Chapter 04: Top-Down Parsing (Part 03)

3. Recursive Descent Parsing

In its fundamental form, a **recursive-descent parser** consists of a collection of **mutually recursive procedures**, each designed to recognize a specific syntactic construct defined by the grammar's production rules. For each non-terminal A in the grammar, there exists a corresponding procedure that attempts to recognize strings derivable from A. When a procedure is invoked, it examines the current input symbol (lookahead) to determine which alternative of the corresponding production rule should be applied.

The parsing process accommodates different types of grammar rules through specific implementation patterns:

- For alternative productions ($A \rightarrow \alpha|\beta$), the procedure employs conditional statements based on the lookahead symbol to select the appropriate alternative.
- For sequential productions ($A \rightarrow XYZ$), the procedure makes sequential calls to the procedures for X, Y, and Z.
- For optional elements ($A \rightarrow \alpha|\varepsilon$), the procedure includes conditional logic to determine whether to process α or skip it entirely.
- For repetitive elements ($A \rightarrow \alpha A|\varepsilon$), the procedure implements iterative constructs to handle multiple occurrences of α.

A crucial requirement for recursive-descent parsing is that the grammar must be LL(1)-compliant, meaning it must be **free from left recursion** and **left factored** appropriately. This ensures that the parser can make correct decisions about which production to apply based solely on the current input symbol, without requiring backtracking.

Example : Consider the previous LL(1) grammar :

E →TE'
E' →+TE' | ε
T →FT'
T' →*FT' | ε
F →id | const | (E)

The recursive descent algorithm for this grammar can be implemented as follows :

Let :

```
input : string                // array of input characters (tokens)
i: integer                    // current position in input
current_token : character     // current character
i ← 0;                         // current position initialized to 0

procedure match(expected_terminal)
    if current_token == expected_terminal
        i ← i+1
        if (i<length(input))
          current_token ← input[i]
        else
          current_token ← '$'
    else
        report error: "Expected " + expected_terminal + ", found " +
current_token

procedure E
    call T
    call E'

procedure E'
    if current_token == '+'
        match('+')
        call T
        call E'
    else
        // do nothing — ε-production
```

```
procedure T
    call F
    call T'

procedure T'
    if current_token == '*'
        match('*')
        call F
        call T'
    else
        // do nothing — ε-production

procedure F
    if current_token == 'id'
        match('id')
    else if current_token == 'const'
        match('const')
    else if current_token == '('
        match('(')
        call E
        match(')')
    else
        report error: "Expected id, const, or ("

procedure Main
    Read (input)
    current_token ← input[0]
    call E

    if current_token == '$'
        print "Parsing succeeded."
    else
        report error: "Unexpected input after valid expression."
```

Let's parse the input string : `id + id * id`

Parsing Input: `"id + id * id$"`

Initialization: `current_token= 'id'`

- As mentioned in the Main function, the procedure E is firstly called :

```
E_____T ____F → 'id' matched, current_token = '+'
|       | ____T' → no match, do nothing
```

```
|
|_____ E’ ____ + → '+' matched, current_token = 'id'
       | ____ T _____ F → 'id' matched, current_token = '*'
              | _____ T’ _____ * → '*' matched, current_token = 'id'
              |       | _____ F → 'id' matched, current_token = '$'
              |       | _____ T’ → no match, do nothing
              |
              E’ → no match,do nothing
```

- By exiting the procedure E, condition **`current_token = '$'` is verified** → "Parsing succeeded."