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:

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

  - 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 w 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 w = $\alpha\ A\ \gamma$)

until str = S (the start symbol)
OR all possibilities are exhausted

A Bottom-up Parse in Detail (1)

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

int + (int) + (int)

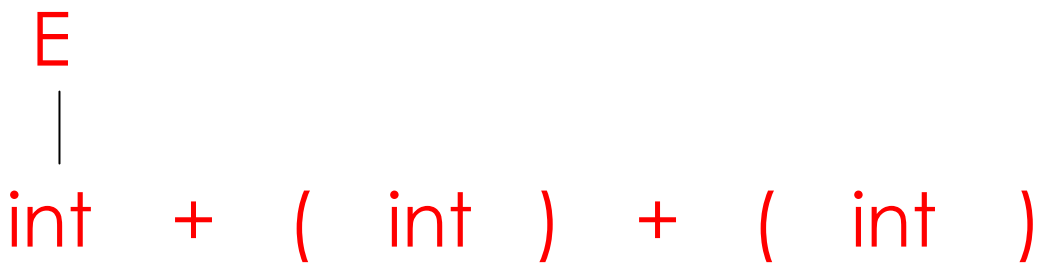
int + (int) + (int)

A Bottom-up Parse in Detail (2)

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

int + (int) + (int)

E + (int) + (int)



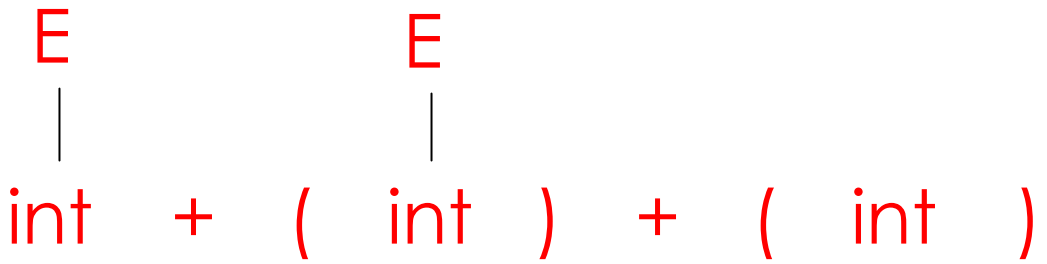
A Bottom-up Parse in Detail (3)

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

int + (int) + (int)

E + (int) + (int)

E + (E) + (int)



A Bottom-up Parse in Detail (4)

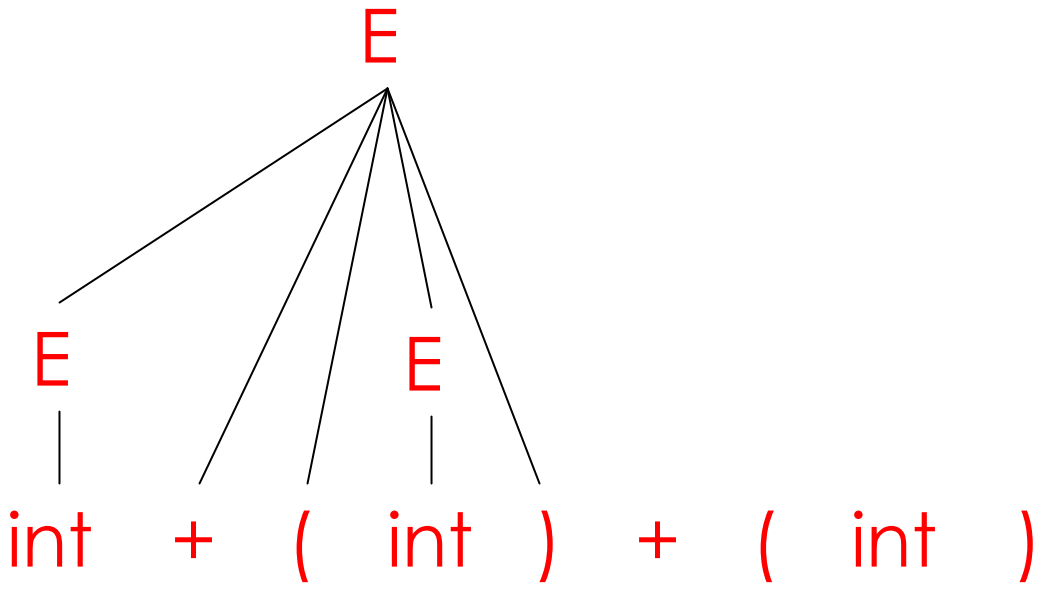
$E \rightarrow E + (E) \mid int$

int + (int) + (int)

E + (int) + (int)

E + (E) + (int)

E + (int)



A Bottom-up Parse in Detail (5)

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

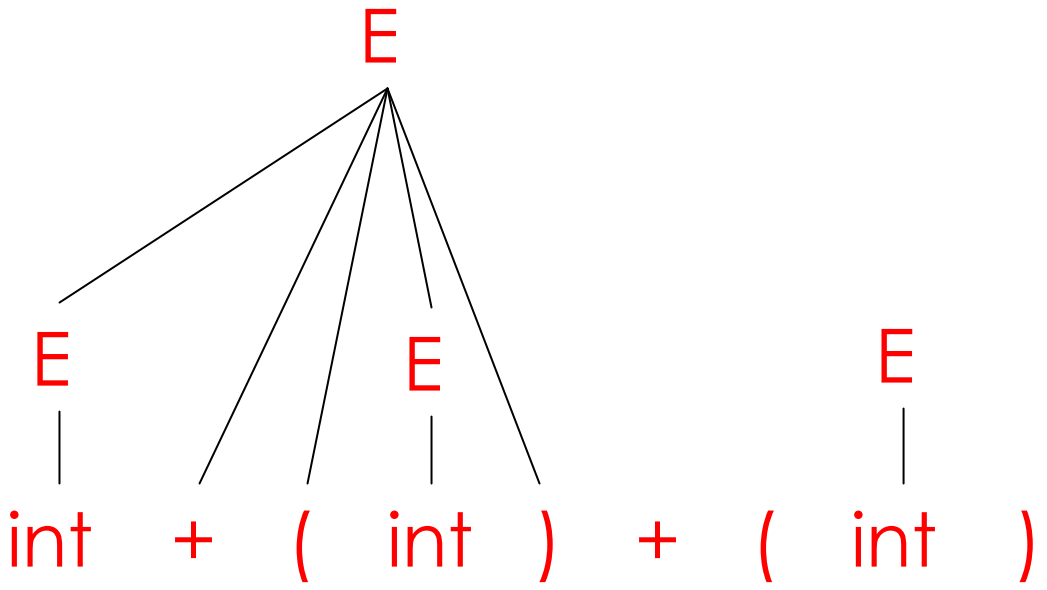
int + (int) + (int)

E + (int) + (int)

E + (E) + (int)

E + (int)

E + (E)

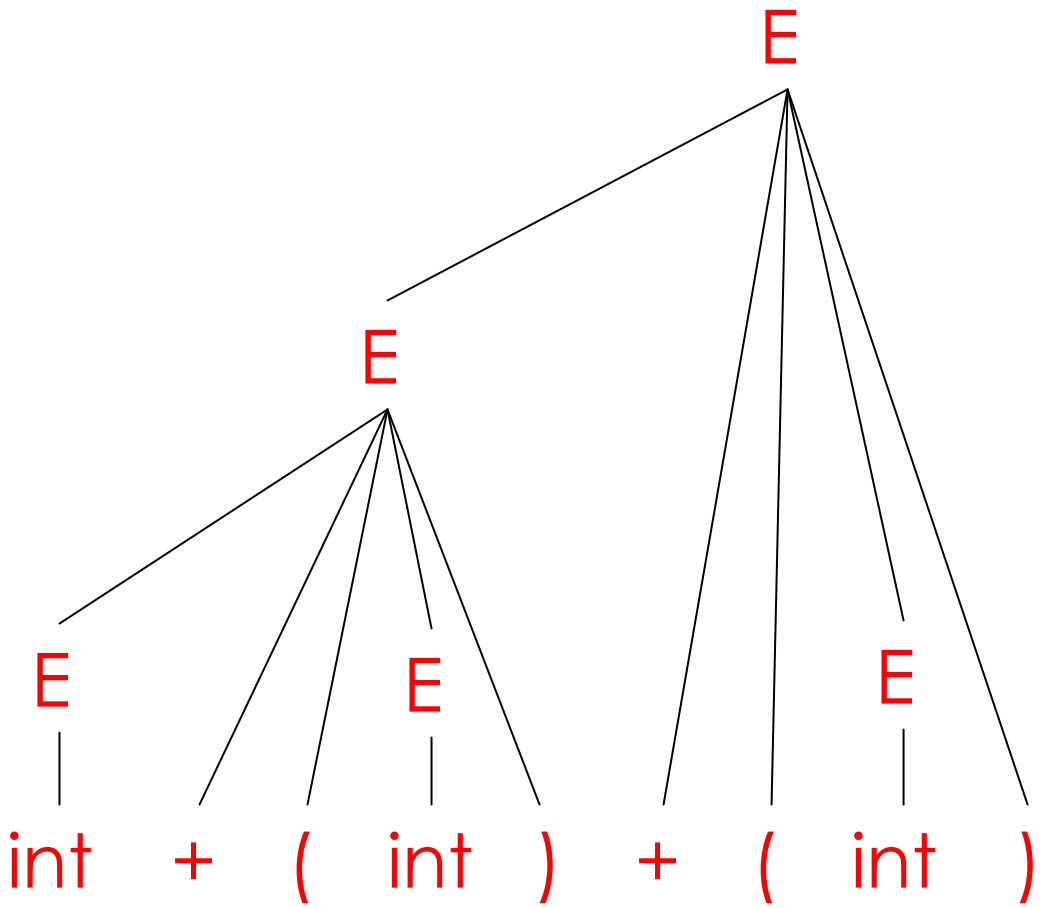


A Bottom-up Parse in Detail (6)

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

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 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 is as yet unexamined by parsing (a string of terminals)
  - Left substring has terminals and non-terminals

- The dividing point is marked by a I
  - The I is not part of the string

- Initially, all input is unexamined: $\mathrm{I}x_1x_2 \ldots x_n$

Shift-Reduce Parsing

Bottom-up parsing uses only two kinds of actions:

Shift

Reduce

Shift

Shift: Move I one place to the right

- Shifts a terminal to the left string

$$E + (\mathrm{I}\ \mathrm{int}\ ) \Rightarrow E + (\mathrm{int}\ \mathrm{I}\ )$$

In general:

$$ABC\ \mathrm{I}\ xyz \Rightarrow ABCx\ \mathrm{I}\ yz$$

Reduce

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

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

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

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

$$Cbxy \mid ijk \Rightarrow CbA \mid ijk$$

Shift-Reduce Example

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

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

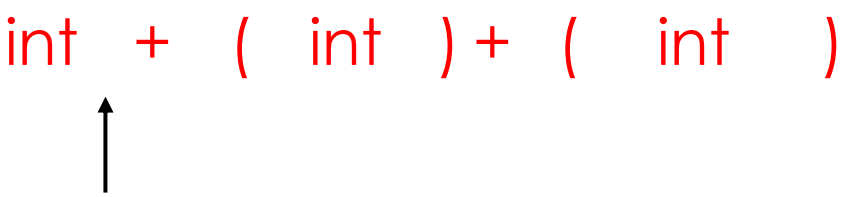
int + (int) + (int)

Shift-Reduce Example

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

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

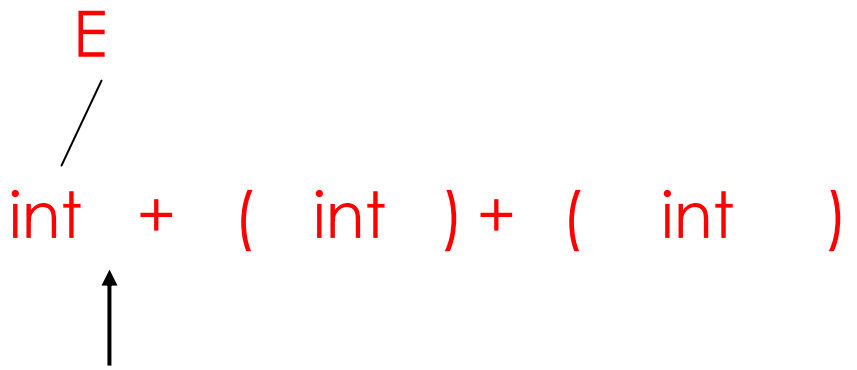
int | + (int) + (int)$ reduce $E \rightarrow int$



Shift-Reduce Example

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

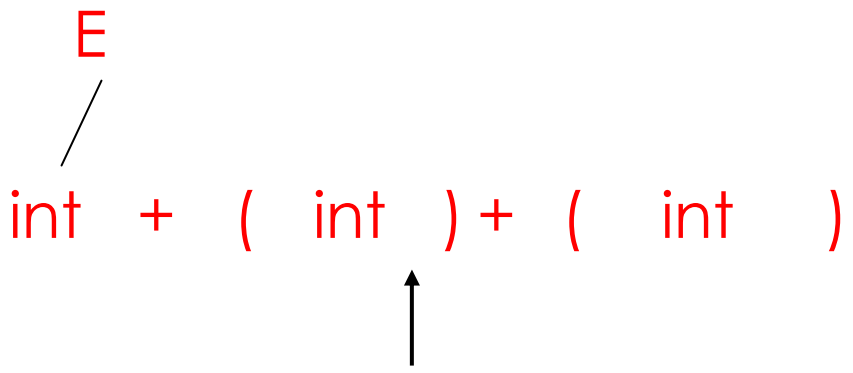
I int + (int) + (int)$	shift
int I + (int) + (int)$	reduce $E \rightarrow int$
E I + (int) + (int)$	shift 3 times



Shift-Reduce Example

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

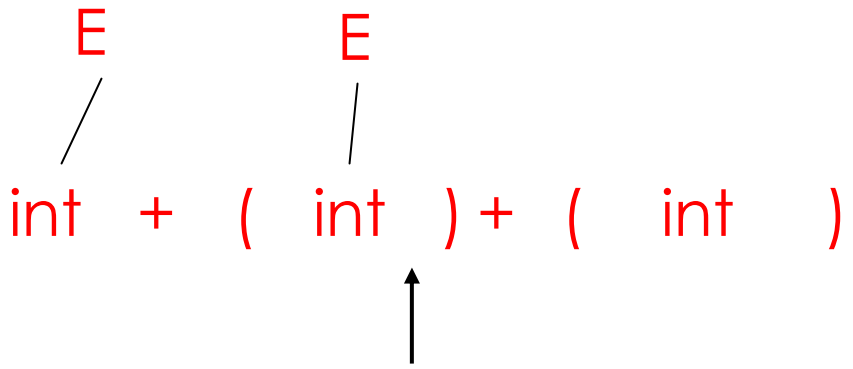
I int + (int) + (int)$	shift
int I + (int) + (int)$	reduce $E \rightarrow int$
E I + (int) + (int)$	shift 3 times
E + (int I) + (int)$	reduce $E \rightarrow int$



Shift-Reduce Example

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

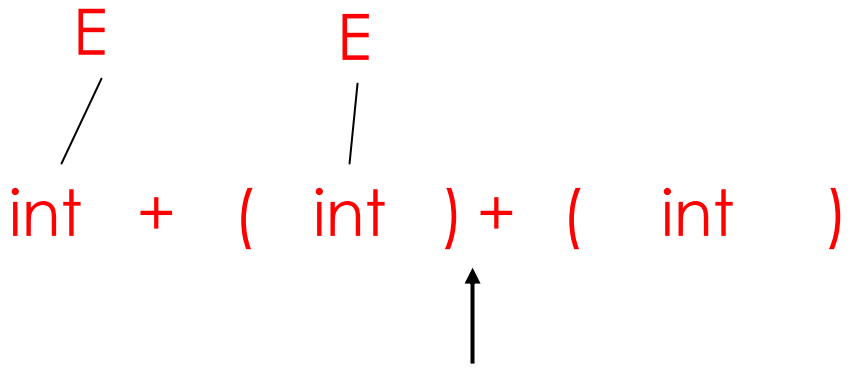
I int + (int) + (int)$	shift
int I + (int) + (int)$	reduce $E \rightarrow int$
E I + (int) + (int)$	shift 3 times
E + (int I) + (int)$	reduce $E \rightarrow int$
E + (E I) + (int)$	shift



Shift-Reduce Example

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

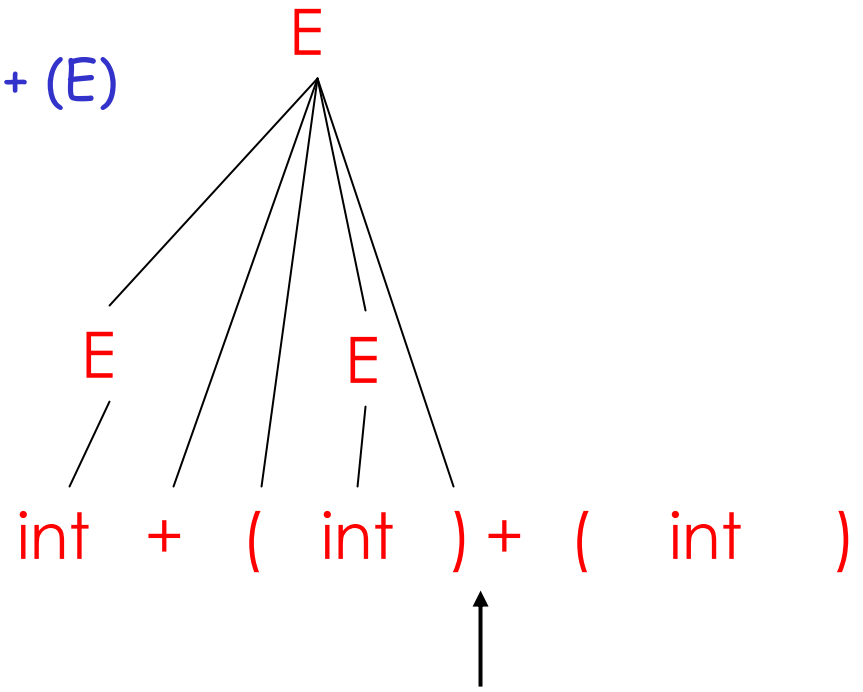
I int + (int) + (int)$	shift
int I + (int) + (int)$	reduce $E \rightarrow int$
E I + (int) + (int)$	shift 3 times
E + (int I) + (int)$	reduce $E \rightarrow int$
E + (E I) + (int)$	shift
E + (E) I + (int)$	reduce $E \rightarrow E + (E)$



Shift-Reduce Example

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

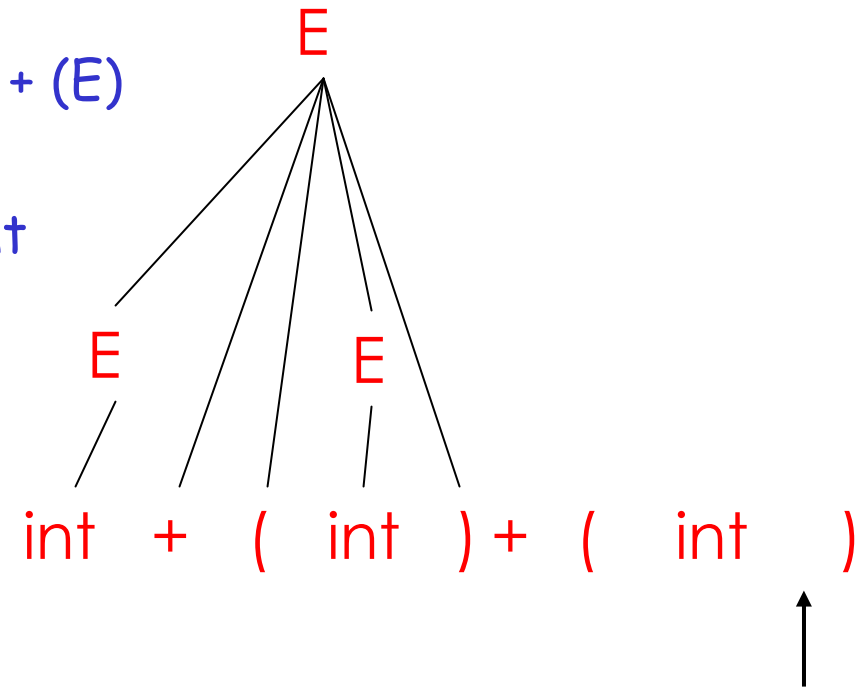
I int + (int) + (int)$	shift
int I + (int) + (int)$	reduce $E \rightarrow int$
E I + (int) + (int)$	shift 3 times
E + (int I) + (int)$	reduce $E \rightarrow int$
E + (E I) + (int)$	shift
E + (E) I + (int)$	reduce $E \rightarrow E + (E)$
E I + (int)$	shift 3 times



Shift-Reduce Example

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

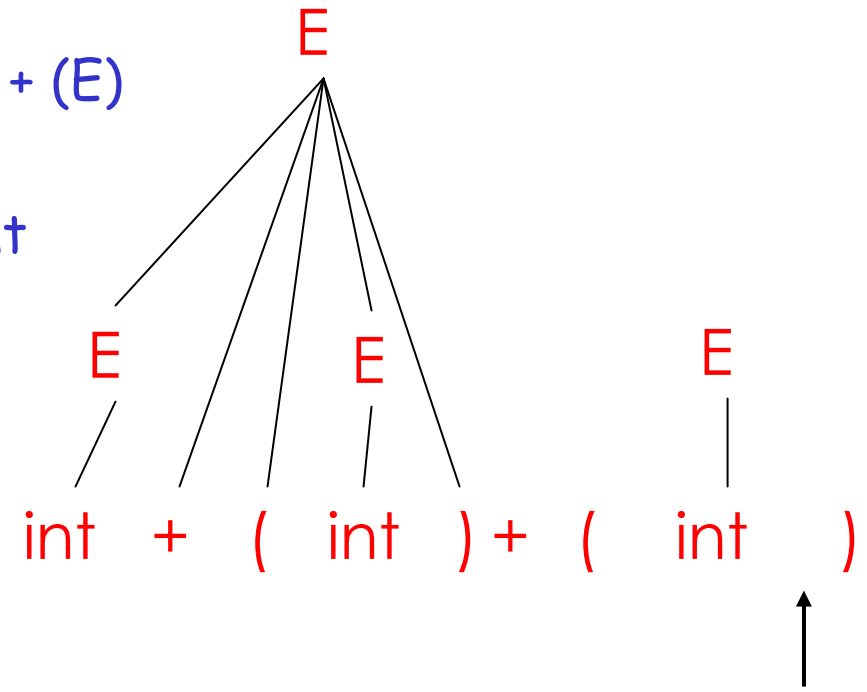
I int + (int) + (int)$	shift
int I + (int) + (int)$	reduce $E \rightarrow int$
E I + (int) + (int)$	shift 3 times
E + (int I) + (int)$	reduce $E \rightarrow int$
E + (E I) + (int)$	shift
E + (E) I + (int)$	reduce $E \rightarrow E + (E)$
E I + (int)$	shift 3 times
E + (int I)$	reduce $E \rightarrow int$



Shift-Reduce Example

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

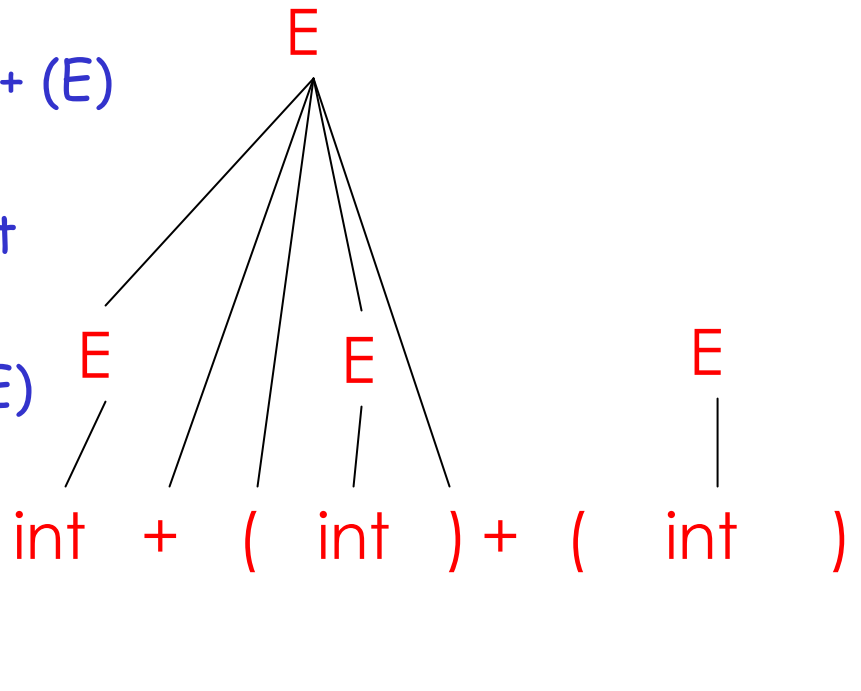
I int + (int) + (int)$	shift
int I + (int) + (int)$	reduce $E \rightarrow int$
E I + (int) + (int)$	shift 3 times
E + (int I) + (int)$	reduce $E \rightarrow int$
E + (E I) + (int)$	shift
E + (E) I + (int)$	reduce $E \rightarrow E + (E)$
E I + (int)$	shift 3 times
E + (int I)$	reduce $E \rightarrow int$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



Shift-Reduce Example

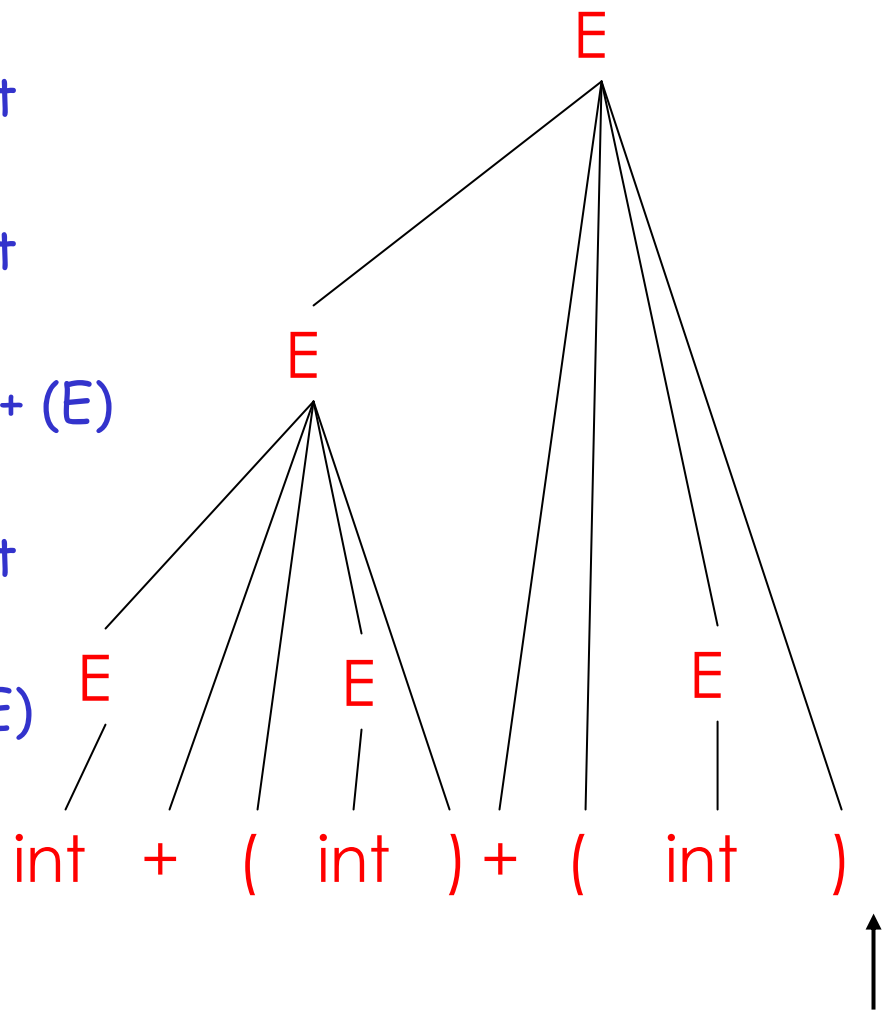
$E \rightarrow E + (E) \mid int$

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



Shift-Reduce Example

I int + (int) + (int)$	shift
int I + (int) + (int)$	reduce E → int
E I + (int) + (int)$	shift 3 times
E + (int I) + (int)$	reduce E → int
E + (E I) + (int)$	shift
E + (E) I + (int)$	reduce E → E + (E)
E I + (int)$	shift 3 times
E + (int I)$	reduce E → int
E + (E I)$	shift
E + (E) I $	reduce E → E + (E)
E I $	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)

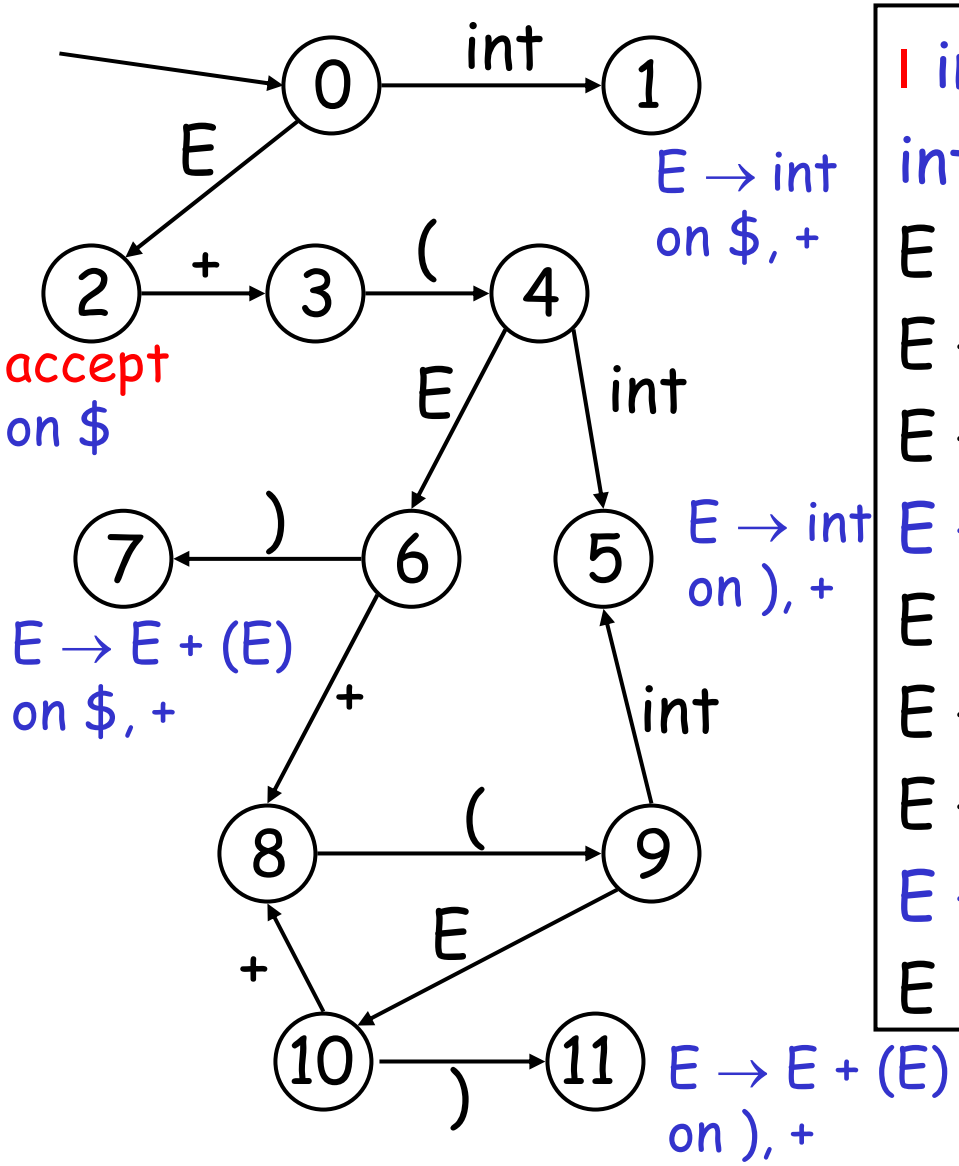
Key Question: To Shift or to Reduce?

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



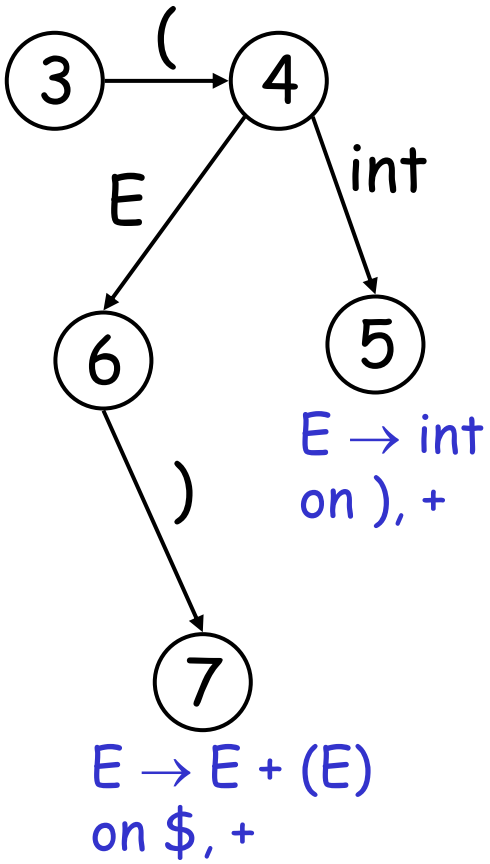
▮ int + (int) + (int)$	shift
int ▮ + (int) + (int)$	E → int
E ▮ + (int) + (int)$	shift(x3)
E + (int ▮) + (int)$	E → int
E + (E ▮) + (int)$	shift
E + (E) ▮ + (int)$	E → E+(E)
E ▮ + (int)$	shift (x3)
E + (int ▮)$	E → int
E + (E ▮)$	shift
E + (E) ▮ $	E → E+(E)
E ▮ $	accept

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
    - 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		$r_{E \to int}$		$r_{E \to int}$		
6		s8		s7		
7		$r_{E \to E+(E)}$			$r_{E \to E+(E)}$	
...						

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

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

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 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 DFA constructed?

LR Parsing
LALR Parser Generators

Key Issue: How is the DFA Constructed?

- The stack describes the context of the parse
  - What non-terminal we are looking for
  - What production RHS we are looking for
  - What we have seen so far from the RHS

- Each DFA state describes several such contexts
  - E.g., when we are looking for non-terminal E, we might be looking either for an int or an E + (E) RHS

LR(0) Items

- An LR(0) item is a production with a "•" somewhere on the RHS

- The LR(0) items for $T \rightarrow (E)$ are
  - $T \rightarrow \bullet\ (E)$
  - $T \rightarrow (\ \bullet E)$
  - $T \rightarrow (E\ \bullet\ )$
  - $T \rightarrow (E)\ \bullet$

- The only LR(0) item for $X \rightarrow \varepsilon$ is $X \rightarrow \bullet$

LR(0) Items: Intuition

- An item $[X \rightarrow \alpha \mid \beta]$ says that the parser
  - is looking for an X
  - has an α on top of the stack
  - expects to find a string derived from β next in the input

- Notes:
  - $[X \rightarrow \alpha \mid a\beta]$ means that a should follow
    - Then we can shift it and still have a viable prefix
  - $[X \rightarrow \alpha \mid]$ means that we could reduce X
    - But this is not always a good idea !

LR(1) Items

- An LR(1) item is a pair:

$$X \rightarrow \alpha \mid \beta, \ a$$

  - $X \rightarrow \alpha\beta$ is a production
  - a is a terminal (the lookahead terminal)
  - LR(1) means 1 lookahead terminal
- $[X \rightarrow \alpha \mid \beta, a]$ describes a context of the parser
  - We are trying to find an X followed by an a, and
  - We have (at least) α already on top of the stack
  - Thus we need to see next a prefix derived from βa

Note

- The symbol I was used before to separate the stack from the rest of input
  - α I γ, where α is the stack and γ is the remaining string of terminals
- In items, I is used to mark a prefix of a production RHS:

$$X \rightarrow \alpha \text{ I } \beta, \quad a$$

  - Here β might contain non-terminals as well
- In either case the stack is on the left of I

Convention

- We add to our grammar a fresh new start symbol S and a production $S \rightarrow E$
  - Where E is the old start symbol

- The initial parsing context contains:

$$S \rightarrow \text{I}\, E \;,\, \$$$

  - Trying to find an S as a string derived from $E\$$
  - The stack is empty

LR(1) Items (Cont.)

- In context containing

$$E \rightarrow E + \cdot (E) \quad , +$$

  - If (follows then we can perform a shift to context containing

$$E \rightarrow E + (\cdot E) \quad , +$$

- In context containing

$$E \rightarrow E + (E) \cdot \quad , +$$

  - We can perform a reduction with $E \rightarrow E + (E)$
  - But only if a + follows

LR(1) Items (Cont.)

- Consider the item

$$E \rightarrow E + (\ \cdot\ E\ ),\ +$$

- We expect a string derived from $E\ ) +$
- Our example has two productions for E

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

- We describe this by extending the context with two more items:

$$E \rightarrow \cdot\ \text{int} \qquad ,\ )$$

$$E \rightarrow \cdot\ E + (\ E\ ),\ )$$

The Closure Operation

- The operation of extending the context with items is called the closure operation

Closure(Items) =
  repeat
    for each [$X \to \alpha \bullet Y\beta$, a] in Items
      for each production $Y \to \gamma$
        for each b in First(βa)
          add [$Y \to \bullet \gamma$, b] to Items
  until Items is unchanged

Constructing the Parsing DFA (1)

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

- Construct the start context:

  Closure({$S \rightarrow \cdot E, \$$})

$$
\begin{array}{lll}
S \rightarrow \cdot E & , & \$ \\
E \rightarrow \cdot E+(E) & , & \$ \\
E \rightarrow \cdot int & , & \$ \\
E \rightarrow \cdot E+(E) & , & + \\
E \rightarrow \cdot int & , & +
\end{array}
$$

- We abbreviate as:

$$
\begin{array}{lll}
S \rightarrow \cdot E & , & \$ \\
E \rightarrow \cdot E+(E) & , & \$/+ \\
E \rightarrow \cdot int & , & \$/+
\end{array}
$$

Constructing the Parsing DFA (2)

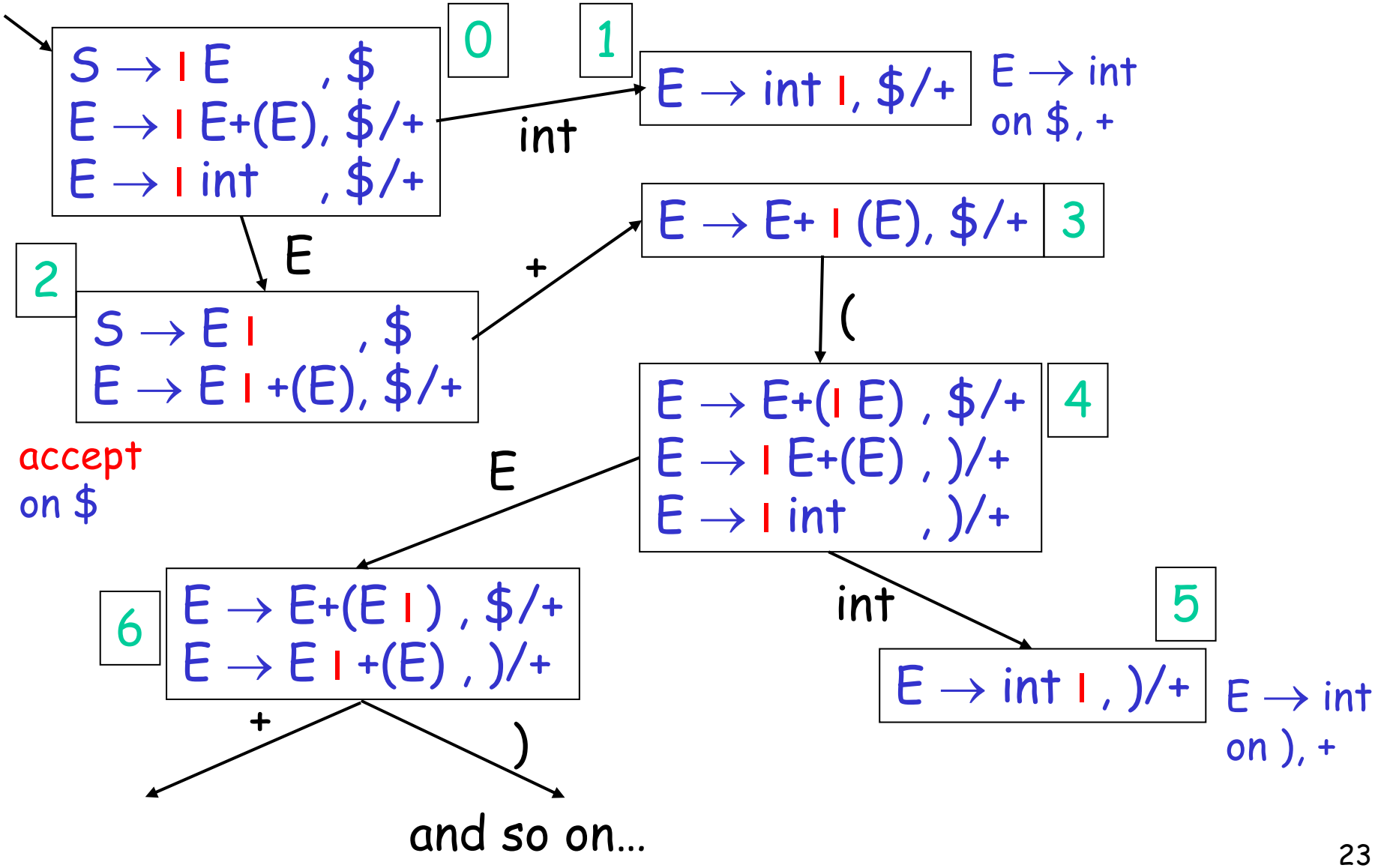
- A DFA state is a closed set of LR(1) items
- The start state contains [S $\rightarrow$ ı E , $]
- A state that contains [X $\rightarrow$ α ı, b] is labeled with "reduce with X $\rightarrow$ α on b"
- And now the transitions …

The DFA Transitions

- A state "State" that contains $[X \rightarrow \alpha \mid y\beta, b]$ has a transition labeled y to a state that contains the items "**Transition**(State, y)"
  - y can be a terminal or a non-terminal

```
Transition(State, y)
  Items = ∅
  for each [X → α | yβ, b] in State
      add [X → αy | β, b] to Items
  return Closure(Items)
```

Constructing the Parsing DFA: Example



LR Parsing Tables: Notes

- Parsing tables (i.e., the DFA) can be constructed automatically for a CFG
- But we still need to understand the construction to work with parser generators
  - E.g., they report errors in terms of sets of items
- What kind of errors can we expect?

Shift/Reduce Conflicts

- If a DFA state contains both

  $[X \to \alpha \mid a\beta, b]$ and $[Y \to \gamma \mid, a]$

- Then on input "a" we could either
  - Shift into state $[X \to \alpha a \mid \beta, b]$, or
  - Reduce with $Y \to \gamma$

- This is called a *shift-reduce conflict*

Shift/Reduce Conflicts

- Typically due to ambiguities in the grammar
- Classic example: the dangling else

  $S \rightarrow$ if E then S | if E then S else S | OTHER
- Will have DFA state containing

  [$S \rightarrow$ if E then S I, else]

  [$S \rightarrow$ if E then S I else S, x]
- If else follows then we can shift or reduce
- Default (yacc, ML-yacc, bison, etc.) is to shift
  - Default behavior is as needed in this case

More Shift/Reduce Conflicts

- Consider the ambiguous grammar

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

- We will have the states containing

$$[E \rightarrow E * \mid E,\ +] \qquad [E \rightarrow E * E \mid,\ +]$$
$$[E \rightarrow \mid E + E,\ +] \quad \Rightarrow^{E} \quad [E \rightarrow E \mid + E,\ +]$$
$$\ldots \qquad \ldots$$

- Again we have a shift/reduce on input +
  - We need to reduce (* binds more tightly than +)
  - Recall solution: declare the precedence of * and +

More Shift/Reduce Conflicts

- In yacc declare precedence and associativity:

```
%left +
%left *
```

- Precedence of a rule = that of its last terminal

  See yacc manual for ways to override this default

- Resolve shift/reduce conflict with a shift if:
  - no precedence declared for either rule or terminal
  - input terminal has higher precedence than the rule
  - the precedences are the same and right associative

Using Precedence to Solve S/R Conflicts

- Back to our example:

$$[E \to E * \mid E,\ +] \qquad [E \to E * E \mid,\ +]$$
$$[E \to \mid E + E,\ +] \Rightarrow^{E} \quad [E \to E \mid + E,\ +]$$
$$\ldots \qquad \ldots$$

- Will choose reduce because precedence of rule $E \to E * E$ is higher than of terminal $+$

Using Precedence to Solve S/R Conflicts

- Same grammar as before

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

- We will also have the states

$$[E \to E + \cdot E,\ +] \qquad [E \to E + E \cdot,\ +]$$
$$[E \to \cdot E + E,\ +] \quad \Rightarrow^{E} \quad [E \to E \cdot + E,\ +]$$
$$\ldots \qquad\qquad \ldots$$

- Now we also have a shift/reduce on input +
  - We choose reduce because $E \to E + E$ and + have the same precedence and + is left-associative

Using Precedence to Solve S/R Conflicts

- Back to our dangling else example

  [S → if E then S I, else]

  [S → if E then S I else S, x]

- Can eliminate conflict by declaring else having higher precedence than then
- But this starts to look like "hacking the tables"
- Best to avoid overuse of precedence declarations or we will end with unexpected parse trees

Precedence Declarations Revisited

The term "precedence declaration" is misleading!

These declarations do not define precedence: they define conflict resolutions

- I.e., they instruct shift-reduce parsers to resolve conflicts in certain ways
- These two are not quite the same!

Reduce/Reduce Conflicts

- If a DFA state contains both

  $[X \to \alpha \,|\, , a]$ and $[Y \to \beta \,|\, , a]$

  - Then on input "a" we don't know which production to reduce

- This is called a *reduce/reduce conflict*

Reduce/Reduce Conflicts

- Usually due to gross ambiguity in the grammar
- 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?

More on Reduce/Reduce Conflicts

- Consider the states

$$
\begin{array}{ll}
[S' \to \cdot\, S, & \$] \\
[S \to \cdot, & \$] \\
[S \to \cdot\, \mathrm{id}, & \$] \\
[S \to \cdot\, \mathrm{id}\ S, & \$]
\end{array}
\quad \Rightarrow^{\mathrm{id}} \quad
\begin{array}{ll}
[S \to \mathrm{id}\ \cdot, & \$] \\
[S \to \mathrm{id}\ \cdot\, S, & \$] \\
[S \to \cdot, & \$] \\
[S \to \cdot\, \mathrm{id}, & \$] \\
[S \to \cdot\, \mathrm{id}\ S, & \$]
\end{array}
$$

- Reduce/reduce conflict on input \$

$$S' \to S \to \mathrm{id}$$

$$S' \to S \to \mathrm{id}\ S \to \mathrm{id}$$

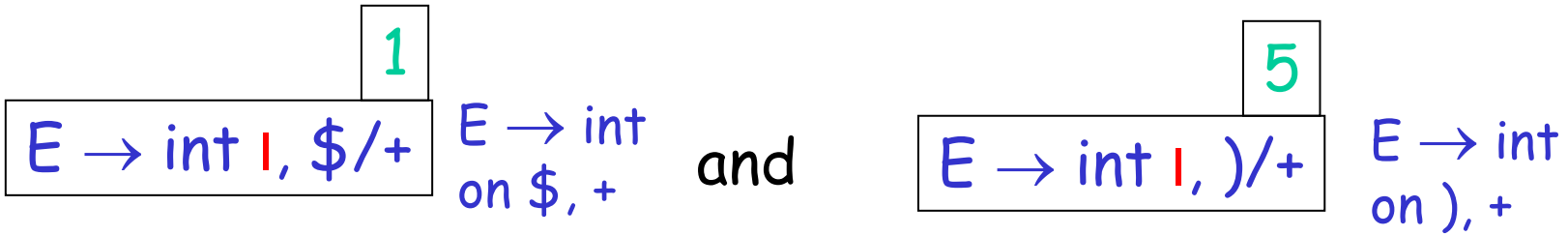
- Better to rewrite the grammar as: $S \to \varepsilon \mid \mathrm{id}\ S$

Using Parser Generators

- Parser generators automatically construct the parsing DFA given a CFG
  - Use precedence declarations and default conventions to resolve conflicts
  - The parser algorithm is the same for all grammars (and is provided as a library function)
- But most parser generators do not construct the DFA as described before
  - Because the LR(1) parsing DFA has 1000s of states even for a simple language

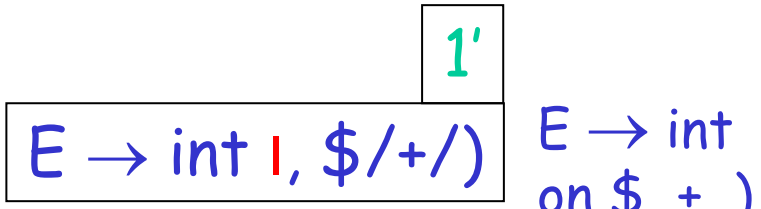
LR(1) Parsing Tables are Big

- But many states are similar, e.g.



- Idea: merge the DFA states whose items differ only in the lookahead tokens
  - We say that such states have the same core

- We obtain



The Core of a Set of LR Items

Definition: The core of a set of LR items is the set of first components

- Without the lookahead terminals

- Example: the core of

$$\{[X \to \alpha \mid \beta, b], [Y \to \gamma \mid \delta, d]\}$$

is

$$\{X \to \alpha \mid \beta,\ \ Y \to \gamma \mid \delta\}$$

LALR States

- Consider for example the LR(1) states

  $\{[X \to \alpha\ I, a], [Y \to \beta\ I, c]\}$

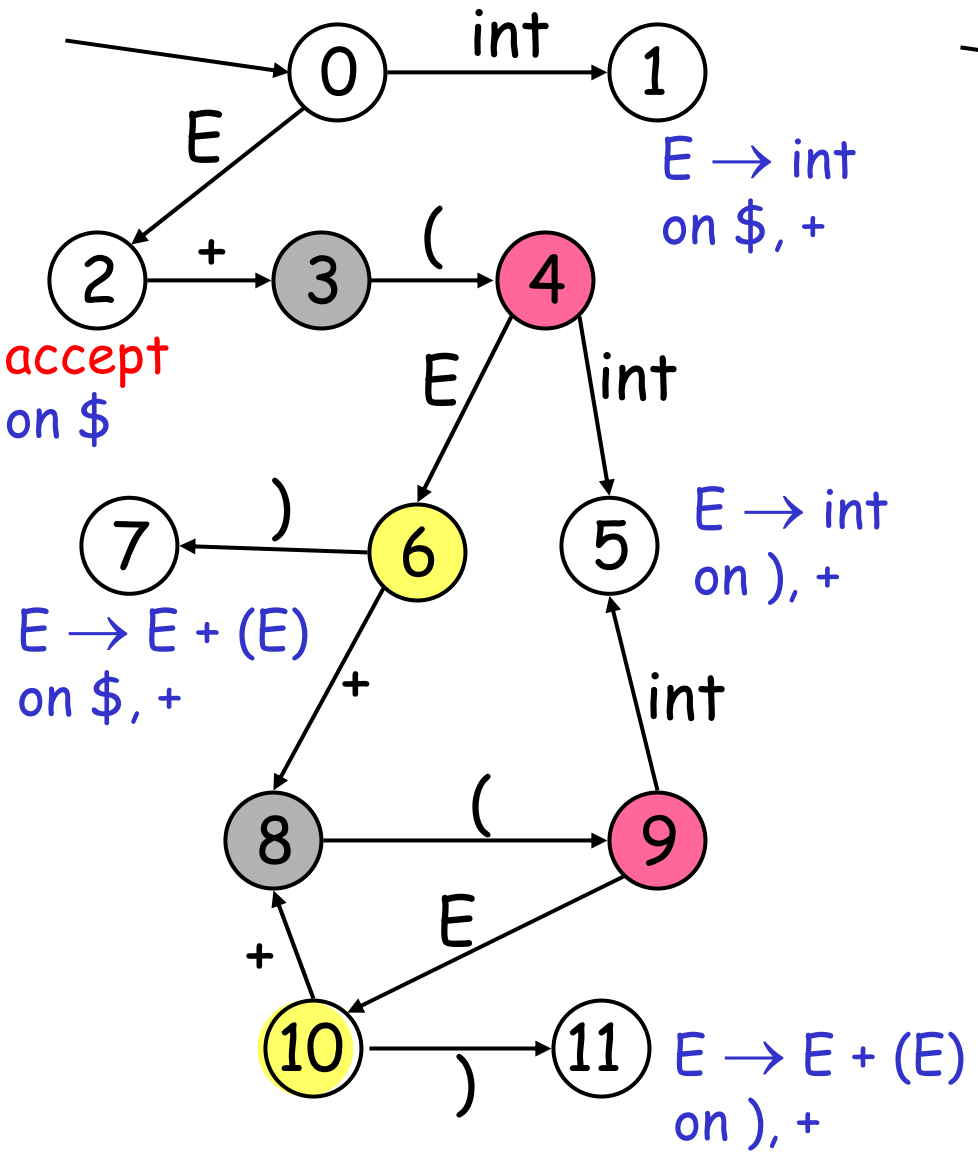
  $\{[X \to \alpha\ I, b], [Y \to \beta\ I, d]\}$
- They have the same core and can be merged
- The merged state contains:

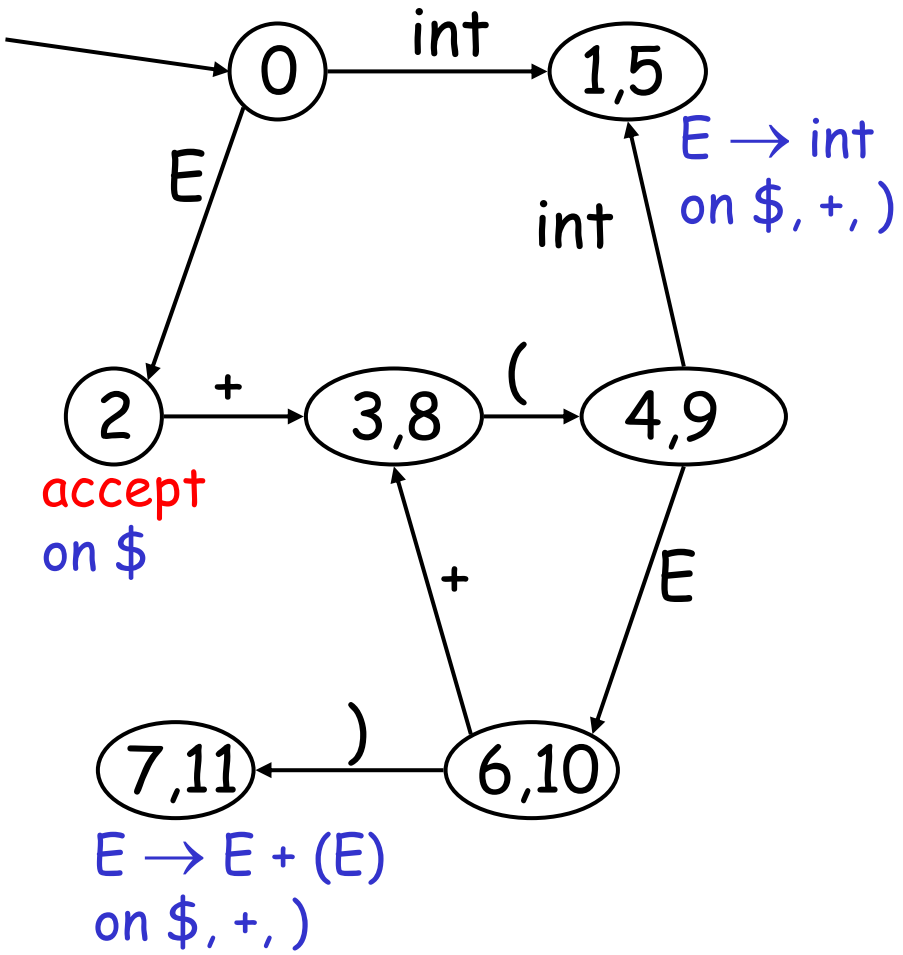
  $\{[X \to \alpha\ I, a/b], [Y \to \beta\ I, c/d]\}$
- These are called LALR(1) states
  - Stands for LookAhead LR
  - Typically 10 times fewer LALR(1) states than LR(1)

A LALR(1) DFA

- Repeat until all states have distinct core
  - Choose two distinct states with same core
  - Merge the states by creating a new one with the union of all the items
  - Point edges from predecessors to new state
  - New state points to all the previous successors

Conversion LR(1) to LALR(1): Example.





The LALR Parser Can Have Conflicts

- Consider for example the LR(1) states

  $\{[X \to \alpha \,|\, , a], [Y \to \beta \,|\, , b]\}$

  $\{[X \to \alpha \,|\, , b], [Y \to \beta \,|\, , a]\}$

- And the merged LALR(1) state

  $\{[X \to \alpha \,|\, , a/b], [Y \to \beta \,|\, , a/b]\}$

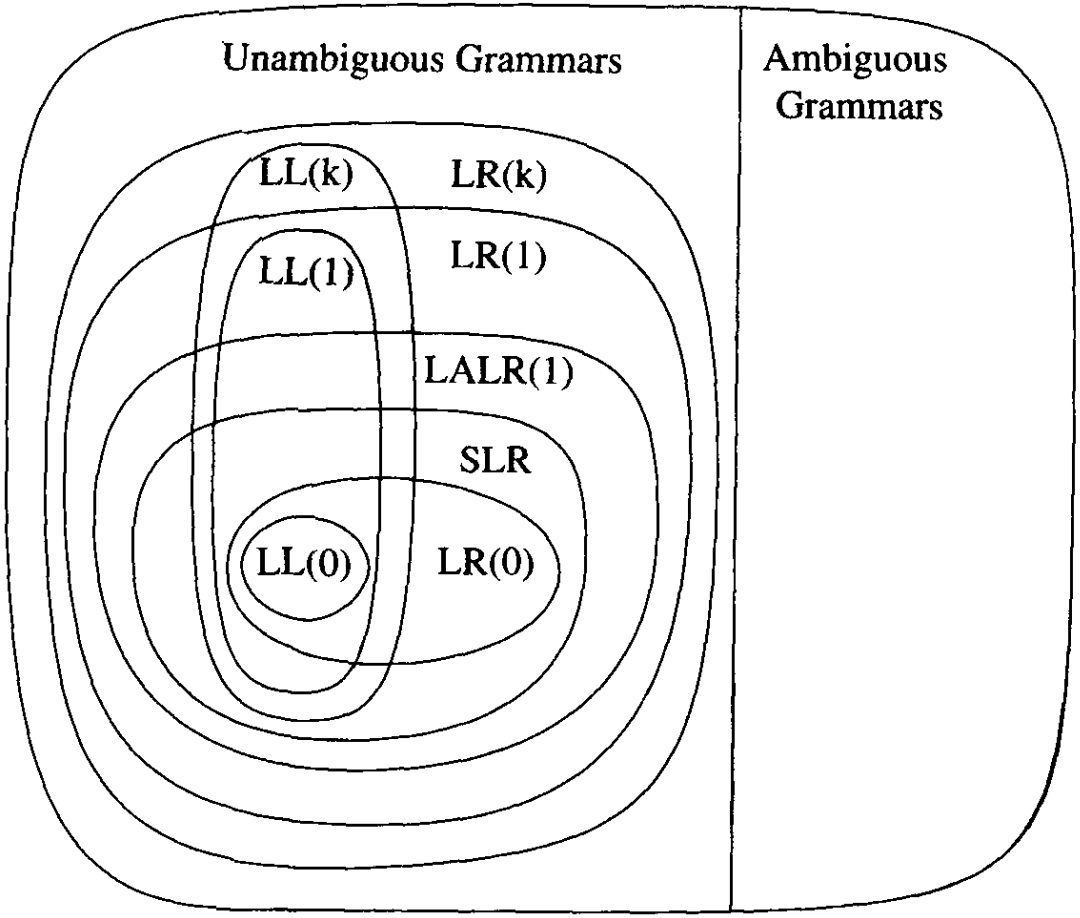
- Has a <u>new</u> reduce/reduce conflict

- In practice such cases are rare

LALR vs. LR Parsing: Things to keep in mind

- LALR languages are not natural
  - They are an efficiency hack on LR languages
- Any reasonable programming language has a LALR(1) grammar
- LALR(1) parsing has become a standard for programming languages and parser generators

A Hierarchy of Grammar Classes



From Andrew Appel, "Modern Compiler Implementation in ML"