Concepts of Programming Languages, CSCI 305, Fall 2020
Recursive Descent Parsing – Subroutines, pages 73-78

Text gives pseudo-code for top-down and bottom-up parsers:
Top-down, LL(1), parser – Figure 2.17, pages 76 & 77
Bottom-up, SLR(1), parser – Figure 2.29, page 99

Two grammars which describe the same language:

Intuitive Grammar (Example 2.8, page 52)	Grammar Good for top-down (LL) parser
	expr → term term_tail
expr → term | expr add_op term	term_tail → add_op term term_tail | ε
term → factor | term mult_op factor	term → factor factor_tail
factor → id | number | - factor | (expr)	factor_tail → mult_op factor | ε
add_op → + | -	factor → id | number | - factor | (expr)
mult_op → * | /	add_op → + | -
id → A | B | C | D	mult_op → * | /
	id → A | B | C | D

LL parsing:
Start with the root and predict the needed production based on:
1. current token returned from the lexical analyzer
2. left-most non-terminal in the parse tree being build

The LL parser can be built in two ways:
1. Individual subroutines corresponding to the variables (non-terminals) of the grammar.
2. Using a table.

More complete example of equivalent grammars:

Grammar (Non-ambiguous)	Equivalent Grammar Good for top-down (LL) parser (Figure 2.16, page 74)
program → stmt_list $$	program → stmt_list $$
stmt_list → stmt stmt_list | ε	stmt_list → stmt stmt_list | ε
stmt → id := expr | read id | write expr	stmt → id := expr | read id | write expr
expr → expr add_op term | term	expr → term term_tail
term → term mult_op factor | factor	term_tail → add_op term term_tail | ε
factor → (expr) | id | number	term → factor factor_tail
add_op → + | -	factor_tail → mult_op factor_tail | ε
mult_op → * | /	factor → (expr) | id | number
	add_op → + | -
	mult_op → * | /

Recursive Descent Parser – write a subroutine for each token and call these subroutines recursively. Code shown in Figure 2.17, pages 76 & 77