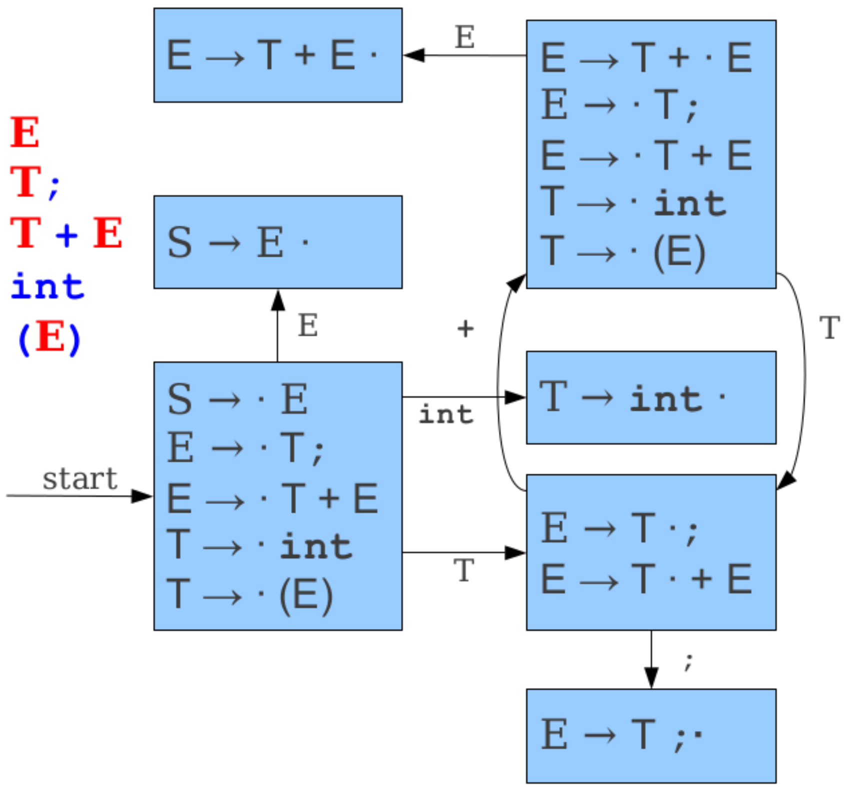
A Deterministic Automaton

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



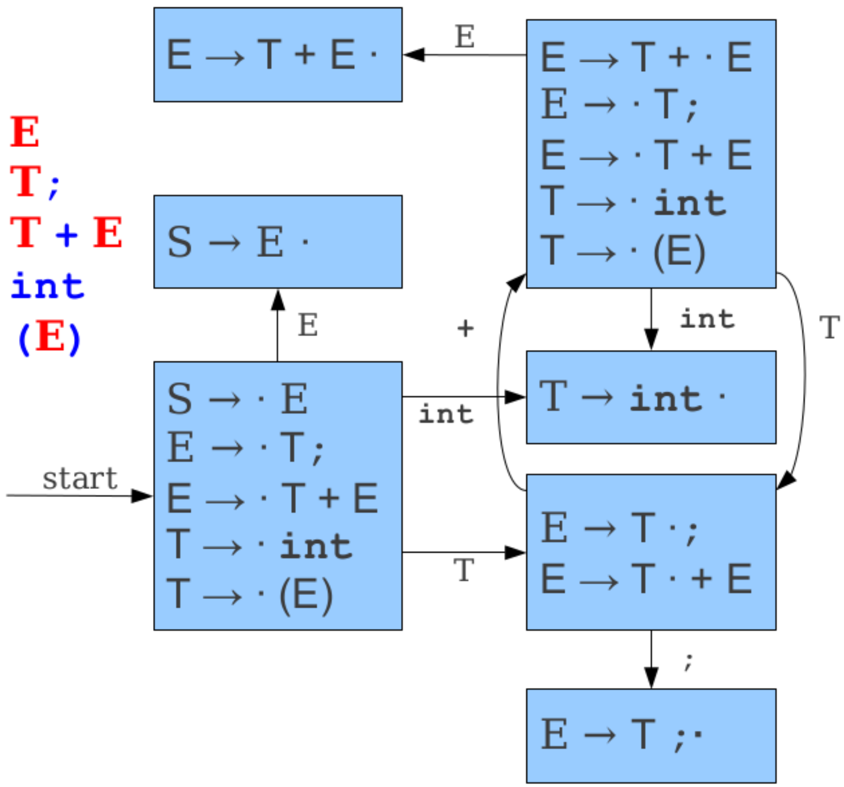
A Deterministic Automaton

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



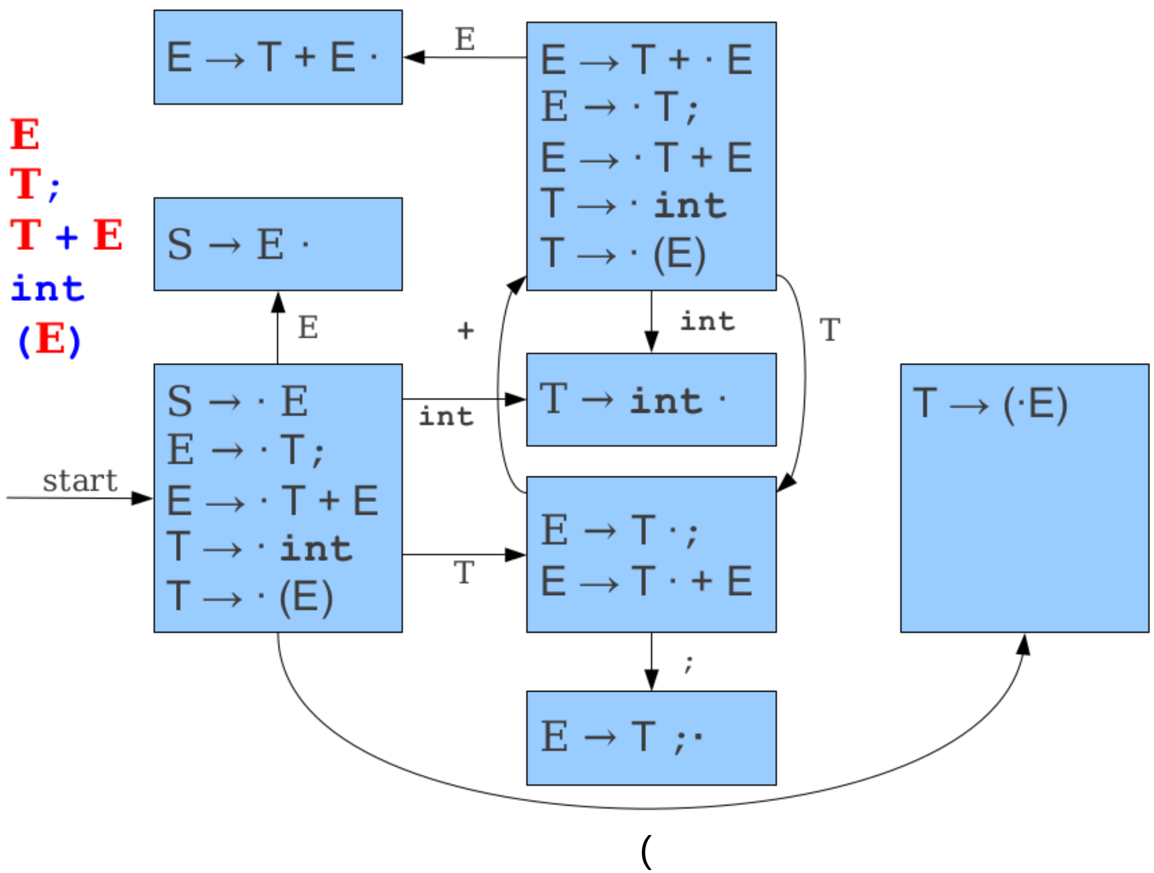
A Deterministic Automaton

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



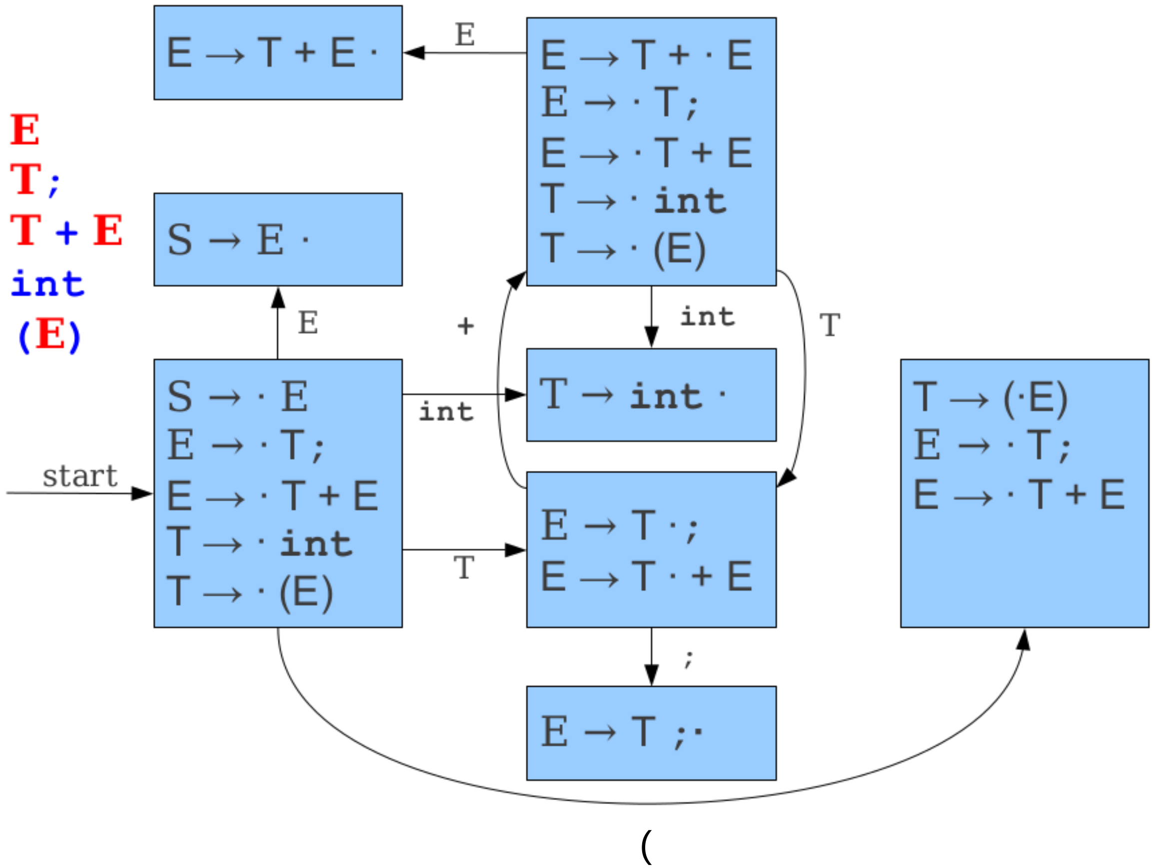
A Deterministic Automaton

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



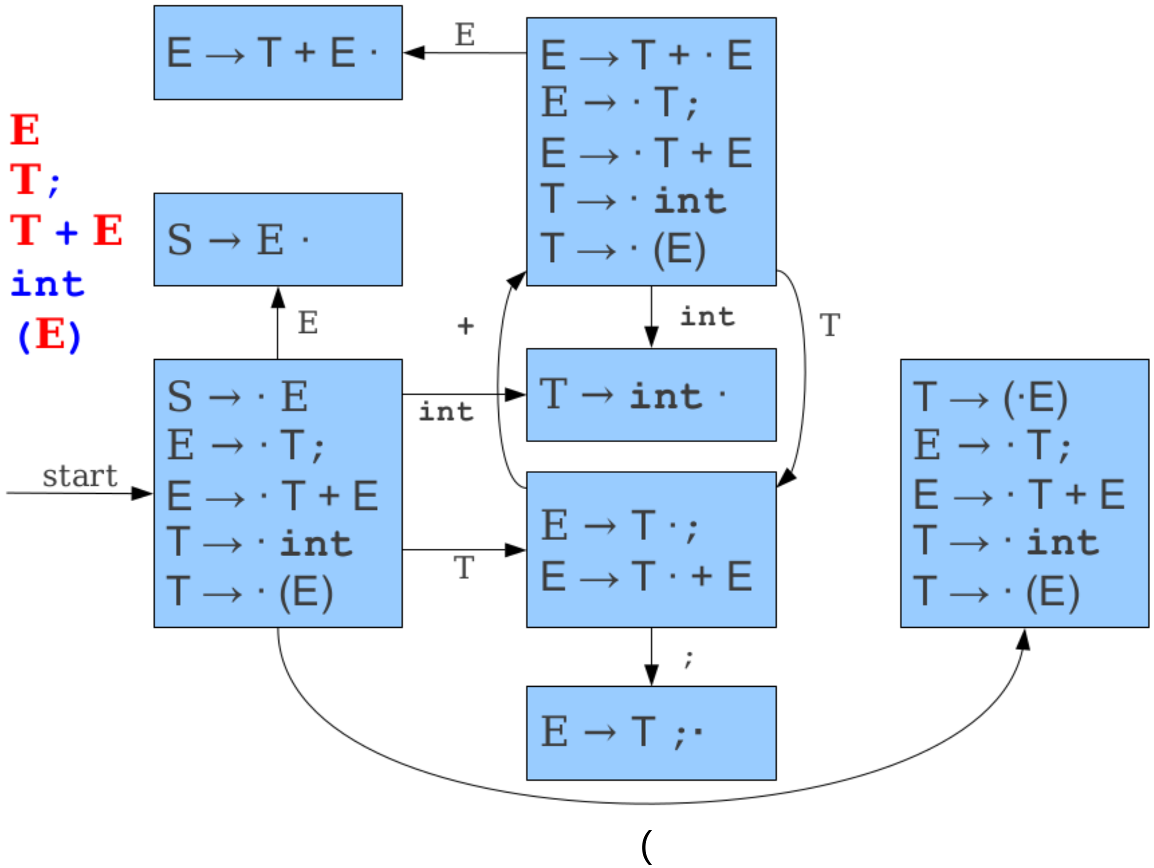
A Deterministic Automaton

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



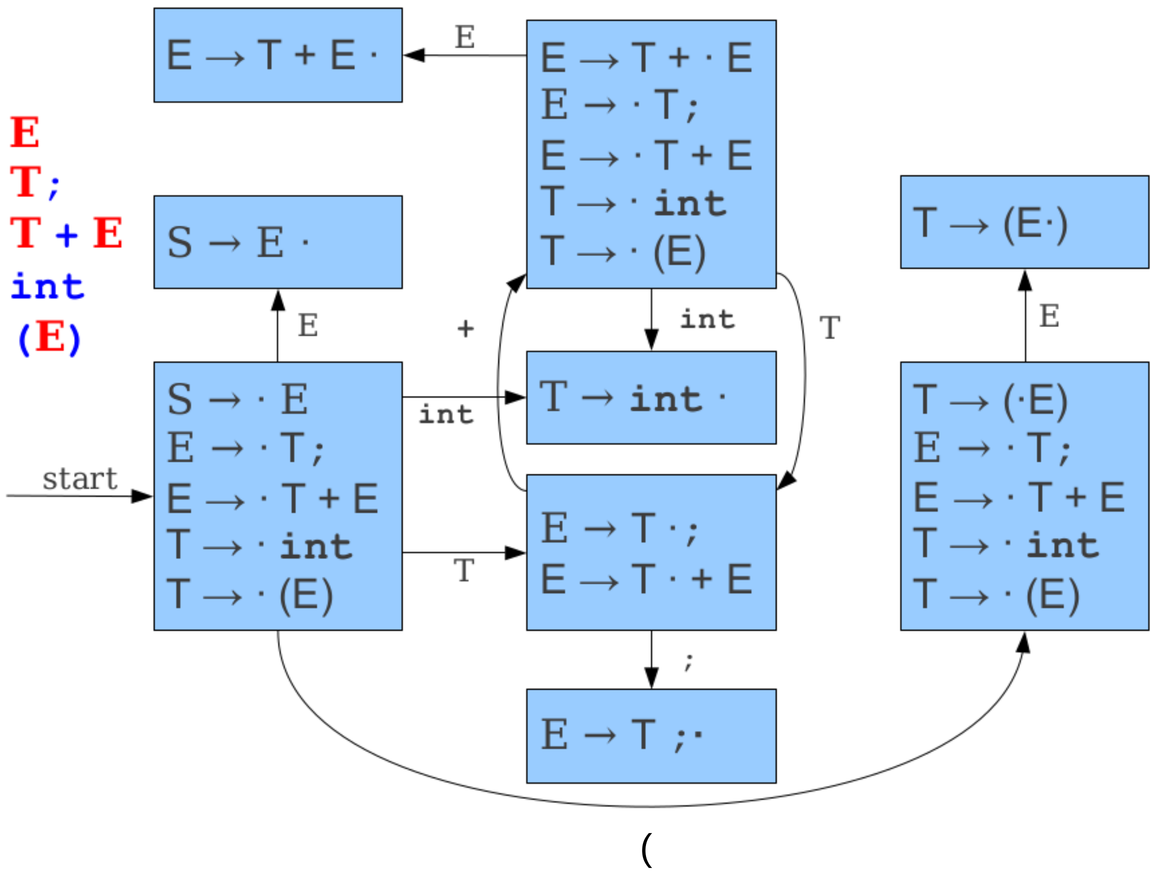
A Deterministic Automaton

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



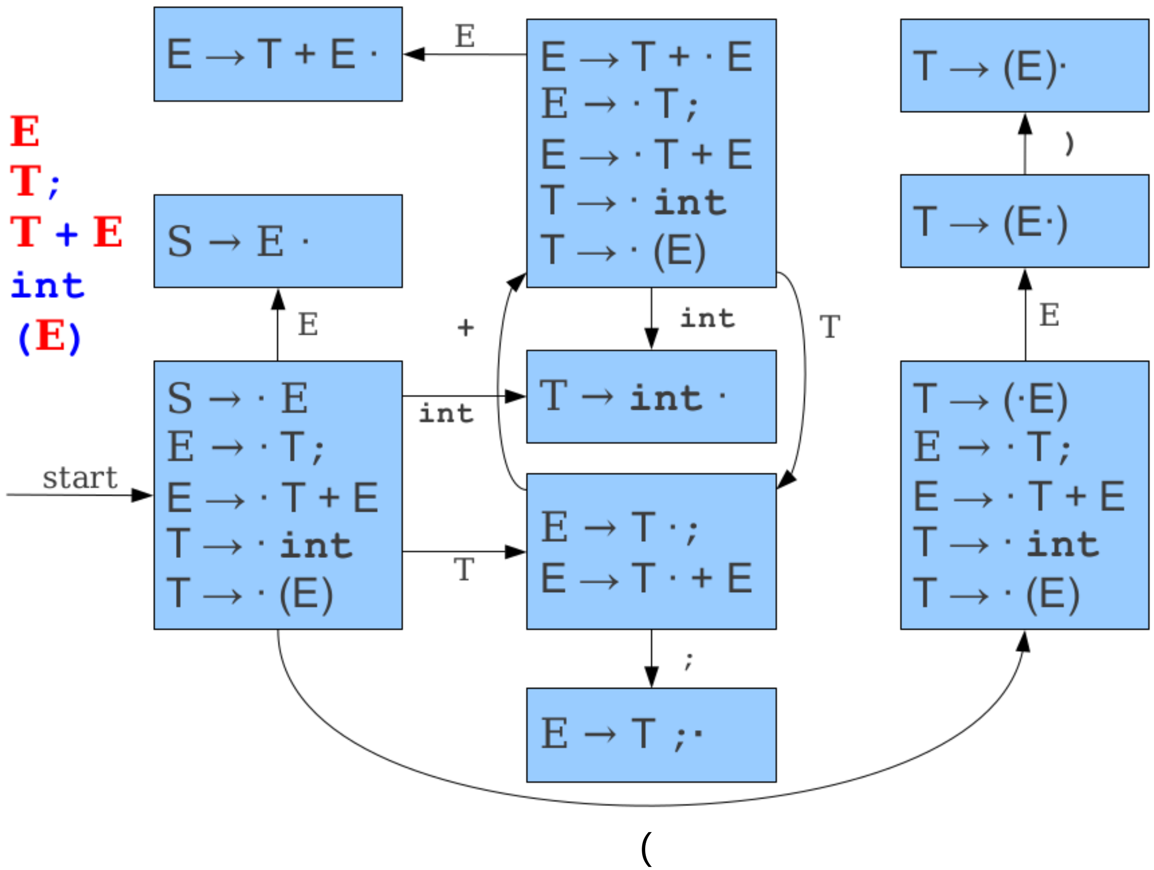
A Deterministic Automaton

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



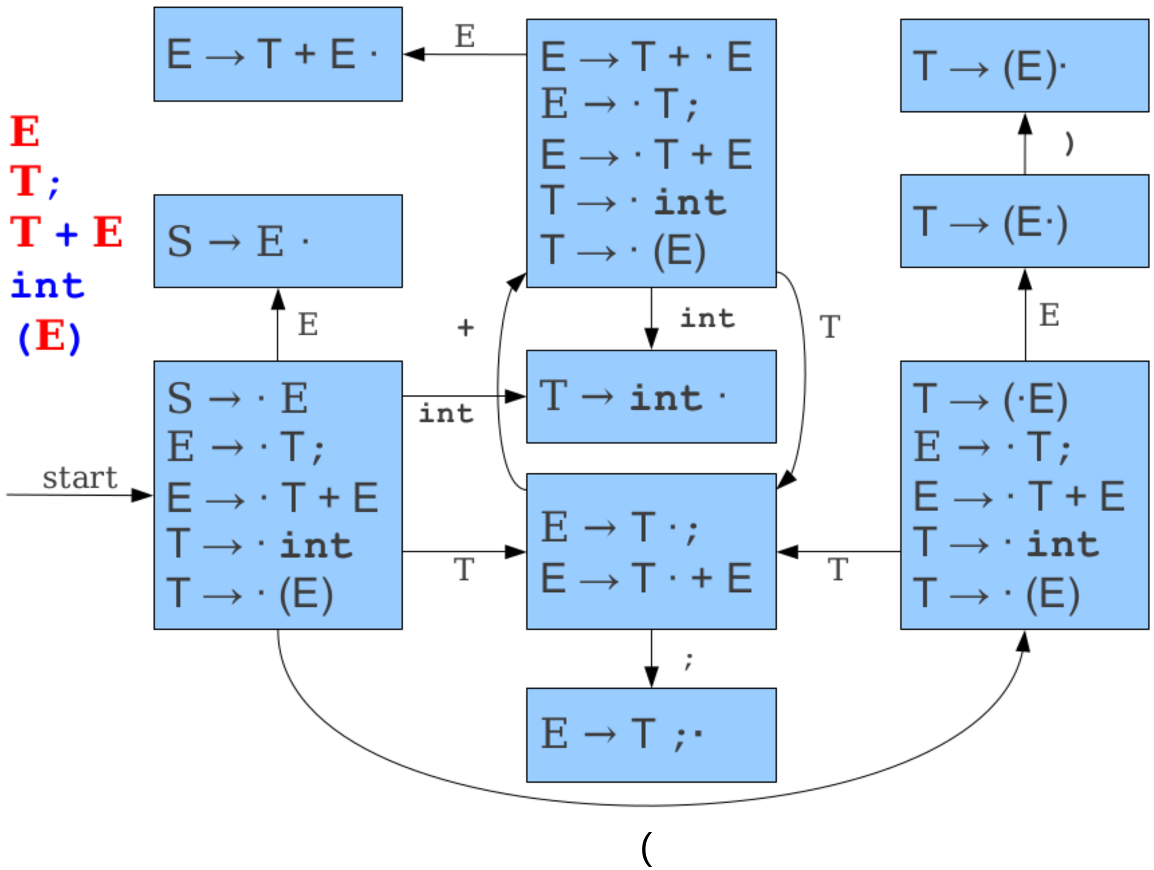
A Deterministic Automaton

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



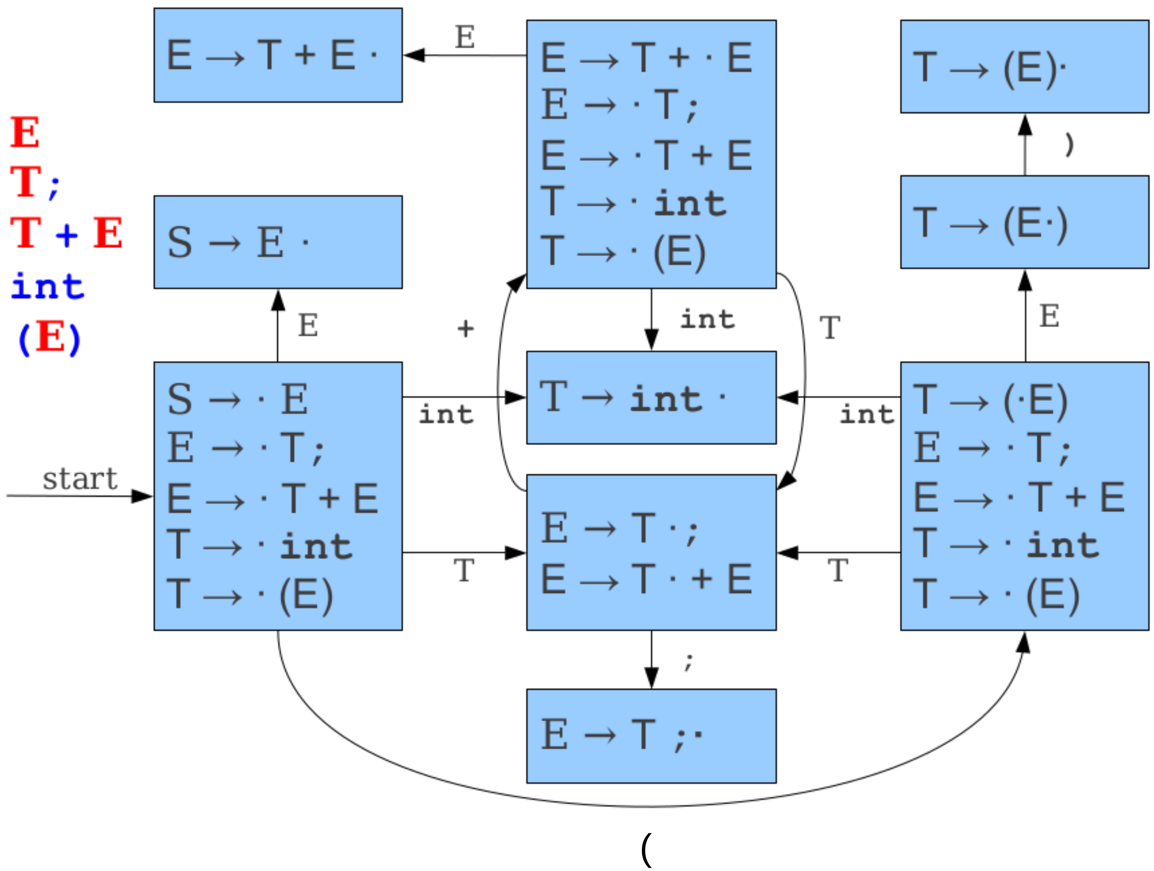
A Deterministic Automaton

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



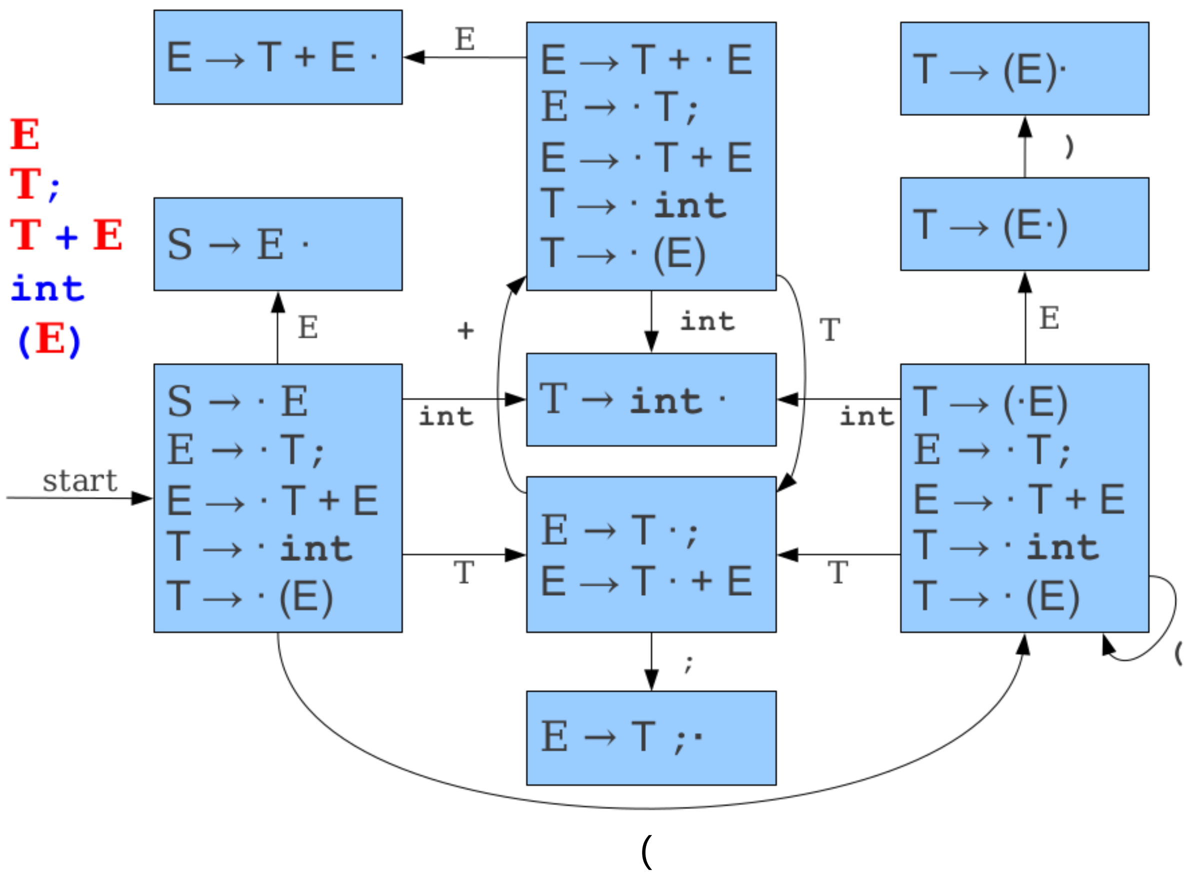
A Deterministic Automaton

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



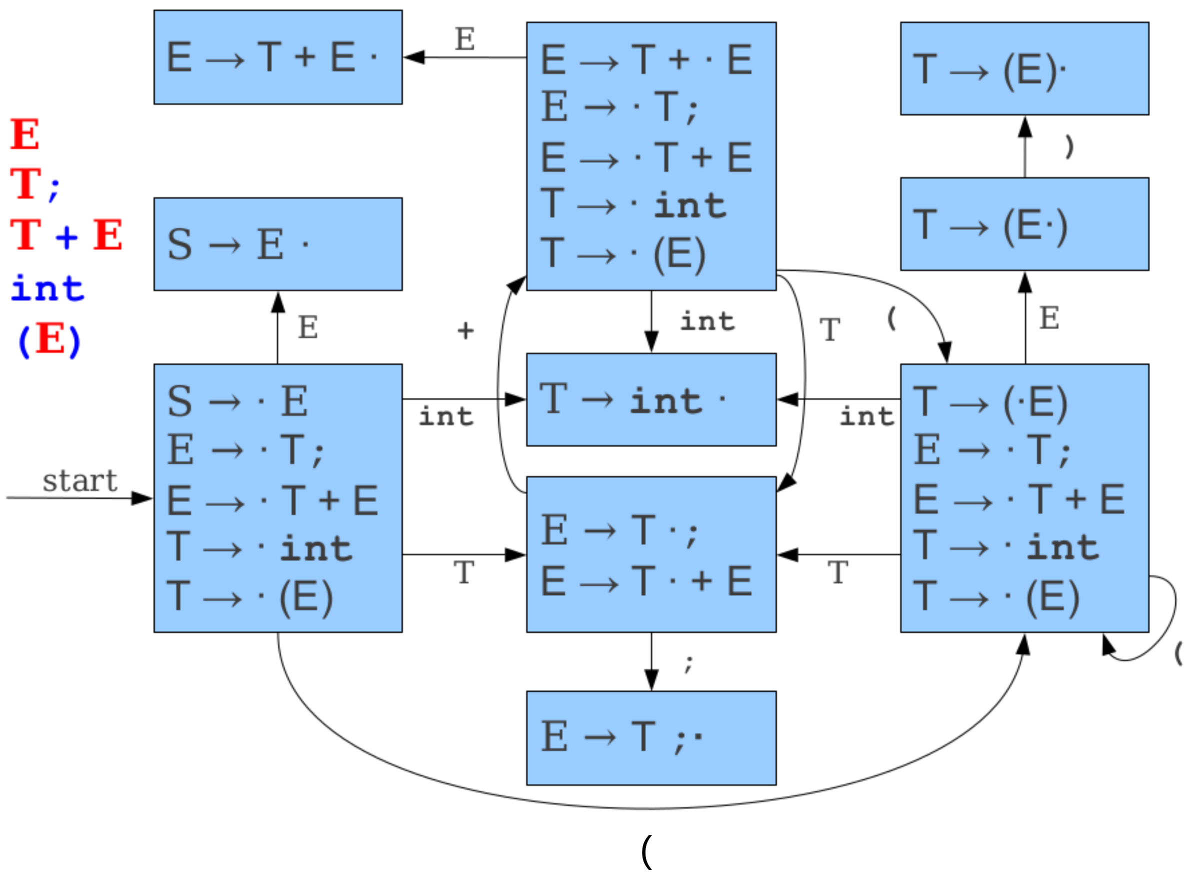
A Deterministic Automaton

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



A Deterministic Automaton

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



Constructing the Automaton II

- Begin in a state containing $S \to \cdot A$, where S is the augmented start symbol.
- Compute the closure of the state:
  - If $A \to \alpha \cdot B\omega$ is in the state, add $B \to \cdot \gamma$ to the state for each production $B \to \gamma$.
  - Yet another fixed-point iteration!
- 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 $A \to \alpha x \cdot \omega$
- This is equivalent to a subset construction on the NFA.

Handle-Finding Automata

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

Finding Handles: Main Questions

- 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 have found a possible handle, how do we confirm that it is correct?

Handle Recognition

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

Outline

- Introduction
- Shift-reduce Parsing
- **LR Parsing**
  - **LR(0)**
  - LR(1)
  - SLR
  - LALR
- Ambiguity
- Error-handling