Lecture D:

LR Parsing

CSE401/501m:

Introduction to Compiler Construction

Instructor: Gilbert Bernstein

Administrivia

- HW1 due last night
  - 1-2 late days? don't blow all your late days now!
- Project
  - Scanner due Thursday; but TEST infrastructure now
  - DO NOT start on the parser yet — just edit the token classes in the .cup file (and small edits to build)
  - Almost everyone paired up; please, please respond if you're not paired
- HW2 (LR Parsing & Grammars) — due in 2 weeks; will post when we get enough background done (prob. Monday)
- Room for Sections next week is moved to Savery 131 (if you show up to SAV 166, there is a notice that redirects you)

Administrivia (Monday)

- Project
  - ✦ Scanner due Thursday; please also complete the Gradescope step (reason: experiment to try to reflect all numeric grades in gradescope this year)
- HW2 (LR Parsing & Grammars) — will wait till Wed.
- Room for Sections this week is Savery 131 (if you show up to SAV 166, there is a notice that redirects you)

Outline

LR Parsing

Automating Parsing

Table-driven Parsers

LR States

Shift-Reduce & Reduce-Reduce Conflicts

Outline

LR Parsing

Automating Parsing

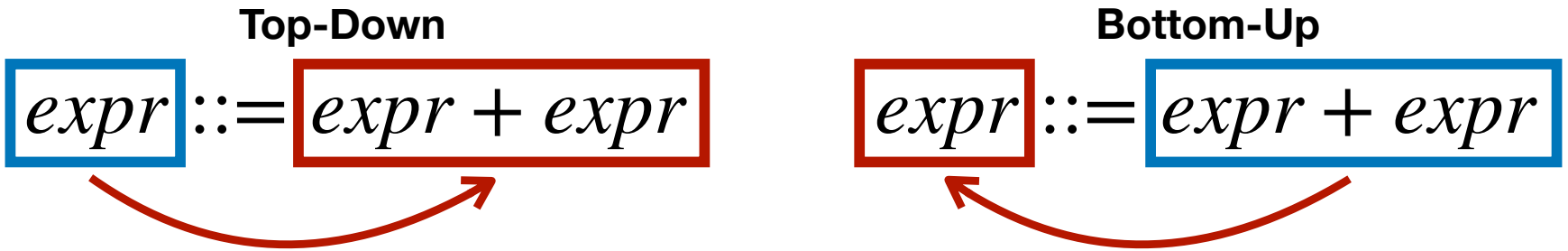
Table-driven Parsers

LR States

Shift-Reduce & Reduce-Reduce Conflicts

Bottom-Up Parsing

- Easy to get all the different directions mixed up
  - ✦ read the input — **left-to-right** not right-to-left
  - ✦ derivation order — will produce a **rightmost** derivation
  - ✦ **"bottom-up"** — will match the **right-hand side** of productions, not the **left-hand side** non-terminal
- **Key idea** — whenever we match a complete right-hand-side pattern of a production, we can replace that series of tokens with the left-hand-side non-terminal, e.g.



Example

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

a b b c d e

Example

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

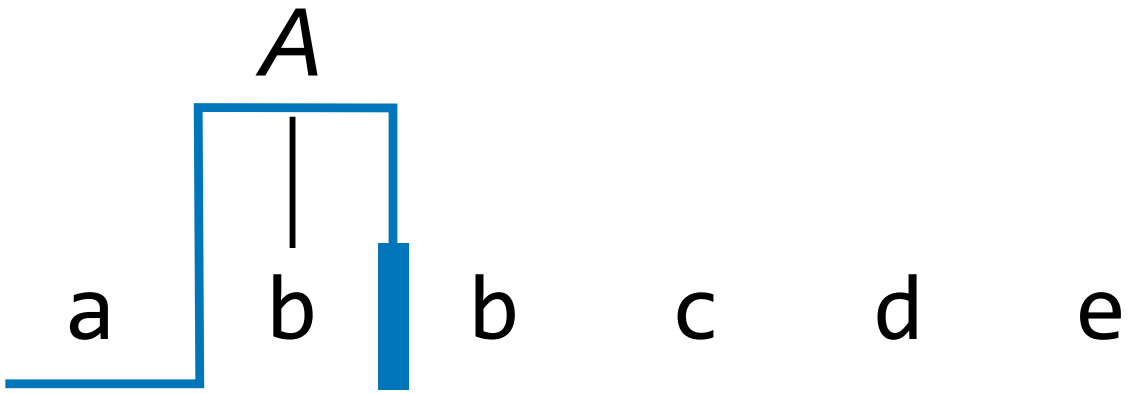
Example

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

Example

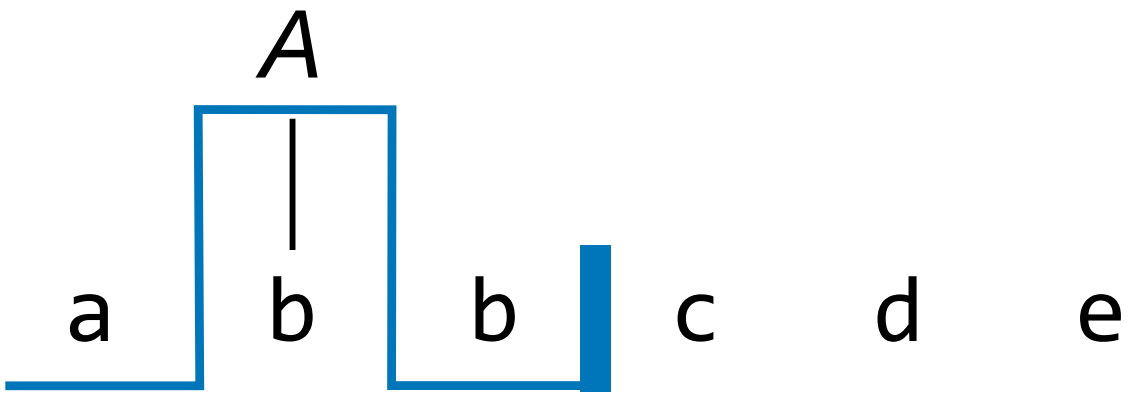
$$S ::= aABe$$
$$A ::= Abc \mid b$$
$$B ::= d$$

$$A ::= b$$



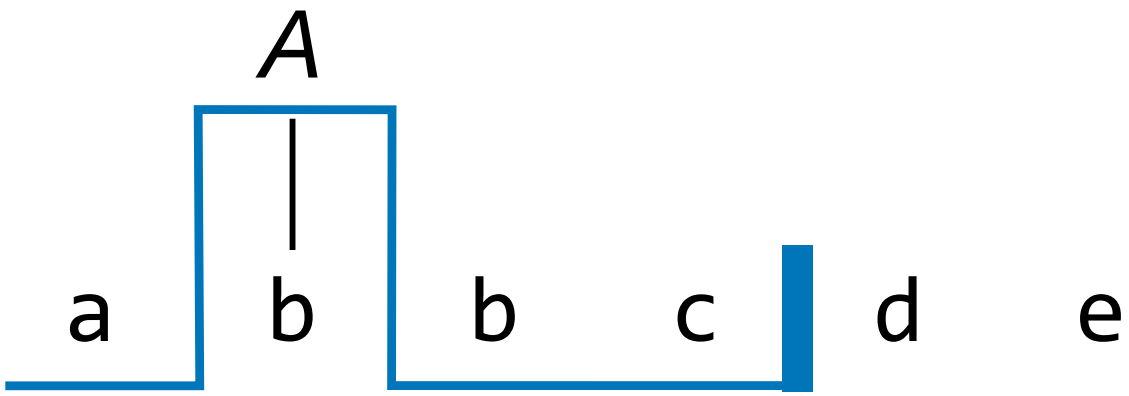
Example

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



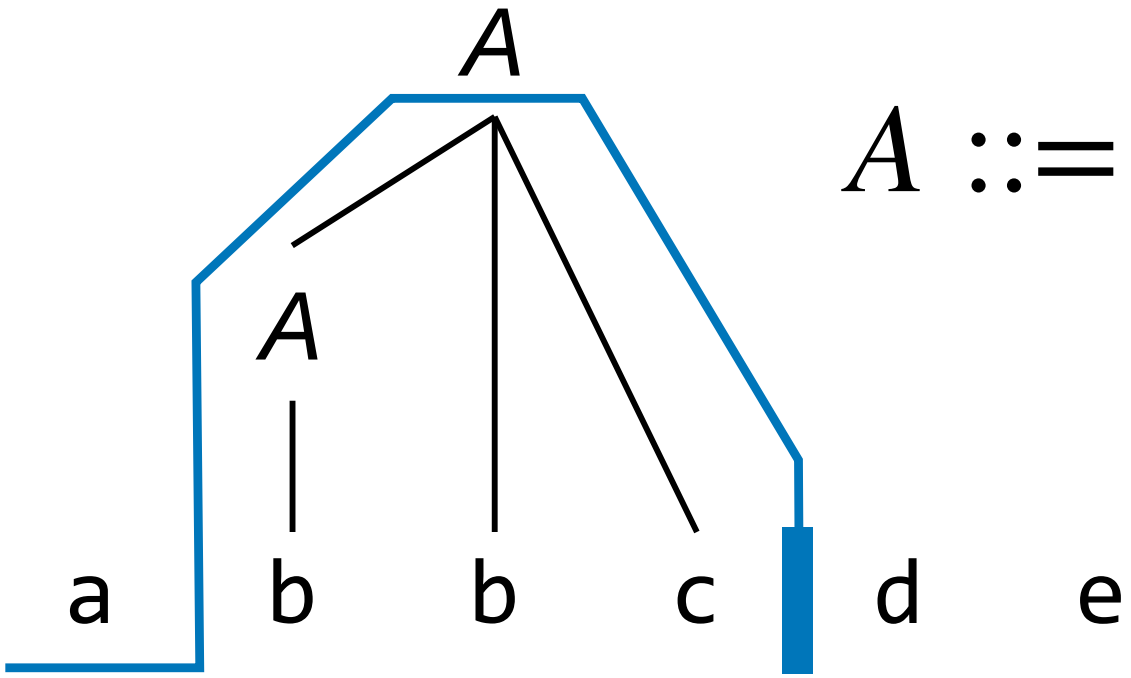
Example

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



Example

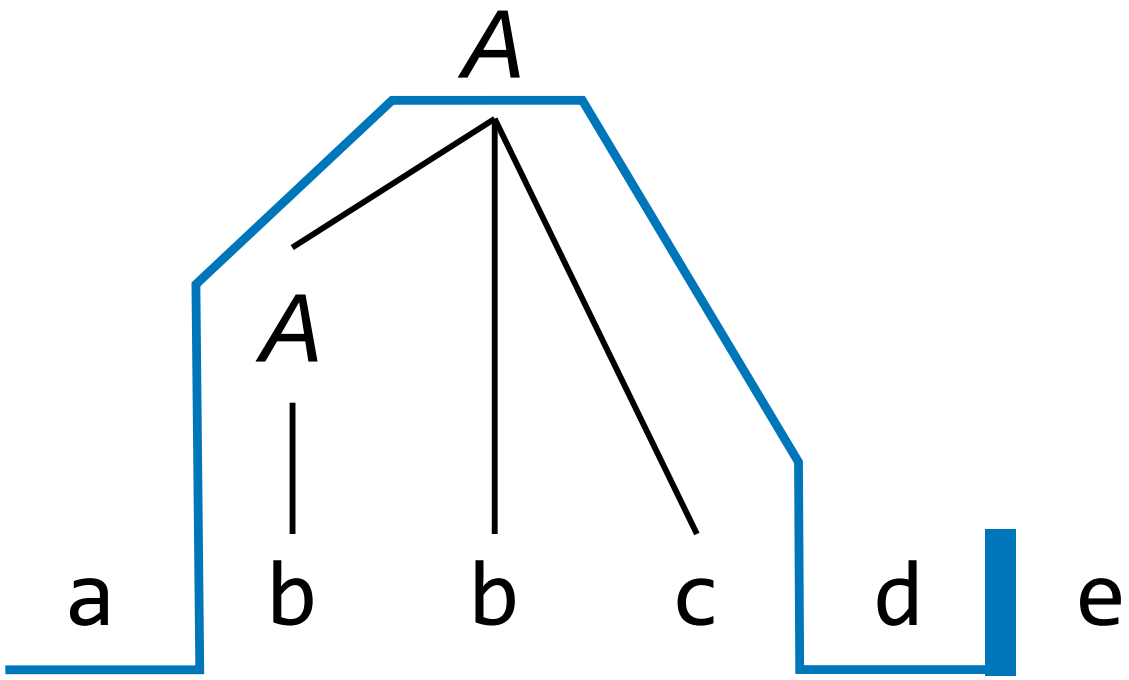
$$S ::= aABe$$
$$A ::= Abc \mid b$$
$$B ::= d$$



$$A ::= Abc$$

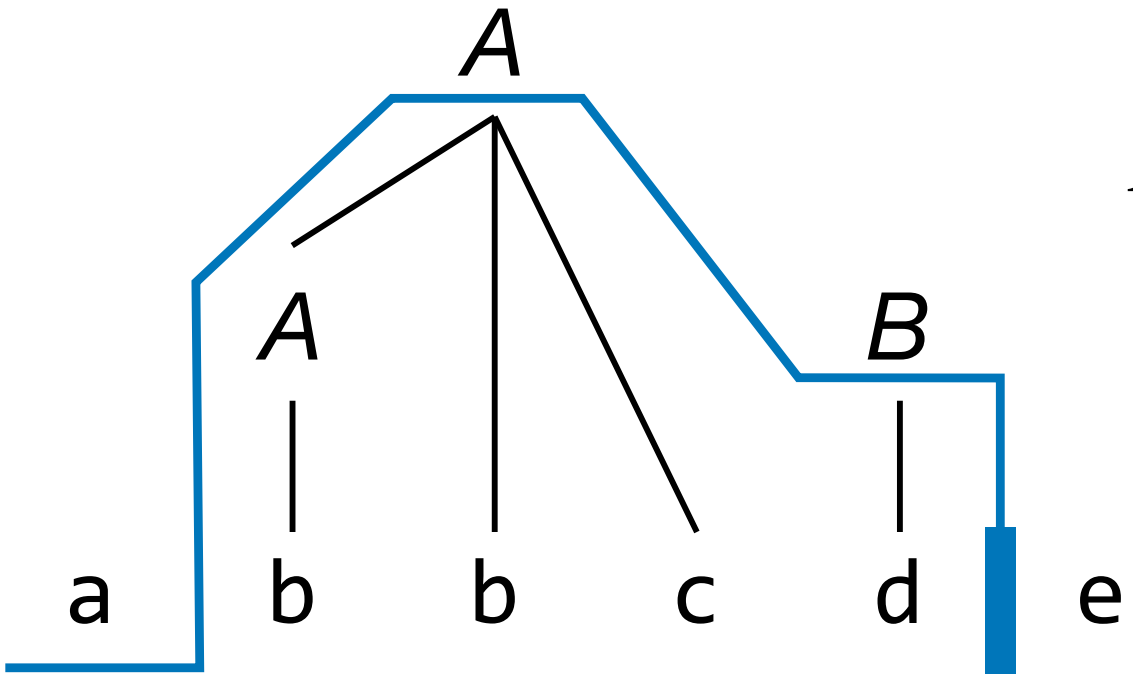
Example

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



Example

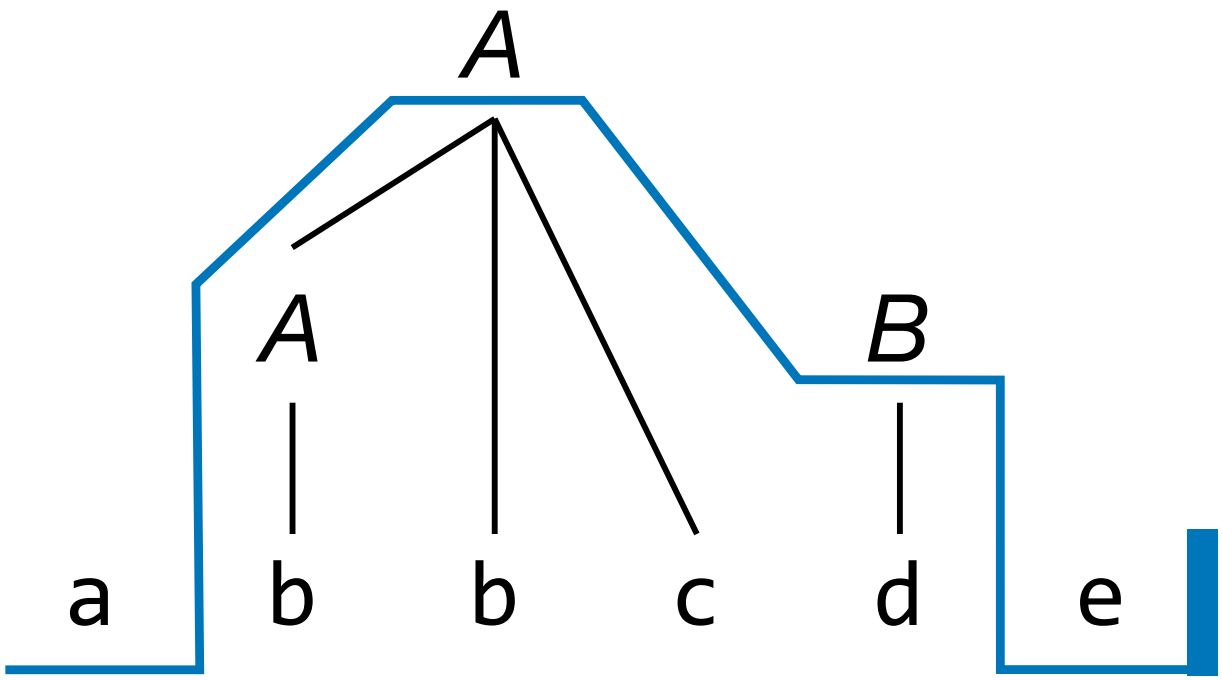
$$S ::= aABe$$
$$A ::= Abc \mid b$$
$$B ::= d$$



$$B ::= d$$

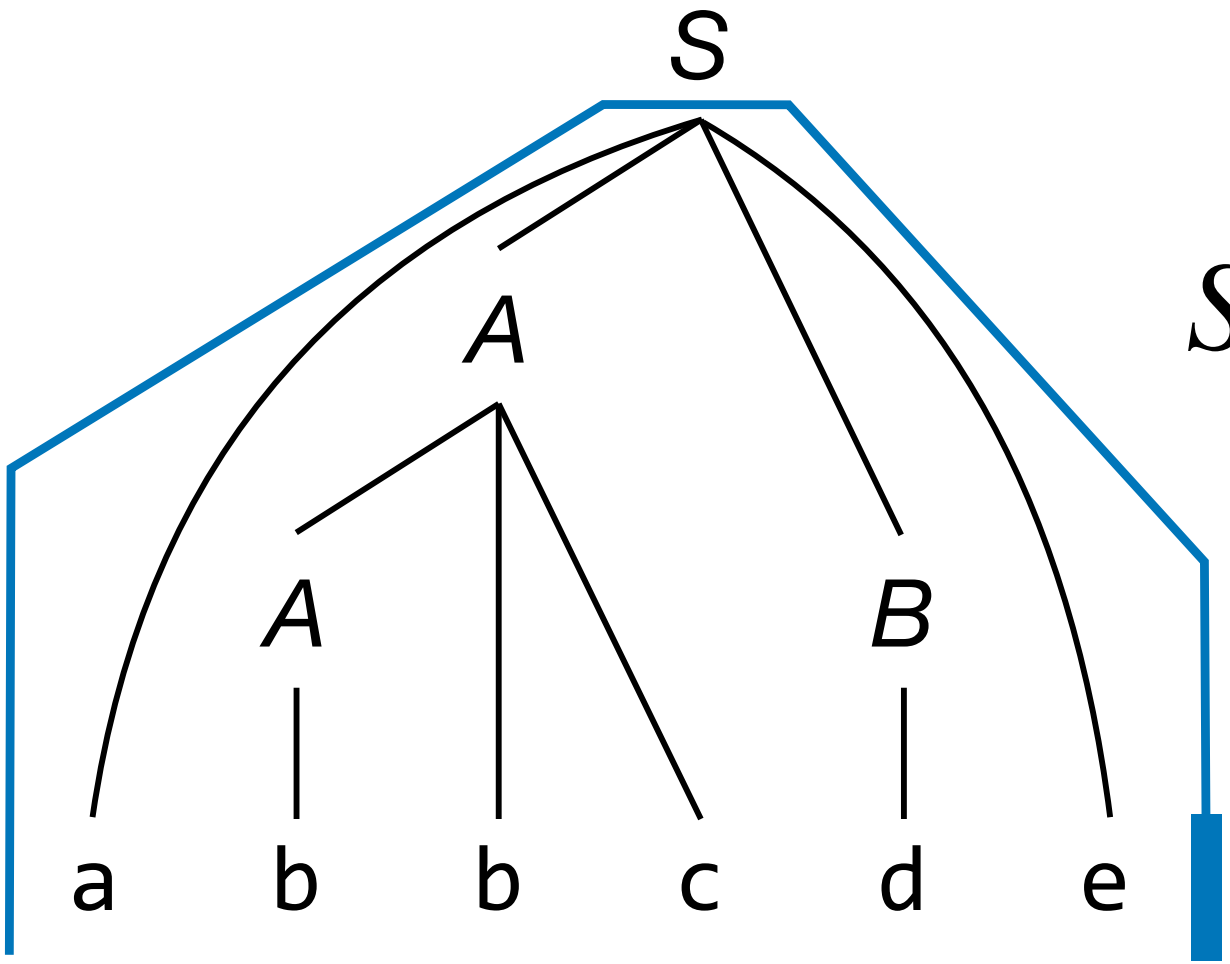
Example

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



Example

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



$$S ::= aABe$$

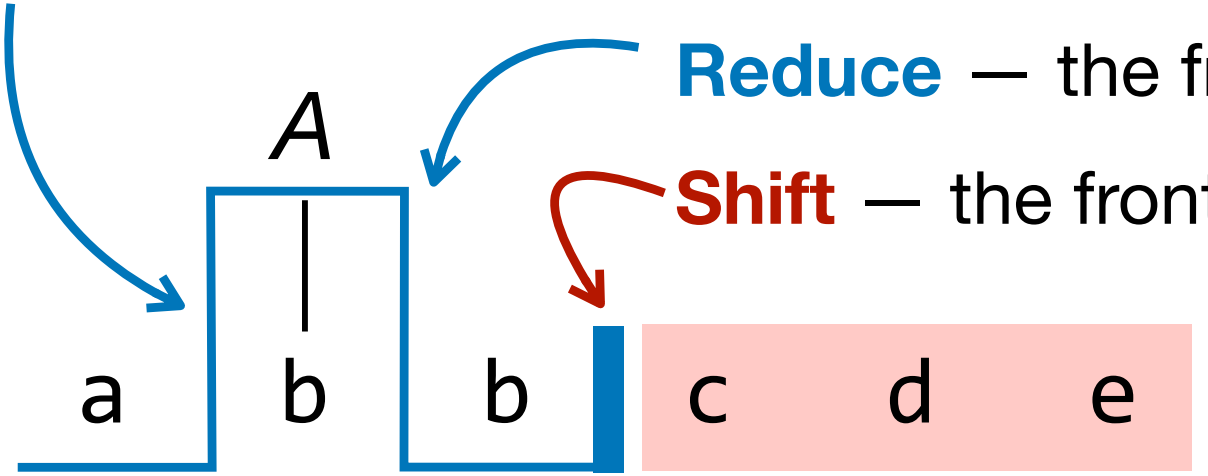
LR(1) Parsing

- We'll look at **LR(1)** parsers
  - **L**eft-to-right scan of input, **R**ightmost derivation, **1** symbol lookahead
  - Almost all practical programming languages have an 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.

Analyzing the Example

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

This is called the **Frontier** of the parse



This is the current position of the **Scanner**

Two Kinds of Steps

Reduce — the frontier moves up

Shift — the frontier moves right

We **haven't scanned** these tokens yet

Recording the Parse

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

Frontier	Action
$\|abbcde$	Shift
$a\|bbcde$	Shift
$ab\|bcde$	Reduce *A* ::= b
$a*A*\|bcde$	Shift
$a*A*b\|cde$	Shift
$a*A*bc\|de$	Reduce *A* ::= *A*bc
$a*A*\|de$	Shift
$a*A*d\|e$	Reduce *B* ::= d
$a*AB*\|e$	Shift
$a*AB*e\|$	Reduce *S* ::= a*AB*e
$*S*\|$	**Accept**

In Reverse

S

$\Rightarrow_{rm}$ a*AB*e

$\Rightarrow_{rm}$ a*A*de

$\Rightarrow_{rm}$ a*A*bcde

$\Rightarrow_{rm}$ abbcde

a rightmost derivation

LR Parsing in Greek

- The bottom-up parser reconstructs the rightmost derivation, in ***reverse***
- Consider the *rightmost* derivation of our input seq. w
  - $S \Rightarrow_{\text{rm}} \beta_1 \Rightarrow_{\text{rm}} \beta_2 \Rightarrow_{\text{rm}} \cdots \Rightarrow_{\text{rm}} \beta_{n-1} \Rightarrow_{\text{rm}} \beta_n = w$
  - The parser discovers each step in reverse, i.e. $\beta_{n-1} \Rightarrow_{\text{rm}} \beta_n$ then $\beta_{n-2} \Rightarrow_{\text{rm}} \beta_{n-1}$, etc.
- Parsing terminates when
  - β_1 is reduced to S (start symbol, success), or
  - no match can be found (syntax error)

How Do We Parse With This?

- Given what we've already seen and the next input symbol (the lookahead) decide what to do
  - **Shift** — Ask the scanner for another token
  - **Reduce** — Perform a reduction (inverse derivation step)
- Can reduce via $\alpha A w \Rightarrow_{\text{rm}} \alpha \beta w$ when
  - $w \in \Sigma^*$ (the rest of the sentence is terminal)
    - *note: guaranteed by left-to-right scan*
  - $A ::= \beta$ is a valid production
- This is the formal justification for **shift-reduce** parsing

Terminology (Useful?)

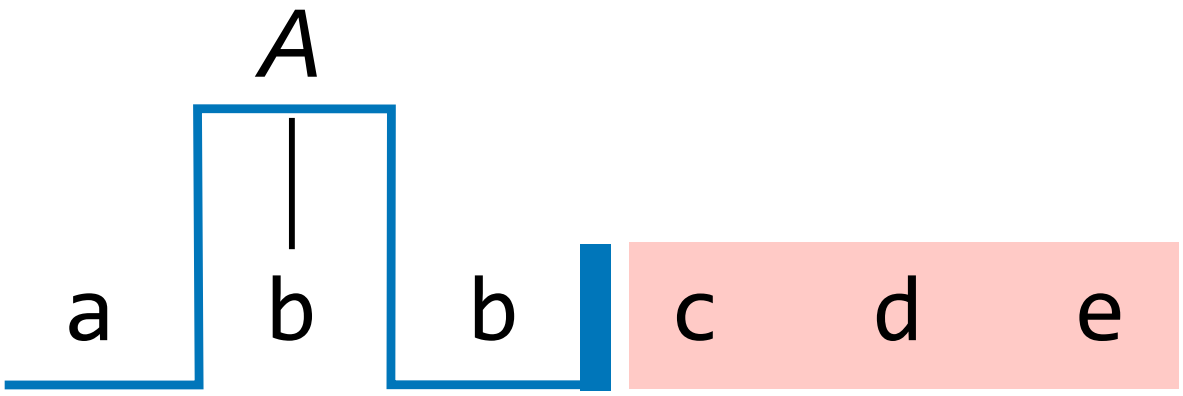
- You will see books and sources refer to the strings of symbols in a derivation as **sentential forms**
  - ✦ This is just a fancy word for sentence, which is just a string of symbols)
  - ✦ In a rightmost derivation, **right-sentential form**
- The site (location) of a reduction is called a **handle**
  - ✦ A handle is a pair of a production and integer telling us which substring to reduce using that production
  - ✦ i.e. for reduction/production step $\alpha A w \Rightarrow_{\text{rm}} \alpha\beta w$, the handle is $\langle A ::= \beta, k \rangle$ where k is the length of $\alpha\beta$ (i.e. right-index convention; some books use left)

Handles: Examples

$\|abbcde$	Shift	
$a\|bbcde$	Shift	
$ab\|bcde$	Reduce	*A* ::= b
$a*A*\|bcde$	Shift	
$a*A*b\|cde$	Shift	
$a*Abc*\|de$	Reduce	*A* ::= *A*bc
$a*A*\|de$	Shift	
$a*Ad*\|e$	Reduce	*B* ::= d
$a*AB*\|e$	Shift	
$a*ABe*\|$	Reduce	*S* ::= a*AB*e
$*S*\|$	**Accept**	

$$S \Rightarrow_{rm} aABe \Rightarrow_{rm} aAde \Rightarrow_{rm} aAbcde \Rightarrow_{rm} abbcde$$

Implementing LR Parsers



- State — Data Structures
  - ✦ a **stack** of symbols — representing the **frontier**
  - ✦ a **stream** of unread terminals — i.e. the **scanner**
- A function that uses (a) the state (i.e. stack) and (b) one-symbol lookahead to decide what ***action*** to take (e.g. to shift, or to reduce using which production)

Shift-Reduce Parser Actions

- **Shift** —Push the next symbol onto the stack and get the next token from the scanner
- **Reduce** — Using production $A ::= \beta$, pop β from the top of the stack and push A in its place
- **Accept** — The stack contains only S. Announce success
- **Error** — Syntax error discovered

Early Errors?

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

- Naively, if we can't **reduce**, we can always **shift**.
  So, how will we get stuck (and thus **error**)?
- Consider the sentence `dadbabe` (aka. hotpop)
  - How long do we need to read input before we can report an error? Why?
- Prefixes — consider valid first terminal symbols
  - $S \Rightarrow aABe$, so a is the ***only*** valid first character
  - Valid second character?

Outline

LR Parsing

Automating Parsing

Table-driven Parsers

LR States

Shift-Reduce & Reduce-Reduce Conflicts

Definition: Viable Prefixes

- (most useful) a **viable prefix** is a sentence that can occur as the ***contents of the stack during LR parsing***
- (with more terminology, but less greek) a **viable prefix** is a prefix of a right-sentential form that does not continue past the rightmost handle of that sentential form
- (with greek) the sentence γ is a **viable prefix** if there exists some derivation $S \Rightarrow^*_{\text{rm}} \alpha A w \Rightarrow_{\text{rm}} \alpha\beta w$ and γ is a prefix of $\alpha\beta$

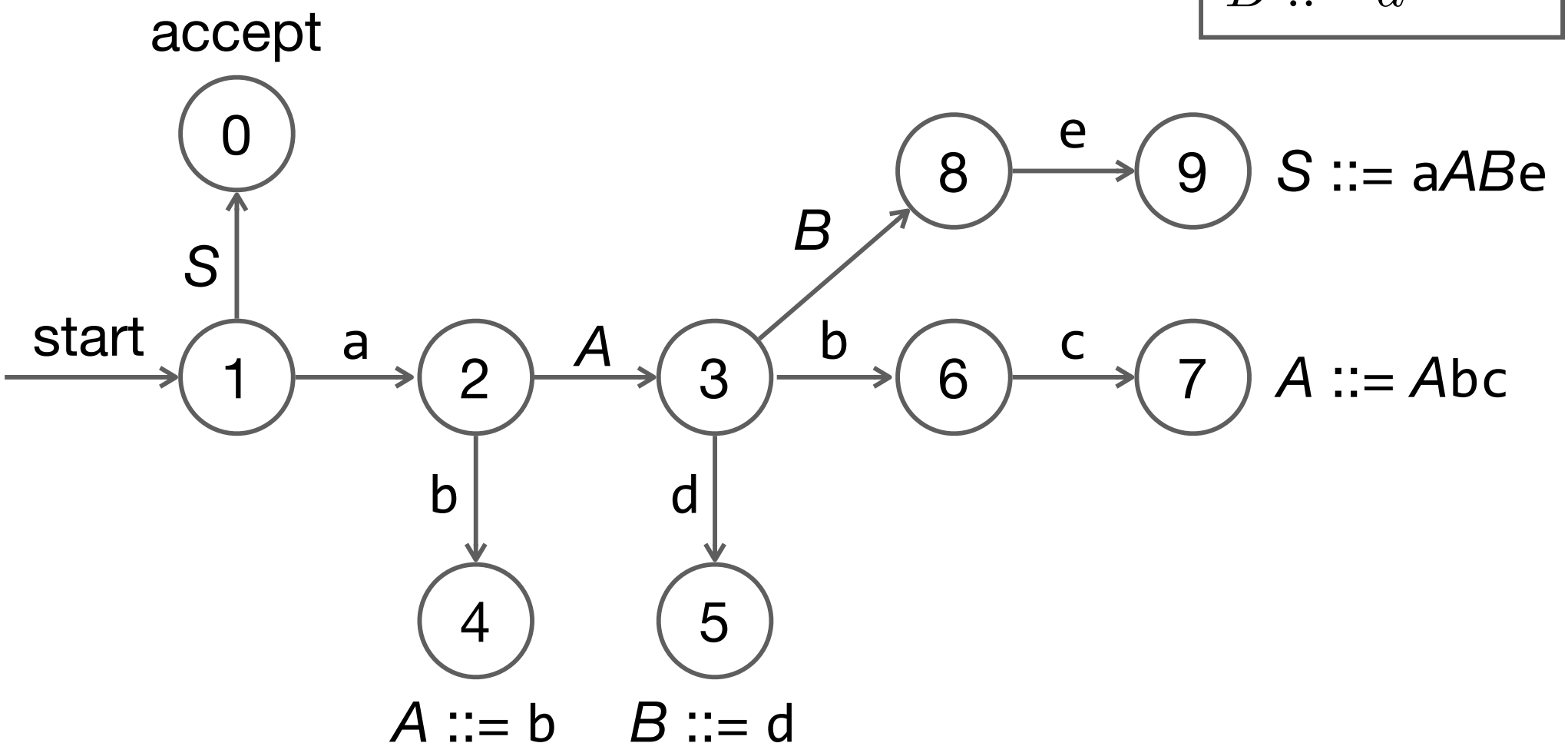
How Do We Automate LR Parsing?

- Let's exploit viable prefixes
  - **Fact — The set of viable prefixes of a CFG is a regular language**
- Idea 1: Construct a DFA to recognize viable prefixes
  - This will help us with errors for sure, but what else?
- Idea 2: The DFA that recognizes viable prefixes can also recognize whether the top of the stack (right of the prefix) is reducible (is a "handle")
  - Thus, we can use a DFA to tell us when to reduce

Step 1
Magically Produce a DFA

DFA for Prefixes of

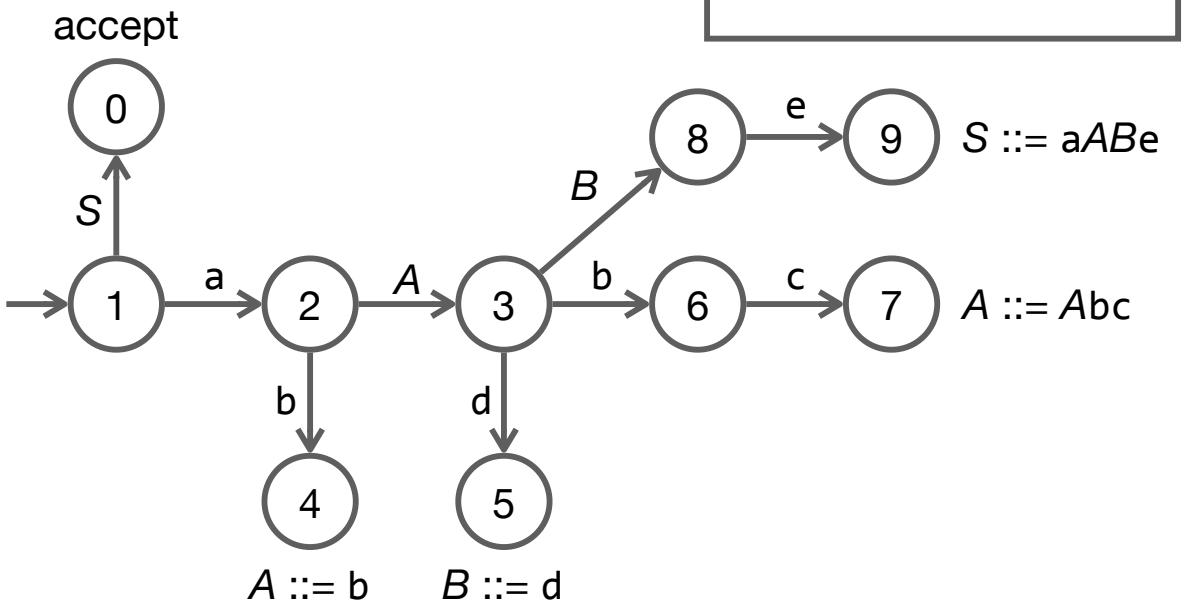
$$S ::= aABe$$
$$A ::= Abc \mid b$$
$$B ::= d$$



Trace Using DFA

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

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



Observations

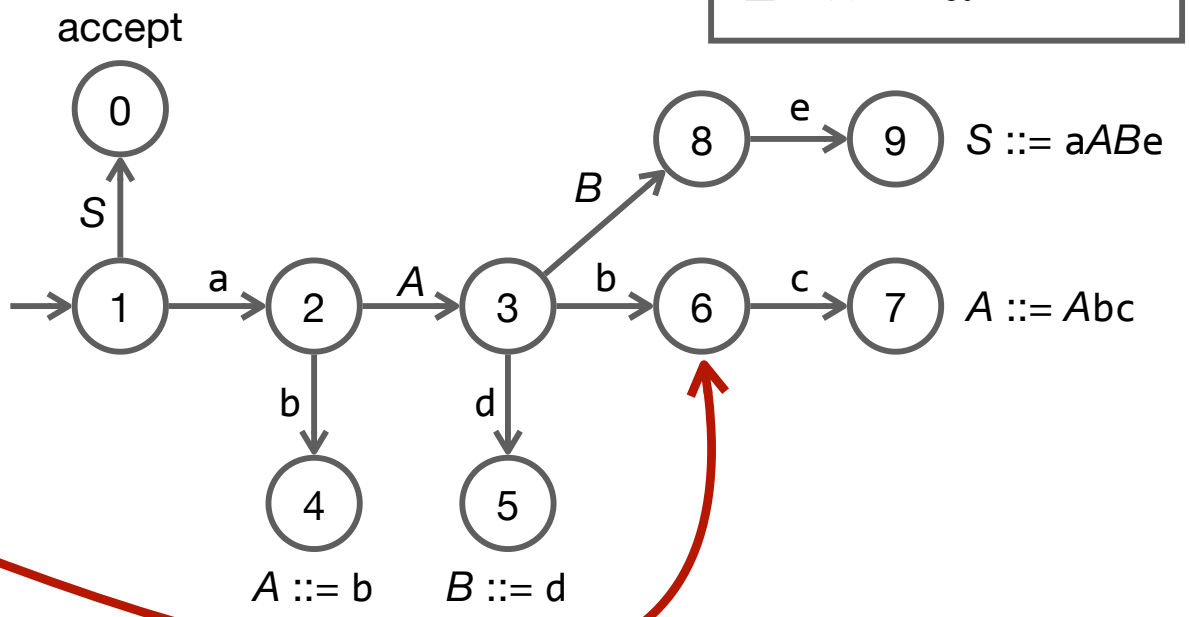
- Way too much backtracking
  - ✦ We want the parser to run in time proportional to the length of the input
- Where did this DFA come from anyway?
  - ✦ From the underlying grammar
  - ✦ We'll defer construction details for now

Avoiding DFA Rescanning

- Observations
  - ✦ There's no need to restart the DFA after a **shift**
  - ✦ After a **reduction**, the stack is the same except a new non-terminal replaces the top k symbols symbols
- Thus, scanning the stack will largely repeat the same, already taken transitions.
- We can record state numbers on the stack to help us back up to the correct state after performing a reduce

Alt. Stack Encoding

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

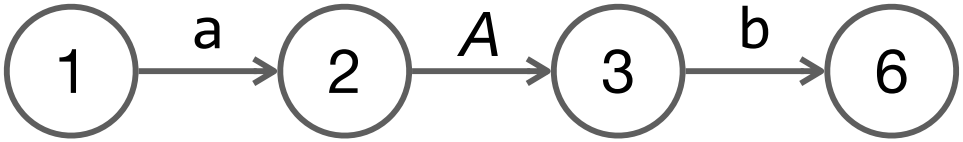


original stack encoding

$aAb|cde$

What state are we in?

What sequence of states did we traverse?



Two alternate encodings of the stack

$1236|$

$1a2A3b6|$

Alternate Stacks in General

- Original Stack — sentence of terminals & non-terminals
  - $\$X_1X_2\ldots X_n|$
- New stack — an interleaved sequence of symbols and ***state identifiers***
  - $\$s_0X_1s_1X_2s_2\ldots X_ns_n|$
- State s_0 is the start state of the DFA
- If shift transitions in the DFA via $\xrightarrow{X} s$, then we push Xs onto the stack. Thus, the stack ***is*** the DFA trace.
- When we reduce, the new top of the stack tells us which state to back up to

Outline

LR Parsing

Automating Parsing

Table-driven Parsers

LR States

Shift-Reduce & Reduce-Reduce Conflicts

Analyzing DFA Actions

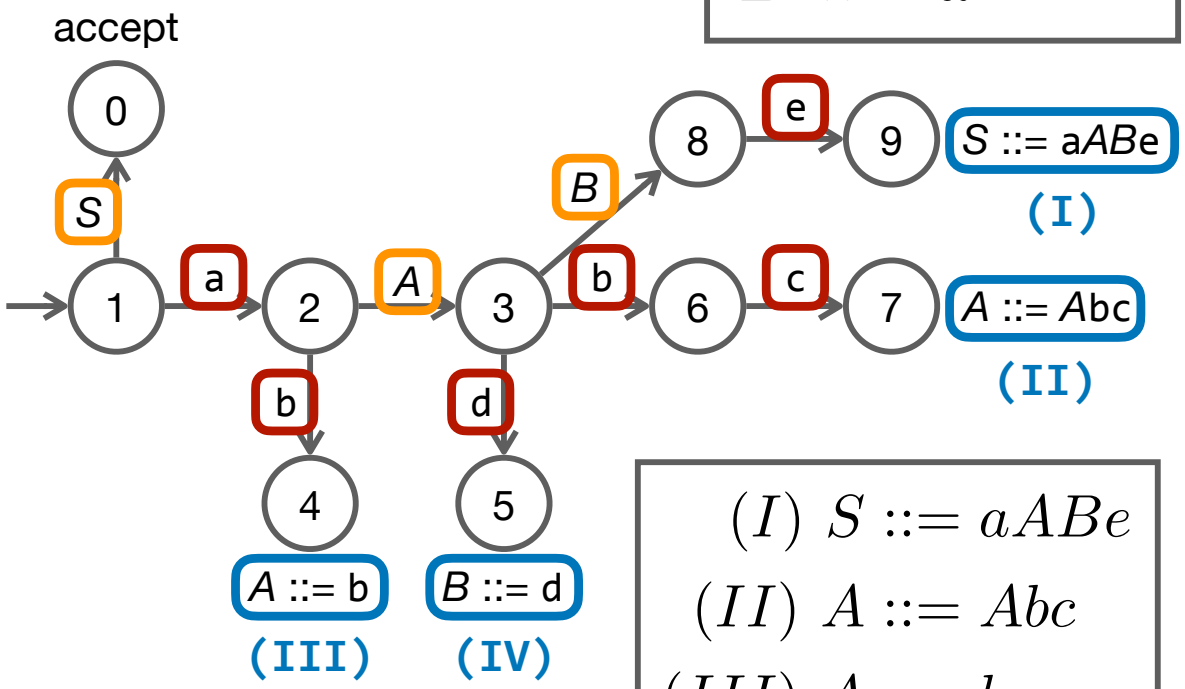
$$S ::= aABe$$
$$A ::= Abc \mid b$$
$$B ::= d$$

Transitions on terminals must be **shift** actions

If we're at a production labeled node, *AND* no shift actions apply, then we should **reduce**

Note: Roman numerals to track productions

pop production RHS from stack, push the nonterminal LHS and **goto** the next state



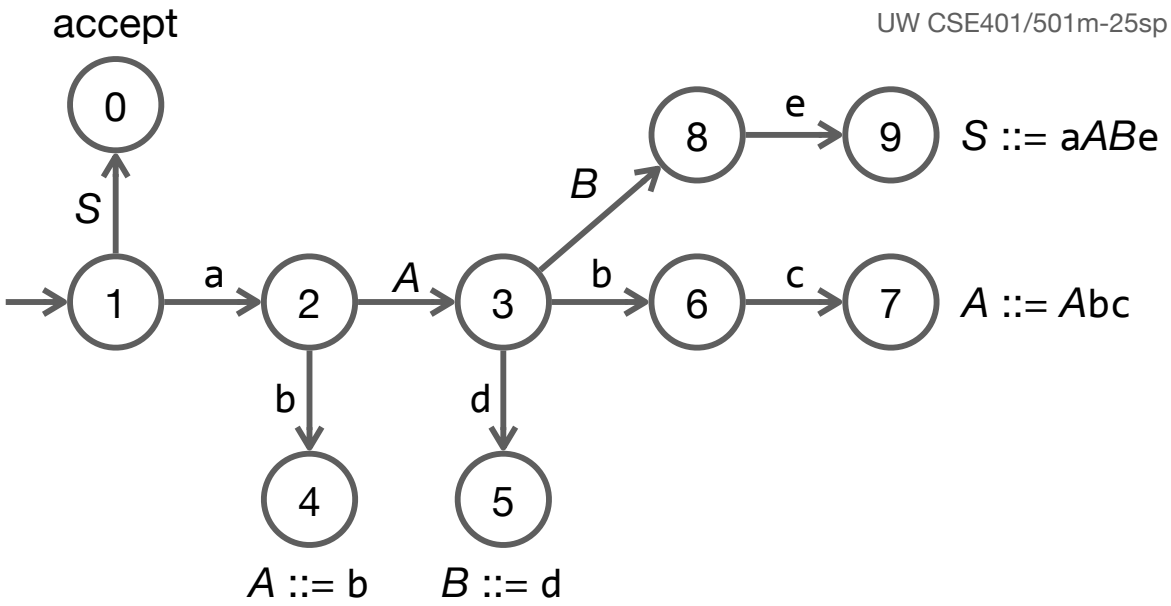
$$(I)\ S ::= aABe$$
$$(II)\ A ::= Abc$$
$$(III)\ A ::= b$$
$$(IV)\ B ::= d$$

Encoding the DFA in a Table

- One row for each state of the DFA
- Two groups of columns
  - **action table** — one column per terminal symbol; tells us which action to take
  - **goto table** — one column per non-terminal symbol; helps us make correct state transitions after a reduce. We'll see how in a second (slightly counter-intuitive)

Example

LR Parse Table

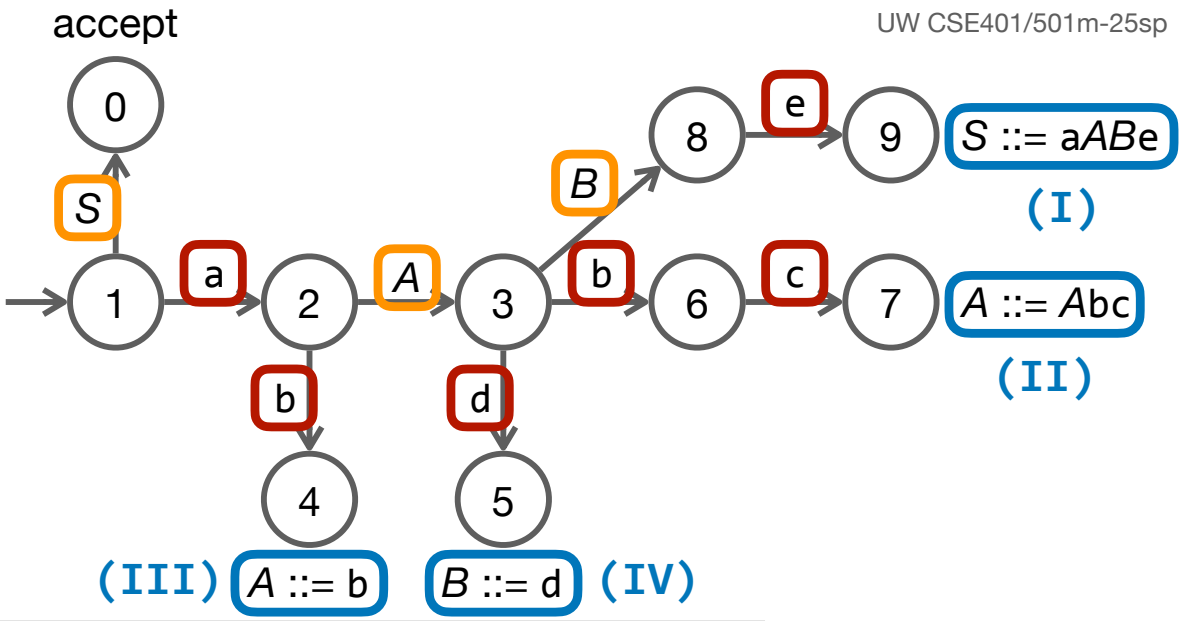


State	*action*						*goto*		
	a	b	c	d	e	$	A	B	S
0									
1									
2									
3									
4									
5									
6									
7									
8									
9									

$$(I)\ S ::= aABe$$
$$(II)\ A ::= Abc$$
$$(III)\ A ::= b$$
$$(IV)\ B ::= d$$

Example

LR Parse Table

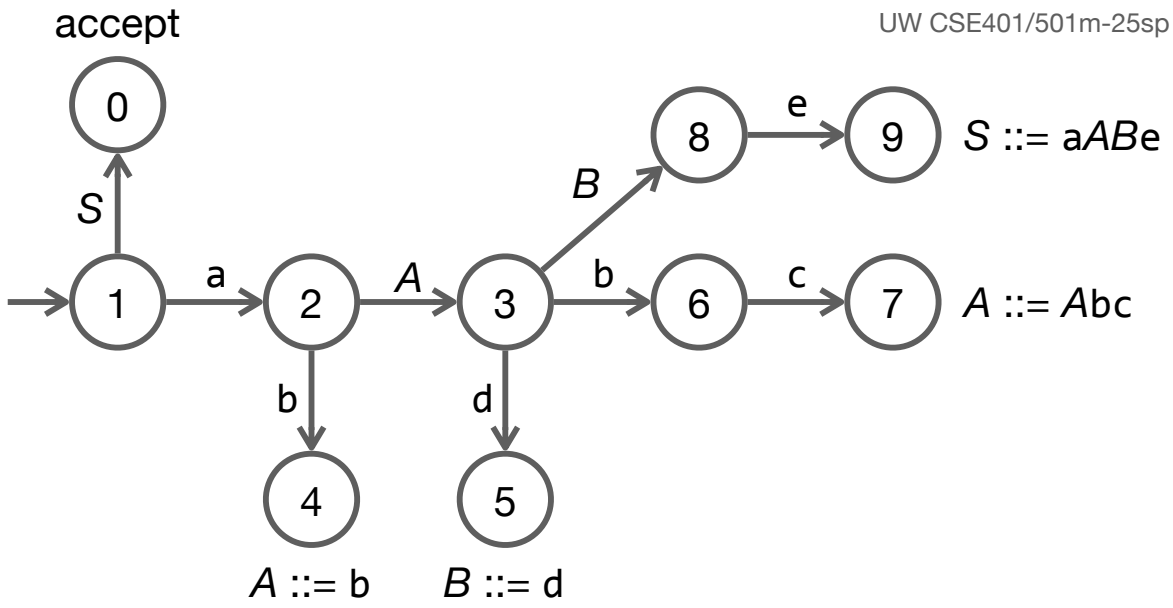


State	action						goto		
	a	b	c	d	e	$	A	B	S
0						acc			
1									
2									
3									
4									
5									
6									
7									
8									
9									

$$(I)\ S ::= aABe$$
$$(II)\ A ::= Abc$$
$$(III)\ A ::= b$$
$$(IV)\ B ::= d$$

Example

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	r-III	r-III	r-III	r-III	r-III	r-III			
5	r-IV	r-IV	r-IV	r-IV	r-IV	r-IV			
6			s7						
7	r-II	r-II	r-II	r-II	r-II	r-II			
8					s9				
9	r-I	r-I	r-I	r-I	r-I	r-I			

$$(I)\ S ::= aABe$$
$$(II)\ A ::= Abc$$
$$(III)\ A ::= b$$
$$(IV)\ B ::= d$$

Lookup: `action[`i`,`X`]` (slide 1)

- **Shift** (sj) — shift input token and state (Xj) onto the stack (advance one token) and then transition to state j
- **Reduce** (r-k) — reduce using grammar production k
  - ✦ note: this can be confusing if productions and states are both integers (common); hence Roman numerals

1. production k ($A ::= \beta$) tells us to pop $2|\beta|$ symbols from the stack (2* for symbol & state)
2. Then read the top state i' from the top of stack
3. Lookup $j' =$ `goto[`i'`,`A`]`, push Aj' onto the stack, and transition to j'

Lookup: `action[`i`,`X`]` (slide 2)

- **accept** — (self explanatory)
- **blank** — no transition — syntax error
  - ✦ LR parsers will detect syntax errors as early as possible
  - ✦ Good parsers ought to produce useful error messages.
    - Doing so requires storing error messages and (potentially) error recovery logic in the action table

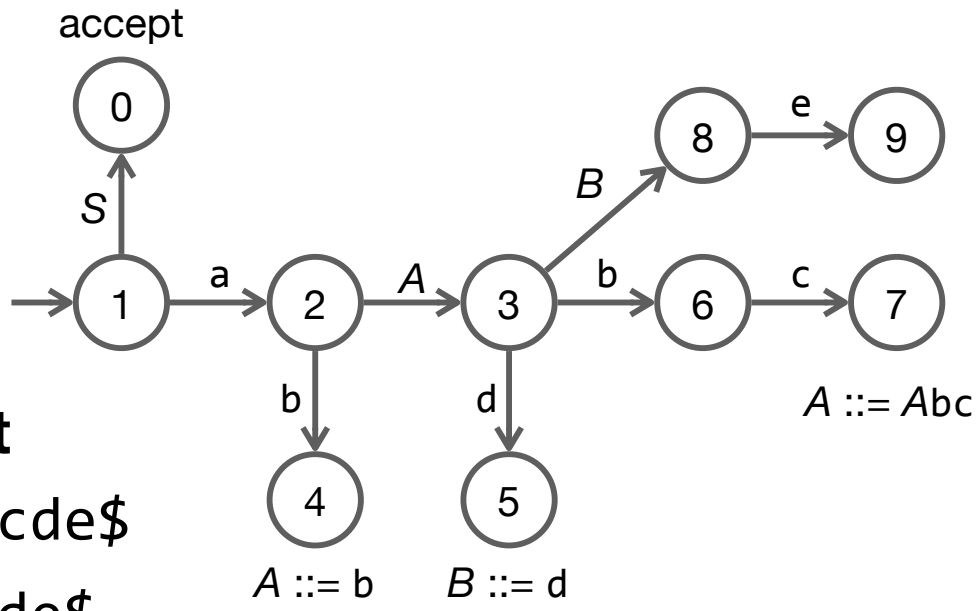
LR Parsing Algorithm (Explicit)

```
X = scanner.getToken();
while (true) {
  i = stack.top();
  act = action[i, X];
  if(act == sj) {
    stack.push(X, j);
    X = scanner.getToken();
  } else if (act == rk) {
    (A ::= β) = production[k];
    stack.pop(2*|β|);
    i = stack.top();
    j = goto[i, A];
    stack.push(A, j);
  }
```

```
else if (act == accept) {
  return;
} else { // blank
  throw SyntaxError;
}
```

Example

LR Table Parse



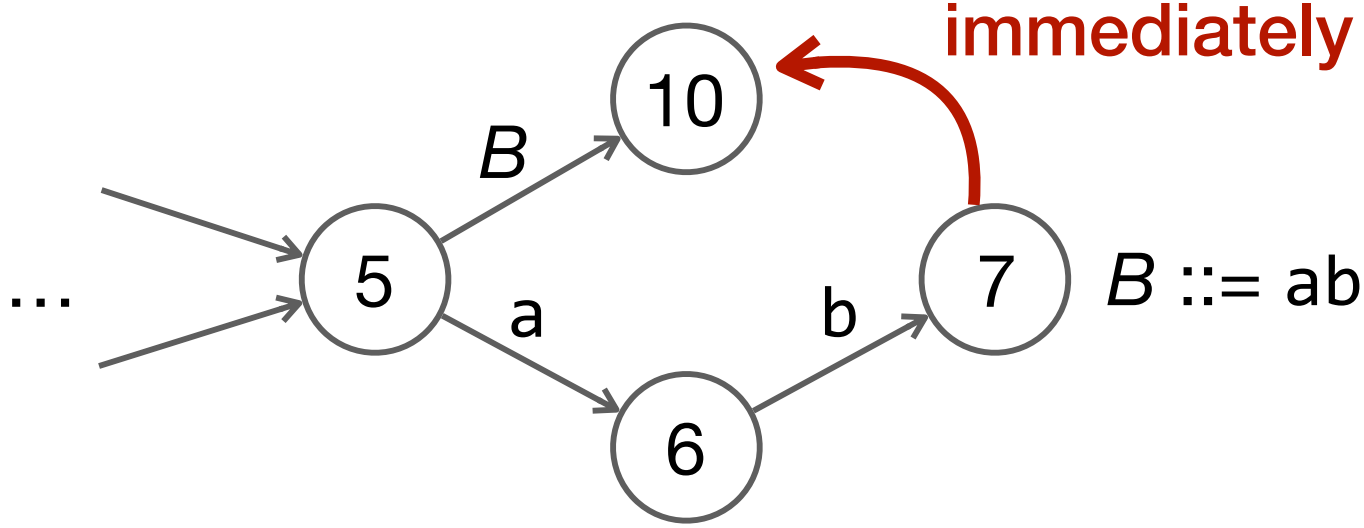
$$(I)\ S ::= aABe$$
$$(II)\ A ::= Abc$$
$$(III)\ A ::= b$$
$$(IV)\ B ::= d$$

Stack	Input
$1	|abbcde$
$1a2	|bbcde$
$1a2b4	|bcde$
$1a2A3	|bcde$
$1a2A3b6	|cde$
$1a2A3b6c7	|de$
$1a2A3	|de$
$1a2A3d5	|e$
$1a2A3B8	|e$
$1a2A3B8e9	|$
$1S0	|$

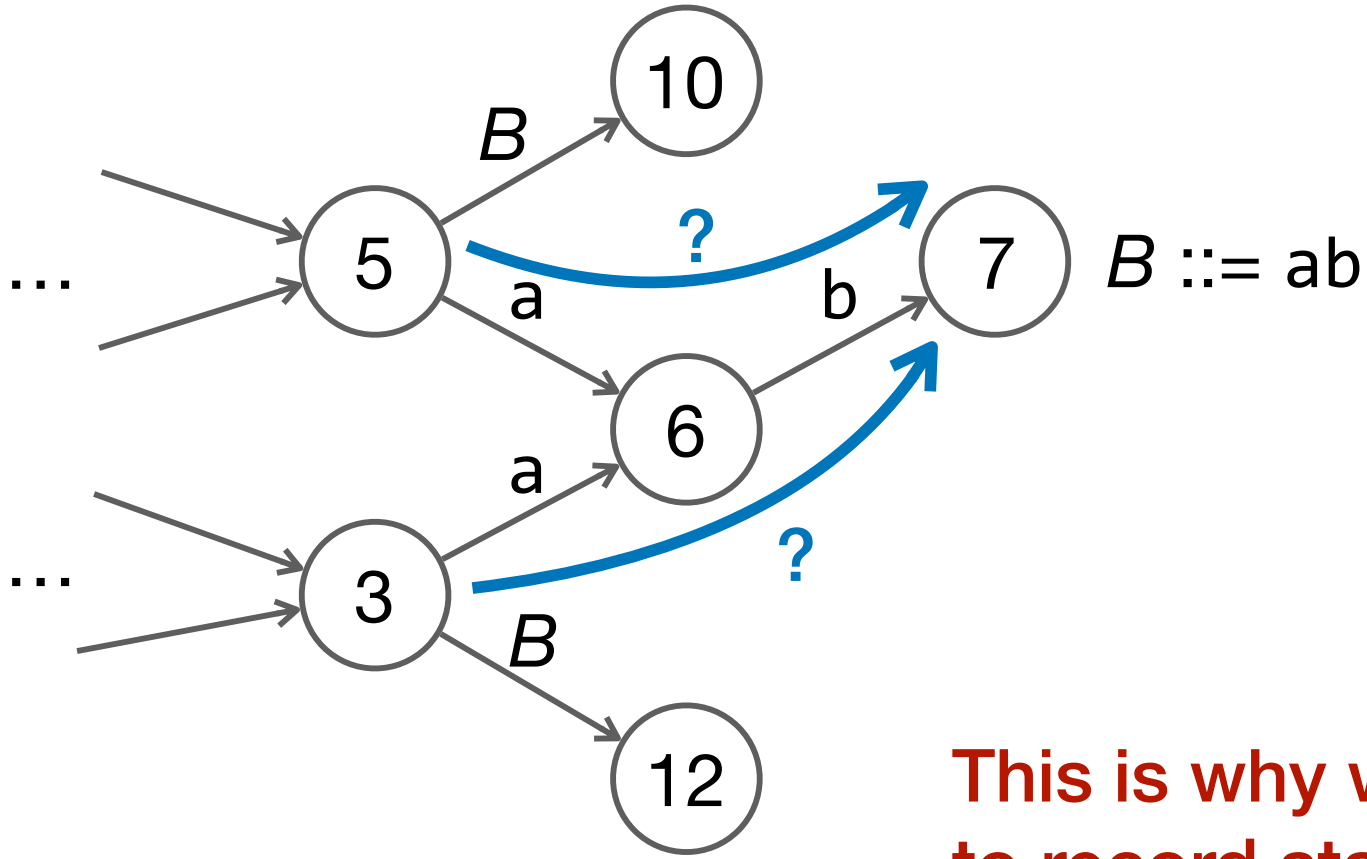
State	*action*						*goto*		
	a	b	c	d	e	$	A	B	S
0						acc			
1	s2								g0
2		s4					g3		
3		s6		s5				g8	
4	r-III	r-III	r-III	r-III	r-III	r-III			
5	r-IV	r-IV	r-IV	r-IV	r-IV	r-IV			
6			s7						
7	r-II	r-II	r-II	r-II	r-II	r-II			
8					s9				
9	r-I	r-I	r-I	r-I	r-I	r-I			

Do We Need the Goto Table?

Why can't the reduce action just jump immediately to here?



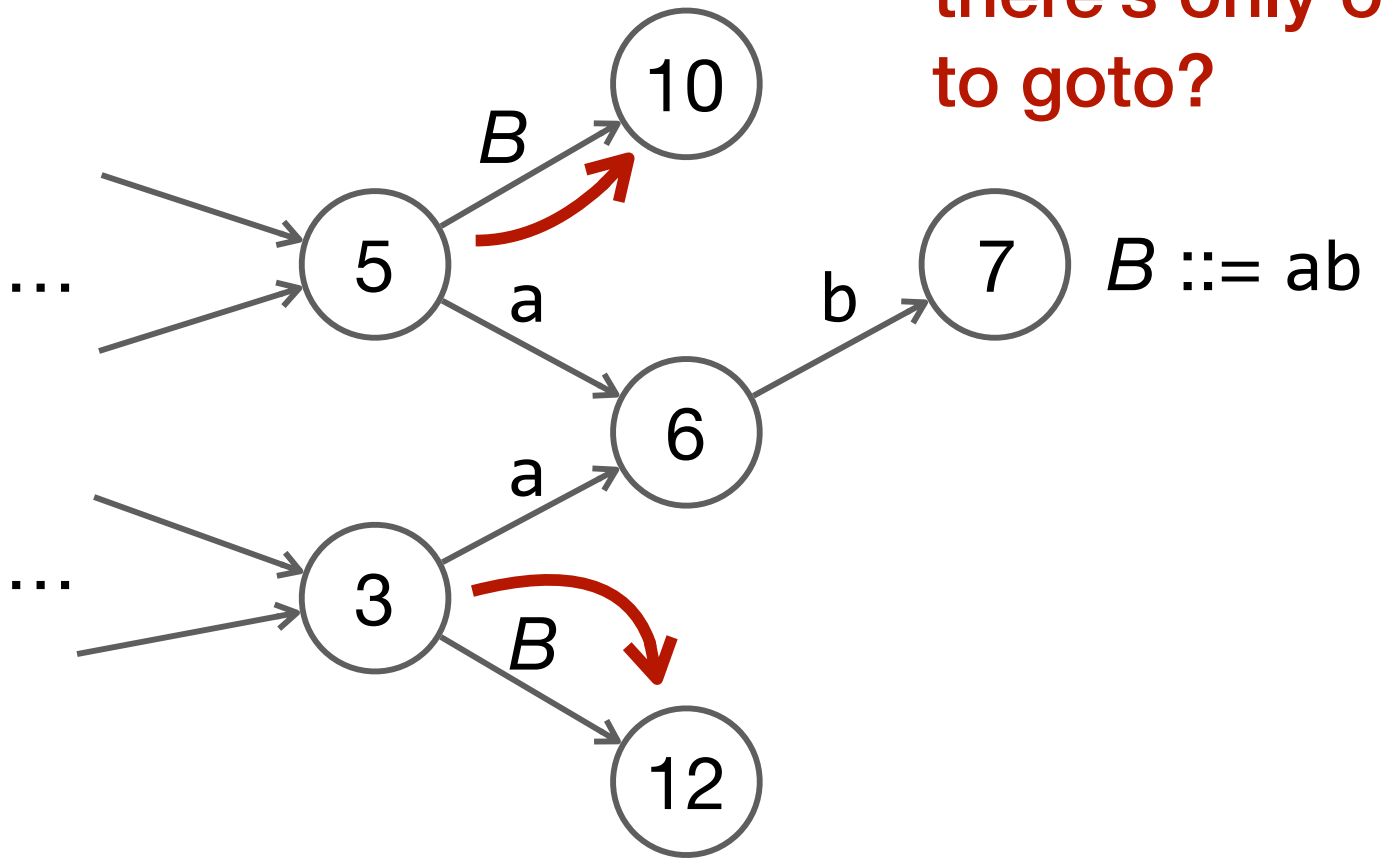
Do We Need the Goto Table?



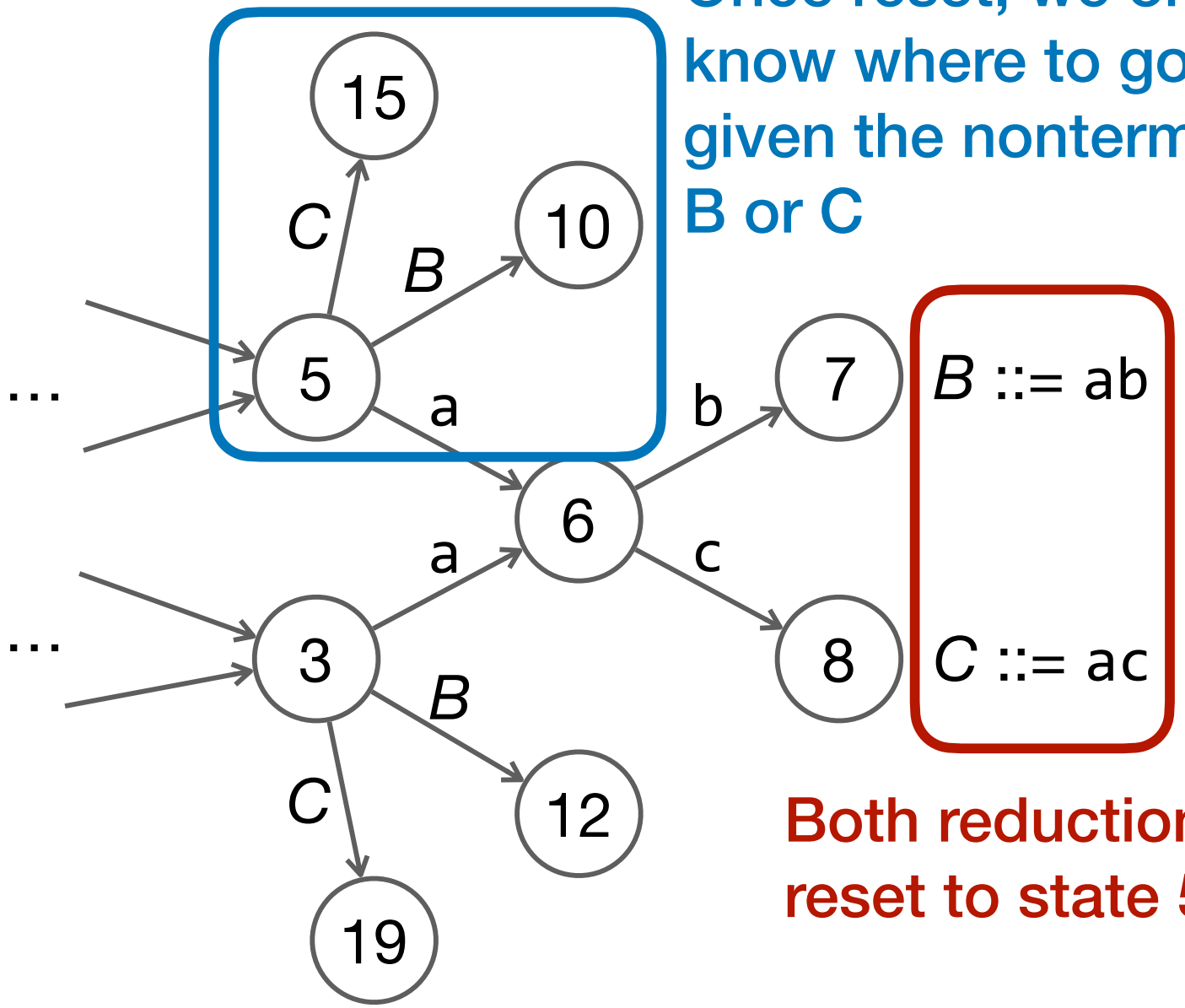
This is why we need to record states on the stack

Do We Need the Goto Table?

Ok, but why do we need the goto table if there's only one place to goto?



Do We Need the Goto Table?



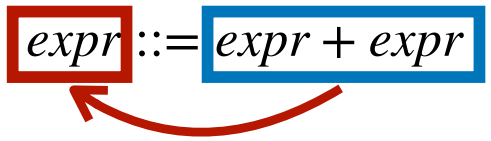
LR Parsing Recap

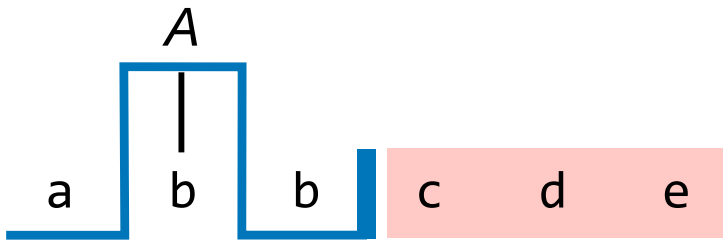
"Bottom-up" Parsing — match **right-hand sides**

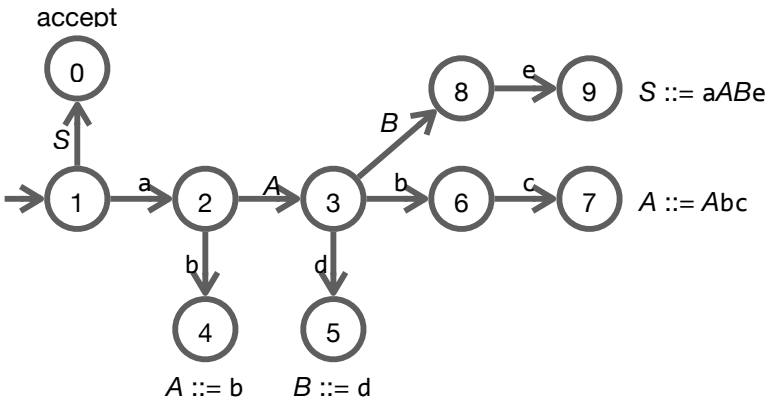
Doing this while scanning **left-to-right** produces a "frontier" (i.e. the stack)

Deciding when to **shift** vs. **reduce** can be decided via a DFA that recognizes valid prefixes

This DFA can be encoded into an **LR 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	r-IV	r-IV	r-IV	r-IV	r-IV	r-IV			
5	r-IV	r-IV	r-IV	r-IV	r-IV	r-IV			
6			s7						
7	r-II	r-II	r-II	r-II	r-II	r-II			
8					s9				
9	r-I	r-I	r-I	r-I	r-I	r-I			

Outline

LR Parsing

Automating Parsing

Table-driven Parsers

LR States

Shift-Reduce & Reduce-Reduce Conflicts

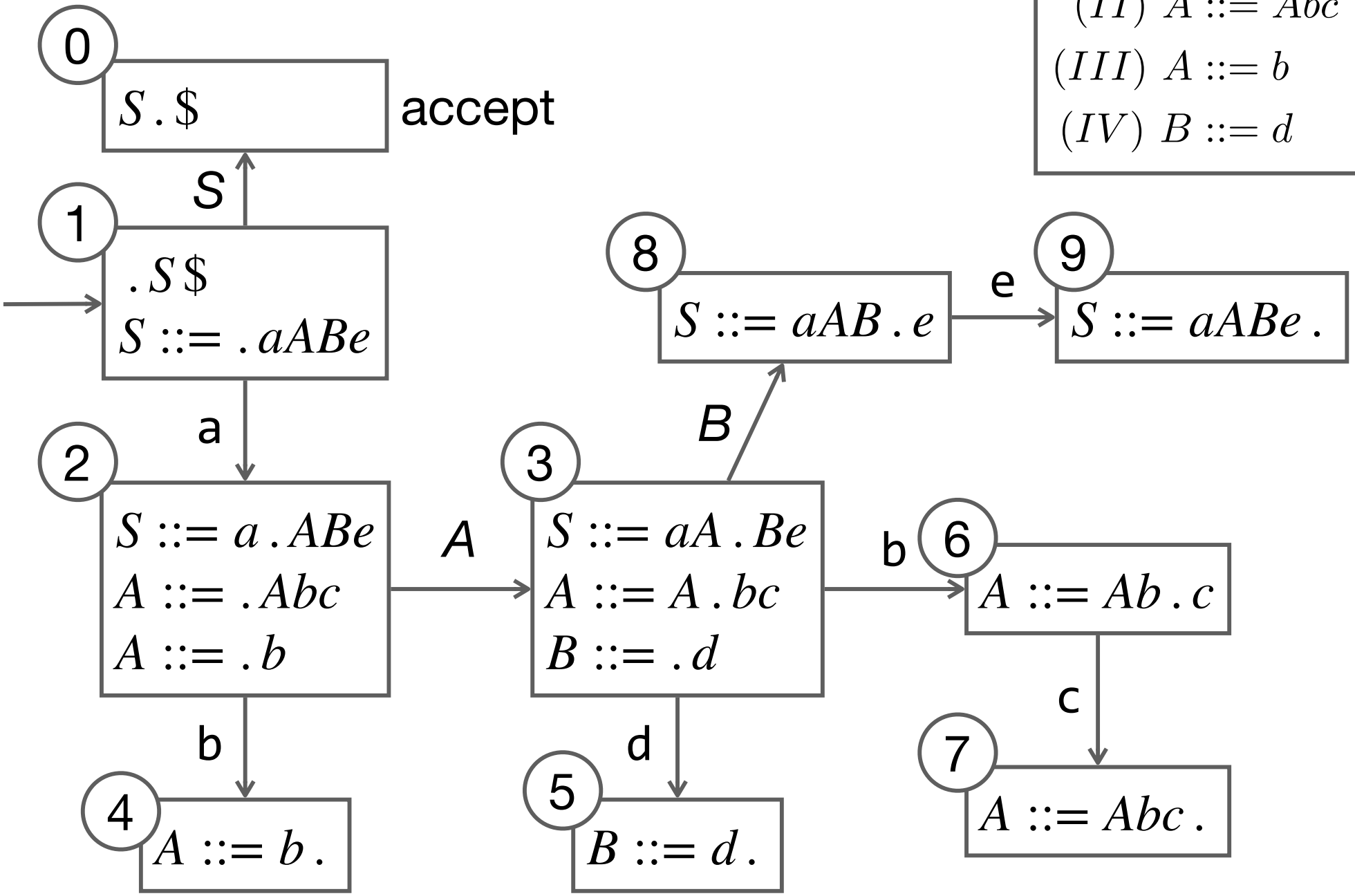
LR States & Items

- Basic Idea — each state encodes
  - The set of productions that we might be in the middle of matching against
  - ***Where*** exactly we are ***in the middle*** of each such potential match (see below)
- Realization of the Idea — "Sets of **Items**"
  - An **item** is a production with a dot in its right-hand side
  - e.g. the production $A ::= XY$ has 3 items

$$A ::= .XY \qquad A ::= X.Y \qquad A ::= XY.$$

DFA with Items for

$(I)\ S ::= aABe$
$(II)\ A ::= Abc$
$(III)\ A ::= b$
$(IV)\ B ::= d$



Outline

LR Parsing

Automating Parsing

Table-driven Parsers

LR States

Shift-Reduce & Reduce-Reduce Conflicts

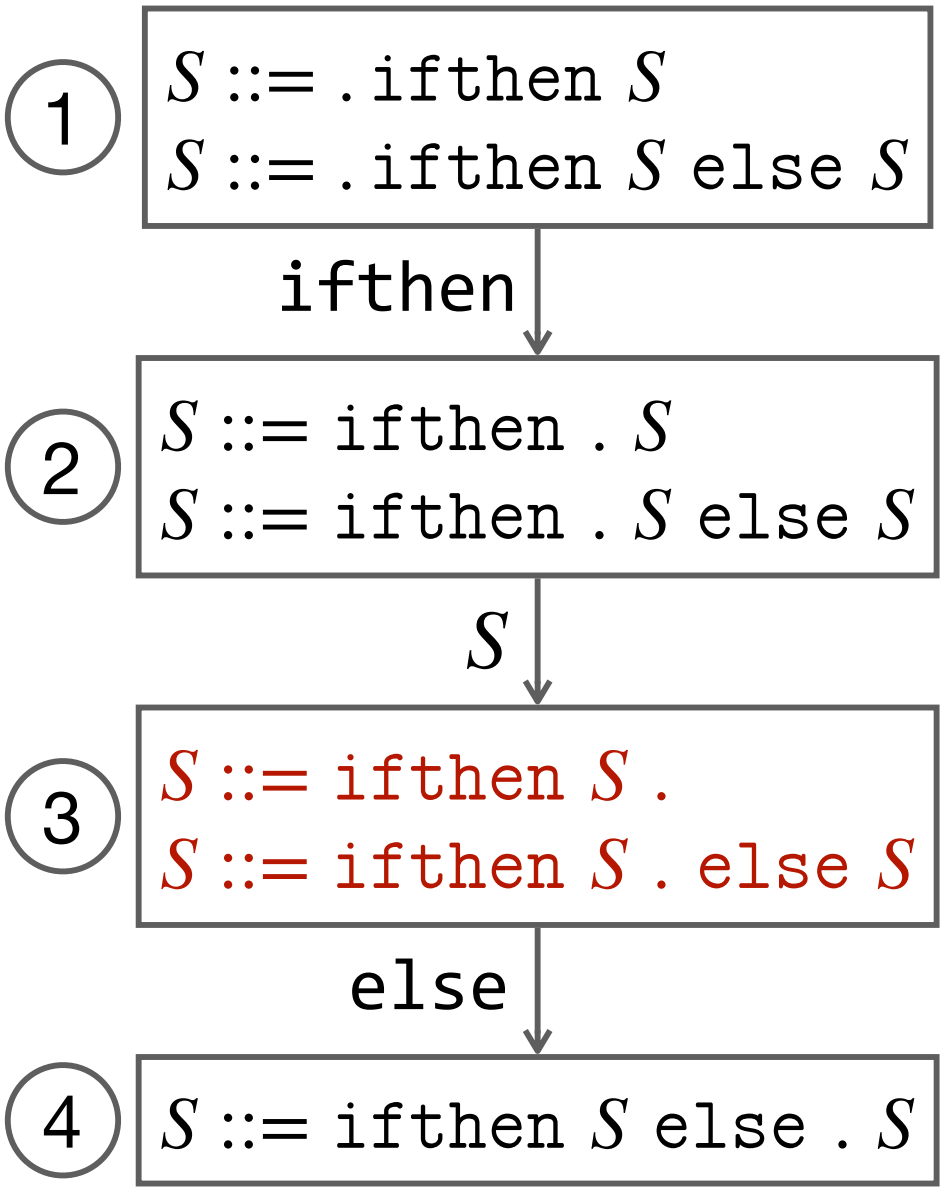
Problems with Grammars

- Previous grammar/DFA was LR(0)
- If the grammar is not LR (of specified variant) then we will encounter problems when constructing an LR parser
  - ✦ Shift-Reduce Conflicts
  - ✦ Reduce-reduce Conflicts
- Both conflicts are situations where two (or more) different actions are called for
- Note: an unambiguous grammar may still have conflicts
  - ✦ however, conflict-free grammars are unambiguous

Shift-Reduce Conflicts

- These happen when both a shift and a reduce are possible at a given point in the parse (equivalently: in a particular state of the DFA)
- A classic example: the “ambiguous else” problem
  - *S* ::= `if then` *S* | `if then` *S* `else` *S*

Example: Shift-Reduce Conflict



Grammar

(I) $S ::= \texttt{ifthen}\ S$

(II) $S ::= \texttt{ifthen}\ S\ \texttt{else}\ S$

- **State 3** has a shift-reduce conflict
  - Could shift past else into state 4 (s4)
  - ***or*** could reduce (r-I)

$$S ::= \texttt{ifthen}\ S$$

note: other items are not included in states 2-4 to save space

Solving Shift-Reduce Conflicts

- Option 1 — Fix the grammar
  - ✦ Done in the Java reference grammar and in many others
- Option 2 — Use a parser generator that has a *longest match* heuristic to systematically resolve shift-reduce conflicts in favor of shift over reduce
  - ✦ This does the right thing for the if-else case
  - ✦ Guidelines — make sure to check that this behavior is what you want if you're going to rely on it. Still not ideal to rely on this behavior.

Reduce-Reduce Conflicts

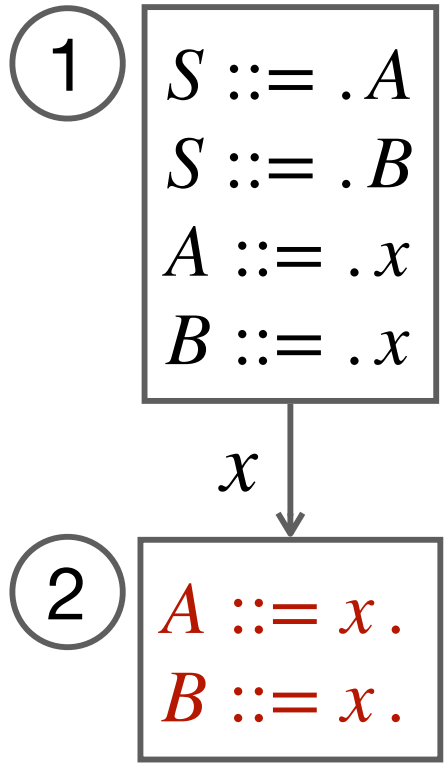
- Problem: two different reductions are possible from a given state
- Contrived Example

(I) $S ::= A$

(II) $S ::= B$

(III) $A ::= x$

(IV) $B ::= x$



- **State 2** has a reduce-reduce conflict (r-II vs. r-IV)

Solving 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
    - Main generator tools (YACC, Bison, CUP, etc.) do this
  - ✦ Fix the grammar

A Real Reduce-Reduce Conflict

- Suppose the grammar tries to separate arithmetic and boolean expressions, but still use variables

$$
\begin{aligned}
expr &::= aexpr \mid bexpr \\
aexpr &::= aexpr * aident \mid aident \\
bexpr &::= bexpr \texttt{ \&\& } bident \mid bident \\
aident &::= id \\
bident &::= id
\end{aligned}
$$

- This will create a reduce-reduce conflict state with at least the items $\{\ aident ::= id\ .\ ,\ \ bident ::= id\ .\ \}$

Covering Grammars

- One solution — Merge $aident$ and $bident$ into a single non-terminal (e.g. use id everywhere in place of these)

$$expr ::= aexpr * id \mid bexpr \texttt{ \&\& } id \mid id$$
$$aexpr ::= aexpr * id \mid id$$
$$bexpr ::= bexpr \texttt{ \&\& } id \mid id$$

- This is a ***covering grammar***
  - ✦ May generate some strings that are not generated by the original grammar; or less than ideal parse trees
  - ✦ Filter out / disambiguate programs at a later stage (e.g. determine type of each id encountered)

Next Time…

- Constructing LR Tables
  - ✦ We'll do a simple version of LR(0) in lecture, and then talk about extending it to LR(1), relation to SLR and LALR as used in most parser generators — basic ideas are very similar across all variants
- After that — LL parsers and recursive descent
- Continue reading chapter 3 to prepare for parsing this week (3.4 & 3.5) (3.6 is optional)