

# ■ Tema 4: VBC (Recursive-Descent Calculator)

---

O objetivo é preencher as funções de parsing (`parse_expr_r`, `parse_term`, `parse_factor`) para construir uma Árvore de Sintaxe Abstrata (AST) que o `eval_tree` (fornecido) possa calcular, respeitando a precedência de `*` sobre `+` e parênteses.

**Funções-Chave:** `malloc` (ou `calloc`), `free`, `printf`, `isdigit`.

---

## Entendendo a Gramática (BNF)

- **Objetivo:** Entender a precedência de operadores.
- **Explicação:** A matemática diz  $3+4*5 = 23$ , não  $35$ . `*` "prende" mais forte que `+`. Parênteses `()` forçam a ordem.

**Dica:** A solução é uma gramática que reflete isso:

- **Expr (Expressão)** = uma soma de `Termos` (`Term (+ Term)*`)
- **Term (Termo)** = uma multiplicação de `Factor(es)` (`Factor (* Factor)*`)
- **Factor (Fator)** = um número `(0-9)` ou uma `Expr` entre parênteses `(( Expr ))`
- **Teste:** Mapeie `(3+4)*5` para esta gramática. `(3+4)` é um `Factor` porque é `(( Expr ))`.

## Entendendo as Funções Fornecidas

- **Objetivo:** Ver o que já está pronto.
- **Dica:** `eval_tree` percorre a árvore após ela ser construída. `new_node` aloca um nó. `destroy_tree` libera a árvore. `unexpected` imprime erros. `accept` consome um char se ele corresponder. `expect` exige que um char corresponda.
- **Teste:** Leia `eval_tree` e entenda como ele calcula `(3+4)` (um nó `ADD` com filhos `VAL(3)` e `VAL(4)`).

## Implementando `parse_factor` (Parte 1: Dígitos)

- **Objetivo:** Analisar a unidade mais simples: um número.

- **Função-Chave:** `isdigit()`.
- **Dica:** A gramática diz: `Factor = Digit | ...`. O `parse_factor` deve primeiro verificar `if (isdigit(**s))`.
- **Tarefa:** Se for um dígito, crie um nó `VAL` (ex: `.type = VAL, .val = **s - '0'`), avance o ponteiro `(*s)++`, e retorne `new_node(n)`.

### Implementando `parse_factor` (Parte 2: Parênteses)

- **Objetivo:** Analisar `( Expr )`.
- **Dica:** Se não for um dígito, verifique `if (accept(s, '('))`.

**Tarefa:** Se um `(` for aceito:

1. Chame recursivamente `node *e = parse_expr_r(s);` (a função que você ainda vai escrever).
2. Exija a parêntese de fechamento: `if (!expect(s, ')')) ...`
3. Se `)` não for encontrado, `destroy_tree(e)` e retorne `NULL`.
4. Retorne `e`.

- **Teste:** Se a entrada for `(5)`, `parse_factor` chama `parse_expr_r`, que chama `parse_term`, que chama `parse_factor`, que retorna `VAL(5)`.

### Lidando com Erros em `parse_factor`

- **Objetivo:** Tratar entradas inválidas.
- **Dica:** Se a entrada não for um dígito e não for `(`, é um erro.
- **Tarefa:** Chame `unexpected(**s)` e retorne `NULL`.

### Implementando `parse_term` (Multiplicação)

- **Objetivo:** Analisar `Factor (* Factor)*`.
- **Dica:** Esta função implementa a "amarração" do `*`.

**Tarefa:**

1. Obtenha o primeiro fator: `node *left = parse_factor(s);`

2. Inicie um loop: `while (accept(s, '*' ))`.
3. Dentro do loop:
  - Obtenha o próximo fator: `node *right = parse_factor(s);`
  - Trate erros (se `right` for `NULL`, `destroy_tree(left)` e retorne `NULL`).
  - **Combine-os:** Crie um novo nó `MULTI`: `node n = { .type = MULTI, .l = left, .r = right };`
  - **A Mágica:** `left = new_node(n);`. O resultado da multiplicação se torna o novo `left` para a próxima iteração.
4. Fora do loop, retorne `left`.
  - **Teste:** Se a entrada for `4*5*6`, o loop roda duas vezes. A árvore se torna `MULTI(MULTI(4, 5), 6)`.

### Implementando `parse_expr_r` (Adição)

- **Objetivo:** Analisar `Term (+ Term)*`.
- **Dica:** O código para esta função é *idêntico* ao `parse_term`, mas troca `parse_factor` por `parse_term` e `*` por `+`.

### Tarefa:

1. Obtenha o primeiro termo: `node *left = parse_term(s);`
2. Inicie um loop: `while (accept(s, '+'))`.
3. Dentro do loop:
  - Obtenha o próximo termo: `node *right = parse_term(s);`
  - Combine-os em um nó `ADD`.
  - `left = new_node(n);`
4. Retorne `left`.

### Entendendo a Precedência

- **Objetivo:** Ver por que isso funciona.

**Dica:** Trace `3+4*5`.

1. `parse_expr_r` chama `parse_term(left)`.
2. `parse_term` chama `parse_factor` (obtém `VAL(3)`). O loop `*` não roda. Retorna `VAL(3)`.
3. `parse_expr_r` (loop `+`): `accept(s, '+')` (sucesso).
4. `parse_expr_r` chama `parse_term(right)`.
5. `parse_term` chama `parse_factor` (obtém `VAL(4)`).
6. `parse_term` (loop `*`): `accept(s, '*')` (sucesso).
7. `parse_term` chama `parse_factor` (obtém `VAL(5)`).
8. `parse_term` combina-os: `left = MULTI(4, 5)`. O loop `*` termina. Retorna `MULTI(4, 5)`.
9. `parse_expr_r` agora tem `left = VAL(3)` e `right = MULTI(4, 5)`.
10. `parse_expr_r` combina-os: `left = ADD(VAL(3), MULTI(4, 5))`.

- **Teste:** `eval_tree` neste nó retornará `3 + (4*5) = 23`. Sucesso!

### Entendendo `parse_expr` (O Wrapper)

- **Objetivo:** O ponto de entrada principal.
- **Dica:** O código fornecido `parse_expr` chama `parse_expr_r` e depois verifica se sobrou lixo no final da string (ex: `1+2)abc`).
- **Tarefa:** Não há nada a fazer aqui, apenas entenda por que `if (*p)` é a verificação de erro final.

### Exercício Composto: Gerenciamento de Memória

- **Objetivo:** Garantir que não haja vazamentos em caso de erro.
- **Dica:** Observe a lógica de erro em `parse_term` e `parse_expr_r`. Se `parse_factor` (ou `parse_term`) à direita (`right`) falhar e retornar `NULL`, o que acontece com o nó `left` que já foi alocado com sucesso?

- **Tarefa:** Certifique-se de que `destroy_tree(left);` é chamado antes de retornar `NULL` em todos os caminhos de erro, como mostrado no código de exemplo.
  - **Teste:** `(3+4 (falta ))`. `parse_factor` chamará `parse_expr_r` (que constrói `ADD(3,4)`) e depois falhará no `expect(s, "'")`. Verifique se `destroy_tree(e)` é chamado para liberar o `ADD(3,4)`.
-