

Lecture 7: Deterministic Bottom-Up Parsing

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

Last modified: Wed Sep 26 12:37:02 2012

CS164: Lecture #7 1

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

Last modified: Wed Sep 26 12:37:02 2012

CS164: Lecture #7 2

An Introductory Example

- LR parsers don't need left-factored grammars and can also handle left-recursive grammars
- Consider the following grammar:
$$E : E + (E) \mid \text{int}$$

(Why is this not LL(1)?)
- Consider the string: $\text{int} + (\text{int}) + (\text{int}) .$

Last modified: Wed Sep 26 12:37:02 2012

CS164: Lecture #7 3

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 $\text{sent} \neq S$:
 Identify first β in sent such that $A : \beta$ is a production
 and $S \xRightarrow{*} \alpha A \gamma \Rightarrow \alpha \beta \gamma = \text{sent}$.
 Replace β by A in sent (so that $\alpha A \gamma$ becomes new sent).
- Such $\alpha\beta$'s are called *handles*.

Last modified: Wed Sep 26 12:37:02 2012

CS164: Lecture #7 4

A Bottom-up Parse in Detail (1)

Grammar:

$E : E + (E) \mid \text{int}$

`int + (int) + (int)`

`int + (int) + (int)`

Last modified: Wed Sep 26 12:37:02 2012

CS164: Lecture #7 5

A Bottom-up Parse in Detail (2)

Grammar:

$E : E + (E) \mid \text{int}$

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

(handles in red)

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

Last modified: Wed Sep 26 12:37:02 2012

CS164: Lecture #7 6

A Bottom-up Parse in Detail (3)

Grammar:

$E : E + (E) \mid \text{int}$

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

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

Last modified: Wed Sep 26 12:37:02 2012

CS164: Lecture #7 7

A Bottom-up Parse in Detail (4)

Grammar:

$E : E + (E) \mid \text{int}$

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

E
/ | | | \
 E $+$ $($ E $)$ $+$ $($ int $)$

Last modified: Wed Sep 26 12:37:02 2012

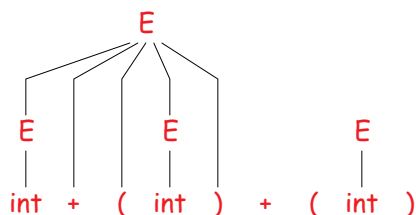
CS164: Lecture #7 8

A Bottom-up Parse in Detail (5)

Grammar:

$E : E + (E) \mid \text{int}$

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



Last modified: Wed Sep 26 12:37:02 2012

CS164: Lecture #7 9

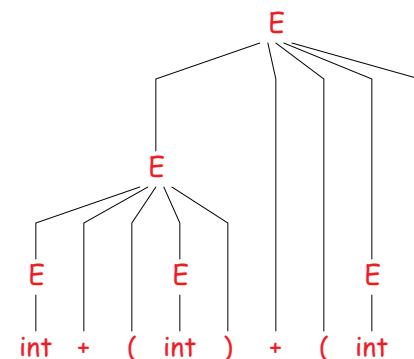
A Bottom-up Parse in Detail (6)

Grammar:

$E : E + (E) \mid \text{int}$

A reverse rightmost
 derivation:

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



Last modified: Wed Sep 26 12:37:02 2012

CS164: Lecture #7 10

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)).

Last modified: Wed Sep 26 12:37:02 2012

CS164: Lecture #7 11

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: $|x_1x_2 \cdots x_n$

Last modified: Wed Sep 26 12:37:02 2012

CS164: Lecture #7 12

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

$$E + (E + (E) |) \longrightarrow E + (E |)$$

Last modified: Wed Sep 26 12:37:02 2012

CS164: Lecture #7 13

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

$$S | \dashv$$

without bothering to do the final shift and reduce.

- This will be the convention from now on.

Last modified: Wed Sep 26 12:37:02 2012

CS164: Lecture #7 14

Shift-Reduce Example (1)

Sent. Form	Actions
$ \text{ int } + (\text{ int }) + (\text{ int }) \dashv$	shift

Grammar:

$$E : E + (E) | \text{ int}$$

\uparrow int + (int) + (int)

Last modified: Wed Sep 26 12:37:02 2012

CS164: Lecture #7 15

Shift-Reduce Example (2)

Sent. Form	Actions
$ \text{ int } + (\text{ int }) + (\text{ int }) \dashv$	shift
$\text{int } + (\text{ int }) + (\text{ int }) \dashv$	reduce by $E : \text{ int}$

Grammar:

$$E : E + (E) | \text{ int}$$

$\begin{array}{c} E \\ | \\ \text{int} \end{array} \uparrow + (\text{ int }) + (\text{ int })$

Last modified: Wed Sep 26 12:37:02 2012

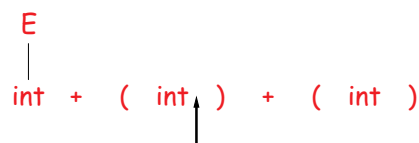
CS164: Lecture #7 16

Shift-Reduce Example (3)

Grammar:

$E : E + (E) \mid \text{int}$

Sent. Form	Actions
<u>int</u> + (int) + (int) ⇐	shift
<u>int</u> + (int) + (int) ⇐	reduce by E: int
E <u>+</u> (int) + (int) ⇐	shift 3 times



Last modified: Wed Sep 26 12:37:02 2012

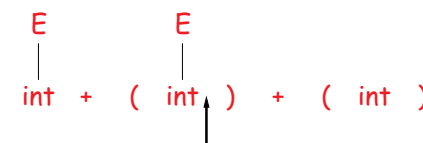
CS164: Lecture #7 17

Shift-Reduce Example (4)

Grammar:

$E : E + (E) \mid \text{int}$

Sent. Form	Actions
<u>int</u> + (int) + (int) ⇐	shift
<u>int</u> + (int) + (int) ⇐	reduce by E: int
E <u>+</u> (int) + (int) ⇐	shift 3 times
E + (<u>int</u>) + (int) ⇐	reduce by E: int



Last modified: Wed Sep 26 12:37:02 2012

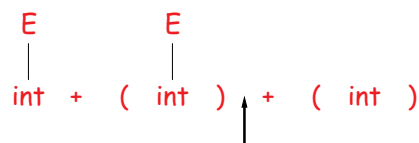
CS164: Lecture #7 18

Shift-Reduce Example (5)

Grammar:

$E : E + (E) \mid \text{int}$

Sent. Form	Actions
<u>int</u> + (int) + (int) ⇐	shift
<u>int</u> + (int) + (int) ⇐	reduce by E: int
E <u>+</u> (int) + (int) ⇐	shift 3 times
E + (<u>int</u>) + (int) ⇐	reduce by E: int
E + (E) + (int) ⇐	shift



Last modified: Wed Sep 26 12:37:02 2012

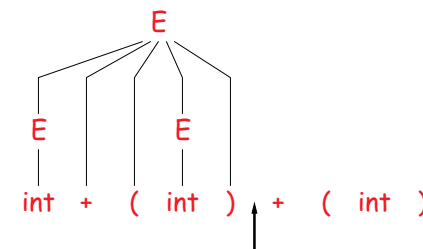
CS164: Lecture #7 19

Shift-Reduce Example (6)

Grammar:

$E : E + (E) \mid \text{int}$

Sent. Form	Actions
<u>int</u> + (int) + (int) ⇐	shift
<u>int</u> + (int) + (int) ⇐	reduce by E: int
E <u>+</u> (int) + (int) ⇐	shift 3 times
E + (<u>int</u>) + (int) ⇐	reduce by E: int
E + (E) + (int) ⇐	shift
<u>E + (E</u> + (int) ⇐	reduce by E: E+(E)



Last modified: Wed Sep 26 12:37:02 2012

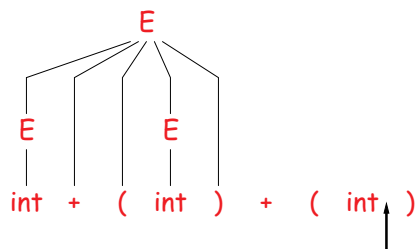
CS164: Lecture #7 20

Shift-Reduce Example (7)

Grammar:

$E : E + (E) \mid \text{int}$

Sent. Form	Actions
<u>int</u> + (int) + (int) ⇐	shift
<u>int</u> + (int) + (int) ⇐	reduce by E: int
E <u>+</u> (int) + (int) ⇐	shift 3 times
E + (<u>int</u>) + (int) ⇐	reduce by E: int
E + (E) + (int) ⇐	shift
E + (E) + (int) ⇐	reduce by E: E+(E)
E <u>+</u> (int) ⇐	shift 3 times



Last modified: Wed Sep 26 12:37:02 2012

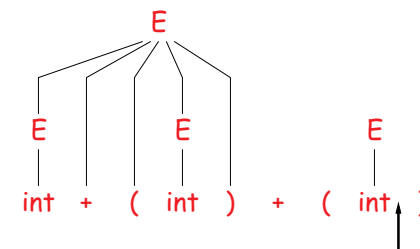
CS164: Lecture #7 21

Shift-Reduce Example (8)

Grammar:

$E : E + (E) \mid \text{int}$

Sent. Form	Actions
<u>int</u> + (int) + (int) ⇐	shift
<u>int</u> + (int) + (int) ⇐	reduce by E: int
E <u>+</u> (int) + (int) ⇐	shift 3 times
E + (<u>int</u>) + (int) ⇐	reduce by E: int
E + (E) + (int) ⇐	shift
E + (E) + (int) ⇐	reduce by E: E+(E)
E <u>+</u> (int) ⇐	shift 3 times
E + (<u>int</u>) ⇐	reduce by E: int



Last modified: Wed Sep 26 12:37:02 2012

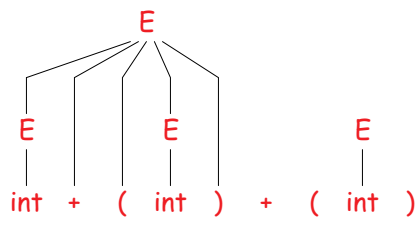
CS164: Lecture #7 22

Shift-Reduce Example (9)

Grammar:

$E : E + (E) \mid \text{int}$

Sent. Form	Actions
<u>int</u> + (int) + (int) ⇐	shift
<u>int</u> + (int) + (int) ⇐	reduce by E: int
E <u>+</u> (int) + (int) ⇐	shift 3 times
E + (<u>int</u>) + (int) ⇐	reduce by E: int
E + (E) + (int) ⇐	shift
E + (E) + (int) ⇐	reduce by E: E+(E)
E <u>+</u> (int) ⇐	shift 3 times
E + (<u>int</u>) ⇐	reduce by E: int
E + (E) ⇐	shift



Last modified: Wed Sep 26 12:37:02 2012

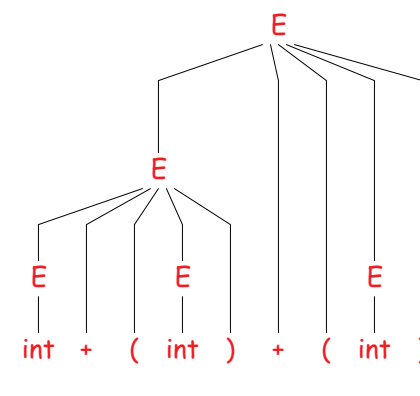
CS164: Lecture #7 23

Shift-Reduce Example (10)

Grammar:

$E : E + (E) \mid \text{int}$

Sent. Form	Actions
<u>int</u> + (int) + (int) ⇐	shift
<u>int</u> + (int) + (int) ⇐	reduce by E: int
E <u>+</u> (int) + (int) ⇐	shift 3 times
E + (<u>int</u>) + (int) ⇐	reduce by E: int
E + (E) + (int) ⇐	shift
E + (E) + (int) ⇐	reduce by E: E+(E)
E <u>+</u> (int) ⇐	shift 3 times
E + (<u>int</u>) ⇐	reduce by E: int
E + (E) ⇐	shift
E + (E) ⇐	reduce by E: E+(E)



Last modified: Wed Sep 26 12:37:02 2012

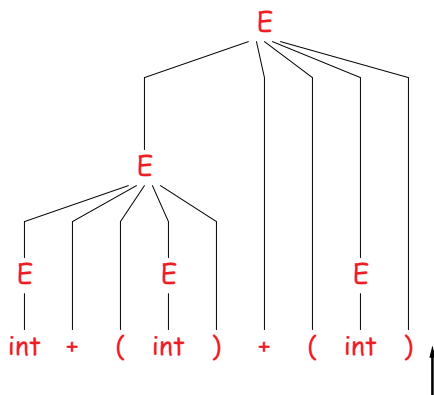
CS164: Lecture #7 24

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:

$E : E + (E) \mid \text{int}$



Last modified: Wed Sep 26 12:37:02 2012

CS164: Lecture #7 25

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).

Last modified: Wed Sep 26 12:37:02 2012

CS164: Lecture #7 26

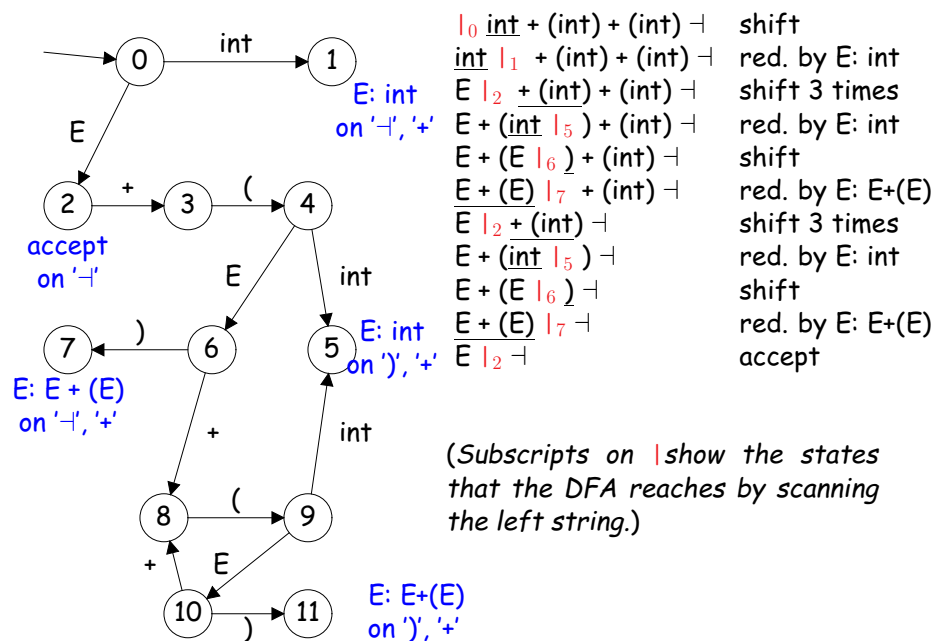
Key Issue: When to Shift or Reduce?

- 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.

Last modified: Wed Sep 26 12:37:02 2012

CS164: Lecture #7 27

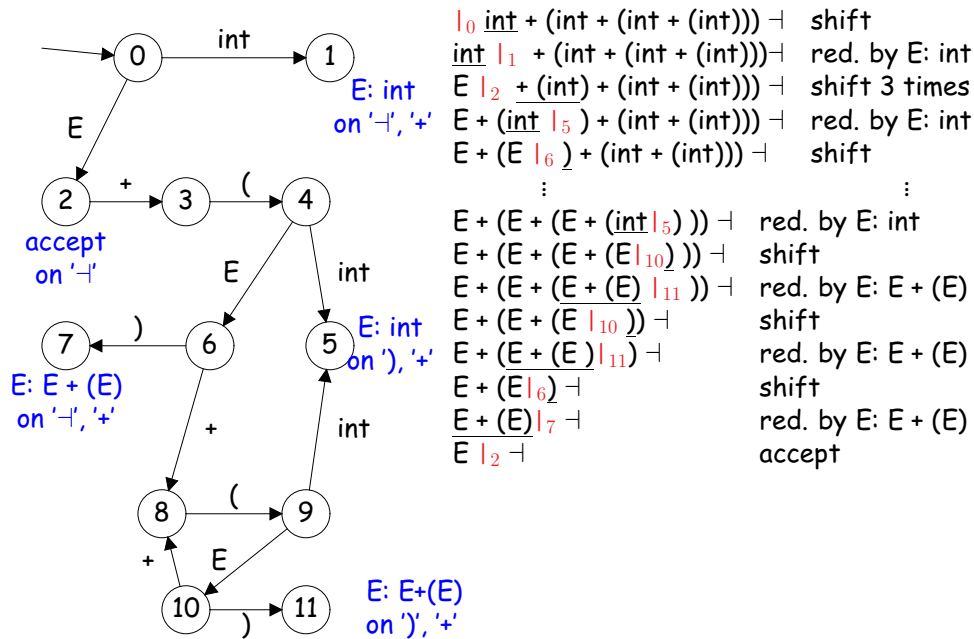
LR(1) Parsing. An Example



Last modified: Wed Sep 26 12:37:02 2012

CS164: Lecture #7 28

LR(1) Parsing. Another Example



Last modified: Wed Sep 26 12:37:02 2012

CS164: Lecture #7 29

Representing the DFA

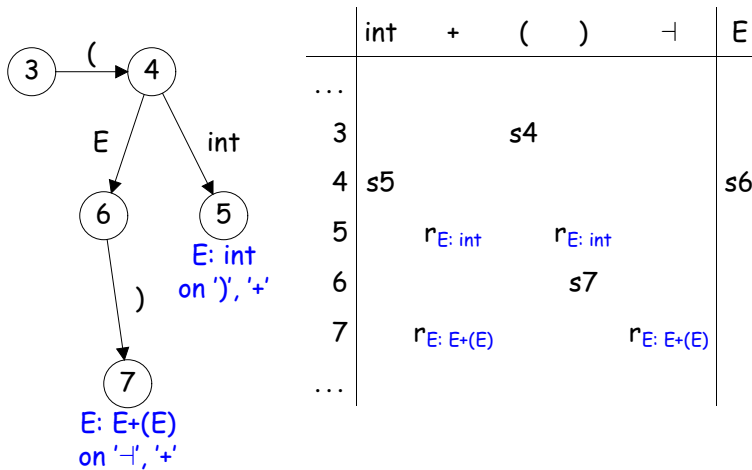
- Parsers represent the DFA as a 2D table, as for table-driven lexical analysis
 - Lines correspond to DFA states
 - Columns correspond to terminals and nonterminals
 - Classical treatments (like Aho, et al) split the columns into:
 - Those for terminals: the *action table*.
 - Those for nonterminals: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

Last modified: Wed Sep 26 12:37:02 2012

CS164: Lecture #7 30

Representing the DFA. Example

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



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

Last modified: Wed Sep 26 12:37:02 2012

CS164: Lecture #7 31

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:

$$E + (E + (E + (int$$
 store the part to the left of int as

$$0\ 2\ 3\ 4\ 6\ 8\ 9\ 10\ 8\ 9\ 5$$
- And don't throw any of these away until you reduce them.

Last modified: Wed Sep 26 12:37:02 2012

CS164: Lecture #7 32

The Actual LR Parsing Algorithm

Let $I = w_1w_2\dots w_n$ be initial input

Let $j = 1$

Let $\text{stack} = \langle 0 \rangle$

repeat

 case $\text{table}[\text{top_state}(\text{stack}), I[j]]$ of

sk :

 push k on the stack; $j += 1$

$rx: \alpha$:

 pop $\text{len}(\alpha)$ symbols from stack

 push j on stack, where $\text{table}[\text{top_state}(\text{stack}), X]$ is sj .

 accept:

 return normally

 error:

 return parsing error indication

Last modified: Wed Sep 26 12:37:02 2012

CS164: Lecture #7 33

Parsing Contexts

- Consider the state describing the situation at the $|$ in the stack $E + (| \text{int}) + (\text{int})$, 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 :
 $E: \text{int}$ 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:

$[E: E + (\bullet E), \dots]$	$[E: \bullet \text{int}, '+']$ (why?)
$[E: \bullet \text{int}, ')']$	$[E: \bullet E+(E), '+']$ (why?)
$[E: \bullet E+(E), ')']$	

- (Traditionally, use \bullet in items to show where the $|$ is.)

Last modified: Wed Sep 26 12:37:02 2012

CS164: Lecture #7 34

LR(1) Items

- An LR(1) item is a pair:
 $X: \alpha \bullet \beta, a$
 - $X: \alpha \beta$ 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

$X: \alpha \bullet \beta, a/b$

to mean the *two* LR(1) items

$X: \alpha \bullet \beta, a$

$X: \alpha \bullet \beta, b$

)

Last modified: Wed Sep 26 12:37:02 2012

CS164: Lecture #7 35

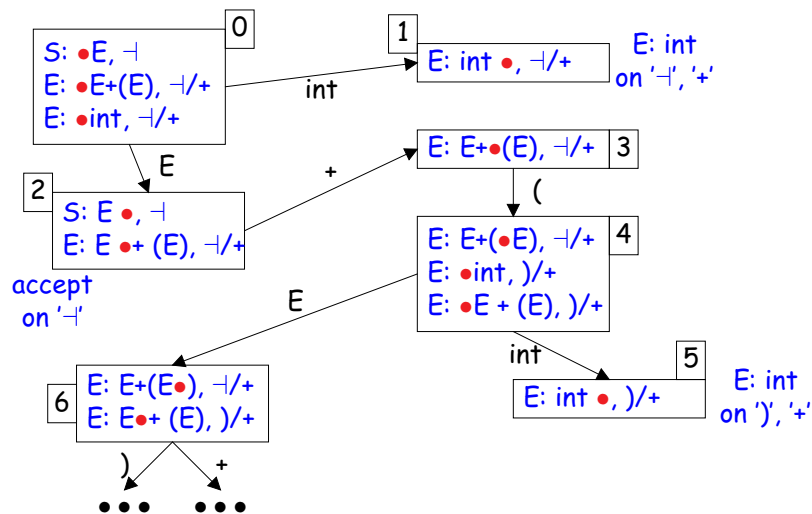
Constructing the Parsing DFA

- 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.

Last modified: Wed Sep 26 12:37:02 2012

CS164: Lecture #7 36

Constructing the Parsing DFA: Partial Example



Last modified: Wed Sep 26 12:37:02 2012

CS164: Lecture #7 37

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?

Last modified: Wed Sep 26 12:37:02 2012

CS164: Lecture #7 38

Shift/Reduce Conflicts

- If a DFA state contains both $[X: \alpha \bullet a \beta, b]$ and $[Y: \gamma \bullet, a]$, then we have two choices when the parser gets into that state at the $|$ and the next input symbol is a :
 - Shift into the state containing $[X: \alpha a \bullet \beta, b]$, or
 - Reduce with $Y: \gamma \bullet$.
- This is called a *shift-reduce conflict*.
- Often due to ambiguities in the grammar. Classic example: the dangling else

$S: \text{"if" } E \text{"then" } S \mid \text{"if" } E \text{"then" } S \text{"else" } S \mid \dots$
- This grammar gives rise to a DFA state containing

$[S: \text{"if" } E \text{"then" } S \bullet, \text{"else"}]$ and $[S: \text{"if" } E \text{"then" } S \bullet \text{"else" } S, \dots]$
- So if "else" is next, we can shift or reduce.

Last modified: Wed Sep 26 12:37:02 2012

CS164: Lecture #7 39

More Shift/Reduce Conflicts

- Consider the ambiguous grammar

$E: E + E \mid E * E \mid int$
- We will have states containing

$[E: E + \bullet E, */+]$	$[E: E + E \bullet, */+]$
$[E: \bullet E + E, */+]$	$[E: E \bullet + E, */+]$
$[E: \bullet E * E, */+]$	$[E: E \bullet * E, */+]$
...	...
- Again we have a shift/reduce conflict on input $'*'$ or $'+'$ (in the item set on the right).
- We probably want to shift on $'*'$ (which is usually supposed to bind more tightly than $'+'$)
- We probably want to reduce on $'+'$ (left-associativity).
- Solution: provide extra information (the precedence of $'*'$ and $'+'$) that allows the parser generator to decide what to do.

Last modified: Wed Sep 26 12:37:02 2012

CS164: Lecture #7 40

Using Precedence in Bison/Horn

- In Bison or Horn, you can declare precedence and associativity of both terminal symbols and rules,
- For terminal symbols (tokens), there are precedence declarations, listed from lowest to highest precedence:

```
%left '+'
%left '*'
%right "**"
```

Symbols on each such line have the same precedence.

- For a rule, precedence = that of its last terminal (Can override with %prec if needed, cf. the Bison manual).
- Now, we resolve shift/reduce conflict with a shift if:
 - The next input token has higher precedence than the rule, or
 - The next input token has the same precedence as the rule and the relevant precedence declaration was %right.
 and otherwise, we choose to reduce the rule.

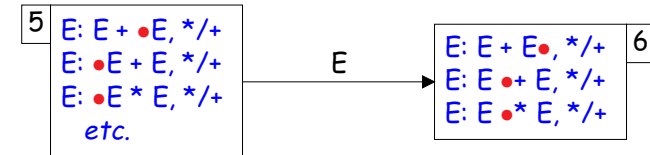
Example of Using Precedence to Solve S/R Conflict (1)

- Assuming we've declared

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

the rule $E: E + E$ will have precedence 1 (left-associative) and the rule $E: E * E$ will have precedence 2.

- So, when the parser confronts the choice in state 6 w/next token '*',



it will choose to shift because the '*' has higher precedence than the rule $E + E$.

- On the other hand, with input symbol '+', it will choose to reduce, because the input token then has the same precedence as the rule to be reduced, and is left-associative.

Example of Using Precedence to Solve S/R Conflict (2)

- Back to our dangling else example. We'll have the state

```
10 S: "if" E "then" S •, "else"
   S: "if" E "then" S • "else" S, "else"
   etc.
```

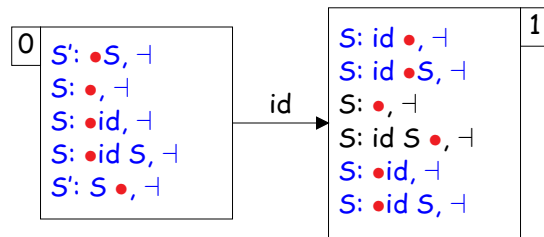
- Can eliminate conflict by declaring the token "else" to have higher precedence than "then" (and thus, than the first rule above).
- HOWEVER:** best to limit use of precedence to these standard examples (expressions, dangling elses). If you simply throw them in because you have a conflict you don't understand, you're like to end up with unexpected parse trees or syntax errors.

Reduce/Reduce Conflicts

- The lookahead symbols in LR(1) items are only considered for reductions in items that end in '•'.
- If a DFA state contains both $[X: \alpha \bullet, a]$ and $[Y: \beta \bullet, a]$ then on input 'a' we don't know which production to reduce.
- Such **reduce/reduce conflicts** are often due to a gross ambiguity in the grammar.
- Example: defining a sequence of identifiers with $S: \epsilon \mid id \mid id S$
- There are two parse trees for the string **id**: $S \Rightarrow id$ or $S \Rightarrow id S \Rightarrow id$.

Reduce/Reduce Conflicts in DFA

- For this example, you'll get states:

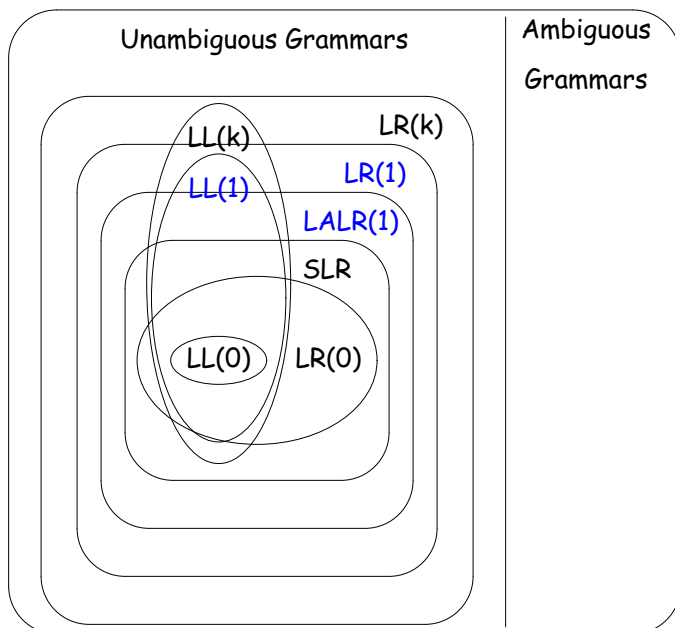


- Reduce/reduce conflict on input '↓'.
- Better rewrite the grammar: $S: \epsilon \mid id S$.

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.

A Hierarchy of Grammar Classes



Summary

- Parsing provides a means of tying translation actions to syntax clearly.
- A simple parser: LL(1), recursive descent
- A more powerful parser: LR(1)
- An efficiency hack: LALR(1), as in Bison.
- Earley's algorithm provides a complete algorithm for parsing all context-free languages.
- We can get the same effect in Bison by other means (the %glr-parser option, for Generalized LR), as seen in one of the examples from lecture #5.