Lecture 12: Deterministic Bottom-Up Parsing

- (From slides by G. Necula & R. Bodik)

Administrivia

- HW4 out.
- HW4 contains a team component, part of Project #1: Test cases.

Avoiding nondeterministic choice: LR

- We've been looking at general context-free parsing.
- It comes at a price, measured in overheads, so in practice, we design programming languages to be parsed by less general but faster means, like top-down recursive descent.
- Deterministic bottom-up parsing is more general than top-down parsing, and just as efficient.
- Most common form is 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 : E + (E) | 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. In the following, sent is a sentential form that starts as the input and is reduced to the start symbol, S:

sent = input string of terminals

while sent $\neq$ S:

Identify first β in sent such that $A : \beta$ is a production

and $S \stackrel{*}{\Longrightarrow} \alpha A \gamma \Rightarrow \alpha\beta\gamma =$ sent.

Replace β by A in sent (so that $\alpha A \gamma$ becomes new sent).

- Such $\alpha\beta$'s are called *handles*.

A Bottom-up Parse in Detail (1)

Grammar:

E : E + (E) | int

`int + (int) + (int)`

int + (int) + (int)

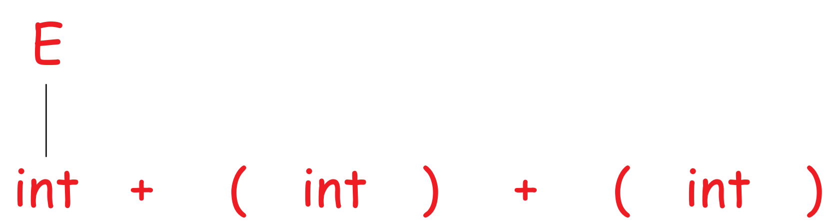
A Bottom-up Parse in Detail (2)

Grammar:

E : E + (E) | int

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

(handles in red)

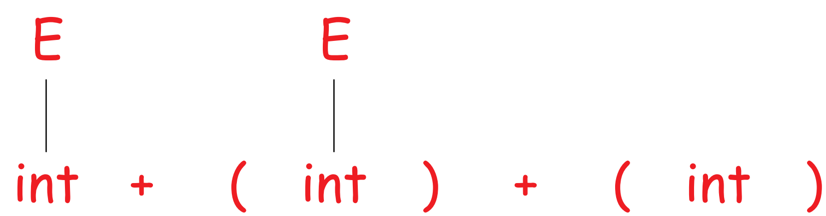


A Bottom-up Parse in Detail (3)

Grammar:

E : E + (E) | int

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

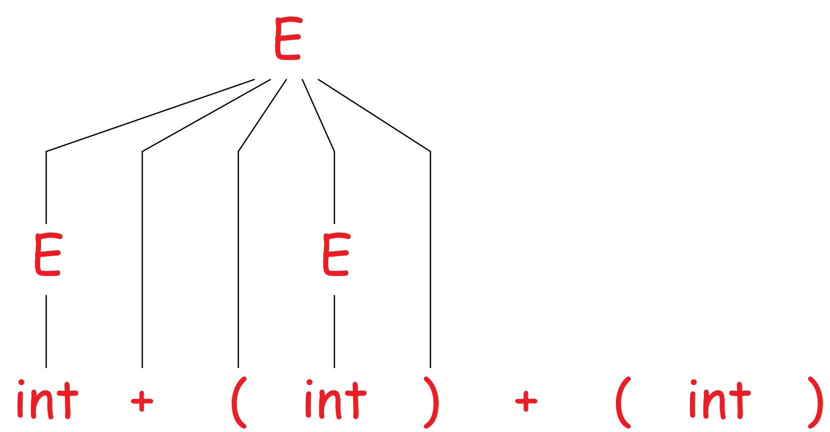


A Bottom-up Parse in Detail (4)

Grammar:

E : E + (E) | int

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

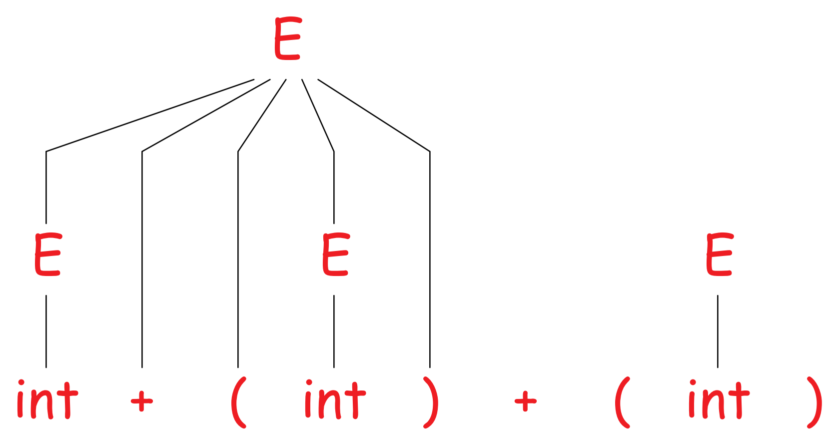


A Bottom-up Parse in Detail (5)

Grammar:

E : E + (E) | int

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



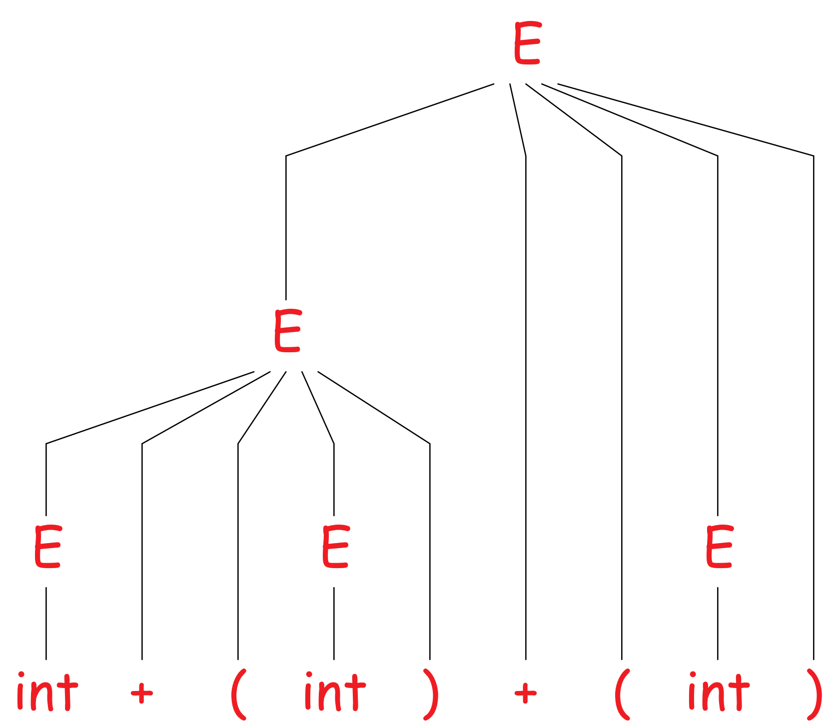
A Bottom-up Parse in Detail (6)

Grammar:

E : E + (E) | int

A reverse rightmost derivation:

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



Where Do Reductions Happen?

Because an LR parser produces a reverse rightmost derivation:

- If $\alpha\beta\gamma$ is one step of a bottom-up parse with handle $\alpha\beta$
- And the next reduction is by $A : \beta$,
- Then γ must be a string of terminals,
- Because $\alpha A\gamma \Rightarrow \alpha\beta\gamma$ is a step in a rightmost derivation

Intuition: We make decisions about what reduction to use after seeing *all* symbols in the handle, rather after seeing only the first (as for LL(1)).

Notation

- Idea: Split the input string into two substrings
  - Right substring (a string of terminals) is as-yet unprocessed by parser
  - Left substring has terminals and nonterminals
  - (In examples, we'll mark the dividing point with |.)
  - The dividing point marks the end of the next potential handle.
  - Initially, all input is unexamined: $|x_1 x_2 \cdots x_n$

Shift-Reduce Parsing

Bottom-up parsing uses only two kinds of actions:

- *Shift:* Move | one place to the right, shifting a terminal to the left string.
  - For example,

    E + (| int) ⟶ E + (int |)

- *Reduce:* Apply an inverse production at the handle.
  - For example, if E : E + (E) is a production, then we might reduce:

    E + (E + (E) |) ⟶ E +(E |)

Accepting a String

- The process ends when we reduce all the input to the start symbol.
- For technical convenience, however, we usually add a new start symbol and a hidden production to handle the end-of-file:

  $S' : S \dashv$

- Having done this, we can now stop parsing and accept the string whenever we reduce the entire input to

  $S \mid \dashv$

  without bothering to do the final shift and reduce.
- This will be the convention from now on.

Shift-Reduce Example (1)

Grammar:

E : E + (E) | int

Sent. Form	Actions
| <u>int</u> + (int) + (int) ⊣	shift

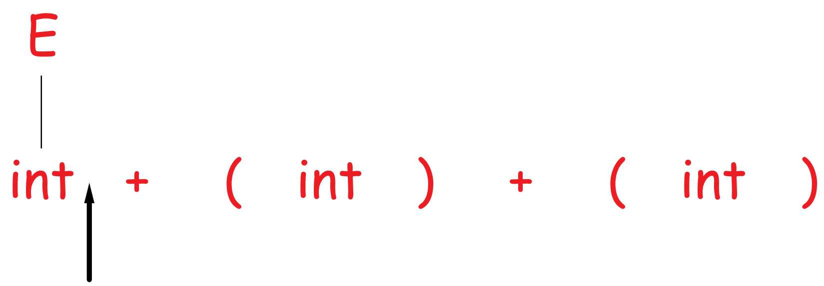
int + (int) + (int)

Shift-Reduce Example (2)

Sent. Form	Actions
| int + (int) + (int) ⊣	shift
int | + (int) + (int) ⊣	reduce by E: int

Grammar:

E : E + (E) | int

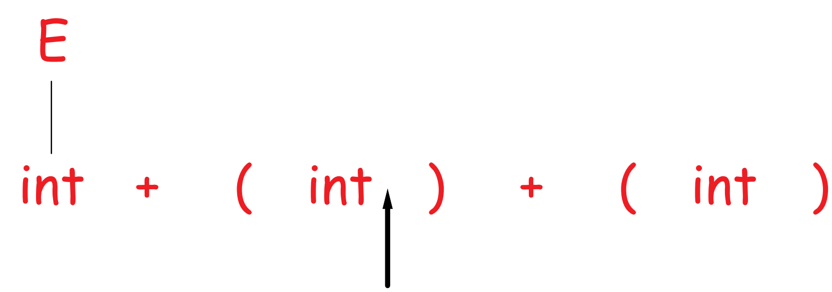


Shift-Reduce Example (3)

Grammar:

E : E + (E) | int

<table>
<tr><th>Sent. Form</th><th>Actions</th></tr>
<tr><td>| int + (int) + (int) ⊣</td><td>shift</td></tr>
<tr><td>int | + (int) + (int) ⊣</td><td>reduce by E: int</td></tr>
<tr><td>E | + (int) + (int) ⊣</td><td>shift 3 times</td></tr>
</table>

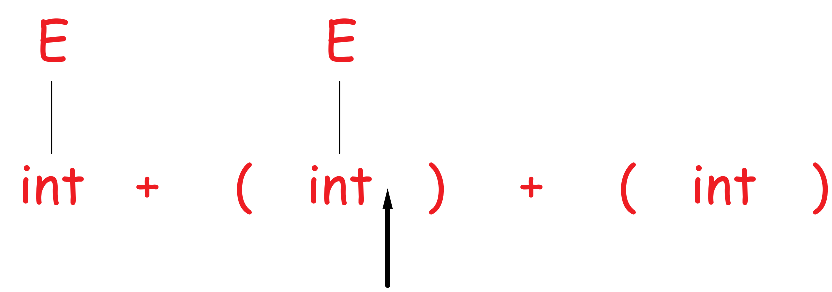


Shift-Reduce Example (4)

<table>
<tr><th>Sent. Form</th><th>Actions</th></tr>
<tr><td>| int + (int) + (int) ⊣</td><td>shift</td></tr>
<tr><td>int | + (int) + (int) ⊣</td><td>reduce by E: int</td></tr>
<tr><td>E | + (int) + (int) ⊣</td><td>shift 3 times</td></tr>
<tr><td>E + (int |) + (int) ⊣</td><td>reduce by E: int</td></tr>
</table>

Grammar:

E : E + (E) | int

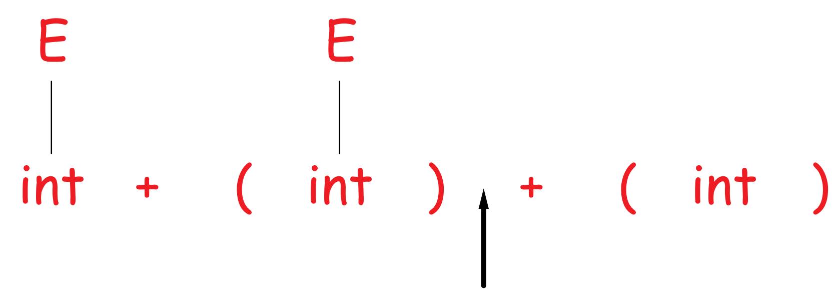


Shift-Reduce Example (5)

Sent. Form	Actions
| int + (int) + (int) ⊣	shift
int | + (int) + (int) ⊣	reduce by E: int
E | + (int) + (int) ⊣	shift 3 times
E + (int |) + (int) ⊣	reduce by E: int
E + (E |) + (int) ⊣	shift

Grammar:

E : E + (E) | int

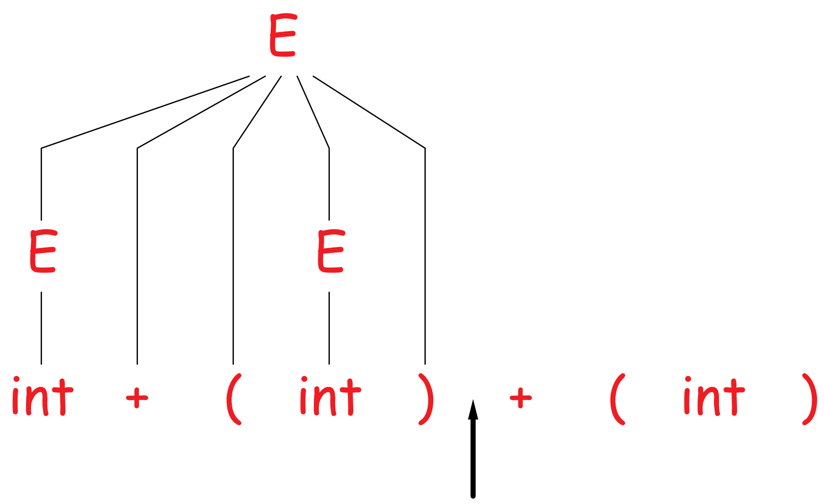


Shift-Reduce Example (6)

Sent. Form	Actions
| int + (int) + (int) ⊣	shift
int | + (int) + (int) ⊣	reduce by E: int
E | + (int) + (int) ⊣	shift 3 times
E + (int |) + (int) ⊣	reduce by E: int
E + (E |) + (int) ⊣	shift
E + (E) | + (int) ⊣	reduce by E: E+(E)

Grammar:

E : E + (E) | int

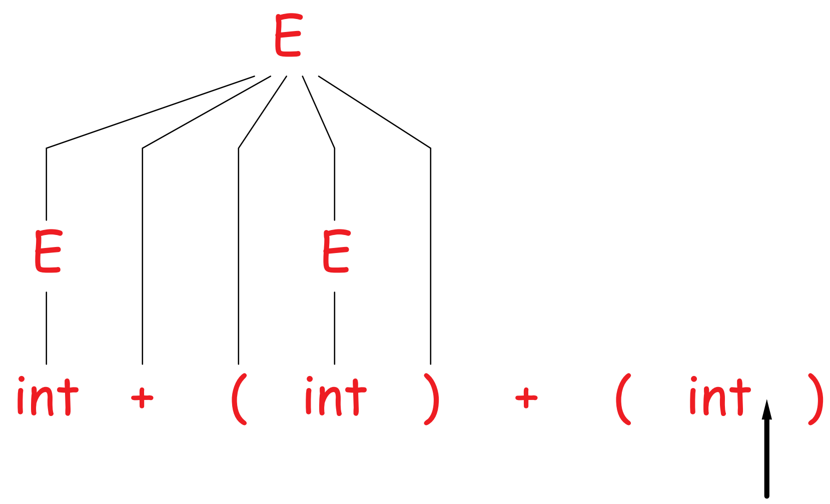


Shift-Reduce Example (7)

Sent. Form	Actions
| int + (int) + (int) ⊣	shift
int | + (int) + (int) ⊣	reduce by E: int
E | + (int) + (int) ⊣	shift 3 times
E + (int |) + (int) ⊣	reduce by E: int
E + (E |) + (int) ⊣	shift
E + (E) | + (int) ⊣	reduce by E: E+(E)
E | + (int) ⊣	shift 3 times

Grammar:

E : E + (E) | int

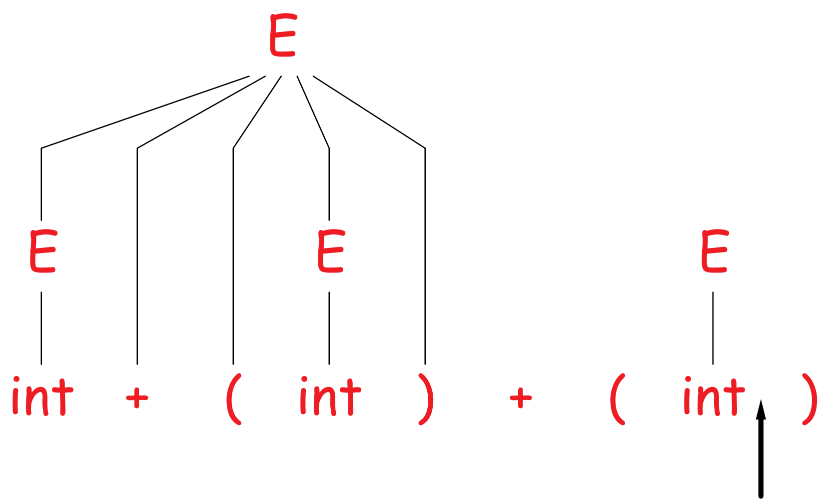


Shift-Reduce Example (8)

Sent. Form	Actions
| int + (int) + (int) ⊣	shift
int | + (int) + (int) ⊣	reduce by E: int
E | + (int) + (int) ⊣	shift 3 times
E + (int |) + (int) ⊣	reduce by E: int
E + (E |) + (int) ⊣	shift
E + (E) | + (int) ⊣	reduce by E: E+(E)
E | + (int) ⊣	shift 3 times
E + (int |) ⊣	reduce by E: int

Grammar:

E : E + (E) | int

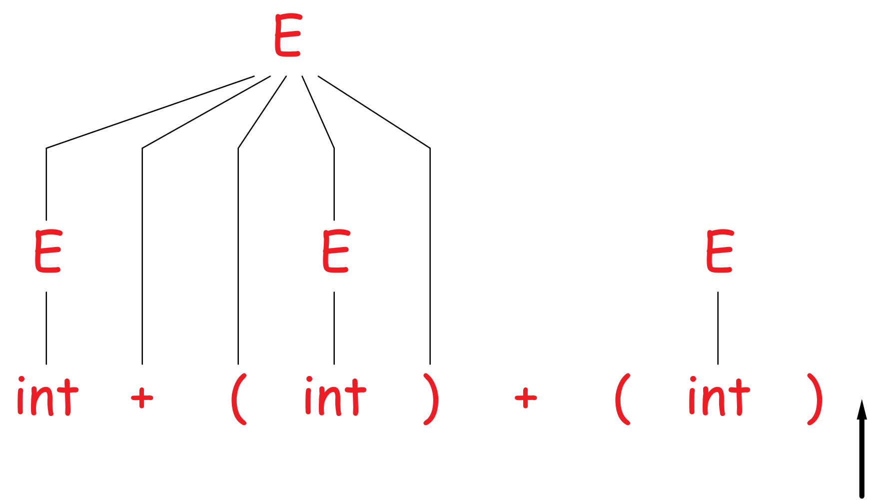


Shift-Reduce Example (9)

Sent. Form	Actions
| int + (int) + (int) ⊣	shift
int | + (int) + (int) ⊣	reduce by E: int
E | + (int) + (int) ⊣	shift 3 times
E + (int |) + (int) ⊣	reduce by E: int
E + (E |) + (int) ⊣	shift
E + (E) | + (int) ⊣	reduce by E: E+(E)
E | + (int) ⊣	shift 3 times
E + (int |) ⊣	reduce by E: int
E + (E |) ⊣	shift

Grammar:

E : E + (E) | int

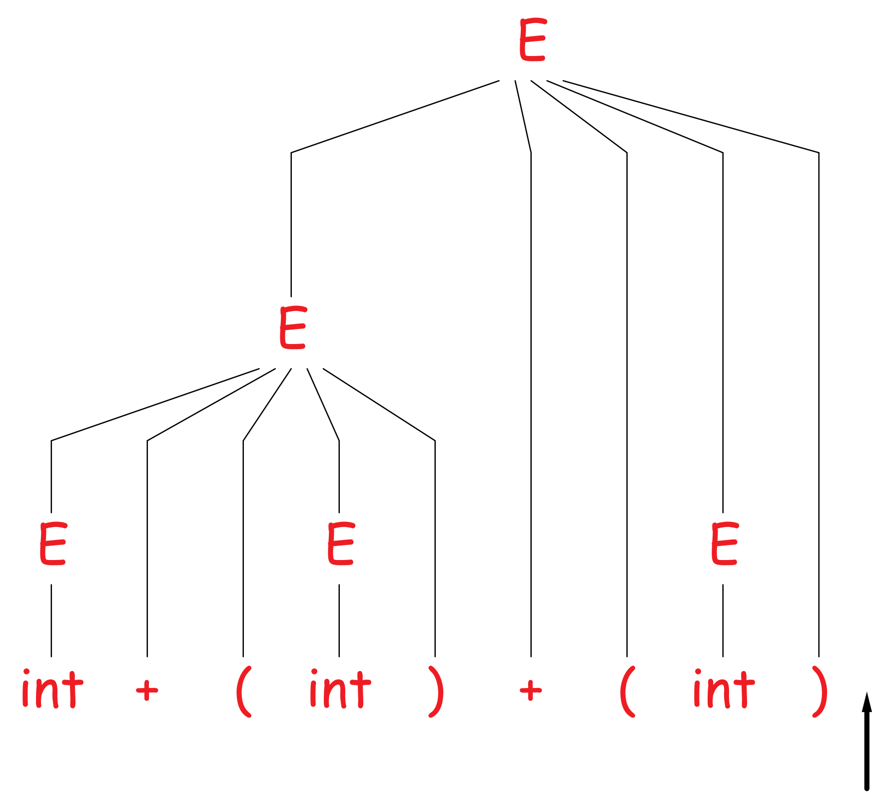


Shift-Reduce Example (10)

Sent. Form	Actions
| int + (int) + (int) ⊣	shift
int | + (int) + (int) ⊣	reduce by E: int
E | + (int) + (int) ⊣	shift 3 times
E + (int |) + (int) ⊣	reduce by E: int
E + (E |) + (int) ⊣	shift
E + (E) | + (int) ⊣	reduce by E: E+(E)
E | + (int) ⊣	shift 3 times
E + (int |) ⊣	reduce by E: int
E + (E |) ⊣	shift
E + (E) | ⊣	reduce by E: E+(E)

Grammar:

E : E + (E) | int

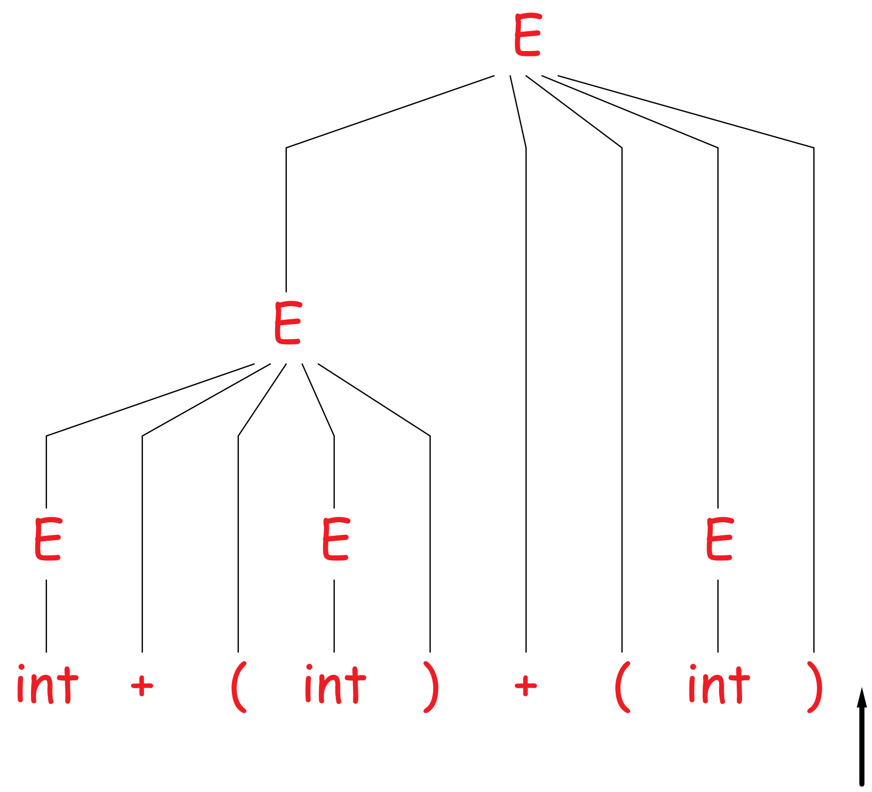


Shift-Reduce Example (11)

Sent. Form	Actions
| int + (int) + (int) ⊣	shift
int | + (int) + (int) ⊣	reduce by E: int
E | + (int) + (int) ⊣	shift 3 times
E + (int |) + (int) ⊣	reduce by E: int
E + (E |) + (int) ⊣	shift
E + (E) | + (int) ⊣	reduce by E: E+(E)
E | + (int) ⊣	shift 3 times
E + (int |) ⊣	reduce by E: int
E + (E |) ⊣	shift
E + (E) |⊣	reduce by E: E+(E)
E |⊣	accept

Grammar:

E : E + (E) | int



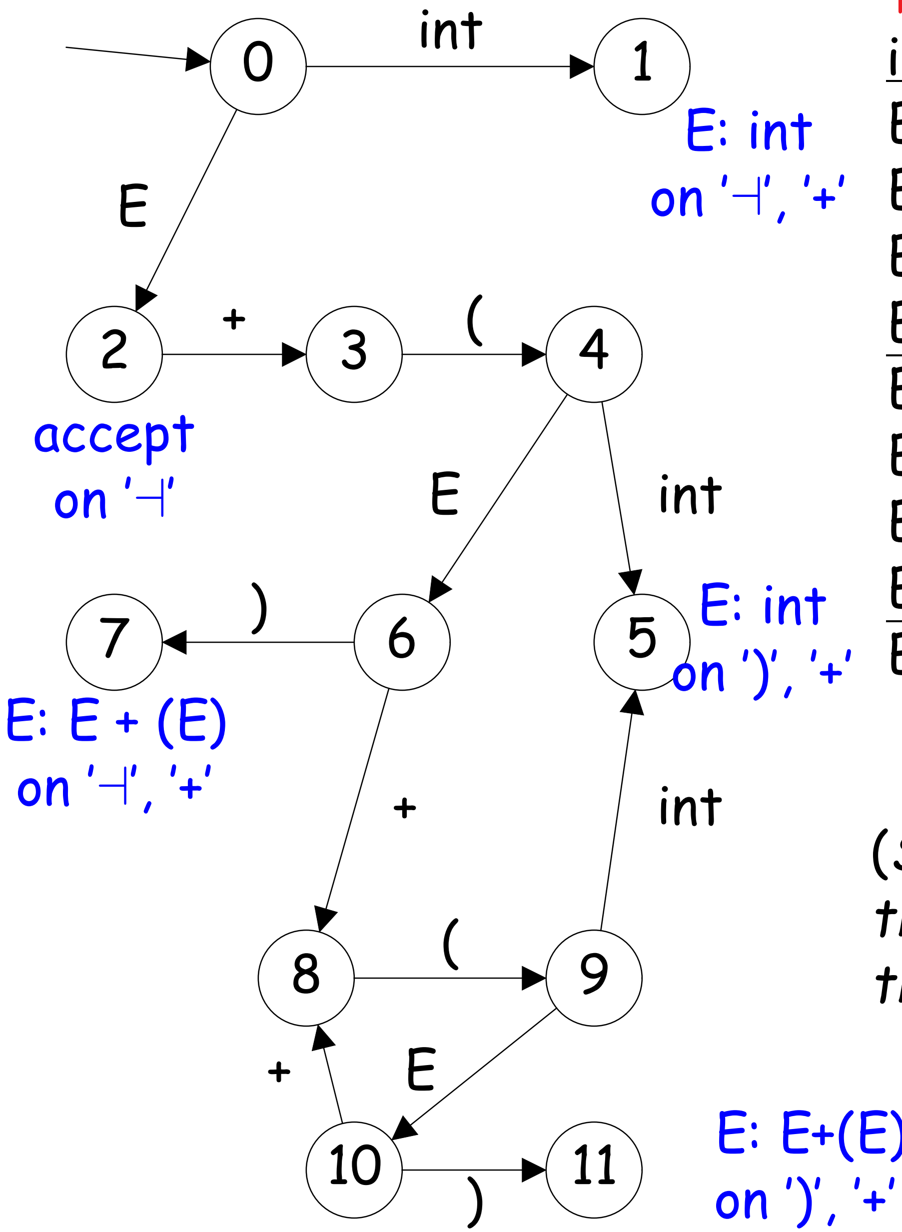
The Parsing Stack

- The left string (left of the |) can be implemented as a stack:
  - Top of the stack is just left of the |.
  - Shift pushes a terminal on the stack.
  - Reduce pops 0 or more symbols from the stack (corresponding to the production's right-hand side) and pushes a nonterminal on the stack (the production's left-hand side).

Key Issue: When to Shift or Reduce?

- Decide based on the left string ("the stack") and some of the remaining input (*lookahead tokens*)—typically one token at most.
- Idea: use a DFA to decide when to shift or reduce:
  - DFA alphabet consists of terminals and nonterminals.
  - The DFA input is the stack up to potential handle.
  - DFA recognizes complete handles.
  - In addition, the final states are labeled with particular productions that might apply, given the possible lookahead symbols.
- We run the DFA on the stack and we examine the resulting state, X and the lookahead token τ after |.
  - If X has a transition labeled τ then shift.
  - If X is labeled with "$A : \beta$ on τ," then reduce.
- So we scan the input from Left to right, producing a (reverse) Rightmost derivation, using 1 symbol of lookahead: giving LR(1) parsing.

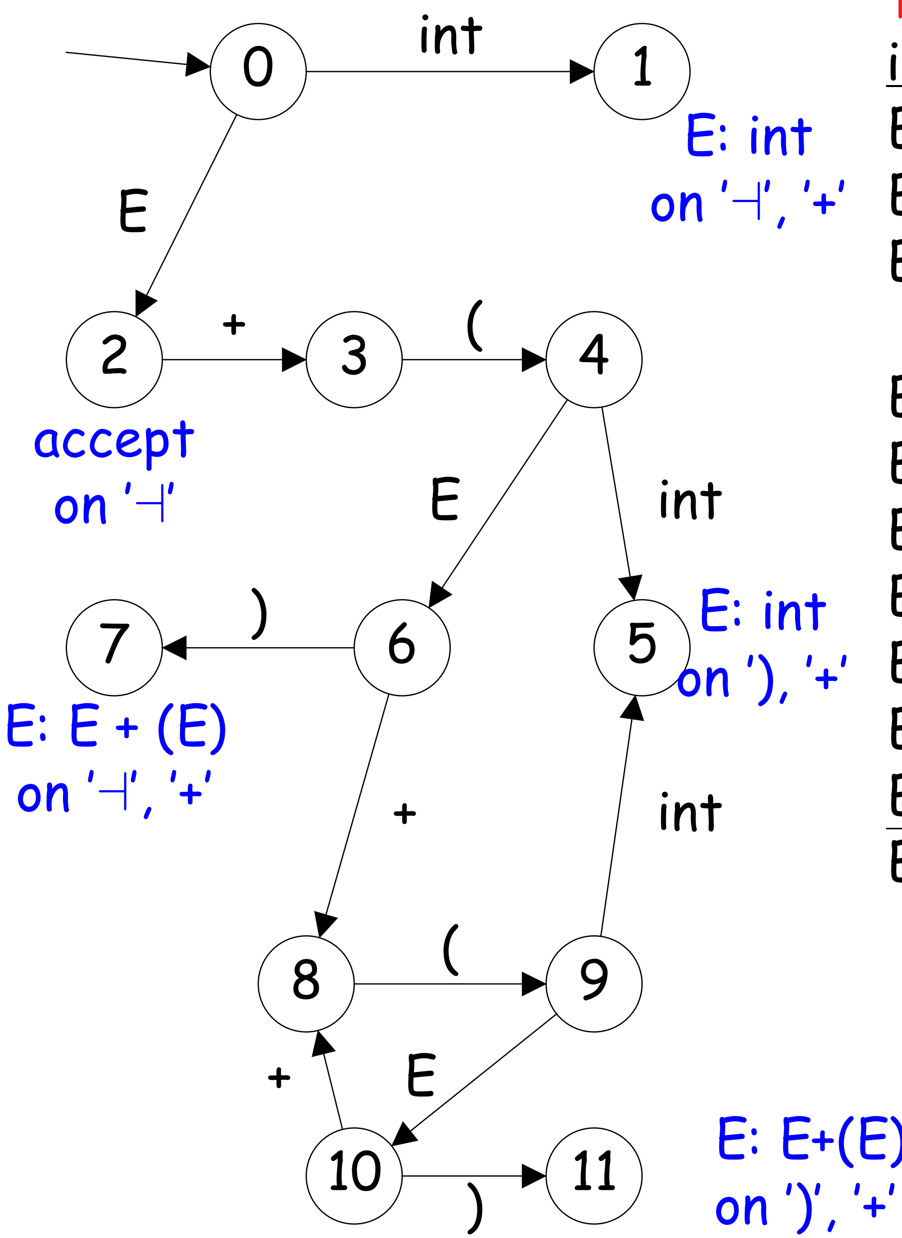
LR(1) Parsing. An Example



$\vert_0$ int + (int) + (int) ⊣	shift
int $\vert_1$ + (int) + (int) ⊣	red. by E: int
E $\vert_2$ + (int) + (int) ⊣	shift 3 times
E + (int $\vert_5$) + (int) ⊣	red. by E: int
E + (E $\vert_6$) + (int) ⊣	shift
E + (E) $\vert_7$ + (int) ⊣	red. by E: E+(E)
E $\vert_2$ + (int) ⊣	shift 3 times
E + (int $\vert_5$) ⊣	red. by E: int
E + (E $\vert_6$) ⊣	shift
E + (E) $\vert_7$ ⊣	red. by E: E+(E)
E $\vert_2$ ⊣	accept

(Subscripts on | show the states that the DFA reaches by scanning the left string.)

LR(1) Parsing. Another Example



$|_0$ int + (int + (int + (int))) ⊣	shift
int $|_1$ + (int + (int + (int)))⊣	red. by E: int
E $|_2$ + (int) + (int + (int))) ⊣	shift 3 times
E + (int $|_5$) + (int + (int))) ⊣	red. by E: int
E + (E $|_6$) + (int + (int))) ⊣	shift
⋮	⋮
E + (E + (E + (int $|_5$))) ⊣	red. by E: int
E + (E + (E + (E $|_{10}$))) ⊣	shift
E + (E + (E + (E) $|_{11}$)) ⊣	red. by E: E + (E)
E + (E + (E $|_{10}$)) ⊣	shift
E + (E + (E) $|_{11}$) ⊣	red. by E: E + (E)
E + (E $|_6$) ⊣	shift
E + (E) $|_7$ ⊣	red. by E: E + (E)
E $|_2$ ⊣	accept

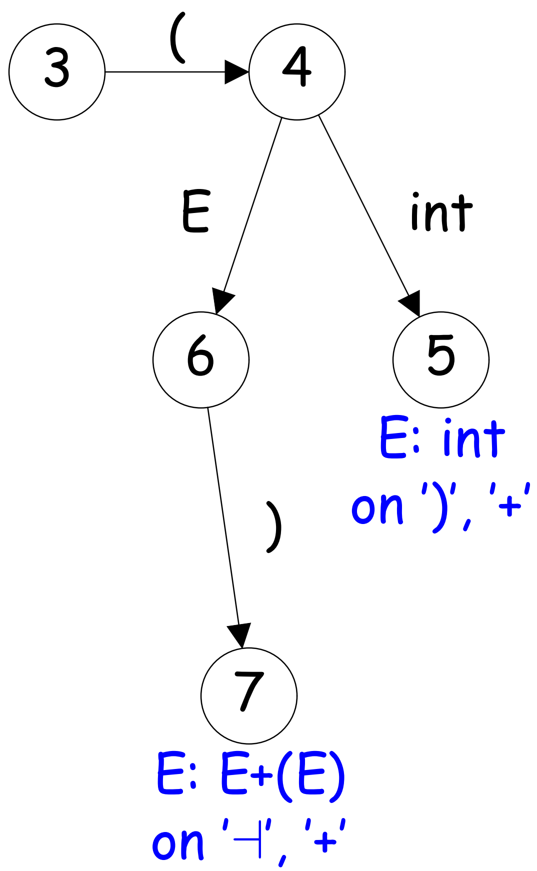
Representing the DFA

- Parsers represent the DFA as a 2D table, as for table-driven lexical analysis
- Lines correspond to DFA states
- Columns correspond to terminals and nonterminals
- Classical treatments (like Aho, *et al*) split the columns into:
  - Those for terminals: the *action table*.
  - Those for nonterminals: the *goto table*.

  The goto table contains only shifts, but conceptually, the tables are very much alike as far as the DFA is concerned.
- The classical division has some advantages when it comes to table compression.

Representing the DFA. Example

Here's the table for a fragment of our DFA:



	int	+	(	)	⊣	E
...						
3			s4			
4	s5					s6
5		$r_{\text{E: int}}$		$r_{\text{E: int}}$		
6				s7		
7		$r_{\text{E: E+(E)}}$			$r_{\text{E: E+(E)}}$	
...						

Legend: 'sN' means "shift (or go to) state N."
'r_P' means "reduce using production P."
blank entries indicate errors.

A Little Optimization

- After a shift or reduce action we rerun the DFA on the entire stack.
- This is wasteful, since most of the work is repeated, so
- Memoize: instead of putting terminal and nonterminal symbols on the stack, put the DFA states you get to after reading those symbols.
- For example, when we've reached this point:

  E + (E + (E + ($\underline{\text{int}}\,|_5$))) $\dashv$

  store the part to the left of | as

  0 2 3 4 6 8 9 10 8 9 5

- And don't throw any of these away until you reduce them.

The Actual LR Parsing Algorithm

```
Let I = w1w2...wn be initial input
Let j = 1
Let stack = < 0 >

repeat
   case table[top_state(stack), I[j]] of
      sk:
           push k on the stack; j += 1
      rX: α:
           pop len(α) symbols from stack
           push j on stack, where table[top_state(stack), X] is sj.
      accept:
           return normally
      error:
           return parsing error indication
```

Parsing Contexts

- Consider the state describing the situation at the | in the stack E + (| int)+(int), which tells us
  - We are looking to reduce E: E + (E), having already seen E + (from the right-hand side.
  - Therefore, we expect that the rest of the input starts with something that will eventually reduce to E:

    E: int or E: E+(E)

    after which we expect to find a ')',
  - but we have as yet seen nothing from the right-hand sides of either of these two possible productions.
- One DFA state captures a set of such contexts in the form of a set of *LR(1) items*, like this:

```
[ E: E + ( • E ), ... ]       [ E: • int, '+' ] (why?)
[ E: • int, ')' ]             [ E: • E+(E), '+' ] (why?)
[ E: • E+(E), ')' ]
```

- (Traditionally, use • in items to show where the | is.)

LR(1) Items

- An LR(1) item is a pair:

  X: α•β, a

  - X: $\alpha\beta$ is a production.
  - a is a terminal symbol (an expected lookahead).
- It says we are trying to find an X followed by a.
- and that we have already accumulated α on top of the parsing stack.
- Therefore, we need to see next a prefix of something derived from βa.
- (As an abbreviation, we'll usually write

  X: α•β, a/b

  to mean the *two* LR(1) items

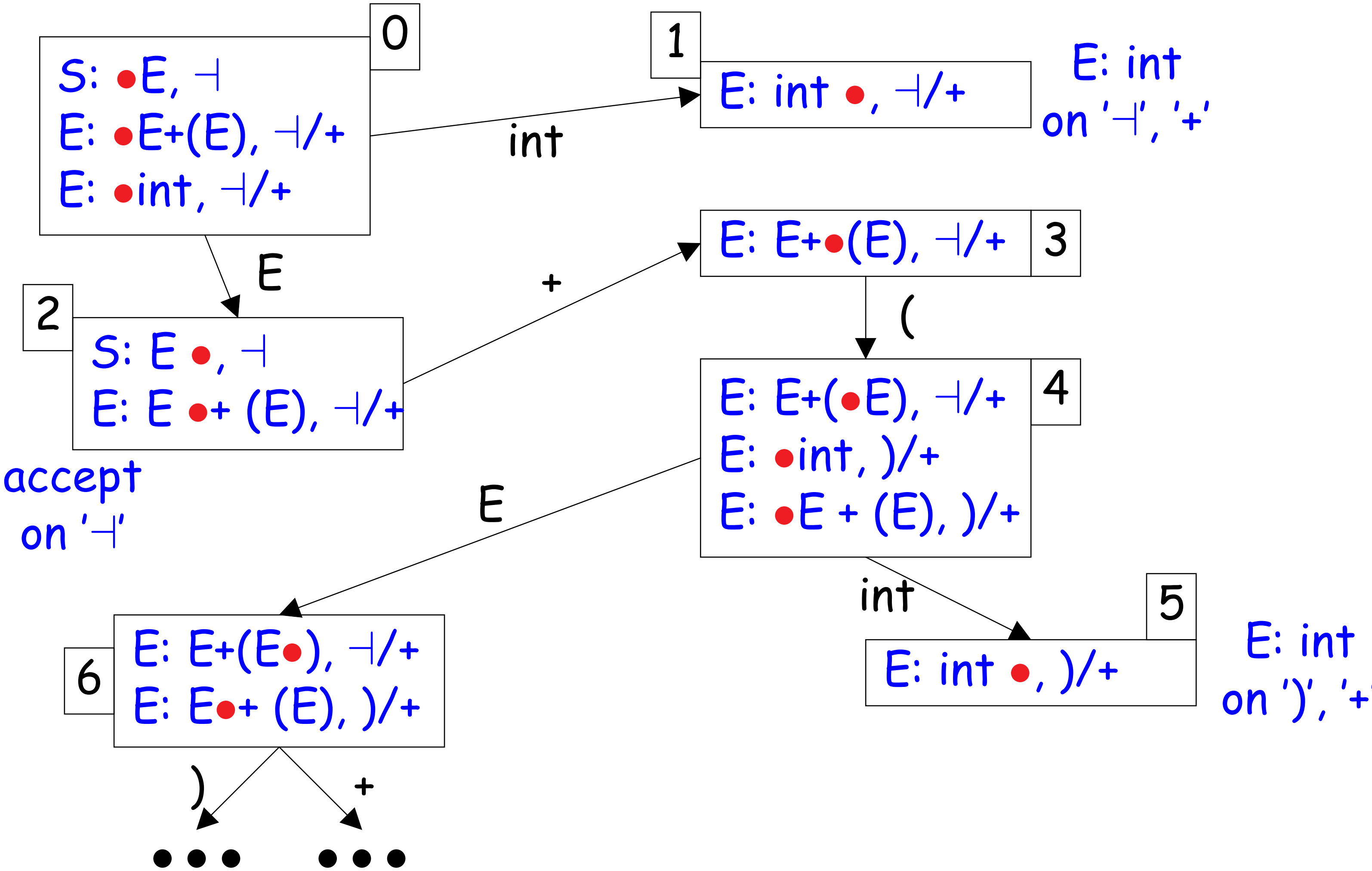
  X: α•β, a
  X: α•β, b

)

Constructing the Parsing DFA

- The idea is to borrow from Earley's algorithm (where we've already seen this notation!).
- We throw away a lot of the information that Earley's algorithm keeps around (notably where in the input each current item got introduced), because when we have a handle, there will only be one possible reduction to take based on what we've seen so far.
- This allows the set of possible item sets to be finite.
- Each state in the DFA has an item set that is derived from what Earley's algorithm would do, but collapsed because of the information we throw away.

Constructing the Parsing DFA: Partial Example



LR Parsing Tables. Notes

- We really want to construct parsing tables (i.e. the DFA) from CFGs automatically, since this construction is tedious.
- But still good to understand the construction to work with parser generators, which report errors in terms of sets of items.
- What kind of errors can we expect?

Relation to Bison

- Bison builds this kind of machine.
- However, for efficiency concerns, collapses many of the states together, namely those that differ only in lookahead sets, but otherwise have identical sets of items. Result is called an *LALR(1) parser* (as opposed to LR(1)).
- Causes some additional conflicts, but these are rare.