CSE 401 – Compilers

Lecture 7: LR Parsing (part II)

Michael Ringenburg

Winter 2013

Reminders/ Announcements

- Project part 1 is due on Monday.
- Part 2 will be assigned early next week.
  - Will be due 2 weeks after it is assigned.
- Will also assign homework 2 (parsing) next week.
  - Will be due 1 week after it is assigned.
- Midterm in class on February 15.

Review From Last Week

aABe
aAde
abde
aAbcde
abbcde
aAbcbcde
abbcbcde
…

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

- Right-sentential form: α is a right-sentential form of the grammar G=<N,Σ,P,S> if S $=>_{rm}^*$ α

Review From Last Week

aABe
aA**d**e
a**b**de
a**Abc**de
a**b**bcde
a**Abc**bcde
a**b**bcbcde
…

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

Bold red: Handle

- Handle: The handle of a right-sentential form is the substring corresponding to the right hand side of the production that produced it from the previous step in the rightmost derivation.

Review From Last Week

aABe	a, aA, aAB, aABe
aA**d**e	a, aA, aAd
a**b**de	a, ab
a**Abc**de	a, aA, aAb, aAbc
a**b**bcde	a, ab
a**Abc**bcde	a, aA, aAb, aAbc
a**b**bcbcde	a, ab
…	

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

Bold red: Handle

- Viable prefix: a prefix of a right-sentential form that does not continue past the rightmost handle of that sentential form.

Review From Last Week

aABe	a, aA, aAB, aABe
aA**d**e	a, aA, aAd
a**b**de	a, ab
a**Abc**de	a, aA, aAb, aAbc
a**b**bcde	a, ab
a**Abc**bcde	a, aA, aAb, aAbc
a**b**bcbcde	a, ab
…	

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

Bold red: Handle

- Viable prefix: a prefix of a right-sentential form that does not continue past the rightmost handle of that sentential form.

Review From Last Week

aABe
aA**d**e
a**b**de
a**Abc**de
a**b**bcde
a**Abc**bcde
a**b**bcbcde
…

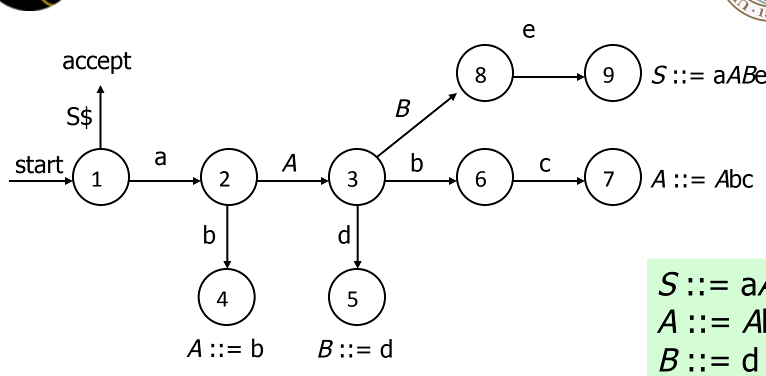
a, aA, aAB, aABe,
aAd,
ab,
aAb, aAbc

S ::= a*AB*e
A ::= *A*bc | b
B ::= d

Bold red: Handle

- Viable prefix: a prefix of a right-sentential form that does not continue past the rightmost handle of that sentential form.

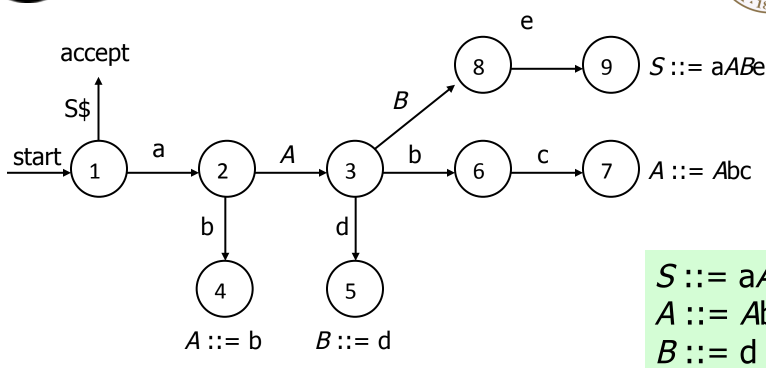
Review From Last Week



S ::= a*AB*e
A ::= *A*bc | b
B ::= d

- Viable prefixes and handles of a CFG are a regular language, thus can recognize with a DFA.

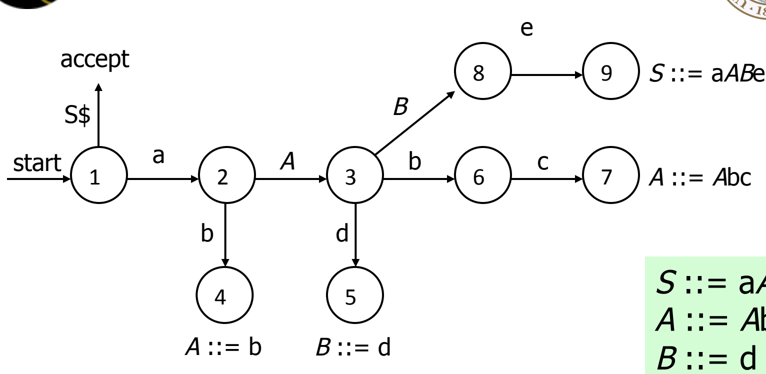
Review From Last Week



S ::= a*AB*e
A ::= *A*bc | b
B ::= d

- Basic idea: Reduce by handle when we reach state corresponding to viable prefix that goes all the way to the end of a handle. Otherwise, shift.

Review From Last Week



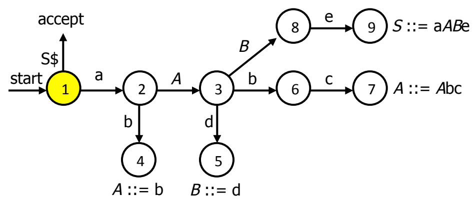
S ::= a*AB*e
A ::= *A*bc | b
B ::= d

- But, recall from last week that this involved a lot of DFA transitions at every step – *not* O(n).

Trace

S ::= a*AB*e
A ::= *A*bc | b
B ::= d

Stack	Input
$	abbcde$
$a	bbcde$
$ab	bcde$
$aA	bcde$

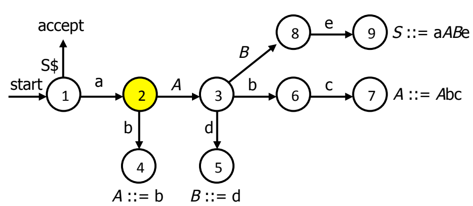


- Consider what happens before and after a shift …

Trace

S ::= a*AB*e
A ::= *A*bc | b
B ::= d

Stack	Input
$	abbcde$
$a	bbcde$
$ab	bcde$
$aA	bcde$

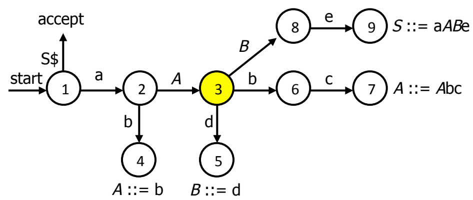


- Consider what happens before and after a shift …

Trace

S ::= a*AB*e
A ::= *A*bc | b
B ::= d

Stack	Input
$	abbcde$
$a	bbcde$
$ab	bcde$
$aA	bcde$

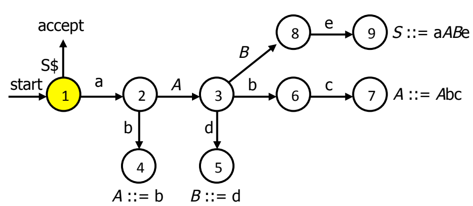


- Consider what happens before and after a shift …

Trace

S ::= a*AB*e
A ::= *A*bc | b
B ::= d

Stack	Input
$	abbcde$
$a	bbcde$
$ab	bcde$
$aA	bcde$
$aAb	cde$

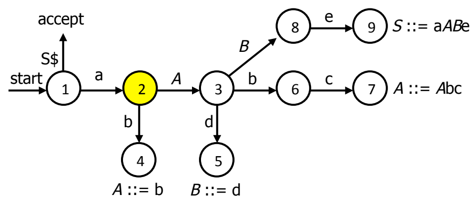


- Consider what happens before and after a shift …

Trace

S ::= a*AB*e
A ::= *A*bc | b
B ::= d

Stack	Input
$	abbcde$
$a	bbcde$
$ab	bcde$
$aA	bcde$
$aAb	cde$

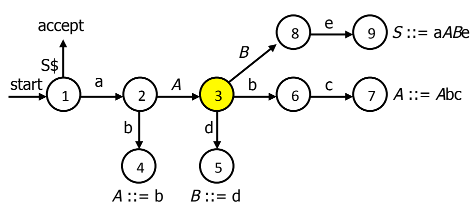


- Consider what happens before and after a shift …

Trace

S ::= a*AB*e
A ::= *A*bc | b
B ::= d

Stack	Input
$	abbcde$
$a	bbcde$
$ab	bcde$
$aA	bcde$
$aAb	cde$

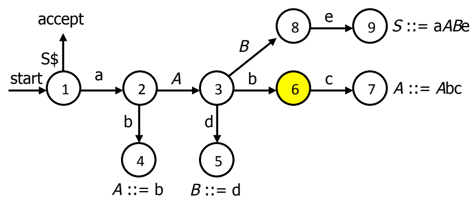


- Consider what happens before and after a shift …
- Repeat all of the states from before the shift, and then make one more transition.

Trace

S ::= a*AB*e
A ::= *A*bc | b
B ::= d

Stack	Input
$	abbcde$
$a	bbcde$
$ab	bcde$
$aA	bcde$
$aAb	cde$

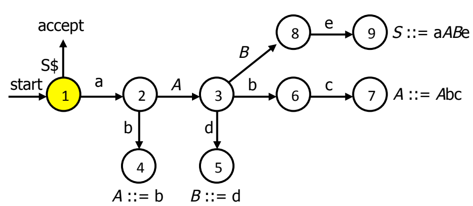


- Consider what happens before and after a shift …
- Repeat all of the states from before the shift, and then make one more transition.

Trace

S ::= a*AB*e
A ::= *A*bc | b
B ::= d

Stack	Input
$	abbcde$
$a	bbcde$
$ab	bcde$
$aA	bcde$
$aAb	cde$
$aAbc	de$
$aA	de$
$aAd	e$

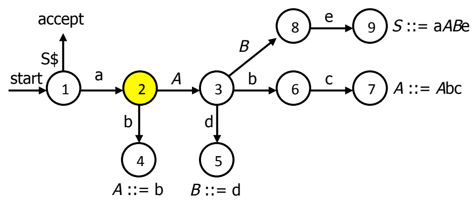


- Consider what happens before and after a reduce…

Trace

S ::= a*AB*e
A ::= *A*bc | b
B ::= d

Stack	Input
$	abbcde$
$a	bbcde$
$ab	bcde$
$aA	bcde$
$aAb	cde$
$aAbc	de$
$aA	de$
$aAd	e$

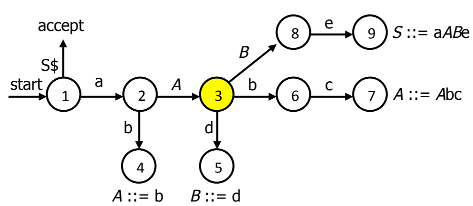


- Consider what happens before and after a reduce…

Trace

S ::= a*AB*e
A ::= *A*bc | b
B ::= d

Stack	Input
$	abbcde$
$a	bbcde$
$ab	bcde$
$aA	bcde$
$aAb	cde$
$aAbc	de$
$aA	de$
$aAd	e$

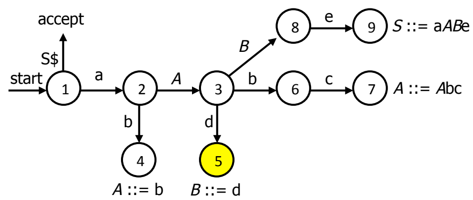


- Consider what happens before and after a reduce…

Trace

S ::= a*AB*e
A ::= *A*bc | b
B ::= d

Stack	Input
$	abbcde$
$a	bbcde$
$ab	bcde$
$aA	bcde$
$aAb	cde$
$aAbc	de$
$aA	de$
$aAd	e$

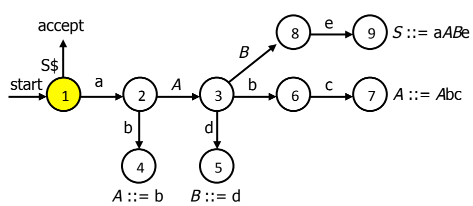


- Consider what happens before and after a reduce…

Trace

S ::= a*AB*e
A ::= *A*bc | b
B ::= d

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

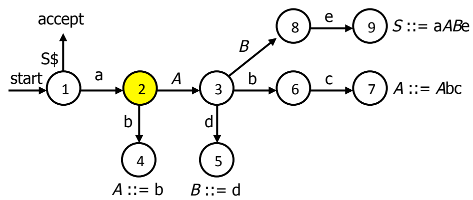


- Pop the handle off the stack

Trace

S ::= a*AB*e
A ::= *A*bc | b
B ::= d

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

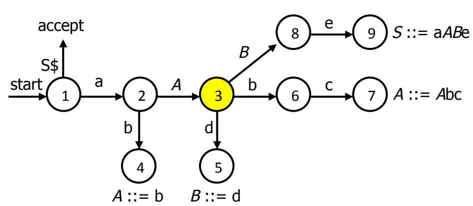


- Pop the handle off the stack

Trace

S ::= a*AB*e
A ::= *A*bc | b
B ::= d

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

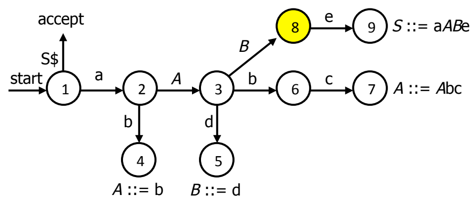


- Repeat all of the states up to the start of the handle

Trace

S ::= a*AB*e
A ::= *A*bc | b
B ::= d

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



- Then make a single transition corresponding to new nonterminal.

Avoiding DFA Rescanning

- Observation 1: no need to restart the DFA after a shift. Stay in same state and process next (shifted) token.
- Observation 2: after a reduction, the contents of the stack prior to the handle are the same as before. After that, new stack will contain a single non-terminal.
  - Scanning the new stack will take us through the same transitions up to the beginning of the handle. Then, one more transition for the new non-terminal.
  - Can record state numbers on the stack with each symbol, and go directly to the appropriate state when we pop the handle from the stack

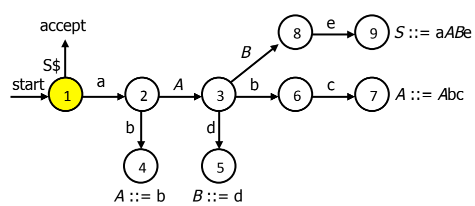
New Stack

- 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 add a symbol to the stack, push the symbol *plus* new FA state
  - If X_i is the beginning of the handle that we reduce, popping it will reveal s_{i-1}, which is precisely the state the FA was in prior to reading the handle.
- Optimization: in an actual parser, only the state numbers need to be pushed, since they implicitly contain the symbol information (actually, items).

Trace

S ::= a*AB*e
A ::= *A*bc | b
B ::= d

Stack	Input
$\$s_1$	abbcde\$

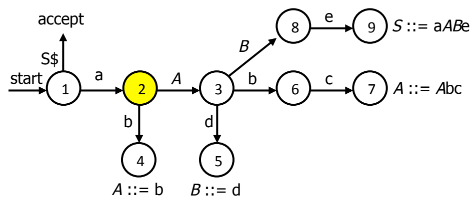


Shift first symbol on to stack.

Trace

S ::= a*AB*e
A ::= *A*bc | b
B ::= d

Stack	Input
$\$s_1$	abbcde$
$\$s_1as_2$	bbcde$

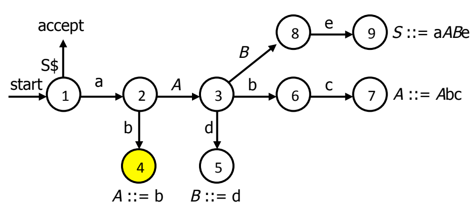


Shifted ‘a’ and state s_2 on to stack, followed ‘a’ transition.
Shift again.

Trace

S ::= a*AB*e
A ::= *A*bc | b
B ::= d

Stack	Input
$\$s_1$	abbcde$
$\$s_1as_2$	bbcde$
$\$s_1as_2bs_4$	bcde$

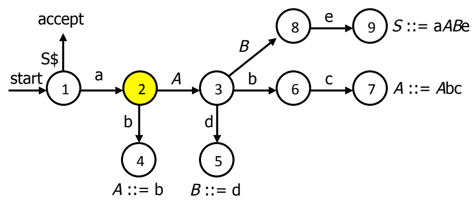


Shifted ‘b’ and state s_4 on to stack, followed ‘b’ transition.
Now, reduce.

Trace

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

Stack	Input
$\$s_1$	abbcde$
$\$s_1as_2$	bbcde$
$\$s_1as_2bs_4$	bcde$
$\$s_1as_2$	bcde$

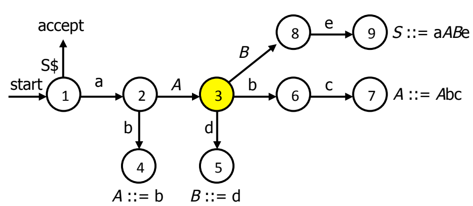


Pop handle (b), revealing new state s_2. Go there.

Trace

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

Stack	Input
$\$s_1$	abbcde$
$\$s_1as_2$	bbcde$
$\$s_1as_2bs_4$	bcde$
$\$s_1as_2As_3$	bcde$



- Push nonterminal on left of reduction (A) onto stack, transition on new nonterminal, and push new state.
- Each step – read an input (shift) or produce an output (reduce). O(n), where n is input + output.

Encoding the DFA in a Table

- Given these optimizations, and *a stack containing states*, a shift-reduce parser's DFA can be encoded in two tables
  - *action table* rows contain state and columns contain input symbols. Encodes what to do given current state and next symbol (e.g., shift and go to state 4).
  - *goto table* rows contain *uncovered* states (states revealed after pop) and columns contain nonterminals. Encodes transition to take after a reduction, given uncovered state and new nonterminal.
    - Based on transition we'd take from uncovered state if we saw the nonterminal. E.g., reduce to A and uncover s_2, goto s_3)
- Note necessity of the stack … can't be done with just an FA, because language grammars are not regular.

Action Table Actions (1)

- Given the current state and input symbol, the main possible actions are
  - *si* – *s*hift the input symbol and state *i* onto the stack (i.e., shift and move to state *i*)
  - r*j* – *r*educe using grammar production *j*
    - The production number tells us how many <symbol, state> pairs to pop off the stack

Action Table 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. Various strategies exist for this, e.g., advance past next semicolon token.

Goto

- When a reduction is performed using production $A ::= \beta$, $|\beta|$ <symbol, state> pairs are popped 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 with production $A ::= \beta$ (after popping handle and pushing A)

Dijkstra

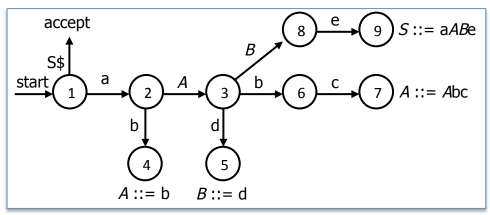
"Thou shalt not use goto"

Dijkstra, revisited

"Thou shalt not use goto, *except as a table in an LR parser*."

LR Parse Table for

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



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			

LR Parsing Algorithm Pseudocode

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

Stack
$\$s_1$

Input
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			

LR States – Where do they come from?

- Idea is that each state of this DFA 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
    - Part-way through: shift
    - All the way through: reduce
    - Preview: Could there be some ambiguity here?
      - Reduce-Reduce and Shift-Reduce conflicts

Items

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

  A ::= *.XY*

  A ::= *X.Y*

  A ::= *XY.*
- Idea: The dot represents a position in the production

DFA with items for $S ::= aABe$, $A ::= Abc \mid b$, $B ::= d$