

LR Parsing. Parser Generators.

Lecture 7-8

Prof. Necula CS 164 Lecture 7-8

1

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!

Prof. Necula CS 164 Lecture 7-8

2

An Introductory Example

- LR parsers don't need left-factored grammars and can also handle left-recursive grammars

- Consider the following grammar:

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

- Why is this not LL(1)?

- Consider the string: `int + (int) + (int)`

Prof. Necula CS 164 Lecture 7-8

3

The Idea

- LR parsing *reduces* a string to the start symbol by inverting productions:

`str` ← input string of terminals

repeat

- Identify β in `str` such that $A \rightarrow \beta$ is a production (i.e., `str = $\alpha \beta \gamma$`)
- Replace β by A in `str` (i.e., `str` becomes `$\alpha A \gamma$`)

until `str = S`

Prof. Necula CS 164 Lecture 7-8

4

A Bottom-up Parse in Detail (1)

`int + (int) + (int)`

`int + (int) + (int)`

Prof. Necula CS 164 Lecture 7-8

5

A Bottom-up Parse in Detail (2)

`int + (int) + (int)`

`E + (int) + (int)`

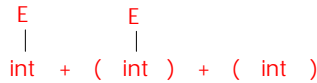
`E`
|
`int + (int) + (int)`

Prof. Necula CS 164 Lecture 7-8

6

A Bottom-up Parse in Detail (3)

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

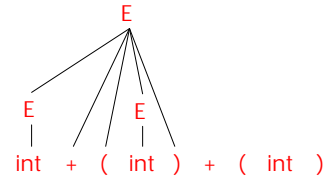


Prof. Necula CS 164 Lecture 7-8

7

A Bottom-up Parse in Detail (4)

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

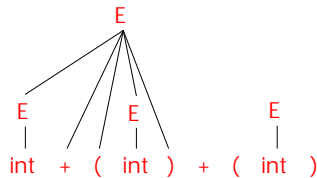


Prof. Necula CS 164 Lecture 7-8

8

A Bottom-up Parse in Detail (5)

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



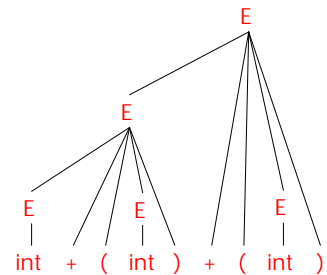
Prof. Necula CS 164 Lecture 7-8

9

A Bottom-up Parse in Detail (6)

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

A rightmost derivation in reverse



Prof. Necula CS 164 Lecture 7-8

10

Important Fact #1

Important Fact #1 about bottom-up parsing:

An LR parser traces a rightmost derivation in reverse

Prof. Necula CS 164 Lecture 7-8

11

Where Do Reductions Happen

Important Fact #1 has an interesting consequence:

- Let $\alpha\beta\gamma$ be a step of a bottom-up parse
- Assume the next reduction is by $A \rightarrow \beta$
- Then γ is a string of terminals!

Why? Because $\alpha A \gamma \rightarrow \alpha \beta \gamma$ is a step in a rightmost derivation

Prof. Necula CS 164 Lecture 7-8

12

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 \blacktriangleright
 - The \blacktriangleright is not part of the string
- Initially, all input is unexamined: $\blacktriangleright X_1 X_2 \dots X_n$

Prof. Necula CS 164 Lecture 7-8

13

Shift-Reduce Parsing

- Bottom-up parsing uses only two kinds of actions:

Shift

Reduce

Prof. Necula CS 164 Lecture 7-8

14

Shift

- Shift:* Move \blacktriangleright one place to the right
- Shifts a terminal to the left string

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

Prof. Necula CS 164 Lecture 7-8

15

Reduce

- Reduce:* Apply an inverse production at the right end of the left string
- If $E \rightarrow E + (E)$ is a production, then

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

Prof. Necula CS 164 Lecture 7-8

16

Shift-Reduce Example

$\blacktriangleright \text{int} + (\text{int}) + (\text{int})\$$ shift

int + (int) + (int)
↑

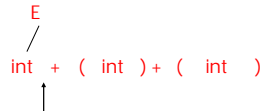
Shift-Reduce Example

$\blacktriangleright \text{int} + (\text{int}) + (\text{int})\$$ shift
int \blacktriangleright + (int) + (int)\$ red. $E \rightarrow \text{int}$

int + (int) + (int)
↑

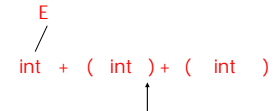
Shift-Reduce Example

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



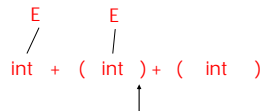
Shift-Reduce Example

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



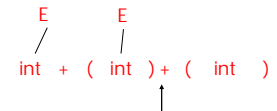
Shift-Reduce Example

▶ 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



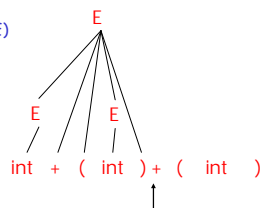
Shift-Reduce Example

▶ 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)



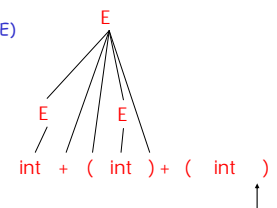
Shift-Reduce Example

▶ 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

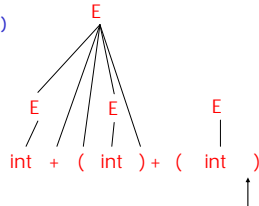


Shift-Reduce Example

```

▶ 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
E + (E) ▶$ red. E → E + (E)

```

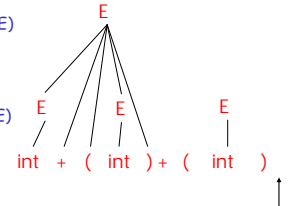


Shift-Reduce Example

```

▶ 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)

```

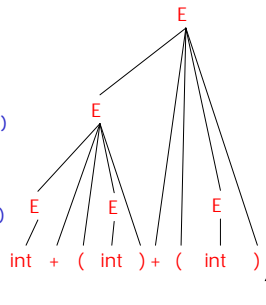


Shift-Reduce Example

```

▶ 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)

Prof. Neucula CS 164 Lecture 7-8

28

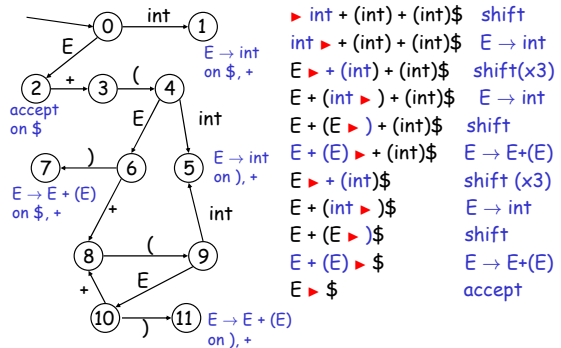
Key Issue: When to Shift or Reduce?

- 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
 - DFA language consists of terminals and nonterminals
- 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 " $A \rightarrow \beta$ on tok " then reduce

Prof. Neucula CS 164 Lecture 7-8

29

LR(1) Parsing. An Example



Representing the DFA

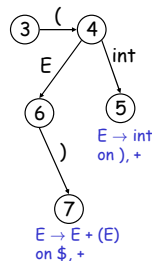
- Parsers represent the DFA as a 2D table
 - Recall table-driven lexical analysis
- Lines correspond to DFA states
- Columns correspond to terminals and non-terminals
- Typically columns are split into:
 - Those for terminals: action table
 - Those for non-terminals: goto table

Prof. Necula CS 164 Lecture 7-8

31

Representing the DFA. Example

- The table for a fragment of our DFA:



	int	+	()	\$	E
...						
3			s4			
4	s5					g6
5	rE→int			rE→int		
6	s8		s7			
7	rE→E+(E)					rE→E+(E)
...						

Prof. Necula CS 164 Lecture 7-8

32

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 to which state it brings the DFA
- LR parser maintains a stack
 - $\langle \text{sym}_1, \text{state}_1 \rangle \dots \langle \text{sym}_n, \text{state}_n \rangle$
 - state_k is the final state of the DFA on $\text{sym}_1 \dots \text{sym}_k$

Prof. Necula CS 164 Lecture 7-8

33

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 → α:
    pop |α| pairs,
    push (X, Goto[top_state(stack), X])
  accept: halt normally
  error: halt and report error
  
```

Prof. Necula CS 164 Lecture 7-8

34

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?

Prof. Necula CS 164 Lecture 7-8

35

Outline

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

Prof. Necula CS 164 Lecture 7-8

36

Bottom-up Parsing (Review)

- A bottom-up parser rewrites the input string to the start symbol
- The state of the parser is described as $\alpha \triangleright \gamma$
 - α is a stack of terminals and non-terminals
 - γ is the string of terminals not yet examined
- Initially: $\triangleright x_1 x_2 \dots x_n$

Prof. Necula CS 164 Lecture 7-8

37

The Shift and Reduce Actions (Review)

- Recall the CFG: $E \rightarrow \text{int} \mid E + (E)$
- A bottom-up parser uses two kinds of actions:
 - Shift pushes a terminal from input on the stack
 $E + (\triangleright \text{int}) \Rightarrow E + (\text{int} \triangleright)$
 - Reduce pops 0 or more symbols from the stack (production rhs) and pushes a non-terminal on the stack (production lhs)
 $E + (E + (E) \triangleright) \Rightarrow E + (E \triangleright)$

Prof. Necula CS 164 Lecture 7-8

38

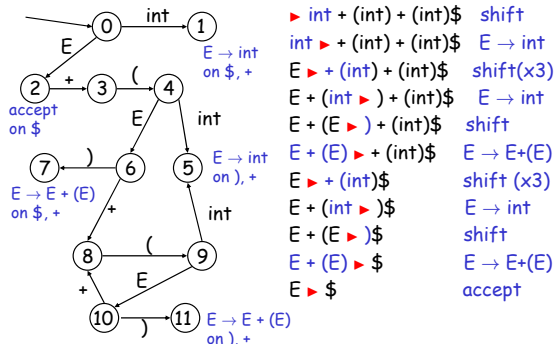
Key Issue: When to Shift or Reduce?

- 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 \triangleright
 - If X has a transition labeled tok then shift
 - If X is labeled with " $A \rightarrow \beta$ on tok " then reduce

Prof. Necula CS 164 Lecture 7-8

39

LR(1) Parsing. An Example



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

End of review

Prof. Necula CS 164 Lecture 7-8

41

Prof. Necula CS 164 Lecture 7-8

42

Parsing Contexts

- Consider the state:

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

- The stack is $E + (\blacktriangleright \text{int}) + (\text{int})$
- Context:
 - We are looking for an $E \rightarrow E + (\bullet E)$
 - Have have seen $E + ($ from the right-hand side
 - We are also looking for $E \rightarrow \bullet \text{int}$ or $E \rightarrow \bullet E + (E)$
 - Have seen nothing from the right-hand side
- One DFA state describes several contexts

Prof. Necula CS 164 Lecture 7-8

43

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

Prof. Necula CS 164 Lecture 7-8

44

Note

- The symbol \blacktriangleright was used before to separate the stack from the rest of input
 - $\alpha \blacktriangleright \gamma$, where α is the stack and γ is the remaining string of terminals
- In LR(1) items \bullet 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

Prof. Necula CS 164 Lecture 7-8

45

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
 - 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

Prof. Necula CS 164 Lecture 7-8

46

LR(1) Items (Cont.)

- In context containing
 - $E \rightarrow E + \bullet (E), +$
 - If $($ follows then we can perform a shift to context containing
 - $E \rightarrow E + (\bullet E), +$
- In context containing
 - $E \rightarrow E + (E) \bullet, +$
 - We can perform a reduction with $E \rightarrow E + (E)$
 - But only if a $+$ follows

Prof. Necula CS 164 Lecture 7-8

47

LR(1) Items (Cont.)

- Consider a context with the item
 - $E \rightarrow E + (\bullet E), +$
- We expect next a string derived from $E) +$
- There are two productions for E
 - $E \rightarrow \text{int}$ and $E \rightarrow E + (E)$
- We describe this by extending the context with two more items:
 - $E \rightarrow \bullet \text{int},)$
 - $E \rightarrow \bullet E + (E),)$

Prof. Necula CS 164 Lecture 7-8

48

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
```

Prof. Nečula CS 164 Lecture 7-8

49

Constructing the Parsing DFA (1)

- Construct the start context: $\text{Closure}(\{S \rightarrow \bullet E, \$\})$

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

- We abbreviate as:

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

Prof. Nečula CS 164 Lecture 7-8

50

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 $\text{Closure}([S \rightarrow \bullet E, \$])$
- A state that contains $[X \rightarrow \alpha \bullet, b]$ is labeled with "reduce with $X \rightarrow \alpha$ on b "
- And now the transitions ...

Prof. Nečula CS 164 Lecture 7-8

51

The DFA Transitions

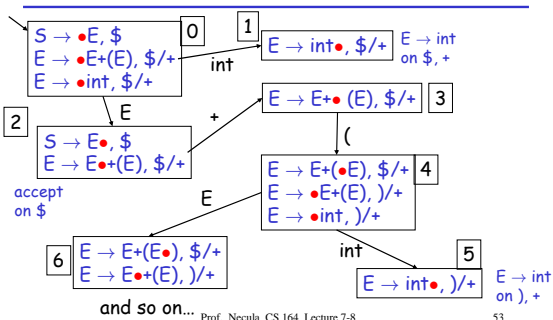
- A state "State" that contains $[X \rightarrow \alpha \bullet y \beta, b]$ has a transition labeled y to a state that contains the items "Transition(State, y)"
 - y can be a terminal or a non-terminal

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

Prof. Nečula CS 164 Lecture 7-8

52

Constructing the Parsing DFA. Example.



Prof. Nečula CS 164 Lecture 7-8

53

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?

Prof. Nečula CS 164 Lecture 7-8

54

Using Precedence to Solve S/R Conflicts

- Back to our dangling else example
 - $[S \rightarrow \text{if } E \text{ then } S \bullet, \quad \text{else}]$
 - $[S \rightarrow \text{if } E \text{ then } S \bullet \text{ else } S, \quad x]$
- Can eliminate conflict by declaring *else* with higher precedence than *then*
 - Or just rely on the default shift action
- But this starts to look like "hacking the parser"
- Best to avoid overuse of precedence declarations or you'll end with unexpected parse trees

Prof. Neucula CS 164 Lecture 7-8

61

Reduce/Reduce Conflicts

- If a DFA state contains both
 - $[X \rightarrow \alpha \bullet, a]$ and $[Y \rightarrow \beta \bullet, a]$
 - Then on input "a" we don't know which production to reduce
- This is called a reduce/reduce conflict

Prof. Neucula CS 164 Lecture 7-8

62

Reduce/Reduce Conflicts

- Usually due to gross ambiguity in the grammar
- Example: a sequence of identifiers
 - $S \rightarrow \epsilon \mid \text{id} \mid \text{id } S$
- There are two parse trees for the string *id*
 - $S \rightarrow \text{id}$
 - $S \rightarrow \text{id } S \rightarrow \text{id}$
- How does this confuse the parser?

Prof. Neucula CS 164 Lecture 7-8

63

More on Reduce/Reduce Conflicts

- Consider the states

$[S' \rightarrow \bullet S, \$]$	\Rightarrow^{id}	$[S \rightarrow \text{id} \bullet, \$]$
$[S \rightarrow \bullet, \$]$		$[S \rightarrow \bullet, \$]$
$[S \rightarrow \bullet \text{id}, \$]$		$[S \rightarrow \bullet \text{id}, \$]$
$[S \rightarrow \bullet \text{id } S, \$]$		$[S \rightarrow \bullet \text{id } S, \$]$
- Reduce/reduce conflict on input \$
 - $S' \rightarrow S \rightarrow \text{id}$
 - $S' \rightarrow S \rightarrow \text{id } S \rightarrow \text{id}$
- Better rewrite the grammar: $S \rightarrow \epsilon \mid \text{id } S$

Prof. Neucula CS 164 Lecture 7-8

64

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

Prof. Neucula CS 164 Lecture 7-8

65

LR(1) Parsing Tables are Big

- But many states are similar, e.g.

$E \rightarrow \text{int} \bullet, \$ / +$	$E \rightarrow \text{int}$ on \$, +	and	$E \rightarrow \text{int} \bullet,) / +$	$E \rightarrow \text{int}$ 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

$E \rightarrow \text{int} \bullet, \$ / + /)$	$E \rightarrow \text{int}$ on \$, +,)
--	---

Prof. Neucula CS 164 Lecture 7-8

66

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

Prof. Necula CS 164 Lecture 7-8

67

LALR States

- Consider for example the LR(1) states $\{ [X \rightarrow \alpha \bullet, a], [Y \rightarrow \beta \bullet, c] \}$ and $\{ [X \rightarrow \alpha \bullet, b], [Y \rightarrow \beta \bullet, d] \}$
- They have the same core and can be merged
- And the merged state contains: $\{ [X \rightarrow \alpha \bullet, a/b], [Y \rightarrow \beta \bullet, c/d] \}$
- These are called LALR(1) states
 - Stands for LookAhead LR
 - Typically 10 times fewer LALR(1) states than LR(1)

Prof. Necula CS 164 Lecture 7-8

68

A LALR(1) DFA

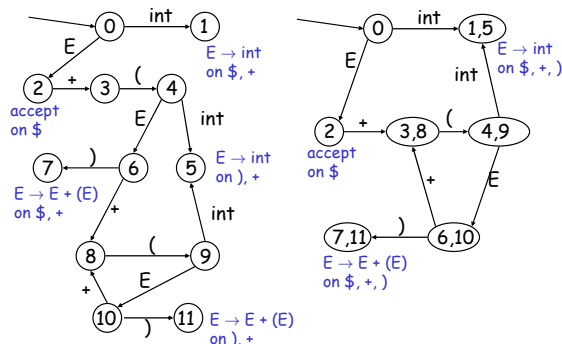
- Repeat until all states have distinct core
 - Choose two distinct states with same core
 - Merge the states by creating a new one with the union of all the items
 - Point edges from predecessors to new state
 - New state points to all the previous successors



Prof. Necula CS 164 Lecture 7-8

69

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



The LALR Parser Can Have Conflicts

- Consider for example the LR(1) states $\{ [X \rightarrow \alpha \bullet, a], [Y \rightarrow \beta \bullet, b] \}$ and $\{ [X \rightarrow \alpha \bullet, b], [Y \rightarrow \beta \bullet, a] \}$
- And the merged LALR(1) state $\{ [X \rightarrow \alpha \bullet, a/b], [Y \rightarrow \beta \bullet, a/b] \}$
- Has a new reduce-reduce conflict
- In practice such cases are rare

Prof. Necula CS 164 Lecture 7-8

71

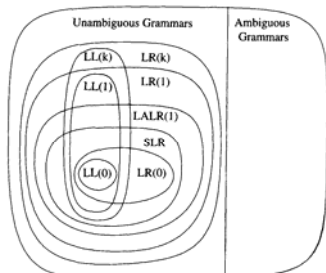
LALR vs. LR Parsing

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

Prof. Necula CS 164 Lecture 7-8

72

A Hierarchy of Grammar Classes



From Andrew Appel, "Modern Compiler Implementation in Java"

Prof. Necula CS 164 Lecture 7-8

73

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
- Now we move on to semantic analysis

Prof. Necula CS 164 Lecture 7-8

74

Supplement to LR Parsing

Strange Reduce/Reduce Conflicts
Due to LALR Conversion
(from the bison manual)

Prof. Necula CS 164 Lecture 7-8

75

Strange Reduce/Reduce Conflicts

- Consider the grammar

$$\begin{aligned} S &\rightarrow PR, & NL &\rightarrow N \mid N, NL \\ P &\rightarrow T \mid NL : T & R &\rightarrow T \mid N : T \\ N &\rightarrow id & T &\rightarrow id \end{aligned}$$
- **P** - parameters specification
- **R** - result specification
- **N** - a parameter or result name
- **T** - a type name
- **NL** - a list of names

Prof. Necula CS 164 Lecture 7-8

76

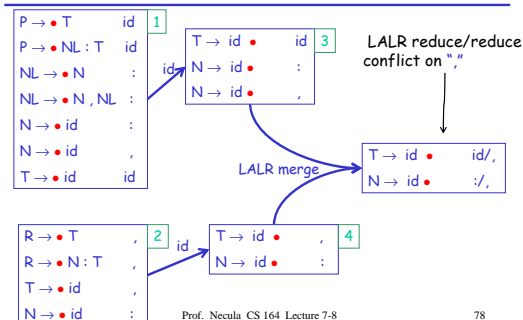
Strange Reduce/Reduce Conflicts

- In **P** an **id** is a
 - **N** when followed by **,** or **:**
 - **T** when followed by **id**
- In **R** an **id** is a
 - **N** when followed by **:**
 - **T** when followed by **,**
- This is an LR(1) grammar.
- But it is not LALR(1). Why?
 - For obscure reasons

Prof. Necula CS 164 Lecture 7-8

77

A Few LR(1) States



Prof. Necula CS 164 Lecture 7-8

78

What Happened?

- Two distinct states were confused because they have the same core
- Fix: add dummy productions to distinguish the two confused states
- E.g., add

$R \rightarrow id\ bogus$

- *bogus* is a terminal not used by the lexer
- This production will never be used during parsing
- But it distinguishes *R* from *P*

Prof. Necula CS 164 Lecture 7-8

79

A Few LR(1) States After Fix

