Compiladores

Aula 16: Verificação de Tipos e Escopo

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

Fevereiro de 2026

1 Objetivos

- Compreender o papel do sistema de tipos em linguagens de programação.
- Diferenciar tipagem estática, dinâmica e inferência de tipos.
- Definir regras de tipo para expressões, atribuições e comandos do Mini-Pascal.
- Implementar um Type Checker como Visitor sobre a AST em Java.
- Entender coerção (conversão implícita) e promoção de tipos.
- Conhecer os fundamentos da inferência de tipos (Hindley-Milner).

2 Introdução: Por Que Verificar Tipos?

Na aula anterior, construímos a Tabela de Símbolos e o Visitor semântico capaz de verificar se cada identificador usado no programa foi previamente declarado. Porém, saber que uma variável `x` existe não é suficiente: precisamos garantir que ela está sendo **usada de forma consistente com seu tipo**.

A principal tarefa da análise semântica é a **verificação de tipos** (*Type Checking*). Cada operador da linguagem espera operandos de certos tipos, e o Type Checker percorre a AST garantindo que essas regras sejam respeitadas.

△ Importante

Sem verificação de tipos, o compilador geraria código que:

- Soma o padrão de bits de um booleano com um inteiro (resultado imprevisível).
- Tenta armazenar 8 bytes (real) em 4 bytes (integer), corrompendo memória.
- Interpreta um endereço de memória como número, causando *crash*.

A verificação de tipos é a **última linha de defesa** antes da geração de código.

3 Sistemas de Tipos

3.1 Tipagem Estática vs. Dinâmica

O momento em que os tipos são verificados define a classificação do sistema de tipos:

- **Tipagem Estática (C, Java, Rust, Mini-Pascal):** Os tipos são verificados **em tempo de compilação**. Erros de tipo são detectados antes da execução do programa. O compilador pode gerar código mais eficiente, pois conhece os tipos antecipadamente.
- **Tipagem Dinâmica (Python, JavaScript, Ruby):** Os tipos são verificados **em tempo de execução**. O compilador/interpretador gera código que verifica tipos antes de cada operação. Mais flexível, mas erros são descobertos tardiamente (potencialmente em produção).

• Teoria

No Mini-Pascal, usamos tipagem estática. Isso significa que nosso Type Checker deve ser capaz de determinar o tipo de **toda expressão** durante a compilação, sem executar o programa.

3.2 Tipos no Mini-Pascal

Nossa linguagem define três tipos primitivos:

Tipo	Valores	Operações	Tamanho
`integer`	$\ldots, -1, 0, 1, 2, \ldots$	$+, -, *, /$, comparações	4 bytes
`real`	$3.14, -0.5, 1.0, \ldots$	$+, -, *, /$, comparações	8 bytes
`boolean`	`true`, `false`	`and, or, not`, $=$, $<>$	1 byte

Table 1: Tipos primitivos do Mini-Pascal

4 Regras de Tipo

O Type Checker precisa de um conjunto de regras que definam, para cada operação, quais combinações de tipos são válidas e qual é o tipo do resultado.

4.1 Expressões Aritméticas

Para os operadores $+, -, *, /$:

Operando Esq.	Operando Dir.	Tipo do Resultado
`integer`	`integer`	`integer`
`real`	`real`	`real`
`integer`	`real`	`real` (promoção do integer)
`real`	`integer`	`real` (promoção do integer)
`boolean`	qualquer	**ERRO semântico**
qualquer	`boolean`	**ERRO semântico**

Table 2: Regras de tipo para operadores aritméticos

4.2 Expressões Relacionais

Para os operadores $=, <>, <, >, <=, >=$:

Operando Esq.	Operando Dir.	Tipo do Resultado
`integer`	`integer`	`boolean`
`real`	`real`	`boolean`
`integer`	`real`	`boolean` (promoção)
`boolean`	`boolean`	`boolean` (apenas = e <>)

Table 3: Regras de tipo para operadores relacionais

Operadores relacionais **sempre** retornam `boolean`, independentemente do tipo dos operandos.

4.3 Comandos

- **Atribuição** (`x := expr`): O tipo de `expr` deve ser **compatível** com o tipo declarado de `x`. Promoção `integer` → `real` é permitida; estreitamento `real` → `integer` é proibido.
- **If / While** (condição): A expressão de condição **deve** ter tipo `boolean`. Usar um `integer` como condição é erro semântico (diferente de C, onde qualquer valor não-zero é "verdadeiro").
- **Write** (`write(expr)`): Aceita qualquer tipo primitivo.
- **Read** (`read(x)`): A variável `x` deve ser `integer` ou `real` (não faz sentido ler um boolean do teclado no Mini-Pascal).

5 Coerção e Promoção de Tipos

5.1 Promoção (Widening)

Quando um `integer` aparece em um contexto que espera `real` (por exemplo, em uma soma com outro `real`), o compilador insere automaticamente uma **conversão implícita** (*implicit cast*). Isso é chamado de **promoção** ou *widening*, porque o tipo de destino é "maior" (mais precisão, mais bytes).

A promoção não perde informação: todo inteiro pode ser representado como número real.

5.2 Estreitamento (Narrowing)

A conversão inversa (`real` → `integer`) é chamada de **estreitamento** ou *narrowing*. Ela **perde informação** (a parte fracionária é descartada). No Mini-Pascal, proibimos estreitamento — o programador deve ser explícito se quiser essa conversão.

5.3 Inserindo Nós de Coerção na AST

Quando o Type Checker detecta que uma promoção é necessária, ele pode **inserir um nó especial** na AST. Esse nó instrui o gerador de código a emitir uma instrução de conversão:

Listing 1: Inserindo nó de coerção na AST

```
// Dentro do Type Checker, ao processar BinaryExpr:
if (lt.equals("integer") && rt.equals("real")) {
    // Promove o operando esquerdo: int -> real
    CastExpr cast = new CastExpr(node.line,
        node.left, "integer", "real");
    node.left = cast;
    node.resolvedType = "real";
}
```

O nó `CastExpr` não existia no código-fonte original — ele é inserido pelo compilador para garantir a corretude da geração de código.

6 Regras Formais de Tipo (Notação de Inferência)

Na teoria de linguagens de programação, as regras de tipo são formalizadas usando **regras de inferência** (também chamadas *typing judgments*). A notação padrão é:

$$\frac{\text{premissas}}{\text{conclusão}}$$

Lê-se: "se as premissas são verdadeiras, então a conclusão é válida". O símbolo Γ representa o **ambiente de tipos** (a tabela de símbolos) e o operador $\vdash$ significa "prova que".

6.1 Regras para Literais

$$\frac{}{\Gamma \vdash n : \texttt{integer}} \quad \text{(T-IntLit)} \tag{1}$$

$$\frac{}{\Gamma \vdash r : \texttt{real}} \quad \text{(T-RealLit)} \tag{2}$$

$$\frac{}{\Gamma \vdash \texttt{true} : \texttt{boolean}} \quad \text{(T-BoolLit)} \tag{3}$$

$$\frac{}{\Gamma \vdash \texttt{false} : \texttt{boolean}} \quad \text{(T-BoolLit)} \tag{4}$$

Essas regras são **axiomas**: não possuem premissas. O tipo de um literal é determinado diretamente pela sua forma sintática.

6.2 Regra para Referência a Variável

$$\frac{(x : T) \in \Gamma}{\Gamma \vdash x : T} \quad \text{(T-Var)}$$

Lê-se: "se x está declarada com tipo T na tabela de símbolos Γ, então x tem tipo T". Essa regra corresponde diretamente ao `lookup` da tabela de símbolos.

6.3 Regras para Expressões Binárias

$$\frac{\Gamma \vdash e_1 : \texttt{integer} \quad \Gamma \vdash e_2 : \texttt{integer} \quad op \in \{+, -, *, /\}}{\Gamma \vdash e_1 \ op \ e_2 : \texttt{integer}} \quad \text{(T-ArithII)}$$

$$\frac{\Gamma \vdash e_1 : T_1 \quad \Gamma \vdash e_2 : T_2 \quad T_1 = \texttt{real} \vee T_2 = \texttt{real} \quad T_1, T_2 \in \{\texttt{integer}, \texttt{real}\}}{\Gamma \vdash e_1 \ op \ e_2 : \texttt{real}} \quad \text{(T-ArithPromo)}$$

$$\frac{\Gamma \vdash e_1 : T_1 \quad \Gamma \vdash e_2 : T_2 \quad T_1, T_2 \in \{\texttt{integer}, \texttt{real}\} \quad op \in \{=, \neq, <, >, \leq, \geq\}}{\Gamma \vdash e_1 \ op \ e_2 : \texttt{boolean}} \quad \text{(T-Rel)}$$

6.4 Regra para Atribuição

$$\frac{(x : T_x) \in \Gamma \quad \Gamma \vdash e : T_e \quad T_e \preceq T_x}{\Gamma \vdash x := e : \texttt{void}} \quad \text{(T-Assign)}$$

Onde $T_e \preceq T_x$ significa que T_e é **compatível por atribuição** com T_x (mesmo tipo, ou promoção `integer` $\rightarrow$ `real`).

6.5 Regra para If e While

$$\frac{\Gamma \vdash e : \texttt{boolean} \quad \Gamma \vdash S_1 : \texttt{void}}{\Gamma \vdash \texttt{if } e \texttt{ then } S_1 : \texttt{void}} \quad \text{(T-If)}$$

• Teoria

Importância das regras formais: Essas regras não são apenas notação matemática — elas servem como **especificação precisa** do que o Type Checker deve implementar. Cada regra mapeia diretamente para um método `visit` no código Java. Compiladores industriais frequentemente incluem as regras formais como comentários no código-fonte.

7 Equivalência de Tipos

Um conceito fundamental em sistemas de tipos é a definição de **quando dois tipos são considerados iguais**. Existem duas abordagens principais:

7.1 Equivalência Nominal

Dois tipos são equivalentes se e somente se possuem o **mesmo nome**. Mesmo que duas definições sejam estruturalmente idênticas, são consideradas diferentes:

```
type Metros = record x: real end;
type Segundos = record x: real end;
var m : Metros;
var s : Segundos;
m := s;   (* ERRO: Metros <> Segundos, mesmo layout *)
```

Linguagens que usam equivalência nominal: **Java, C#, Rust, Pascal padrão**.

7.2 Equivalência Estrutural

Dois tipos são equivalentes se possuem a **mesma estrutura interna**, independentemente do nome:

```
type T1 = record x: integer; y: real end;
type T2 = record x: integer; y: real end;
(* T1 e T2 sao equivalentes estruturalmente *)
```

Linguagens que usam equivalência estrutural: **TypeScript, OCaml, Go (interfaces)**.

◇ Informação

No Mini-Pascal: Como temos apenas tipos primitivos (`integer`, `real`, `boolean`), a distinção entre equivalência nominal e estrutural não se aplica diretamente. Porém, o conceito é essencial para entender sistemas de tipos de linguagens com tipos compostos (records, classes, etc.).

8 Expressões Unárias e Lógicas

Além dos operadores binários, o Type Checker deve tratar operadores unários e lógicos.

8.1 Negação Aritmética

O operador unário `-` (negação) exige um operando numérico:

$$\frac{\Gamma \vdash e : T \quad T \in \{\texttt{integer}, \texttt{real}\}}{\Gamma \vdash -e : T} \quad \text{(T-Neg)}$$

Se o operando for `boolean`, é erro semântico.

8.2 Negação Lógica

O operador `not` exige um operando booleano:

$$\frac{\Gamma \vdash e : \texttt{boolean}}{\Gamma \vdash \texttt{not}\ e : \texttt{boolean}} \quad \text{(T-Not)}$$

8.3 Operadores Lógicos Binários

Os operadores `and` e `or` exigem ambos operandos booleanos:

$$\frac{\Gamma \vdash e_1 : \texttt{boolean} \quad \Gamma \vdash e_2 : \texttt{boolean} \quad op \in \{\texttt{and}, \texttt{or}\}}{\Gamma \vdash e_1\ op\ e_2 : \texttt{boolean}} \quad \text{(T-Logic)}$$

9 Verificação de Tipos em Funções

Funções introduzem verificações adicionais que vão além das expressões simples.

9.1 Verificação de Chamadas

Ao encontrar uma chamada de função, o Type Checker deve verificar:

1. **Existência:** A função foi declarada? (`lookup` na tabela de símbolos)
2. **Aridade:** O número de argumentos passados corresponde ao número de parâmetros declarados?
3. **Tipos dos argumentos:** Cada argumento é compatível com o tipo do parâmetro correspondente?
4. **Tipo de retorno:** O resultado da chamada recebe o tipo de retorno declarado.

Listing 2: Type Checker — chamada de função

```
@Override
public Void visitFuncCall(FuncCall node) {
    Symbol funcSym = symTable.lookup(node.funcName);
    if (funcSym == null || !funcSym.category.equals("function"))
       {
        error("Funcao '" + node.funcName + "' nao declarada",
           node);
        node.resolvedType = "error";
        return null;
    }
    // Verificar aridade
    List<String> paramTypes = funcSym.paramTypes;
    if (node.args.size() != paramTypes.size()) {
        error("Funcao '" + node.funcName + "' espera "
            + paramTypes.size() + " argumentos, recebeu "
            + node.args.size(), node);
    }
    // Verificar tipo de cada argumento
    for (int i = 0; i < node.args.size(); i++) {
        visit(node.args.get(i));
        if (i < paramTypes.size()) {
            String argType = node.args.get(i).resolvedType;
            String paramType = paramTypes.get(i);
            if (!isAssignmentCompatible(paramType, argType)) {
                error("Argumento " + (i+1) + ": esperado '"
                    + paramType + "', recebido '" + argType + "'"
                       , node);
            }
        }
    }
    node.resolvedType = funcSym.returnType;
    return null;
}
```

9.2 Verificação do Corpo da Função

Dentro do corpo de uma função, deve-se verificar que o valor atribuído ao nome da função (que representa o retorno em Pascal) é compatível com o tipo de retorno declarado.

Listing 3: Verificação do retorno da função

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

```
8
9       // Abrir escopo, inserir parametros, visitar corpo
10      symTable.enterScope();
11      currentFunctionReturn = node.returnType; // salvar para
            checar
12      for (Param p : node.params) {
13          symTable.insert(p.name, new Symbol(p.name,
14              p.type, "parameter", p.line));
15      }
16      visit(node.body);
17      symTable.exitScope();
18      currentFunctionReturn = null;
19      return null;
20  }
```

10 Estratégias de Recuperação de Erros de Tipo

Um bom Type Checker não para no primeiro erro — ele tenta continuar a análise para reportar o máximo de erros possível em uma única compilação.

10.1 O Tipo “error” (Poison Type)

Quando um erro de tipo é detectado, o nó recebe o tipo especial `"error"`. Esse tipo é **compatível com qualquer outro tipo** nas verificações subsequentes, evitando erros em cascata:

Listing 4: Tratamento do tipo error

```
1   private boolean isAssignmentCompatible(String target, String src)
        {
2       if ("error".equals(target) || "error".equals(src)) return
            true;
3       if (target.equals(src)) return true;
4       if ("real".equals(target) && "integer".equals(src)) return
            true;
5       return false;
6   }
```

△ Importante

Sem o poison type, um único erro de tipo pode gerar dezenas de erros em cascata. Por exemplo, se `x + y` gera erro, todas as expressões que usam esse resultado também gerariam erro, confundindo o programador.

10.2 Acumulação de Erros

Em vez de lançar exceções, o Type Checker acumula erros em uma lista e continua a análise:

Listing 5: Acumulação de erros semânticos

```
private List<String> errors = new ArrayList<>();

private void error(String msg, ASTNode node) {
    errors.add("Linha " + node.line + ": " + msg);
    // NAO lanca excecao --- continua a analise
}

public boolean hasErrors() { return !errors.isEmpty(); }
public List<String> getErrors() { return errors; }
```

11 Implementação do Type Checker

O Type Checker é implementado como um **Visitor** sobre a AST. Ele estende o Visitor semântico da aula anterior, adicionando a lógica de verificação de tipos.

Listing 6: Type Checker — expressões binárias

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

    @Override
    public Void visitBinaryExpr(BinaryExpr node) {
        visit(node.left);   // resolve tipo do filho esquerdo
        visit(node.right);  // resolve tipo do filho direito

        String lt = node.left.resolvedType;
        String rt = node.right.resolvedType;

        if (isArithmeticOp(node.operator)) {
            // +, -, *, /
            if ("boolean".equals(lt) || "boolean".equals(rt)) {
                error("Operador aritmetico com boolean", node);
                node.resolvedType = "error";
            } else if ("real".equals(lt) || "real".equals(rt)) {
                node.resolvedType = "real"; // promocao
            } else {
                node.resolvedType = "integer";
            }
        } else if (isRelationalOp(node.operator)) {
            // =, <>, <, >, <=, >=
            node.resolvedType = "boolean";
        }
```

```
        return null;
    }
}
```

Listing 7: Type Checker — atribuição

```
@Override
public Void visitAssignStmt(AssignStmt node) {
    Symbol sym = symTable.lookup(node.varName);
    if (sym == null) {
        error("Variavel '" + node.varName
            + "' nao declarada", node);
        return null;
    }
    visit(node.value); // resolve o tipo da expressao

    String varType  = sym.type;
    String exprType = node.value.resolvedType;

    if (!isAssignmentCompatible(varType, exprType)) {
        error("Tipo incompativel: nao pode atribuir '"
            + exprType + "' a variavel do tipo '"
            + varType + "'", node);
    }
    return null;
}

private boolean isAssignmentCompatible(String target, String src)
    {
    if (target.equals(src)) return true;
    // Promocao: integer -> real e permitida
    if ("real".equals(target) && "integer".equals(src)) return
       true;
    return false;
}
```

Listing 8: Type Checker — condições

```
@Override
public Void visitIfStmt(IfStmt node) {
    visit(node.condition);
    if (!"boolean".equals(node.condition.resolvedType)) {
        error("Condicao do 'if' deve ser boolean, encontrado: '"
            + node.condition.resolvedType + "'", node);
    }
    visit(node.thenBranch);
    if (node.elseBranch != null) {
        visit(node.elseBranch);
    }
    return null;
```

```
}

@Override
public Void visitWhileStmt(WhileStmt node) {
    visit(node.condition);
    if (!"boolean".equals(node.condition.resolvedType)) {
        error("Condicao do 'while' deve ser boolean, encontrado: '"
            + node.condition.resolvedType + "'", node);
    }
    visit(node.body);
    return null;
}
```

Listing 9: Type Checker — literais

```
@Override
public Void visitNumberLiteral(NumberLiteral node) {
    node.resolvedType = "integer";
    return null;
}

@Override
public Void visitRealLiteral(RealLiteral node) {
    node.resolvedType = "real";
    return null;
}

@Override
public Void visitBoolLiteral(BoolLiteral node) {
    node.resolvedType = "boolean";
    return null;
}

@Override
public Void visitVarRef(VarRef node) {
    Symbol sym = symTable.lookup(node.name);
    if (sym != null) {
        node.resolvedType = sym.type;
    } else {
        node.resolvedType = "error";
    }
    return null;
}
```

12 Inferência de Tipos (Visão Geral)

Embora o Mini-Pascal não utilize inferência de tipos, é importante conhecer o conceito, pois linguagens modernas como Kotlin, Rust, Haskell e até Java (com `var` a partir do Java 10) o implementam.

12.1 O Conceito

Na inferência de tipos, o programador **não declara explicitamente** o tipo de uma variável. O compilador **deduz** o tipo a partir do contexto de uso:

```
var x = 42;          // compilador infere x : integer
var y = x + 3.14;    // compilador infere y : real
var z = x > 0;       // compilador infere z : boolean
```

12.2 O Algoritmo de Hindley-Milner

O algoritmo clássico de inferência de tipos funciona em três passos:

1. **Geração de restrições:** Cada expressão gera uma "equação de tipo". Por exemplo, `x + 3.14` gera a restrição: "o tipo de x deve ser numérico e o resultado é real".
2. **Unificação:** O algoritmo resolve o sistema de equações, encontrando uma atribuição de tipos que satisfaz todas as restrições simultaneamente.
3. **Substituição:** Os tipos inferidos são propagados de volta para a AST, anotando cada nó com seu tipo resolvido.

◇ Informação

Para saber mais: O livro *Types and Programming Languages* de Benjamin Pierce é a referência definitiva sobre sistemas de tipos e inferência.

13 Exercício Guiado

Analise o seguinte programa Mini-Pascal e identifique **todos os erros semânticos de tipo**:

```
program exercicio;
var a : integer;
var b : real;
var c : boolean;
begin
    a := 10;              // (1)
    b := a + 3.5;         // (2)
    c := a + b;           // (3)
    if a + 1 then         // (4)
```

```
        a := a + 1;
    b := c;                 // (5)
    a := b;                 // (6)
    c := a > b;             // (7)
    if c then               // (8)
        b := a;
end.
```

Análise linha a linha:

1 `a := 10`: `integer := integer`. **OK**.

2 `b := a + 3.5`: `a` é integer, `3.5` é real ⇒ promoção ⇒ resultado real. `b` é real. **OK**.

3 `c := a + b`: `a + b` resulta em real (promoção). `c` é boolean. **ERRO: real → boolean**.

4 `if a + 1`: `a + 1` resulta em integer. Condição deve ser boolean. **ERRO: integer como condição**.

5 `b := c`: `c` é boolean, `b` é real. **ERRO: boolean → real**.

6 `a := b`: `b` é real, `a` é integer. **ERRO: narrowing (real → integer)**.

7 `c := a > b`: `a > b` resulta em boolean (operador relacional). `c` é boolean. **OK**.

8 `if c then`: `c` é boolean. Condição correta. **OK**. Corpo: `b := a`, promoção integer → real. **OK**.

Total: 4 erros semânticos de tipo (linhas 3, 4, 5 e 6).

13.1 Exercício 2: Verificação com Funções

Analise o programa e identifique os erros semânticos:

```
program exercicio2;
var x : integer;
var y : real;

function dobro(n: integer): integer;
begin
    dobro := n * 2;
end;

function media(a: real; b: real): real;
begin
    media := (a + b) / 2.0;
end;
```

```
begin
    x := dobro(5);          (* (1) *)
    y := dobro(3.14);       (* (2) *)
    x := media(1, 2);       (* (3) *)
    y := media(1, 2);       (* (4) *)
    x := dobro(true);       (* (5) *)
    y := dobro(1, 2);       (* (6) *)
end.
```

Análise:

1 `x := dobro(5)`: argumento integer, parâmetro integer. Retorno integer, variável integer. **OK**.

2 `y := dobro(3.14)`: argumento `real`, parâmetro espera `integer`. **ERRO: narrowing real → integer no argumento**.

3 `x := media(1, 2)`: argumentos integer, parâmetros real — promoção OK. Mas retorno é `real` e `x` é `integer`. **ERRO: narrowing real → integer na atribuição**.

4 `y := media(1, 2)`: promoção dos argumentos OK, retorno real, variável real. **OK**.

5 `dobro(true)`: argumento boolean, parâmetro integer. **ERRO: boolean incompatível com integer**.

6 `dobro(1, 2)`: 2 argumentos, função espera 1. **ERRO: aridade incorreta**.

13.2 Exercício 3: Trace do Type Checker na AST

Considere a expressão `a + 3.5 > b * 2` onde `a : integer` e `b : real`. Trace a propagação de tipos na AST:

1 `a` → `integer` (lookup na tabela)

2 `3.5` → `real` (literal real)

3 `a + 3.5`: integer + real → promoção → `real`

4 `b` → `real` (lookup na tabela)

5 `2` → `integer` (literal inteiro)

6 `b * 2`: real * integer → promoção → `real`

7 `(a + 3.5) > (b * 2)`: real > real → `boolean`

O Type Checker percorre a AST em **pós-ordem**: primeiro resolve os filhos, depois o pai. Isso garante que, ao processar um nó binário, os tipos dos operandos já estão disponíveis.

14 Conexão com a Geração de Código

As informações de tipo anotadas na AST pelo Type Checker são essenciais para a geração de código:

- **Seleção de instruções:** Operações com `integer` usam instruções como `iadd`, `imul`; operações com `real` usam `fadd`, `fmul`.
- **Nós de coerção:** O nó `IntToReal` gera a instrução `i2f` (integer-to-float) no bytecode.
- **Alocação de memória:** `integer` aloca 4 bytes; `real` aloca 8 bytes; `boolean` aloca 1 byte na pilha de ativação.
- **Formatação de E/S:** O comando `write` precisa saber o tipo para escolher o formato de impressão correto.

▶ Prática

Pipeline completo: Código-fonte → Léxico → Sintático → AST → **Tabela de Símbolos (Aula 15)** → **Type Checker (Aula 16)** → AST anotada → Geração de Código (Aula 17).

✓

Resumo da Aula 16

1. O **Type Checker** garante que cada operação recebe operandos de tipos compatíveis.
2. Tipagem pode ser **estática** (compilação) ou **dinâmica** (execução); **forte** (conversões explícitas) ou **fraca** (conversões implícitas).
3. **Regras formais de tipo** (notação de inferência) especificam precisamente o comportamento do Type Checker.
4. **Equivalência de tipos**: nominal (por nome) vs. estrutural (por layout).
5. **Promoção** (integer → real) é permitida; **estreitamento** (real → integer) é proibido.
6. Nós de **coerção** (`IntToReal`) são inseridos na AST para guiar o gerador de código.
7. O **poison type** (`"error"`) evita erros em cascata durante a análise.
8. A verificação de **funções** inclui aridade, tipos dos argumentos e tipo de retorno.

15 Referências

- Aho, A. V., Lam, M. S., Sethi, R., & Ullman, J. D. (2006). *Compilers: Principles, Techniques, and Tools*. 2ª Edição. Cap. 6 (Seções 6.3–6.5: Type Checking).
- Cooper, K., & Torczon, L. (2011). *Engineering a Compiler*. 2ª Edição. Cap. 4 (Context-Sensitive Analysis, Type Systems).
- Pierce, B. C. (2002). *Types and Programming Languages*. MIT Press. Caps. 1–3 (Untyped/Typed Systems), Cap. 8 (Typed Arithmetic Expressions), Cap. 22 (Type Reconstruction).
- Nystrom, R. (2021). *Crafting Interpreters*. Cap. 7 (Evaluating Expressions), Cap. 12 (Classes).
- Cardelli, L. (1996). *Type Systems*. ACM Computing Surveys, 28(1), 263–264.
- Milner, R. (1978). *A Theory of Type Polymorphism in Programming*. Journal of Computer and System Sciences, 17(3), 348–375.