Compiladores

Aula 15: Análise Semântica: Tabelas de Símbolos

Prof. Aléssio Miranda Júnior
alessio@cefetmg.br
CEFET-MG - Campus Timóteo

Fevereiro de 2026

1 Objetivos

- Compreender o papel da Análise Semântica no pipeline do compilador.
- Identificar classes de erros que a análise sintática sozinha não consegue detectar.
- Projetar e implementar uma Tabela de Símbolos em Java.
- Gerenciar escopos aninhados (léxicos) com pilha de tabelas.
- Implementar a resolução de nomes e verificação de declarações como um Visitor sobre a AST.
- Aplicar os conceitos ao compilador do Mini-Pascal.

2 Introdução: O Limite das Gramáticas Livres de Contexto

Nas aulas anteriores, construímos o front-end léxico e sintático do compilador. A análise sintática, baseada em Gramáticas Livres de Contexto (GLC), garante que o programa obedece à **estrutura** da linguagem — parênteses balanceados, operadores nos lugares corretos, comandos bem formados. Porém, há uma classe inteira de erros que a GLC é **matematicamente incapaz** de capturar.

△ Importante

Um programa pode ser sintaticamente perfeito e semanticamente absurdo. A sentença "o número 42 é maior que a cor azul" é gramaticalmente correta em português, mas não faz sentido. Da mesma forma, `x := "hello" + true` pode ser sintaticamente válida, mas é semanticamente incoerente.

Os erros que dependem de **contexto** incluem:

- **Variável não declarada:** Usar `y` sem nunca ter escrito `var y : integer`.
- **Declaração duplicada:** Declarar `var x` duas vezes no mesmo escopo.
- **Incompatibilidade de tipos:** Atribuir um `boolean` a uma variável `integer`.
- **Número incorreto de argumentos:** Chamar `f(1, 2)` quando `f` espera 3 parâmetros.
- **Uso indevido de categoria:** Usar o nome de uma função como variável, ou vice-versa.

A **Análise Semântica** é a fase do compilador responsável por detectar esses erros. Ela percorre a AST construída pelo parser e verifica regras dependentes de contexto, utilizando como principal ferramenta a **Tabela de Símbolos**.

3 A Tabela de Símbolos

3.1 Definição e Propósito

A Tabela de Símbolos é a **estrutura de dados central** do compilador. Ela funciona como um "dicionário" que associa cada identificador declarado no programa-fonte às suas propriedades relevantes. Todas as fases posteriores do compilador — verificação de tipos, geração de código intermediário, alocação de registradores — consultam a tabela de símbolos para obter informações sobre as variáveis e funções.

As informações tipicamente armazenadas para cada entrada são:

- **Nome:** O lexema do identificador (`"x"`, `"somaTotal"`, `"calcular"`).
- **Tipo:** O tipo da variável (`"integer"`, `"real"`, `"boolean"`) ou a assinatura da função.
- **Categoria:** Se é variável, constante, função ou parâmetro.
- **Nível de Escopo:** 0 para global, 1 para o primeiro bloco aninhado, etc.
- **Linha de Declaração:** Para gerar mensagens de erro precisas.
- **Endereço/Offset:** Para geração de código (alocação em memória/registrador).

3.2 Operações Fundamentais

A tabela de símbolos deve suportar quatro operações essenciais:

1. `insert(name, info)` — Chamada ao processar uma **declaração**. Insere o símbolo na tabela do escopo atual.
2. `lookup(name)` — Chamada ao encontrar um **uso** de identificador. Busca o símbolo na tabela, começando pelo escopo mais interno.
3. `enterScope()` — Chamada ao entrar em um **bloco** (`begin...end`). Empilha uma nova tabela.
4. `exitScope()` — Chamada ao sair de um bloco. Desempilha a tabela do escopo atual.

3.3 Implementação em Java

Listing 1: Classe Symbol — entrada da tabela

```java
public class Symbol {
    String name;          // "x", "somaTotal"
    String type;          // "integer", "real", "boolean"
    String category;      // "variable", "constant", "function"
    int scopeLevel;       // 0 = global, 1 = primeiro bloco...
    int declarationLine; // Linha no codigo-fonte

    public Symbol(String name, String type,
                  String category, int line) {
        this.name = name;
        this.type = type;
        this.category = category;
        this.declarationLine = line;
    }
}
```

Listing 2: Tabela de Símbolos simples (escopo único)

```java
public class SymbolTable {
    private Map<String, Symbol> symbols = new HashMap<>();

    public void insert(String name, Symbol sym) {
        if (symbols.containsKey(name)) {
            throw new SemanticError(
                "Variavel '" + name + "' ja declarada"
                + " na linha " + sym.declarationLine);
        }
        symbols.put(name, sym);
    }

    public Symbol lookup(String name) {
        return symbols.get(name); // null se nao encontrar
    }
}
```

4 Gerenciamento de Escopos

4.1 O Problema dos Escopos Aninhados

Linguagens de programação modernas permitem **blocos aninhados** onde variáveis locais podem ter o mesmo nome de variáveis de escopos externos. Esse fenômeno é chamado de **sombreamento** (*shadowing*):

▷ Exemplo

Exemplo em Mini-Pascal:

```
program teste;
var x : integer;          // x no escopo global
begin
    x := 10;
    begin
        var x : real;     // x no escopo local (sombreia)
        x := 3.14;        // refere-se ao x : real
    end;
    write(x);             // refere-se ao x : integer (= 10)
end.
```

O `x : real` do bloco interno "sombreia" o `x : integer` do escopo global. Ao sair do bloco, o `x` global volta a ser visível.

4.2 A Pilha de Tabelas (Scoped Symbol Table)

A solução clássica para gerenciar escopos é manter uma **pilha de tabelas hash**, onde cada tabela representa um escopo:

- **enterScope():** Empilha uma nova tabela vazia (ao entrar em um `begin...end`).
- **exitScope():** Desempilha a tabela do topo (ao sair do bloco).
- **insert():** Insere na tabela do topo (escopo atual).
- **lookup():** Busca do topo para a base — a primeira ocorrência encontrada é a que vale (escopo mais interno vence).

Listing 3: Tabela de Símbolos com escopos aninhados

```
1  public class ScopedSymbolTable {
2      private Deque<Map<String, Symbol>> scopes = new ArrayDeque
           <>();
3      private int currentLevel = 0;
4
5      public ScopedSymbolTable() {
6          enterScope(); // escopo global inicial
7      }
8
9      public void enterScope() {
10         scopes.push(new HashMap<>());
11         currentLevel++;
12     }
13
14     public void exitScope() {
```

```
        scopes.pop();
        currentLevel--;
    }

    public void insert(String name, Symbol sym) {
        Map<String, Symbol> currentScope = scopes.peek();
        if (currentScope.containsKey(name)) {
            throw new SemanticError(
                "Variavel '" + name
                + "' ja declarada neste escopo");
        }
        sym.scopeLevel = currentLevel;
        currentScope.put(name, sym);
    }

    public Symbol lookup(String name) {
        // Busca do escopo mais interno para o mais externo
        for (Map<String, Symbol> scope : scopes) {
            Symbol sym = scope.get(name);
            if (sym != null) return sym;
        }
        return null; // nao declarada em nenhum escopo
    }

    // Busca apenas no escopo atual (util para checar duplicatas)
    public Symbol lookupLocal(String name) {
        return scopes.peek().get(name);
    }
}
```

• Teoria

Complexidade: A operação `insert()` é $O(1)$ (acesso direto ao HashMap do topo). A operação `lookup()` é $O(d)$ no pior caso, onde d é a profundidade de aninhamento de escopos. Na prática, d raramente ultrapassa 5–10, então o custo é negligível.

5 Integração com a AST: O Visitor Semântico

5.1 A Arquitetura do Analisador Semântico

O analisador semântico é implementado como um **Visitor** que percorre a AST e utiliza a Tabela de Símbolos como estrutura auxiliar. O Visitor segue um padrão simples:

- Ao visitar um nó de **declaração** (`VarDecl`): insere o símbolo na tabela.
- Ao visitar um nó de **uso** (`VarRef`): busca o símbolo na tabela.
- Ao visitar um nó de **bloco** (`Block`): abre e fecha escopo.

- Ao visitar um nó de **atribuição** (`AssignStmt`): busca a variável e verifica compatibilidade de tipo.

Listing 4: Visitor Semântico — Declarações e Usos

```
public class SemanticVisitor extends ASTBaseVisitor<Void> {
    private ScopedSymbolTable symTable = new ScopedSymbolTable();
    private List<String> errors = new ArrayList<>();

    @Override
    public Void visitVarDecl(VarDecl node) {
        Symbol sym = new Symbol(node.name, node.type,
                                "variable", node.line);
        try {
            symTable.insert(node.name, sym);
        } catch (SemanticError e) {
            errors.add("[linha " + node.line + "] " + e.
               getMessage());
        }
        return null;
    }

    @Override
    public Void visitVarRef(VarRef node) {
        Symbol sym = symTable.lookup(node.name);
        if (sym == null) {
            errors.add("[linha " + node.line + "] Variavel '"
                + node.name + "' nao declarada");
        } else {
            node.resolvedType = sym.type; // anota o tipo na AST
        }
        return null;
    }

    @Override
    public Void visitBlock(Block node) {
        symTable.enterScope();
        for (Stmt stmt : node.statements) {
            visit(stmt);
        }
        symTable.exitScope();
        return null;
    }

    @Override
    public Void visitAssignStmt(AssignStmt node) {
        Symbol sym = symTable.lookup(node.varName);
        if (sym == null) {
            errors.add("[linha " + node.line + "] Variavel '"
                + node.varName + "' nao declarada");
```

```
45          }
46          visit(node.value); // processa a expressao
47          return null;
48      }
49
50      public List<String> getErrors() { return errors; }
51  }
```

5.2 Boas Práticas de Reportagem de Erros

Um compilador de qualidade **não para no primeiro erro**. Ele acumula todos os erros semânticos encontrados e os reporta ao final, para que o programador possa corrigi-los de uma só vez:

Listing 5: Execução do analisador semântico no pipeline

```
// No pipeline principal do compilador:
SemanticVisitor semantic = new SemanticVisitor();
semantic.visit(ast); // percorre toda a AST

List<String> errors = semantic.getErrors();
if (!errors.isEmpty()) {
    System.err.println("Erros semanticos encontrados:");
    for (String err : errors) {
        System.err.println("  " + err);
    }
    System.exit(1);
}
// Se chegou aqui, a AST esta semanticamente correta
```

6 Exercício Guiado

Considere o seguinte programa Mini-Pascal e faça o **trace manual** da tabela de símbolos:

```
program exercicio;
var a : integer;
var b : real;
begin
    a := 10;
    begin
        var c : boolean;
        c := true;
        a := a + 1;
    end;
    b := a;
end.
```

Tarefas:

1 Quantos escopos são criados ao longo da execução?

2 Desenhe o estado da pilha de escopos no momento em que `c := true` é processado.

3 A variável `c` é visível na linha `b := a`? Por quê?

4 Se adicionássemos `var a : real` no bloco interno, o que aconteceria com o acesso a `a` dentro e fora do bloco?

5 Qual o tipo resolvido de `a` na expressão `a + 1`?

Solução do trace:

Passo	Ação	Estado da Pilha
1	Início	Escopo 0: {}
2	`var a : integer`	Escopo 0: {a:integer}
3	`var b : real`	Escopo 0: {a:integer, b:real}
4	`begin` (externo)	Escopo 0 + Escopo 1: {}
5	`a := 10`	lookup(a) → Escopo 0, OK
6	`begin` (interno)	Esc.0 + Esc.1 + Esc.2: {}
7	`var c : boolean`	Esc.2: {c:boolean}
8	`c := true`	lookup(c) → Esc.2, OK
9	`end` (interno)	Pop Esc.2. Variável c desaparece.
10	`b := a`	lookup(b) → Esc.0, lookup(a) → Esc.0, OK

7 Referências

- Aho, A. V., Lam, M. S., Sethi, R., & Ullman, J. D. (2006). *Compilers: Principles, Techniques, and Tools*. 2ª Edição. Cap. 2 (Seção 2.7) e Cap. 6 (Seção 6.1–6.3).
- Cooper, K., & Torczon, L. (2011). *Engineering a Compiler*. 2ª Edição. Cap. 5 (Seção 5.5).
- Appel, A. W. (2002). *Modern Compiler Implementation in Java*. Cap. 5 (Semantic Analysis).
- Nystrom, R. (2021). *Crafting Interpreters*. Cap. 8 (Statements and State) e Cap. 11 (Resolving and Binding).