

# Introduction to Bottom-Up Parsing

# Outline

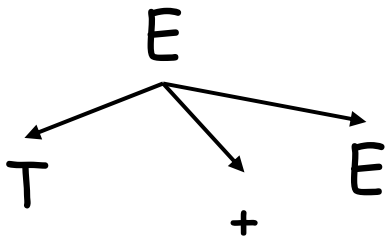
---

- 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

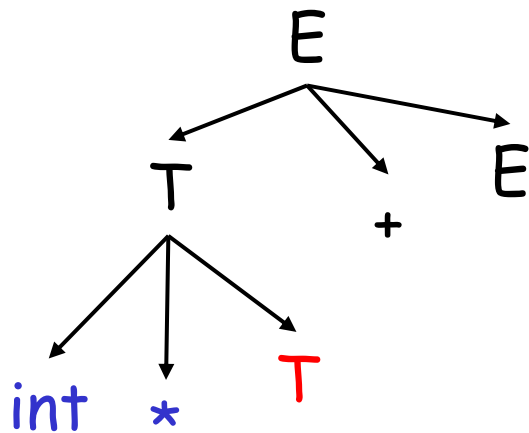


int \* int + int

$E \rightarrow T + E \mid T$   
 $T \rightarrow (E) \mid \text{int} \mid \text{int} * T$

# 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

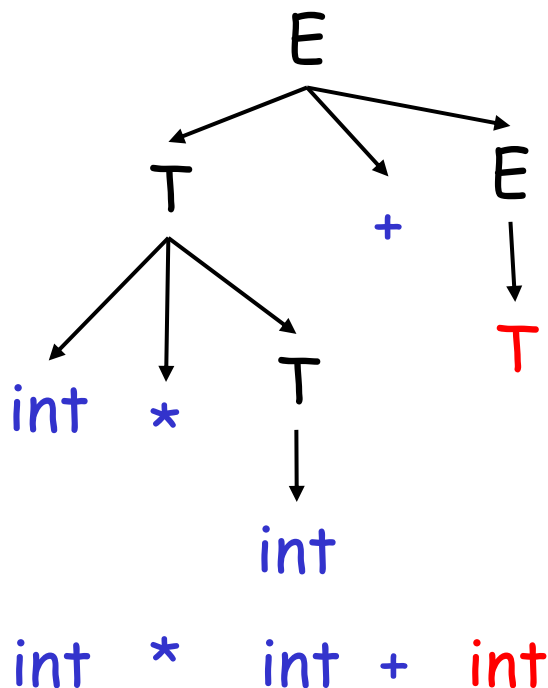
- The leaves at any point form a string  $\beta A \gamma$ 
  - $\beta$  contains only terminals
  - The input string is  $\beta b \delta$
  - The prefix  $\beta$  matches
  - The next token is  $b$

$E \rightarrow T + E \mid T$   
 $T \rightarrow (E) \mid \text{int} \mid \text{int} * T$

# Top-Down Parsing: Review

---

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

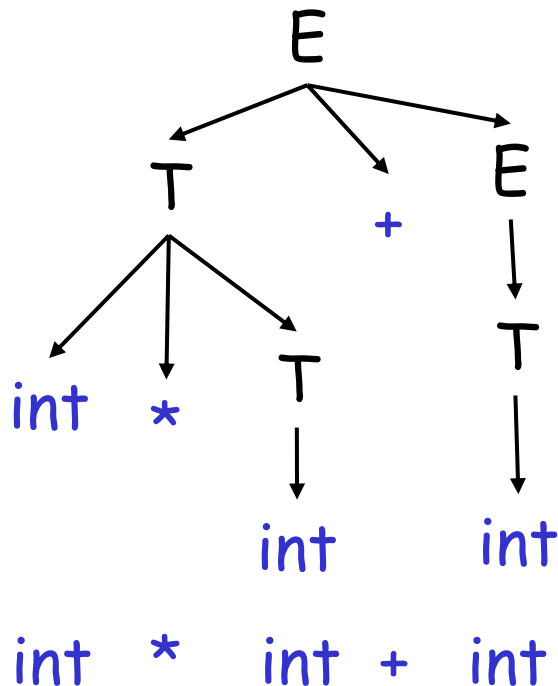


- The leaves at any point form a string  $\beta A \gamma$ 
  - $\beta$  contains only terminals
  - The input string is  $\beta b \delta$
  - The prefix  $\beta$  matches
  - The next token is  $b$

# 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  $b$  is the token that follows next
- Once we have the table
  - The parsing algorithm is simple and fast
  - No backtracking is necessary

# Constructing Predictive Parsing Tables

---

Consider the state  $S \rightarrow^* \beta A \gamma$

- With  $b$  the next token
- Trying to match  $\beta b \delta$

There are two possibilities:

1. Token  $b$  belongs to an expansion of  $A$ 
  - Any  $A \rightarrow \alpha$  can be used if  $b$  can start a string derived from  $\alpha$
  - We say that  $b \in \text{First}(\alpha)$

Or...



## Constructing Predictive Parsing Tables (Cont.)

---

2. Token  $b$  does not belong to an expansion of  $A$
- The expansion of  $A$  is empty and  $b$  belongs to an expansion of  $\gamma$
  - Means that  $b$  can appear after  $A$  in a derivation of the form  $S \rightarrow^* \beta A b \omega$
  - We say that  $b \in \text{Follow}(A)$  in this case
  
  - What productions can we use in this case?
    - Any  $A \rightarrow \alpha$  can be used if  $\alpha$  can expand to  $\varepsilon$
    - We say that  $\varepsilon \in \text{First}(A)$  in this case

# Computing First Sets

---

## Definition

$$\text{First}(X) = \{ b \mid X \rightarrow^* b\alpha \} \cup \{ \varepsilon \mid X \rightarrow^* \varepsilon \}$$

## Algorithm sketch

1.  $\text{First}(b) = \{ b \}$
2.  $\varepsilon \in \text{First}(X)$  if  $X \rightarrow \varepsilon$  is a production
3.  $\varepsilon \in \text{First}(X)$  if  $X \rightarrow A_1 \dots A_n$   
and  $\varepsilon \in \text{First}(A_i)$  for  $1 \leq i \leq n$
4.  $\text{First}(\alpha) \subseteq \text{First}(X)$  if  $X \rightarrow A_1 \dots A_n \alpha$   
and  $\varepsilon \in \text{First}(A_i)$  for  $1 \leq i \leq n$

# First Sets: Example

---

- Recall the grammar

$$E \rightarrow TX$$

$$T \rightarrow (E) \mid \text{int } Y$$

$$X \rightarrow + E \mid \varepsilon$$

$$Y \rightarrow * T \mid \varepsilon$$

- First sets

$$\text{First}( ( ) ) = \{ ( \}$$

$$\text{First}( ) ) = \{ ) \}$$

$$\text{First}( \text{int} ) = \{ \text{int} \}$$

$$\text{First}( + ) = \{ + \}$$

$$\text{First}( * ) = \{ * \}$$

$$\text{First}( T ) = \{ \text{int}, ( \}$$

$$\text{First}( E ) = \{ \text{int}, ( \}$$

$$\text{First}( X ) = \{ +, \varepsilon \}$$

$$\text{First}( Y ) = \{ *, \varepsilon \}$$

# Computing Follow Sets

---

- Definition

$$\text{Follow}(X) = \{ b \mid S \rightarrow^* \beta X b \delta \}$$

- Intuition

- If  $X \rightarrow A B$  then  $\text{First}(B) \subseteq \text{Follow}(A)$   
and  $\text{Follow}(X) \subseteq \text{Follow}(B)$
- Also if  $B \rightarrow^* \varepsilon$  then  $\text{Follow}(X) \subseteq \text{Follow}(A)$
- If  $S$  is the start symbol then  $\$ \in \text{Follow}(S)$

# Computing Follow Sets (Cont.)

---

## Algorithm sketch

1.  $\$ \in \text{Follow}(S)$
2.  $\text{First}(\beta) - \{\varepsilon\} \subseteq \text{Follow}(X)$ 
  - For each production  $A \rightarrow \alpha X \beta$
3.  $\text{Follow}(A) \subseteq \text{Follow}(X)$ 
  - For each production  $A \rightarrow \alpha X \beta$  where  $\varepsilon \in \text{First}(\beta)$

# Follow Sets: Example

First( T ) = { int, ( }

First( E ) = { int, ( }

First( X ) = { +, ε }

First( Y ) = { \*, ε }

- Recall the grammar

$$E \rightarrow TX$$

$$T \rightarrow (E) \mid \text{int } Y$$

$$X \rightarrow + E \mid \varepsilon$$

$$Y \rightarrow * T \mid \varepsilon$$

- Follow sets

$$\text{Follow}( + ) = \{ \text{int}, ( \}$$

$$\text{Follow}( ( ) = \{ \text{int}, ( \}$$

$$\text{Follow}( X ) = \{ \$, ) \}$$

$$\text{Follow}( ) ) = \{ +, ), \$ \}$$

$$\text{Follow}( \text{int} ) = \{ *, +, ), \$ \}$$

$$\text{Follow}( * ) = \{ \text{int}, ( \}$$

$$\text{Follow}( E ) = \{ ), \$ \}$$

$$\text{Follow}( T ) = \{ +, ), \$ \}$$

$$\text{Follow}( Y ) = \{ +, ), \$ \}$$

# Constructing LL(1) Parsing Tables

---

- Construct a parsing table  $T$  for CFG  $G$
- For each production  $A \rightarrow \alpha$  in  $G$  do:
  - For each terminal  $b \in \text{First}(\alpha)$  do
$$T[A, b] = \alpha$$
  - If  $\varepsilon \in \text{First}(\alpha)$ , for each  $b \in \text{Follow}(A)$  do
$$T[A, b] = \alpha$$
  - If  $\varepsilon \in \text{First}(\alpha)$  and  $\$ \in \text{Follow}(A)$  do
$$T[A, \$] = \alpha$$

# Constructing LL(1) Tables: Example

---

- Recall the grammar

$$E \rightarrow TX$$

$$T \rightarrow (E) \mid \text{int } Y$$

$$X \rightarrow + E \mid \varepsilon$$

$$Y \rightarrow * T \mid \varepsilon$$

- Where in the line of  $Y$  do we put  $Y \rightarrow * T$ ?
  - In the lines of  $\text{First}(*T) = \{ * \}$
- Where in the line of  $Y$  do we put  $Y \rightarrow \varepsilon$ ?
  - In the lines of  $\text{Follow}(Y) = \{ \$, +, ) \}$



# 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
- For some grammars there is a simple parsing strategy: *Predictive parsing*
- Most programming language grammars are not LL(1)
- Thus, we need more powerful parsing strategies

# Bottom Up Parsing

# Bottom-Up Parsing

---

- Bottom-up parsing is more general than top-down parsing
  - 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:

$$E \rightarrow E + ( E ) \mid \text{int}$$

- Why is this not LL(1)?
- Consider the string:  $\text{int} + ( \text{int} ) + ( \text{int} )$

# The Idea

---

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

str  $w$  input string of terminals

repeat

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

until  $\text{str} = S$  (the start symbol)

OR all possibilities are exhausted

# A Bottom-up Parse in Detail (1)

---

$E \rightarrow E + (E) \mid \text{int}$

$\text{int} + (\text{int}) + (\text{int})$

$\text{int} \quad + \quad ( \text{int} ) \quad + \quad ( \text{int} )$

# A Bottom-up Parse in Detail (2)

---

$E \rightarrow E + (E) \mid \text{int}$

$\text{int} + (\text{int}) + (\text{int})$

$E + (\text{int}) + (\text{int})$

$E$   
|  
 $\text{int} + (\text{int}) + (\text{int})$

# A Bottom-up Parse in Detail (3)

---

$E \rightarrow E + (E) \mid \text{int}$

int + (int) + (int)

E + (int) + (int)

E + (E) + (int)

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



# A Bottom-up Parse in Detail (4)

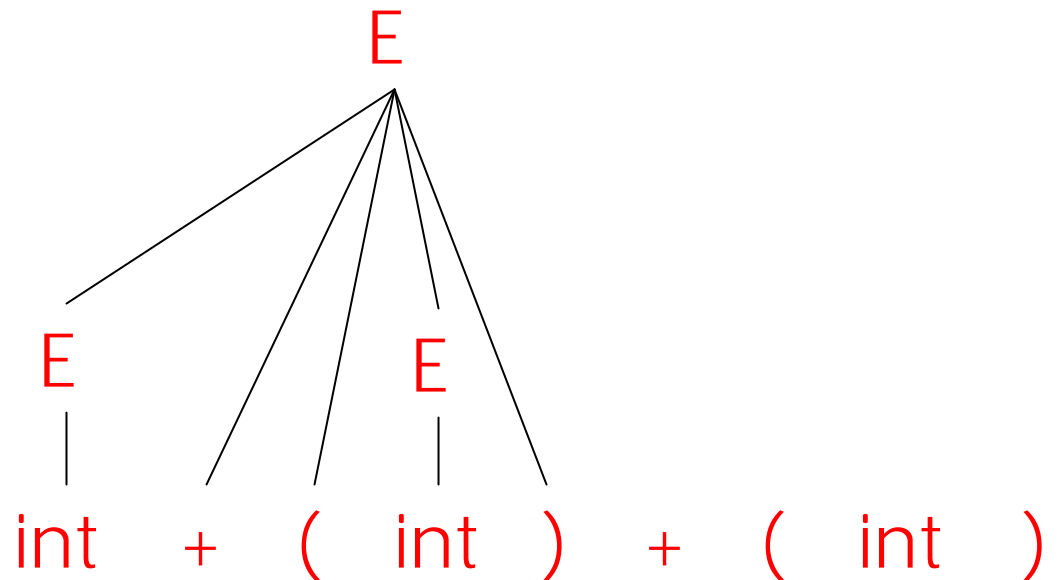
$E \rightarrow E + (E) \mid \text{int}$

int + (int) + (int)

E + (int) + (int)

E + (E) + (int)

E + (int)



# A Bottom-up Parse in Detail (5)

$E \rightarrow E + (E) \mid \text{int}$

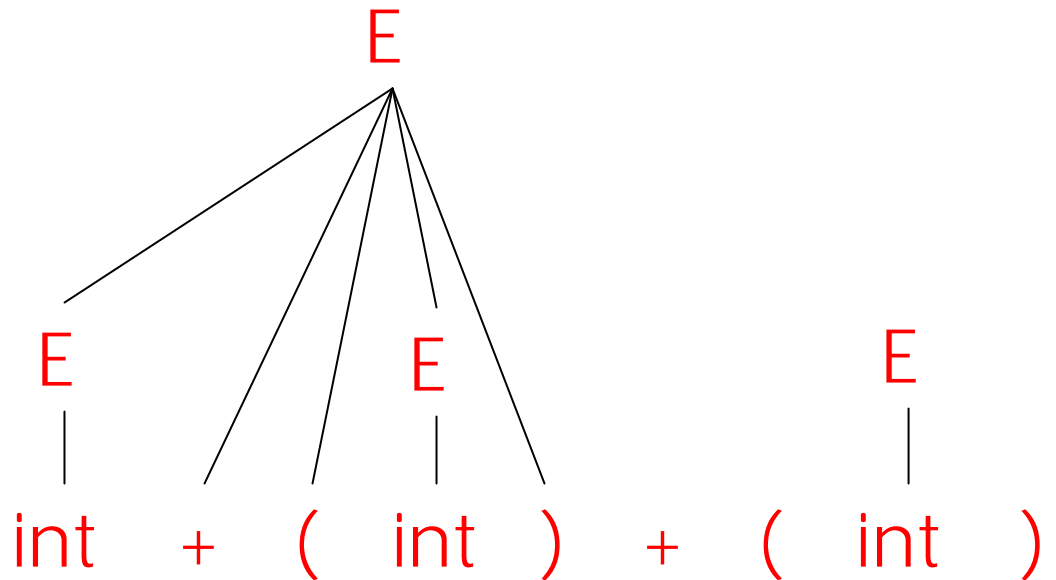
int + (int) + (int)

E + (int) + (int)

E + (E) + (int)

E + (int)

E + (E)

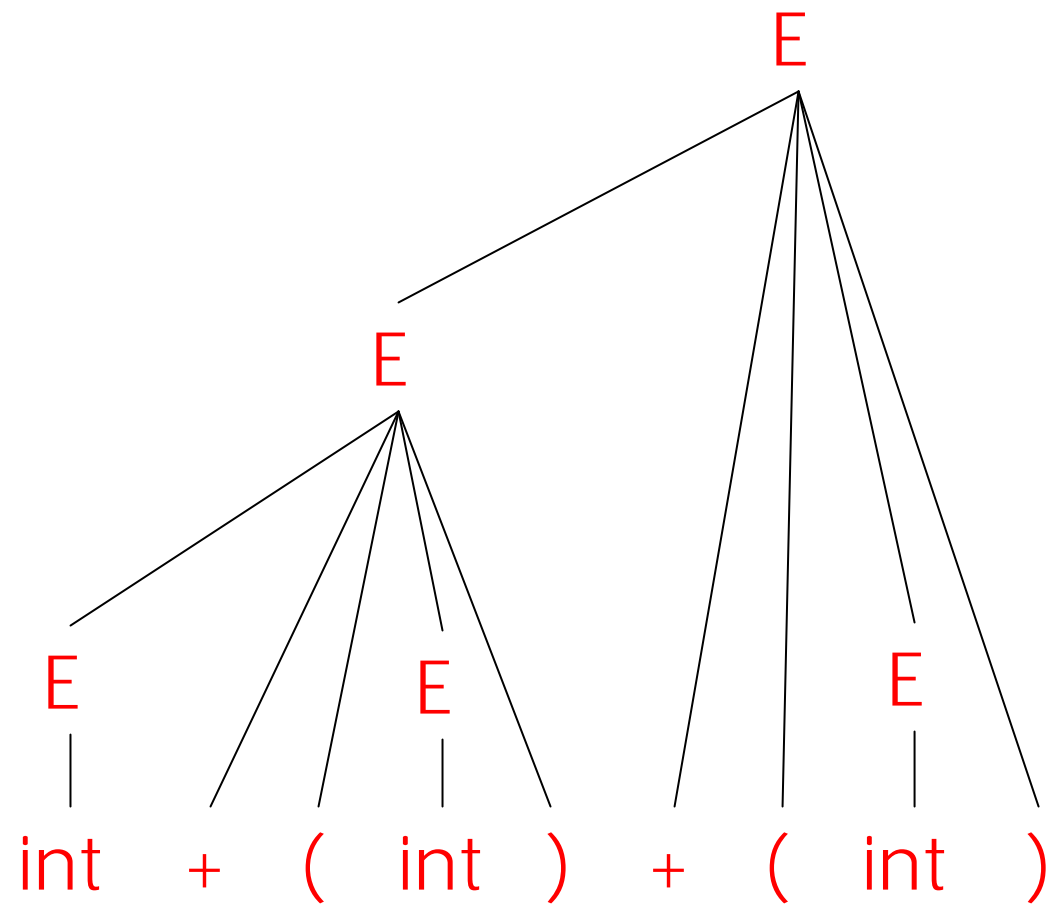


# A Bottom-up Parse in Detail (6)

$E \rightarrow E + (E) \mid \text{int}$

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

A rightmost  
derivation in reverse



# Important Fact #1 about Bottom-up Parsing

---

*An LR parser traces a rightmost derivation in reverse*

# Where Do Reductions Happen

---

Fact #1 has an interesting consequence:

- Let  $\alpha\beta\gamma$  be a step of a bottom-up parse
- Assume the next reduction is by using  $A \rightarrow \beta$
- 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 |
  - The | is not part of the string
- Initially, all input is unexamined: | $x_1x_2 \dots x_n$

# 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

$$E + ( | \text{int} ) \Rightarrow E + ( \text{int} | )$$

In general:

$$ABC | xyz \Rightarrow ABCx | yz$$



# Reduce

---

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

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

$$E + (\underline{E + (E)} | ) \Rightarrow E + (\underline{E} | )$$

In general, given  $A \rightarrow xy$ , then:

$$Cbxy | ijk \Rightarrow CbA | ijk$$

# Shift-Reduce Example

---

$E \rightarrow E + ( E ) \mid \text{int}$

| int + (int) + (int)\$      shift

int + ( int ) + ( int )



# Shift-Reduce Example

---

$E \rightarrow E + ( E ) \mid \text{int}$

| int + (int) + (int)\$

shift

int | + (int) + (int)\$

reduce  $E \rightarrow \text{int}$

int + ( int ) + ( int )



# Shift-Reduce Example

---

$E \rightarrow E + (E) \mid \text{int}$

| int + (int) + (int)\$

shift

int | + (int) + (int)\$

reduce  $E \rightarrow \text{int}$

E | + (int) + (int)\$

shift 3 times

E  
/  
int + ( int ) + ( int )  
↑

# Shift-Reduce Example

---

$E \rightarrow E + ( E ) \mid \text{int}$

int + (int) + (int)\$	shift
int   + (int) + (int)\$	reduce $E \rightarrow \text{int}$
E   + (int) + (int)\$	shift 3 times
E + (int   ) + (int)\$	reduce $E \rightarrow \text{int}$

$E$   
/  
int + ( int ) + ( int )  
↑

# Shift-Reduce Example

$E \rightarrow E + ( E ) \mid \text{int}$

int + (int) + (int)\$	shift
int   + (int) + (int)\$	reduce $E \rightarrow \text{int}$
E   + (int) + (int)\$	shift 3 times
E + (int   ) + (int)\$	reduce $E \rightarrow \text{int}$
E + (E   ) + (int)\$	shift

$$\begin{array}{ccccccc} & & E & & E & & \\ & & / & & | & & \\ \text{int} & + & ( & \text{int} & ) & + & ( & \text{int} & ) \end{array}$$

↑

# Shift-Reduce Example

$E \rightarrow E + (E) \mid \text{int}$

int + (int) + (int)\$	shift
int   + (int) + (int)\$	reduce $E \rightarrow \text{int}$
E   + (int) + (int)\$	shift 3 times
E + (int   ) + (int)\$	reduce $E \rightarrow \text{int}$
E + (E   ) + (int)\$	shift
E + (E)   + (int)\$	reduce $E \rightarrow E + (E)$

$\begin{array}{c} E \\ / \\ \text{int} \end{array} + ( \begin{array}{c} E \\ | \\ \text{int} \end{array} ) + ( \text{int} )$

↑

# Shift-Reduce Example

$E \rightarrow E + (E) \mid \text{int}$

| int + (int) + (int)\$

shift

int | + (int) + (int)\$

reduce  $E \rightarrow \text{int}$

E | + (int) + (int)\$

shift 3 times

E + (int | ) + (int)\$

reduce  $E \rightarrow \text{int}$

E + (E | ) + (int)\$

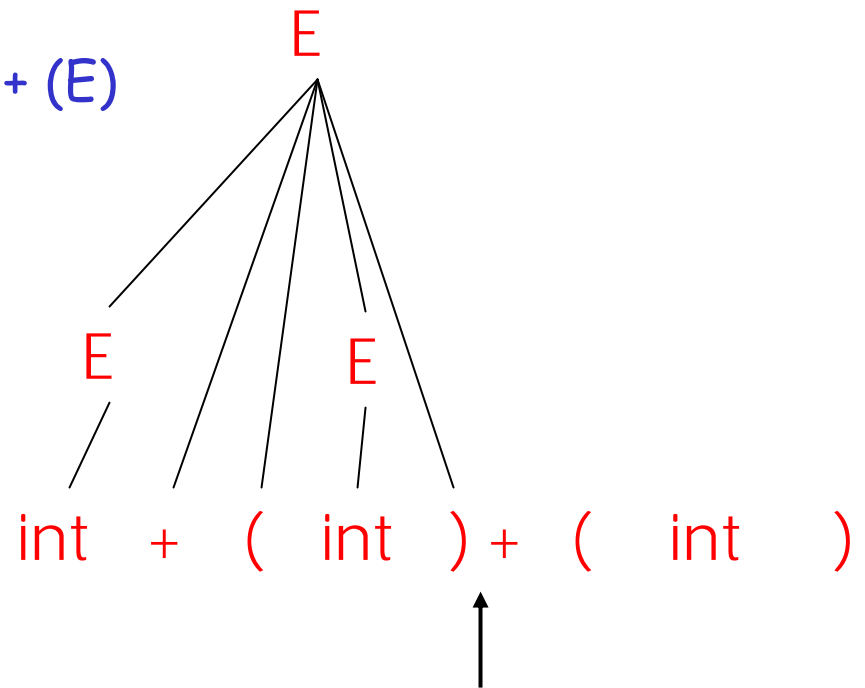
shift

E + (E) | + (int)\$

reduce  $E \rightarrow E + (E)$

E | + (int)\$

shift 3 times





# Shift-Reduce Example

$E \rightarrow E + (E) \mid \text{int}$

| int + (int) + (int)\$

shift

int | + (int) + (int)\$

reduce  $E \rightarrow \text{int}$

E | + (int) + (int)\$

shift 3 times

E + (int | ) + (int)\$

reduce  $E \rightarrow \text{int}$

E + (E | ) + (int)\$

shift

E + (E) | + (int)\$

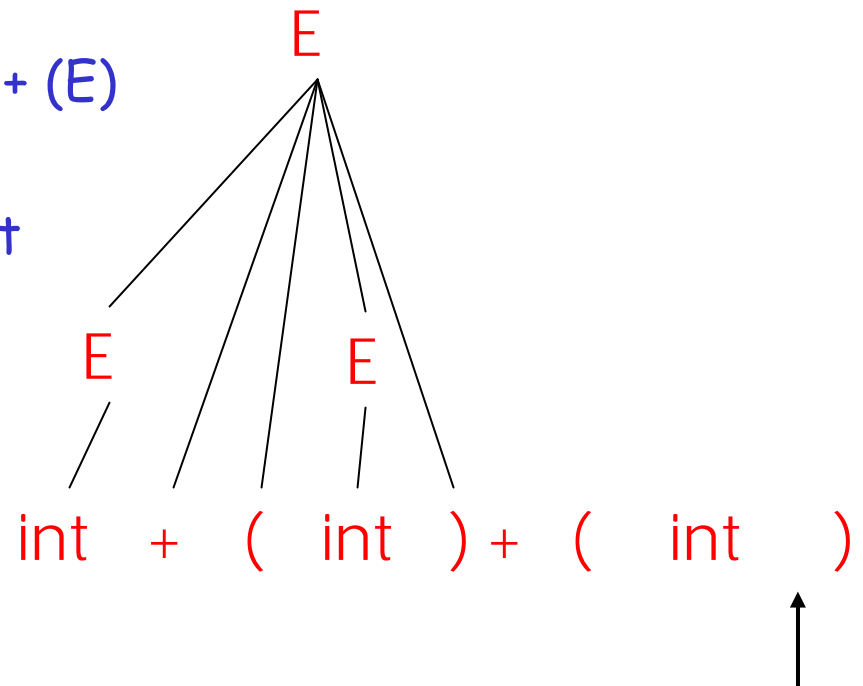
reduce  $E \rightarrow E + (E)$

E | + (int)\$

shift 3 times

E + (int | )\$

reduce  $E \rightarrow \text{int}$



# Shift-Reduce Example

$E \rightarrow E + (E) \mid \text{int}$

| int + (int) + (int)\$

shift

int | + (int) + (int)\$

reduce  $E \rightarrow \text{int}$

E | + (int) + (int)\$

shift 3 times

E + (int | ) + (int)\$

reduce  $E \rightarrow \text{int}$

E + (E | ) + (int)\$

shift

E + (E) | + (int)\$

reduce  $E \rightarrow E + (E)$

E | + (int)\$

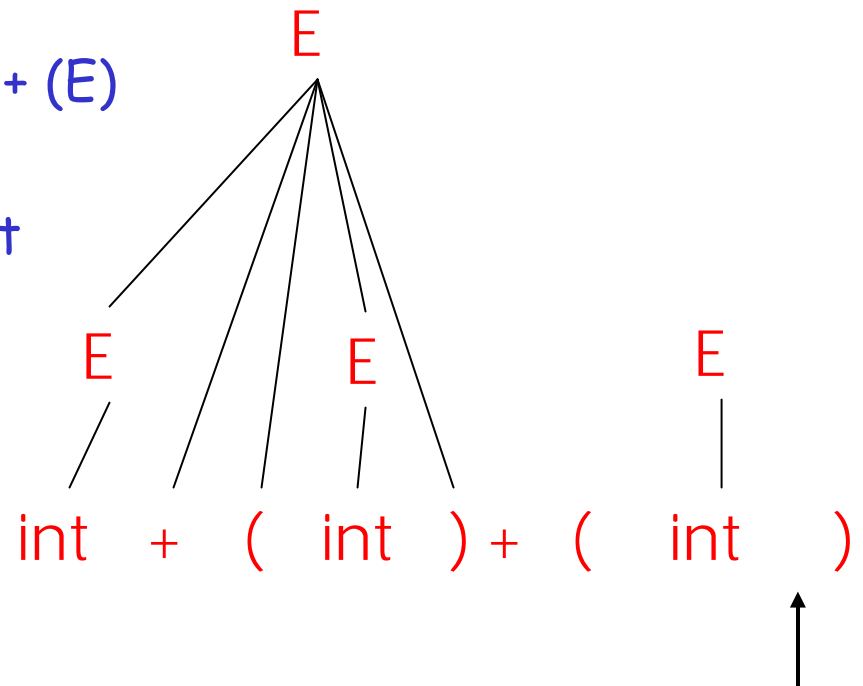
shift 3 times

E + (int | )\$

reduce  $E \rightarrow \text{int}$

E + (E | )\$

shift



# Shift-Reduce Example

$E \rightarrow E + (E) \mid \text{int}$

| int + (int) + (int)\$

shift

int | + (int) + (int)\$

reduce  $E \rightarrow \text{int}$

E | + (int) + (int)\$

shift 3 times

E + (int | ) + (int)\$

reduce  $E \rightarrow \text{int}$

E + (E | ) + (int)\$

shift

E + (E) | + (int)\$

reduce  $E \rightarrow E + (E)$

E | + (int)\$

shift 3 times

E + (int | )\$

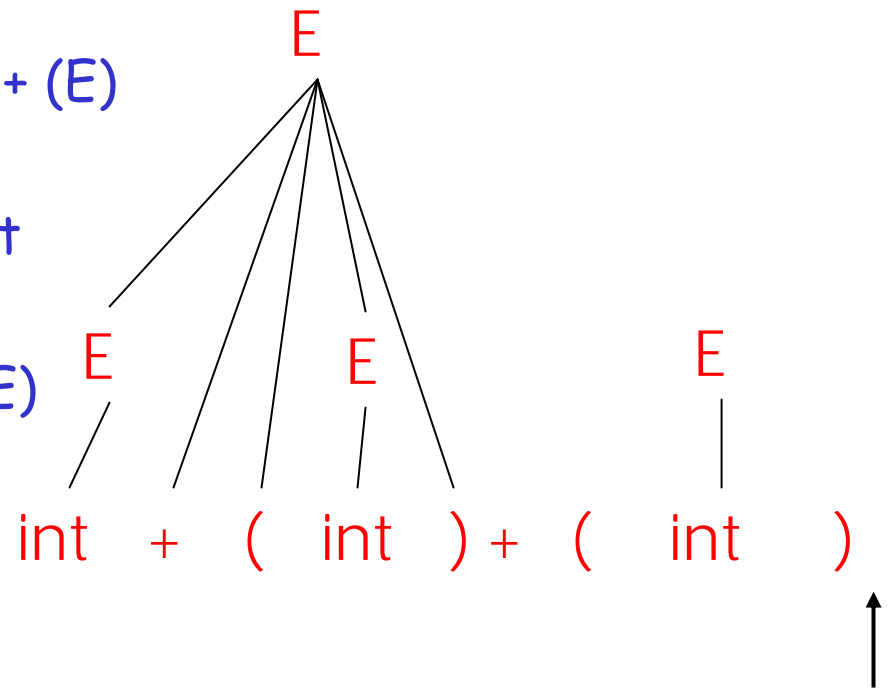
reduce  $E \rightarrow \text{int}$

E + (E | )\$

shift

E + (E) | \$

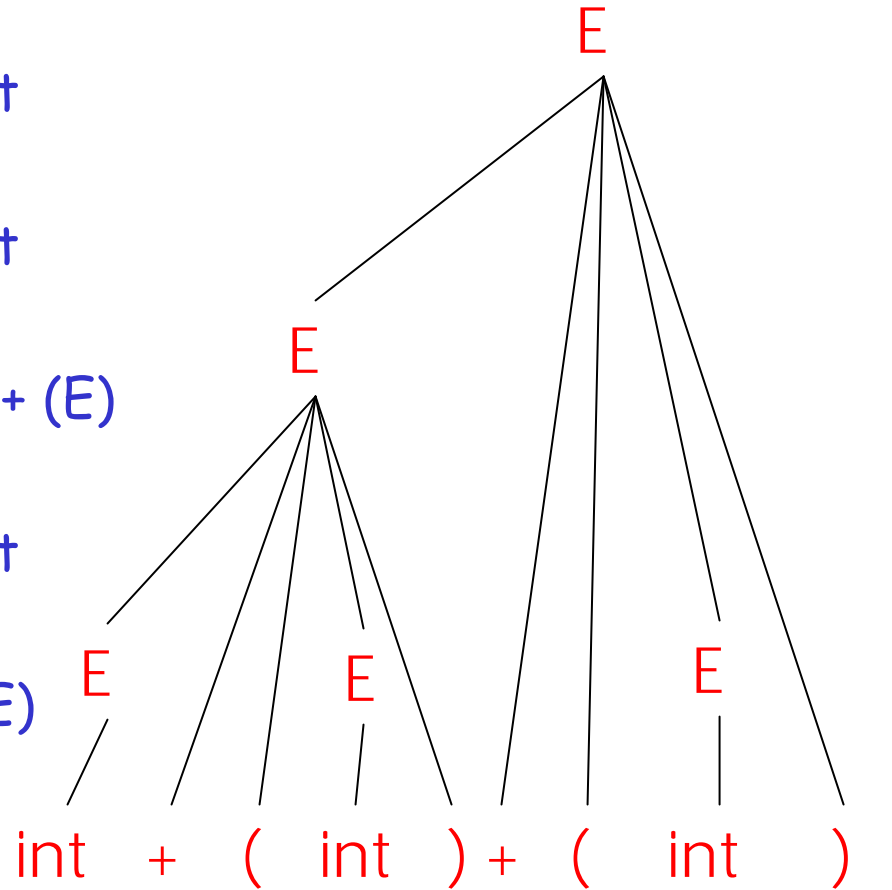
reduce  $E \rightarrow E + (E)$



# Shift-Reduce Example

| int + (int) + (int)\$  
int | + (int) + (int)\$  
E | + (int) + (int)\$  
E + (int | ) + (int)\$  
E + (E | ) + (int)\$  
E + (E) | + (int)\$  
E | + (int)\$  
E + (int | )\$  
E + (E | )\$  
E + (E) | \$  
E | \$

shift  
reduce  $E \rightarrow \text{int}$   
shift 3 times  
reduce  $E \rightarrow \text{int}$   
shift  
reduce  $E \rightarrow E + (E)$   
shift 3 times  
reduce  $E \rightarrow \text{int}$   
shift  
reduce  $E \rightarrow E + (E)$   
**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)

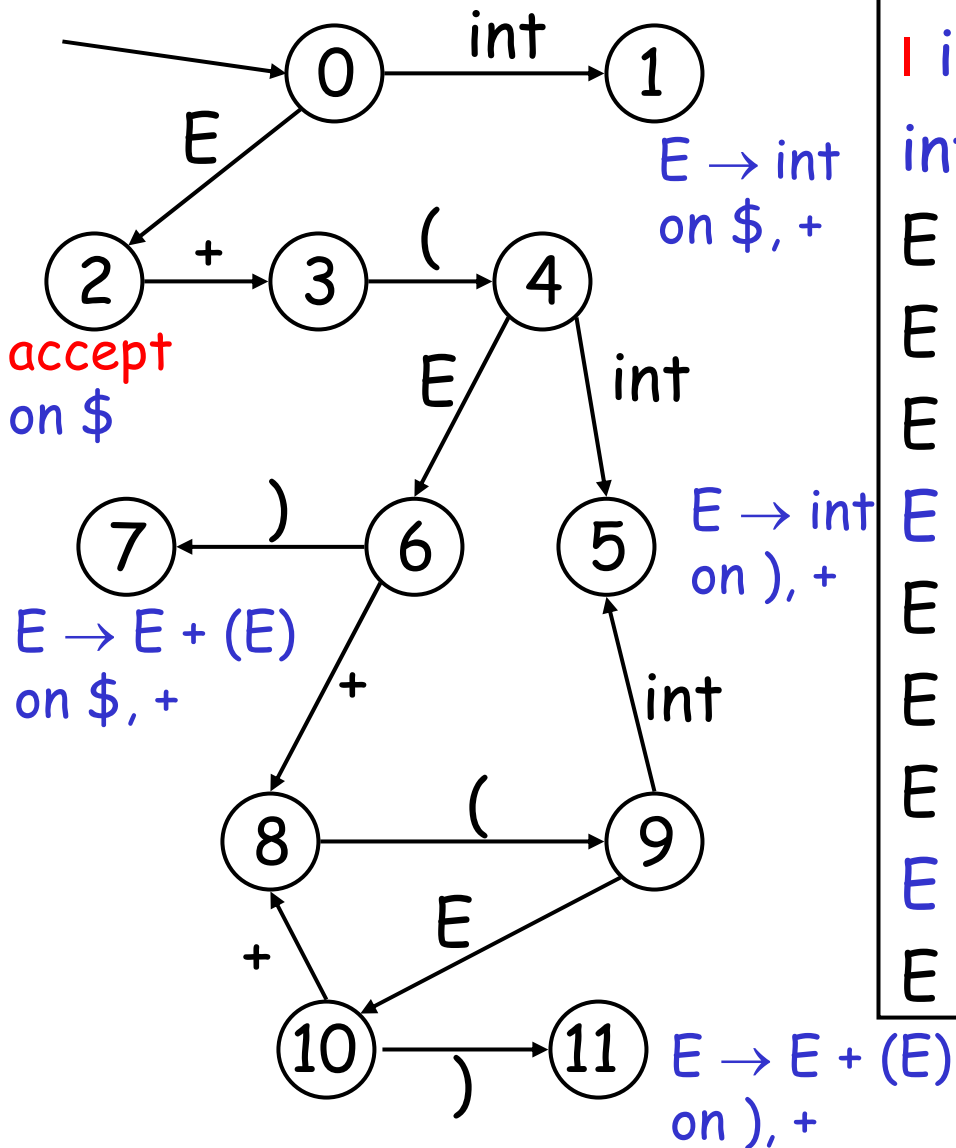
# Key Question: To Shift or to Reduce?

---

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 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)\$	$E \rightarrow \text{int}$
E   + (int) + (int)\$	shift(x3)
E + (int   ) + (int)\$	$E \rightarrow \text{int}$
E + (E   ) + (int)\$	shift
E + (E)   + (int)\$	$E \rightarrow E + (E)$
E   + (int)\$	shift (x3)
E + (int   )\$	$E \rightarrow \text{int}$
E + (E   )\$	shift
E + (E)   \$	$E \rightarrow E + (E)$
E   \$	<b>accept</b>

# Representing the DFA

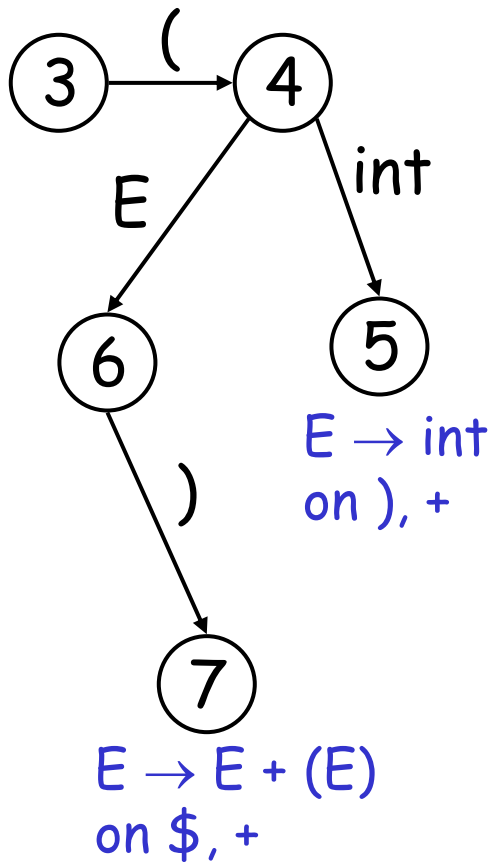
---

- 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
    - action = shift or reduce
  - 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		$r_{E \rightarrow \text{int}}$		$r_{E \rightarrow \text{int}}$		
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
- Remember for each stack element on which state it brings the DFA
- LR parser maintains a stack
$$\langle \text{sym}_1, \text{state}_1 \rangle \dots \langle \text{sym}_n, \text{state}_n \rangle$$
$$\text{state}_k \text{ is the final state of the DFA on } \text{sym}_1 \dots \text{sym}_k$$

# 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 Parsers

---

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