

Bottom Up Parsing

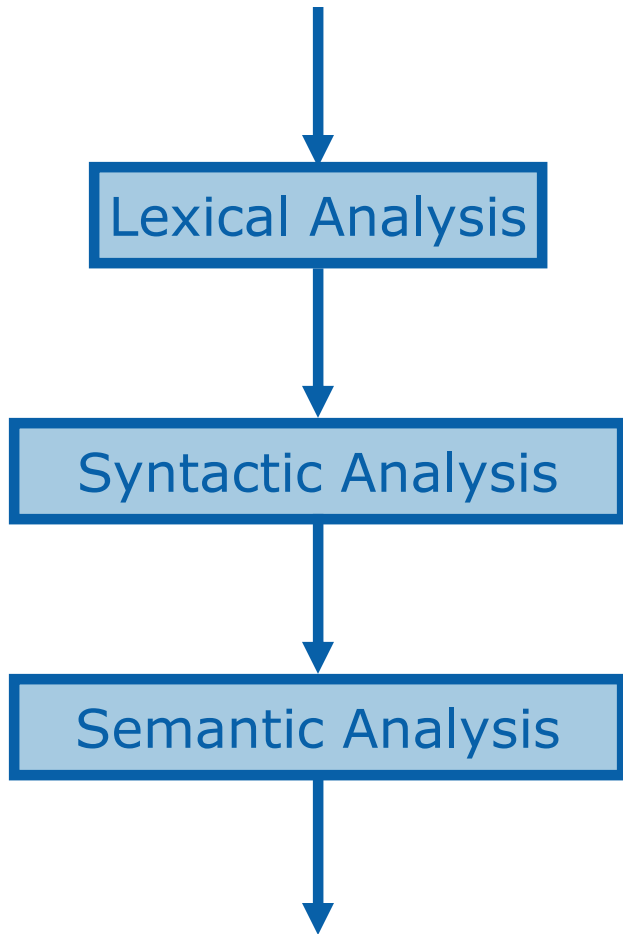
Compiler Construction EECS 483

Winter 2011

Lecture 6

Ack: Kim Hazelwood

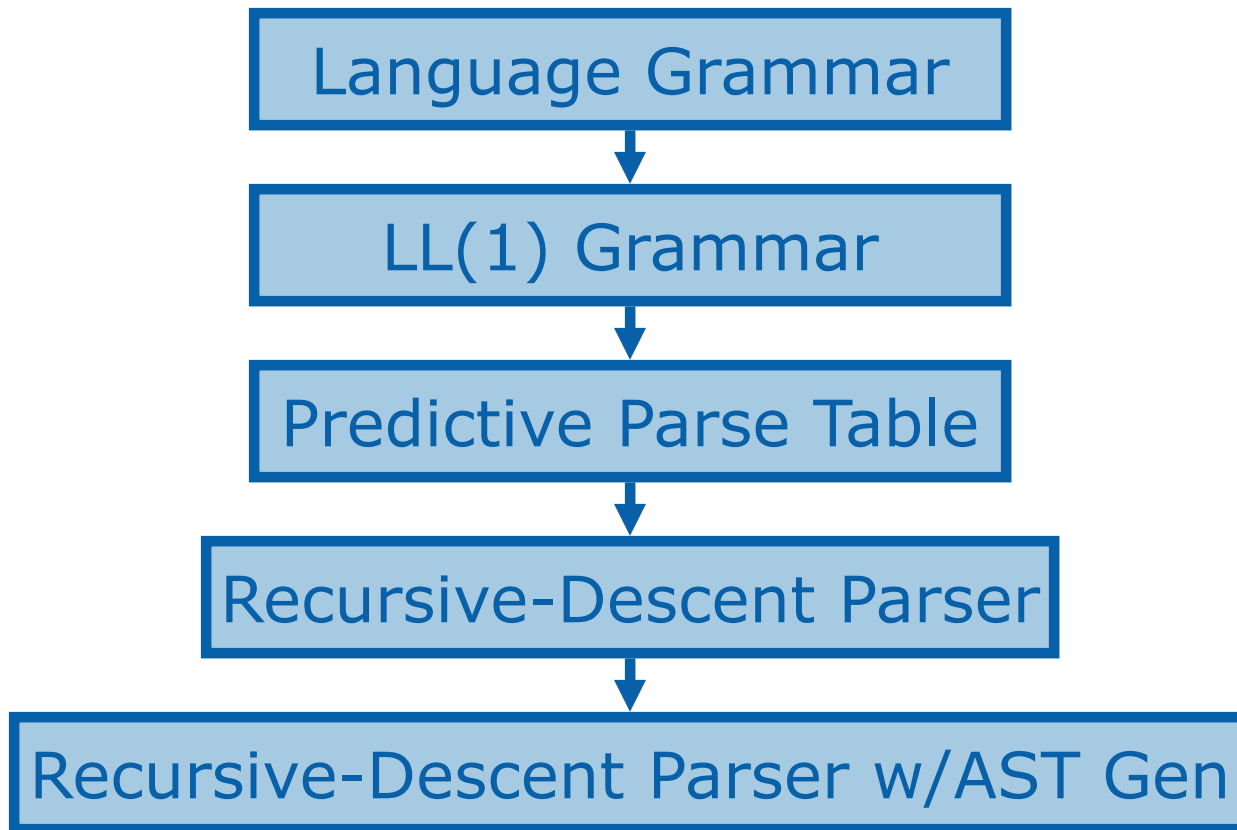
Where Are We?



**Finished Top-Down Parsing
Starting Bottom-Up Parsing**

Last Time ... Building a LL Parser

Have a complete recipe for building a parser



Bottom-Up Parsing

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

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., $\text{str} = \alpha \beta \gamma$)
- Replace β by A in str (i.e., str becomes $\alpha A \gamma$)

until str = G

A Bottom-up Parse in Detail (1)

int + (int) + (int)

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

int + (int) + (int)

A Bottom-up Parse in Detail (2)

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

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

E
|
int + (int) + (int)

A Bottom-up Parse in Detail (3)

int + (int) + (int)

E + (int) + (int)

E + (E) + (int)

E → int
E → E + (E)

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

A Bottom-up Parse in Detail (4)

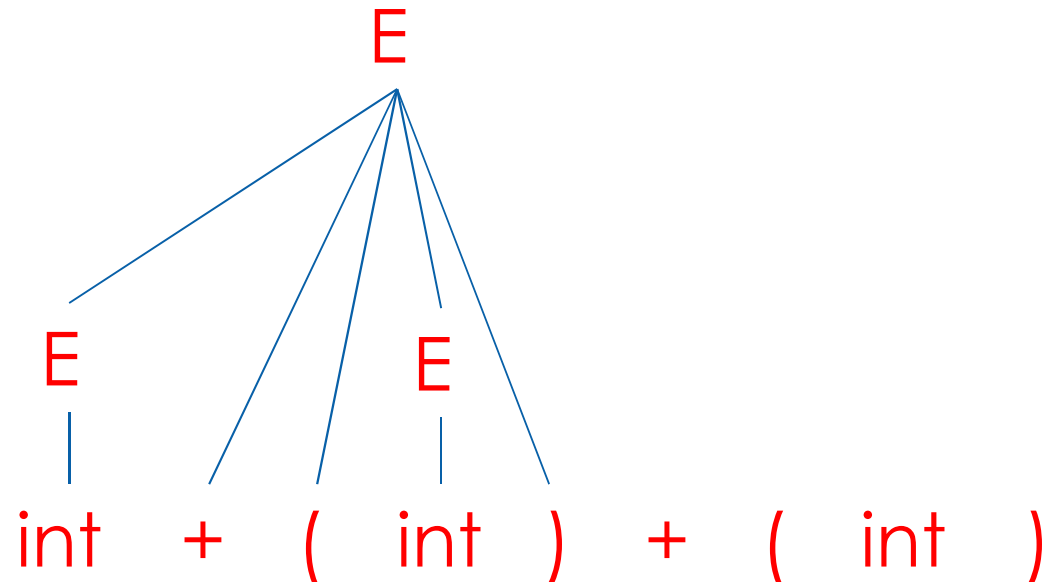
int + (int) + (int)

E + (int) + (int)

E + (E) + (int)

E + (int)

E → int
E → E + (E)



A Bottom-up Parse in Detail (5)

int + (int) + (int)

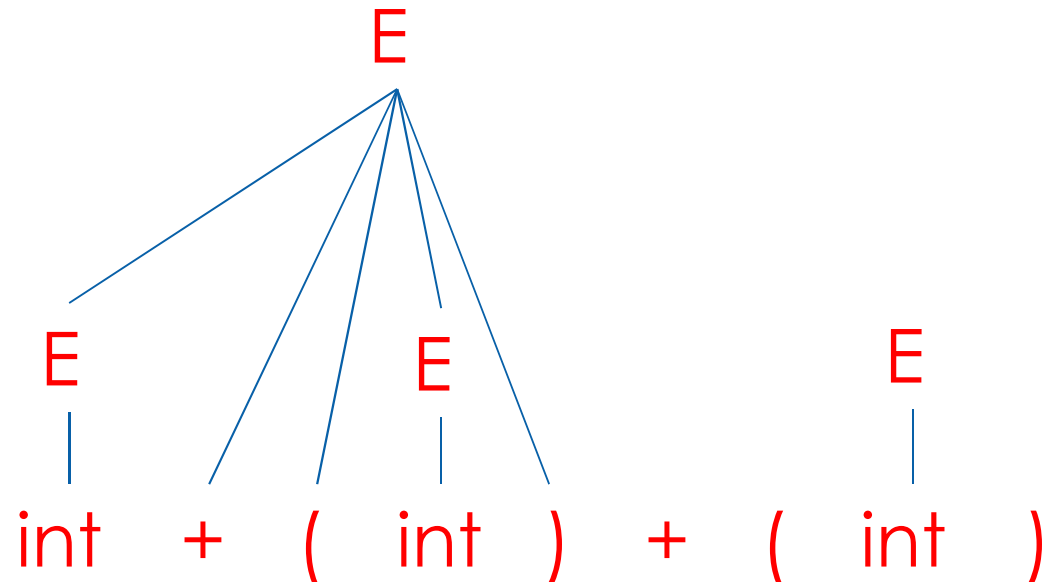
E + (int) + (int)

E + (E) + (int)

E + (int)

E + (E)

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



Notation

Split input 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 \uparrow

- The \uparrow is not part of the string

Initially, all input is unexamined: $\uparrow x_1 x_2 \dots x_n$

Shift-Reduce Parsing

Bottom-up parsing uses only two kinds of actions: **Shift** and **Reduce**

Shift: Move \uparrow one place to the right

- Shifts a terminal to the left string

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

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

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

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

Shift-Reduce Example

↑ **int** + (int) + (int)\$ **shift**

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

int + (int) + (int)
↑

Shift-Reduce Example

\uparrow int + (int) + (int)\$ **shift**
int \uparrow + (int) + (int)\$ **red. E \rightarrow int**

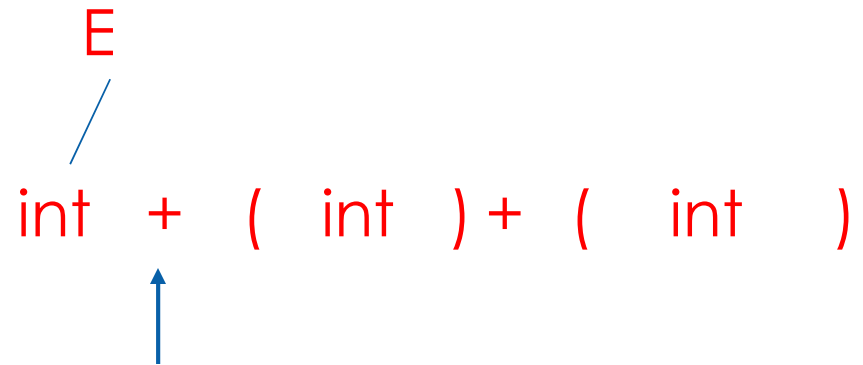
E \rightarrow int
E \rightarrow E + (E)

int + (int) + (int)
 ↑

Shift-Reduce Example

\uparrow int + (int) + (int)\$ **shift**
int \uparrow + (int) + (int)\$ **red. $E \rightarrow \text{int}$**
E \uparrow + (int) + (int)\$ **shift 3 times**

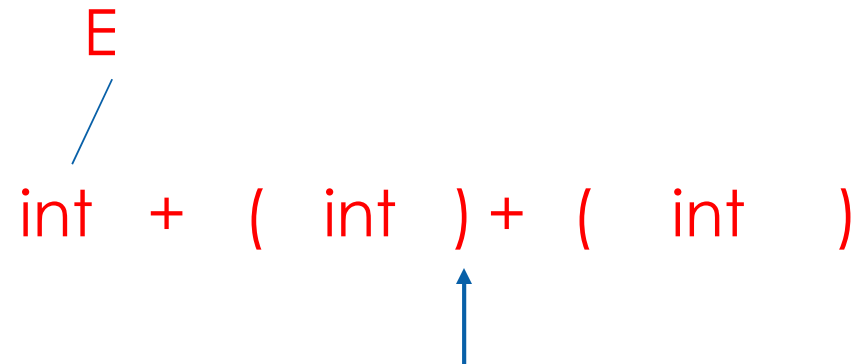
$E \rightarrow \text{int}$
$E \rightarrow E + (E)$



Shift-Reduce Example

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

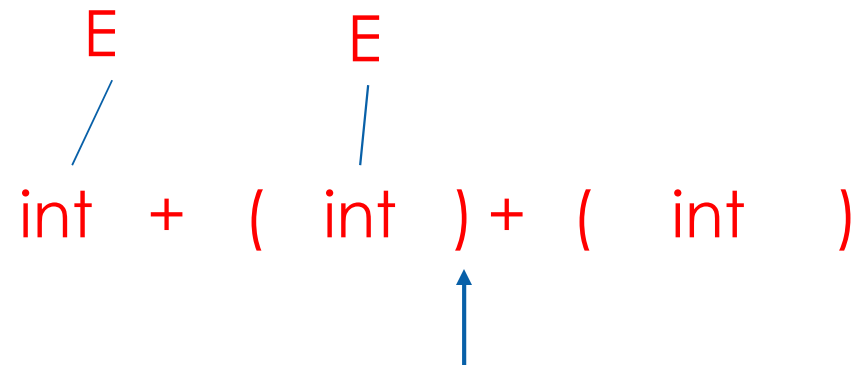
E \rightarrow int
E \rightarrow E + (E)



Shift-Reduce Example

\uparrow int + (int) + (int)\$ **shift**
int \uparrow + (int) + (int)\$ **red. $E \rightarrow \text{int}$**
E \uparrow + (int) + (int)\$ **shift 3 times**
E + (int \uparrow) + (int)\$ **red. $E \rightarrow \text{int}$**
E + (E \uparrow) + (int)\$ **shift**

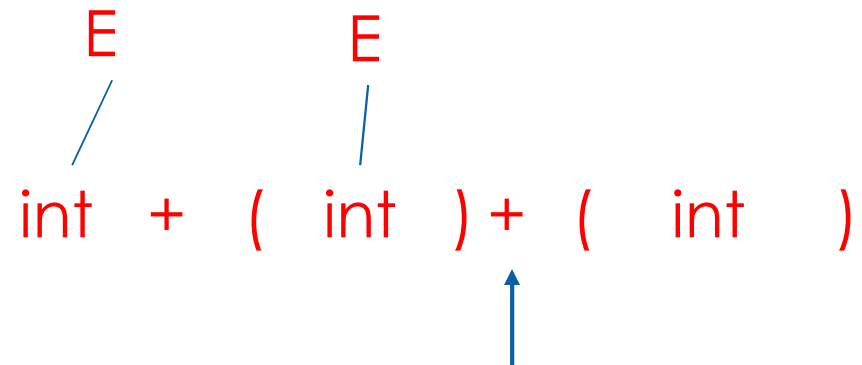
$E \rightarrow \text{int}$
$E \rightarrow E + (E)$



Shift-Reduce Example

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

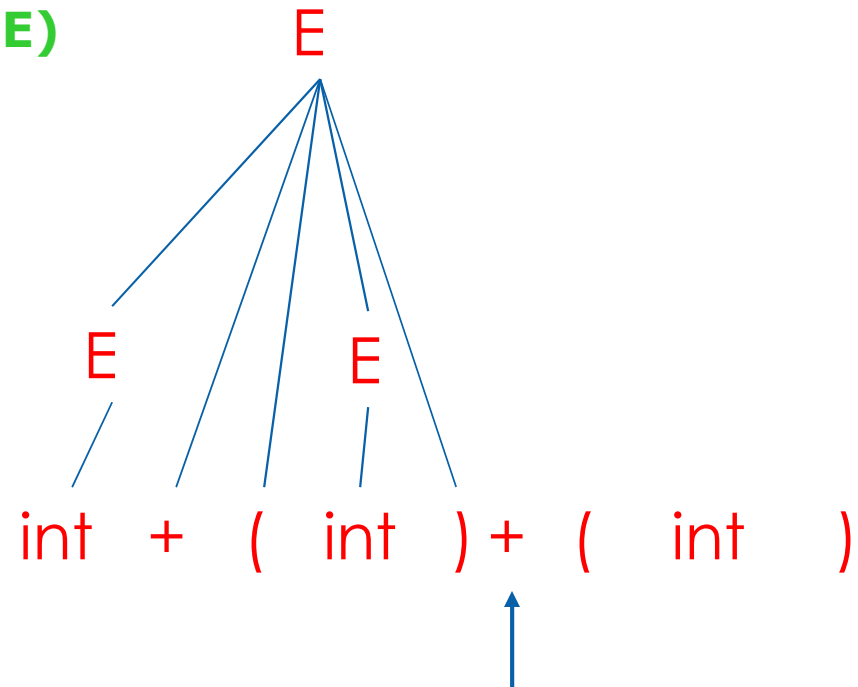
E \rightarrow int
E \rightarrow E + (E)



Shift-Reduce Example

\uparrow int + (int) + (int)\$ shift
int \uparrow + (int) + (int)\$ red. $E \rightarrow \text{int}$
E \uparrow + (int) + (int)\$ shift 3 times
E + (int \uparrow) + (int)\$ red. $E \rightarrow \text{int}$
E + (E \uparrow) + (int)\$ shift
E + (E) \uparrow + (int)\$ red. $E \rightarrow E + (E)$
E \uparrow + (int)\$ shift 3 times

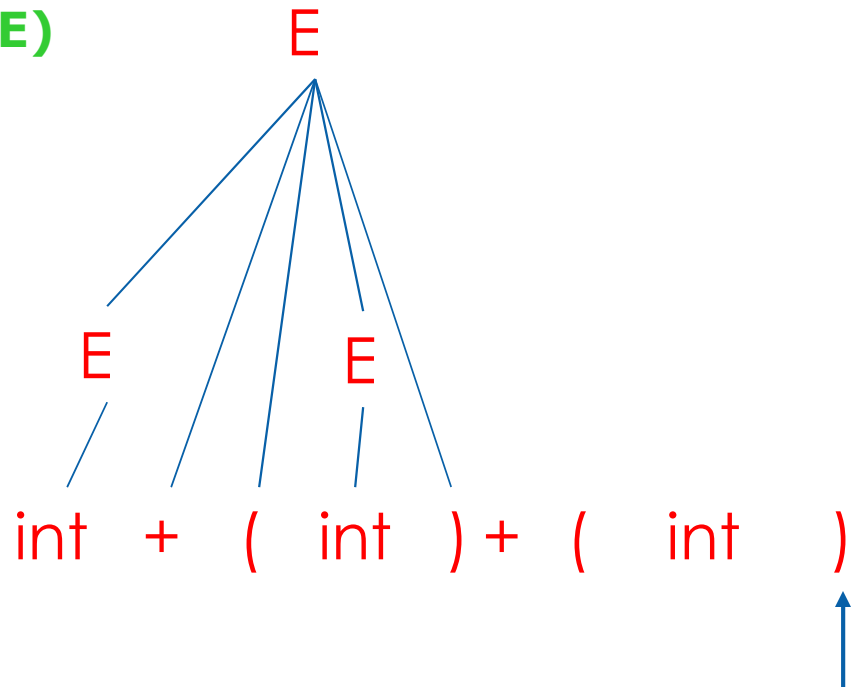
$E \rightarrow \text{int}$
$E \rightarrow E + (E)$



Shift-Reduce Example

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

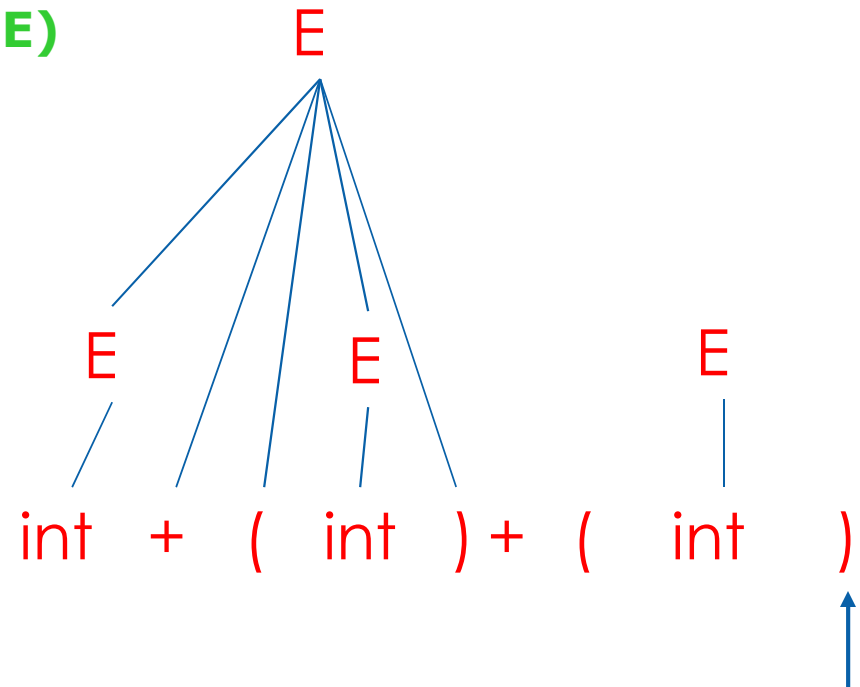
$E \rightarrow \text{int}$
 $E \rightarrow E + (E)$



Shift-Reduce Example

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

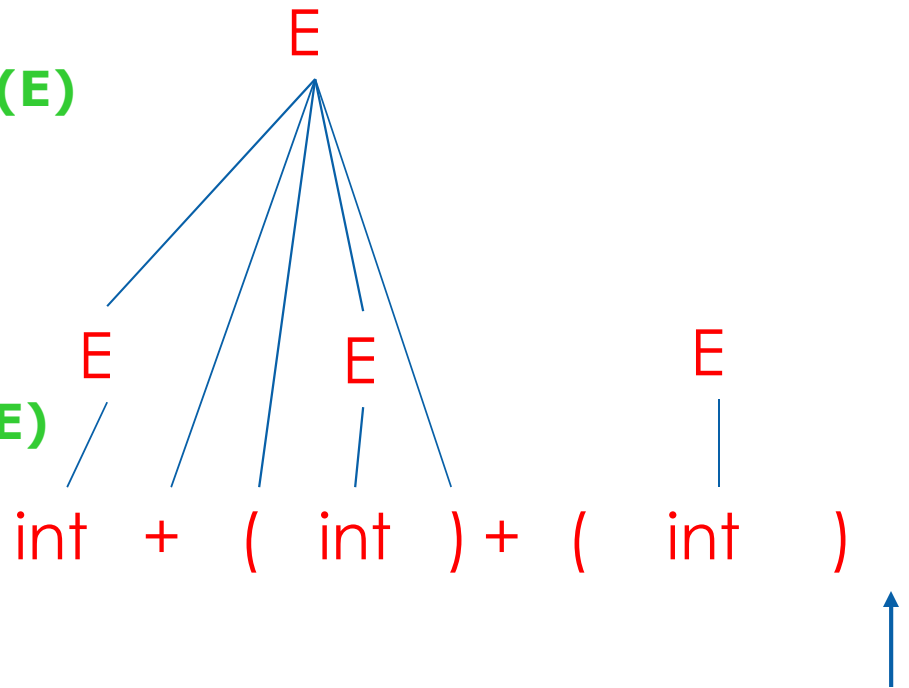
$E \rightarrow \text{int}$
 $E \rightarrow E + (E)$



Shift-Reduce Example

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

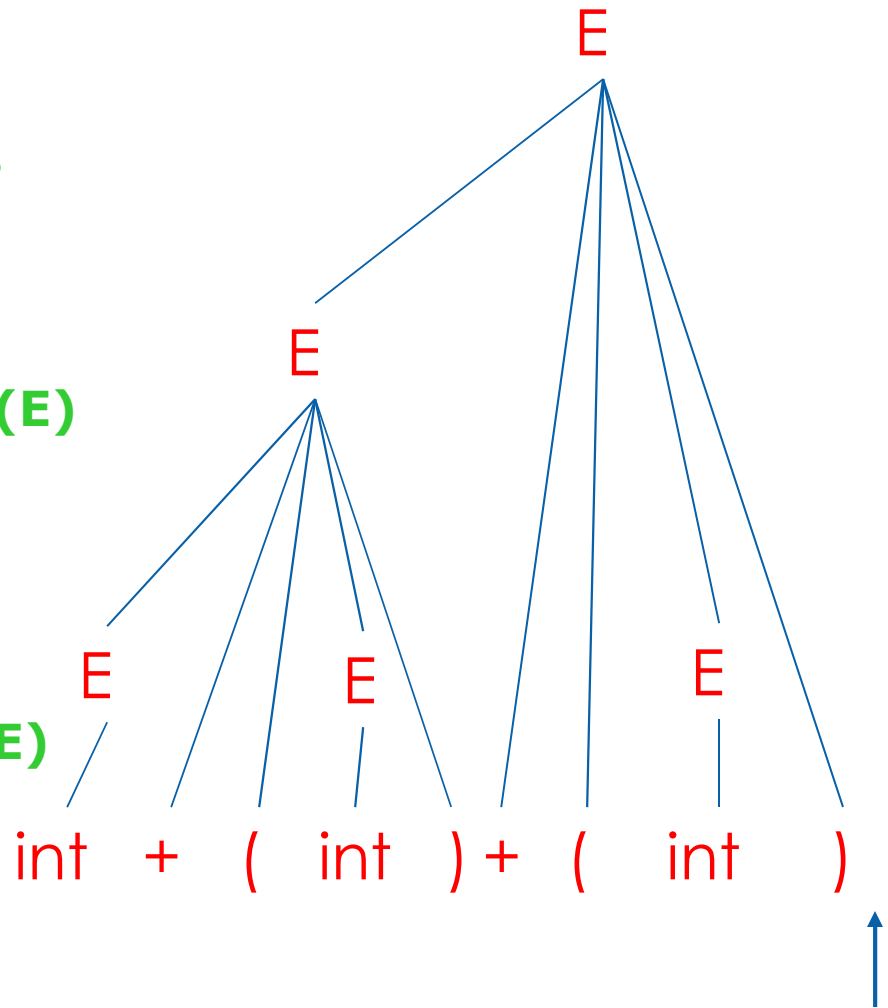
$E \rightarrow \text{int}$
 $E \rightarrow E + (E)$



Shift-Reduce Example

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

$\uparrow \text{int} + (\text{int}) + (\text{int})\$$ **shift**
 $\text{int} \uparrow + (\text{int}) + (\text{int})\$$ **red. $E \rightarrow \text{int}$**
 $E \uparrow + (\text{int}) + (\text{int})\$$ **shift 3 times**
 $E + (\text{int} \uparrow) + (\text{int})\$$ **red. $E \rightarrow \text{int}$**
 $E + (E \uparrow) + (\text{int})\$$ **shift**
 $E + (E) \uparrow + (\text{int})\$$ **red. $E \rightarrow E + (E)$**
 $E \uparrow + (\text{int})\$$ **shift 3 times**
 $E + (\text{int} \uparrow) \$$ **red. $E \rightarrow \text{int}$**
 $E + (E \uparrow) \$$ **shift**
 $E + (E) \uparrow \$$ **red. $E \rightarrow E + (E)$**
 $E \uparrow \$$ **accept**



How do we keep track?

Left part string implemented as a **stack**

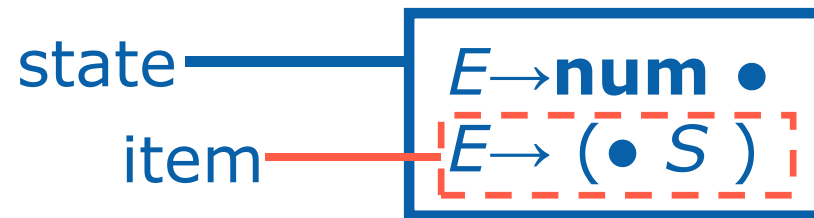
- Top of the stack is the ↑
- **Shift:**
 - Pushes a terminal on the stack
- **Reduce:**
 - Pops 0 or more symbols off of the stack
 - Symbols are right-hand side of a production
 - Pushes a non-terminal on the stack (production LHS)

LR(0) Parser

- **L**eft-to-right scanning, **R**ight-most derivation, “**zero**” look-ahead characters
- Too weak to handle most language grammars (*e.g.*, “sum” grammar)
- But will help us lay the groundwork for more sophisticated parsers

LR(0) States

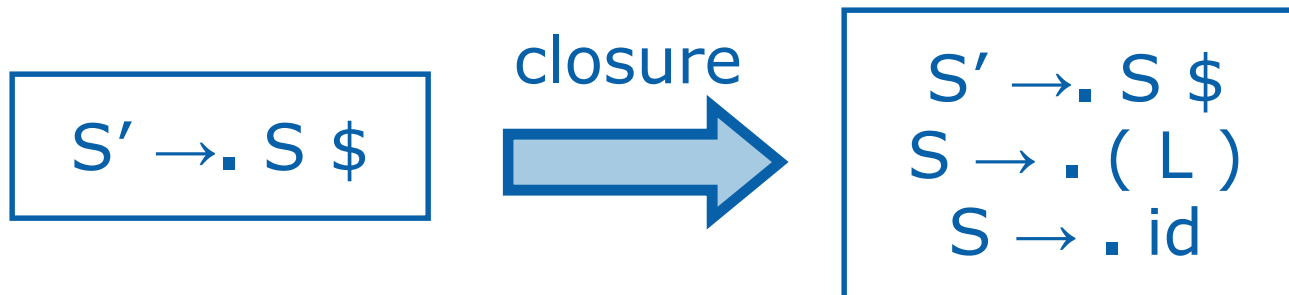
- A **state** is a set of *items* keeping track of progress on possible upcoming reductions
- An *LR(0) item* is a production from the language with a separator "." somewhere in the RHS of the production



- Stuff before "." is already on stack (beginnings of possible γ 's to be reduced)
- Stuff after "." : what we might see next
- The prefixes α represented by state itself

Start State & Closure

$S \rightarrow (L) \mid id$
 $L \rightarrow S \mid L , S$

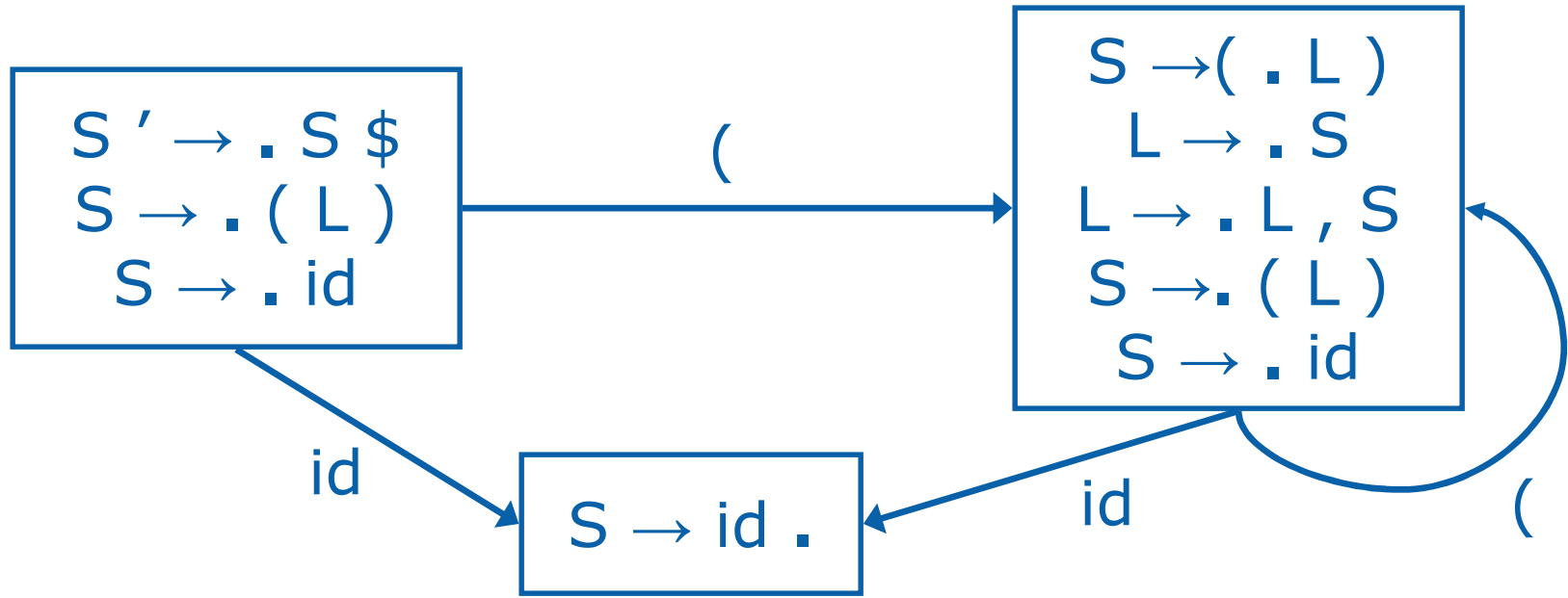


Constructing a DFA to read stack:

- First step: augment grammar with prod'n $S' \rightarrow S \$$
- Start state of DFA: empty stack = $S' \rightarrow \cdot S \$$
- *Closure* of a state adds items for all productions whose LHS occurs in an item in the state, just after "."
 - Set of possible productions to be reduced next
 - Added items have the "." located at the beginning: no symbols for these items on the stack yet

Applying Terminal Symbols

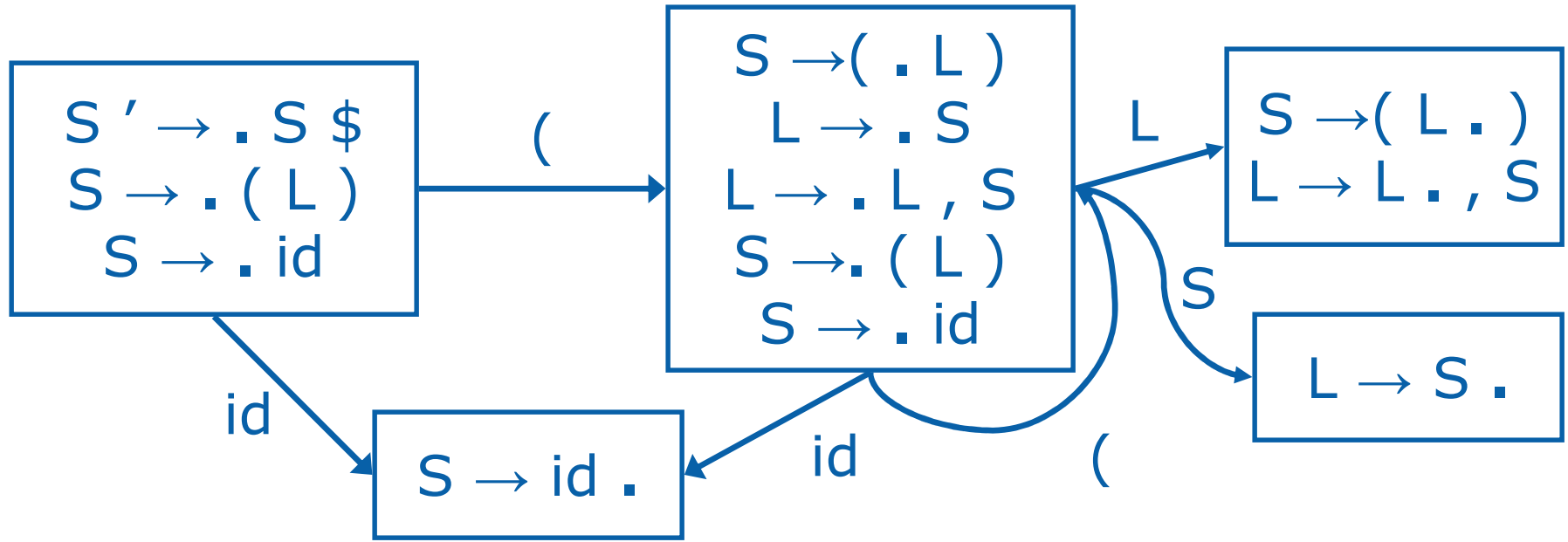
$S \rightarrow (L) \mid id$
 $L \rightarrow S \mid L , S$



In new state, include all items that have appropriate input symbol just after dot, advance dot in those items, and take closure.

Applying Nonterminal Symbols

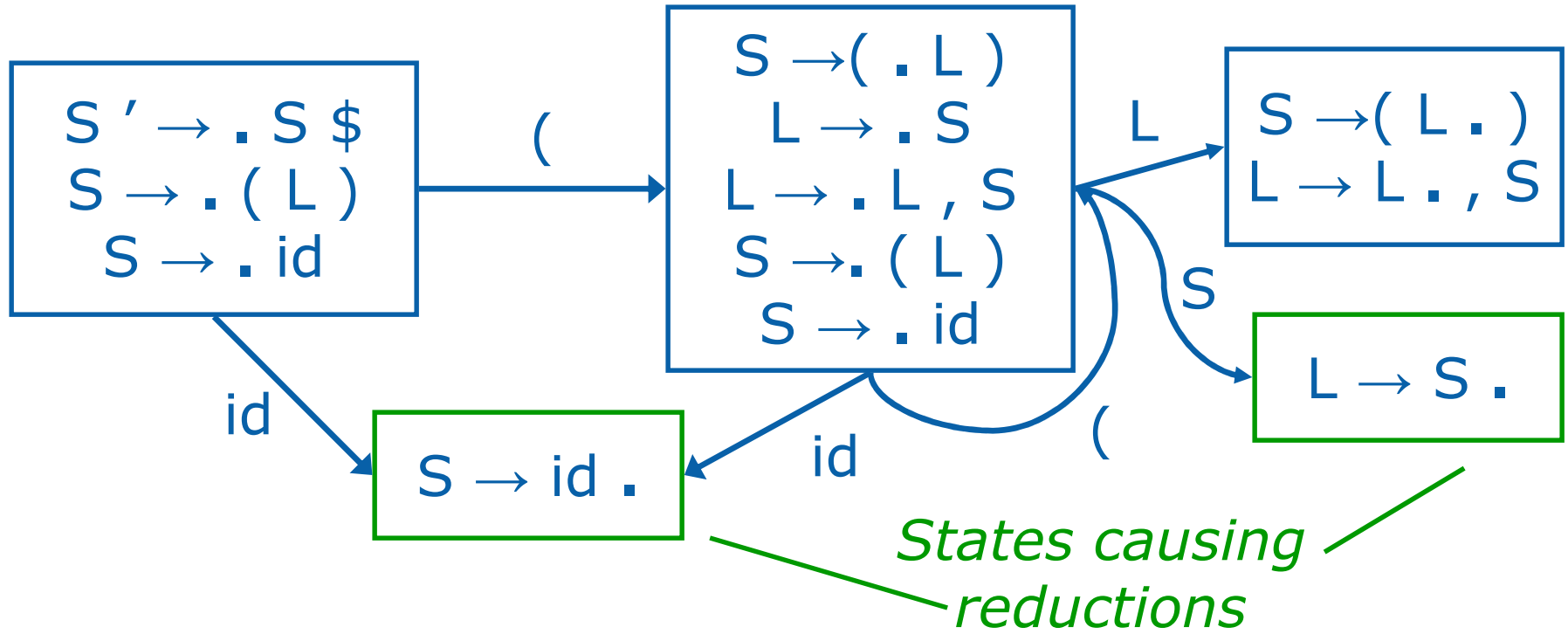
$S \rightarrow (L) \mid id$
 $L \rightarrow S \mid L , S$



- Non-terminals on stack treated just like terminals (except added by reductions)

Applying Reduce Actions

$S \rightarrow (L) \mid id$
 $L \rightarrow S \mid L , S$



- Pop RHS off stack, replace with LHS X ($X \rightarrow Y$)

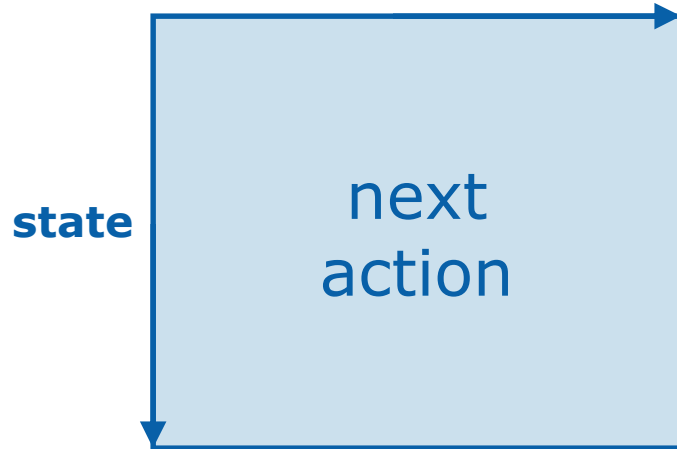
Parsing Example: ((x),y)

$S \rightarrow (L) \mid id$
 $L \rightarrow S \mid L , S$

<i>derivation</i>	<i>stack</i>	<i>input</i>	<i>action</i>
((x),y) ←	1	((x),y)	shift, goto 3
((x),y) ←	1 (3	(x),y)	shift, goto 3
((x),y) ←	1 (3 (3	x),y)	shift, goto 2
((x),y) ←	1 (3 (3 x2),y)	reduce S→id
((S),y) ←	1 (3 (3 S7),y)	reduce L→S
((L),y) ←	1 (3 (3 L5),y)	shift, goto 6
((L),y) ←	1 (3 (3 L5)6	,y)	reduce S→(L)
(S ,y) ←	1 (3 S7	,y)	reduce L→S
(L ,y) ←	1 (3 L5	,y)	shift, goto 8
(L,y) ←	1 (3 L5 , 8	y)	shift, goto 9
(L,y) ←	1 (3 L5 , 8 y2)	reduce S→id
(L, S) ←	1 (3 L5 , 8 S9)	reduce L→L,S
(L) ←	1 (3 L5)	shift, goto 6
(L) ←	1 (3 L5)6		reduce S→(L)
S	1 S4	\$	done

Implementation: LR Parsing Table

input (terminal) symbols

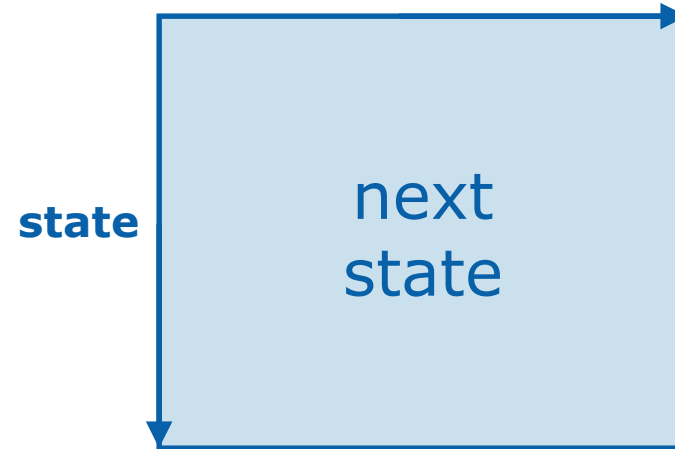


Action table

Used at every step to decide whether to shift or reduce



non-terminal symbols

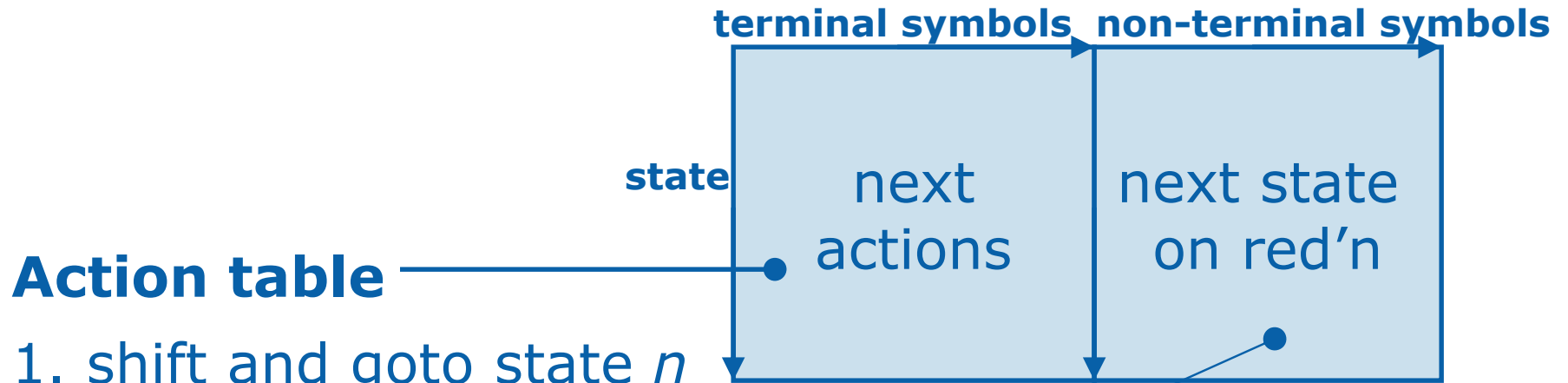


Goto table

Used only when reducing, to determine next state



Shift-Reduce Parsing Table



Action table

1. shift and goto state n
2. reduce using $X \rightarrow \gamma$
 - pop symbols γ off stack
 - using state label of top (end) of stack, look up X in goto table and goto that state

- DFA + stack = push-down automaton (PDA)

LR(0) Limitations

- An LR(0) machine only works if states with reduce actions have a *single* reduce action – in those states, *always* reduce ignoring lookahead
- With more complex grammar, construction gives states with shift/reduce or reduce/reduce conflicts
- Need to use look-ahead to choose

ok

$L \rightarrow L, S \cdot$

shift/reduce

$L \rightarrow L, S \cdot$
 $S \rightarrow S \cdot, L$

reduce/reduce

$L \rightarrow L, S \cdot$
 $L \rightarrow S \cdot$

LR(1) Parsing

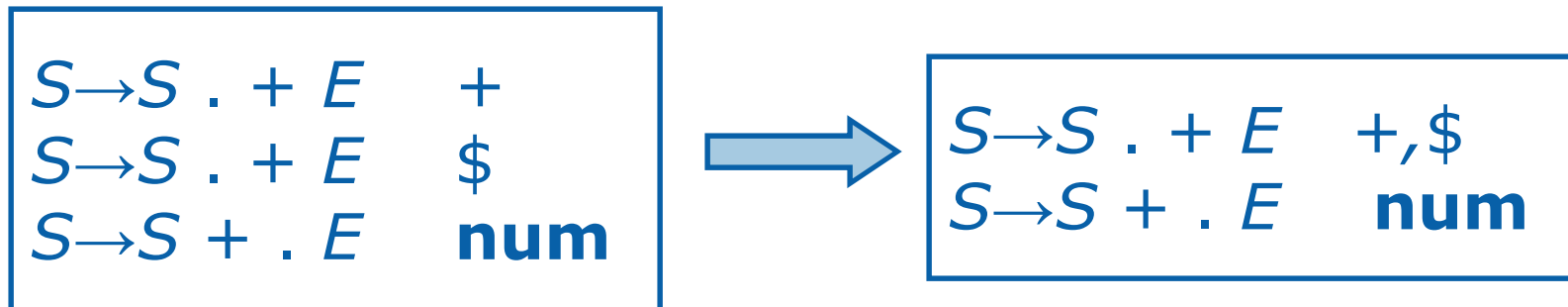
- As much power as possible out of 1 lookahead symbol parsing table
- LR(1) grammar = recognizable by a shift/reduce parser with 1 look-ahead.
- LR(1) item = LR(0) item + look-ahead symbols possibly following production

LR(0): $S \rightarrow \cdot S + E$

LR(1): $S \rightarrow \cdot S + E \quad +$

LR(1) State

- LR(1) state = set of LR(1) items
- LR(1) item = LR(0) item + set of lookahead symbols
- No two items in state have same production + dot configuration



LALR Grammars

- Problem with LR(1): too many states
- LALR(1) (Look-Ahead LR)
 - Merge any two LR(1) states whose items are identical except look-ahead
 - Results in smaller parser tables—works extremely well in practice
 - Usual technology for automatic parser generators

$$\begin{array}{|l} S \rightarrow id \cdot + \\ S \rightarrow E \cdot \$ \end{array} + \begin{array}{|l} S \rightarrow id \cdot \$ \\ S \rightarrow E \cdot + \end{array} = ?$$

The LALR(1) DFA

Algorithm:

repeat

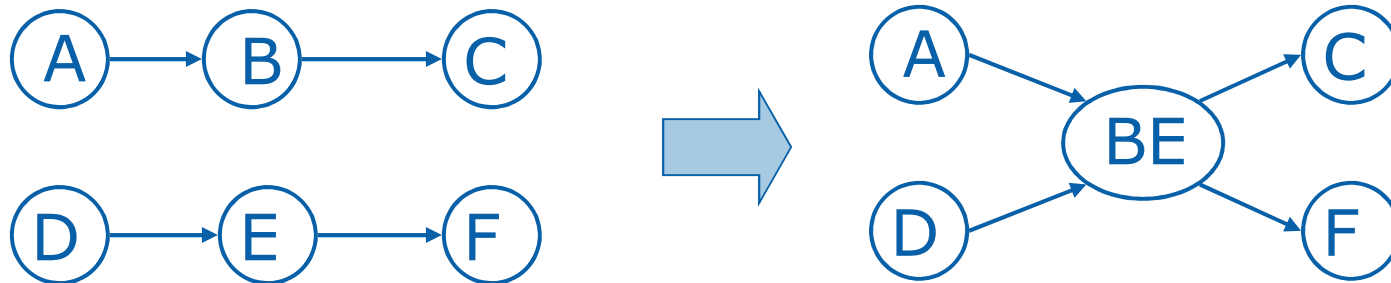
Choose two states with same core

Merge the states by combining the items

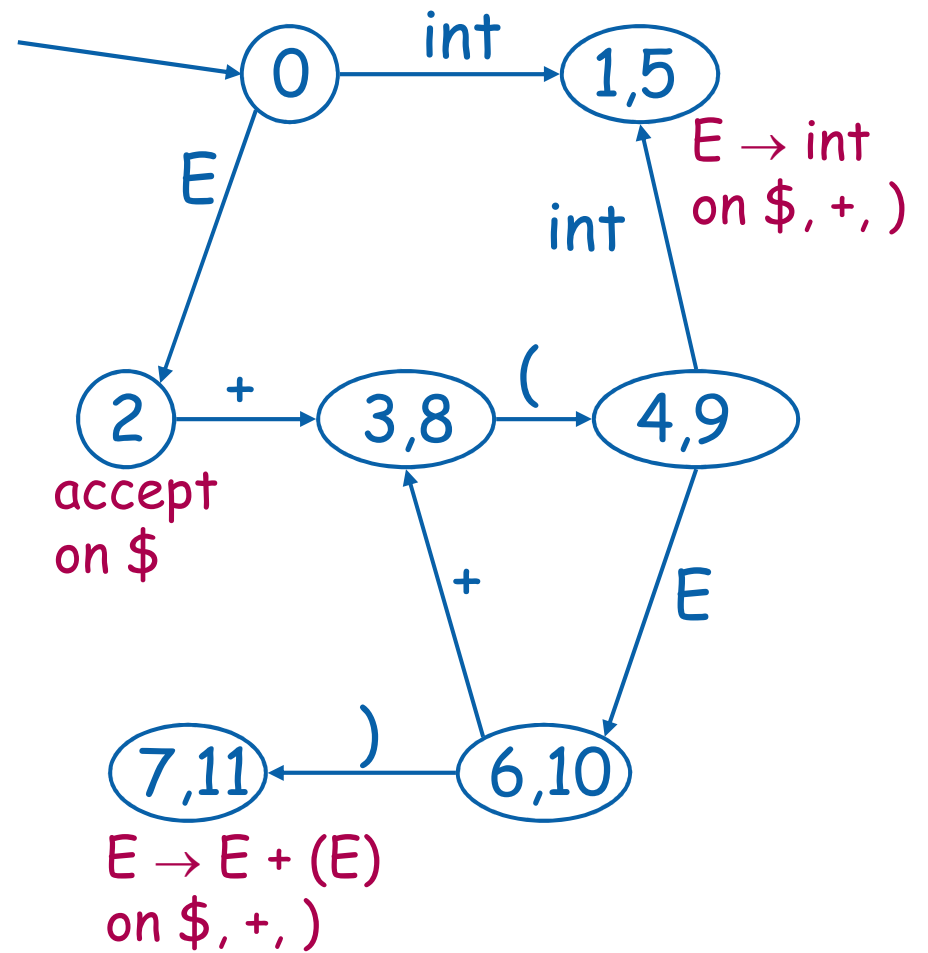
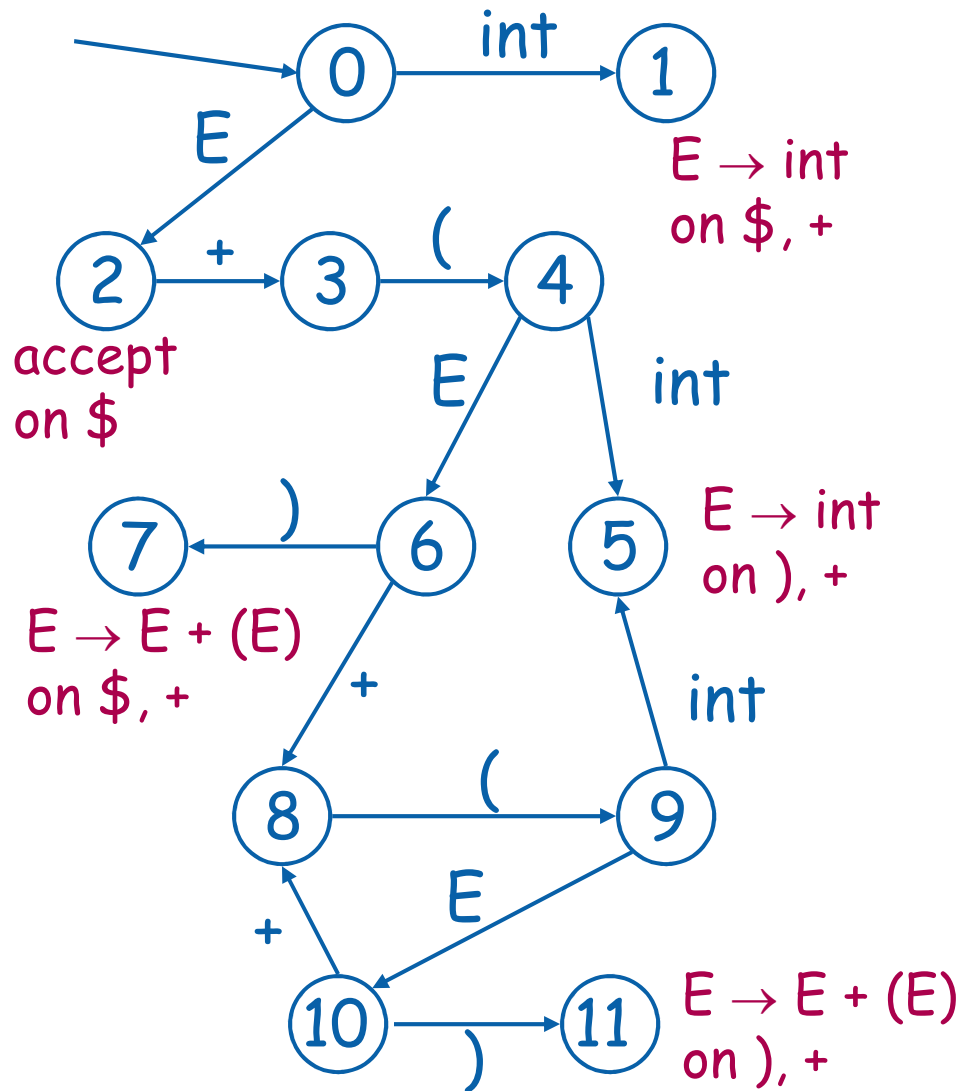
Point edges from predecessors to new state

New state points to all the previous successors

until all states have distinct core



Conversion LR(1) to LALR(1)



Notes on Parsing

Parsing

- A solid foundation: context-free grammars
- A simple parser: LL(1)
- A more powerful parser: LR(1)
- An efficiency hack: LALR(1)

- Issues with LR parsers
- LALR(1) parser generators

Issues with LR Parsers

What happens if a state contains:

$[X \rightarrow \alpha \bullet \underline{a}\beta, \underline{b}]$ and $[Y \rightarrow \gamma \bullet, \underline{a}]$

Then on input “a” we could either

- Shift into state $[X \rightarrow \alpha \underline{a} \bullet \beta, \underline{b}]$, or
- Reduce with $Y \rightarrow \gamma$

This is called a ***shift-reduce conflict***

- Typically due to ambiguity

Shift/Reduce Conflicts

Classic example: the dangling else

$S \rightarrow \text{if } E \text{ then } S \mid \text{if } E \text{ then } S \text{ else } S \mid \text{OTHER}$

Will have DFA state containing

$[S \rightarrow \underline{\text{if}} \ E \ \underline{\text{then}} \ S \ \bullet, \quad \underline{\text{else}}]$

$[S \rightarrow \underline{\text{if}} \ E \ \underline{\text{then}} \ S \ \bullet \ \underline{\text{else}} \ S, \quad \underline{x}]$

Practical solutions:

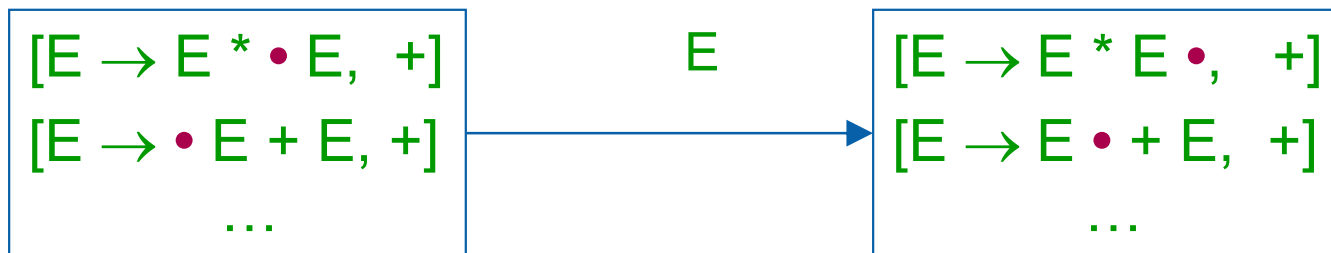
- Modify grammar to reflect the precedence of else
- Many LR parsers default to "shift"
- Often have a precedence declaration

Another Example

Consider the ambiguous grammar

$$E \rightarrow E + E \mid E * E \mid \text{int}$$

Part of the DFA:



We have a shift/reduce on input $+$

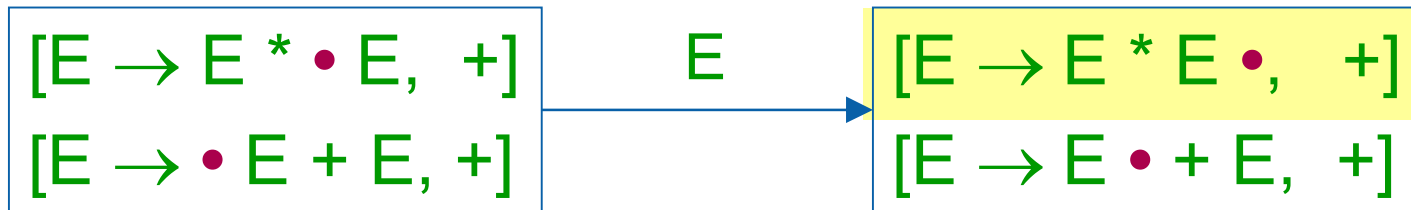
What do we want to happen?

- Consider: $x * y + z$
- We *need* to reduce ($*$ binds more tightly than $+$)
- Default action is shift

Precedence

Declare relative precedence

- Explicitly resolve conflict
- Tell parser: we prefer the action involving * over +



In practice:

- Parser generators support a precedence declaration for operators

Other Problems

If a DFA state contains both

$$[X \rightarrow \alpha \bullet, \underline{a}] \text{ and } [Y \rightarrow \beta \bullet, \underline{a}]$$

- What's the problem here?
- Two reductions to choose from when next token is **a**

This is called a *reduce/reduce* conflict

- Usually a serious ambiguity in the grammar
- Must be fixed in order to generate parser

Reduce/Reduce Conflicts

Example: a sequence of identifiers

$S \rightarrow \varepsilon \mid id \mid id S$

There are two parse trees for the string **id**

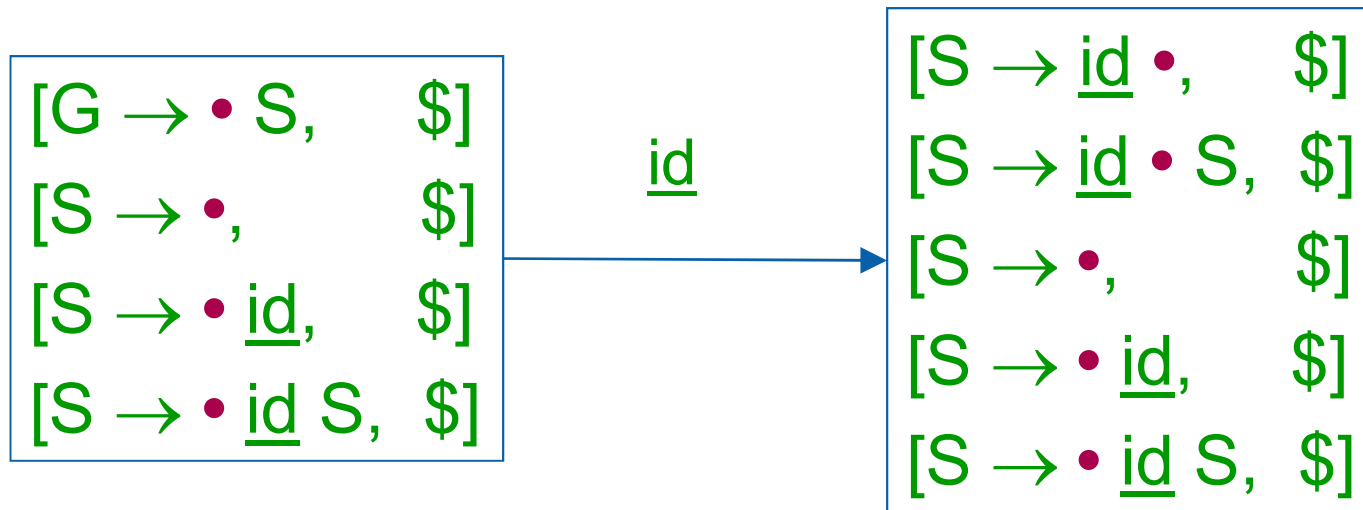
$S \rightarrow id$

$S \rightarrow id S \rightarrow id$

How does this confuse the parser?

Reduce/Reduce Conflicts

Consider the DFA states:



Reduce/reduce conflict on input \$

$G \rightarrow S \rightarrow id$

$G \rightarrow S \rightarrow id S \rightarrow id$

Instead rewrite the grammar: $S \rightarrow \epsilon \mid id S$

LR Parsing

