COP-3402 Systems Software
10.08 Thu

Bottom-up Parsing:
Unlike recursive descent parsing which started at the root node and tried to guess which production we need to expand in order to reach our input string, bottom-up parsing goes the other way. We start with the characters of the input string and try to figure out which productions will immediately match the input strings. Then we gradually try to build up the derivation in the language.

Generative grammar starts from the starting symbol and creates some utterance in the language. However, the parsing problem is the reverse of this. Given some string in the language, we find a way to derive that from our grammar. There are some strategies for parsing, bottom-up is one of them.

Top-down is producing an input from start symbol
Bottom-up is reducing the string to the start symbol
Reduction is the reverse of production

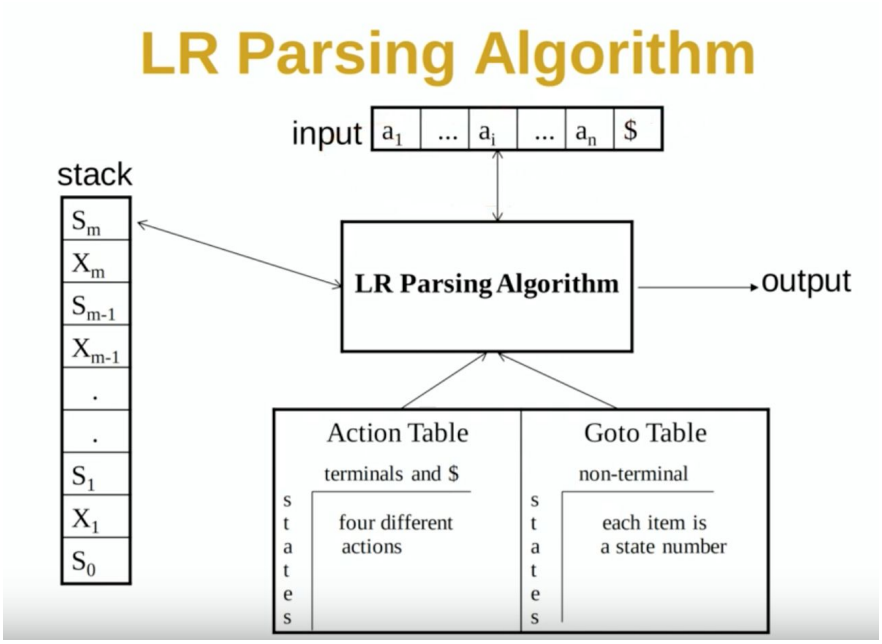
State of parsing is stored on a stack
Initially the stack has a bottom of stack, and the input is the entire string to be parsed, plus an end marker
Goal is to consume string and end up with start symbol on stack

Shift the next input symbol onto stack
Reduce handle on top of stack
Accept if successfully get to the start symbol

Our goal is to find a useful CF grammar that:
Covers unambiguous CF languages
Is Easy to recognize
Avoids conflicts
Is amenable to a fast parsing algorithm

LR(k) parsing
Left to right scanning, **R**ight-most derivation, **k** lookahead

LR is associated with bottom-up, LL with top-down
Shift consumes a terminal from the input, **Reduce** elements off the stack and replaces it with the left-hand side of the production.



It uses the top of the stack, the latest state that has been seen, and the next character in the input to determine what the next parsing action it is. Stack has symbols and states.

1. **shift s** -- shifts the next input symbol onto the stack. Shift is performed only if **action[s_m,a_i] = sk**, where k is the new state. In this case
   (S_o X_1 S_1 ... X_m S_m, a_i a_{i+1} ... a_n $) ➔ (S_o X_1 S_1 ... X_m S_m a_i k, a_{i+1} ... a_n $)
2. **reduce A→β** (if **action[s_m,a_i] = rn** where n is a production number)
   - pop 2|β| items from the stack;
   - then push **A** and **k** where **k=goto[$s_{m-|\beta|}$,A]**

   (S_o X_1 S_1 ... X_m S_m, a_i a_{i+1} ... a_n $) ➔ (S_o X_1 S_1 ... $X_{m-|\beta|}$ $S_{m-|\beta|}$ A k, a_i ... a_n $)

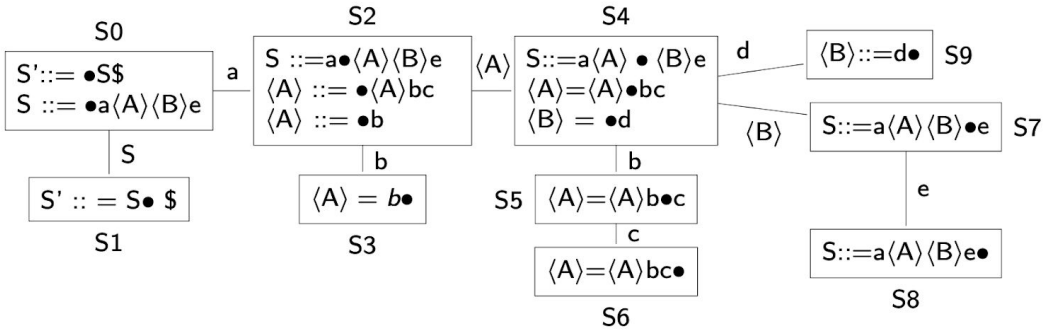
   - Output is the reducing production reduce A→β or the associated semantic action or both
3. **Accept** – Parsing successfully completed
4. **Error** -- Parser detected an error (empty entry in action table)

Shift takes the next input from the stack, looking at the top state on the stack Sm and the next character in the input ai, shifts into the state k, we push ai onto the stack, and also push the state that we are transitioning to.
Reduce basically reduces the production, no input is being consumed, instead, the number of elements on the right-hand side of production being reduced gets popped off the stack and gets replaced with the left-hand side of the non-terminal.

Notes from RUTGERS:
Shift reduce parsers are easily built and easily understood

A shift-reduce parser has just four actions
• Shift — next word is shifted onto the stack
• Reduce — right end of handle is at top of stack
Locate left end of handle within the stack
Pop handle off stack & push appropriate lhs
• Accept — stop parsing & report success
• Error — call an error reporting/recovery routine
Accept & Error are simple
Shift is just a push and a call to the scanner
Reduce takes |rhs| pops (or 2*|rhs| pops) & 1 push
If handle-finding requires state, put it in the stack ⇒ 2x work

Construct LR(0) Parse Table



1	⟨S⟩::=a ⟨A⟩ ⟨B⟩ e
2	⟨A⟩::=⟨A⟩ b c
3	⟨A⟩::=b
4	⟨B⟩::=d

s i − > shift to State i;
g i − > goto State i;
r i − > reduce with Rule i;

	a	b	c	d	e	A	B	S
S0	s2							g1
S1								
S2								
S3								
S4								
S5								
S6								
S7								
S8								
S9								

The idea is, for each of these productions, to look at every possible position inside of the production.

LR(0) Pushdown Automata

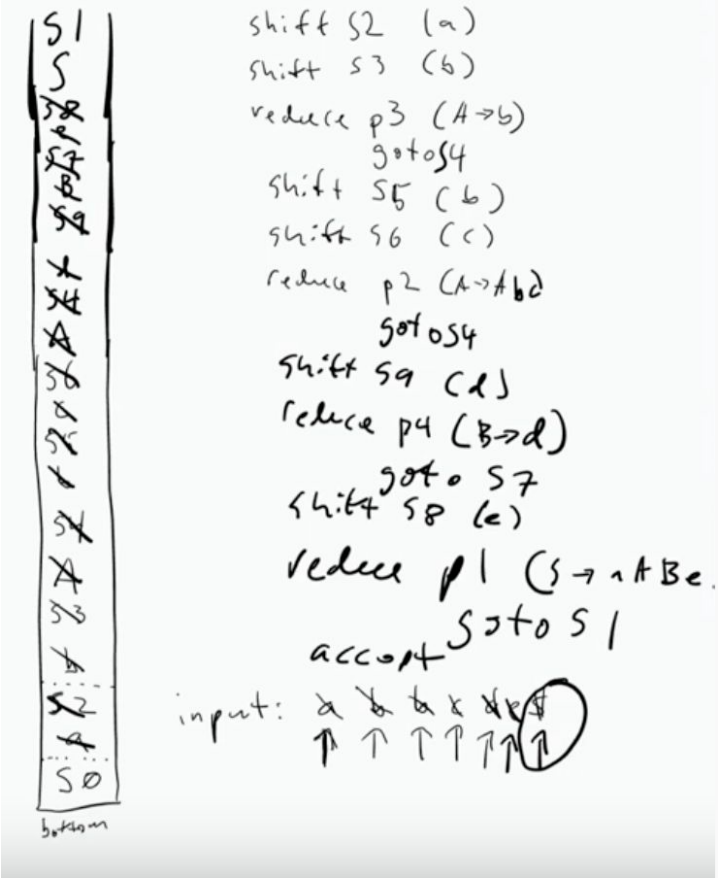
- Example

1	⟨S⟩::=a ⟨A⟩ ⟨B⟩ e
2	⟨A⟩::=⟨A⟩ b c
3	⟨A⟩::=b
4	⟨B⟩::=d

S0	

	a	b	c	d	e	A	B	S
S0	s2							g1
S1								
S2		s3				g4		
S3	r3	r3	r3	r3	r3			
S4		s5		s9			g7	
S5			s6					
S6	r2	r2	r2	r2	r2			
S7					s8			
S8	r1	r1	r1	r1	r1			
S9	r4	r4	r4	r4	r4			

Remaining INPUT: abbcde



Once we get to accept state in our DFA, we can pop off we replace it with the left-hand side, so we can transition to our previous states; state1 for instance.