Chapter 2

Parsing

Objective

Today's primary objective:

- You know the difference between top-down and bottom-up parsing
- You are familiar with ad-hoc and table driven top-down parsing

Parsing: Top-down Vs. Bottom-Up

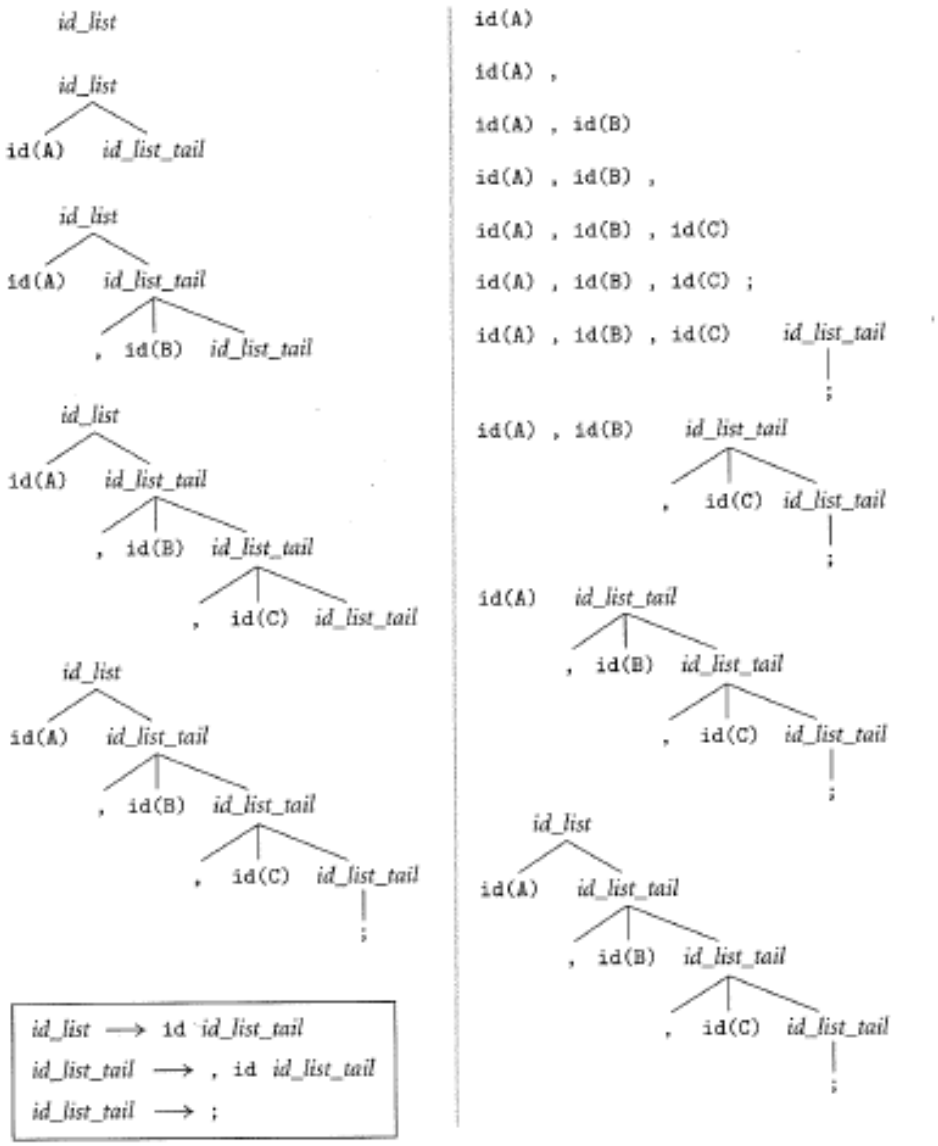

Figure 2.13 **Top-down (left) and bottom-up parsing (right)** of the input string `A, B, C;`. Grammar appears at lower left.

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

Recursive Descent Parsing via Subroutines

Grammar:

- program → stmt_list $$
- stmt_list → stmt stmt_list | ε
- stmt → id := expr | read id | write expr
- expr → term term_tail
- term_tail → add_op term term_tail | ε
- term → factor factor_tail
- factor_tail → mult_op factor_tail | ε
- factor → (expr) | id | number
- add_op → + | -
- mult_op → * | /

Recursive Descent Parsing via Subroutines

```
procedure match(expected)
    if input_token = expected then consume input_token
    else parse_error

-- this is the start routine:
procedure program
    case input_token of
        id, read, write, $$ :
            stmt_list
            match($$)
        otherwise parse_error

procedure stmt_list
    case input_token of
        id, read, write : stmt; stmt_list
        $$ : skip       -- epsilon production
        otherwise parse_error

procedure stmt
    case input_token of
        id : match(id); match(:=); expr
        read : match(read); match(id)
        write : match(write); expr
        otherwise parse_error

procedure expr
    case input_token of
        id, number, ( : term; term_tail
        otherwise parse_error

procedure term_tail
    case input_token of
        +, - : add_op; term; term_tail
        ), id, read, write, $$ :
            skip       -- epsilon production
        otherwise parse_error

procedure term
    case input_token of
        id, number, ( : factor; factor_tail
        otherwise parse_error
```

Figure 2.16 Recursive descent parser for the calculator language. Execution begins in procedure program. The recursive calls trace out a traversal of the parse tree. Not shown is code to save this tree (or some similar structure) for use by later phases of the compiler. *(continued)*

Recursive Descent Parsing via Subroutines

```
procedure factor_tail
    case input_token of
        *, / : mult_op; factor; factor_tail
        +, -, ), id, read, write, $$ :
            skip        -- epsilon production
        otherwise parse_error

procedure factor
    case input_token of
        id : match(id)
        number : match(number)
        ( : match(); expr; match())
        otherwise parse_error

procedure add_op
    case input_token of
        + : match(+)
        - : match(-)
        otherwise parse_error

procedure mult_op
    case input_token of
        * : match(*)
        / : match(/)
        otherwise parse_error
```

Figure 2.16 *(continued)*

Recursive Descent Parsing Table Driven

```
terminal = 1 .. number_of_terminals
non_terminal = number_of_terminals + 1 .. number_of_symbols
symbol = 1 .. number_of_symbols
production = 1 .. number_of_productions

parse_tab : array [non_terminal, terminal] of record
    action : (predict, error)
    prod : production
prod_tab : array [production] of list of symbol
-- these two tables are created by a parser generator tool

parse_stack : stack of symbol

parse_stack.push(start_symbol)
loop
    expected_sym : symbol := parse_stack.pop
    if expected_sym ∈ terminal
        match(expected_sym)                    -- as in Figure 2.16
        if expected_sym = $$ then return       -- success!
    else
        if parse_tab[expected_sym, input_token].action = error
            parse_error
        else
            prediction : production := parse_tab[expected_sym, input_token].prod
            foreach sym : symbol in reverse prod_tab[prediction]
                parse_stack.push(sym)
```

Figure 2.18 Driver for a table-driven LL(1) parser.

Recursive Descent 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	–

Figure 2.19 **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.