Compiladores

Aula 18: Geração de Código Intermediário (IR)

Prof. Aléssio Miranda Júnior
alessio@cefetmg.br

CEFET-MG - Campus Timoteo
Dep. Engenharia de Computação

Maio de 2026

Sumário

Objetivos da Aula

- Justificar a necessidade de uma **representação intermediária (IR)**.
- Descrever **código de três endereços (3AC)** e suas vantagens.
- Representar instruções 3AC como **quádruplas** ou **triplas**.
- Gerar 3AC para expressões, atribuições e fluxo de controle.
- Relacionar a IR com o pipeline do compilador e o projeto Mini-Pascal.

O Problema de Gerar Código Diretamente

Gerar assembly **diretamente** da AST é possível, mas:

O Problema de Gerar Código Diretamente

Gerar assembly **diretamente** da AST é possível, mas:

- Dificulta **otimizações** (precisam de forma linear e uniforme).

O Problema de Gerar Código Diretamente

Gerar assembly **diretamente** da AST é possível, mas:

- Dificulta **otimizações** (precisam de forma linear e uniforme).
- Acopla o front-end a uma **arquitetura específica**.

O Problema de Gerar Código Diretamente

Gerar assembly **diretamente** da AST é possível, mas:

- Dificulta **otimizações** (precisam de forma linear e uniforme).
- Acopla o front-end a uma **arquitetura específica**.
- N linguagens $\times$ M arquiteturas $= N \times M$ tradutores!

O Problema de Gerar Código Diretamente

Gerar assembly **diretamente** da AST é possível, mas:

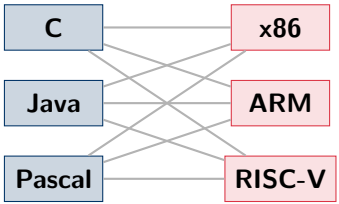
- Dificulta **otimizações** (precisam de forma linear e uniforme).
- Acopla o front-end a uma **arquitetura específica**.
- N linguagens $\times$ M arquiteturas $= N \times M$ tradutores!

Solução: IR como "Língua Franca"

N front-ends $\rightarrow$ **1 IR** $\rightarrow$ M back-ends = apenas $N + M$ tradutores.

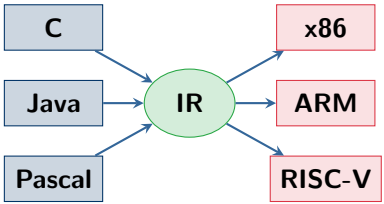
Visualização: $N \times M$ vs $N + M$

Sem IR: $N \times M$ tradutores



$3 \times 3 = \mathbf{9}$ tradutores

Com IR: $N + M$ tradutores

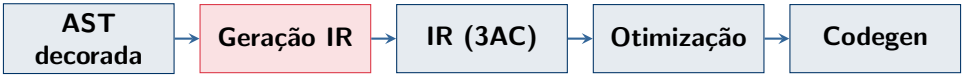


$3 + 3 = \mathbf{6}$ tradutores

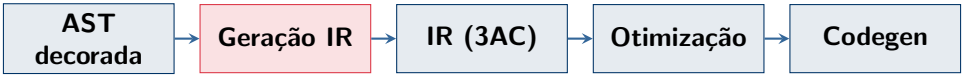
Economia

Com 10 linguagens e 5 arquiteturas: $10 \times 5 = 50$ vs $10 + 5 = 15$ tradutores!

A IR no Pipeline do Compilador

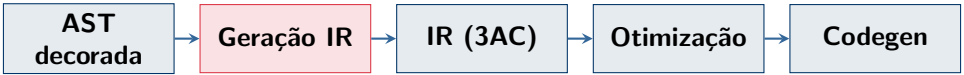


A IR no Pipeline do Compilador



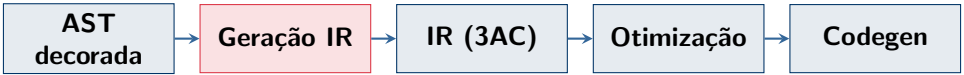
1. **Geração de IR:** Visitor percorre AST decorada e emite instruções 3AC.

A IR no Pipeline do Compilador



1. **Geração de IR:** Visitor percorre AST decorada e emite instruções 3AC.
2. **Otimização:** transformações sobre a IR (constant folding, dead code, etc.).

A IR no Pipeline do Compilador



1. **Geração de IR:** Visitor percorre AST decorada e emite instruções 3AC.
2. **Otimização:** transformações sobre a IR (constant folding, dead code, etc.).
3. **Codegen:** mapeia IR para assembly/bytecode da máquina alvo.

Níveis de Representação Intermediária

Nível	Exemplos	Características
Alto	AST, GIMPLE	Preserva loops, arrays, objetos
Médio	**3AC**, SSA, LLVM IR	Independente de linguagem e máquina
Baixo	MachineIR	Registradores virtuais, próximo de ISA

Níveis de Representação Intermediária

Nível	Exemplos	Características
Alto	AST, GIMPLE	Preserva loops, arrays, objetos
Médio	**3AC**, SSA, LLVM IR	Independente de linguagem e máquina
Baixo	MachineIR	Registradores virtuais, próximo de ISA

Nosso foco: Código de Três Endereços (3AC)

Cada instrução tem no máximo **um operador** e (em geral) **um destino explícito**. É o representante clássico do nível médio.

O Que é Código de Três Endereços?

Cada instrução tem a forma:

$$\texttt{destino} = \texttt{arg1}\ \textit{op}\ \texttt{arg2}$$

"Três endereços" = dois operandos + um resultado.

O Que é Código de Três Endereços?

Cada instrução tem a forma:

$$\texttt{destino} = \texttt{arg1}\ \textit{op}\ \texttt{arg2}$$

"Três endereços" = dois operandos + um resultado.

Tipos de instrução 3AC:

- **Binária:** `t1 = a + b`

O Que é Código de Três Endereços?

Cada instrução tem a forma:

$$\texttt{destino} = \texttt{arg1}\ \textit{op}\ \texttt{arg2}$$

"Três endereços" = dois operandos + um resultado.

Tipos de instrução 3AC:

- **Binária:** `t1 = a + b`
- **Unária:** `t2 = -a` (negação, conversão)

O Que é Código de Três Endereços?

Cada instrução tem a forma:

$$\texttt{destino} = \texttt{arg1}\ \textit{op}\ \texttt{arg2}$$

"Três endereços" = dois operandos + um resultado.

Tipos de instrução 3AC:

- **Binária:** `t1 = a + b`
- **Unária:** `t2 = -a` (negação, conversão)
- **Cópia:** `x = t1`

O Que é Código de Três Endereços?

Cada instrução tem a forma:

$$\texttt{destino} = \texttt{arg1}\ \textit{op}\ \texttt{arg2}$$

"Três endereços" = dois operandos + um resultado.

Tipos de instrução 3AC:

- **Binária:** `t1 = a + b`
- **Unária:** `t2 = -a` (negação, conversão)
- **Cópia:** `x = t1`
- **Salto:** `goto L1`, `if a < b goto L1`

O Que é Código de Três Endereços?

Cada instrução tem a forma:

$$\texttt{destino} = \texttt{arg1}\ \textit{op}\ \texttt{arg2}$$

"Três endereços" = dois operandos + um resultado.

Tipos de instrução 3AC:

- **Binária:** `t1 = a + b`
- **Unária:** `t2 = -a` (negação, conversão)
- **Cópia:** `x = t1`
- **Salto:** `goto L1`, `if a < b goto L1`
- **Chamada:** `param p1; call f, 2; t3 = return`

Exemplo: Expressão Complexa em 3AC

Código fonte

```
a := b + c * d - e;
```

Expressões complexas são "quebradas" em operações **elementares**.

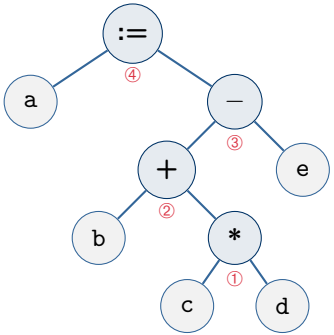
```
t1 = c * d
t2 = b + t1
t3 = t2 - e
a  = t3
```

Vantagens:

- Cada instrução $\approx$ 1 instrução de máquina
- Facilita otimizações e alocação de registradores

Da Árvore ao Código Linear: Travessia Pós-Ordem

AST de `a := b + c * d - e`



Instruções emitidas (pós-ordem):

① `t1 = c * d`

② `t2 = b + t1`

③ `t3 = t2 - e`

④ `a = t3`

Padrão

Cada nó interno da AST gera **uma instrução 3AC**. Folhas (variáveis/constantes) apenas retornam seus nomes.

Exemplo: Fluxo de Controle em 3AC

Código fonte

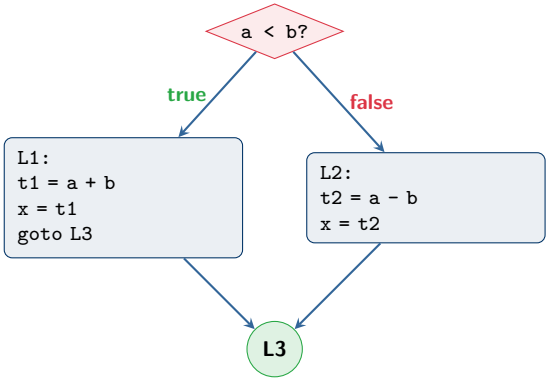
```
if (a < b) then
  x := a + b
else
  x := a - b;
```

```
    if a < b goto L1
    goto L2
L1:
    t1 = a + b
    x  = t1
    goto L3
L2:
    t2 = a - b
    x  = t2
L3:
    ...
```

Observação

O fluxo de controle fica **explícito**: condicionais viram saltos, labels marcam os alvos.

Grafo de Fluxo de Controle (CFG) do If/Else



Blocos Básicos:

- **Teste:** avalia condição
- **L1, L2:** caminhos then/else
- **L3:** ponto de junção

Definição

Um **bloco básico** é uma sequência maximal de instruções sem saltos internos nem labels intermediários.

Exemplo: Laço While em 3AC

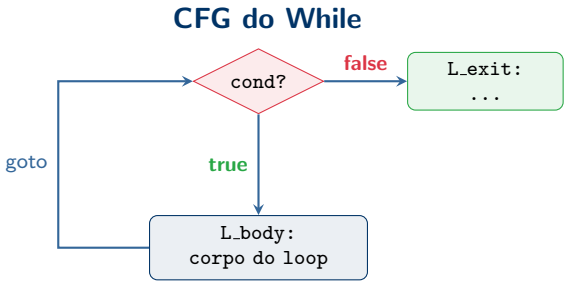
Código fonte

```
while (i < n) do
begin
  s := s + i;
  i := i + 1;
end;
```

```
L1:
    if i < n goto L2
    goto L3
L2:
    t1 = s + i
    s  = t1
    t2 = i + 1
    i  = t2
    goto L1
L3:
    ...
```

Padrão: `L1` = teste, `L2` = corpo, `L3` = saída. O `goto L1` fecha o loop.

Padrão de Geração: Estrutura do While



Esqueleto 3AC:

```
L_test:  // rótulo do teste
         avaliar cond → t
         if t goto L_body
         goto L_exit
L_body:  // corpo do loop
         instruções...
         goto L_test
L_exit:  // saída
         ...
```

Regra

Todo `while` segue este padrão: **3 labels**, teste no topo, `goto` fecha o ciclo.

Representação Interna: Quádruplas

Cada instrução 3AC é armazenada como uma tupla de 4 campos:

op	arg1	arg2	result
*	c	d	t1
+	b	t1	t2
-	t2	e	t3
=	t3	—	a

Representação Interna: Quádruplas

Cada instrução 3AC é armazenada como uma tupla de 4 campos:

op	arg1	arg2	result
*	c	d	t1
+	b	t1	t2
-	t2	e	t3
=	t3	—	a

Vantagem: resultado nomeado explicitamente $\Rightarrow$ **reordenar** instruções não quebra referências.

Representação Interna: Quádruplas

Cada instrução 3AC é armazenada como uma tupla de 4 campos:

op	arg1	arg2	result
*	c	d	t1
+	b	t1	t2
-	t2	e	t3
=	t3	—	a

Vantagem: resultado nomeado explicitamente $\Rightarrow$ **reordenar** instruções não quebra referências.

Desvantagem: ocupa mais espaço (4 campos por instrução).

Representação Interna: Triplas

Cada instrução tem apenas 3 campos; o resultado é o **índice** da instrução:

#	op	arg1	arg2
(0)	*	c	d
(1)	+	b	(0)
(2)	-	(1)	e
(3)	=	a	(2)

Representação Interna: Triplas

Cada instrução tem apenas 3 campos; o resultado é o **índice** da instrução:

#	op	arg1	arg2
(0)	*	c	d
(1)	+	b	(0)
(2)	-	(1)	e
(3)	=	a	(2)

Vantagem: mais compacto (3 campos).

Representação Interna: Triplas

Cada instrução tem apenas 3 campos; o resultado é o **índice** da instrução:

#	op	arg1	arg2
(0)	*	c	d
(1)	+	b	(0)
(2)	-	(1)	e
(3)	=	a	(2)

Vantagem: mais compacto (3 campos).
Desvantagem: reordenar instruções **altera os índices** $\Rightarrow$ referências precisam ser atualizadas.

Quádruplas vs Triplas: Quando Usar Cada?

Quádruplas (recomendado):

- Quando há muitas **otimizações** que reordenam código.
- Mais fácil de implementar transformações.
- Usado na maioria dos compiladores didáticos.

Triplas:

- Quando memória é crítica e o código não será muito reordenado.
- Usado em alguns compiladores embarcados.

No Mini-Pascal

Usaremos **quádruplas** (ou lista de objetos `Instruction`) por simplicidade e clareza.

Comparação Visual: Quádruplas vs Triplas

Para `a = b + c * d`:

Quádruplas

op	arg1	arg2	result
*	c	d	t1
+	b	t1	t2
=	t2	—	a

Referências por **nome**
$\Rightarrow$ reordenar = **seguro**

Triplas

#	op	arg1	arg2
(0)	*	c	d
(1)	+	b	(0)
(2)	=	a	(1)

Referências por **índice**
$\Rightarrow$ reordenar = **quebra!**

Conclusão

Quádruplas usam mais memória, mas são **robustas** para otimizações. Triplas são compactas mas **frágeis** à reordenação.

Geração de IR: Estratégia

O gerador de IR é um **Visitor** sobre a AST decorada:

Geração de IR: Estratégia

O gerador de IR é um **Visitor** sobre a AST decorada:

1. **Expressões:** gerar instruções que computam o valor em um **temporário** (`t1`, `t2`, ...).

Geração de IR: Estratégia

O gerador de IR é um **Visitor** sobre a AST decorada:

1. **Expressões:** gerar instruções que computam o valor em um **temporário** (`t1`, `t2`, ...).
2. **Atribuições:** gerar instrução `x = tN`.

Geração de IR: Estratégia

O gerador de IR é um **Visitor** sobre a AST decorada:

1. **Expressões:** gerar instruções que computam o valor em um **temporário** (`t1`, `t2`, ...).
2. **Atribuições:** gerar instrução `x = tN`.
3. **If/While:** gerar **labels** e **saltos** condicionais.

Geração de IR: Estratégia

O gerador de IR é um **Visitor** sobre a AST decorada:

1. **Expressões:** gerar instruções que computam o valor em um **temporário** (`t1`, `t2`, ...).
2. **Atribuições:** gerar instrução `x = tN`.
3. **If/While:** gerar **labels** e **saltos** condicionais.
4. **Funções:** gerar instruções `param`, `call`, `return`.

Geração de IR: Estratégia

O gerador de IR é um **Visitor** sobre a AST decorada:

1. **Expressões:** gerar instruções que computam o valor em um **temporário** (`t1`, `t2`, ...).
2. **Atribuições:** gerar instrução `x = tN`.
3. **If/While:** gerar **labels** e **saltos** condicionais.
4. **Funções:** gerar instruções `param`, `call`, `return`.

Recursos necessários:

- Contador de temporários: `newTemp()` $\rightarrow$ `t1`, `t2`, ...
- Contador de labels: `newLabel()` $\rightarrow$ `L1`, `L2`, ...
- Lista de instruções: `emit(op, arg1, arg2, result)`

Visitor IR: Expressões Binárias

```
Overridepublic String visitBinaryExpr(BinaryExpr node) String left = visit(node.left); //
    gera IRString right = visit(node.right); // gera IRString temp = newTemp(); // t1,
    t2...emit(node.operator, left, right, temp);return temp; // retorna o nome do
    temporarioOverride
public String visitNumberLiteral(NumberLiteral n) {
    return String.valueOf(n.value); // constante
}

Overridepublic String visitVarRef(VarRef node) return node.name; // nome da variavel
```

Visitor IR: Atribuição e If

```
Overridepublic String visitAssignStmt(AssignStmt node) String val =
    visit(node.value);emit("=", val, null, node.varName);return null;Override
public String visitIfStmt(IfStmt node) {
    String cond = visit(node.condition);
    String labelTrue  = newLabel(); // L1
    String labelEnd   = newLabel(); // L2
    emit("ifTrue", cond, null, labelTrue);
    if (node.elseBranch != null)
        visit(node.elseBranch);
    emit("goto", null, null, labelEnd);
    emitLabel(labelTrue);
    visit(node.thenBranch);
    emitLabel(labelEnd);
    return null;
}
```

Exercício: Tradução Manual para 3AC

Traduza o seguinte código Mini-Pascal para 3AC:

Código

```
var x, y, z :  integer;
begin
  x := 2 + 3 * 4;
  if x > 10 then
    y := x - 1
  else
    y := x + 1;
  z := y * 2;
end.
```

Perguntas:

1. Quantos temporários são necessários?
2. Quantos labels são necessários?
3. Escreva a sequência completa de instruções 3AC.
4. Represente as instruções como quádruplas.

Solução do Exercício

```
    t1 = 3 * 4
    t2 = 2 + t1
    x  = t2
    t3 = x > 10          // resultado boolean
    if t3 goto L1
    goto L2
L1: t4 = x - 1
    y  = t4
    goto L3
L2: t5 = x + 1
    y  = t5
L3: t6 = y * 2
    z  = t6
```

Resposta: 6 temporários (t1–t6), 3 labels (L1–L3), 13 instruções.

Na Prática: LLVM IR e Bytecode JVM

Para `x = a + b * c`:

LLVM IR (registradores)

```
%t1 = mul i32 %b, %c
%t2 = add i32 %a, %t1
store i32 %t2, i32* %x
```

- Forma SSA (cada %t definido 1x)
- Tipos explícitos (i32)
- **Mesmo princípio do 3AC!**

Bytecode JVM (pilha)

```
iload_1     // empilha b
iload_2     // empilha c
imul        // b*c -> topo
iload_0     // empilha a
iadd        // a+(b*c) -> topo
istore_3    // desempilha em x
```

- Sem temporários explícitos
- Pilha de operandos
- **Alvo do Mini-Pascal!**

Insight

3AC $\approx$ máquina de registradores vs JVM $\approx$ máquina de pilha. Ambas são formas de IR

Para Pensar...

1. Por que uma IR em nível "médio" é preferível a gerar assembly diretamente da AST para um compilador que pretende suportar **várias arquiteturas**?

Para Pensar...

1. Por que uma IR em nível "médio" é preferível a gerar assembly diretamente da AST para um compilador que pretende suportar **várias arquiteturas**?
2. Em que situações **triplas** podem ser preferíveis a quádruplas?

Para Pensar...

1. Por que uma IR em nível "médio" é preferível a gerar assembly diretamente da AST para um compilador que pretende suportar **várias arquiteturas**?
2. Em que situações **triplas** podem ser preferíveis a quádruplas?
3. O bytecode JVM (pilha) pode ser considerado uma forma de IR? Quais as diferenças em relação ao 3AC (registradores)?

Para Pensar...

1. Por que uma IR em nível "médio" é preferível a gerar assembly diretamente da AST para um compilador que pretende suportar **várias arquiteturas**?
2. Em que situações **triplas** podem ser preferíveis a quádruplas?
3. O bytecode JVM (pilha) pode ser considerado uma forma de IR? Quais as diferenças em relação ao 3AC (registradores)?
4. Como você geraria 3AC para uma chamada de função com **3 parâmetros**?

Resumo da Aula

- A **IR** desacopla front-end e back-end ($N + M$ em vez de $N \times M$).
- **3AC:** cada instrução com no máximo 1 operador e 1 destino.
- **Quádruplas** (op, arg1, arg2, result) facilitam otimizações; **triplas** são mais compactas.
- O gerador de IR é um **Visitor** com `newTemp()`, `newLabel()` e `emit()`.
- Fluxo de controle: condicionais e loops viram **labels** + **gotos**.

Conexão com o Projeto Mini-Pascal

Etapa 7: bytecode JVM como IR de pilha. Os mesmos princípios: decomposição de expressões, temporários, fluxo explícito.

Próxima Aula: Representação SSA — forma estática de atribuição única, ϕ-funções, e otimizações baseadas em SSA.

Referências

- **AHO, A. V. et al.** *Compiladores: Princípios, Técnicas e Ferramentas*. 2ª Edição. Cap. 6 (Seções 6.2–6.4: código de três endereços).
- **COOPER, K.; TORCZON, L.** *Engineering a Compiler*. 2ª Edição. Cap. 5 (IR Design).
- **APPEL, A. W.** *Modern Compiler Implementation in Java*. Cap. 7 (Translation to IR).
- **NYSTROM, R.** *Crafting Interpreters*. Cap. 15 (A Virtual Machine) e Cap. 17 (Compiling Expressions).

Dúvidas?

Obrigado!

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