Atividade Prática: Construindo um Compilador para CalcVar do Zero

Objetivo

Desenvolver um compilador completo em Java para a mini-linguagem "CalcVar", implementando análise léxica, sintática e semântica do zero, para processar declarações de variáveis e expressões aritméticas.

Parte 1: Contextualização Teórica

Fases de um Compilador

1. **Análise Léxica**: Transforma o código-fonte em uma sequência de tokens (unidades básicas como palavras-chave, números, operadores).
2. **Análise Sintática**: Verifica se os tokens seguem a gramática da linguagem, construindo uma árvore sintática abstrata (AST).
3. **Análise Semântica**: Analisa o significado do código, verificando tipos, declarações e consistência.

CalcVar: Uma Mini-Linguagem

CalcVar é uma linguagem simples para cálculos com variáveis:

- **Tipos**: `int` (números inteiros).
- **Operações**: `+` (soma), (subtração).
- **Sintaxe**:
  - Declaração: `int nome = valor;` (ex.: `int x = 10;`).
  - Atribuição: `nome = expressao;` (ex.: `x = x + 5;`).
  - Expressões: Combinam variáveis, números e operações (ex.: `x + 3 - 2`).
- **Regras**:
  - Variáveis devem ser declaradas antes do uso.
  - Todas as operações e atribuições devem envolver apenas `int`.

Exemplo de Código Válido

```
int x = 10;
int y = 5;
x = x + y - 3;
```

Exemplo de Código Inválido

```
x = 5;         // Variável não declarada
int z = "abc";  // Tipo inválido
int x = 10 + y; // y não declarada
```

Parte 2: Instruções para Implementação

Instrução Geral

Vocês criarão um compilador para CalcVar em Java do zero, com:

1. **Analisador Léxico**: Gera uma lista de tokens a partir do código-fonte.
2. **Analisador Sintático**: Constrói uma AST a partir dos tokens.
3. **Analisador Semântico**: Verifica tipos e declarações na AST.

Passos para Construção

1. Analisador Léxico

- **Objetivo**: Converter a entrada (string) em uma lista de tokens.
- **Tokens a Reconhecer**:
  - `int` (palavra-chave para declaração).
  - Identificadores (nomes de variáveis: letras seguidas de letras/números).
  - `=` (atribuição).
  - `+` (soma), (subtração).
  - Números inteiros (ex.: `123`, `0`).
  - `;` (fim de instrução).
  - Fim de arquivo (EOF).

- **Como Fazer**:
  - Crie uma classe `Token` com atributos `tipo` (enum) e `valor` (string).
  - Crie uma classe `AnalisadorLexico` com:
    - Um construtor que recebe a string de entrada.
    - Um método `proximoToken()` que lê a entrada caractere por caractere e retorna o próximo token.
  - Ignore espaços em branco.
  - Exemplo de saída para `int x = 5;`:

    ```
    [INT: int], [IDENT: x], [IGUAL: =], [NUMERO: 5], [PONTO_VIRGULA: ;]
    ```

2. Analisador Sintático

- **Objetivo**: Construir uma AST a partir da lista de tokens.
- **Estrutura da AST**:
  - Nó base abstrato (`NoAST`) com subclasses:
    - `NoDeclaracao`: Para `int nome = valor;` (campos: nome, valor).
    - `NoAtribuicao`: Para `nome = expressao;` (campos: nome, expressão).
    - `NoExpressao`: Subclasses:
      - `NoNumero`: Um número (campo: valor).
      - `NoVariavel`: Uma variável (campo: nome).
      - `NoOperacao`: Uma operação (campos: operador, esquerdo, direito).
- **Como Fazer**:
  - Crie uma classe `AnalisadorSintatico` com:
    - Um construtor que recebe a lista de tokens.
    - Um método `analisar()` que retorna uma lista de nós da AST (o programa).
    - Um método auxiliar `parseExpressao()` para processar expressões com `+` e .
  - Use uma abordagem top-down (parser descendente):
    - Para declarações: Espere `int`, identificador, `=`, expressão, `;`.

        - Para atribuições: Espere identificador, `=`, expressão, `;`.
        - Para expressões: Processea da esquerda para a direita (sem precedência, para simplificar).
    - Exemplo de AST para `x = x + 5`:
        - `NoAtribuicao(nome="x", expressao=NoOperacao(operador="+", esquerdo=NoVariavel("x"), direito=NoNumero("5")))`

3. Analisador Semântico

- **Objetivo**: Verificar a consistência semântica da AST.
- **Tarefas**:
    - Garantir que variáveis sejam declaradas antes do uso.
    - Verificar que valores em declarações e expressões sejam `int`.
    - Checar que operações (`+`,) usem apenas `int`.
- **Como Fazer**:
    - Crie uma classe `AnalisadorSemantico` com:
        - Uma tabela de símbolos (ex.: `Map<String, String>` para nome → tipo).
        - Métodos para:
            - Declarar variáveis e validar valores.
            - Verificar atribuições (tipo da expressão deve ser `int`).
            - Analisar operações (operandos devem ser `int`).
            - Percorrer a AST e executar as verificações.
    - Imprima mensagens de erro quando necessário (ex.: "Variável 'x' não declarada").

Integração

- Crie uma classe principal (`CompiladorCalcVar`) que:
    - Receba uma string de entrada (o código CalcVar).
    - Execute o analisador léxico para gerar tokens.
    - Passe os tokens ao analisador sintático para criar a AST.
    - Use o analisador semântico para verificar a AST.
    - Imprima resultados (tokens, AST simplificada, erros semânticos).

Parte 3: Tarefa dos Alunos

Implementação

- Escreva o código do zero em Java, seguindo as orientações acima.
- Estruture o projeto com pelo menos as classes:
  - `Token`
  - `AnalisadorLexico`
  - `NoAST` (e subclasses)
  - `AnalisadorSintatico`
  - `AnalisadorSemantico`
  - `CompiladorCalcVar` (main)

Requisitos Mínimos

- O analisador léxico deve reconhecer todos os tokens listados.
- O analisador sintático deve construir uma AST correta para declarações e atribuições.
- O analisador semântico deve detectar:
  - Uso de variáveis não declaradas.
  - Tipos inválidos (ex.: atribuir texto a `int`).

Exemplo de Saída Esperada

Para `int x = 10; x = x + 5;`:

```
Tokens: [INT: int], [IDENT: x], [IGUAL: =], [NUMERO: 10], [PONTO_VIRGUL
A: ;], ...
AST: [Declaracao: x = 10], [Atribuicao: x = (x + 5)]
Semântica: Programa válido
```

Para `x = 5;`:

```
Tokens: [IDENT: x], [IGUAL: =], [NUMERO: 5], [PONTO_VIRGULA: ;]
AST: [Atribuicao: x = 5]
Erro semântico: Variável 'x' não declarada
```

Parte 4: Testes e Reflexão

Testes

- **Código 1 (Válido)**:

```
int a = 7;
int b = 3;
a = a + b - 2;
```

- **Código 2 (Inválido)**:

```
int x = "texto"; // Tipo errado
x = x + 1;
```

- **Código 3 (Inválido)**:

```
y = 10; // Não declarada
int x = y - 5;
```