The more things change...

LR Parsing

Bottom-Up Parsing

Outline

- No Stopping The Parsing!
  - LL(1) Construction
- Bottom-Up Parsing

- LR Parsing
  - Shift and Reduce
  - LR(1) Parsing Algorithm

- LR(1) Parsing Tables

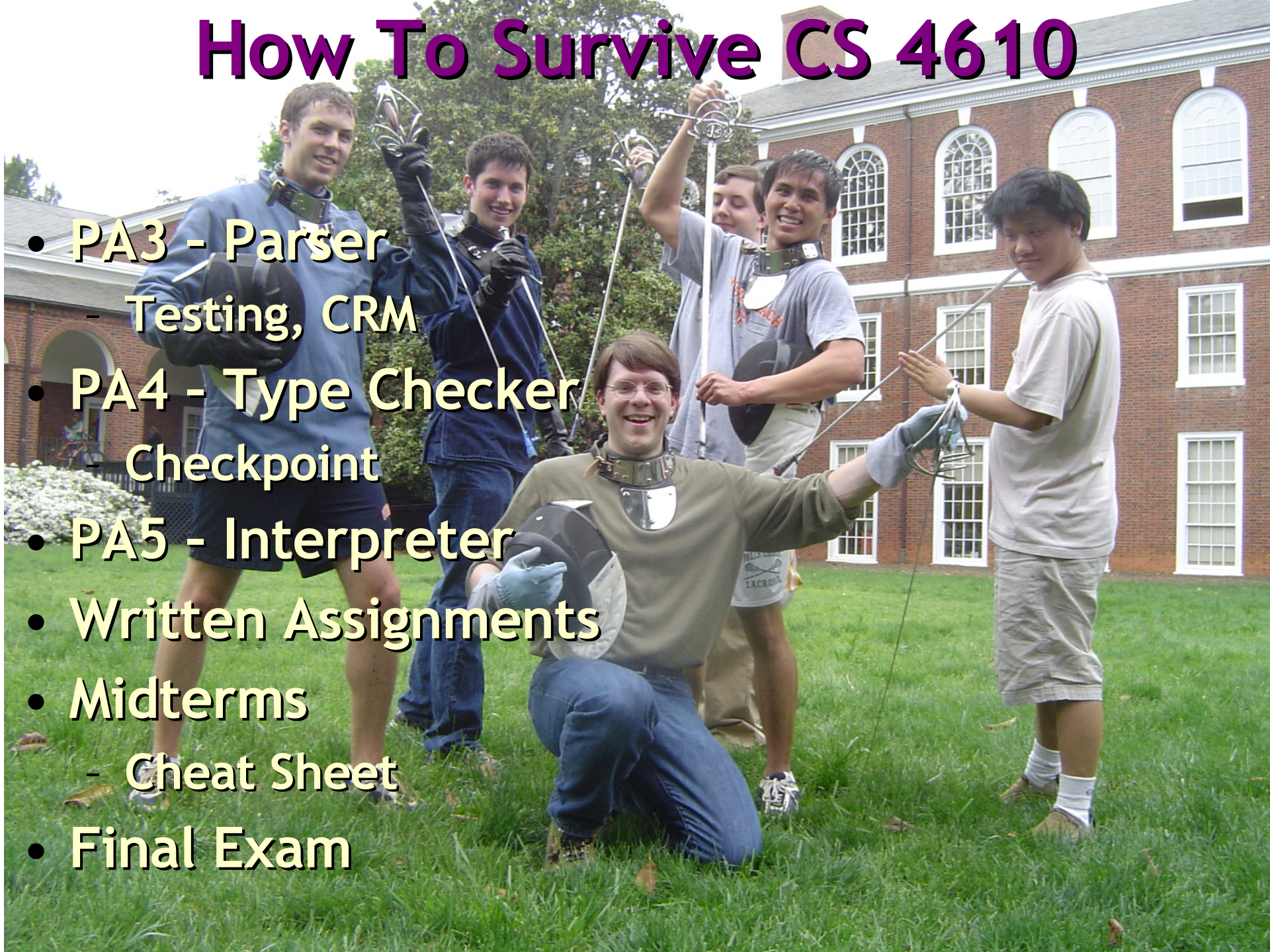
Software Engineering

- The most important consideration is that the software do what it is supposed to.
  - and be delivered on time and within budget ...
- You gather **requirements**
- And refine them into a **specification**
- And then write **code** to meet that specification
  - **Correctly** meeting the spec is paramount!

Lexer Commentary

- Compilation
  - naming of files, syntax errors, include .mll files
    - non-standard compile: -5; no compile: -all, README not plain text: -2
- Output
  - should be: if error, report to stdout, else write tokens to .cl-lex file
  - reality: write to stderr, write error to file, write tokens to stdout, ...
- Corner Cases and Correctness
  - long strings, junk in strings, EOF in comment, ...
  - Average: 34.6 / 50
    - Make an appointment with me via email to demonstrate that yours compiles and regain (some) points

How To Survive CS 4610



- PA3 – Parser
  - Testing, CRM
- PA4 – Type Checker
  - Checkpoint
- PA5 – Interpreter
- Written Assignments
- Midterms
  - Cheat Sheet
- Final Exam

In One Slide

- If the LL(1) table for a grammar G has multiple entries, then G is **not LL(1)**!
- An **LR(1) parser** reads tokens from left to right and constructs a bottom-up rightmost derivation. LR(1) parsers shift terminals and reduce the input by application productions in reverse. LR(1) parsing is fast and easy, and uses a finite automaton with a stack. LR(1) works fine if the grammar is left-recursive, or not left-factored.

Constructing LL(1) Parsing Tables

- Here is how to construct a parsing table T for context-free grammar G

- For each production $A \rightarrow \alpha$ in G do:
  - For each terminal $b \in First(\alpha)$ do
    - $T[A, b] = \alpha$
  - If $\alpha \rightarrow^* \varepsilon$, for each $b \in Follow(A)$ do
    - $T[A, b] = \alpha$

LL(1) Table Construction Example

- Recall the grammar

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

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

- Where in the row of Y do we put $Y \rightarrow *\ T$?
  - In the columns of First(*T) = { * }

	int	*	+	(	)	$
T	int Y			(E)		
E	T X			T X		
X			+ E		ε	ε
Y		* T	ε		ε	ε

LL(1) Table Construction Example

- Recall the grammar

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

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

- Where in the row of Y we put $Y \rightarrow \varepsilon$?
  - In the columns of Follow(Y) = { $, +,) }

	int	*	+	(	)	$
T	int Y			(E)		
E	T X			T X		
X			+ E		ε	ε
Y		* T	ε		ε	ε

Avoid Multiple Definitions!

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*
- Most programming language grammars are not LL(1) (e.g., Java, Ruby, C++, OCaml, Cool, Perl, ...)
- There are tools that build LL(1) tables

Top-Down Parsing Strategies

- Recursive Descent Parsing
  - But backtracking is too annoying, etc.
- Predictive Parsing, aka. **LL(k)**
  - Predict production from k tokens of lookahead
  - Build LL(1) table
  - Parsing using the table is fast and easy
  - But many grammars are not LL(1) (or even LL(k))
- Next: a more powerful parsing strategy for grammars that are not LL(1)

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)? (Guess before I show you!)

- 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** → **β** is a production (i.e., **str** = **α β γ**)
- Replace **β** by **A** in **str** (i.e., **str** becomes **α A γ**)

until **str** = **S**

A Bottom-up Parse in Detail (1)

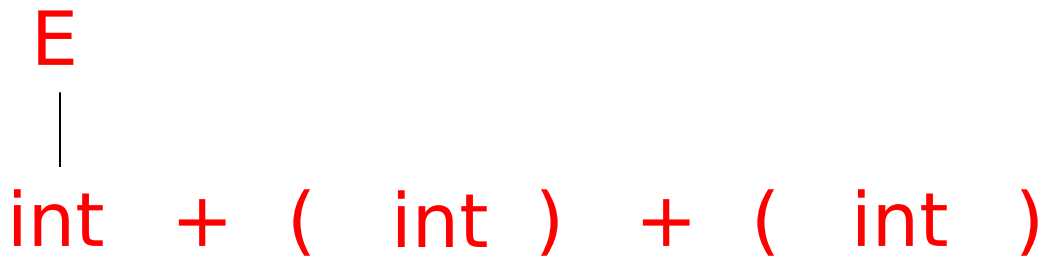
int + (int) + (int)

int + (int) + (int)

A Bottom-up Parse in Detail (2)

int + (int) + (int)

E + (int) + (int)

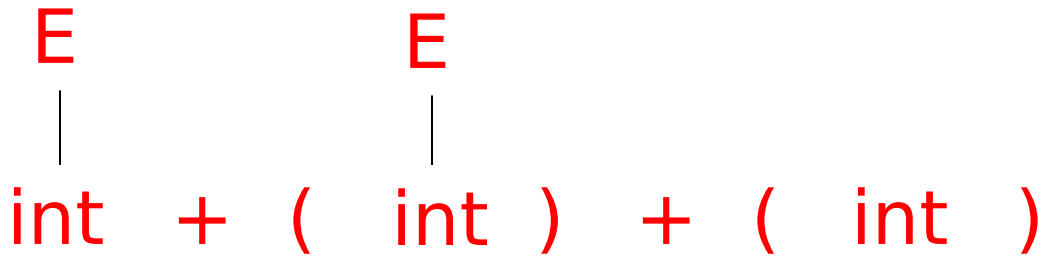


A Bottom-up Parse in Detail (3)

int + (int) + (int)

E + (int) + (int)

E + (E) + (int)



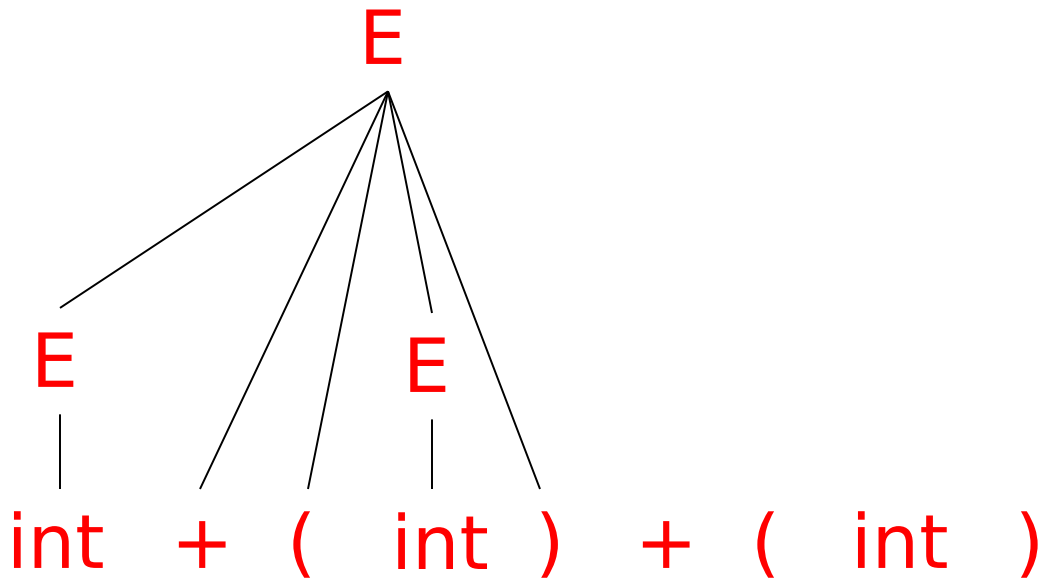
A Bottom-up Parse in Detail (4)

int + (int) + (int)

E + (int) + (int)

E + (E) + (int)

E + (int)



A Bottom-up Parse in Detail (5)

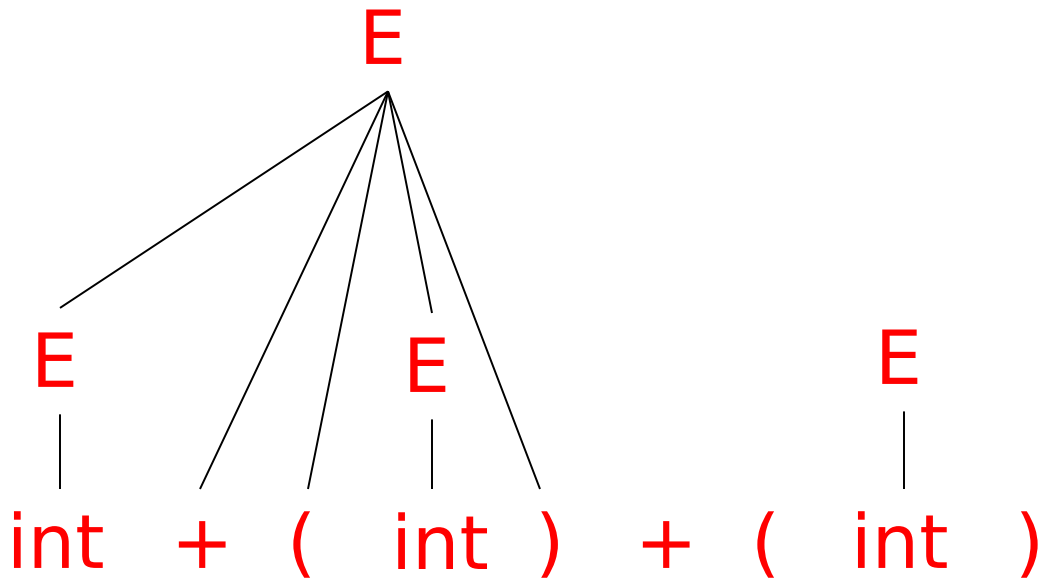
int + (int) + (int)

E + (int) + (int)

E + (E) + (int)

E + (int)

E + (E)



A Bottom-up Parse in Detail (6)

int + (int) + (int)

E + (int) + (int)

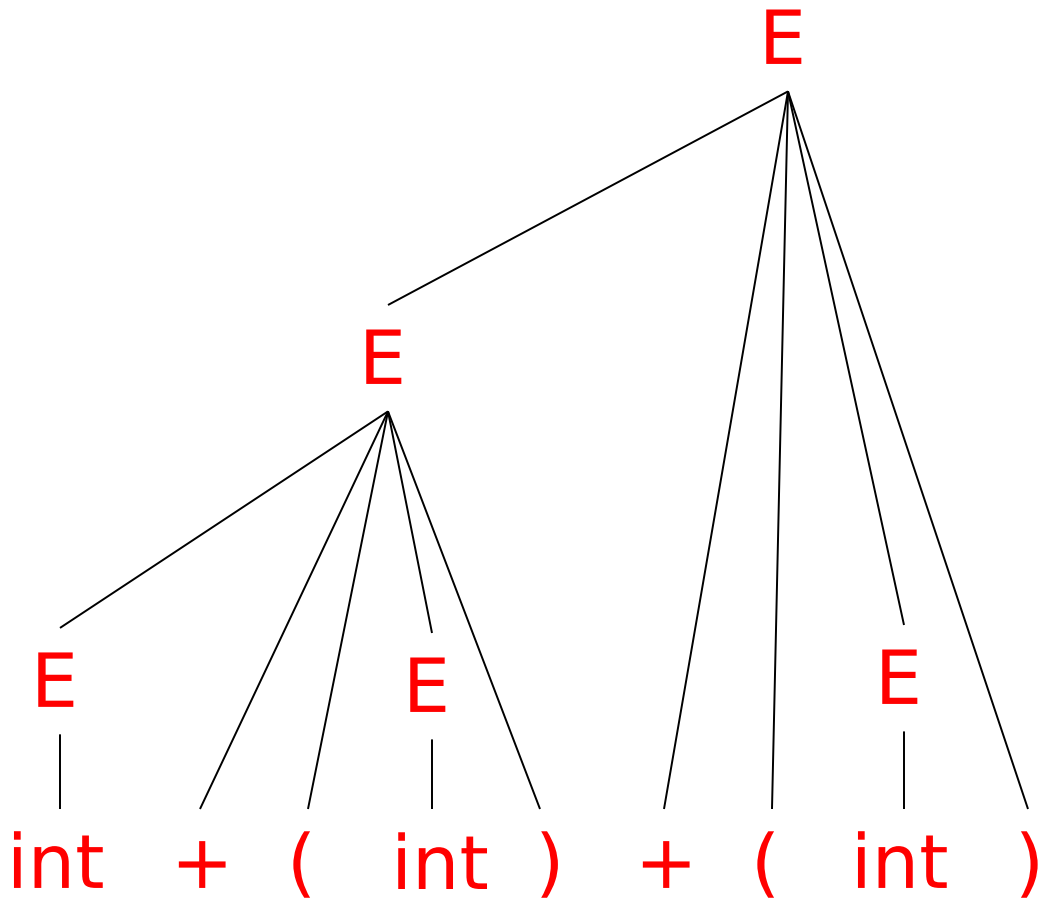
E + (E) + (int)

E + (int)

E + (E)

E

A rightmost derivation
in reverse



Important Fact

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 the 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 ►
  - The ► is not part of the string

- Initially, all input is new: ►$x_1 x_2 \ldots x_n$

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)

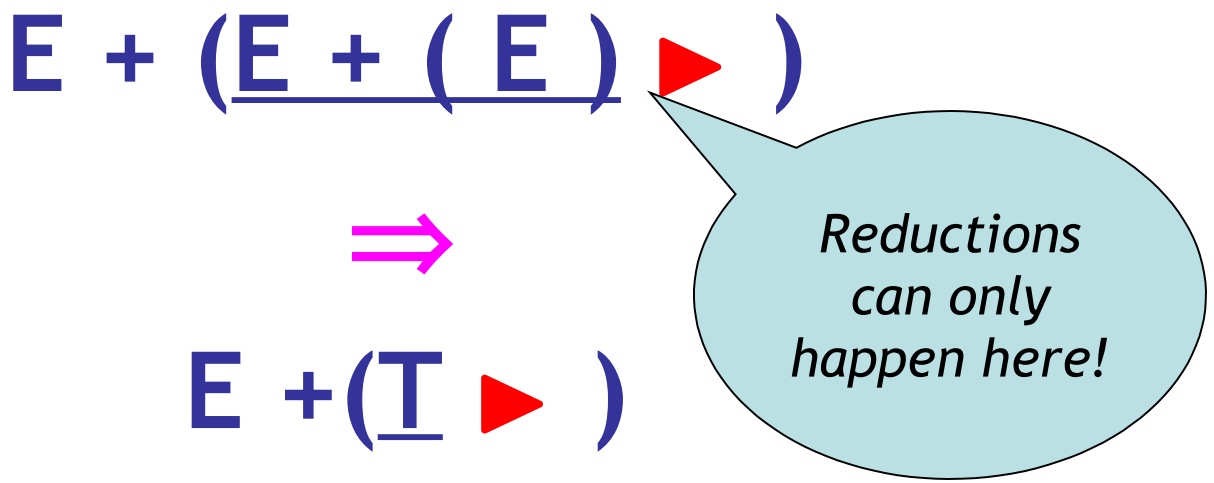
⇒

E + (int ▶)

Reduce

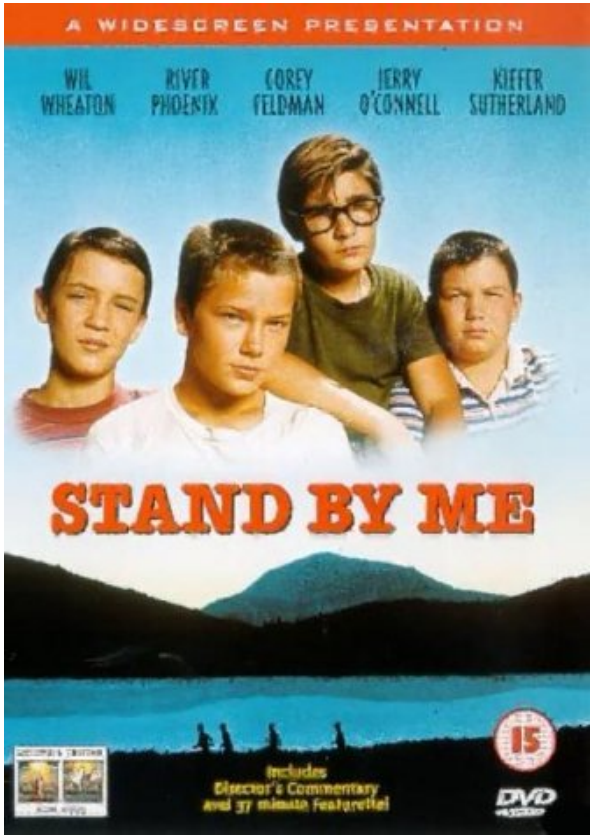
Reduce: Apply an inverse production at the right end of the left string

- If **T → E + (E)** is a production, then



Q: Movies (268 / 842)

- In the 1986 movie **Stand By Me**, what do the kids journey to see?



Q: Books (769 / 842)

- Who wrote the 1978 book entitled **What is the Name of this Book?**, obliquely referenced here by critic Andrew Maizels:
  - In fact, between SG&E and the Illuminatus! trilogy, I was sufficiently intrigued that I went out and bought a copy of Ayn Rand's **What is the Name of this Book?**. No, sorry, **Atlas Shrugged**, that's the one. Of course, I haven't actually read it; every so often I pick it up, and say to myself: "it's a lovely day today, I think I shall go for a walk". This book has improved my life immensely.

Current Events

- Name the scientist to whom this September 11, 2009 formal apology was directed by British Prime Minister Gordon Brown:
- "[name] was a quite brilliant mathematician, most famous for his work on breaking the German Enigma codes. It is no exaggeration to say that, without his outstanding contribution, the history of the Second World War could well have been very different. In 1952, he was convicted of gross indecency - in effect, tried for being gay. His sentence [...] was chemical castration by a series of injections of female hormones. He took his own life just two years later. I am pleased to have the chance to say how deeply sorry I and we all are for what happened to him. [He] and the many thousands of other gay men who were convicted as he was convicted, under homophobic laws, were treated terribly."

Q: Movies (287 / 842)

- This 1995 Pixar animated film featured "Forrest Gump" and "Tim The Tool-Man Taylor". Initially jealous and resentful, the two eventually work together.

Shift-Reduce Example

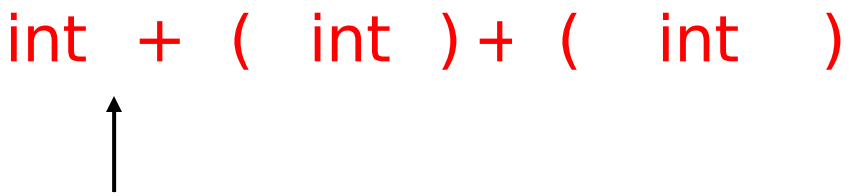
- int + (int) + (int)$ shift

int + (int) + (int)

Shift-Reduce Example

- ▶ int + (int) + (int)$ shift

int ▶ + (int) + (int)$ red. E → int

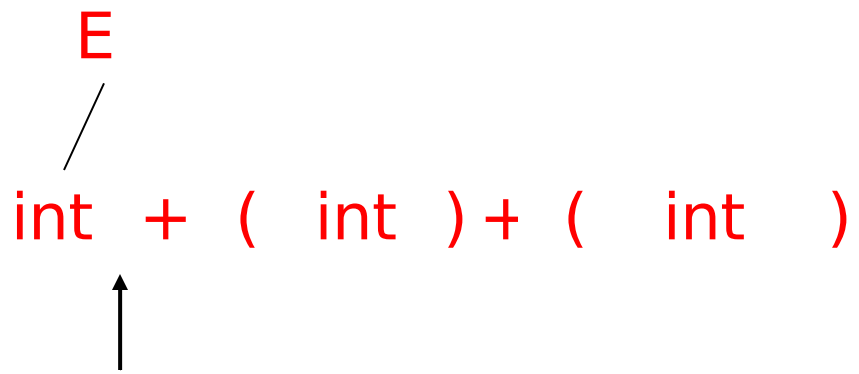


Shift-Reduce Example

▶ int + (int) + (int)$ shift

int ▶ + (int) + (int)$ red. E → int

E ▶ + (int) + (int)$ shift 3 times



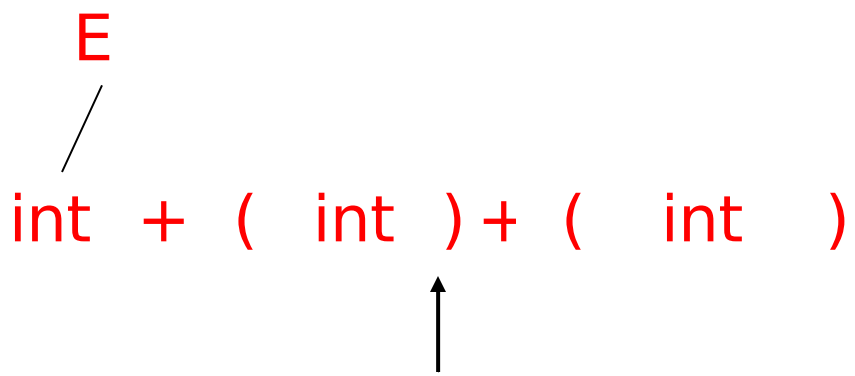
Shift-Reduce Example

▶ 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

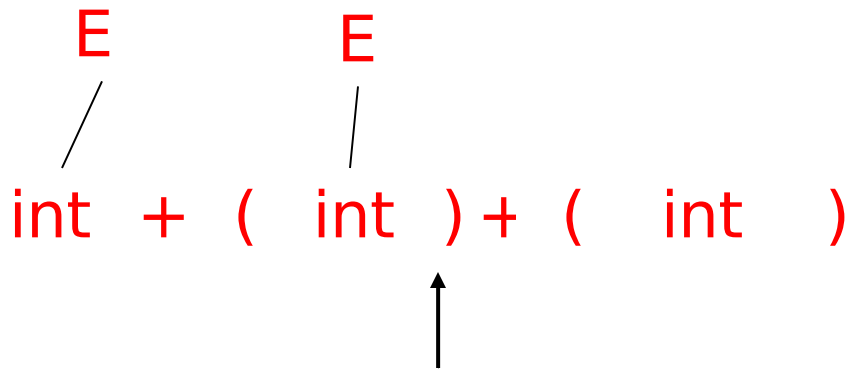
▶ int + (int) + (int)$ shift

int ▶ + (int) + (int)$ red. E → int

E ▶ + (int) + (int)$ shift 3 times

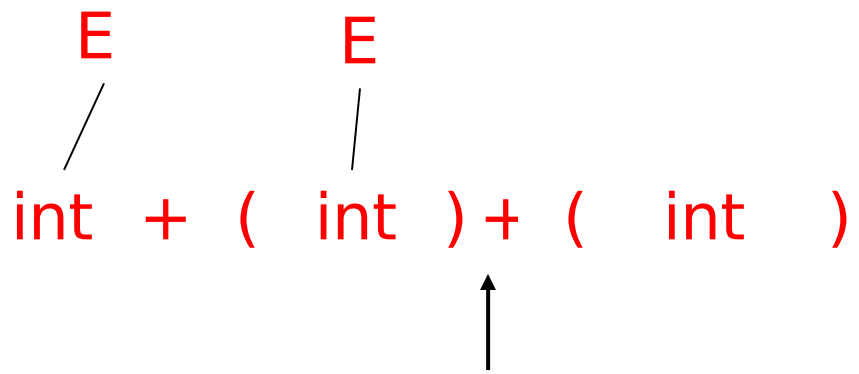
E + (int ▶) + (int)$ red. E → int

E + (E ▶) + (int)$ shift



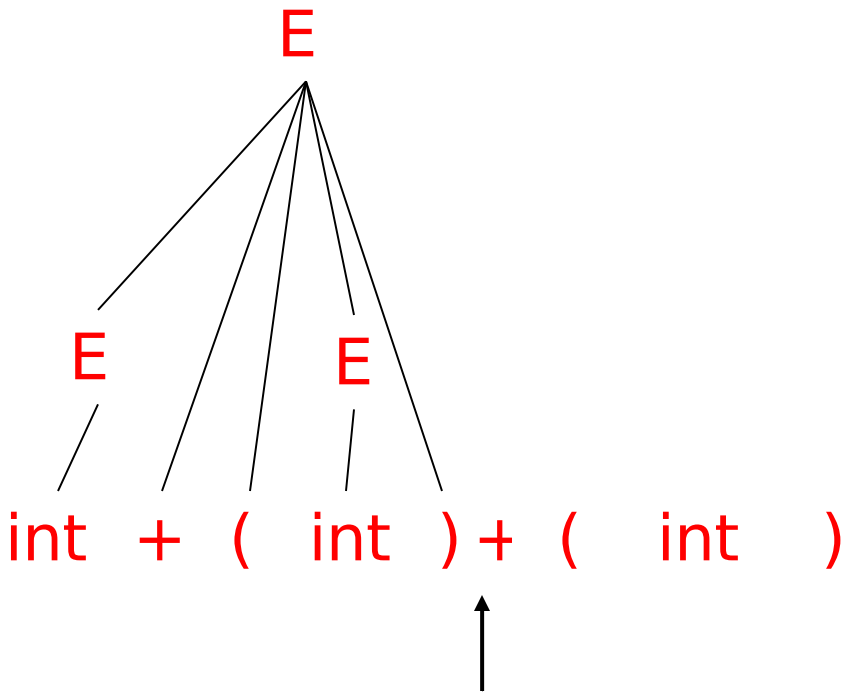
Shift-Reduce Example

▶ 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)



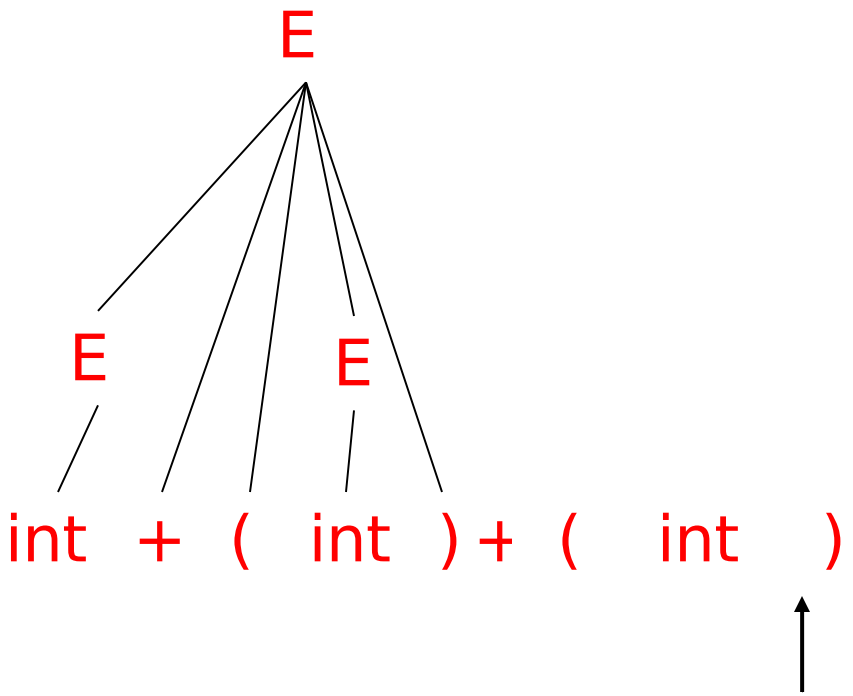
Shift-Reduce Example

▶ 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



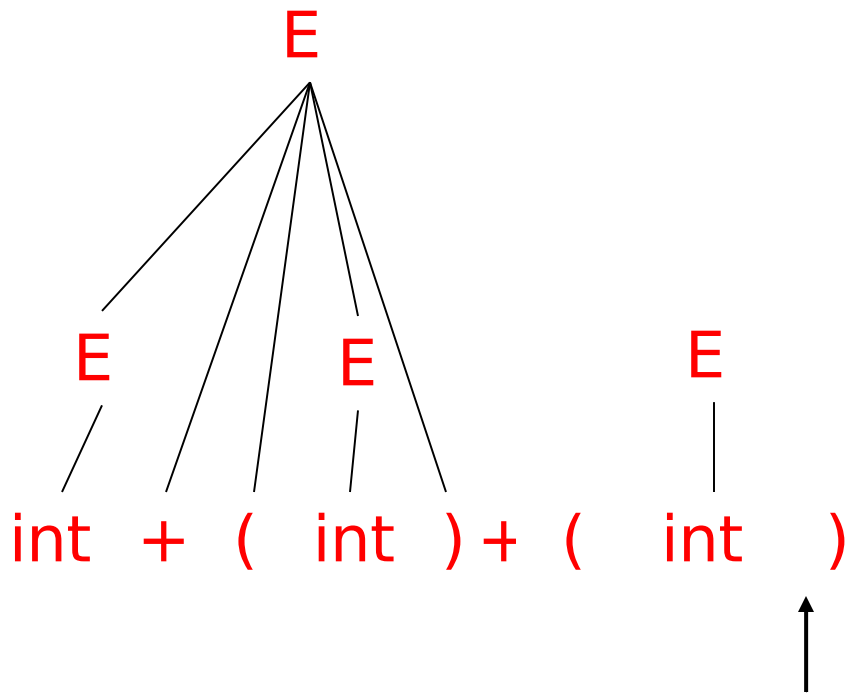
Shift-Reduce Example

▶ 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



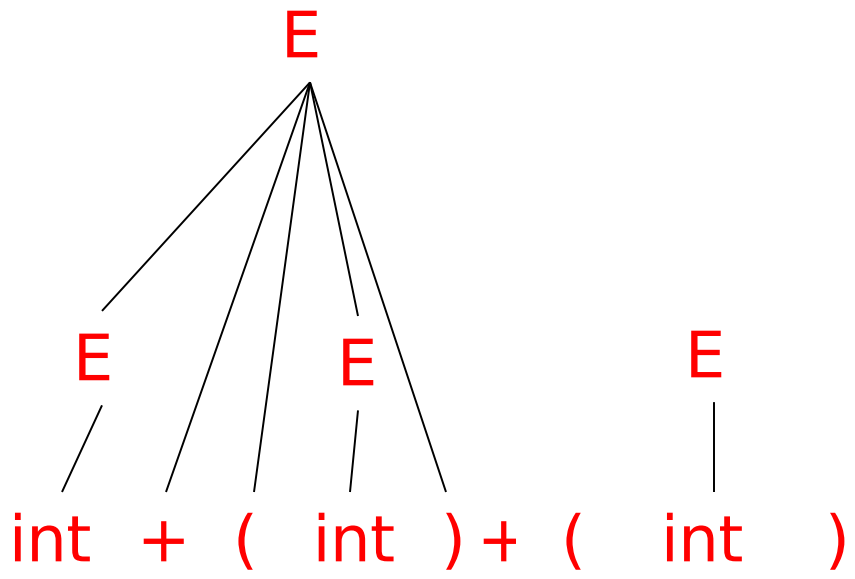
Shift-Reduce Example

▶ 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



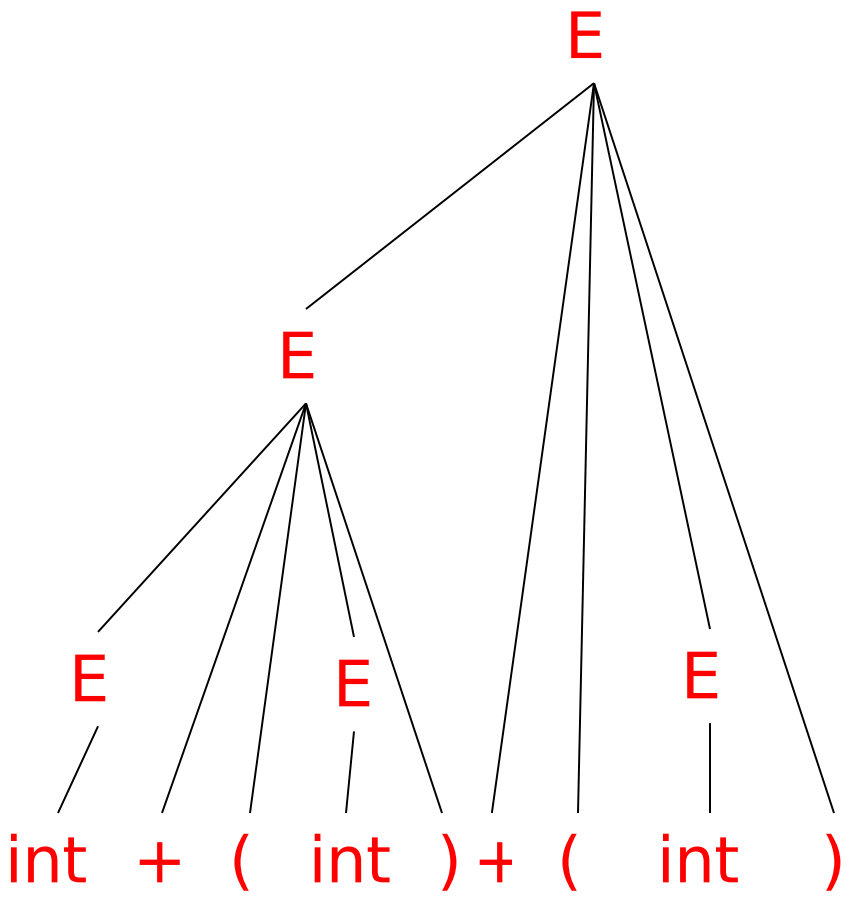
Shift-Reduce Example

▶ 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

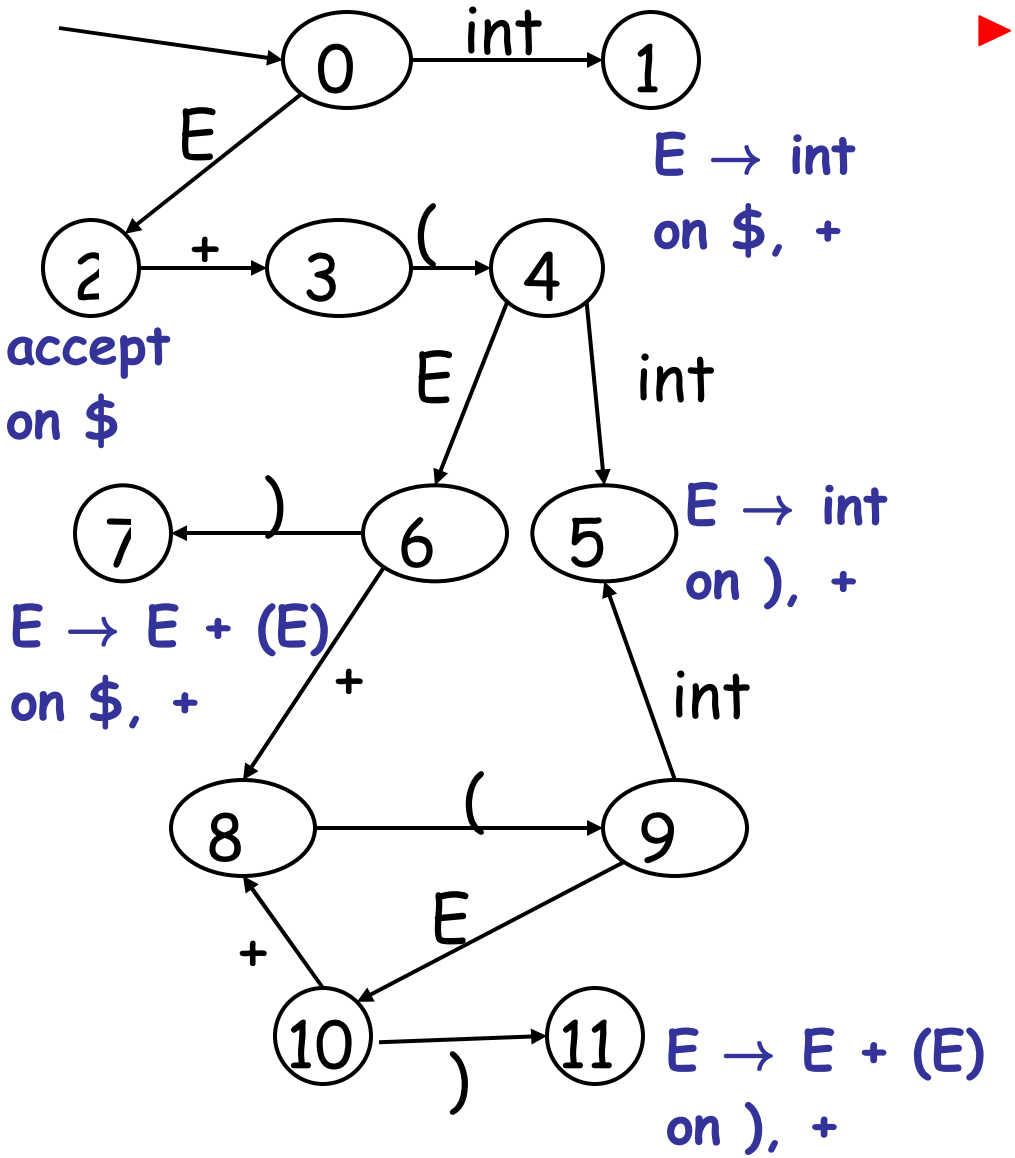
- Left string can be implemented **as a stack**
  - Top of the stack is the ▸
- Shift pushes a terminal on the stack
- Reduce pops 0 or more symbols from the stack (production RHS) and pushes a non-terminal on the stack (production LHS)

Key Issue: When to Shift or Reduce?

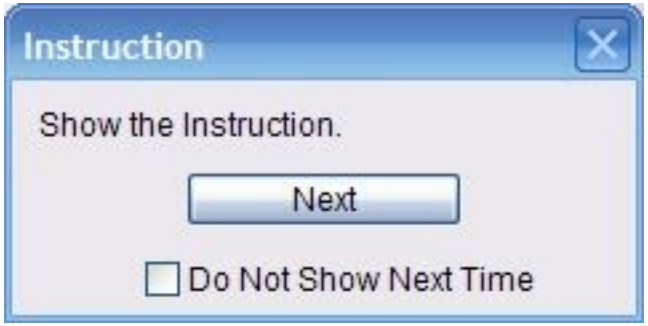
- Decide based on the **left string** (**the stack**)
- Idea: use a finite automaton (DFA) to decide when to shift or reduce
  - The DFA input is the stack (this is tricky!)
  - DFA language consists of terminals and nonterminals

- 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 → β on tok**" then **reduce**

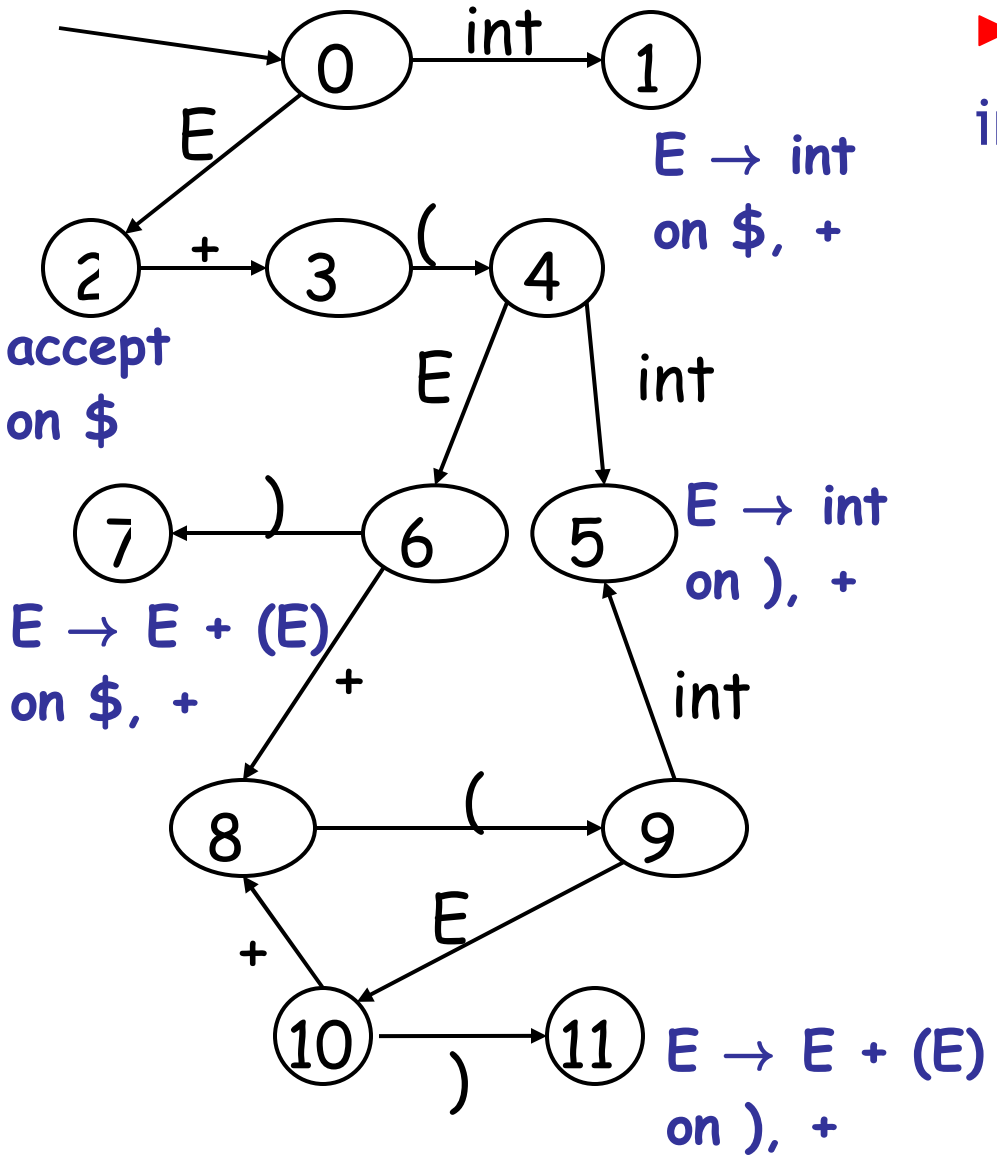
LR(1) Parsing Example



- int + (int) + (int)$



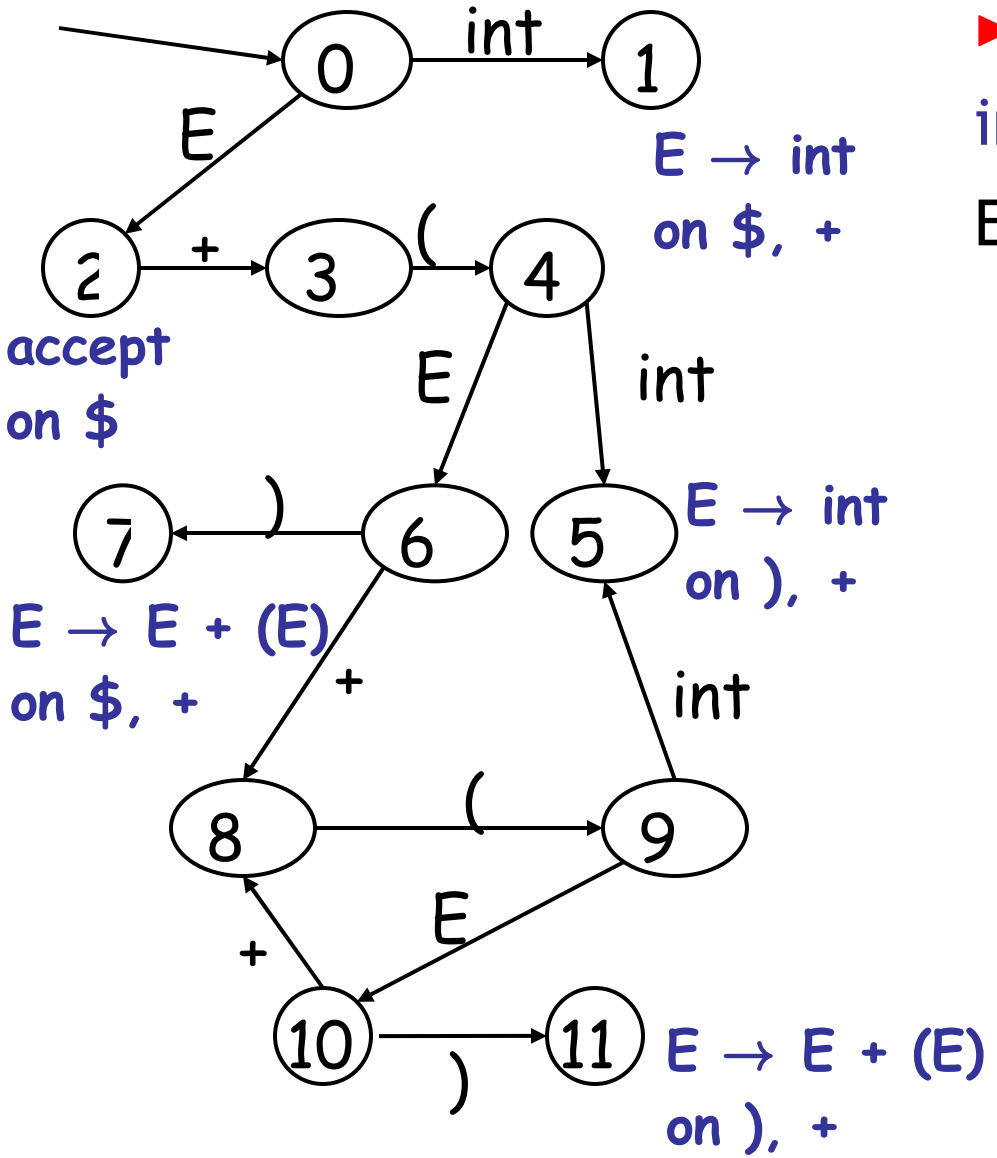
LR(1) Parsing Example



- ▶ int + (int) + (int)$ shift

int ▶ + (int) + (int)$

LR(1) Parsing Example

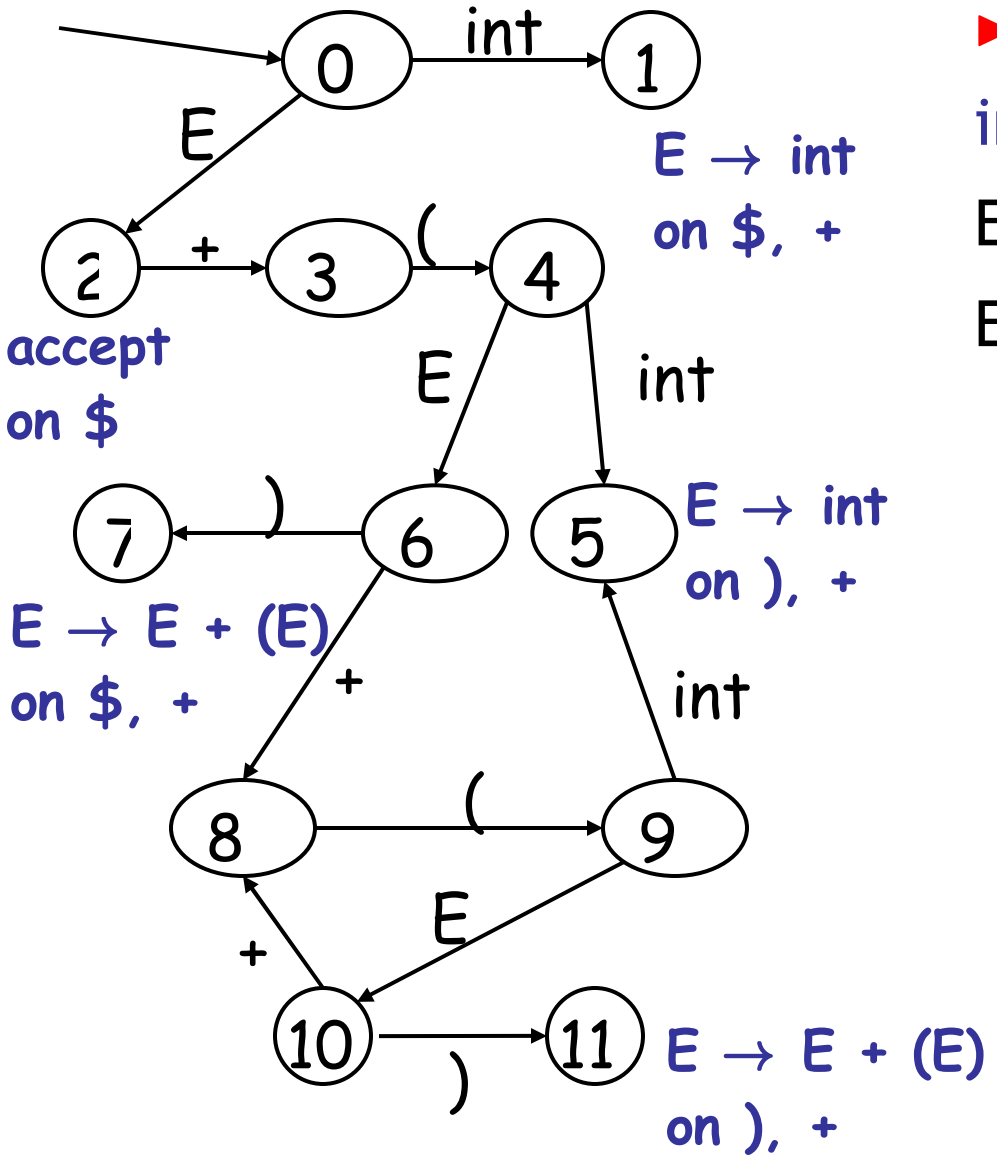


▶ int + (int) + (int)$ shift

int ▶ + (int) + (int)$ E → int

E ▶ + (int) + (int)$

LR(1) Parsing Example



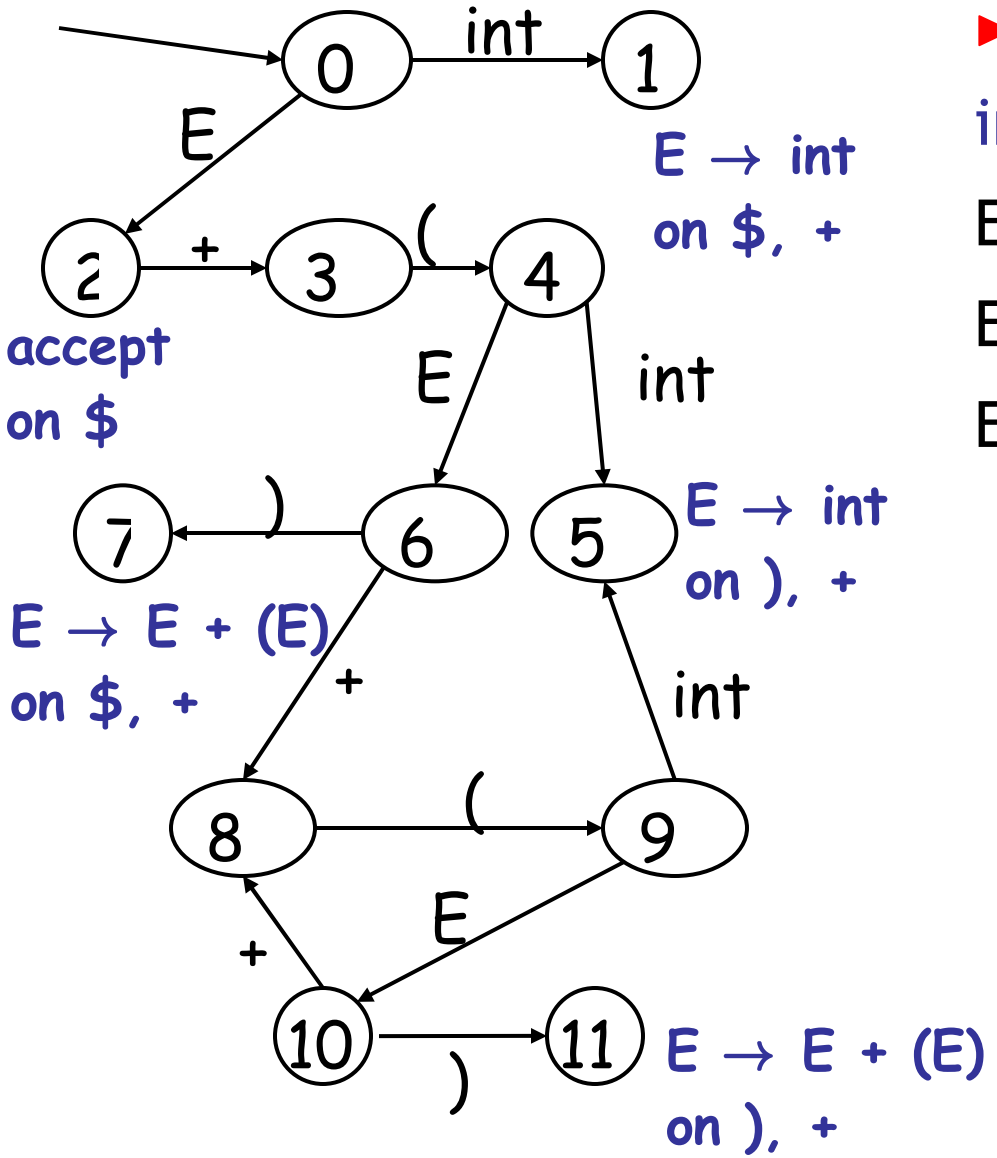
▶ int + (int) + (int)$ shift

int ▶ + (int) + (int)$ E → int

E ▶ + (int) + (int)$ shift(x3)

E + (int ▶) + (int)$

LR(1) Parsing Example



▶ int + (int) + (int)$ shift

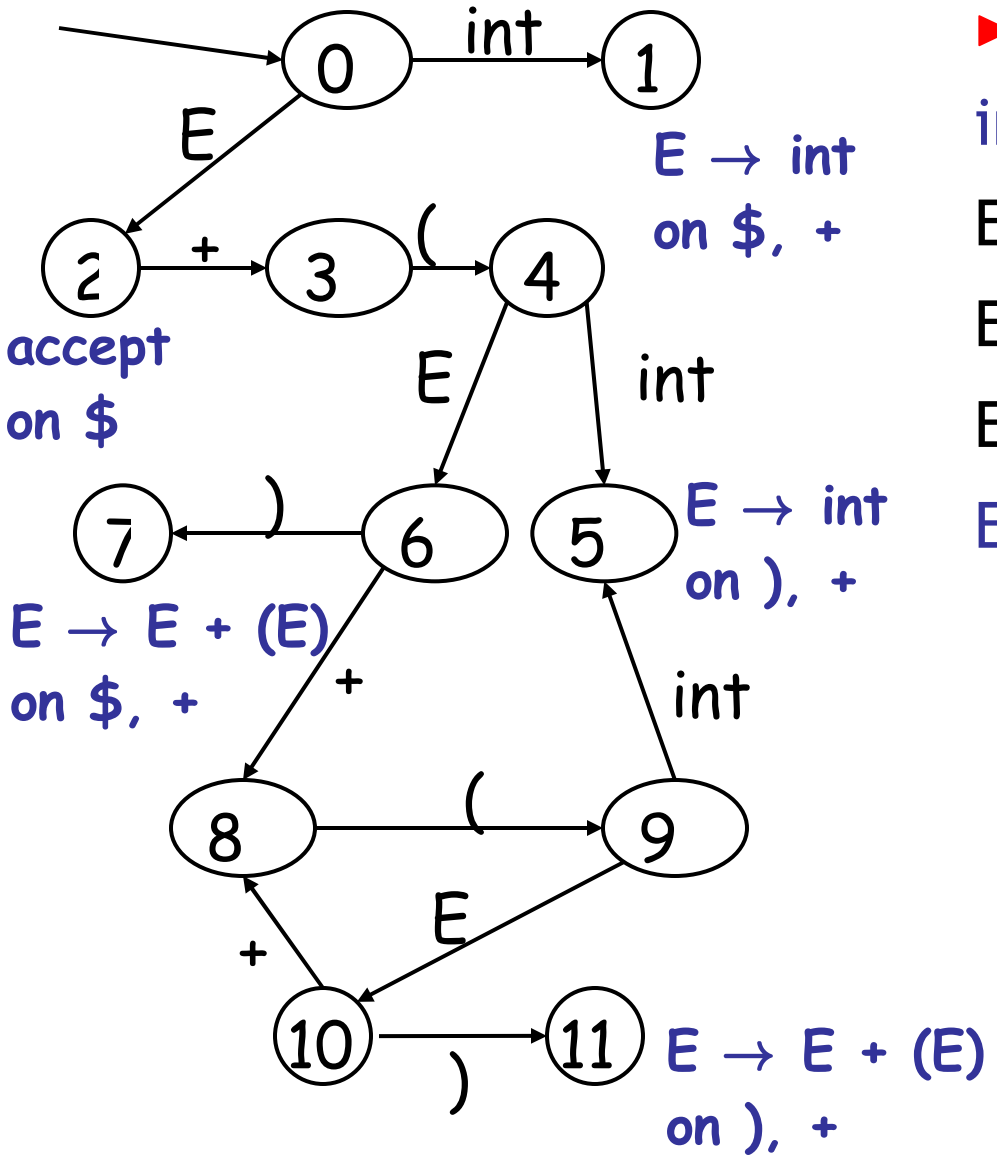
int ▶ + (int) + (int)$ E → int

E ▶ + (int) + (int)$ shift(x3)

E + (int ▶) + (int)$ E → int

E + (E ▶) + (int)$

LR(1) Parsing Example



▶ int + (int) + (int)$ shift

int ▶ + (int) + (int)$ E → int

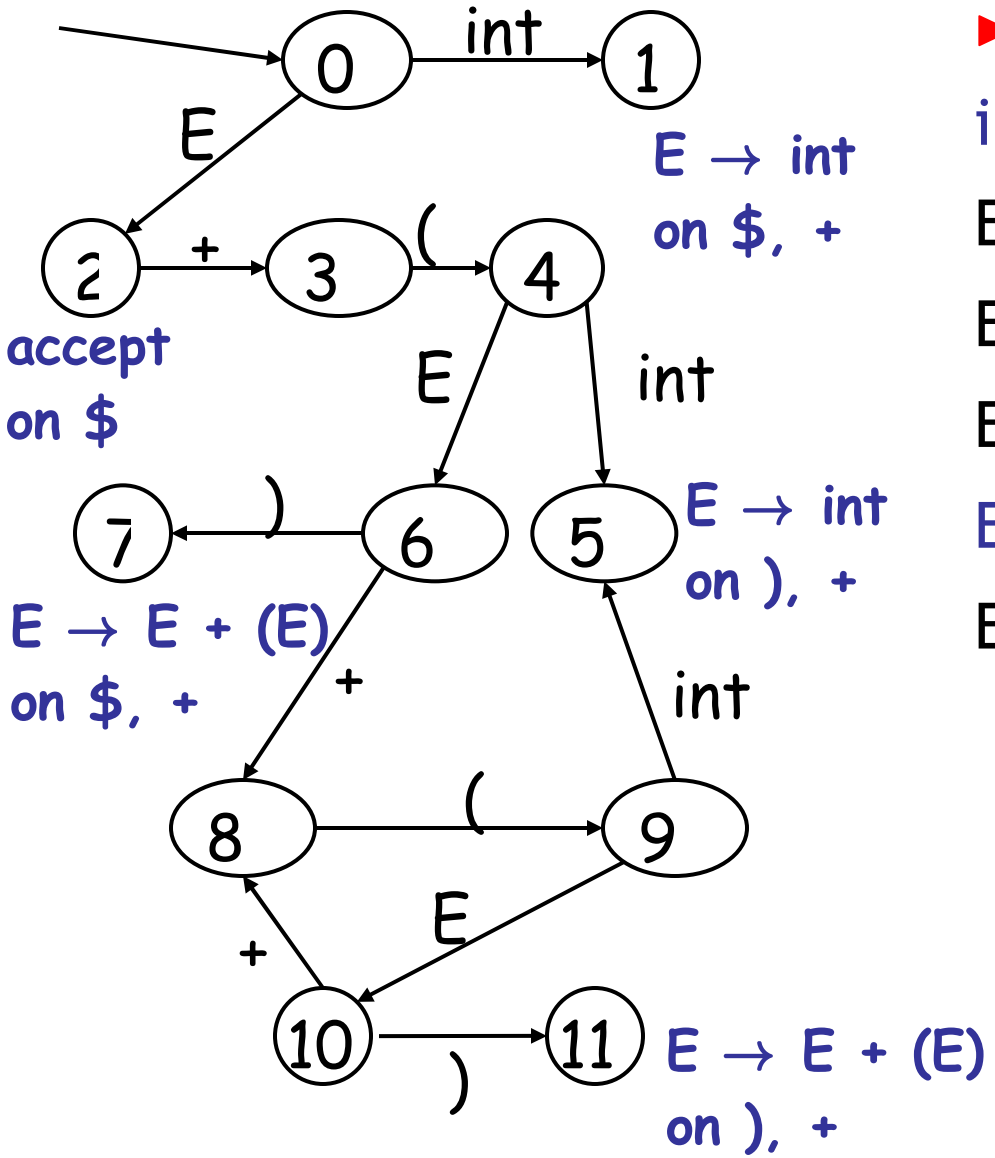
E ▶ + (int) + (int)$ shift(x3)

E + (int ▶) + (int)$ E → int

E + (E ▶) + (int)$ shift

E + (E) ▶ + (int)$

LR(1) Parsing Example



▶ int + (int) + (int)$ shift

int ▶ + (int) + (int)$ E → int

E ▶ + (int) + (int)$ shift(x3)

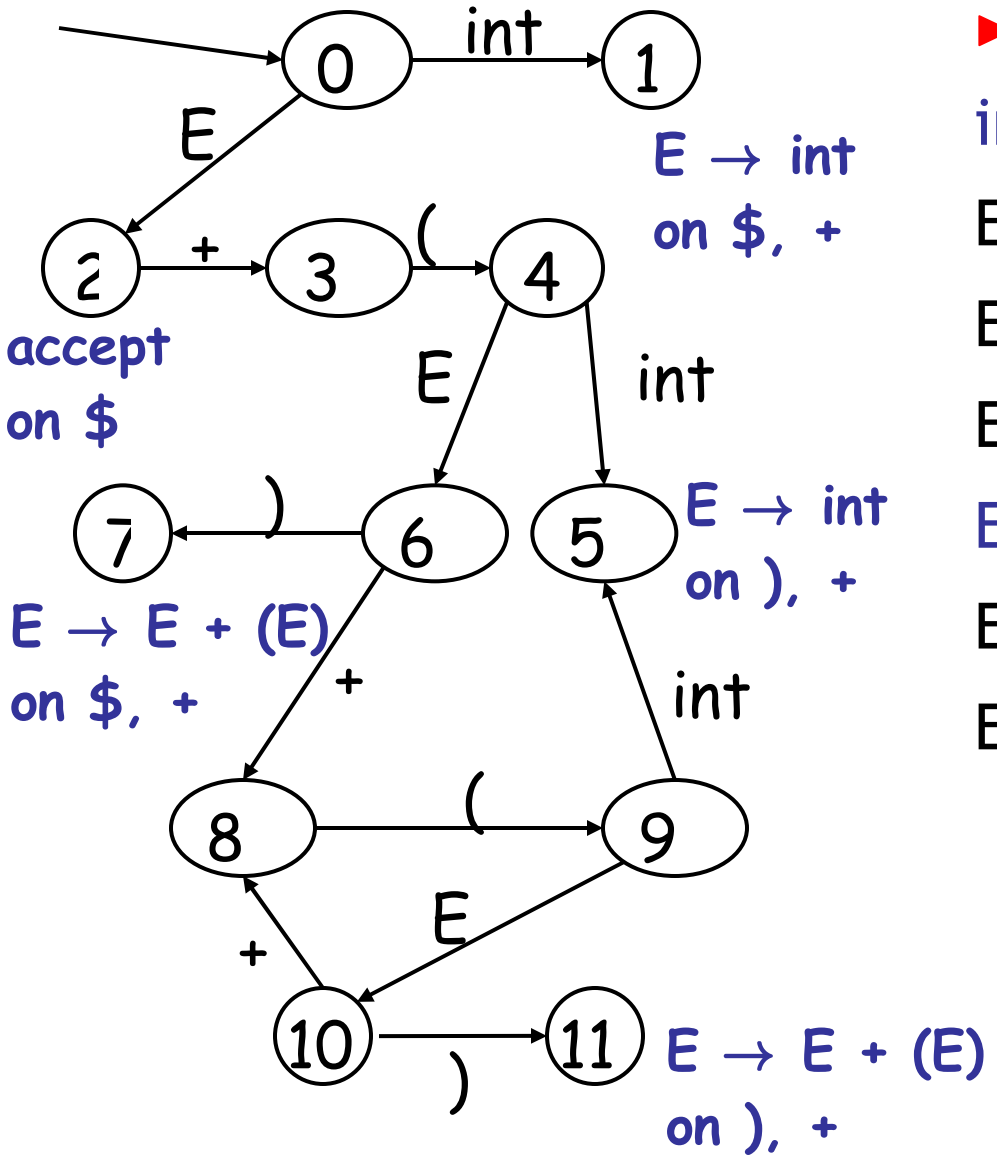
E + (int ▶) + (int)$ E → int

E + (E ▶) + (int)$ shift

E + (E) ▶ + (int)$ E → E+(E)

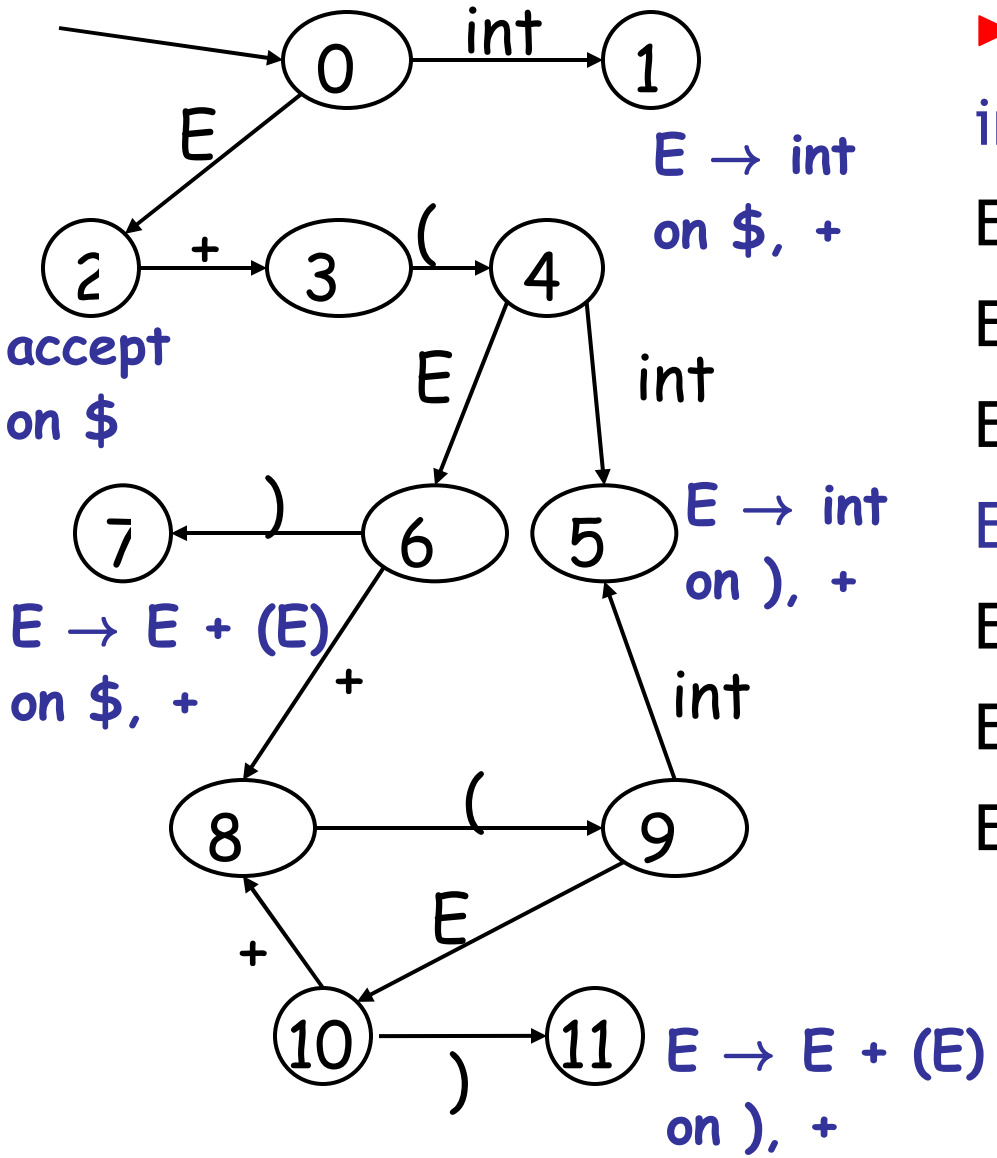
E ▶ + (int)$

LR(1) Parsing 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 ▶)$	

LR(1) Parsing Example



▸ int + (int) + (int)$ shift

int ▸ + (int) + (int)$ E → int

E ▸ + (int) + (int)$ shift(x3)

E + (int ▸) + (int)$ E → int

E + (E ▸) + (int)$ shift

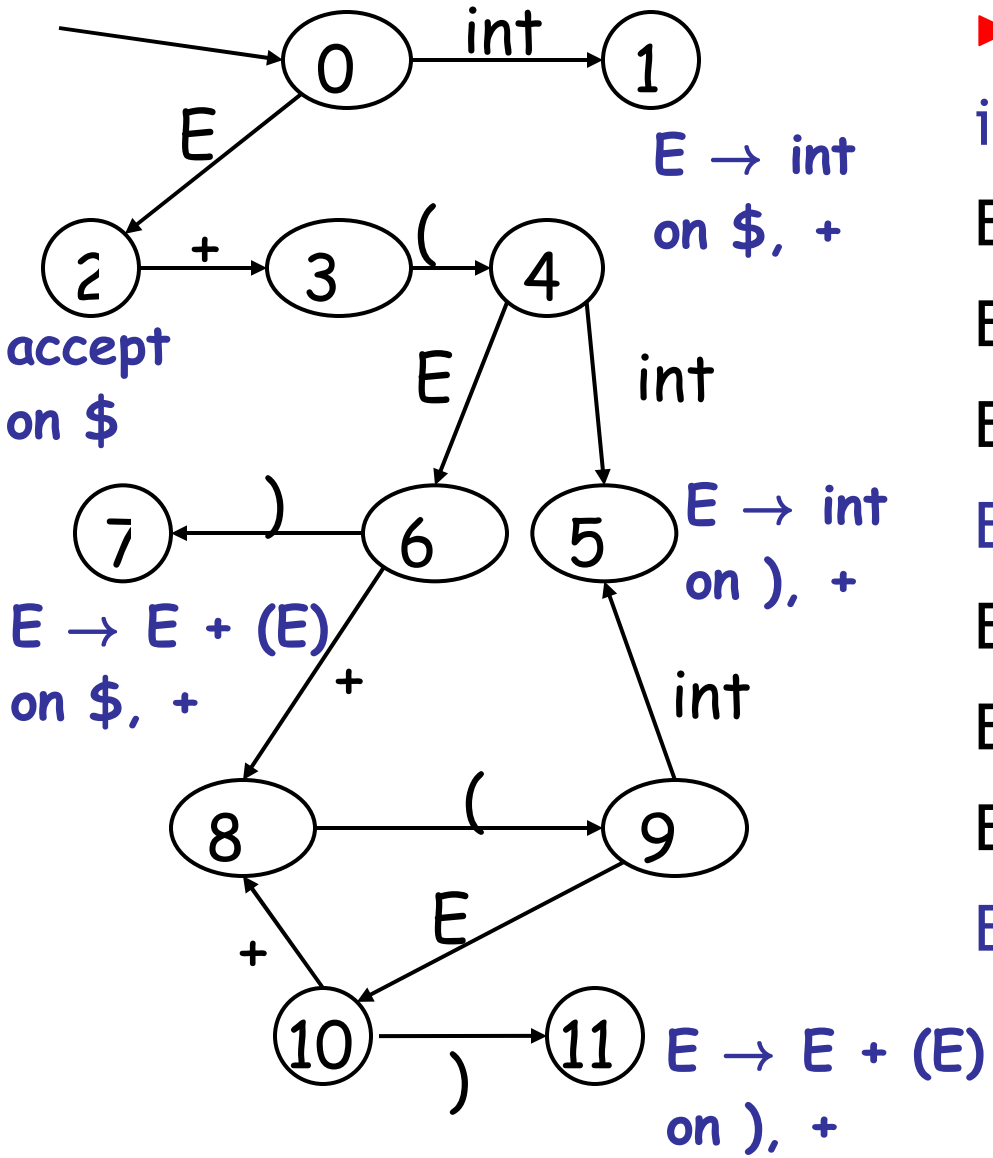
E + (E) ▸ + (int)$ E → E+(E)

E ▸ + (int)$ shift (x3)

E + (int ▸)$ E → int

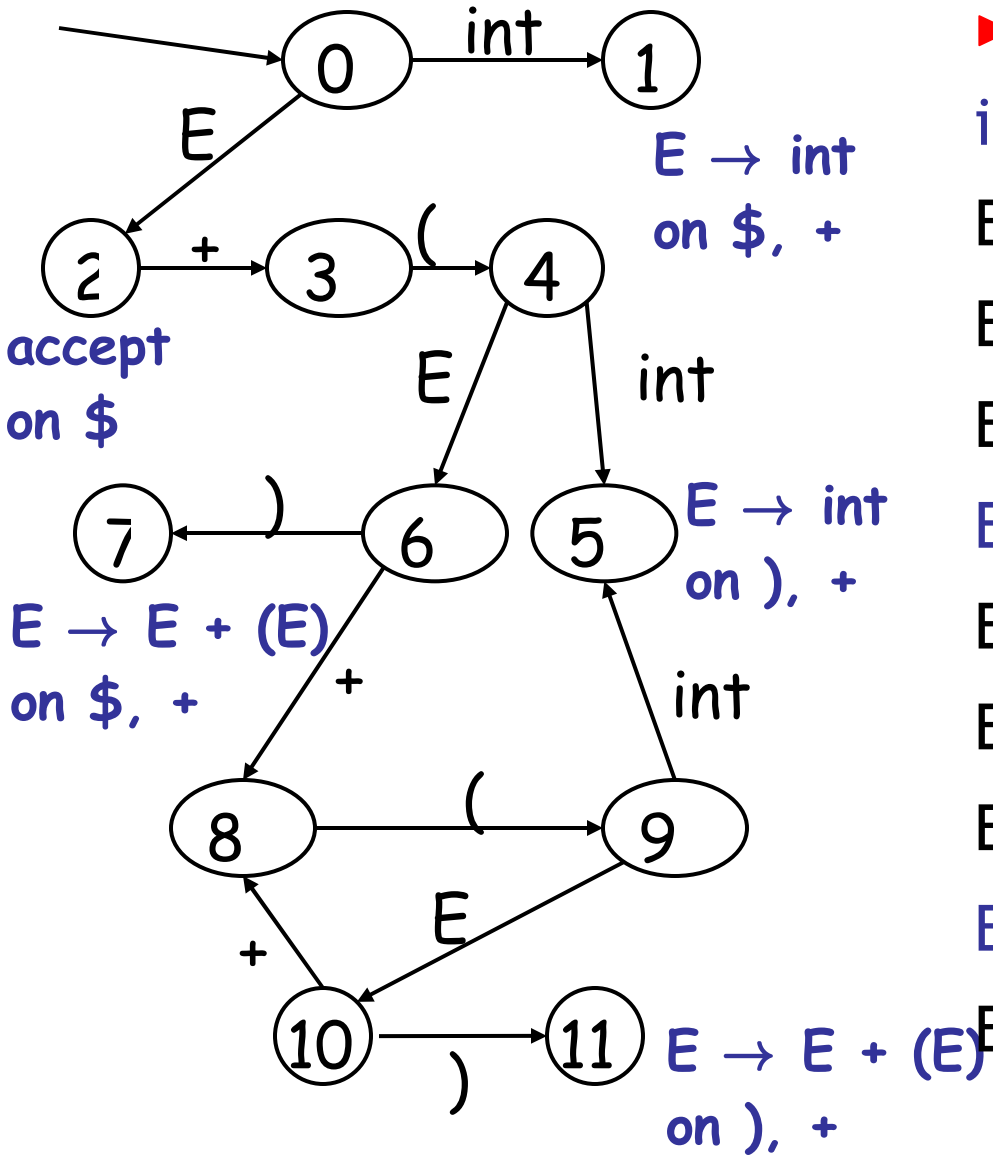
E + (E ▸)$

LR(1) Parsing 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) ▸ $	

LR(1) Parsing Example



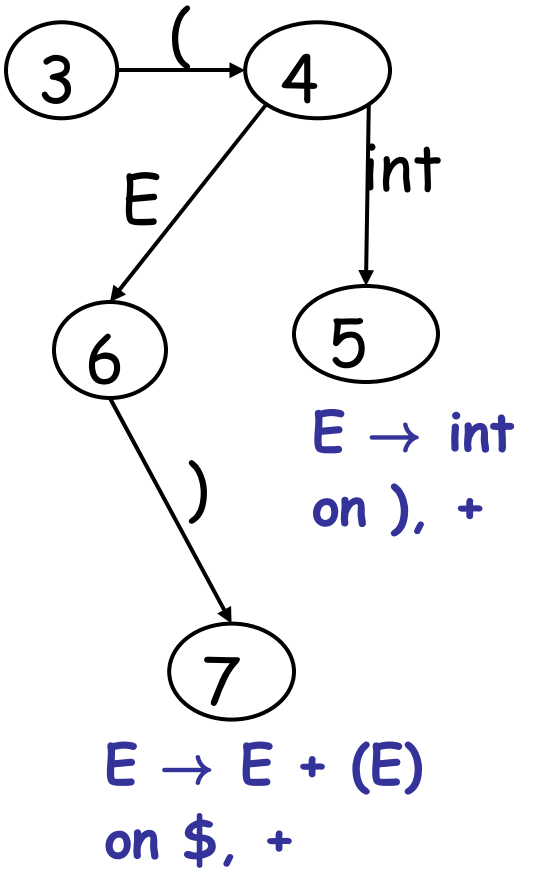
▸ int + (int) + (int)$	shift
int ▸ + (int) + (int)$	E → int
E ▸ + (int) + (int)$	shift(x3)
E + (int ▸) + (int)$	E → int
E + (E ▸) + (int)$	shift
E + (E) ▸ + (int)$	E → E+(E)
E ▸ + (int)$	shift (x3)
E + (int ▸)$	E → int
E + (E ▸)$	shift
E + (E) ▸ $	E → E+(E)
E ▸ $	accept

Representing the DFA

- Parsers represent the DFA as a 2D table
  - Recall table-driven lexical analysis
- Lines (rows) 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

- Optimization: remember for each stack element to which state it brings the DFA

- LR parser maintains a stack

$$< \mathbf{sym}_1, \mathbf{state}_1 > \ldots < \mathbf{sym}_n, \mathbf{state}_n >$$

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

The LR Parsing Algorithm

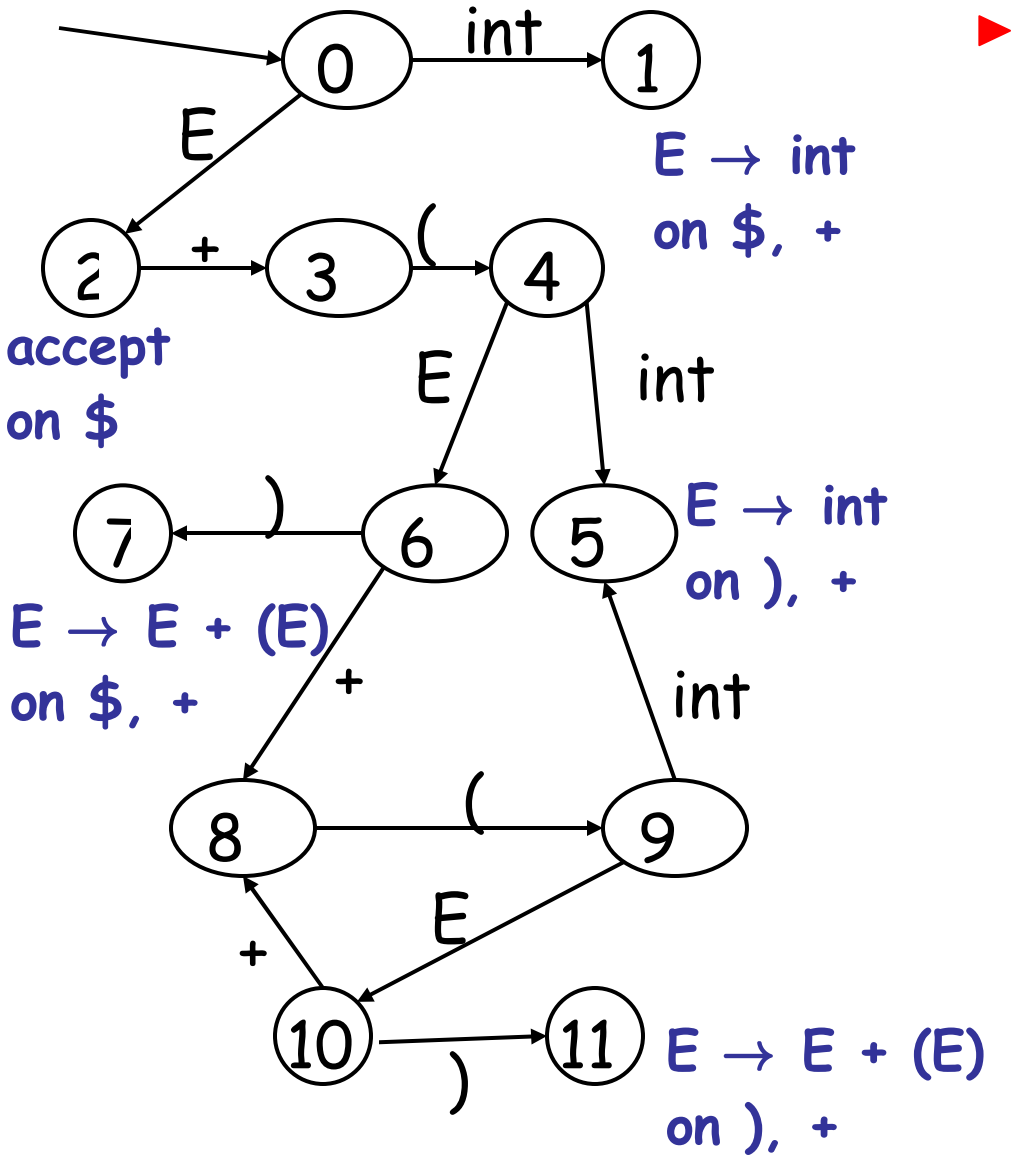
```
Let S = w$ be initial input
Let j = 0
Let DFA state 0 be the start state
Let stack = < dummy, 0 >
 repeat
        match action[top_state(stack), S[j]] with
                | shift k:  push < S[j++], k >
                | reduce X → α:
                     pop |α| pairs,
                     push < 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 PL grammars are LR
- Can be described as a simple table
- There are tools for building the table
  - Often called “yacc” or “bison”

- How is the table constructed? Next time!

Son of LR(1) Parsing Example



- int + (int + (int)) $

Homework

- Tuesday: WA2 due
- Tuesday: Read 2.3.4-2.3.5, 2.4.2-2.4.3
- **Wednesday Sep 30: PA3 due**
  - Parsing!
  - When should I start? Early or late?