# Bottom-Up Parsing

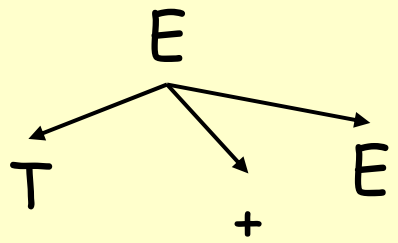
Lecture 9

# Outline

- Review LL parsing
- Shift-reduce parsing
- The LR parsing algorithm
- Constructing LR parsing tables

# Top-Down Parsing. Review

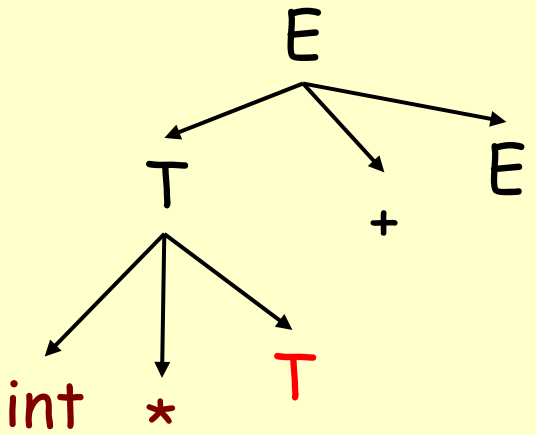
- Top-down parsing expands a parse tree from the start symbol to the leaves
  - Always expand the leftmost non-terminal




int * int + int

# Top-Down Parsing. Review

- Top-down parsing expands a parse tree from the start symbol to the leaves
  - Always expand the leftmost non-terminal

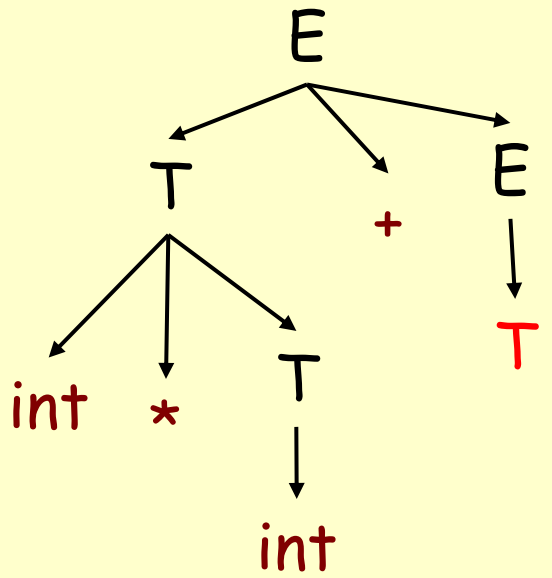



- The leaves at any point form a string $\beta A \gamma$
  - $\beta$ contains only terminals
  - The input string is $\beta b \delta$
  - The prefix $\beta$ matches
  - The next token is b

int * int + int

# Top-Down Parsing. Review

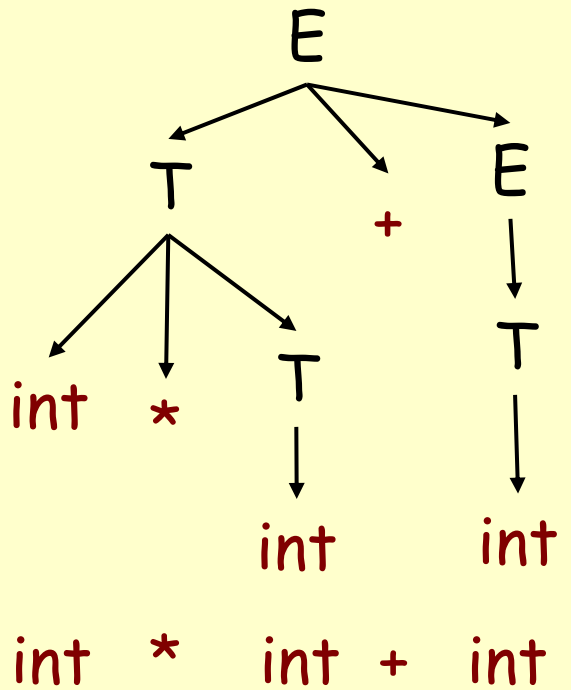
- Top-down parsing expands a parse tree from the start symbol to the leaves
  - Always expand the leftmost non-terminal




- The leaves at any point form a string $\beta A \gamma$
  - $\beta$ contains only terminals
  - The input string is $\beta b \delta$
  - The prefix $\beta$ matches
  - The next token is $b$

# Top-Down Parsing. Review

- Top-down parsing expands a parse tree from the start symbol to the leaves
  - Always expand the leftmost non-terminal




- The leaves at any point form a string $\beta A \gamma$
  - $\beta$ contains only terminals
  - The input string is $\beta b \delta$
  - The prefix $\beta$ matches
  - The next token is $b$

# Predictive Parsing. Review.

- A predictive parser is described by a table
  - For each non-terminal A and for each token b we specify a production $A \rightarrow \alpha$
  - When trying to expand A we use $A \rightarrow \alpha$ if b follows next

- Once we have the table
  - The parsing algorithm is simple and fast
  - No backtracking is necessary

# Notes on LL(1) Parsing Tables

- If any entry is multiply defined then G is not LL(1)
  - If G is ambiguous
  - If G is left recursive
  - If G is not left-factored
  - And in other cases as well
- There are tools that build LL(1) tables.
- Most programming language grammars are very nearly but not quite LL(1).

# What next?

- Good but we can do better with a more powerful parsing strategy that allows us to look at more of the input before deciding to do a reduction. We could try LL(2) but that would require a 3-D table. We have a better way.

# Bottom Up Parsing

# Bottom-Up Parsing

- Bottom-up parsing is more general than top-down parsing (handles more abstract languages)
  - And just as efficient
  - Builds on ideas in top-down parsing
  - Preferred method in practice

- Also called LR parsing
  - L means that tokens are read left to right
  - R means that it constructs a rightmost derivation -->

# An Introductory Example

- LR parsers don't need left-factored grammars and can also handle left-recursive grammars
- Consider the following grammar:

$$E \rightarrow E + ( E ) \mid int$$

  - Why is this not LL(1)?
- Consider the string: int + ( int ) + ( int )

# The Idea

- LR parsing *reduces* a string to the start symbol by inverting productions:

Str := input string of terminals

repeat

- Identify $\beta$ in **str** such that $A \rightarrow \beta$ is a production (i.e., str = $\alpha\ \beta\ \gamma$)
- Replace $\beta$ by A in str (i.e., str $\Rightarrow \alpha\ A\ \gamma$)

until str = S

# A Bottom-up Parse in Detail (1)

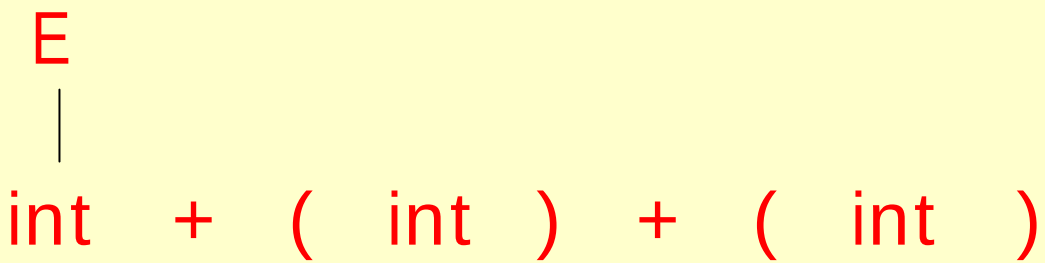
int + (int) + (int)

int + ( int ) + ( int )

# A Bottom-up Parse in Detail (2)

int + (int) + (int)

E + (int) + (int)

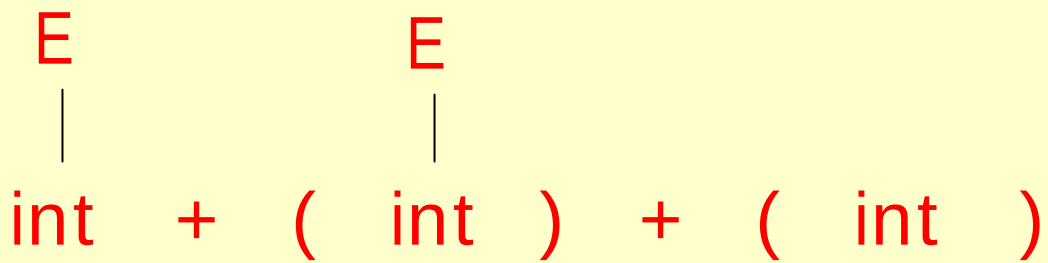


# A Bottom-up Parse in Detail (3)

int + (int) + (int)

E + (int) + (int)

E + (E) + (int)



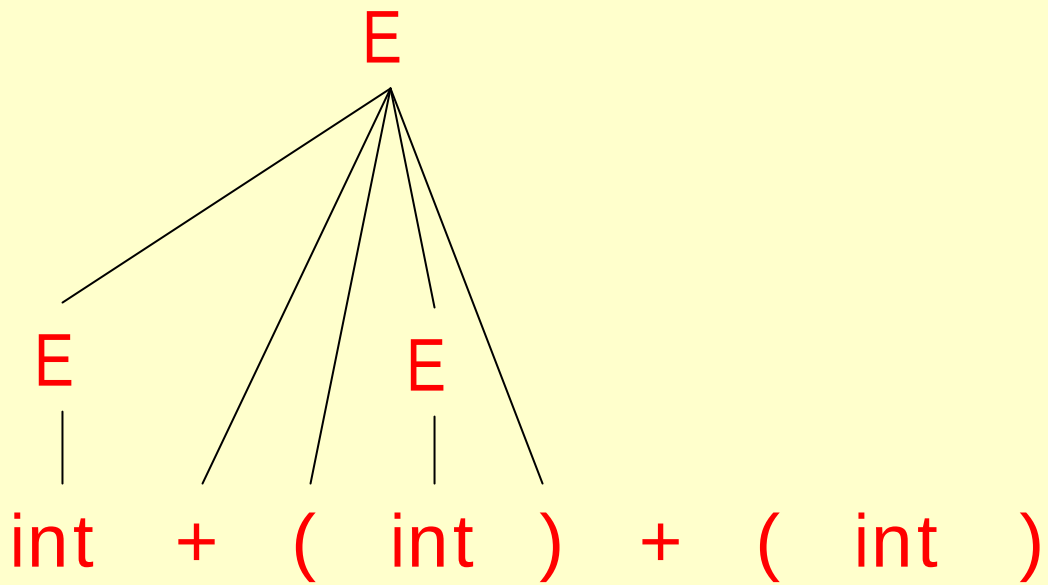
# A Bottom-up Parse in Detail (4)

int + (int) + (int)

E + (int) + (int)

E + (E) + (int)

E + (int)



# A Bottom-up Parse in Detail (5)

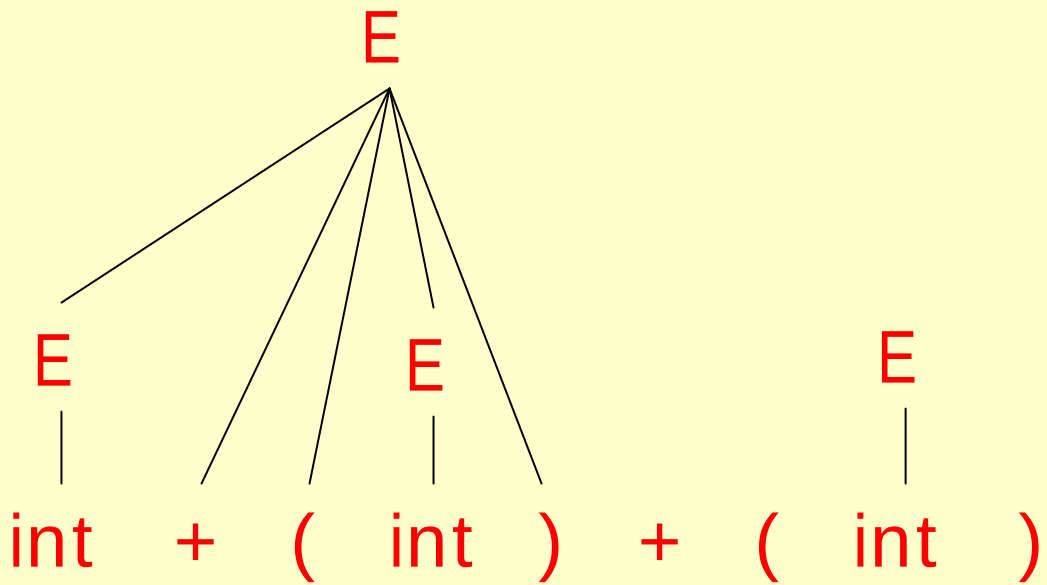
int + (int) + (int)

E + (int) + (int)

E + (E) + (int)

E + (int)

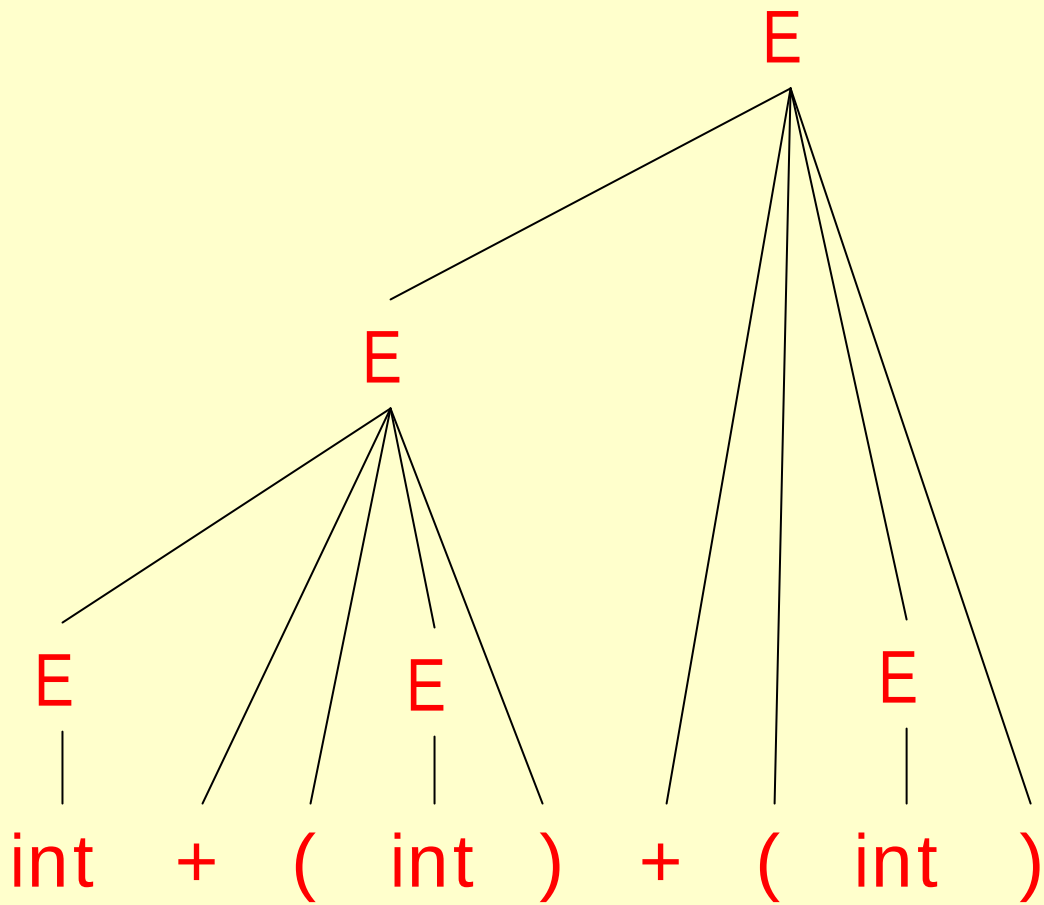
E + (E)



# A Bottom-up Parse in Detail (6)

int + (int) + (int)
E + (int) + (int)
E + (E) + (int)
E + (int)
E + (E)
E

A rightmost derivation in reverse



# Important Fact #1

Important Fact #1 about bottom-up parsing:

*An LR parser traces a rightmost derivation in reverse*

# Where Do Reductions Happen

Important Fact #1 has an interesting consequence:

- Let $\alpha\beta\gamma$ be a step of a bottom-up parse
- Assume the next reduction is by $A \rightarrow \beta$
- Then $\gamma$ is a string of terminals

Why? Because $\alpha A\gamma \rightarrow \alpha\beta\gamma$ is a step in a right-most derivation

# Notation

- Idea: Split string into two substrings
  - Right substring is as yet unexamined by parsing (a string of terminals)
  - Left substring has terminals and non-terminals
- The dividing point is marked by a | or ►
  - The ► is not part of the string
- Initially, all input is unexamined: ►$x_1 x_2 \ldots x_n$

# Shift-Reduce Parsing

- Bottom-up parsing uses only two kinds of actions:

*Shift*

*Reduce*

# Shift

*Shift:* Move ► one place to the right

- Shifts a terminal to the left string

E + (► int ) shifts to E + (int ► )

# Reduce

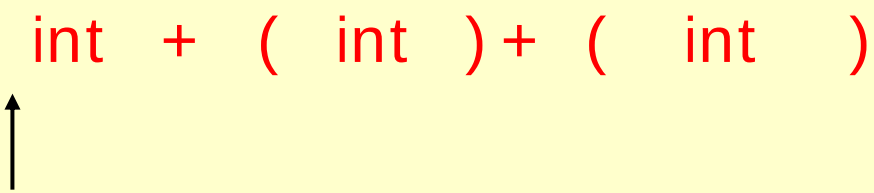
*Reduce:* Apply an inverse production at the right end of the left string

- If $E \rightarrow E + ( E )$ is a production, then

$E + (\underline{E + ( E )} \blacktriangleright )$ reduces to $E + (\underline{E} \blacktriangleright )$

# Shift-Reduce Example

- int + (int) + (int)$  shift



# Shift-Reduce Example

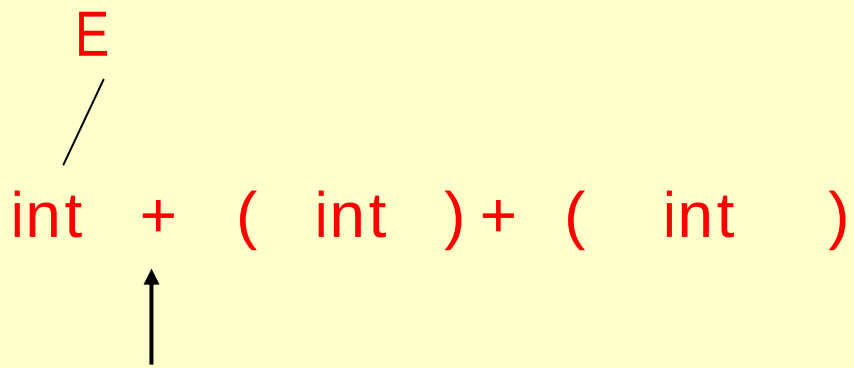
► int + (int) + (int)$  shift

int ► + (int) + (int)$  red. E → int

int + ( int ) + ( int )

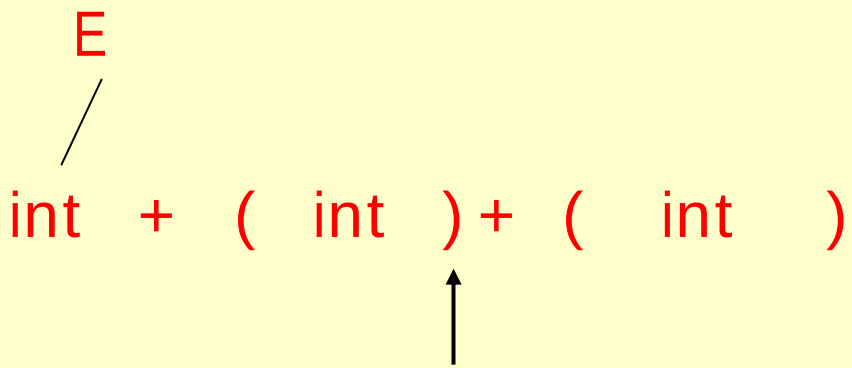
# Shift-Reduce Example

► int + (int) + (int)$ shift
int ► + (int) + (int)$ red. E → int
E ► + (int) + (int)$ shift 3 times



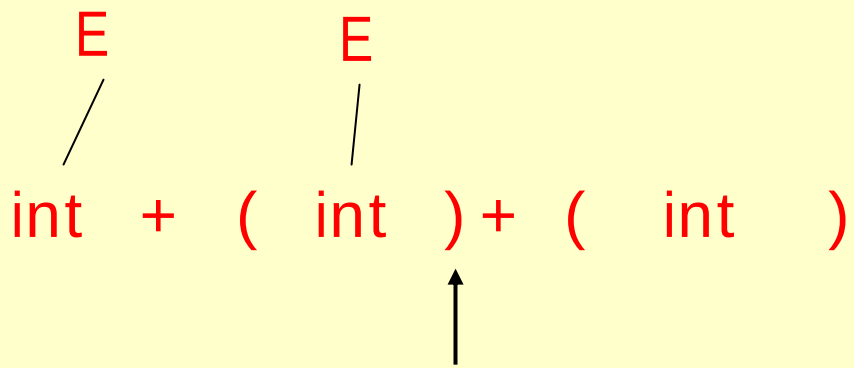
# Shift-Reduce Example

| | |
|---|---|
| ► int + (int) + (int)$ | shift |
| int ► + (int) + (int)$ | red. E → int |
| E ► + (int) + (int)$ | shift 3 times |
| E + (int ► ) + (int)$ | red. E → int |



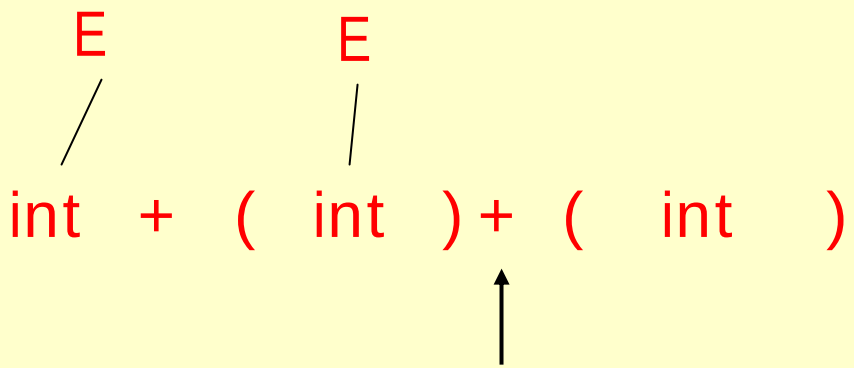
# Shift-Reduce Example

| | |
|---|---|
| ► int + (int) + (int)$ | shift |
| int ► + (int) + (int)$ | red. E → int |
| E ► + (int) + (int)$ | shift 3 times |
| E + (int ► ) + (int)$ | red. E → int |
| E + (E ► ) + (int)$ | shift |



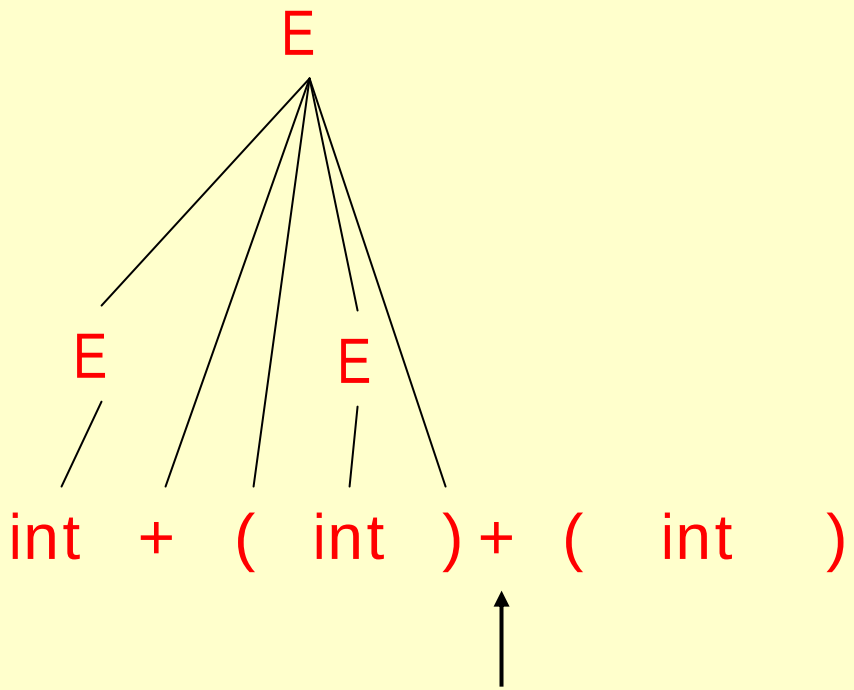
# Shift-Reduce Example

| | |
|---|---|
| ► int + (int) + (int)$ | shift |
| int ► + (int) + (int)$ | red. E → int |
| E ► + (int) + (int)$ | shift 3 times |
| E + (int ► ) + (int)$ | red. E → int |
| E + (E ► ) + (int)$ | shift |
| E + (E) ► + (int)$ | red. E → E + (E) |



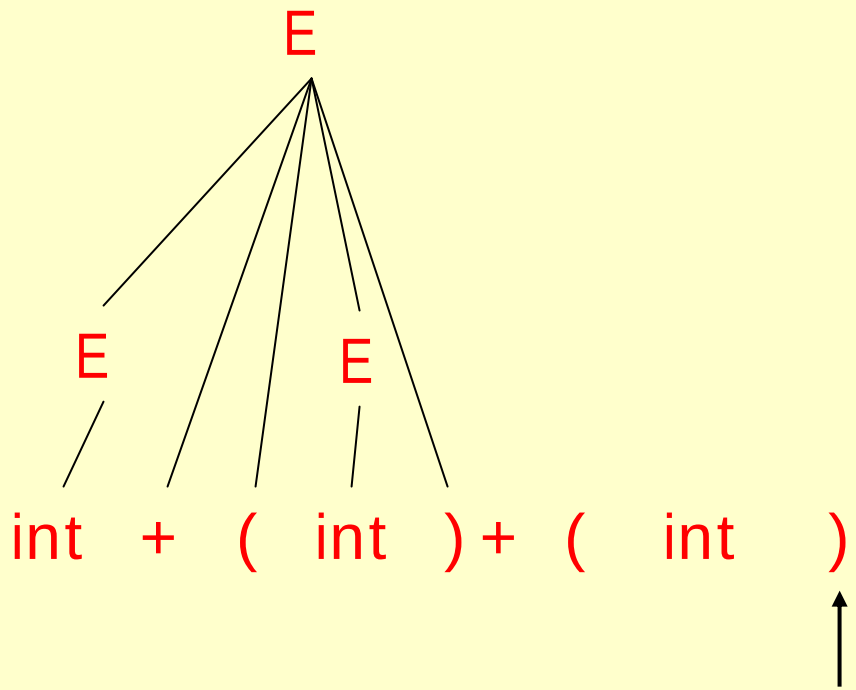
# Shift-Reduce Example

► int + (int) + (int)$ shift
int ► + (int) + (int)$ red. E → int
E ► + (int) + (int)$ shift 3 times
E + (int ► ) + (int)$ red. E→ int
E + (E ► ) + (int)$ shift
E + (E) ► + (int)$ red. E →E + (E)
E ► + (int)$ shift 3 times



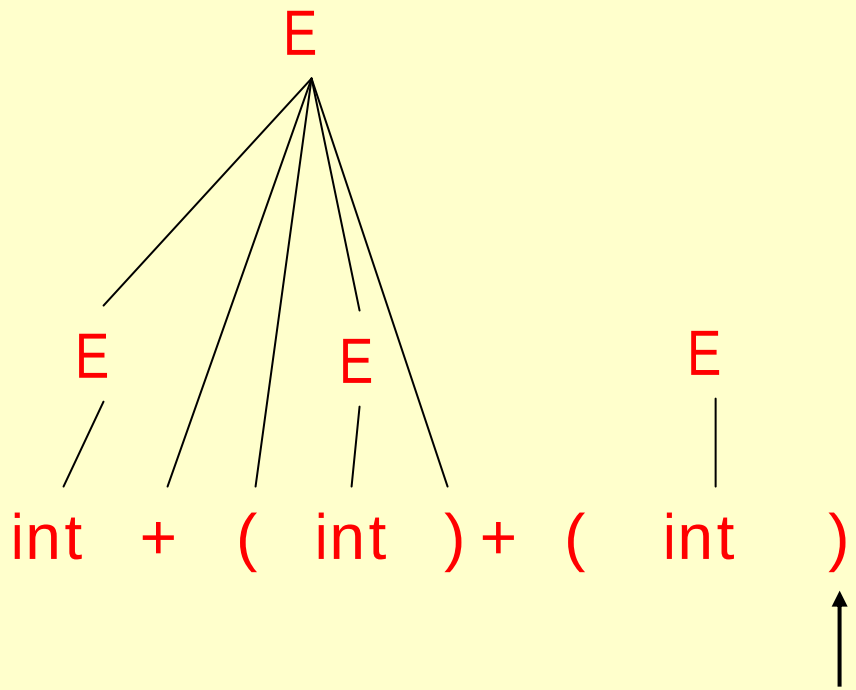
# Shift-Reduce Example

| | |
|---|---|
| ► int + (int) + (int)$ | shift |
| int ► + (int) + (int)$ | red. E → int |
| E ► + (int) + (int)$ | shift 3 times |
| E + (int ► ) + (int)$ | red. E → int |
| E + (E ► ) + (int)$ | shift |
| E + (E) ► + (int)$ | red. E → E + (E) |
| E ► + (int)$ | shift 3 times |
| E + (int ► )$ | red. E → int |



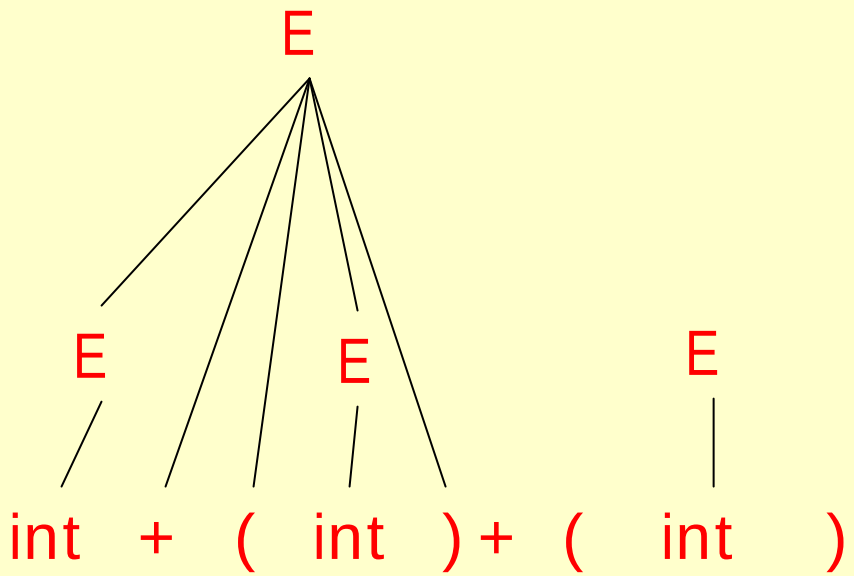
# Shift-Reduce Example

| | |
|---|---|
| ▶ int + (int) + (int)$ | shift |
| int ▶ + (int) + (int)$ | red. E → int |
| E ▶ + (int) + (int)$ | shift 3 times |
| E + (int ▶ ) + (int)$ | red. E → int |
| E + (E ▶ ) + (int)$ | shift |
| E + (E) ▶ + (int)$ | red. E →E + (E) |
| E ▶ + (int)$ | shift 3 times |
| E + (int ▶ )$ | red. E →int |
| E + (E ▶ )$ | shift |



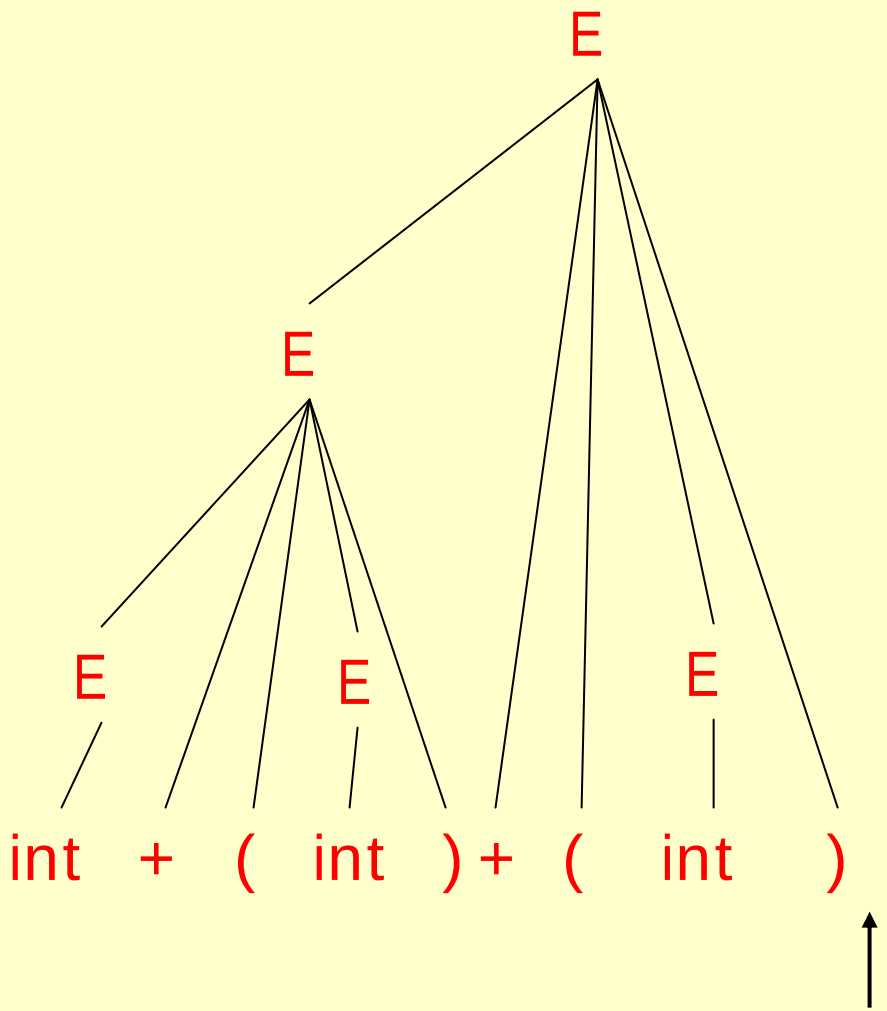
# Shift-Reduce Example

| | |
|---|---|
| ▶ int + (int) + (int)$ | shift |
| int ▶ + (int) + (int)$ | red. E → int |
| E ▶ + (int) + (int)$ | shift 3 times |
| E + (int ▶ ) + (int)$ | red. E → int |
| E + (E ▶ ) + (int)$ | shift |
| E + (E) ▶ + (int)$ | red. E →E + (E) |
| E ▶ + (int)$ | shift 3 times |
| E + (int ▶ )$ | red. E → int |
| E + (E ▶ )$ | shift |
| E + (E) ▶ $ | red. E → E + (E) |



# Shift-Reduce Example

| | |
|---|---|
| ▶ int + (int) + (int)$ | shift |
| int ▶ + (int) + (int)$ | red. E → int |
| E ▶ + (int) + (int)$ | shift 3 times |
| E + (int ▶ ) + (int)$ | red. E → int |
| E + (E ▶ ) + (int)$ | shift |
| E + (E) ▶ + (int)$ | red. E →E + (E) |
| E ▶ + (int)$ | shift 3 times |
| E + (int ▶ )$ | red. E → int |
| E + (E ▶ )$ | shift |
| E + (E) ▶ $ | red. E → E + (E) |
| E ▶ $ | accept |



# The Stack

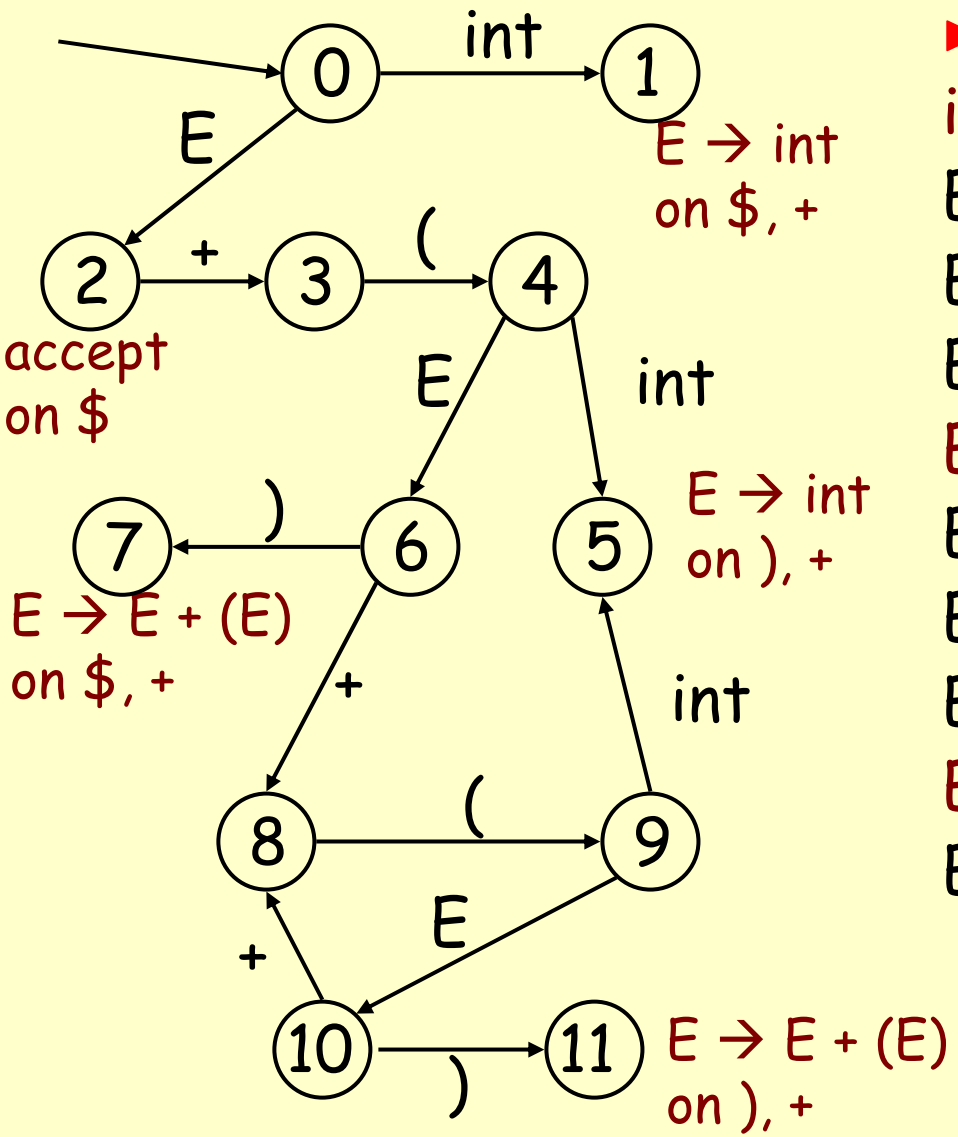
- Left string can be implemented by a stack
  - Top of the stack is the ►
- Shift pushes a terminal on the stack
- Reduce pops 0 or more symbols off of the stack (production rhs) and pushes a non-terminal on the stack (production lhs)

# Key Issue: When to Shift or Reduce?

- Idea: use a finite automaton (DFA) to decide when to shift or reduce
  - The input is the stack
  - The language consists of terminals and non-terminals

- We run the DFA on the stack and we examine the resulting state X and the token tok after ▸
  - If X has a transition labeled tok then shift
  - If X is labeled with "$A \rightarrow \beta$ on tok" then reduce

# LR(1) Parsing. An Example



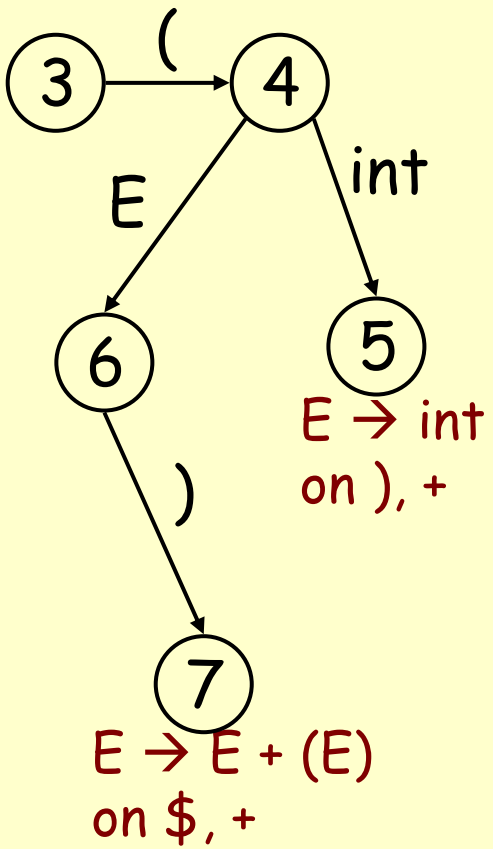

| | |
|---|---|
| ▸ int + (int) + (int)$ | shift |
| int ▸ + (int) + (int)$ | E → int |
| E ▸ + (int) + (int)$ | shift(x3) |
| E + (int ▸ ) + (int)$ | E → int |
| E + (E ▸ ) + (int)$ | shift |
| E + (E) ▸ + (int)$ | E→E+(E) |
| E ▸ + (int)$ | shift (x3) |
| E + (int ▸ )$ | E → int |
| E + (E ▸ )$ | shift |
| E + (E) ▸ $ | E → E+(E) |
| E ▸ $ | accept |

# Representing the DFA

- Parsers represent the DFA as a 2D table
  - Recall table-driven lexical analysis
- Lines correspond to DFA states
- Columns correspond to terminals and non-terminals
- Typically columns are split into:
  - Those for terminals: action table
  - Those for non-terminals: goto table

# Representing the DFA. Example

- The table for a fragment of our DFA:




| | int | + | ( | ) | $ | E |
|---|---|---|---|---|---|---|
| … | | | | | | |
| 3 | | | s4 | | | |
| 4 | s5 | | | | | g6 |
| 5 | | $r_{E\rightarrow int}$ | | $r_{E\rightarrow int}$ | | |
| 6 | s8 | | s7 | | | |
| 7 | | $r_{E\rightarrow E+(E)}$ | | | $r_{E\rightarrow E+(E)}$ | |
| … | | | | | | |

# The LR Parsing Algorithm

- After a shift or reduce action we rerun the DFA on the entire stack
  - This is wasteful, since most of the work is repeated

- Faster: Remember for each stack element to which state it brings the DFA. Extra memory means it is no longer a DFA, but memory is cheap.

- LR parser maintains a stack

$$\langle \text{sym}_1, \text{state}_1 \rangle \ldots \langle \text{sym}_n, \text{state}_n \rangle$$

$\text{state}_k$ is the final state of the DFA on $\text{sym}_1 \ldots \text{sym}_k$

# The LR Parsing Algorithm

```
Let I = w$ be initial input
Let j = 0
Let DFA state 0 be the start state
Let stack = ⟨ dummy, 0 ⟩
    repeat
        case action[top_state(stack), I[j]] of
                shift k:  push ⟨ I[j++], k ⟩
                reduce X → A:
                    pop |A| pairs,
                    push ⟨X, Goto[top_state(stack), X]⟩
                accept: halt normally
                error: halt and report error
```

# LR Parsing Notes

- Can be used to parse more grammars than LL
- Most programming languages grammars are LR
- Can be described as a simple table
- There are tools for building the table
- How is the table constructed?