Compiladores

Aula 15: Análise Semântica: Tabelas de Símbolos

Prof. Aléssio Miranda Júnior
alessio@cefetmg.br

CEFET-MG - Campus Timoteo
Dep. Engenharia de Computação

Abril de 2026

Sumário

1. Objetivos
2. Motivação: Além da Sintaxe
3. A Tabela de Símbolos
4. Gerenciamento de Escopos
5. Visitor Semântico sobre a AST
6. Funções e Procedimentos na Tabela
7. Erros Semânticos Comuns
8. Conexão com Geração de Código
9. Exercícios Guiados
10. Perguntas de Reflexão
11. Conclusão e Próximos Passos
12. Referências

Objetivos da Aula

- Compreender o papel da Análise Semântica no pipeline do compilador.
- Identificar erros que a sintaxe 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.

O Que a Sintaxe NÃO Detecta?

Um programa pode ser **sintaticamente perfeito** e ainda assim estar **semanticamente errado**:

Erros de Declaração:

- Usar variável não declarada
- Declarar a mesma variável duas vezes
- Chamar função com número errado de argumentos

Erros de Tipo:

- Somar `string` com `integer`
- Atribuir `boolean` a variável `real`
- Usar `integer` como condição de `if`

Papel da Análise Semântica

Verificar tudo que **depende de contexto** — ou seja, tudo que uma Gramática Livre de Contexto não consegue expressar.

Exemplos Concretos de Erros Semânticos

Todos os exemplos abaixo são **sintaticamente corretos**, mas **semanticamente errados**:

Mini-Pascal

```
var x :  integer;
begin
  y := 10; // y não declarada
  x := "abc"; // tipo incompatível
end.
```

Java (análogo)

```
int x = 10;
System.out.println(z); // z?
x = true; // int ← boolean
```

Insight

O parser aceita qualquer expressão à direita de `:=`. Cabe à **análise semântica** verificar se ela faz *sentido*.

Analogia: Gramática vs Semântica

Linguagem Natural:

- "Colorless green ideas sleep furiously."
- Frase **gramaticalmente perfeita** (sujeito + verbo + advérbio).
- Mas **sem sentido** — ideias não dormem, verde e incolor são contraditórios.

Linguagem de Programação:

- `x := y + z;`
- Sintaticamente perfeito: `id := expr;`
- Mas: `y` foi declarada? `z` é do tipo certo? `x` é compatível com o resultado?

Conclusão

Sintaxe = **forma**. Semântica = **significado**. A análise semântica é o "bom senso" do compilador.

Onde Estamos no Pipeline?

A análise semântica **percorre a AST** (via Visitor) e:

1. Popula a **Tabela de Símbolos** com declarações.
2. Verifica que cada uso de variável/função está **declarado**.
3. Anota os nós da AST com **informações de tipo** (próxima aula).

Saída do analisador semântico

AST **decorada** (nós anotados com tipo) + Tabela de Símbolos preenchida → prontas para *codegen*.

O Que é a Tabela de Símbolos?

É o **“banco de dados” do compilador**. Mapeia **nomes** (strings) em **informações** sobre cada identificador declarado no programa.

Operações fundamentais:

O Que é a Tabela de Símbolos?

É o **"banco de dados" do compilador**. Mapeia **nomes** (strings) em **informações** sobre cada identificador declarado no programa.

Operações fundamentais:

- `insert(name, info)` — declaração

O Que é a Tabela de Símbolos?

É o **"banco de dados" do compilador**. Mapeia **nomes** (strings) em **informações** sobre cada identificador declarado no programa.

Operações fundamentais:

- `insert(name, info)` — declaração
- `lookup(name)` — uso

O Que é a Tabela de Símbolos?

É o **“banco de dados” do compilador**. Mapeia **nomes** (strings) em **informações** sobre cada identificador declarado no programa.

Operações fundamentais:

- `insert(name, info)` — declaração
- `lookup(name)` — uso
- `lookupLocal(name)` — só escopo atual

O Que é a Tabela de Símbolos?

É o **"banco de dados" do compilador**. Mapeia **nomes** (strings) em **informações** sobre cada identificador declarado no programa.

Operações fundamentais:

- `insert(name, info)` — declaração
- `lookup(name)` — uso
- `lookupLocal(name)` — só escopo atual
- `enterScope()` — entrar em bloco
- `exitScope()` — sair de bloco

O Que é a Tabela de Símbolos?

É o **"banco de dados" do compilador**. Mapeia **nomes** (strings) em **informações** sobre cada identificador declarado no programa.

Operações fundamentais:

- `insert(name, info)` — declaração
- `lookup(name)` — uso
- `lookupLocal(name)` — só escopo atual
- `enterScope()` — entrar em bloco
- `exitScope()` — sair de bloco

Quem usa a tabela?

- **Análise semântica:** verificar declarações, tipos
- **Geração de código:** offsets, endereços
- **Debugger:** mapear endereços $\rightarrow$ nomes
- **IDE:** autocompletar, refatorar

Atributos Armazenados por Símbolo

Atributo	Descrição	Exemplo
Nome	Lexema do identificador	`"x"`, `"calcular"`
Tipo	Tipo de dado	`integer`, `real`, `boolean`
Categoria	Papel no programa	variável, constante, função
Escopo	Bloco onde foi declarado	global, função `f`, bloco 2
Linha	Posição no código-fonte	42
Endereço	Offset na pilha / rótulo	`[BP-8]`, `_func_f`
Tamanho	Bytes para alocação	4 (int), 8 (real)
# Params	Nº de parâmetros (funções)	3
Tipo Retorno	Tipo de retorno (funções)	`integer`

Observação

Nem todos os atributos são preenchidos de uma vez. O **endereço** e o **tamanho** podem ser definidos na fase de geração de código.

Escolha da Estrutura de Dados

Estrutura	Insert	Lookup	Uso típico
Lista linear	$O(1)$	$O(n)$	Protótipos simples
Árvore BST	$O(\log n)$	$O(\log n)$	Saída ordenada
Tabela Hash	$O(1)^*$	$O(1)^*$	**Compiladores reais**

*Tempo esperado, assumindo boa função de hash e fator de carga baixo.

Na prática

`HashMap<String, Symbol>` (Java) ou `std::unordered_map` (C++) — acesso em $O(1)$ amortizado. Em Mini-Pascal, o número de símbolos é pequeno, mas o hábito de usar hash vale para compiladores industriais.

Estrutura de Dados: A Classe Symbol

```
public class Symbol {
    String name;          // "x", "somaTotal", "calcular"
    String type;          // "integer", "real", "boolean"
    String category;      // "variable","constant","function"
    int scopeLevel;       // 0 = global, 1 = primeiro bloco
    int declarationLine; // Linha no codigo-fonte
    // Para funcoes/procedimentos:
    List<String> paramTypes;  // tipos dos parametros
    String returnType;        // tipo de retorno (ou null)
}
```

Dica de projeto: usar enum para `category` e uma classe `Type` em vez de `String` torna o código mais robusto.

Implementação Básica em 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");
        }
        symbols.put(name, sym);
    }

    public Symbol lookup(String name) {
        Symbol sym = symbols.get(name);
        if (sym == null) {
            throw new SemanticError(
                "Variavel '" + name + "' nao declarada");
        }
        return sym;
    }
}
```

O Problema dos Escopos Aninhados

Linguagens modernas permitem **blocos aninhados** com variáveis locais:

Código Mini-Pascal

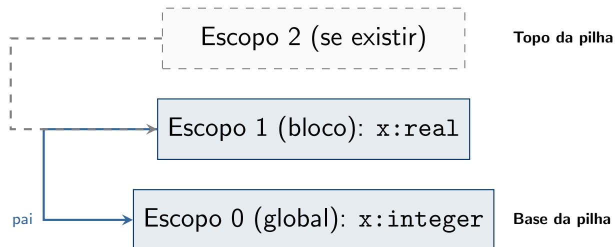
```
var x : integer;
begin
  x := 10;
  begin
    var x : real;
    x := 3.14; // qual x?
  end;
  write(x); // qual x?
end.
```

Regras de Escopo Léxico:

- Variável interna **sombreia** a externa (*shadowing*).
- Ao sair do bloco, a variável interna "desaparece".
- `write(x)` acessa o `x : integer` original.

Pilha de Escopos (Scoped Symbol Table)

A solução clássica: manter uma **pilha de tabelas**, uma por escopo.



Lookup: busca do topo para a base. A primeira ocorrência encontrada é a que vale (escopo mais interno vence).

Implementação da Pilha de Escopos

```
public class ScopedSymbolTable {
    private Deque<Map<String, Symbol>> scopes
        = new ArrayDeque<>();

    public void enterScope() {
        scopes.push(new HashMap<>());
    }
    public void exitScope() {
        scopes.pop();
    }
    public void insert(String name, Symbol sym) {
        scopes.peek().put(name, sym);  // escopo atual
    }
    public Symbol lookup(String name) {
        for (Map<String, Symbol> scope : scopes) {
            if (scope.containsKey(name))
                return scope.get(name);
        }
        return null; // nao declarada
    }
}
```

lookupLocal: Detectando Duplicatas

`lookup` busca em **todos** os escopos. Para detectar redeclarações, precisamos de `lookupLocal` — busca **apenas no escopo atual**:

```
public Symbol lookupLocal(String name) {
    return scopes.peek().get(name); // so o topo
}

public void insert(String name, Symbol sym) {
    if (lookupLocal(name) != null) {
        throw new SemanticError(
            "'" + name + "' ja declarada neste escopo");
    }
    scopes.peek().put(name, sym);
}
```

Regra: redeclarar no **mesmo escopo** é erro. Redeclarar em escopo **interno** é *shadowing* (válido, mas pode gerar warning).

Escopo Léxico vs Escopo Dinâmico

Escopo Léxico (Estático):

- O escopo é determinado pela **estrutura do código-fonte**.
- Pode ser resolvido em **tempo de compilação**.
- Usado por: Pascal, C, Java, Python, Rust.
- **É o que implementaremos!**

Escopo Dinâmico:

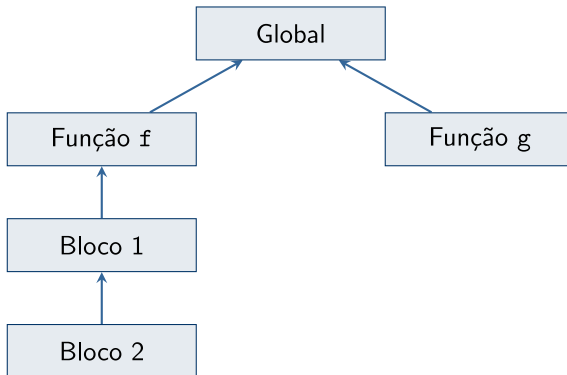
- O escopo depende da **ordem de execução** (call stack).
- Só pode ser resolvido em **tempo de execução**.
- Usado por: Bash, Emacs Lisp, LaTeX (!)
- Mais difícil de raciocinar e otimizar.

Mini-Pascal

Usa **escopo léxico**. A resolução de nomes é feita na análise semântica, antes da execução.

Alternativa: Árvore de Escopos

Em vez de pilha, manter uma **árvore** onde cada nó é um escopo com ponteiro para o pai:



Vantagem: preserva informações de escopo **após** o fim do bloco (útil para debuggers e IDEs).

Desvantagem: mais memória; a pilha descarta escopos encerrados automaticamente.

Trace Passo-a-Passo da Pilha de Escopos

Código

```
program p;
var x : integer;
begin
  x := 10;
  begin
    var y : real;
    y := 3.14;
  end;
  write(x);
end.
```

Evolução da pilha:

1. `enterScope()` → Pilha: [{ }]

Trace Passo-a-Passo da Pilha de Escopos

Código

```
program p;
var x : integer;
begin
  x := 10;
  begin
    var y : real;
    y := 3.14;
  end;
  write(x);
end.
```

Evolução da pilha:

1. `enterScope()` → Pilha: [{ }]
2. `insert(x, int)` → [{x:int}]

Trace Passo-a-Passo da Pilha de Escopos

Código

```
program p;
var x : integer;
begin
  x := 10;
  begin
    var y : real;
    y := 3.14;
  end;
  write(x);
end.
```

Evolução da pilha:

1. `enterScope()` → Pilha: [{ }]
2. `insert(x, int)` → [{x:int}]
3. `enterScope()` → [{x:int}, { }]

Trace Passo-a-Passo da Pilha de Escopos

Código

```
program p;
var x : integer;
begin
  x := 10;
  begin
    var y : real;
    y := 3.14;
  end;
  write(x);
end.
```

Evolução da pilha:

1. `enterScope()` → Pilha: [{ }]
2. `insert(x, int)` → [{x:int}]
3. `enterScope()` → [{x:int}, { }]
4. `insert(y, real)` → [{x:int}, {y:real}]

Trace Passo-a-Passo da Pilha de Escopos

Código

```
program p;
var x : integer;
begin
  x := 10;
  begin
    var y : real;
    y := 3.14;
  end;
  write(x);
end.
```

Evolução da pilha:

1. `enterScope()` → Pilha: [{ }]
2. `insert(x, int)` → [{x:int}]
3. `enterScope()` → [{x:int}, { }]
4. `insert(y, real)` → [{x:int}, {y:real}]
5. `lookup(y)` → encontra no topo!

Trace Passo-a-Passo da Pilha de Escopos

Código

```
program p;
var x : integer;
begin
  x := 10;
  begin
    var y : real;
    y := 3.14;
  end;
  write(x);
end.
```

Evolução da pilha:

1. `enterScope()` → Pilha: [{ }]
2. `insert(x, int)` → [{x:int}]
3. `enterScope()` → [{x:int}, { }]
4. `insert(y, real)` → [{x:int}, {y:real}]
5. `lookup(y)` → encontra no topo!
6. `exitScope()` → [{x:int}]

Trace Passo-a-Passo da Pilha de Escopos

Código

```
program p;
var x : integer;
begin
  x := 10;
  begin
    var y : real;
    y := 3.14;
  end;
  write(x);
end.
```

Evolução da pilha:

1. `enterScope()` → Pilha: [{ }]
2. `insert(x, int)` → [{x:int}]
3. `enterScope()` → [{x:int}, { }]
4. `insert(y, real)` → [{x:int}, {y:real}]
5. `lookup(y)` → encontra no topo!
6. `exitScope()` → [{x:int}]
7. `lookup(x)` → encontra na base!
8. `lookup(y)` → null! (y saiu de escopo)

Integrando Tabela de Símbolos + AST

O analisador semântico é um **Visitor** que percorre a AST e usa a Tabela de Símbolos:

Integrando Tabela de Símbolos + AST

O analisador semântico é um **Visitor** que percorre a AST e usa a Tabela de Símbolos:

1. **VarDecl:** Insere a variável na tabela (`insert`).

Integrando Tabela de Símbolos + AST

O analisador semântico é um **Visitor** que percorre a AST e usa a Tabela de Símbolos:

1. **VarDecl:** Insere a variável na tabela (`insert`).
2. **VarRef:** Busca a variável na tabela (`lookup`).

Integrando Tabela de Símbolos + AST

O analisador semântico é um **Visitor** que percorre a AST e usa a Tabela de Símbolos:

1. **VarDecl:** Insere a variável na tabela (`insert`).
2. **VarRef:** Busca a variável na tabela (`lookup`).
3. **AssignStmt:** Busca a variável + verifica compatibilidade de tipo.

Integrando Tabela de Símbolos + AST

O analisador semântico é um **Visitor** que percorre a AST e usa a Tabela de Símbolos:

1. **VarDecl:** Insere a variável na tabela (`insert`).
2. **VarRef:** Busca a variável na tabela (`lookup`).
3. **AssignStmt:** Busca a variável + verifica compatibilidade de tipo.
4. **Block:** Abre escopo (`enterScope`) ao entrar, fecha (`exitScope`) ao sair.

Integrando Tabela de Símbolos + AST

O analisador semântico é um **Visitor** que percorre a AST e usa a Tabela de Símbolos:

1. **VarDecl:** Insere a variável na tabela (`insert`).
2. **VarRef:** Busca a variável na tabela (`lookup`).
3. **AssignStmt:** Busca a variável + verifica compatibilidade de tipo.
4. **Block:** Abre escopo (`enterScope`) ao entrar, fecha (`exitScope`) ao sair.
5. **IfStmt / WhileStmt:** Verifica que a condição é `boolean`.

Visitor Semântico: Declarações e Usos

```java
public class SemanticVisitor extends ASTBaseVisitor<Void> {
    private ScopedSymbolTable symTable
        = new ScopedSymbolTable();

    @Override
    public Void visitVarDecl(VarDecl node) {
        Symbol sym = new Symbol(node.name,
            node.type, "variable", node.line);
        symTable.insert(node.name, sym);
        return null;
    }

    @Override
    public Void visitVarRef(VarRef node) {
        Symbol sym = symTable.lookup(node.name);
        if (sym == null) {
            error("Variavel '" + node.name
                + "' nao declarada", node.line);
        }
```

Visitor Semântico: Blocos e Escopos

```java
@Override
public Void visitBlock(Block node) {
    symTable.enterScope();        // abre novo escopo
    for (Stmt stmt : node.statements) {
        visit(stmt);              // processa cada comando
    }
    symTable.exitScope();         // fecha o escopo
    return null;
}

@Override
public Void visitAssignStmt(AssignStmt node) {
    Symbol sym = symTable.lookup(node.varName);
    if (sym == null) {
        error("Variavel '" + node.varName
            + "' nao declarada", node.line);
    }
    visit(node.value); // processa a expressao
    // Verificacao de tipo sera feita na proxima aula
```

Funções e procedimentos introduzem complexidades extras:

Funções e procedimentos introduzem complexidades extras:

1. **Inserção dupla:** a função é inserida no escopo **pai** (para ser chamável), e seus parâmetros são inseridos no escopo **interno** da função.

Funções e Procedimentos

Funções e procedimentos introduzem complexidades extras:

1. **Inserção dupla:** a função é inserida no escopo **pai** (para ser chamável), e seus parâmetros são inseridos no escopo **interno** da função.
2. **Escopo próprio:** ao entrar na função, abre-se um novo escopo; ao sair, fecha-se.

Funções e Procedimentos

Funções e procedimentos introduzem complexidades extras:

1. **Inserção dupla:** a função é inserida no escopo **pai** (para ser chamável), e seus parâmetros são inseridos no escopo **interno** da função.
2. **Escopo próprio:** ao entrar na função, abre-se um novo escopo; ao sair, fecha-se.
3. **Verificação de chamadas:** ao encontrar uma chamada, verificar aridade (nº de args) e tipos dos argumentos.

Funções e Procedimentos

Funções e procedimentos introduzem complexidades extras:

1. **Inserção dupla:** a função é inserida no escopo **pai** (para ser chamável), e seus parâmetros são inseridos no escopo **interno** da função.
2. **Escopo próprio:** ao entrar na função, abre-se um novo escopo; ao sair, fecha-se.
3. **Verificação de chamadas:** ao encontrar uma chamada, verificar aridade (nº de args) e tipos dos argumentos.

Exemplo Mini-Pascal

```
function soma(a:  integer; b:  integer):  integer;
begin
  soma := a + b; // ‘‘soma’’ é o retorno
end;
```

Visitor: Declaração de Função

```java
@Override
public Void visitFuncDecl(FuncDecl node) {
    // 1. Inserir a funcao no escopo ATUAL (pai)
    Symbol funcSym = new Symbol(node.name,
        node.returnType, "function", node.line);
    funcSym.paramTypes = node.getParamTypes();
    symTable.insert(node.name, funcSym);

    // 2. Abrir escopo da funcao
    symTable.enterScope();

    // 3. Inserir parametros no escopo da funcao
    for (Param p : node.params) {
        Symbol paramSym = new Symbol(p.name,
            p.type, "parameter", p.line);
        symTable.insert(p.name, paramSym);
    }
    // 4. Visitar o corpo
    visit(node.body);
```

Visitor: Chamada de Função

```java
@Override
public Void visitFuncCall(FuncCall node) {
    // 1. Verificar se a funcao existe
    Symbol funcSym = symTable.lookup(node.funcName);
    if (funcSym == null) {
        error("Funcao '" + node.funcName
              + "' nao declarada", node.line);
        return null;
    }
    // 2. Verificar que eh uma funcao
    if (!funcSym.category.equals("function")) {
        error("'" + node.funcName
              + "' nao eh uma funcao", node.line);
    }
    // 3. Verificar aridade
    if (node.args.size()
              != funcSym.paramTypes.size()) {
        error("Num. args incorreto", node.line);
    }
```

Catálogo de Erros Semânticos

Erro	Exemplo	Detecção
Variável não declarada	`y := 5` (sem `var y`)	`lookup` retorna null
Declaração duplicada	`var x; var x;`	`lookupLocal` não é null
Tipo incompatível	`x := "abc"` (x é int)	Próxima aula
Função não declarada	`foo()` sem definição	`lookup` retorna null
Nº args errado	`f(1,2)` mas f tem 3 params	Verificar aridade
Uso de função como var	`soma := 10`	Verificar categoria
Var como função	`x(1,2)` (x é variável)	Verificar categoria

Estratégia de Recuperação de Erros

Abordagem ingênua:

- Parar no **primeiro erro** semântico.
- Programador corrige **um erro por vez**.
- Frustrante e lento.

Boa prática:

- **Acumular** erros em uma lista.
- Continuar a análise mesmo após erro.
- Reportar **todos** ao final.

Dica de implementação

Em vez de `throw new SemanticError(...)`, usar: `errors.add(new SemanticError(...));` e continuar a visita. Ao final, se `errors.isEmpty()`, prosseguir para codegen; senão, imprimir todos os erros.

Da Tabela de Símbolos para a Geração de Código

A tabela de símbolos não serve só para verificação — ela é a **ponte** para o código gerado:

Informações para codegen:

- **Offset:** posição na pilha de ativação (`[BP-4]`, `[BP-8]`).
- **Tamanho:** quantos bytes alocar.
- **Rótulo:** para funções (`_func_soma`).
- **Nível de escopo:** para acessar variáveis não-locais (static link).

Fluxo típico:

1. **Passada 1:** preencher nomes, tipos, categorias.
2. **Passada 2:** verificação de tipos.
3. **Passada 3:** atribuir offsets e gerar código.

Etapa 7 do Mini-Pascal

Na geração de bytecode, cada `lookup` retornará o **offset** do símbolo para emitir instruções como `LOAD [BP-4]` ou `STORE [BP-8]`.

Exercício 1: Trace da Tabela de Símbolos

Trace manual para o seguinte programa Mini-Pascal:

Código

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

Perguntas:

1. Quantos escopos são criados?
2. Qual o conteúdo da tabela ao processar `c := true`?
3. A variável `c` é visível na linha `b := a + 1`?
4. Se adicionássemos `var a : real` no bloco interno, o que acontece?

Exercício 2: Trace com Funções

Trace manual incluindo funções:

Código

```
program p;
var x : integer;
function dobro(n: integer):
integer;
begin
  dobro := n * 2;
end;
begin
  x := dobro(5);
end.
```

Perguntas:

1. Em qual escopo o símbolo `dobro` é inserido?
2. Em qual escopo o parâmetro `n` é inserido?
3. Quantos escopos existem quando `dobro := n * 2` é processado?
4. O que acontece se chamarmos `dobro(1, 2)`?
5. `n` é visível no bloco principal?

Para Pensar...

1. Como você representaria **sobrecarga de funções** (mesmo nome, assinaturas diferentes) na tabela de símbolos?

Para Pensar...

1. Como você representaria **sobrecarga de funções** (mesmo nome, assinaturas diferentes) na tabela de símbolos?
2. Que informações adicionais (além de tipo e categoria) podem ser úteis para **otimizações** ou para um **debugger**?

Para Pensar...

1. Como você representaria **sobrecarga de funções** (mesmo nome, assinaturas diferentes) na tabela de símbolos?
2. Que informações adicionais (além de tipo e categoria) podem ser úteis para **otimizações** ou para um **debugger**?
3. Em uma linguagem com **blocos aninhados** mas sem funções, a pilha de tabelas e a árvore de escopos são equivalentes para busca? E para escopo após o fim do bloco?

Para Pensar...

1. Como você representaria **sobrecarga de funções** (mesmo nome, assinaturas diferentes) na tabela de símbolos?
2. Que informações adicionais (além de tipo e categoria) podem ser úteis para **otimizações** ou para um **debugger**?
3. Em uma linguagem com **blocos aninhados** mas sem funções, a pilha de tabelas e a árvore de escopos são equivalentes para busca? E para escopo após o fim do bloco?
4. Como linguagens como **Python** e **JavaScript** tratam o uso de uma variável *antes* de sua declaração no mesmo escopo? (Pesquise: *hoisting*).

Resumo da Aula

- A Análise Semântica detecta erros **dependentes de contexto** que a gramática não captura.
- A **Tabela de Símbolos** armazena nome, tipo, escopo, categoria e endereço de cada identificador.
- Escopos aninhados: **pilha de tabelas** (descarta ao sair) ou **árvore de escopos** (preserva).
- `lookup` busca do topo à base; `lookupLocal` só no escopo atual.
- O analisador semântico é implementado como um **Visitor** sobre a AST.
- Funções: inserir no escopo pai, parâmetros no escopo interno, verificar aridade nas chamadas.

Conexão com o Projeto Mini-Pascal

Etapa 5: tabela de símbolos + resolução de nomes. **Etapa 6:** verificação de tipos. **Etapa 7:** offsets e endereços para codegen.

Próxima Aula: Verificação de Tipos — regras de tipo para cada operador, coerção, e implementação do Type Checker como Visitor.

Referências

- **AHO, A. V. et al.** *Compiladores: Princípios, Técnicas e Ferramentas*. 2ª Edição. Cap. 2 (Seção 2.7) e Cap. 6 (Seção 6.1–6.3).
- **COOPER, K.; TORCZON, L.** *Engineering a Compiler*. 2ª Edição. Cap. 5 (Seção 5.5 — Symbol Tables).
- **APPEL, A. W.** *Modern Compiler Implementation in Java*. Cap. 5 (Semantic Analysis).
- **NYSTROM, R.** *Crafting Interpreters*. Cap. 8 (Statements and State) e Cap. 11 (Resolving and Binding).

Dúvidas?

Obrigado!

Email: alessio@cefetmg.br
Web: alessiojr.com