Aula de Compiladores: Análise Sintática Top-Down com Descida Recursiva

Parte Teórica: Fundamentos da Análise Sintática Top-Down

O que é Análise Sintática?

A análise sintática (ou *parsing*) é a etapa de um compilador que verifica se uma sequência de tokens gerada pelo analisador léxico segue as regras de uma gramática formal. Ela constrói uma representação intermediária, geralmente uma árvore sintática, que reflete a estrutura hierárquica da linguagem. Existem duas abordagens principais para a análise sintática: **top-down** (de cima para baixo) e **bottom-up** (de baixo para cima). Nesta aula, focaremos na abordagem top-down.

Análise Top-Down: Conceito

Na análise top-down, o processo começa com o símbolo inicial da gramática (a "raiz" da árvore sintática) e tenta derivar a sequência de entrada expandindo as regras de produção até chegar aos tokens reais (as "folhas"). O objetivo é encontrar uma derivação que corresponda à entrada fornecida. Essa abordagem é intuitiva porque segue a estrutura natural da gramática, como ela foi definida.

Descida Recursiva: Uma Técnica Top-Down

A **descida recursiva** (*recursive descent parsing*) é uma das formas mais simples e diretas de implementar a análise top-down. Nessa técnica:

- Cada **não-terminal** da gramática é representado por uma função.
- Essas funções chamam umas às outras (ou a si mesmas) para processar a entrada, seguindo as produções da gramática.
- O processo é guiado pelo **lookahead**, que é o próximo token da entrada, usado para decidir qual produção aplicar.

Características da Descida Recursiva

- **Simplicidade**: O código reflete diretamente a gramática, facilitando a implementação manual.

- **Recursividade**: Usa a pilha de chamadas do sistema para gerenciar as expansões, o que elimina a necessidade de uma pilha explícita (como em analisadores bottom-up).
- **Limitações**: Funciona melhor com gramáticas LL(1), ou seja, gramáticas que podem ser analisadas com um único token de lookahead sem ambiguidade ou recursão à esquerda.

Gramáticas LL(1)

Uma gramática é dita LL(1) se:

- **L**: Lê a entrada da esquerda para a direita (*left-to-right*).
- **L**: Produz uma derivação mais à esquerda (*leftmost derivation*).
- **1**: Usa apenas um token de lookahead para decidir qual produção aplicar.

Para que a descida recursiva funcione sem problemas, a gramática não deve ter:

- **Recursão à esquerda**: Ex.: `A → A b` causa chamadas infinitas. Isso pode ser resolvido por reescrita (eliminação de recursão à esquerda).
- **Ambiguidade**: Múltiplas produções possíveis para o mesmo lookahead confundem o analisador.

Exemplo Teórico

Considere a gramática:

```
S → a S b | c
```

- Para a entrada `a a c b b`:
  - Começamos com `S`.
  - Vemos `a` como lookahead, aplicamos `S → a S b`.
  - Processamos o próximo `S`, vemos outro `a`, aplicamos novamente `S → a S b`.
  - Por fim, vemos `c`, aplicamos `S → c`.
  - Consumimos os `b` restantes, completando a derivação.

A árvore sintática seria construída “de cima para baixo”, guiada pela entrada.

Vantagens e Desvantagens

- **Vantagens**:
  - Fácil de entender e implementar.
  - Ideal para gramáticas simples e ensino.
- **Desvantagens**:
  - Não lida bem com gramáticas complexas ou ambíguas.
  - Pode ser ineficiente em linguagens com muitas regras.

Parte Prática: Implementação Recursiva

Agora que os alunos entendem os fundamentos, vamos aplicar a teoria com uma gramática prática e um exemplo implementado.

1. Gramática Exemplo

Vamos usar uma gramática para expressões aritméticas:

```
E → T + E | T
T → F * T | F
F → num | ( E )
```

- `E`: Expressão (ex.: `2 + 3 * 4`).
- `T`: Termo (ex.: `3 * 4`).
- `F`: Fator (ex.: `num` ou `(E)`).
- Terminais: `num` (número), `+`, `*`, `(`, `)`.

Essa gramática é LL(1) porque o lookahead (`+`, `*`, `(`, ou `num`) determina claramente qual produção escolher.

2. Implementação em Pseudocódigo

Aqui está uma implementação em pseudocódigo (baseada em Python simplificado) que reflete a descida recursiva:

```
# Variáveis globais
tokens = []  # Lista de tokens da entrada, ex.: ["2", "+", "3", "*", "4"]
pos = 0      # Posição atual no vetor de tokens

# Função para pegar o próximo token
def lookahead():
```

```
    if pos < len(tokens):
        return tokens[pos]
    return None

# Função para avançar o ponteiro
def consume(expected):
    if lookahead() == expected:
        global pos
        pos += 1
        return True
    raise Exception(f"Erro: esperado '{expected}', encontrado '{lookahead()

# Regra E → T + E | T
def E():
    T()  # Primeiro processa um termo
    if lookahead() == "+":  # Se houver soma, continua recursivamente
        consume("+")
        E()

# Regra T → F * T | F
def T():
    F()  # Primeiro processa um fator
    if lookahead() == "*":  # Se houver multiplicação, continua
        consume("*")
        T()

# Regra F → num | ( E )
def F():
    if lookahead() == "(":  # Caso de expressão entre parênteses
        consume("(")
        E()
        consume(")")
    elif lookahead() in ["0", "1", "2", "3", "4", "5", "6", "7", "8", "9"]:
        consume(lookahead())
    else:
        raise Exception(f"Erro em F: esperado 'num' ou '(', encontrado '{lo

# Função principal
def parse(input_tokens):
    global tokens, pos
    tokens = input_tokens
    pos = 0
    E()
    if lookahead() is None:
        print("Análise concluída com sucesso!")
    else:
```

```
        raise Exception("Erro: entrada não consumida completamente")

# Teste
try:
    parse(["2", "+", "3", "*", "4"])
except Exception as e:
    print(e)
```

Explicação do Código

- **`lookahead()`** : Retorna o próximo token sem avançar.
- **`consume()`** : Verifica e consome o token esperado, avançando o ponteiro.
- **`E()`, `T()`, `F()`** : Funções recursivas que implementam as regras da gramática.
- **Erros**: São lançados quando o token atual não corresponde ao esperado ou sobram tokens.

Execução Passo a Passo

Para `2 + 3 * 4`:

1. `E()` chama `T()`.
2. `T()` chama `F()`, consome `2`.
3. Volta a `T()`, não vê `*`, retorna a `E()`.
4. `E()` vê `+`, consome e chama `E()` novamente.
5. Novo `E()` chama `T()`, que chama `F()`, consome `3`.
6. `T()` vê `*`, consome e chama `T()`, que consome `4`.
7. Fim da entrada, sucesso!

Árvore Sintática

```
      E
     /|\
    T + E
    |   |\
    F   T * T
    |   |   |
   "2" "3" "4"
```

---

3. Exercício Prático

Proponha aos alunos:

1. **Modificação**:
   - Adicione o operador `-` à regra `E`:

     ```
     E → T + E | T - E | T
     ```

   - Ajuste a função `E()` para suportar `-`.
2. **Testes**:
   - `"5 - 2 + 3"`
   - `"2 * (3 + 4)"`
3. **Desafio**:
   - Adicione variáveis (ex.: `x`, `y`) à regra `F`:

     ```
     F → num | ( E ) | id
     ```

   - Teste com `"x + 2 * y"`.