Compiladores

Aula 16: Verificação de Tipos e Escopo

Prof. Aléssio Miranda Júnior
alessio@cefetmg.br

CEFET-MG - Campus Timoteo
Dep. Engenharia de Computação

Abril de 2026

Sumário

Objetivos da Aula

- 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.
- Implementar um Type Checker como Visitor sobre a AST.
- Entender coerção (conversão implícita) e promoção de tipos.
- Aplicar a verificação de tipos ao compilador do Mini-Pascal.

Onde Paramos na Aula Anterior?

Na Aula 15, construímos a **Tabela de Símbolos** e o Visitor semântico que:

- Registra declarações (`insert`)
- Verifica usos (`lookup`)
- Gerencia escopos (`enterScope`/`exitScope`)

O que falta?

Onde Paramos na Aula Anterior?

Na Aula 15, construímos a **Tabela de Símbolos** e o Visitor semântico que:

- Registra declarações (`insert`)
- Verifica usos (`lookup`)
- Gerencia escopos (`enterScope`/`exitScope`)

O que falta?

"A variável x existe. Mas posso somar x com "hello"?"

Precisamos de **regras de tipo** que governem cada operação da linguagem.

O Que é um Sistema de Tipos?

Um **sistema de tipos** é um conjunto de regras que associa tipos a construções do programa e verifica a **consistência** dessas associações.

Tipagem Estática (C, Java, Rust)

O Que é um Sistema de Tipos?

Um **sistema de tipos** é um conjunto de regras que associa tipos a construções do programa e verifica a **consistência** dessas associações.

Tipagem Estática (C, Java, Rust)

- Tipos verificados **em compilação**.
- Erros detectados antes da execução.
- Mais seguro e eficiente.
- **Nosso foco no Mini-Pascal.**

O Que é um Sistema de Tipos?

Um **sistema de tipos** é um conjunto de regras que associa tipos a construções do programa e verifica a **consistência** dessas associações.

Tipagem Estática (C, Java, Rust)

- Tipos verificados **em compilação**.
- Erros detectados antes da execução.
- Mais seguro e eficiente.
- **Nosso foco no Mini-Pascal.**

Tipagem Dinâmica (Python, JS)

- Tipos verificados **em execução**.
- Mais flexível, mas erros tardios.
- Geração de código para checagem em runtime.

Tipagem Forte vs Fraca

Tipagem Forte (Haskell, Rust):

Tipagem Forte vs Fraca

Tipagem Forte (Haskell, Rust):

- Conversões devem ser **explícitas**.
- O compilador recusa operações entre tipos incompatíveis.
- Mais seguro, menos surpresas.

Tipagem Forte vs Fraca

Tipagem Forte (Haskell, Rust):

- Conversões devem ser **explícitas**.
- O compilador recusa operações entre tipos incompatíveis.
- Mais seguro, menos surpresas.

Tipagem Fraca (C, JS):

- Conversões **implícitas** frequentes.
- `char + int` em C: converte char silenciosamente.
- `"5" + 3` em JS: resulta em `"53"`.

Tipagem Forte vs Fraca

Tipagem Forte (Haskell, Rust):

- Conversões devem ser **explícitas**.
- O compilador recusa operações entre tipos incompatíveis.
- Mais seguro, menos surpresas.

Tipagem Fraca (C, JS):

- Conversões **implícitas** frequentes.
- `char + int` em C: converte char silenciosamente.
- `"5" + 3` em JS: resulta em `"53"`.

Mini-Pascal

Tipagem **estática** e **moderadamente forte**: permite apenas promoção `integer` → `real`.

Tipos no Mini-Pascal

Nossa linguagem tem três tipos primitivos:

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

O Type Checker deve garantir que cada operador receba operandos de tipos **compatíveis** e que cada atribuição armazene um valor do tipo **esperado**.

Regras para Expressões Aritméticas

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

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

Promoção de Tipo (Coerção)

Quando um `integer` é operado com um `real`, o `integer` é promovido automaticamente a `real`. O resultado é `real`.

Regras para Expressões Relacionais

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

Operando Esq.	Operando Dir.	Resultado
`integer`	`integer`	`boolean`
`real`	`real`	`boolean`
`integer`	`real`	`boolean` (promoção)
`boolean`	`boolean`	`boolean` (apenas $=$, $<>$)

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

Regras para Comandos

- **Atribuição** (`x := expr`):
  - O tipo de `expr` deve ser **compatível** com o tipo de `x`.
  - `integer` → `real`: OK (promoção).
  - `real` → `integer`: **ERRO** (perda de precisão).

Regras para Comandos

- **Atribuição** (`x := expr`):
  - O tipo de `expr` deve ser **compatível** com o tipo de `x`.
  - `integer` → `real`: OK (promoção).
  - `real` → `integer`: **ERRO** (perda de precisão).

- **If / While** (condição):
  - A condição **deve** ter tipo `boolean`.
  - `if x > 0 then ...` ✓ (resultado de $>$ é boolean)
  - `if x + 1 then ...` **ERRO** (integer não é boolean)

Regras para Comandos

- **Atribuição** (`x := expr`):
  - O tipo de `expr` deve ser **compatível** com o tipo de `x`.
  - `integer` → `real`: OK (promoção).
  - `real` → `integer`: **ERRO** (perda de precisão).

- **If / While** (condição):
  - A condição **deve** ter tipo `boolean`.
  - `if x > 0 then ...` ✓ (resultado de $>$ é boolean)
  - `if x + 1 then ...` **ERRO** (integer não é boolean)

- **Write** (`write(expr)`):
  - Aceita qualquer tipo (imprime a representação textual).

Type Checker: Expressões Binárias

```
@Override
public Void visitBinaryExpr(BinaryExpr node) {
    visit(node.left);
    visit(node.right);
    String lt = node.left.resolvedType;
    String rt = node.right.resolvedType;

    if (isArithOp(node.operator)) {
        if (lt.equals("boolean")
                rt.equals("boolean"))
            error("Aritmetico com boolean", node);
        node.resolvedType = (lt.equals("real")
             rt.equals("real"))
            ? "real" : "integer";
    } else if (isRelOp(node.operator)) {
        node.resolvedType = "boolean";
    }
    return null;
}
```

Type Checker: Atribuição

```
@Override
public Void visitAssignStmt(AssignStmt node) {
    Symbol sym = symTable.lookup(node.varName);
    visit(node.value);
    String varType  = sym.type;
    String exprType = node.value.resolvedType;
    if (!isCompatible(varType, exprType)) {
        error("Tipo incompativel: '" + exprType
            + "' -> '" + varType + "'", node);
    }
    return null;
}

boolean isCompatible(String target, String source) {
    if (target.equals(source)) return true;
    // promocao integer -> real
    if (target.equals("real"))
        if (source.equals("integer"))
            return true;
```

Type Checker: Condições (If / While)

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

O mesmo padrão se aplica ao `WhileStmt`: verificar que a condição é `boolean` antes de processar o corpo.

Type Checker: Literais e Referências

```
@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
```

Coerção: Conversão Implícita de Tipos

Coerção (ou *implicit cast*) ocorre quando o compilador converte automaticamente um valor de um tipo para outro, sem que o programador peça explicitamente.

Promoção (widening):

- `integer` → `real`: OK
- Não perde informação
- Inserida automaticamente pelo compilador

Estreitamento (narrowing):

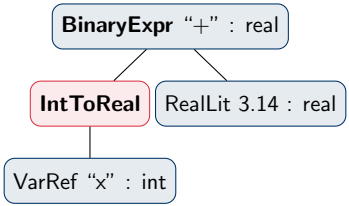
- `real` → `integer`: **Proibido**
- Perde a parte fracionária
- Requer cast explícito (em linguagens que permitem)

No Mini-Pascal

Permitimos apenas **promoção** (`integer` → `real`). Qualquer outra conversão é erro semântico.

Inserindo Nós de Coerção na AST

Quando o Type Checker detecta uma promoção necessária, ele pode **inserir um nó especial** na AST para que o gerador de código saiba que precisa converter:



O nó `IntToReal` instrui o gerador de código a emitir uma instrução de conversão (ex: `i2f` no bytecode JVM).

Notação de Inferência (Typing Judgments)

As regras de tipo são formalizadas com a notação:

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

- Γ = ambiente de tipos (tabela de símbolos)
- $\vdash$ = "prova que"
- $\Gamma \vdash e : T$ = "no ambiente Γ, a expressão e tem tipo T"

Exemplos de Axiomas (sem premissas)

$$\frac{}{\Gamma \vdash 42 : \texttt{int}}\ (\text{T-IntLit}) \qquad \frac{}{\Gamma \vdash 3.14 : \texttt{real}}\ (\text{T-RealLit}) \qquad \frac{}{\Gamma \vdash \texttt{true} : \texttt{bool}}\ (\text{T-BoolLit})$$

Como Ler uma Regra de Inferência

Considere a regra (T-Var):

$$\frac{(x : T) \in \Gamma}{\Gamma \vdash x : T}$$

Como Ler uma Regra de Inferência

Considere a regra (T-Var):

$$\frac{(x : T) \in \Gamma}{\Gamma \vdash x : T}$$

Leitura de baixo para cima: "Queremos provar que x tem tipo T."

Como Ler uma Regra de Inferência

Considere a regra (T-Var):

$$\frac{(x : T) \in \Gamma}{\Gamma \vdash x : T}$$

Leitura de baixo para cima: "Queremos provar que x tem tipo T."

1. **Abaixo da linha** (conclusão): o que queremos provar.
   $\Gamma \vdash x : T$ = "no ambiente Γ, x tem tipo T".

Como Ler uma Regra de Inferência

Considere a regra (T-Var):

$$\frac{(x : T) \in \Gamma}{\Gamma \vdash x : T}$$

Leitura de baixo para cima: "Queremos provar que x tem tipo T."

1. **Abaixo da linha** (conclusão): o que queremos provar.
   $\Gamma \vdash x : T$ = "no ambiente Γ, x tem tipo T".
2. **Acima da linha** (premissa): o que precisamos verificar.
   $(x : T) \in \Gamma$ = "x está declarada com tipo T na tabela".

Como Ler uma Regra de Inferência

Considere a regra (T-Var):

$$\frac{(x : T) \in \Gamma}{\Gamma \vdash x : T}$$

Leitura de baixo para cima: "Queremos provar que x tem tipo T."

1. **Abaixo da linha** (conclusão): o que queremos provar.
   $\Gamma \vdash x : T$ = "no ambiente Γ, x tem tipo T".
2. **Acima da linha** (premissa): o que precisamos verificar.
   $(x : T) \in \Gamma$ = "x está declarada com tipo T na tabela".
3. **A linha** = "se a premissa vale, então a conclusão vale".

Como Ler uma Regra de Inferência

Considere a regra (T-Var):

$$\frac{(x : T) \in \Gamma}{\Gamma \vdash x : T}$$

Leitura de baixo para cima: "Queremos provar que x tem tipo T."

1. **Abaixo da linha** (conclusão): o que queremos provar.
   $\Gamma \vdash x : T$ = "no ambiente Γ, x tem tipo T".
2. **Acima da linha** (premissa): o que precisamos verificar.
   $(x : T) \in \Gamma$ = "x está declarada com tipo T na tabela".
3. **A linha** = "se a premissa vale, então a conclusão vale".

Exemplo Concreto

Dado `var a : integer;` $\Rightarrow \Gamma = \{a : \texttt{integer}\}$.

Para tipar a: $(a : \texttt{integer}) \in \Gamma$? **Sim!** $\therefore\ \Gamma \vdash a : \texttt{integer}$

Regras Formais: Variáveis e Operações

Referência a variável:

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

Aritmética (integer $\times$ integer):

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

Atribuição com compatibilidade:

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

Onde $T_e \preceq T_x$ = compatível (mesmo tipo ou promoção int$\rightarrow$real).

Equivalência Nominal vs Estrutural

Equivalência Nominal:

Equivalência Nominal vs Estrutural

Equivalência Nominal:

- Tipos iguais $\iff$ **mesmo nome**.
- Mesmo layout $\neq$ mesmo tipo.
- Linguagens: **Java, C#, Rust, Pascal**.

Equivalência Nominal vs Estrutural

Equivalência Nominal:

- Tipos iguais $\iff$ **mesmo nome**.
- Mesmo layout $\neq$ mesmo tipo.
- Linguagens: **Java, C#, Rust, Pascal**.

Exemplo

```
type Metros = record x:  real end;
type Segundos = record x:  real end;
m := s; // ERRO!
```

Equivalência Nominal vs Estrutural

Equivalência Nominal:

- Tipos iguais $\iff$ **mesmo nome**.
- Mesmo layout $\neq$ mesmo tipo.
- Linguagens: **Java, C#, Rust, Pascal**.

Equivalência Estrutural:

- Tipos iguais $\iff$ **mesmo layout**.
- Nome é irrelevante.
- Linguagens: **TypeScript, OCaml, Go**.

Exemplo

```
type Metros = record x:  real end;
type Segundos = record x:  real end;
m := s; // ERRO!
```

Exemplo

```
type T1 = {x:  int, y:  real};
type T2 = {x:  int, y:  real};
T1 === T2 // OK!
```

Equivalência Nominal vs Estrutural

Equivalência Nominal:

- Tipos iguais $\iff$ **mesmo nome**.
- Mesmo layout $\neq$ mesmo tipo.
- Linguagens: **Java, C#, Rust, Pascal**.

Equivalência Estrutural:

- Tipos iguais $\iff$ **mesmo layout**.
- Nome é irrelevante.
- Linguagens: **TypeScript, OCaml, Go**.

Exemplo

```
type Metros = record x:  real end;
type Segundos = record x:  real end;
m := s; // ERRO!
```

Exemplo

```
type T1 = {x:  int, y:  real};
type T2 = {x:  int, y:  real};
T1 === T2 // OK!
```

Mini-Pascal

Apenas tipos primitivos — a distinção não se aplica. Mas é conceito fundamental para linguagens reais.

Operadores Unários e Lógicos

Negação aritmética (-):

$$\frac{\Gamma \vdash e : T \quad T \in \{\texttt{int}, \texttt{real}\}}{\Gamma \vdash -e : T}$$

- `-5` → integer **OK**
- `-3.14` → real **OK**
- `-true` → **ERRO**

Negação lógica (`not`):

$$\frac{\Gamma \vdash e : \texttt{boolean}}{\Gamma \vdash \texttt{not}\ e : \texttt{boolean}}$$

Operadores `and`/`or`:

$$\frac{\Gamma \vdash e_1 : \texttt{bool} \quad \Gamma \vdash e_2 : \texttt{bool}}{\Gamma \vdash e_1\ op\ e_2 : \texttt{boolean}}$$

Verificação de Tipos em Chamadas de Função

Ao encontrar `f(arg1, arg2, ...)`, o Type Checker verifica:

Verificação de Tipos em Chamadas de Função

Ao encontrar `f(arg1, arg2, ...)`, o Type Checker verifica:

1. **Existência:** `f` foi declarada como função?

Verificação de Tipos em Chamadas de Função

Ao encontrar `f(arg1, arg2, ...)`, o Type Checker verifica:

1. **Existência:** `f` foi declarada como função?
2. **Aridade:** Número de argumentos = número de parâmetros?

Verificação de Tipos em Chamadas de Função

Ao encontrar `f(arg1, arg2, ...)`, o Type Checker verifica:

1. **Existência:** `f` foi declarada como função?
2. **Aridade:** Número de argumentos = número de parâmetros?
3. **Tipos:** Cada argumento é compatível com o parâmetro correspondente?

Verificação de Tipos em Chamadas de Função

Ao encontrar `f(arg1, arg2, ...)`, o Type Checker verifica:

1. **Existência:** `f` foi declarada como função?
2. **Aridade:** Número de argumentos = número de parâmetros?
3. **Tipos:** Cada argumento é compatível com o parâmetro correspondente?
4. **Resultado:** O nó recebe o tipo de retorno da função.

Verificação de Tipos em Chamadas de Função

Ao encontrar `f(arg1, arg2, ...)`, o Type Checker verifica:

1. **Existência:** `f` foi declarada como função?
2. **Aridade:** Número de argumentos = número de parâmetros?
3. **Tipos:** Cada argumento é compatível com o parâmetro correspondente?
4. **Resultado:** O nó recebe o tipo de retorno da função.

Exemplo

```
function dobro(n: integer): integer;
```

`dobro(5)` **OK** `dobro(3.14)` **ERRO (narrowing)** `dobro(1,2)` **ERRO (aridade)**

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 nao declarada", node);
        node.resolvedType = "error";
        return null;
    }
    // Verificar aridade
    if (node.args.size()
            != funcSym.paramTypes.size())
        error("Num. argumentos incorreto", node);

    // Verificar tipo de cada argumento
    for (int i = 0; i < node.args.size(); i++) {
        visit(node.args.get(i));
        if (i < funcSym.paramTypes.size())
            if (!isCompatible(funcSym.paramTypes.get(i),
```

Poison Type: Evitando Erros em Cascata

Problema: erros em cascata

Poison Type: Evitando Erros em Cascata

Problema: erros em cascata

- Se `x + y` gera erro, o nó fica sem tipo.
- Toda expressão que usa o resultado **também** gera erro.
- Um único erro vira dezenas!

Poison Type: Evitando Erros em Cascata

Problema: erros em cascata

- Se `x + y` gera erro, o nó fica sem tipo.
- Toda expressão que usa o resultado **também** gera erro.
- Um único erro vira dezenas!

Solução: tipo “error”

- Nó com erro recebe tipo `"error"`.
- `"error"` é compatível com **tudo**.
- Erros subsequentes são suprimidos.

Poison Type: Evitando Erros em Cascata

Problema: erros em cascata

- Se `x + y` gera erro, o nó fica sem tipo.
- Toda expressão que usa o resultado **também** gera erro.
- Um único erro vira dezenas!

Solução: tipo "error"

- Nó com erro recebe tipo `"error"`.
- `"error"` é compatível com **tudo**.
- Erros subsequentes são suprimidos.

Implementação

```
boolean isCompatible(String t, String s) {
  if ("error".equals(t) || "error".equals(s))
    return true;
  // ...  regras normais ...
}
```

Type Inference: Deduzindo Tipos Automaticamente

Linguagens modernas (Kotlin, Rust, Haskell, `var` em Java 10+) permitem que o compilador **deduza** o tipo sem anotação explícita:

Exemplos

`var x = 42;` $\Rightarrow$ compilador infere `x : integer`

`var y = x + 3.14;` $\Rightarrow$ compilador infere `y : real`

`var z = x > 0;` $\Rightarrow$ compilador infere `z : boolean`

Algoritmo de Hindley-Milner: Propaga restrições de tipo pela AST usando **unificação**. Cada expressão gera uma equação de tipo; o algoritmo resolve o sistema de equações.

No Mini-Pascal

Não usamos inferência — todos os tipos são declarados explicitamente. Mas é importante saber que existe para linguagens mais avançadas.

Exercício 1: Verificação de Tipos Manual

Analise o seguinte programa e identifique **todos os erros semânticos**:

Código Mini-Pascal

```
var a :  integer;
var b :  real;
var c :  boolean;
begin
  a := 10;
  b := a + 3.5;
  c := a + b;
  if a + 1 then
    a := a + 1;
  b := c;
  a := b;
end.
```

Perguntas:

1. Qual o tipo de `a + 3.5`?
2. A linha `c := a + b` está correta?
3. A condição `a + 1` é válida para `if`?
4. A linha `b := c` está correta?
5. A linha `a := b` está correta?

Solução do Exercício 1

1. `a + 3.5`: `integer + real` $\Rightarrow$ promoção $\Rightarrow$ tipo `real`. **OK**
2. `c := a + b`: `a + b` é `real`. `c` é `boolean`. **ERRO: real $\rightarrow$ boolean**
3. `if a + 1`: `a + 1` é `integer`. **ERRO: condição deve ser boolean**
4. `b := c`: `c` é `boolean`. `b` é `real`. **ERRO: boolean $\rightarrow$ real**
5. `a := b`: `b` é `real`. `a` é `integer`. **ERRO: narrowing (real $\rightarrow$ integer)**

Total: 4 erros semânticos de tipo. Apenas as linhas 1, 2 e 5 estão corretas.

Exercício 2: Verificação com Funções

Código Mini-Pascal

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

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

var x :  integer;
var y :  real;
begin
  x := dobro(5);
  y := dobro(3.14);
  x := media(1, 2);
  y := media(1, 2);
  x := dobro(true);
  y := dobro(1, 2);
end.
```

Exercício 2: Verificação com Funções

Código Mini-Pascal

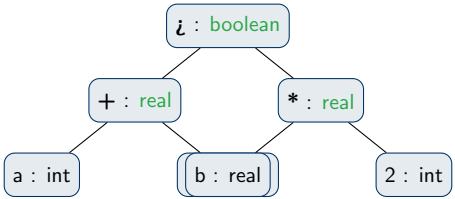
```
function dobro(n: integer): integer;
begin dobro := n * 2 end;

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

var x : integer;
var y : real;
begin
  x := dobro(5);
  y := dobro(3.14);
  x := media(1, 2);
  y := media(1, 2);
  x := dobro(true);
  y := dobro(1, 2);
end.
```

Análise:

1. `dobro(5)`: int→int. **OK**
2. `dobro(3.14)`: arg real, param int. **ERRO: narrowing**
3. `media(1,2)`: promoção OK, mas retorno real → int. **ERRO**
4. `media(1,2)`: promoção + real→real. **OK**
5. `dobro(true)`: boolean→int. **ERRO**
6. `dobro(1,2)`: 2 args, espera 1. **ERRO: aridade**

Exercício 3: Trace do Type Checker na AST

Trace a propagação de tipos para `a + 3.5 > b * 2` onde `a : integer`, `b : real`:



Ordem pós-ordem: folhas → pais → raiz. O Type Checker resolve os filhos **antes** do pai, garantindo que os tipos dos operandos estejam disponíveis.

Resumo da Aula

- O **Type Checker** garante que cada operação recebe operandos de tipos compatíveis.
- Tipagem: estática vs dinâmica, forte vs fraca. Mini-Pascal = **estática e moderadamente forte**.
- **Regras formais** ($\Gamma \vdash e : T$) especificam precisamente o comportamento do Type Checker.
- **Equivalência de tipos**: nominal (por nome) vs estrutural (por layout).
- **Promoção** (integer→real) é permitida; **estreitamento** (real→integer) é erro.
- O Type Checker verifica **funções**: aridade, tipos de argumentos e retorno.
- **Poison type** (`"error"`) evita erros em cascata.
- Nós de **coerção** (`IntToReal`) são inseridos na AST para guiar codegen.

Próxima Aula: Geração de Código Intermediário — representação de três endereços, tradução da AST anotada para instruções de máquina virtual.

Referências

- **AHO, A. V. et al.** *Compiladores: Princípios, Técnicas e Ferramentas*. 2ª Edição. Cap. 6 (Seções 6.3–6.5).
- **COOPER, K.; TORCZON, L.** *Engineering a Compiler*. 2ª Edição. Cap. 4 (Type Systems).
- **PIERCE, B. C.** *Types and Programming Languages*. MIT Press, 2002. Caps. 1–3.
- **NYSTROM, R.** *Crafting Interpreters*. Cap. 7 (Evaluating Expressions) e Cap. 12 (Classes).

Dúvidas?

Obrigado!

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