Análise Semântica

O que é Análise Semântica?

A análise semântica é a fase do compilador que verifica o significado do código, garantindo que ele seja consistente com as regras da linguagem de programação. Ela sucede a análise léxica (tokenização) e sintática (verificação gramatical), focando em aspectos como tipos, escopos, declarações e restrições específicas.

Objetivos:

- Detectar erros semânticos (ex.: tipos incompatíveis, variáveis não declaradas).
- Coletar informações para geração de código.

Tarefas da Análise Semântica

1. **Verificação de Tipos**: Compatibilidade entre operandos e operações.
2. **Checagem de Declarações**: Identificadores devem ser declarados antes do uso.
3. **Resolução de Escopo**: Visibilidade de variáveis e métodos.
4. **Regras Específicas**: Restrições da linguagem (ex.: classes abstratas em Java).

Formas de Implementar a Análise Semântica

Existem quatro abordagens principais, que podem ser combinadas em um compilador real:

1. **Percurso da Árvore Sintática Abstrata (AST)**:

- Analisa o código percorrendo a AST em uma ordem específica (ex.: pós-ordem ou pré-ordem).
- Ideal para verificar relações entre nós (ex.: operandos de uma expressão).

2. **Tabela de Símbolos**:

   - Estrutura de dados que armazena informações sobre identificadores (nome, tipo, escopo, etc.).
   - Usada para checar declarações e resolver referências.

3. **Anotação de Tipos**:

   - Adiciona metadados (ex.: tipos) aos nós da AST durante o percurso.
   - Facilita verificações posteriores e geração de código.

4. **Análise Baseada em Regras**:

   - Codifica regras específicas da linguagem (ex.: “não instanciar classes abstratas”).
   - Aplicada durante o percurso ou em passes separados.

Implementação Completa em Java

Vamos criar um analisador semântico que combine todas essas abordagens. Ele analisará um código fictício representado como uma AST simplificada, com suporte a declarações, operações e escopos.

Estrutura do Código

- **Nós da AST**: Representam construções do código (declarações, expressões, blocos).

- **Tabela de Símbolos**: Gerencia identificadores.
- **Anotação de Tipos**: Adiciona tipos aos nós.
- **Regras**: Verifica restrições específicas.

```
import java.util.*;

// Classe para representar um símbolo na tabela
class Simbolo {
    String nome;
    String tipo;
    int escopo;

    Simbolo(String nome, String tipo, int escopo) {
        this.nome = nome;
        this.tipo = tipo;
        this.escopo = escopo;
    }
}

// Classe base para nós da AST
abstract class NoAST {
    String tipoAnotado; // Para anotação de tipos
    abstract void analisar(AnalisadorSemantico analisador);
}

// Nó para declaração de variável
class NoDeclaracao extends NoAST {
    String nome;
    String tipo;

    NoDeclaracao(String nome, String tipo) {
```

```
        this.nome = nome;
        this.tipo = tipo;
    }

    @Override
    void analisar(AnalisadorSemantico analisador) {
        analisador.declararVariavel(nome, tipo);
        this.tipoAnotado = tipo;
    }
}

// Nó para expressão (ex.: soma)
class NoExpressao extends NoAST {
    String operador;
    NoAST esquerdo;
    NoAST direito;

    NoExpressao(String operador, NoAST esquerdo, NoAST direito) {
        this.operador = operador;
        this.esquerdo = esquerdo;
        this.direito = direito;
    }

    @Override
    void analisar(AnalisadorSemantico analisador) {
        esquerdo.analisar(analisador);
        direito.analisar(analisador);
        String tipoEsq = esquerdo.tipoAnotado;
        String tipoDir = direito.tipoAnotado;

        if (tipoEsq == null || tipoDir == null) return;
```

```
        if (operador.equals("+")) {
            if (tipoEsq.equals("int") && tipoDir.equals("int")) {
                this.tipoAnotado = "int";
                System.out.println("Expressão '" + toString() + "' válida (int).");
            } else if (tipoEsq.equals("String") || tipoDir.equals("String")) {
                this.tipoAnotado = "String";
                System.out.println("Expressão '" + toString() + "' válida (String).");
            } else {
                System.out.println("Erro semântico: Tipos incompatíveis em '" + toString() + "'.");
            }
        }
    }

    @Override
    public String toString() {
        return esquerdo + " " + operador + " " + direito;
    }
}

// Nó para referência a variável
class NoVariavel extends NoAST {
    String nome;

    NoVariavel(String nome) {
        this.nome = nome;
    }

    @Override
    void analisar(AnalisadorSemantico analisador) {
        Simbolo simbolo = analisador.buscarVariavel(nome);
```

```
        if (simbolo != null) {
            this.tipoAnotado = simbolo.tipo;
        }
    }

    @Override
    public String toString() {
        return nome;
    }
}

// Nó para bloco (escopo)
class NoBloco extends NoAST {
    List<NoAST> instrucoes;

    NoBloco(List<NoAST> instrucoes) {
        this.instrucoes = instrucoes;
    }

    @Override
    void analisar(AnalisadorSemantico analisador) {
        analisador.entrarEscopo();
        for (NoAST instrucao : instrucoes) {
            instrucao.analisar(analisador);
        }
        analisador.sairEscopo();
    }
}

// Classe principal do analisador semântico
class AnalisadorSemantico {
```

```
private Map<String, Simbolo> tabelaSimbolos = new HashMap<>();
private int escopoAtual = 0;

public void declararVariavel(String nome, String tipo) {
    String chave = nome + "_" + escopoAtual; // Evita conflitos entre escopos
    if (tabelaSimbolos.containsKey(chave)) {
        System.out.println("Erro semântico: Variável '" + nome + "' já declarada no escopo " + escopoAtual);
    } else {
        tabelaSimbolos.put(chave, new Simbolo(nome, tipo, escopoAtual));
        System.out.println("Variável '" + nome + "' declarada como " + tipo + " no escopo " + escopoAtual);
    }
}

public Simbolo buscarVariavel(String nome) {
    for (int i = escopoAtual; i >= 0; i--) {
        String chave = nome + "_" + i;
        if (tabelaSimbolos.containsKey(chave)) {
            return tabelaSimbolos.get(chave);
        }
    }
    System.out.println("Erro semântico: Variável '" + nome + "' não declarada.");
    return null;
}

public void entrarEscopo() {
    escopoAtual++;
    System.out.println("Entrando no escopo " + escopoAtual);
}

public void sairEscopo() {
    tabelaSimbolos.entrySet().removeIf(entry -> entry.getValue().escopo == escopoAtual);
```

```
        System.out.println("Saindo do escopo " + escopoAtual);
        escopoAtual--;
    }

    public void analisar(NoAST raiz) {
        raiz.analisar(this);
    }

    // Regra específica: checa se uma classe abstrata está sendo instanciada
    public void verificarClasseAbstrata(String nomeClasse) {
        if (nomeClasse.equals("Animal")) { // Simulação de classe abstrata
            System.out.println("Erro semântico: Não é possível instanciar a classe abstrata 'Animal'.");
        }
    }
}

public class Main {
    public static void main(String[] args) {
        AnalisadorSemantico analisador = new AnalisadorSemantico();

        // Exemplo 1: Declaração e soma válida
        NoDeclaracao decl1 = new NoDeclaracao("x", "int");
        NoDeclaracao decl2 = new NoDeclaracao("y", "int");
        NoExpressao soma = new NoExpressao("+", new NoVariavel("x"), new NoVariavel("y"));

        List<NoAST> programa1 = Arrays.asList(decl1, decl2, soma);
        NoBloco bloco1 = new NoBloco(programa1);
        System.out.println("=== Análise do Programa 1 ===");
        analisador.analisar(bloco1);

        // Exemplo 2: Tipos incompatíveis
```

```
        NoDeclaracao decl3 = new NoDeclaracao("a", "int");
        NoDeclaracao decl4 = new NoDeclaracao("b", "String");
        NoExpressao somaInvalida = new NoExpressao("+", new NoVariavel("a"), new NoVariavel("b"));

        List<NoAST> programa2 = Arrays.asList(decl3, decl4, somaInvalida);
        NoBloco bloco2 = new NoBloco(programa2);
        System.out.println("\n=== Análise do Programa 2 ===");
        analisador.analisar(bloco2);

        // Exemplo 3: Escopo e variável não declarada
        NoDeclaracao decl5 = new NoDeclaracao("temp", "int");
        NoBloco subBloco = new NoBloco(Arrays.asList(decl5));
        NoVariavel varForaEscopo = new NoVariavel("temp");
        List<NoAST> programa3 = Arrays.asList(subBloco, varForaEscopo);
        NoBloco bloco3 = new NoBloco(programa3);
        System.out.println("\n=== Análise do Programa 3 ===");
        analisador.analisar(bloco3);

        // Exemplo 4: Regra específica (classe abstrata)
        System.out.println("\n=== Verificação de Regra Específica ===");
        analisador.verificarClasseAbstrata("Animal");
    }
}
```

Saída Esperada

```
=== Análise do Programa 1 ===
Entrando no escopo 1
Variável 'x' declarada como int no escopo 1
```

```
Variável 'y' declarada como int no escopo 1
Expressão 'x + y' válida (int).
Saindo do escopo 1

=== Análise do Programa 2 ===
Entrando no escopo 1
Variável 'a' declarada como int no escopo 1
Variável 'b' declarada como String no escopo 1
Expressão 'a + b' válida (String).
Saindo do escopo 1

=== Análise do Programa 3 ===
Entrando no escopo 1
Entrando no escopo 2
Variável 'temp' declarada como int no escopo 2
Saindo do escopo 2
Erro semântico: Variável 'temp' não declarada.
Saindo do escopo 1

=== Verificação de Regra Específica ===
Erro semântico: Não é possível instanciar a classe abstrata 'Animal'.
```

Explicação das Abordagens na Implementação

1. **Percurso da Árvore Sintática (AST)**:
    - A classe `NoAST` e suas subclasses (`NoDeclaracao` , `NoExpressao` , etc.) formam uma AST simplificada.
    - O método `analisar` percorre a árvore recursivamente, analisando cada nó.

2. **Tabela de Símbolos**:

    - O `HashMap` em `AnalisadorSemantico` armazena símbolos com chaves únicas (nome + escopo).
    - `declararVariavel` e `buscarVariavel` gerenciam declarações e resoluções.

3. **Anotação de Tipos**:

    - Cada nó da AST tem um campo `tipoAnotado`, preenchido durante a análise.
    - Exemplo: em `NoExpressao`, o tipo resultante da soma é anotado como `int` ou `String`.

4. **Análise Baseada em Regras**:

    - O método `verificarClasseAbstrata` simula uma regra específica de Java (proibição de instanciar classes abstratas).
    - Pode ser expandido para outras regras, como modificadores de acesso.

Como Funciona

- **Programa 1**: Declara `x` e `y` como `int` e verifica a soma (válida).
- **Programa 2**: Declara `a` como `int` e `b` como `String`, aceitando a concatenação.
- **Programa 3**: Testa escopo, mostrando erro ao acessar `temp` fora do bloco.
- **Programa 4**: Demonstra uma regra específica com classe abstrata.

Expansão Possível

- **Herança e Métodos**: Adicionar nós para classes e chamadas de métodos, verificando compatibilidade.
- **Controle de Fluxo**: Suportar `if`, `while`, etc., com análise de escopo.
- **Tipos Complexos**: Incluir arrays, objetos e conversões implícitas.