CSE 401/M501 – Compilers

LR Parsing

Hal Perkins

Fall 2023

Administrivia (added Fri.)

- HW1 due last night, but still have a late day or two if you need it (but try to save them)
- Project:
  - Scanner due Thursday night, but please shake down infrastructure well before then
    - DO NOT start on the parser yet – just edit token classes in the .cup file (and any other small edits there needed to get a clean build)
  - If you're still looking for a partner / need a project repo set up and haven't contacted us yet, send email to cse401-staff@cs
- HW2: LR parsing and grammars – due in 2 weeks, but lectures aren't quite far enough along. Will post when we get enough background, probably Monday.

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

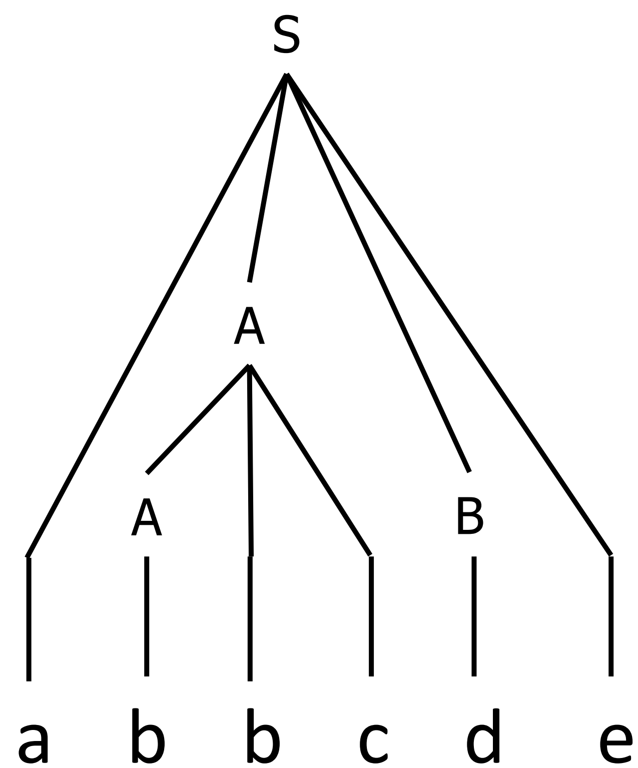
- Grammar

  S ::= aABe

  A ::= Abc | 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 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.

LR Parsing in Greek

- The bottom-up parser reconstructs a reverse rightmost derivation
- Given the rightmost derivation

  $S \Rightarrow \beta_1 \Rightarrow \beta_2 \Rightarrow \ldots \Rightarrow \beta_{n-2} \Rightarrow \beta_{n-1} \Rightarrow \beta_n = w$

  the parser will first discover $\beta_{n-1} \Rightarrow \beta_n$, then $\beta_{n-2} \Rightarrow \beta_{n-1}$, etc.
- Parsing terminates when
  - β_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=>β if both of these hold:
  - A=>β is a valid production
  - A=>β is a step in *this* rightmost derivation that produced this input string
- This is known as a *shift-reduce* parser

Sentential Forms

- If S =>* α, the string α is called a *sentential form* of the grammar
- In the derivation
  $S \Rightarrow \beta_1 \Rightarrow \beta_2 \Rightarrow \ldots \Rightarrow \beta_{n-2} \Rightarrow \beta_{n-1} \Rightarrow \beta_n = w$
  each of the β_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 β of a production *that is part of the rightmost derivation of the current input string*
  - Even if A::=β is a production, it is a handle only if β matches the parse tree frontier at a point where A::=β was used in *this particular* derivation
  - β 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 => a*AB*e => a*A*de => a*A*bcde => abbcde

  – abbcde 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*::=*A*bc at position 4

    - Note: some books take the left end of the match as the position

Handles Defined

- Formally, a *handle* of a right-sentential form γ is a production A ::= β and a position in γ where β may be replaced by A to produce the previous right-sentential form in the rightmost derivation of γ
  – Some sources use "handle" to refer only to the right-hand side β and its position. Others mean the entire production A::=β. 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
- Also need to encode the rules that tell us what action to take 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 β and push the left side A
- *Accept* – announce success
- *Error* – syntax error discovered

Shift-Reduce Example

S ::= aABe
A ::= Abc | b
B ::= d

Stack	Input	Action
$	abbcde$	*shift*
$a	bbcde$	*shift*
$ab	bcde$	*reduce*
$aA	bcde$	*shift*
$aAb	cde$	*shift*
$aAbc	de$	*reduce*
$aA	de$	*shift*
$aAd	e$	*reduce*
$aAB	e$	*shift*
$aABe	$	*reduce*
$S	$	*accept*

How Do We Automate This?

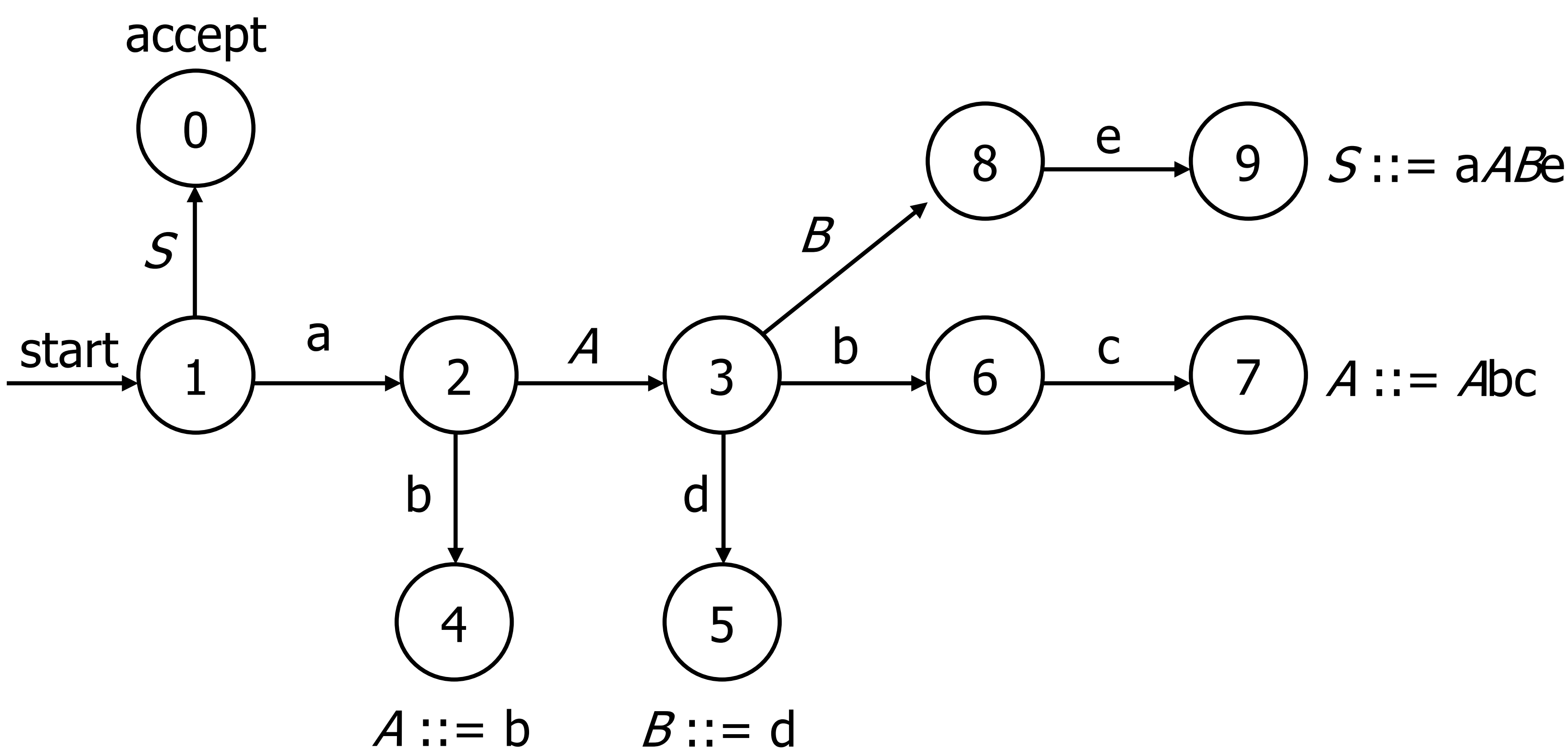
- Cannot use clairvoyance in a real parser (alas…)
- Defn. *Viable prefix* – a prefix of any right-sentential form that can appear on the stack of the shift-reduce parser
  - Equivalent: a prefix of a right-sentential form that does not continue past the rightmost handle of that sentential form
  - In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

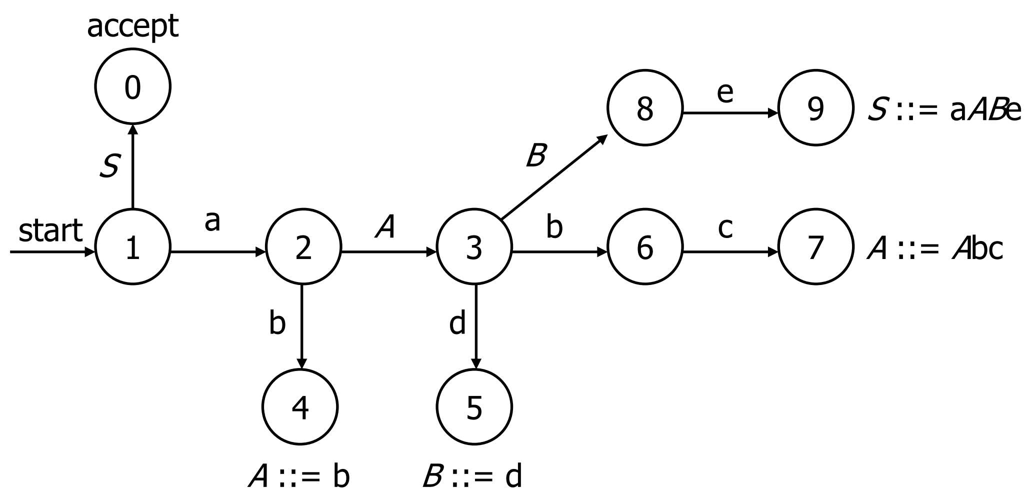
S ::= aABe
A ::= Abc | b
B ::= d



Trace

S ::= a*AB*e
A ::= *A*bc | 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
  - We'll 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 rhs of the production

  $\therefore$ 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

  $\$s_0\ X_1\ s_1\ X_2\ s_2\ \ldots\ X_n\ s_n$
  - 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 FA 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 and 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 tells us how many <symbol, state> 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 ::= α | β needs to be split into A ::= α and A ::= β

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

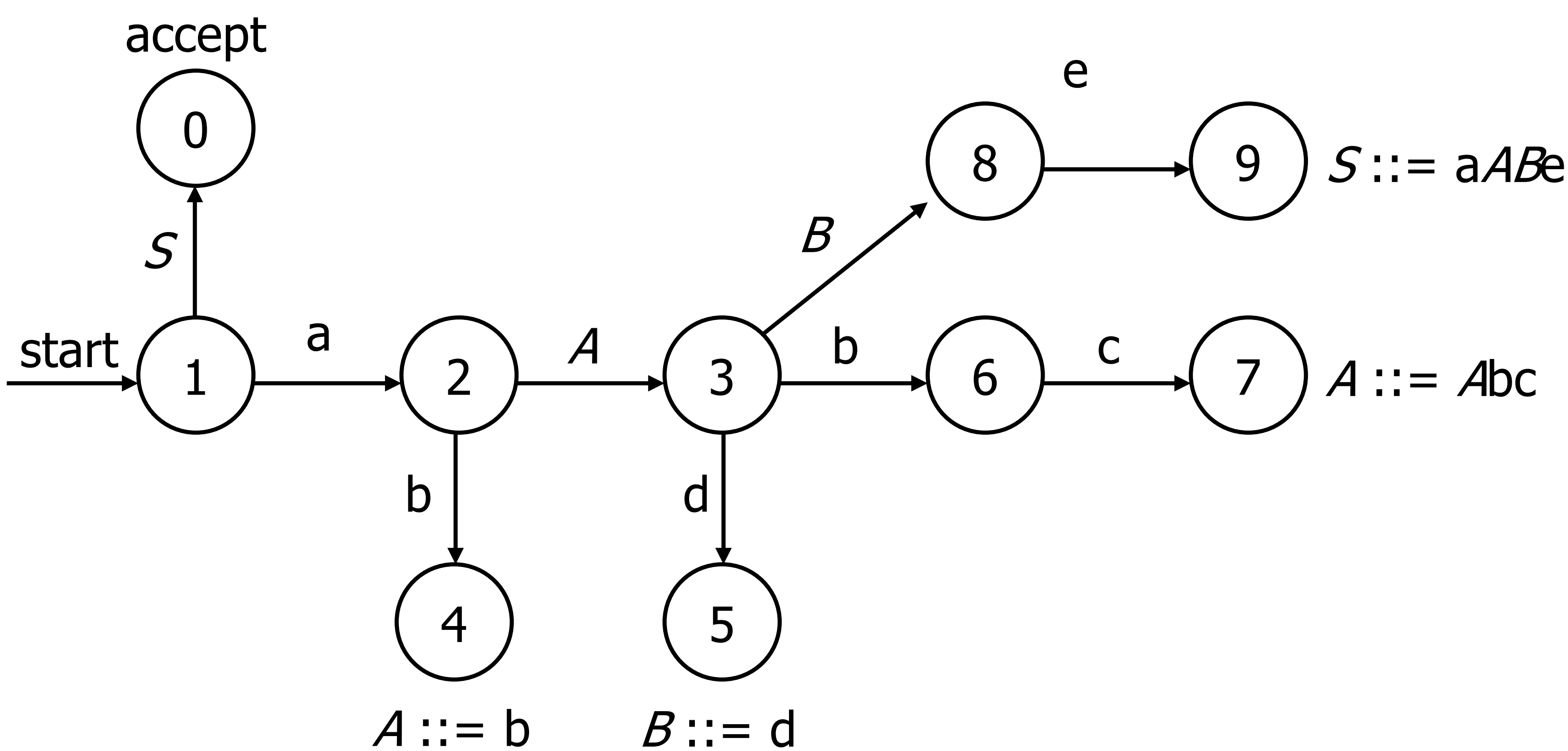
- When a reduction is performed using A ::= β, we pop |β| <symbol, state> pairs from the stack revealing a state *uncovered_s* on the top of the stack
- goto[*uncovered_s* , *A*] is the new state to push on the stack when reducing production *A* ::= β (after popping handle β and pushing A)

Aside: Extra Initial Production

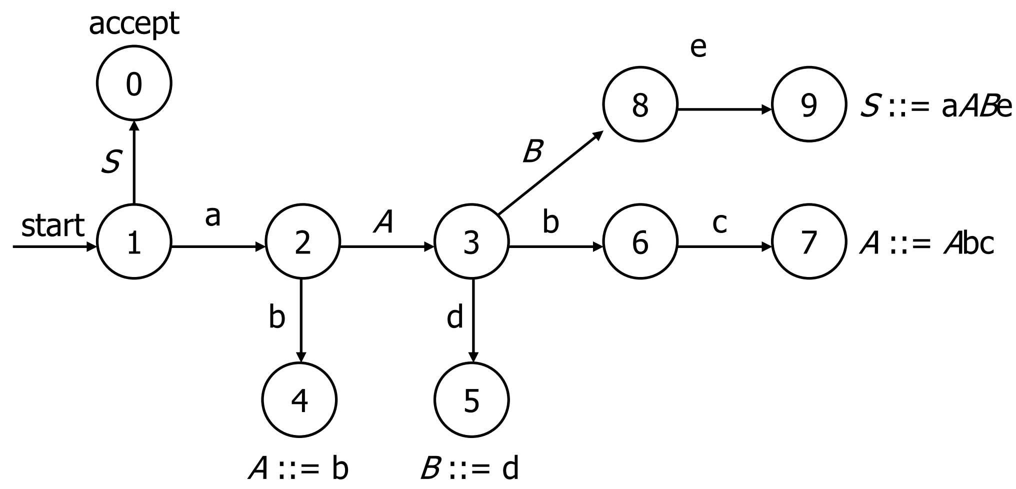
- When we construct the DFA we'll need to add a new production to handle end-of-file (i.e., end-of-input) correctly
- If *S* is the start state of the original grammar, add an initial production *S'* ::= *S* $
  - $ represents end-of-file (input)
  - Accept when we've reduced the input to *S* and there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

Reminder: DFA for

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

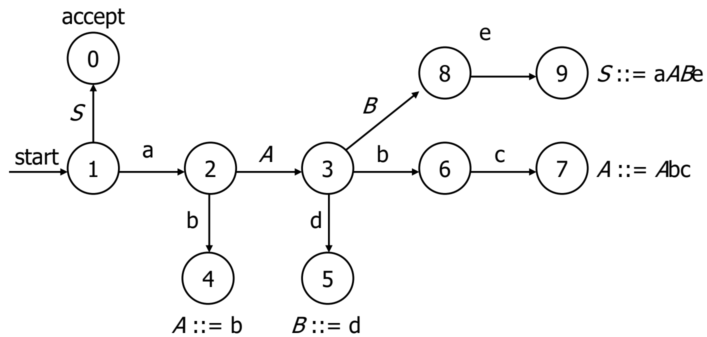
0. *S′* ::= *S* $
1. *S* ::= a*AB*e
2. *A* ::= *A*bc
3. *A* ::= b
4. *B* ::= d

LR Parsing Algorithm

```
word = scanner.getToken();
while (true) {
    s = top of stack;
    if (action[s, word] = si ) {
      push word; push i  (state);
      word = scanner.getToken();
    } else if (action[s, word] = rj ) {
      pop 2 * length of right side of
      production j  (2*|β|);
      uncovered_s = top of stack;
      push left side A of production j ;
      push state goto[uncovered_s, A];
    }
```

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

Example

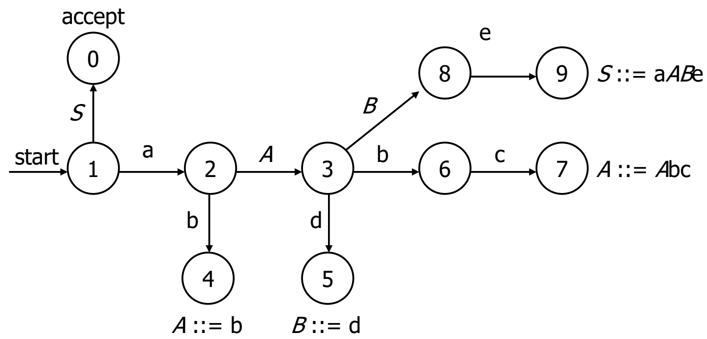


0. $S' ::= S\,\$$
1. $S ::= \mathrm{a}AB\mathrm{e}$
2. $A ::= A\mathrm{bc}$
3. $A ::= \mathrm{b}$
4. $B ::= \mathrm{d}$

Stack	Input
\$1	abbcde\$

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			

Example



0. $S' ::= S\,\$$
1. $S ::= \mathrm{a}A B\mathrm{e}$
2. $A ::= A\mathrm{bc}$
3. $A ::= \mathrm{b}$
4. $B ::= \mathrm{d}$

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

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			

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

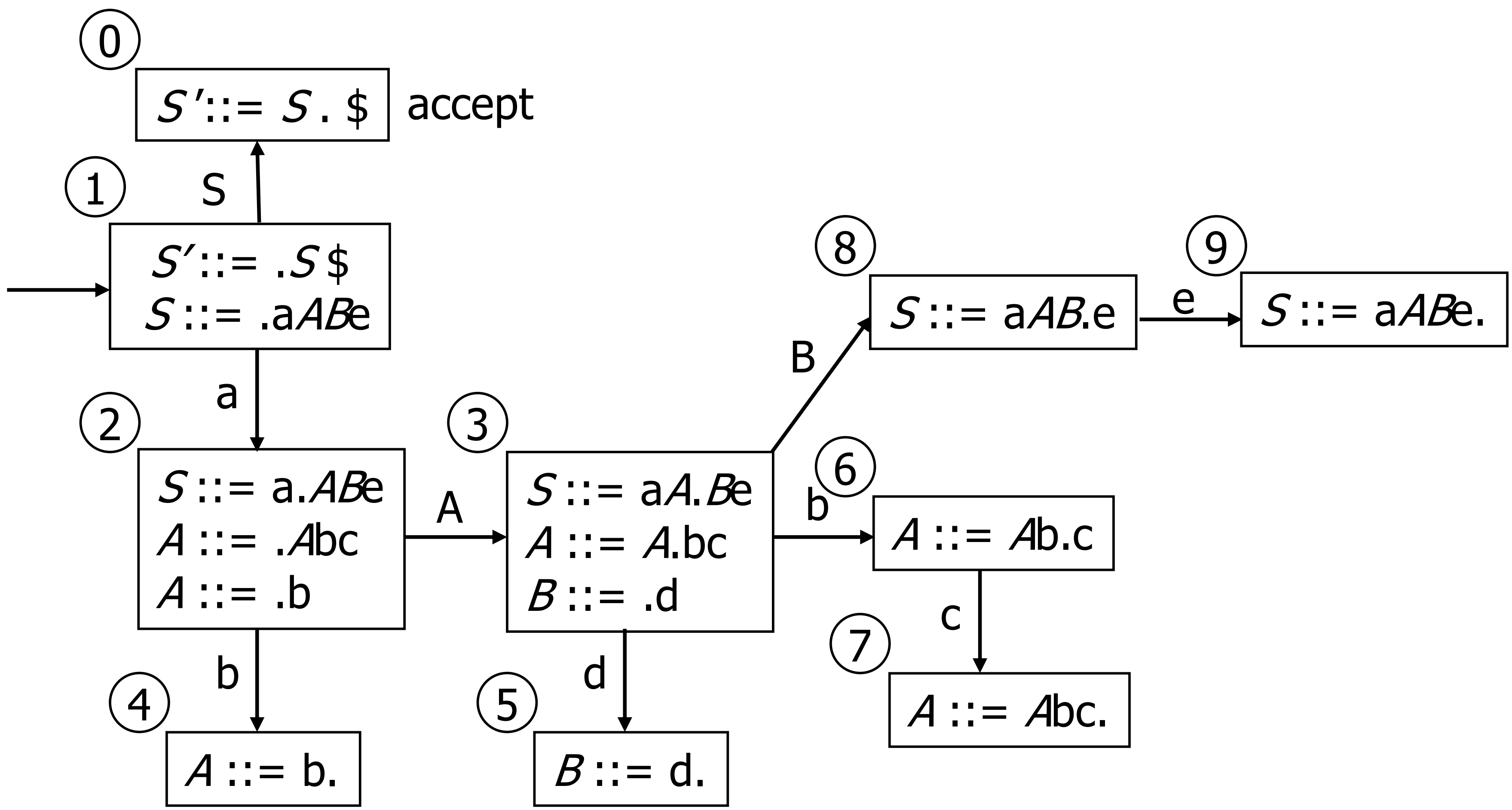
- An *item* is a production with a dot in the right hand side
- Example: Items for production $A ::= X\ Y$

  $A ::= .\ X\ Y$

  $A ::= X\ .\ Y$

  $A ::= X\ Y\ .$
- Idea: The dot represents a position in the production – partial match to rhs

DFA for $S' ::= S\,\$$ / $S ::= \text{a}AB\text{e}$ / $A ::= A\text{bc}$ / $A ::= \text{b}$ / $B ::= \text{d}$



Problems with Grammars

- Non-LR grammars cause problems when constructing an LR parser (that's how you know 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 ::= ifthen S | ifthen S else S

Parser States for

1. S ::= ifthen S
2. S ::= ifthen S else S

① S ::= . ifthen S
S ::= . ifthen S else S

$\xrightarrow{\text{ifthen}}$

② S ::= ifthen . S
S ::= ifthen . S else S

$\xrightarrow{S}$

③ S ::= ifthen S .
S ::= ifthen S . else S

$\xrightarrow{\text{else}}$

④ S ::= ifthen S else . S

- State 3 has a shift-reduce conflict
  - Can shift past else into state 4 (s4)
  - Can reduce (r1)

    S ::= ifthen S

(Note: other S ::= . 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

$S ::= A$

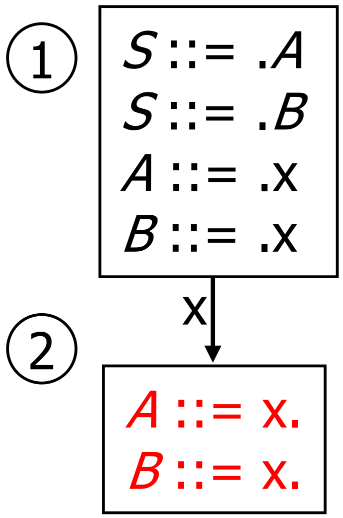
$S ::= B$

$A ::= \mathrm{x}$

$B ::= \mathrm{x}$

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 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
- LL parsers and recursive descent
- Continue reading ch. 3