Lecture VI—Part 2: Syntactic Analysis

Bottom-up Parsing: LR 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:

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

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

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)

E
|
int + (int) + (int)

A Bottom-up Parse in Detail (3)

int + (int) + (int)

E + (int) + (int)

E + (E) + (int)

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

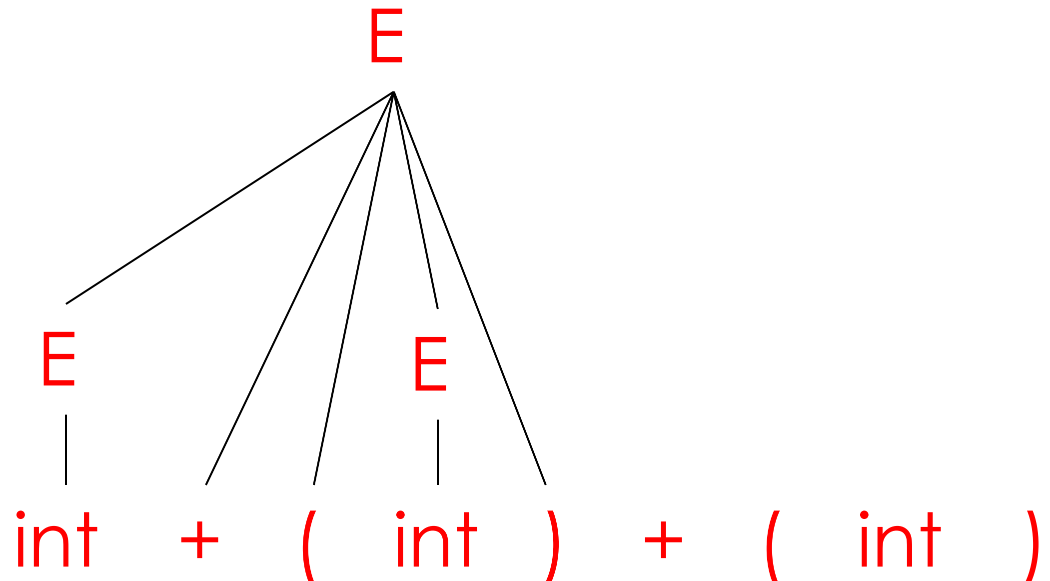
A Bottom-up Parse in Detail (4)

int + (int) + (int)

E + (int) + (int)

E + (E) + (int)

E + (int)



A Bottom-up Parse in Detail (5)

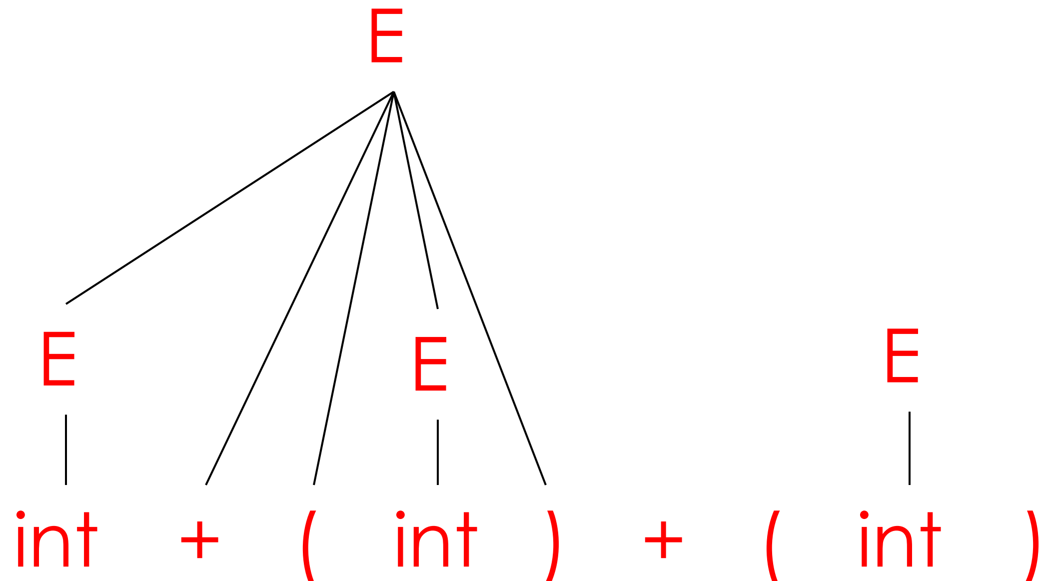
int + (int) + (int)

E + (int) + (int)

E + (E) + (int)

E + (int)

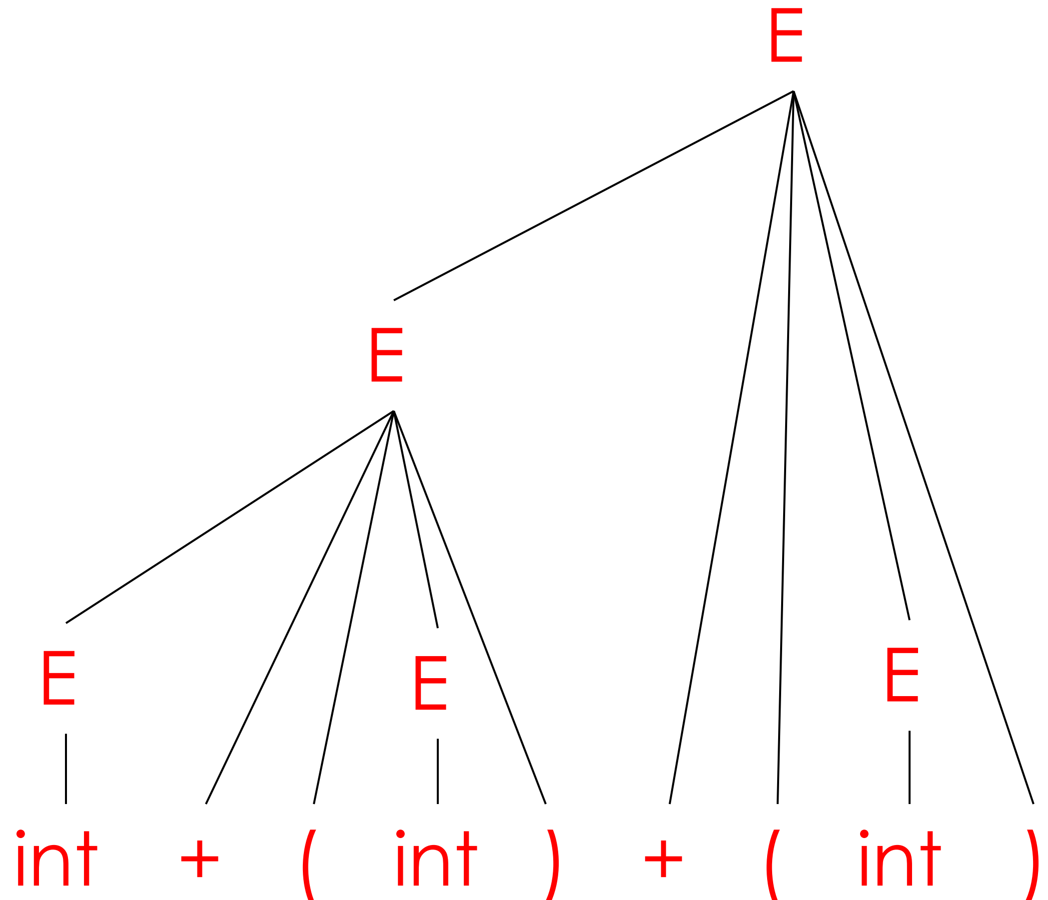
E + (E)



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 γ 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 (a string of terminals) is as yet unexamined by parser
  - Left substring has terminals and non-terminals
- The dividing point is marked by a ı
  - The ı is not part of the string
- Initially, all input is unexamined: $\mid x_1 x_2 \ldots x_n$

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) ⇒ 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

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

Shift-Reduce Example

I int + (int) + (int)$ shift

int + (int) + (int)

Shift-Reduce Example

| int + (int) + (int)$ shift

int | + (int) + (int)$ red. $E \rightarrow int$

int + (int) + (int)

Shift-Reduce Example

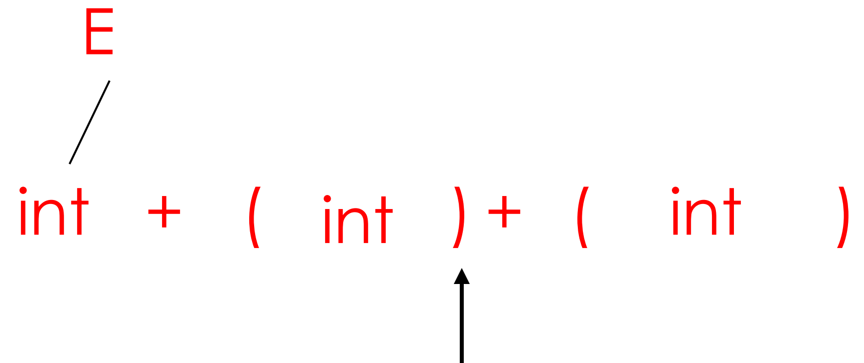
| int + (int) + (int)$ shift

int | + (int) + (int)$ red. $E \rightarrow int$

E | + (int) + (int)$ shift 3 times

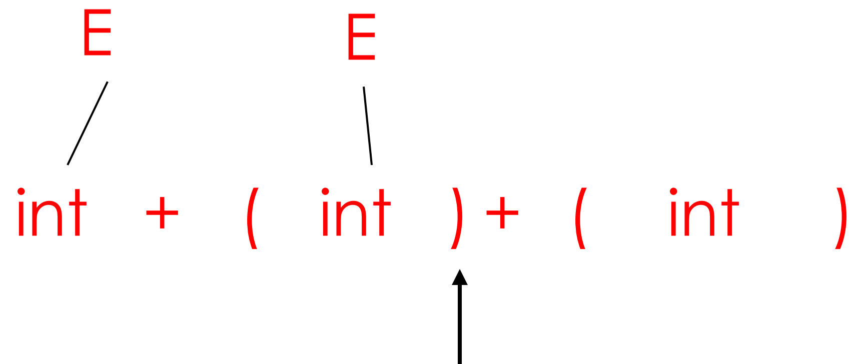
Shift-Reduce Example

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



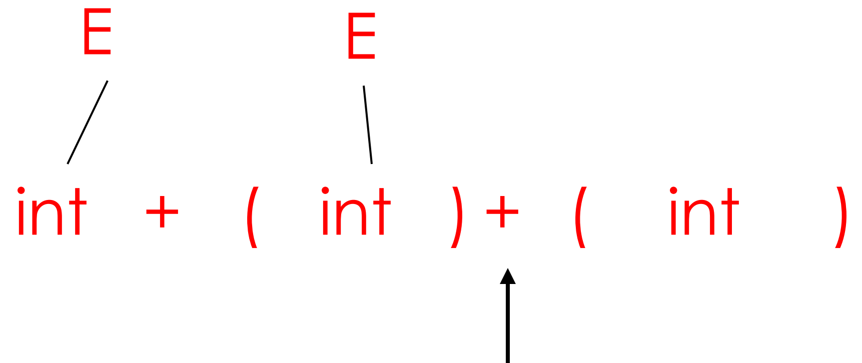
Shift-Reduce Example

I int + (int) + (int)$	shift
int I + (int) + (int)$	red. $E \rightarrow$ int
E I + (int) + (int)$	shift 3 times
E + (int I) + (int)$	red. $E \rightarrow$ int
E + (E I) + (int)$	shift



Shift-Reduce Example

∣ int + (int) + (int)$	shift
int ∣ + (int) + (int)$	red. E $\rightarrow$ int
E ∣ + (int) + (int)$	shift 3 times
E + (int ∣) + (int)$	red. E $\rightarrow$ int
E + (E ∣) + (int)$	shift
E + (E) ∣ + (int)$	red. E $\rightarrow$ E + (E)



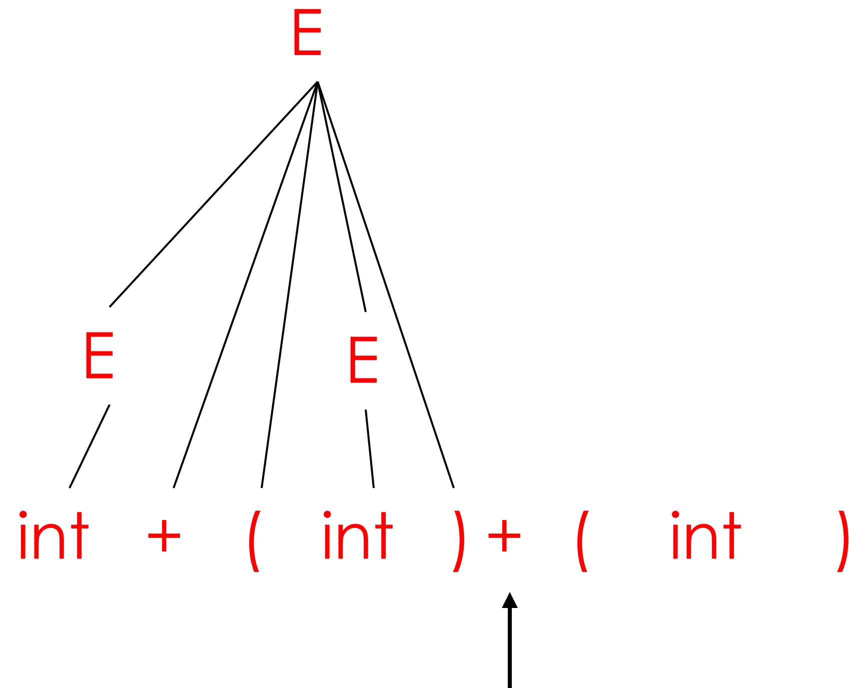
Shift-Reduce Example

| 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

ı int + (int) + (int)$ shift

int ı + (int) + (int)$ red. E → int

E ı + (int) + (int)$ shift 3 times

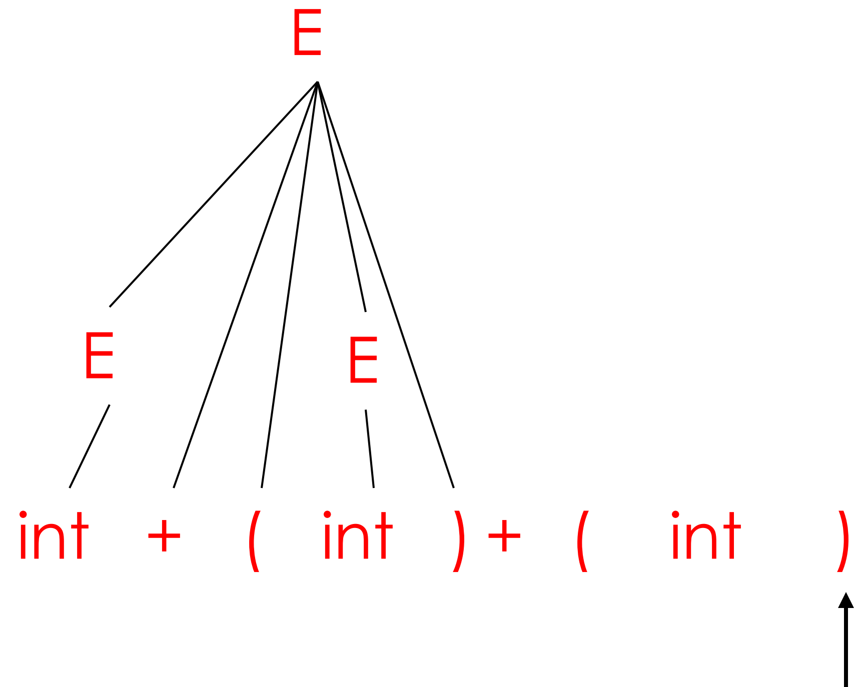
E + (int ı) + (int)$ red. E → int

E + (E ı) + (int)$ shift

E + (E) ı + (int)$ red. E → E + (E)

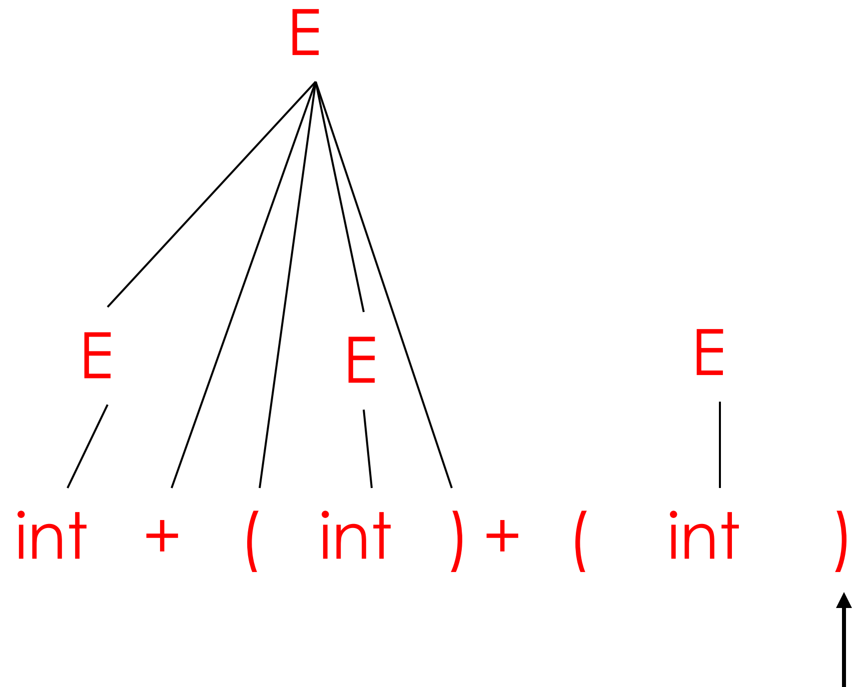
E ı + (int)$ shift 3 times

E + (int ı)$ red. E → int



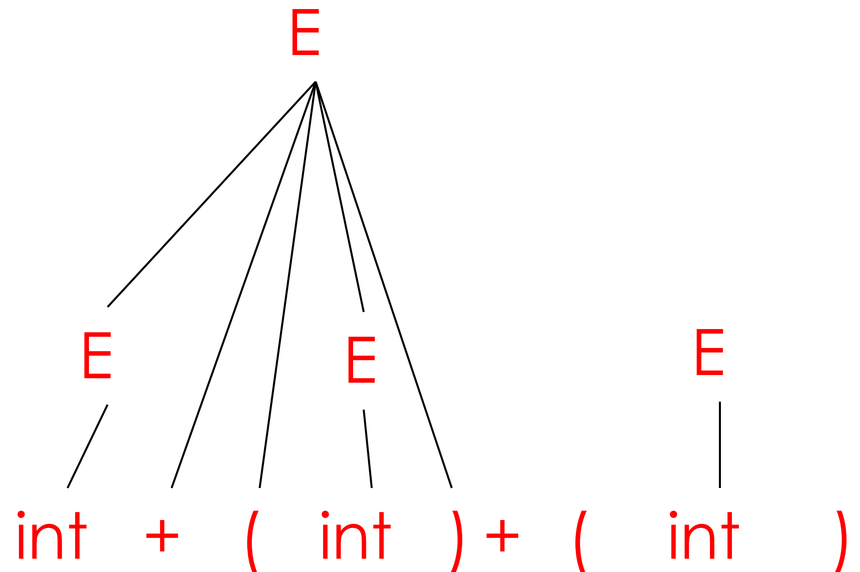
Shift-Reduce Example

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



Shift-Reduce Example

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

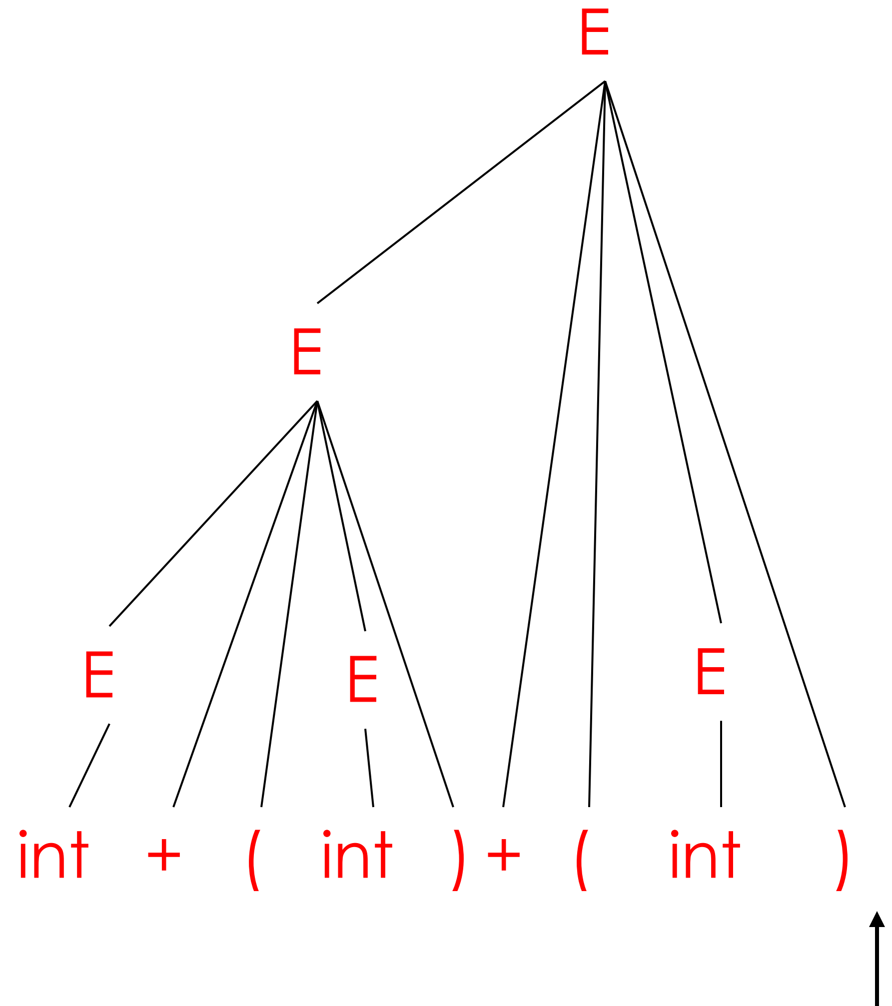
E | + (int)$ — shift 3 times

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

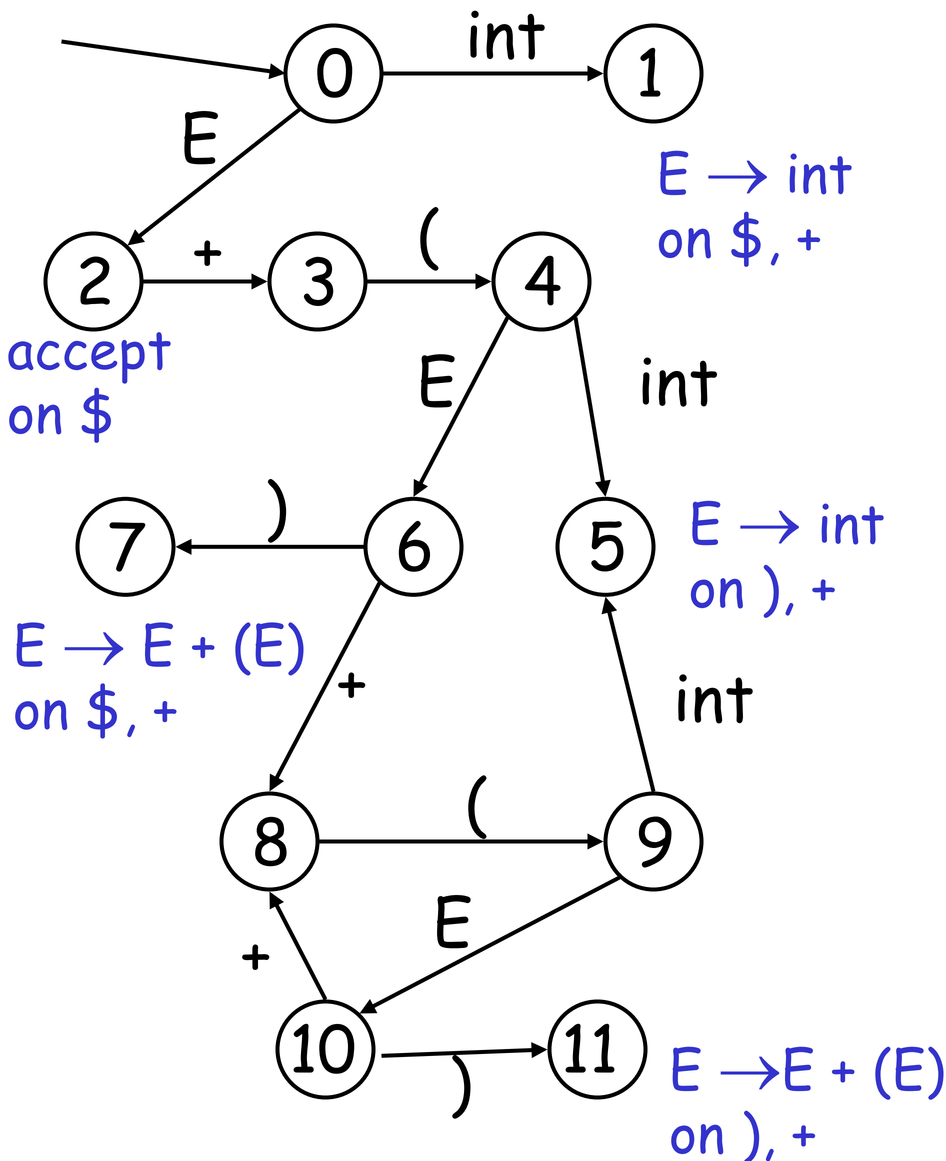
- Reduce pops 0 or more symbols off of the stack (production rhs) and pushes a non-terminal on the stack (production lhs)

Key Issue: When to Shift or Reduce?

- Decide based on the stack's content and the lookahead
- Idea: use a finite automaton (DFA) to decide when to shift or reduce
  - The DFA input is the stack's content
  - The alphabet 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 <u>shift</u>
  - If X is labeled with "$A \rightarrow \beta$ on tok" then <u>reduce</u>

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)

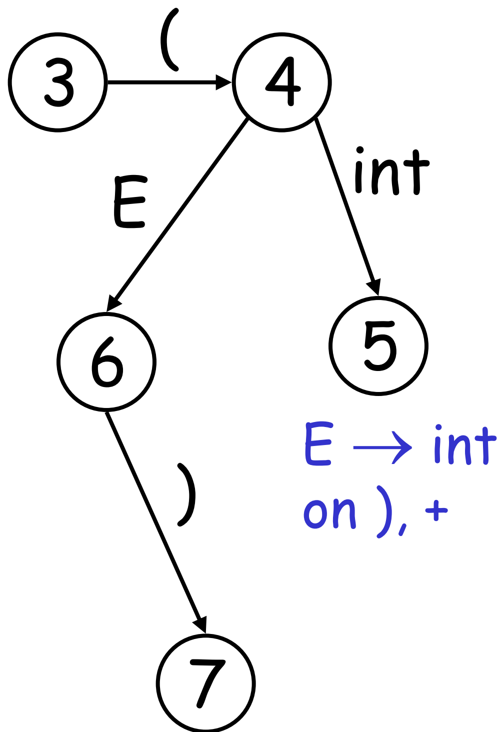
E | $ accept

Representing the DFA

- Parsers represent the DFA as a 2D table
- 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

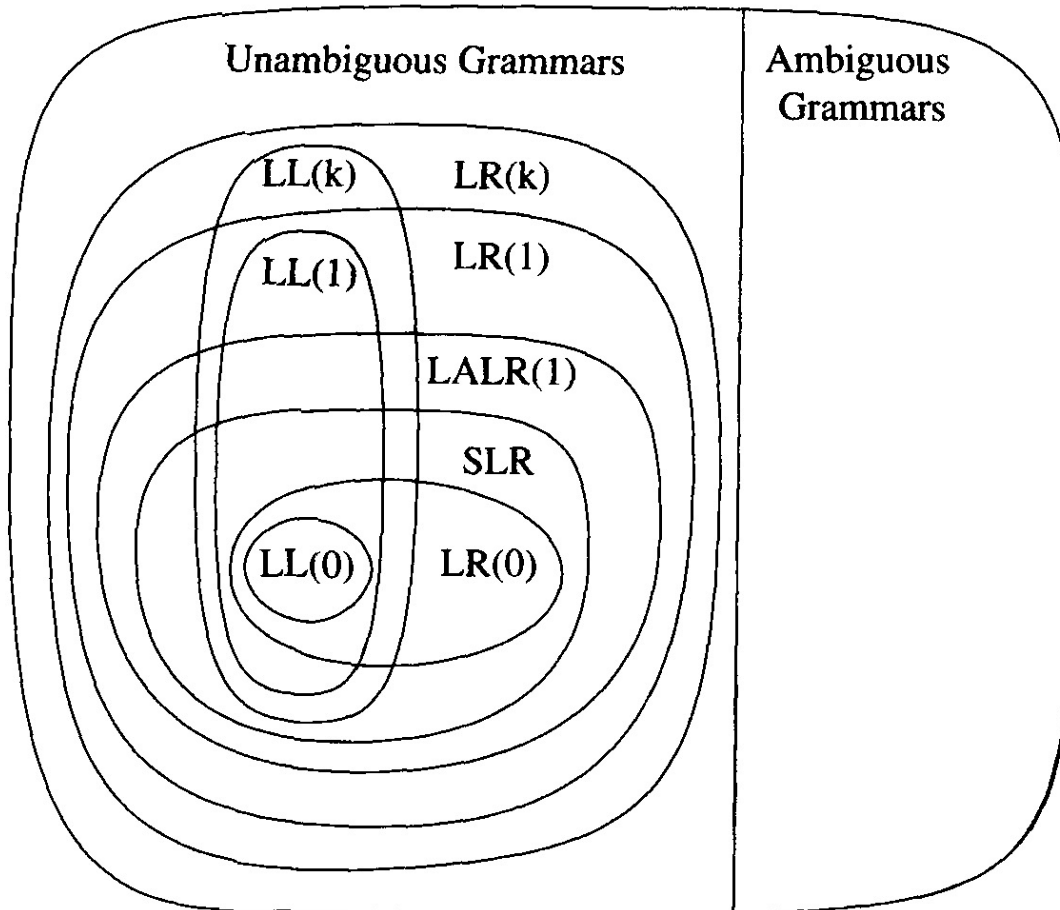
Representing the DFA. Example

- The table for a fragment of our DFA:



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

A Hierarchy of Grammar Classes



From Andrew Appel, "Modern Compiler Implementation in Java"