

# CSE P 501 – Compilers

LR Parsing  
Hal Perkins  
Autumn 2025

# Agenda

- LR Parsing
- Table-driven Parsers
- Parser States
- Shift-Reduce and Reduce-Reduce conflicts

# Bottom-Up Parsing

- Idea: Read the input left to right
- Whenever we've matched the right hand side of a production, reduce it to the appropriate non-terminal and add that non-terminal to the parse tree
- The upper edge of this partial parse tree is known as the *frontier*

# Example

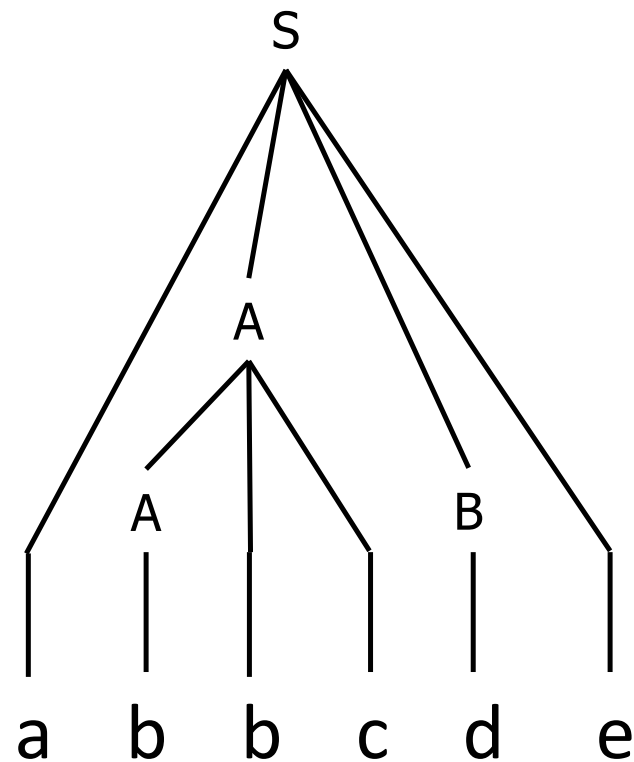
- Grammar

$S ::= aABe$

$A ::= Abc \mid b$

$B ::= d$

- Bottom-up Parse



# LR(1) Parsing

- We'll look at LR(1) parsers
  - Left to right scan, Rightmost derivation, 1 symbol lookahead
  - Almost all practical programming languages have a LR(1) grammar
  - LALR(1), SLR(1), etc. – subsets of LR(1)
    - LALR(1) can parse most real programming languages, tables are more compact, and is used by YACC/Bison/CUP/etc.

# LR Parsing in Greek

- The bottom-up parser reconstructs a reverse rightmost derivation
- Given the rightmost derivation
$$S \Rightarrow \beta_1 \Rightarrow \beta_2 \Rightarrow \dots \Rightarrow \beta_{n-2} \Rightarrow \beta_{n-1} \Rightarrow \beta_n = w$$
the parser will first discover  $\beta_{n-1} \Rightarrow \beta_n$ , then  $\beta_{n-2} \Rightarrow \beta_{n-1}$ , etc.
- Parsing terminates when
  - $\beta_1$  reduced to  $S$  (start symbol, success), or
  - No match can be found (syntax error)

# How Do We Parse with This?

- Key: given what we've already seen and the next input symbol (the lookahead), decide what to do.
- Choices:
  - Shift: Advance 1 token further in the input
  - Reduce: Perform a reduction
- Can reduce  $A \Rightarrow \beta$  if both of these hold:
  - $A \Rightarrow \beta$  is a valid production, *and*
  - $A \Rightarrow \beta$  is a step in *this* rightmost derivation that produced *this* input string
- This is known as a *shift-reduce* parser

# Sentential Forms

- If  $S \Rightarrow^* \alpha$ , the string  $\alpha$  is called a *sentential form* of the grammar
- In the derivation  
$$S \Rightarrow \beta_1 \Rightarrow \beta_2 \Rightarrow \dots \Rightarrow \beta_{n-2} \Rightarrow \beta_{n-1} \Rightarrow \beta_n = w$$
each of the  $\beta_i$  are sentential forms
- A sentential form in a rightmost derivation is called a right-sentential form (similarly for leftmost and left-sentential)



# Handles

- Informally, a substring of the tree frontier that matches the right side  $\beta$  of a production *that is part of the rightmost derivation of the current input string*
  - Even if  $A ::= \beta$  is a production, it is a handle only if  $\beta$  matches the parse tree frontier at a point where  $A ::= \beta$  was used in *this specific* derivation
  - $\beta$  may appear in many other places in the frontier without being the rhs of a handle for that particular production
- Bottom-up parsing is all about finding handles

# Handle Examples

- In the derivation
$$S \Rightarrow aABe \Rightarrow aAde \Rightarrow aAbcde \Rightarrow abbcde$$
  - a**b**cde is a right sentential form whose handle is **A::=b** at position 2
  - a**A**bcde is a right sentential form whose handle is **A::=Abc** at position 4
    - Note: some books take the left end of the match as the position

# Handles – The Dragon Book Defn.

- Formally, a *handle* of a right-sentential form  $\gamma$  is a production  $A ::= \beta$  and a position in  $\gamma$  where  $\beta$  may be replaced by  $A$  to produce the previous right-sentential form in the rightmost derivation of  $\gamma$ 
  - Some sources use “handle” to refer only to the right-hand side  $\beta$  and its position. Others mean the entire production  $A ::= \beta$ . Which one should be clear from context.

# Implementing Shift-Reduce Parsers

- Key Data structures
  - A stack holding the frontier of the tree
  - A string with the remaining input
- We also need to encode the rules that tell us what action to take next, given (a) the state of the stack and (b) the lookahead symbol
  - Typically a table that encodes a finite automata

# Shift-Reduce Parser Operations

- *Shift* – push the next input symbol onto the stack
- *Reduce* – if the top of the stack is the right side of a handle  $A ::= \beta$ , pop the right side  $\beta$  and push the left side  $A$
- *Accept* – announce success
- *Error* – syntax error discovered

# Shift-Reduce Example

$S ::= aABe$   
 $A ::= Abc \mid b$   
 $B ::= d$

Stack	Input	Action
\$	abbcde\$	<i>shift</i>
\$a	bbcde\$	<i>shift</i>
\$ab	bcde\$	<i>reduce</i>
\$aA	bcde\$	<i>shift</i>
\$aAb	cde\$	<i>shift</i>
\$aAbc	de\$	<i>reduce</i>
\$aA	de\$	<i>shift</i>
\$aAd	e\$	<i>reduce</i>
\$aAB	e\$	<i>shift</i>
\$aABe	\$	<i>reduce</i>
\$S	\$	<i>accept</i>

# How Do We Automate This?

- Cannot use clairvoyance in a real parser (alas...)
- Defn. *Viable prefix* – a prefix of a right-sentential form that can appear on the stack of the shift-reduce parser
  - Equivalent: a prefix of a right-sentential form that does not continue past the rightmost handle of that sentential form
  - In Greek:  $\gamma$  is a *viable prefix* of  $G$  if there is some derivation  $S \Rightarrow_{rm}^* \alpha A w \Rightarrow_{rm} \alpha \beta w$  and  $\gamma$  is a prefix of  $\alpha \beta$ .
  - The occurrence of  $\beta$  in  $\alpha \beta w$  is the right side of a *handle* of  $\alpha \beta w$

# How Do We Automate This?

- Fact: the set of viable prefixes of a CFG is a regular language(!)
- Idea: Construct a DFA to recognize viable prefixes given the stack and remaining input
  - Perform reductions when we recognize the rhs of handles

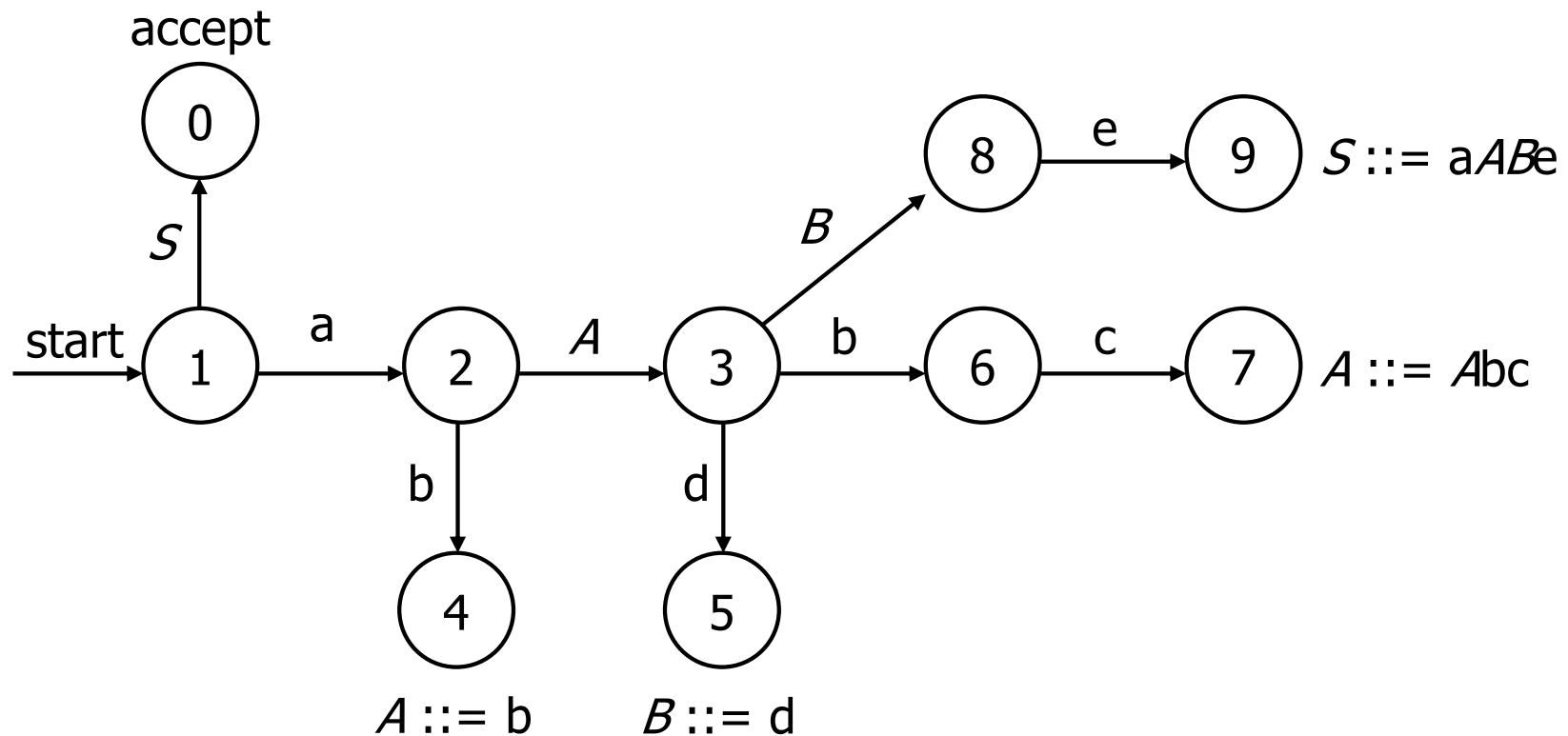


# DFA for prefixes of

$S ::= aABe$

$A ::= Abc \mid b$

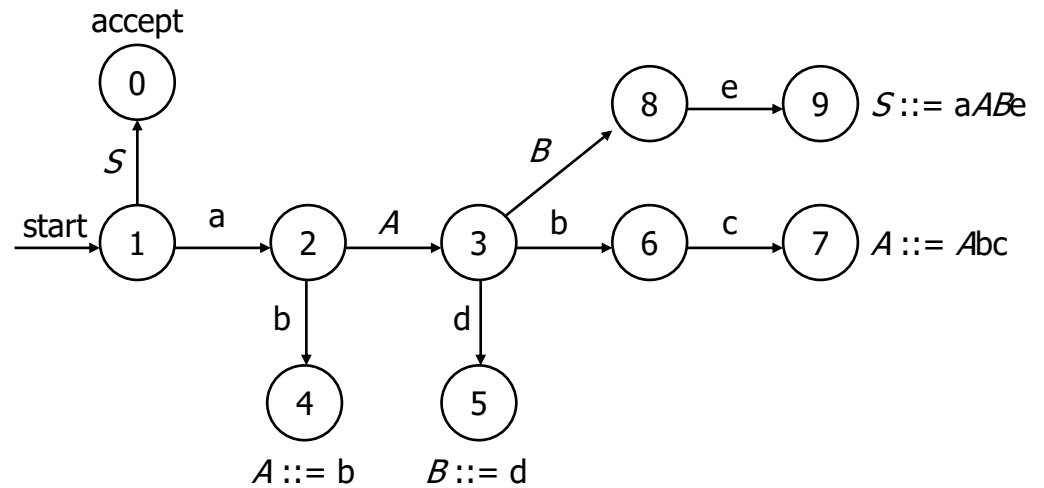
$B ::= d$



# Trace

$S ::= aABe$   
 $A ::= Abc \mid b$   
 $B ::= d$

Stack	Input
\$	abbcde\$
\$a	bbcde\$
\$ab	bcde\$
\$aA	bcde\$
\$aAb	cde\$
\$aAbc	de\$
\$aA	de\$
\$aAd	e\$
\$aAB	e\$
\$aABe	\$
\$S	\$



# Observations

- Way too much backtracking
  - We want the parser to run in time proportional to the length of the input
- Where the heck did this DFA come from anyway?
  - From the underlying grammar
  - Defer construction details for now

# Avoiding DFA Rescanning

- Observation: no need to restart DFA after a shift. Stay in the same state and process next token.
- Observation: after a reduction, the contents of the stack are the same as before except for the new non-terminal on top that replaced the production rhs
  - $\therefore$  Scanning the stack will take us through the same transitions as before until the last one
  - $\therefore$  If we record state numbers on the stack, we can back up directly to the appropriate state when we pop the right hand side of a production from the stack

# Stack

- Change the stack to contain pairs of states and symbols from the grammar

$\$s_0 X_1 s_1 X_2 s_2 \dots X_n s_n$

- State  $s_0$  is the start state
  - When we push a symbol on the stack, push the symbol plus the new parser DFA state that we reach
  - When we reduce, popping the handle will reveal the state of the DFA just prior to reading the handle
- Observation: in an actual parser, only the state numbers are needed since they implicitly contain the symbol information, but for explanations / examples it can help to show both.

# Encoding the DFA in a Table

- A shift-reduce parser's DFA can be encoded in two tables
  - One row for each state
  - *action* table encodes what to do given the current state and the next input symbol
  - *goto* table encodes the transitions to take when we back up into a state after a reduction and then make a transition using the newly pushed (reduced) non-terminal

# Actions (1)

- Given the current state and input symbol, the main possible actions are
  - $si$  – shift the input symbol and state  $i$  onto the stack (i.e., shift and move to state  $i$ )
  - $rj$  – reduce using grammar production  $j$ 
    - The production number tells us how many  $\langle \text{symbol}, \text{state} \rangle$  pairs to pop off the stack (= length of RHS of production), and the LHS nonterminal to push
    - Each production needs a unique number, i.e.,  $A ::= \alpha \mid \beta$  needs to be split into  $A ::= \alpha$  and  $A ::= \beta$

# Actions (2)

- Other possible *action* table entries
  - *accept*
  - **blank** – no transition – syntax error
    - A LR parser will detect an error as soon as possible on a left-to-right scan
    - A real compiler needs to produce an error message, recover, and continue parsing when this happens
      - (Often involves encoding error handling/recovery info in the action table)



# Goto

- When a reduction is performed using  $A ::= \beta$ , we pop  $|\beta|$   $\langle \text{symbol}, \text{state} \rangle$  pairs from the stack revealing a state *uncovered\_s* on the top of the stack
- $\text{goto}[\text{uncovered\_s}, A]$  is the new state to push on the stack when reducing production  $A ::= \beta$  (after popping handle  $\beta$  and pushing  $A$ )

## Aside: Extra Initial Production

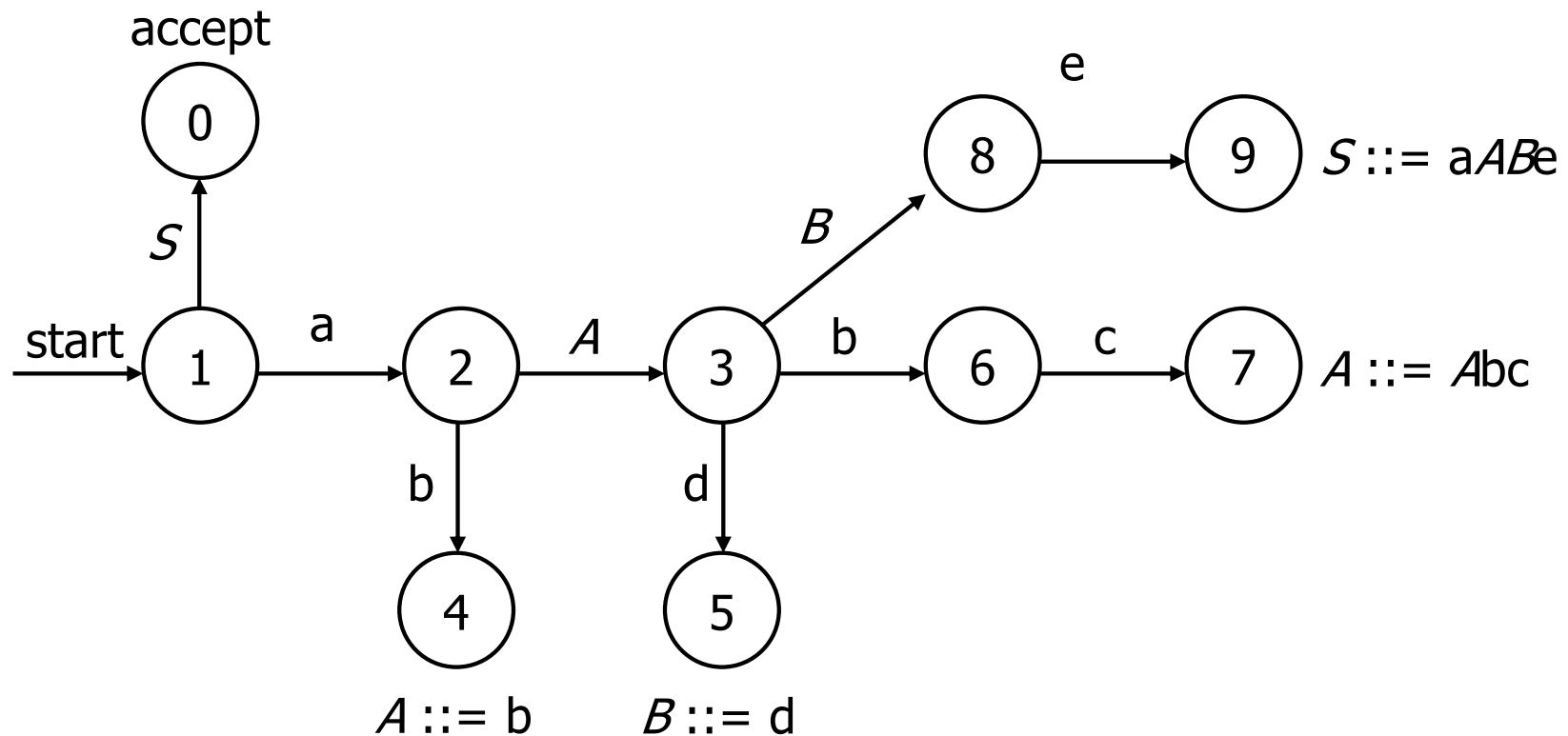
- When we construct the DFA we'll need to add a new production to handle end-of-file (i.e., end-of-input) correctly
- If  $S$  is the start state of the original grammar, add an initial production  $S' ::= S \$$ 
  - $\$$  represents end-of-file (input)
  - Accept when we've reduced the input to  $S$  and there is no more input (i.e., lookahead is  $\$$ )

# Reminder: DFA for

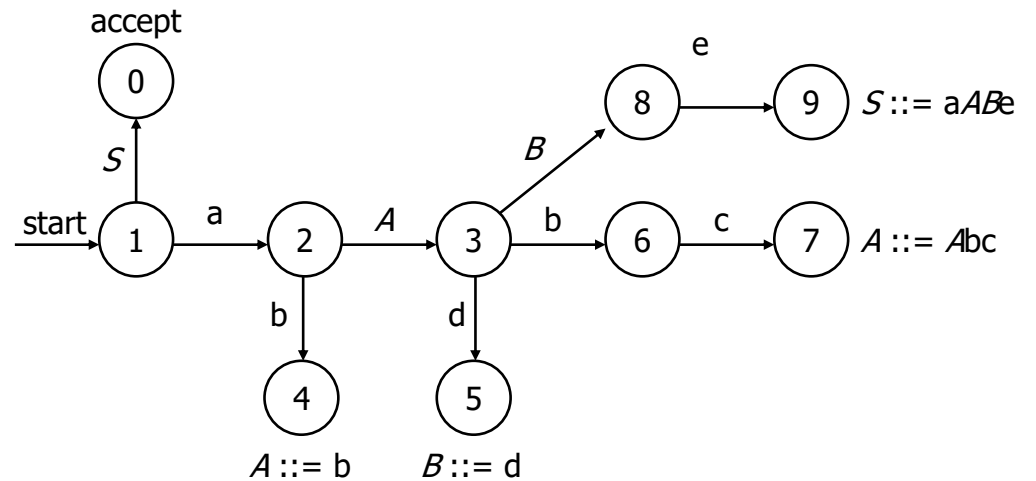
$S ::= aABe$

$A ::= Abc \mid b$

$B ::= d$



# LR Parse Table



State	action						goto		
	a	b	c	d	e	\$	A	B	S
0						acc			
1	s2								g0
2		s4					g3		
3		s6		s5				g8	
4	r3	r3	r3	r3	r3	r3			
5	r4	r4	r4	r4	r4	r4			
6			s7						
7	r2	r2	r2	r2	r2	r2			
8					s9				
9	r1	r1	r1	r1	r1	r1			

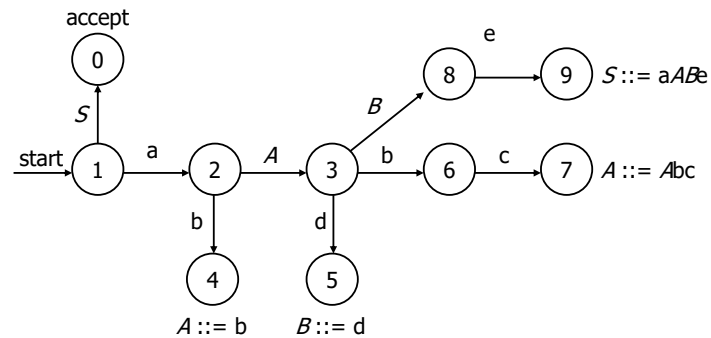
0.  $S' ::= S\$$
1.  $S ::= aABe$
2.  $A ::= Abc$
3.  $A ::= b$
4.  $B ::= d$

# LR Parsing Algorithm

```
tok = scanner.getToken();
while (true) {
    s = top of stack;
    if (action[s, tok] =  $s_i$ ) {
        push tok; push  $i$  (state);
        tok = scanner.getToken();
    } else if (action[s, tok] =  $r_j$ ) {
        pop 2 * length of right side of
        production  $j$  ( $2 * |\beta|$ );
        uncovered_s = top of stack;
        push left side  $A$  of production  $j$ ;
        push state goto[uncovered_s,  $A$ ];
    }
}
```

```
} else if (action[s, tok] = accept ) {
    return;
} else {
    // no entry in action table
    report syntax error;
    halt or attempt recovery;
}
```

# Example



0.  $S' ::= S\$$
1.  $S ::= aABe$
2.  $A ::= Abc$
3.  $A ::= b$
4.  $B ::= d$

Stack

\$1

\$1a2

\$1a2b4

\$1a2A3

\$1a2A3b6

\$1a2A3b6c7

\$1a2A3

\$1a2A3d5

\$1a2A3B8

\$1a2A3B8e9

\$1S0

Input

abbcde\$

bbcde\$

bcde\$

bcde\$

cde\$

de\$

de\$

e\$

e\$

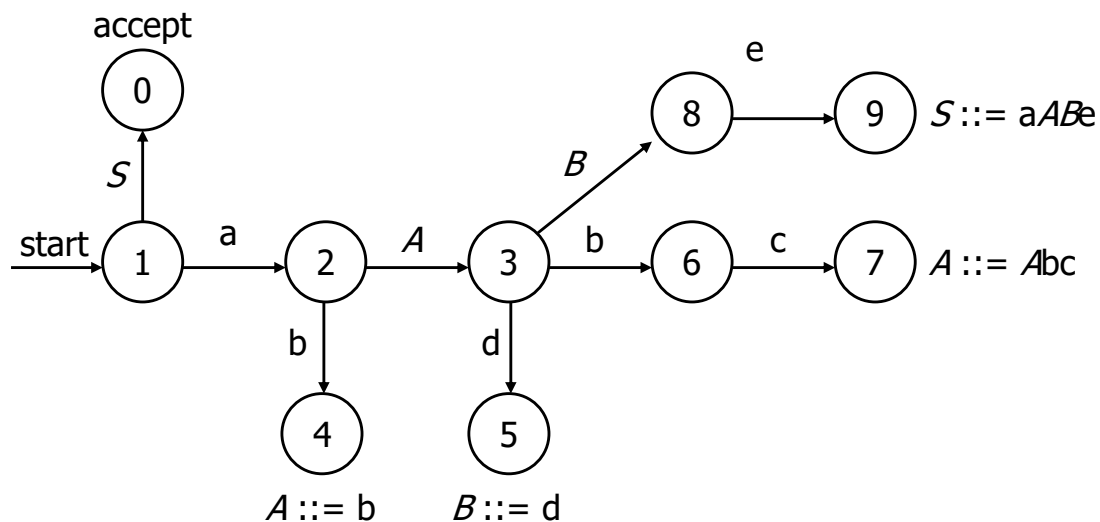
\$

\$

S	action						goto		
	a	b	c	d	e	\$	A	B	S
0						ac			
1	s2								g0
2		s4					g3		
3		s6		s5				g8	
4	r3	r3	r3	r3	r3	r3			
5	r4	r4	r4	r4	r4	r4			
6			s7						
7	r2	r2	r2	r2	r2	r2			
8					s9				
9	r1	r1	r1	r1	r1	r1			

# Why do we need the goto table?

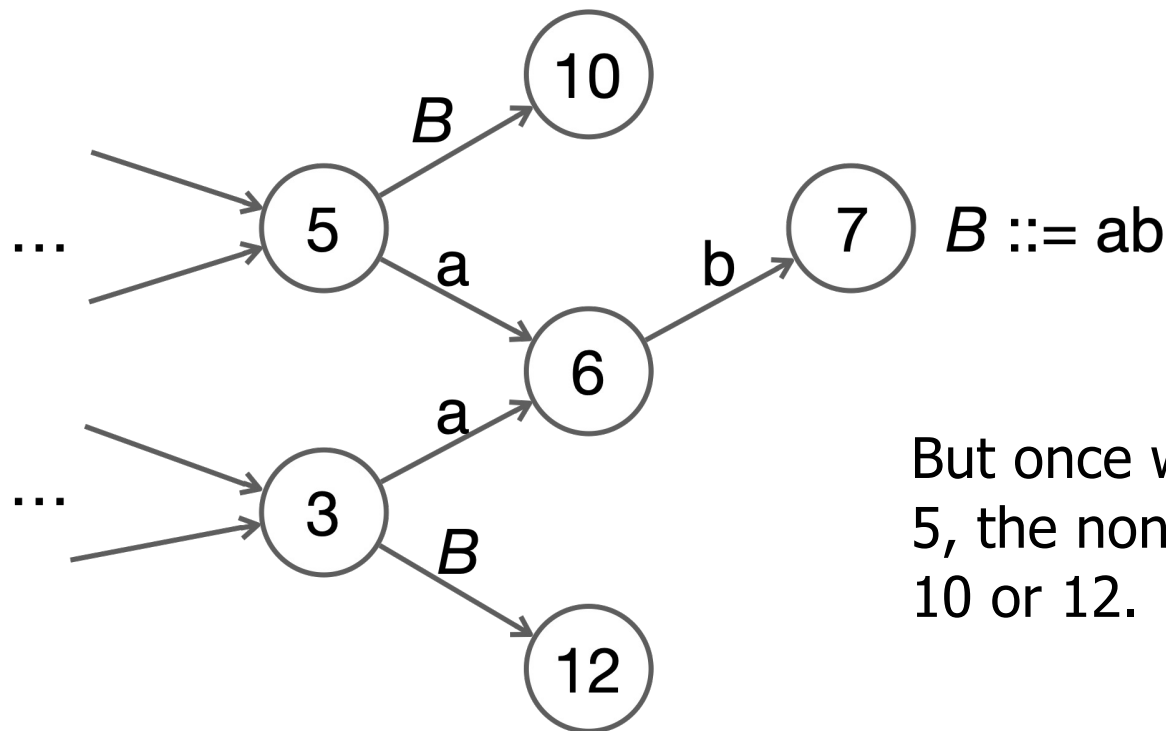
- In our example, if we reach state 5 and reduce with rule 4, we will always go back to state 3 and then transition on B to state 8. Similarly with the other reductions. So no goto table needed here! (maybe?)



0.  $S' ::= S\$$
1.  $S ::= aABe$
2.  $A ::= Abc$
3.  $A ::= b$
4.  $B ::= d$

# What if things are more complicated?

- When we reduce  $B ::= ab$  in state 7, we would uncover state 3 or 5 after popping the stack

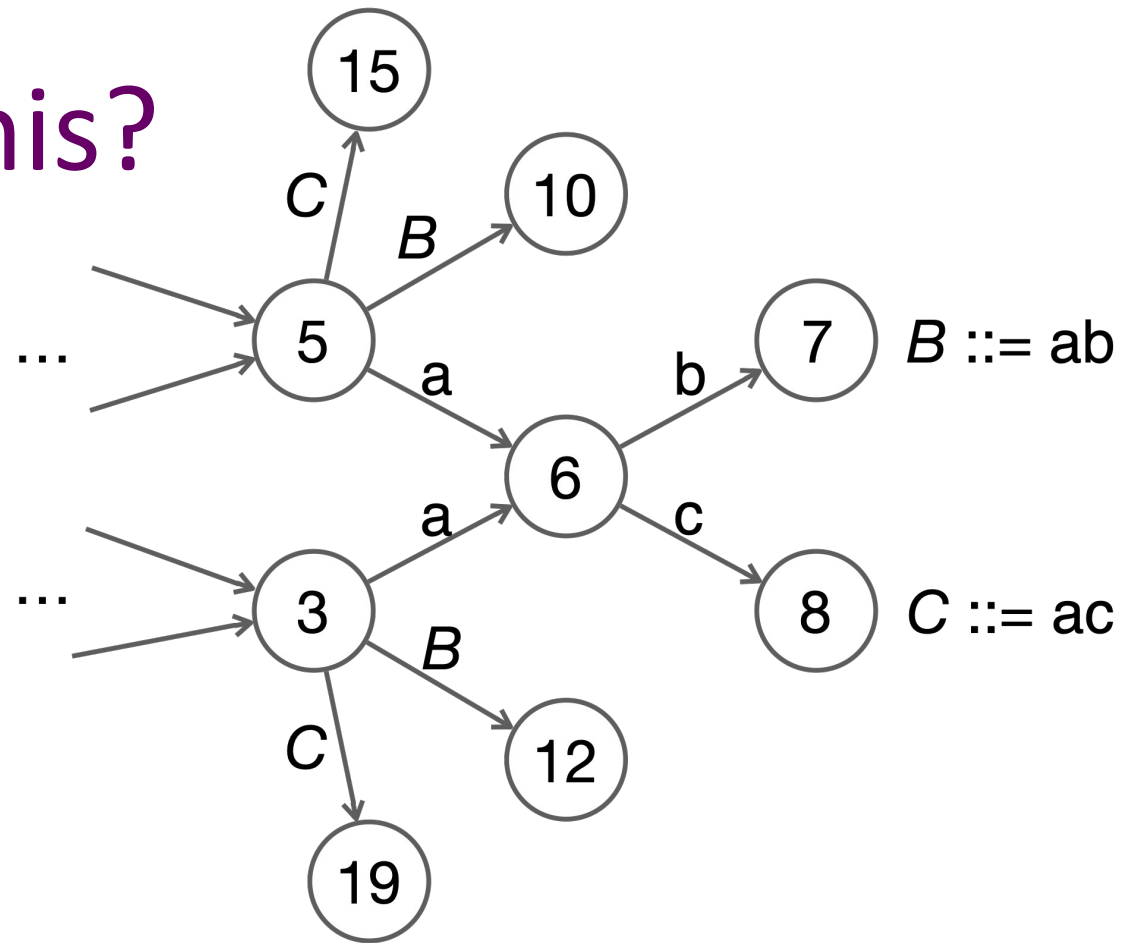


But once we get back to state 3 or 5, the non-terminal transition is to 10 or 12. So why the goto table?



# What about this?

If there are multiple reductions to different non-terminals that can pop us back to some state, we need the goto information to figure out which transition to take after backing up to that state.



Here the reductions in state 7 or 8 can return us to state 3 or 5, and then we need the goto information to know what the successor state is after we back up.

# LR States

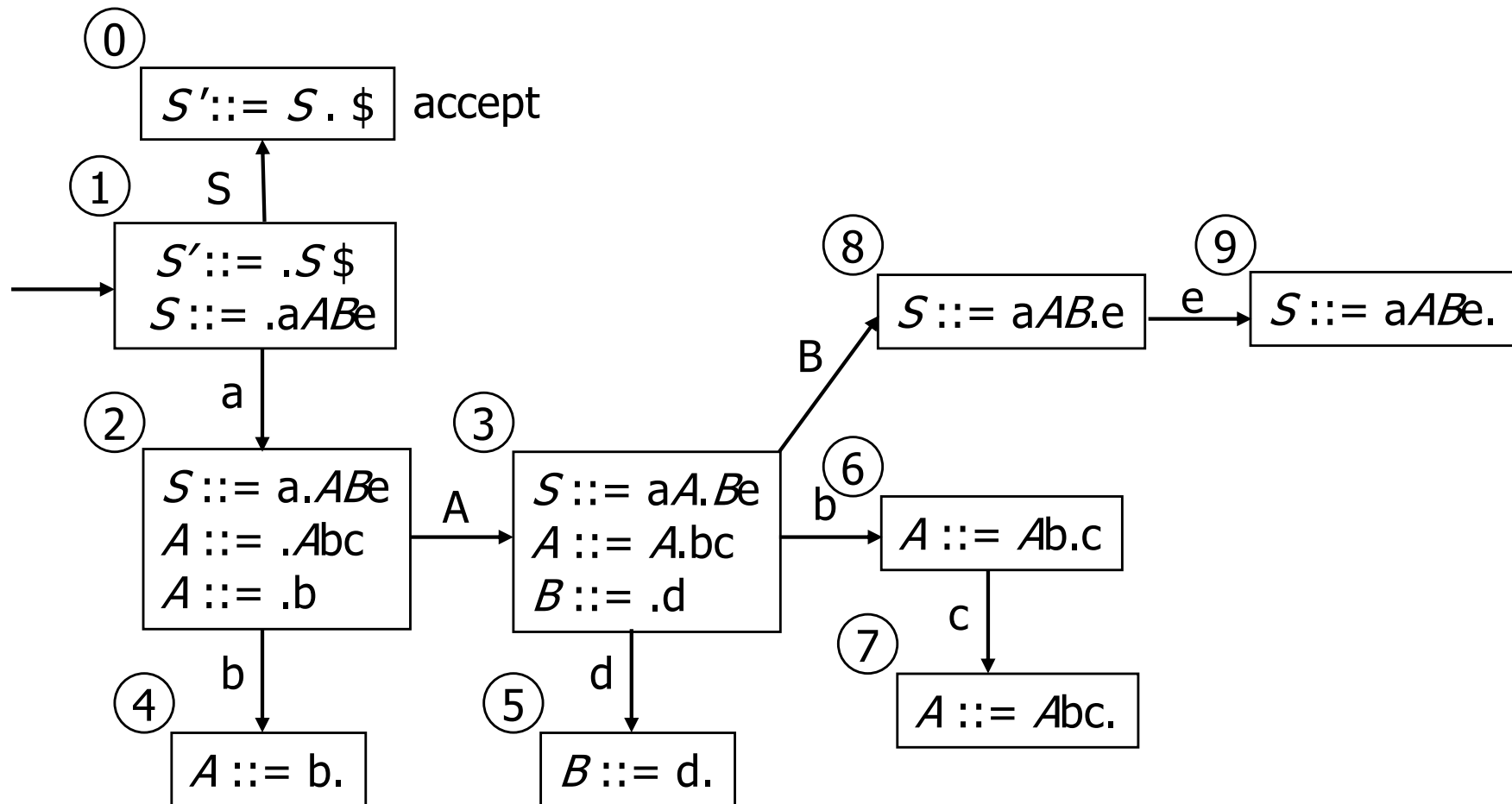
- Idea is that each state encodes
  - The set of all possible productions that we could be looking at, given the current state of the parse, and
  - *Where* we are in the right hand side of each of those productions

# Items

- An *item* is a production with a dot in the right hand side
- Example: Items for production  $A ::= X Y$ 
  - $A ::= . X Y$
  - $A ::= X . Y$
  - $A ::= X Y .$
- Idea: The dot represents a position in the production

# DFA for

$S' ::= S\$$   
 $S ::= aABe$   
 $A ::= Abc$   
 $A ::= b$   
 $B ::= d$



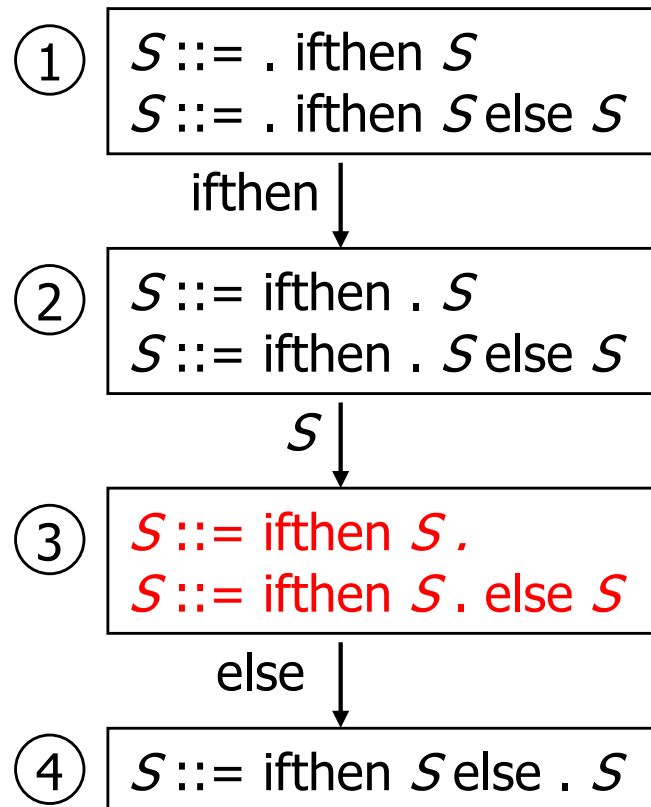
# Problems with Grammars

- Non-LR grammars can cause problems when constructing a LR parser
  - (that's how you it's not an LR grammar!)
  - Shift-reduce conflicts
  - Reduce-reduce conflicts
- i.e., arrive at a situation when two (or more) conflicting actions are called for

# Shift-Reduce Conflicts

- Situation: both a shift and a reduce are possible at a given point in the parse (equivalently: in a particular state of the DFA)
- Classic example: if-else statement  
$$S ::= \text{ifthen } S \mid \text{ifthen } S \text{ else } S$$

# Parser States for



1.  $S ::= \text{ifthen } S$
2.  $S ::= \text{ifthen } S \text{ else } S$

- State 3 has a shift-reduce conflict
  - Can shift past else into state 4 (s4)
  - Can reduce (r1)  
 $S ::= \text{ifthen } S$

(Note: other  $S ::= . \text{ifthen}$  items not included in states 2-4 to save space)

# Solving Shift-Reduce Conflicts

- Option 1: Fix the grammar
  - Done in Java reference grammar, others
- Option 2: Use a parse tool with a “longest match” rule – i.e., if there is a conflict, choose to shift instead of reduce
  - Does exactly what we want for if-else case
  - Guideline: a few shift-reduce conflicts are fine, but be sure they do what you want (and that this behavior is guaranteed by the tool specification)



# Reduce-Reduce Conflicts

- Situation: two different reductions are possible in a given state
- Contrived example

$S ::= A$

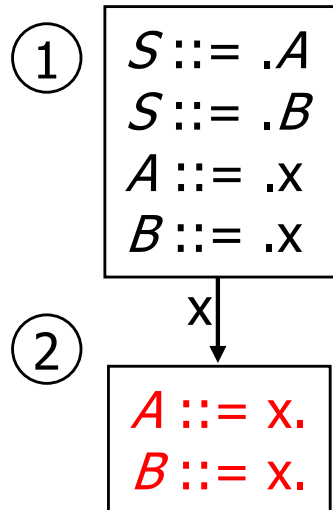
$S ::= B$

$A ::= x$

$B ::= x$

# Parser States for

1.  $S ::= A$
2.  $S ::= B$
3.  $A ::= x$
4.  $B ::= x$



- State 2 has a reduce-reduce conflict (r3, r4)

# Handling Reduce-Reduce Conflicts

- These normally indicate a serious problem with the grammar.
- Fixes
  - Use a different kind of parser generator that takes lookahead information into account when constructing the states
    - Most practical tools (Yacc, Bison, CUP, et al) do this
  - Fix the grammar

# Another Reduce-Reduce Conflict

- Suppose the grammar tries to separate arithmetic and boolean expressions

*expr ::= aexp | bexp*

*aexp ::= aexp \* aident | aident*

*bexp ::= bexp && bident | bident*

*aident ::= id*

*bident ::= id*

- This will create a reduce-reduce conflict state with items [ *aident ::= id .* , *bident ::= id .* ]

# Covering Grammars

- A solution is to merge *aident* and *bident* into a single non-terminal (basically use *id* in place of *aident* and *bident* everywhere they appear)
- This is a *covering grammar*
  - Will generate some programs (sentences) that are not generated by the original grammar
  - Use the type checker or other static semantic analysis to weed out illegal programs later

# Coming Attractions

- Constructing LR tables
  - We'll present a simple version (SLR(0)) in lecture, then talk about extending it to LR(1) and then a little bit about how this relates to LALR(1) used in most parser generators – the basic ideas behind the construction algorithm and set of states are the same for all of these
- LL parsers and recursive descent
- Continue reading ch. 3