Bottom-Up Parsing

Lecture 11-12
(From slides by G. Necula & R. Bodik)

Bottom-Up Parsing

- Bottom-up parsing is more general than top-down parsing
  - And just as efficient
  - Builds on ideas in top-down parsing
- 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:

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

  - 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:

$str \leftarrow$ input string of terminals

while $str \neq S$:

  - Identify first β in str such that $A \rightarrow \beta$ is a production and $S \rightarrow^* \alpha A \gamma \rightarrow \alpha \beta \gamma = str$
  - Replace β by A in str (so $\alpha A \gamma$ becomes new str)
- Such $\alpha \beta$'s are called *handles*

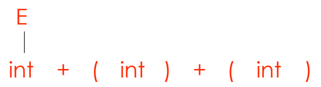
A Bottom-up Parse in Detail (1)

int + (int) + (int)

int + (int) + (int)

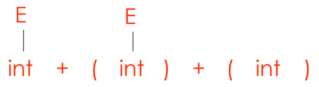
A Bottom-up Parse in Detail (2)

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



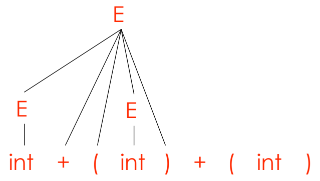
A Bottom-up Parse in Detail (3)

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



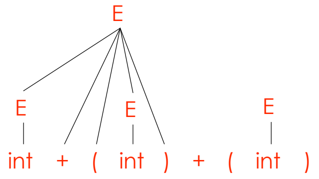
A Bottom-up Parse in Detail (4)

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



A Bottom-up Parse in Detail (5)

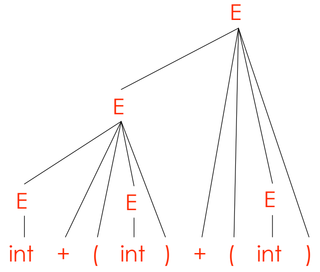
int + (int) + (int)
E + (int) + (int)
E + (E) + (int)
E + (int)
E + (E)



A Bottom-up Parse in Detail (6)

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

A reverse rightmost derivation



Where Do Reductions Happen

Because an LR parser produces a reverse rightmost derivation:

- If $\alpha\beta\gamma$ is step of a bottom-up parse with handle $\alpha\beta$
- And the next reduction is by $A \rightarrow \beta$
- Then γ is a string of terminals !

… Because $\alpha A\gamma \rightarrow \alpha\beta\gamma$ is a step in a right-most derivation

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

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
  - Marks end of next potential handle
- Initially, all input is unexamined: $|x_1x_2 \ldots x_n$

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

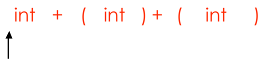
  E + (| int) ⇒ E + (int |)

  Reduce: Apply an inverse production at the handle. If E → E + (E) is a production, then

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

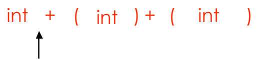
Shift-Reduce Example

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



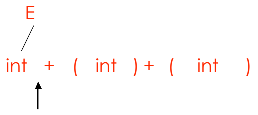
Shift-Reduce Example

| int + (int) + (int)$ shift
int | + (int) + (int)$ red. E → int



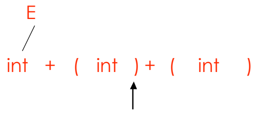
Shift-Reduce Example

| int + (int) + (int)$ shift
int | + (int) + (int)$ red. E → int
E | + (int) + (int)$ shift 3 times



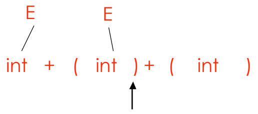
Shift-Reduce Example

| int + (int) + (int)$ shift
int | + (int) + (int)$ red. E → int
E | + (int) + (int)$ shift 3 times
E + (int |) + (int)$ red. E → int



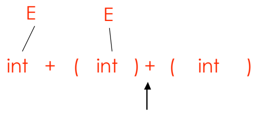
Shift-Reduce Example

| 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



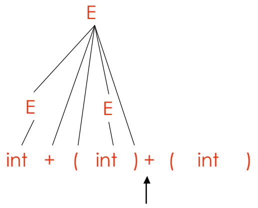
Shift-Reduce Example

| 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)



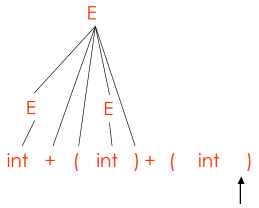
Shift-Reduce Example

| 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



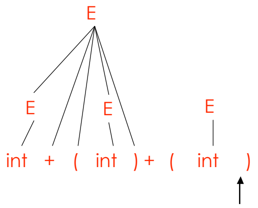
Shift-Reduce Example

| 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



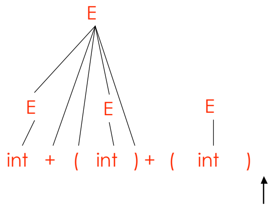
Shift-Reduce Example

| 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



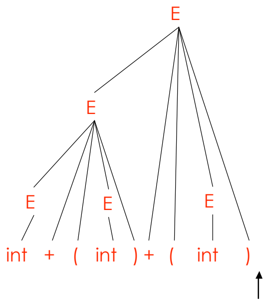
Shift-Reduce Example

| 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



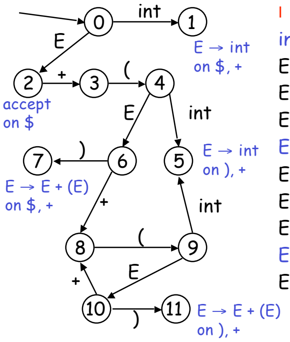
The Stack

- 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 up to potential handle
  - DFA alphabet consists of terminals and nonterminals
  - DFA *recognizes complete handles*
- 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. An Example



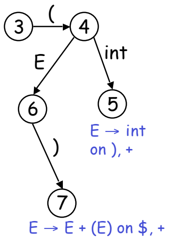
❘ int + (int) + (int)$	shift
int ❘ + (int) + (int)$	E → int
E ❘ + (int) + (int)$	shift(x3)
E + (int ❘) + (int)$	E → int
E + (E ❘) + (int)$	shift
E + (E) ❘ + (int)$	E → E+(E)
E ❘ + (int)$	shift (x3)
E + (int ❘)$	E → int
E + (E ❘)$	shift
E + (E) ❘ $	E → E+(E)
E ❘ $	accept

Representing the DFA

- Parsers represent the DFA as a 2D table
  - As for table-driven lexical analysis
- Lines correspond to DFA states
- Columns correspond to terminals and non-terminals
- In classical treatments, 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		$r_{E \to int}$		$r_{E \to int}$		
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
- So record, for each stack element, state of the DFA after that state
- LR parser maintains a stack

  ⟨ sym$_1$, state$_1$ ⟩ . . . ⟨ sym$_n$, state$_n$ ⟩

  state$_k$ is the final state of the DFA on sym$_1$ … sym$_k$

The LR Parsing Algorithm

```
Let I = w1w2…wn$ be initial input
Let j = 1
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 ⟩; j += 1
                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 programming languages grammars are LR
- Can be described as a simple table
- There are tools for building the table
- How is the table constructed?

To Be Done

- Review of bottom-up parsing
- Computing the parsing DFA
- Using parser generators

Bottom-up Parsing (Review)

- A bottom-up parser rewrites the input string to the start symbol
- The state of the parser is described as

$$\alpha \mid \gamma$$

  - α is a stack of terminals and non-terminals
  - γ is the string of terminals not yet examined
- Initially: $\mid x_1x_2 \ldots x_n$

The Shift and Reduce Actions (Review)

- Recall the CFG: E → int | E + (E)
- A bottom-up parser uses two kinds of actions:
- Shift pushes a terminal from input on the stack

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

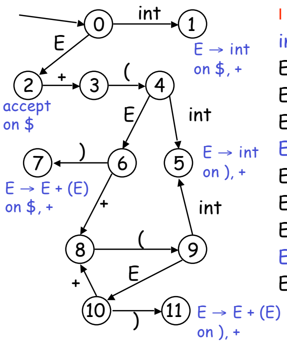
- Reduce pops 0 or more symbols from the stack (production rhs) and pushes a non-terminal on the stack (production lhs)

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

Key Issue: When to Shift or 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 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. An Example



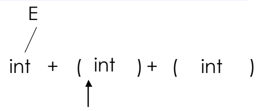
| int + (int) + (int)$	shift
int | + (int) + (int)$	E → int
E | + (int) + (int)$	shift(x3)
E + (int |) + (int)$	E → int
E + (E |) + (int)$	shift
E + (E) | + (int)$	E → E+(E)
E | + (int)$	shift (x3)
E + (int |)$	E → int
E + (E |)$	shift
E + (E) | $	E → E+(E)
E | $	accept

Key Issue: How is the DFA Constructed?

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

Parsing Contexts

- Consider the state:



  - The stack is E + (| int) + (int)
- Context:
  - We are looking for an E → E + (• E)
    - Have have seen E + (from the right-hand side
  - We are also looking for E → • int *or* E → • E + (E)
    - Have seen nothing from the right-hand side
- One DFA state describes several contexts

LR(1) Items

- An *LR(1) item* is a pair:

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

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

Note

- The symbol | was used before to separate the stack from the rest of input
  - α | γ, where α is the stack and γ is the remaining string of terminals
- In LR(1) items • is used to mark a prefix of a production rhs:

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

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

Convention

- We add to our grammar a fresh new start symbol S and a production S → E
  - Where E is the old start symbol
  - No need to do this if E had only one production
- The initial parsing context contains:

$$S \rightarrow \bullet E, \$$$

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

LR(1) Items (Cont.)

- In context containing

  E → E + • (E), +

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

    E → E + (• E), +

- In context containing

  E → E + (E) •, +

  - We can perform a *reduction* with E → E + (E)
  - But only if a + follows

LR(1) Items (Cont.)

- Consider a context with the item

  E → E + (• E) , +

- We expect next a string derived from E) +
- There are two productions for E

  E → int and E → E + (E)

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

  E → • int,)

  E → • E + (E) ,)

The Closure Operation

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

```
Closure(Items) =
  repeat
    for each [X → α•Yβ, a] in Items
      for each production Y → γ
        for each b ∈ First(βa)
          add [Y → •γ, b] to Items
  until Items is unchanged
```

Constructing the Parsing DFA (1)

- Construct the start context: Closure({S → •E, $})

  S → •E, $
  E → •E+(E), $
  E → •int, $
  E → •E+(E), +
  E → •int, +

- We abbreviate as:

  S → •E, $
  E → •E+(E), $/+
  E → •int, $/+

Constructing the Parsing DFA (2)

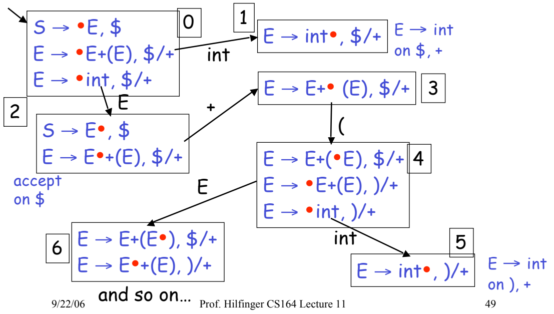
- A DFA state is a *closed* set of LR(1) items
  - This means that we performed Closure
- The start state is Closure([S → •E, $])
- A state that contains [X → α•, b] is labeled with "reduce with X → α on b"
- And now the transitions …

The DFA Transitions

- A state "State" that contains [X → α•yβ, 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] ∈ State
    add [X → αy•β, b] to Items
  return Closure(Items)
```

Constructing the Parsing DFA. Example.



0
S → •E, $
E → •E+(E), $/+
E → •int, $/+

int

1
E → int•, $/+
E → int on $, +

E

2
S → E•, $
E → E•+(E), $/+
accept on $

+

3
E → E+• (E), $/+

(

4
E → E+(•E), $/+
E → •E+(E),)/+
E → •int,)/+

E

6
E → E+(E•), $/+
E → E•+(E),)/+

int

5
E → int•,)/+
E → int on), +

and so on…

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 → α•aβ, b] and [Y → γ•, a]

- Then on input "a" we could either
  - Shift into state [X → αa•β, b], or
  - Reduce with Y → γ
- This is called a *shift-reduce conflict*

Shift/Reduce Conflicts

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

  S → if E then S | if E then S else S | OTHER

- Will have DFA state containing

  [S → if E then S•, else]

  [S → if E then S• else S, $]

- If else follows then we can shift or reduce

More Shift/Reduce Conflicts

- Consider the ambiguous grammar

  E → E + E | E * E | int

- We will have the states containing

  [E → E * • E, +] [E → E * E•, +]

  [E → • E + E, +] $\Rightarrow^{E}$ [E → E• + E, +]

  … …

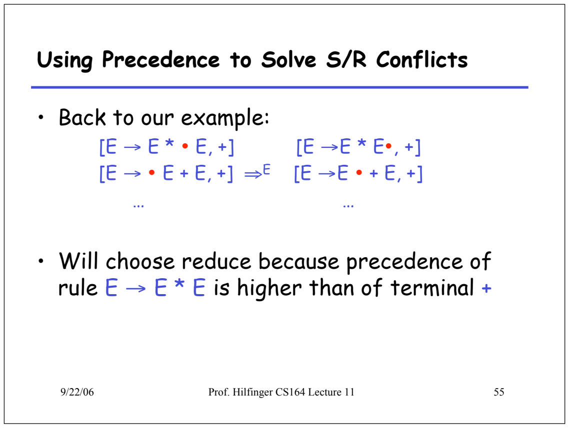
- Again we have a shift/reduce on input +
  - We need to reduce (* binds more tightly than +)
  - Solution: declare the precedence of * and +

More Shift/Reduce Conflicts

- In bison declare precedence and associativity of *terminal symbols*:

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

- Precedence *of a rule* = that of its last terminal
  - See bison manual for ways to override this default
- Resolve shift/reduce conflict with a *shift* if:
  - input terminal has higher precedence than the rule
  - the precedences are the same and right associative

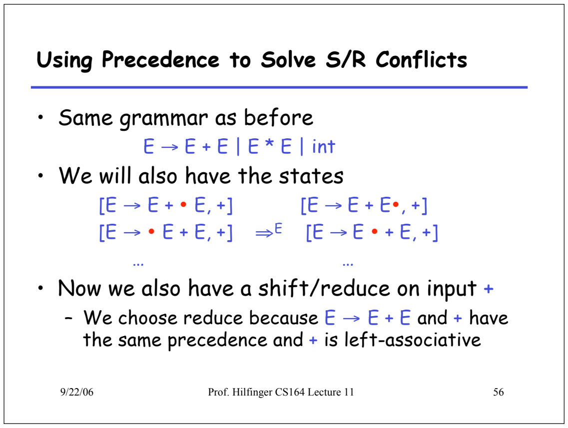


Using Precedence to Solve S/R Conflicts

- Back to our example:

$[E \to E * \bullet E, +]$ $[E \to E * E\bullet, +]$
$[E \to \bullet E + E, +]$ $\Rightarrow^{E}$ $[E \to E \bullet + E, +]$
… …

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



Using Precedence to Solve S/R Conflicts

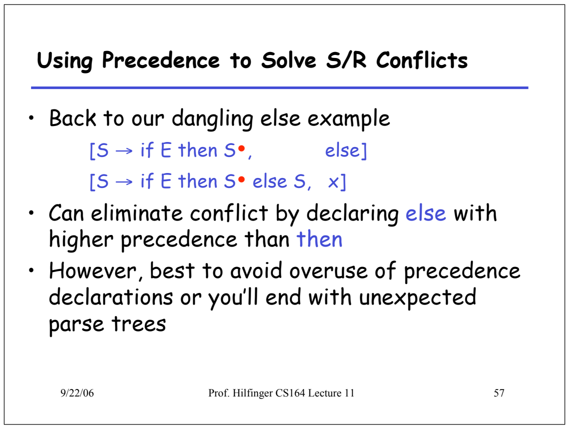
- Same grammar as before

$E \to E + E \mid E * E \mid int$

- We will also have the states

$[E \to E + \bullet E, +]$ $[E \to E + E\bullet, +]$
$[E \to \bullet E + E, +]$ $\Rightarrow^{E}$ $[E \to E \bullet + E, +]$
… …

- 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

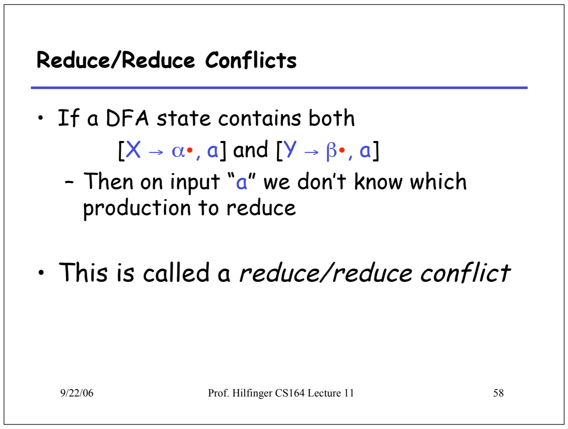


Using Precedence to Solve S/R Conflicts

- Back to our dangling else example

$[S \to$ if E then S$\bullet$, else$]$
$[S \to$ if E then S$\bullet$ else S, x$]$

- Can eliminate conflict by declaring else with higher precedence than then
- However, best to avoid overuse of precedence declarations or you'll end with unexpected parse trees



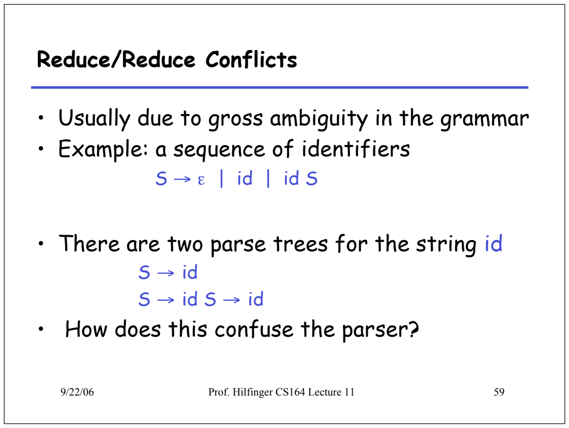
Reduce/Reduce Conflicts

- If a DFA state contains both

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

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

- This is called a *reduce/reduce conflict*



Reduce/Reduce Conflicts

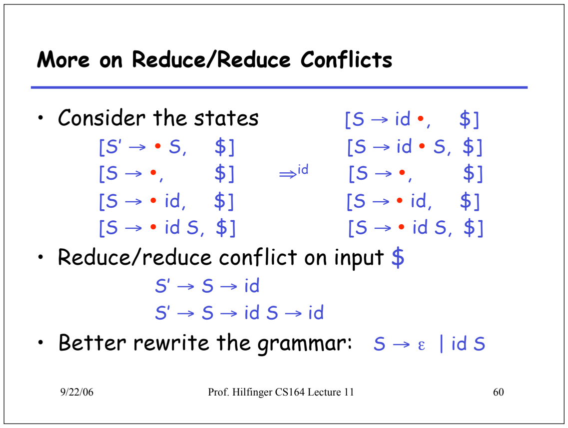
- Usually due to gross ambiguity in the grammar
- Example: a sequence of identifiers

$S \to \varepsilon \mid id \mid id\ S$

- There are two parse trees for the string id

$S \to id$
$S \to id\ S \to id$

- How does this confuse the parser?



More on Reduce/Reduce Conflicts

- Consider the states

$[S' \to \bullet S, \$]$
$[S \to \bullet, \$]$
$[S \to \bullet id, \$]$
$[S \to \bullet id\ S, \$]$

$\Rightarrow^{id}$

$[S \to id \bullet, \$]$
$[S \to id \bullet S, \$]$
$[S \to \bullet, \$]$
$[S \to \bullet id, \$]$
$[S \to \bullet id\ S, \$]$

- Reduce/reduce conflict on input $

$S' \to S \to id$
$S' \to S \to id\ S \to id$

- Better rewrite the grammar: $S \to \varepsilon \mid id\ S$

Using Parser Generators

- Parser generators 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.

1
E → int•, $/+ E → int on $, + and
5
E → int•,)/+ E → int on), +

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

1'
E → int•, $/+/) E → int on $, +,)

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

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

is

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

LALR States

- Consider for example the LR(1) states

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

$\{[X \to \alpha\bullet, b], [Y \to \beta\bullet, d]\}$

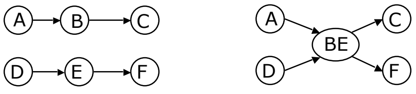
- They have the same core and can be merged
- And the merged state contains:

$\{[X \to \alpha\bullet, a/b], [Y \to \beta\bullet, c/d]\}$

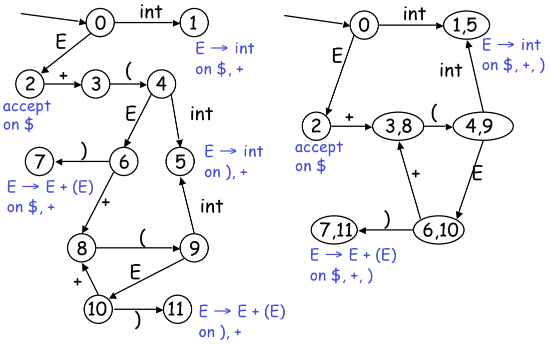
- These are called LALR(1) states
  - Stands for LookAhead LR
  - Typically 10 times fewer LALR(1) states than LR(1)

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 → α•, a], [Y → β•, b]}

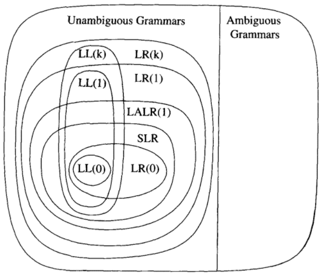
  {[X → α•, b], [Y → β•, a]}
- And the merged LALR(1) state

  {[X → α•, a/b], [Y → β•, a/b]}
- Has a new reduce-reduce conflict
- In practice such cases are rare

LALR vs. LR Parsing

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

A Hierarchy of Grammar Classes



From Andrew Appel, "Modern Compiler Implementation in Java"

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)
  - We use LALR(1) parser generators