Concepts of Programming Languages, CSCI 305, Fall 2021
Bottom-Up Parsing, Oct. 25
Bottom-Up Parsing, Section 2.3, pages 87-96

Bottom-up parsing

- Collect input until a sequence that can be reduced with a symbol is found.
- Bottom-up parsing is also knows as shift-reduce parsing.

Called “shift-reduce” parsing because the actions in the cells of the table (LR parsers are almost always table-driven) are shift, reduce and shift-reduce:

- sn, where s says to shift (push onto the stack) and n tells the state to be pushed onto the stack with the state.
- rn, where r says to reduce and n tells what production to use in the reduction. The top of the stack will hold the right-hand side of production n, and these items will be popped. The left-hand side of production n, will eventually be pushed onto the stack. For now it is saved in cur_sym. To avoid losing the last scanned symbol, put it back into the input stream.
- bn, where b says to shift-reduce and n tells what production to use in the reduction. The cur_sym, along with the top of the stack, holds the right-had side of production n, and these will be popped. The left-had side of production n will eventually be pushed onto the stack. For now it is saved in cur_sym.

Both top-down and bottom-up parsers use finite state machines and stacks. The stacks, however, are used for different purposes.

Top-Down (LL)	Bottom-Up (LR)
Stack holds what parser expects to see in the future.	Stack holds what the parser has already seen.

The left version of the grammar is good for LL and the right is good for LR:

Top-Down (LL)	Bottom-Up (LR)
1. program → stmt_list $$	1. program → stmt_list $$
2. stmt_list → stmt stmt_list	*2. stmt_list → stmt_list stmt*
3. stmt_list → ε	*3. stmt_list → stmt*
4. stmt → id := expr	4. stmt → id := expr
5. stmt → read id	5. stmt → read id
6. stmt → write expr	6. stmt → write expr
	7. expr → term
7. expr → term term_tail	*8. expr → expr add_op term*
8. term_tail → add_op term term_tail	
9. term_tail → ε	*9. term → factor*
10. term → factor factor_tail	
11. factor_tail → mult_op factor factor_tail	*10. term → term mult_op factor*
12. factor_tail → ε	
13. factor → (expr)	11. factor → (expr)
14. factor → id	12. factor → id
15. factor → number	13. factor → number
16. add_op → +	14. add_op → +
17. add_op → -	15. add_op → -
18. mult_op → *	16. mult_op → *
19. mult_op → /	17. mult_op → /

A grammar is right-recursive if there is a nonterminal A such that $A \Rightarrow^{+} \alpha A$ for some alpha.

Example (Figure 2.20, page 70):

1. id_list → id id_list_tail
2. id_list_tail → , id id_list_tail
3. id_list_tail → ;

Handle – symbols on the stack that represent the right part of a production, and will be popped so the symbol on the left side of the reduction can be pushed.