Chapter 2

Bottom-Up Parsing

Objective

Today's primary objective:

- Understand the difference between top-down and bottom-up parsing
- Know that grammar structures are different for top-down and bottom-up parsing
- Be able to hand-execute pseudo code for a bottom-up parser

Parsing: Top-down Vs. Bottom-Up

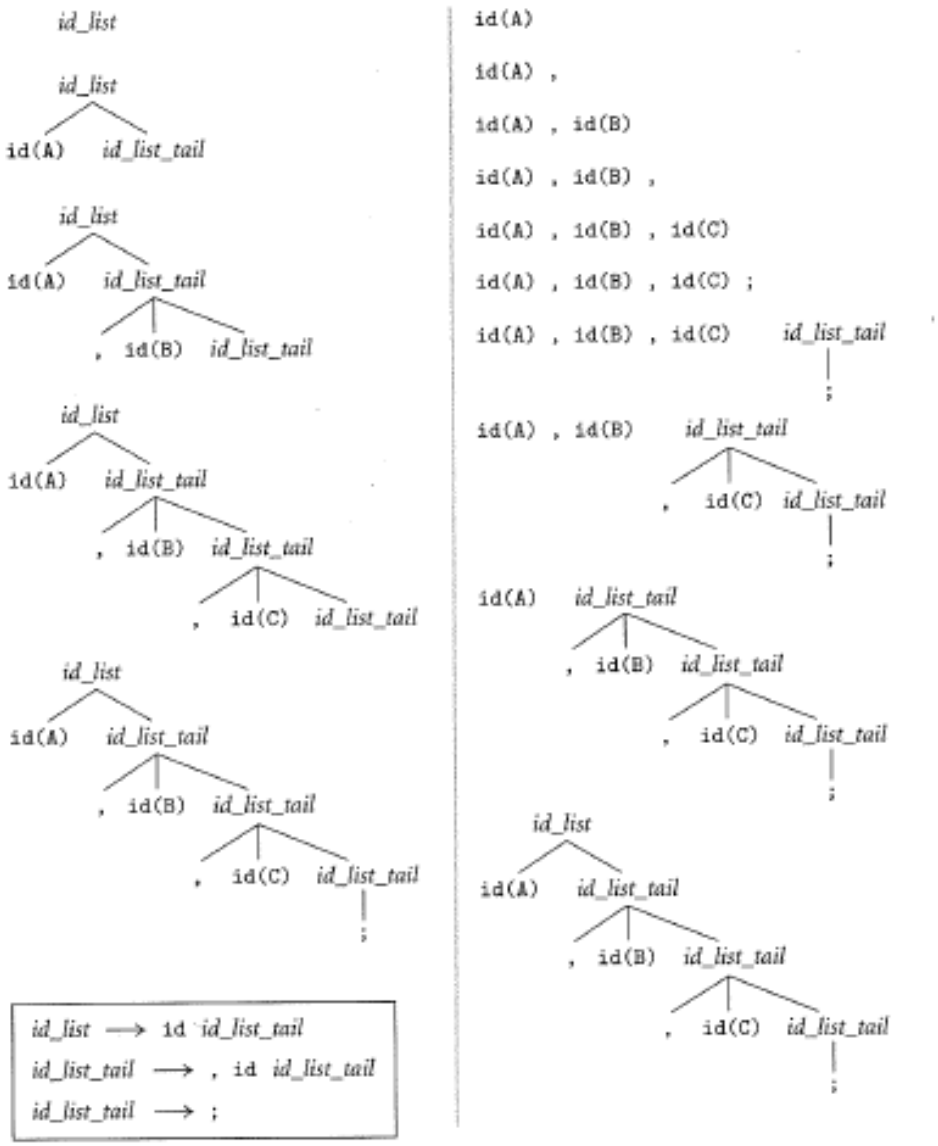

Figure 2.13 **Top-down (left) and bottom-up parsing (right)** of the input string `A, B, C;`. Grammar appears at lower left.

Parsers: Top-Down (LL) versus Bottom-Up (LR)

LL parsers begin at the start symbol and try to apply productions to arrive at the target string

LR parsers begin at the target string and try to arrive back at the start symbol

Top-Down Parsing - Table Driven

Grammar:

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

Top-of-stack nonterminal	Current input token											
	id	number	read	write	:=	(	)	+	-	*	/	$$
program	1	–	1	1	–	–	–	–	–	–	–	1
stmt_list	2	–	2	2	–	–	–	–	–	–	–	3
stmt	4	–	5	6	–	–	–	–	–	–	–	–
expr	7	7	–	–	–	7	–	–	–	–	–	–
term_tail	9	–	9	9	–	–	9	8	8	–	–	9
term	10	10	–	–	–	10	–	–	–	–	–	–
factor_tail	12	–	12	12	–	–	12	12	12	11	11	12
factor	14	15	–	–	–	13	–	–	–	–	–	–
add_op	–	–	–	–	–	–	–	16	17	–	–	–
mult_op	–	–	–	–	–	–	–	–	–	18	19	–

LL(1) parse table for the calculator language. Table entries indicate the production to predict (as numbered in Figure 2.22). A dash indicates an error. When the top-of-stack symbol is a terminal, the appropriate action is always to match it against an incoming token from the scanner. An auxiliary table, not shown here, gives the right-hand-side symbols for each production.

Comparison – Grammar for LL and 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 → /

Bottom-up Parsing

```
state = 1 . . number_of_states
symbol = 1 . . number_of_symbols
production = 1 . . number_of_productions
action_rec = record
    action : (shift, reduce, shift_reduce, error)
    new_state : state
    prod : production

parse_tab : array [symbol, state] of action_rec
prod_tab : array [production] of record
    lhs : symbol
    rhs_len : integer
-- these two tables are created by a parser generator tool

parse_stack : stack of record
    sym : symbol
    st : state
parse_stack.push(⟨null, start_state⟩)
cur_sym : symbol := scan                    -- get new token from scanner
loop
    cur_state : state := parse_stack.top.st -- peek at state at top of stack
    if cur_state = start_state and cur_sym = start_symbol
        return                              -- success!
    ar : action_rec := parse_tab[cur_state, cur_sym]
    case ar.action
        shift:
            parse_stack.push(⟨cur_sym, ar.new_state⟩)
            cur_sym := scan                 -- get new token from scanner
        reduce:
            cur_sym := prod_tab[ar.prod].lhs
            parse_stack.pop(prod_tab[ar.prod].rhs_len)
        shift_reduce:
            cur_sym := prod_tab[ar.prod].lhs
            parse_stack.pop(prod_tab[ar.prod].rhs_len-1)
        error:
            parse_error
```

Figure 2.28 **Driver for a table-driven SLR(1) parser.** We call the scanner directly, rather than using the global input_token of Figures 2.16 and 2.18, so that we can set cur_sym to be an arbitrary symbol.

SLR(1) Parse Table for Calculator Language

Top-of-stack state	Current input symbol																		
	sl	*s*	*e*	*t*	*f*	*ao*	*mo*	id	lit	r	w	:=	(	)	+	-	*	/	$$
0	s2	b3	–	–	–	–	–	s3	–	s1	s4	–	–	–	–	–	–	–	–
1	–	–	–	–	–	–	–	b5	–	–	–	–	–	–	–	–	–	–	–
2	–	b2	–	–	–	–	–	s3	–	s1	s4	–	–	–	–	–	–	–	b1
3	–	–	–	–	–	–	–	–	–	–	–	s5	–	–	–	–	–	–	–
4	–	–	s6	s7	b9	–	–	b12	b13	–	–	–	s8	–	–	–	–	–	–
5	–	–	s9	s7	b9	–	–	b12	b13	–	–	–	s8	–	–	–	–	–	–
6	–	–	–	–	–	s10	–	r6	–	r6	r6	–	–	–	b14	b15	–	–	r6
7	–	–	–	–	–	–	s11	r7	–	r7	r7	–	–	r7	r7	r7	b16	b17	r7
8	–	–	s12	s7	b9	–	–	b12	b13	–	–	–	s8	–	–	–	–	–	–
9	–	–	–	–	–	s10	–	r4	–	r4	r4	–	–	–	b14	b15	–	–	r4
10	–	–	–	s13	b9	–	–	b12	b13	–	–	–	s8	–	–	–	–	–	–
11	–	–	–	–	b10	–	–	b12	b13	–	–	–	s8	–	–	–	–	–	–
12	–	–	–	–	–	s10	–	–	–	–	–	–	–	b11	b14	b15	–	–	–
13	–	–	–	–	–	–	s11	r8	–	r8	r8	–	–	r8	r8	r8	b16	b17	r8

Figure 2.27 SLR(1) parse table for the calculator language. Table entries indicate whether to shift (s), reduce (r), or shift and then reduce (b). The accompanying number is the new state when shifting, or the production that has been recognized when (shifting and) reducing. Production numbers are given in Figure 2.24. Symbol names have been abbreviated for the sake of formatting. A dash indicates an error. An auxiliary table, not shown here, gives the left-hand-side symbol and right-hand-side length for each production.

Characteristic Equation

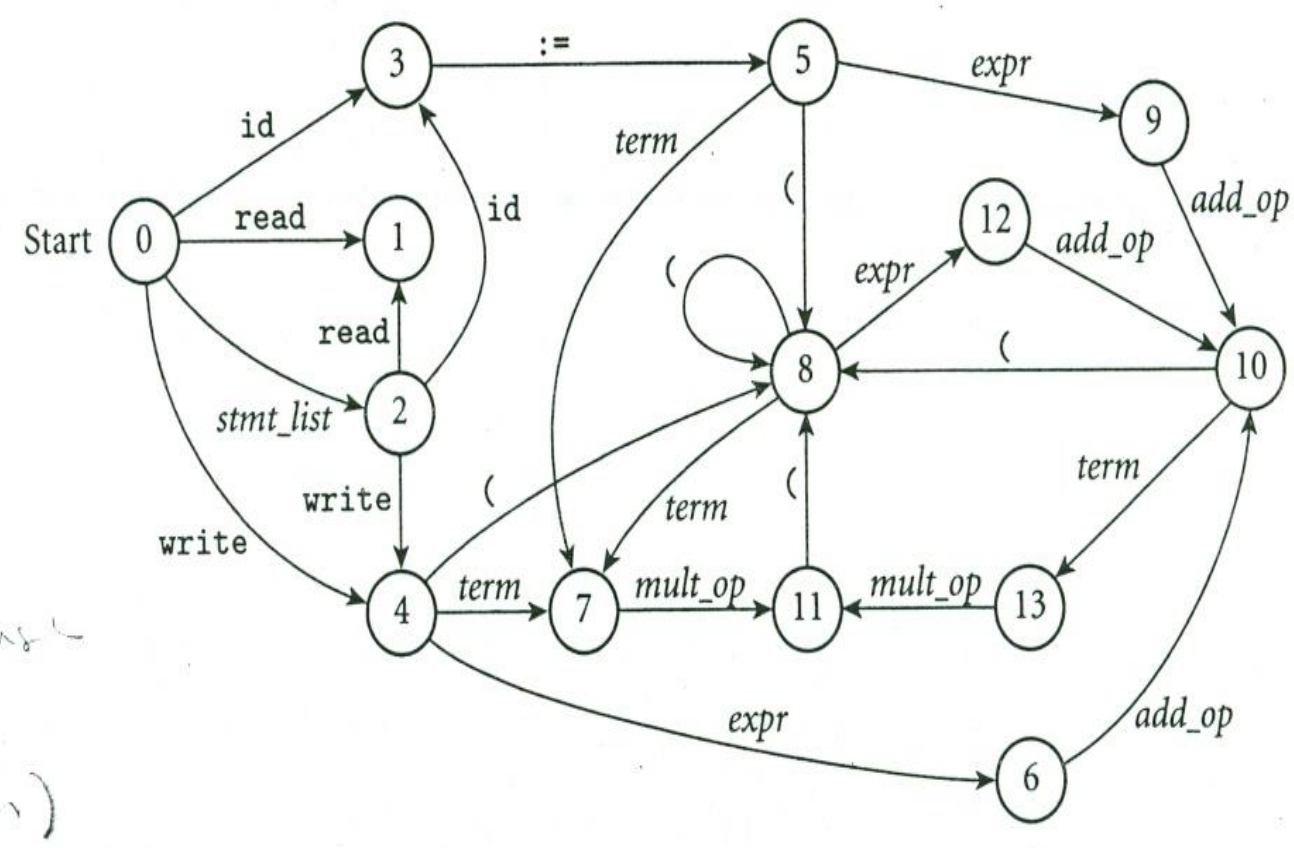

Figure 2.26 Pictorial representation of the CFSM of Figure 2.25. Reduce actions are not shown.