Compiladores

Aula 18: Geração de Código Intermediário (IR)

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

Maio de 2026

1 Objetivos

⋆ title=Objetivos da Aula

Ao final deste capítulo, o estudante deverá ser capaz de:

- Justificar a necessidade de uma **representação intermediária (IR)** no design de compiladores modernos.
- Descrever o **Código de Três Endereços (3AC)** e enumerar suas vantagens para análise e otimização.
- Compreender e gerar 3AC para expressões aritméticas, atribuições e estruturas de controle de fluxo (condicionais e laços).
- Comparar as estruturas de dados usadas para representar 3AC na memória, como **quádruplas** e **triplas**.
- Relacionar a geração de IR com o padrão Visitor, construindo a base para a geração de código em projetos práticos.

2 Motivação: Por Que Usar IR?

Em teoria, um compilador poderia traduzir a Árvore Sintática Abstrata (AST) diretamente para o código de máquina (Assembly) do processador alvo. Compiladores mais antigos e simples operavam dessa forma. No entanto, a construção de compiladores modernos adota uma **Representação Intermediária (IR)**.

2.1 O Problema de $N \times M$ Tradutores

Suponha que uma empresa deseje construir compiladores para N linguagens de programação diferentes (C, C++, Java, Rust, etc.) e dar suporte a M arquiteturas de processador distintas (x86, ARM, RISC-V, etc.).

Se o código for gerado diretamente da AST (específica de cada linguagem) para o código de máquina (específico da arquitetura), será necessário desenvolver $N \times M$ tradutores completos. Isso é altamente ineficiente em termos de esforço de desenvolvimento e manutenção.

A IR como Língua Franca ($N + M$)

A solução arquitetural é introduzir uma representação comum entre o front-end (análise de código-fonte) e o back-end (geração de código de máquina). Assim, desenvolve-se apenas:

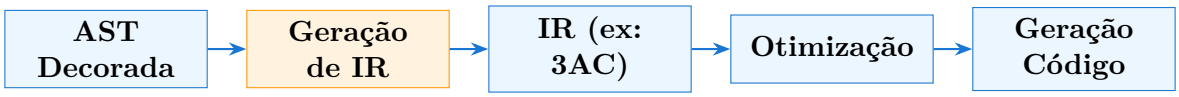
- N front-ends que traduzem das linguagens para a IR comum.
- M back-ends que traduzem da IR comum para as arquiteturas alvo.

O total de módulos cai para $N + M$, viabilizando projetos como o LLVM e o GCC.

Além da questão arquitetural, a IR possui propriedades cruciais para o **otimizador**: ela lineariza o programa, facilitando algoritmos de fluxo de dados que seriam muito complexos se executados diretamente sobre a estrutura em árvore da AST.

2.2 A IR no Pipeline do Compilador

No pipeline de um compilador clássico, a geração de IR atua como uma ponte entre as fases analíticas e sintéticas:



3 Níveis de Representação Intermediária

A IR pode variar muito dependendo dos objetivos do compilador:

- **Alto Nível:** Mantém informações estruturais ricas (como laços de repetição 'for'/'while', arrays n-dimensionais e classes). Útil para otimizações de escopo de alto nível (ex.: *loop unrolling*, vetorização). Exemplo: GIMPLE no GCC.
- **Médio Nível:** Independente tanto da linguagem de origem quanto da máquina de destino. As expressões são planificadas e o controle de fluxo se dá exclusivamente por saltos ('goto'). O **Código de Três Endereços (3AC)** e a LLVM IR se enquadram aqui.

- **Baixo Nível:** Muito próximo da arquitetura alvo, incorporando convenções de chamada específicas e detalhes do processador, mas ainda retendo registradores virtuais ilimitados. Exemplo: MachineIR no LLVM.

O restante deste capítulo foca no **Código de Três Endereços (3AC)**, o representante clássico do nível médio, ideal para análises e otimizações escalares.

4 Código de Três Endereços (3AC)

O que é 3AC?

No Código de Três Endereços, cada instrução possui **no máximo um operador** do lado direito da atribuição. Expressões complexas do código-fonte são decompostas numa sequência de operações elementares, cujos resultados parciais são armazenados em variáveis geradas pelo compilador, chamadas **temporários** ('t1', 't2', etc.).

A forma geral de uma instrução 3AC é:

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

Por possuir até dois operandos de leitura e um operando de escrita, o formato recebe o nome "três endereços".

4.1 Tipos de Instrução 3AC

O conjunto de instruções 3AC geralmente suporta as seguintes categorias:

- **Binárias:** `t1 = a + b`
- **Unárias:** `t2 = -a` (negação aritmética, ou cast/conversão de tipos).
- **Cópia (Atribuição):** `x = t1`
- **Salto incondicional:** `goto L1`
- **Salto condicional:** `if a < b goto L1`
- **Chamada de função:**

```
param p1
param p2
t3 = call f, 2     // 2 representa o número de parâmetros
```

4.2 Geração para Expressões

A quebra de uma expressão se dá através da travessia em pós-ordem da AST. Para a expressão `a := b + c * d - e`, a AST processa a precedência garantindo que a multiplicação venha antes da soma:

Decomposição de Expressão

Código-fonte:

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

Instruções 3AC geradas:

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

Cada instrução gerada em 3AC equivale quase de um para um com instruções de máquina comuns em arquiteturas RISC, o que facilita enormemente a fase final do compilador.

4.3 Geração para Fluxo de Controle

No 3AC, não existem blocos estruturados como `if-else` ou laços de repetição `while`. O fluxo é linearizado usando rótulos (**labels**) e saltos condicionais e incondicionais (**goto**).

Geração para if-then-else

Código-fonte:

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

Código 3AC:

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

Note que o fim do bloco 'then' necessita de um salto incondicional (`goto L3`) para impedir que o fluxo transborde e execute o código do bloco 'else'. O mesmo princípio se aplica para laços `while`, onde três rótulos coordenam o fluxo de repetição.

5 Representação em Memória: Quádruplas e Triplas

Enquanto a representação textual do 3AC (como 't1 = a + b') é útil para humanos debugarem o compilador, internamente a IR é uma estrutura de dados. Há duas abordagens clássicas para representar as instruções: Quádruplas e Triplas.

5.1 Quádruplas

Uma quádrupla representa a instrução através de uma tupla com quatro campos (ou atributos de uma classe): `(op, arg1, arg2, result)`. Onde `result` é o nome do temporário que armazenará o valor da operação.

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

Vantagem: Como os valores gerados têm nomes explícitos (`t1`, `t2`), o compilador pode alterar a ordem de duas instruções independentes de maneira completamente segura durante as fases de otimização, sem quebrar os links de dependência. É o modelo mais comum.

5.2 Triplas

Na representação por triplas, o campo `result` não existe. Em vez disso, se uma instrução posterior precisar do resultado de uma instrução anterior, ela faz referência diretamente ao **índice** posicional daquela instrução geradora.

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

Vantagem / Desvantagem: Triplas ocupam menos memória, mas são frágeis. Se uma etapa de otimização quiser mover ou remover uma instrução do meio do array, todos os índices mudam e todas as referências cruzadas precisam ser consertadas com alto custo de processamento.

6 Implementação: Geração de IR como Visitor

Na prática, a geração do código intermediário se dá caminhando na Árvore Sintática Abstrata (já devidamente verificada pela Análise Semântica). O Padrão de Projeto **Visitor** é a solução natural.

Para isso, a classe geradora manterá estado para produzir rótulos e temporários em sequência (`newTemp()` que gera `t1`, `t2`, ...; e `newLabel()` que gera `L1`, `L2`, ...). Uma chamada à função `emit(...)` salva uma instrução (por exemplo, na forma de Quádrupla) em uma lista.

Implementação em Java (Visitor de Expressões)

```java
@Override
public String visitBinaryExpr(BinaryExpr node) {
    // Visita filhos em pós-ordem
    String left  = visit(node.left);
    String right = visit(node.right);

    // Aloca um novo nome de temporário
    String temp  = newTemp();

    // Emite a instrução quádrupla para a lista do
       compilador
    emit(node.operator, left, right, temp);

    // Devolve para quem chamou o nome do temporário onde o
       resultado estará
    return temp;
}
```

Quando processamos comandos condicionados, o *Visitor* solicitará a criação dos rótulos necessários antes de visitar os sub-blocos, organizando os saltos para criar o fluxo de controle, mapeando o controle aninhado para controle plano.

7 Exercício Guiado

Exercício: Tradução Manual para 3AC

Traduza o seguinte fragmento de código Mini-Pascal para a representação de 3 endereços. Defina também a quantidade de temporários e labels necessários.

```
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.
```

Solução

```
1       t1 = 3 * 4
2       t2 = 2 + t1
3       x  = t2
4       t3 = x > 10
5       if t3 goto L1
6       goto L2
7   L1: t4 = x - 1
8       y  = t4
9       goto L3
10  L2: t5 = x + 1
11      y  = t5
12  L3: t6 = y * 2
13      z  = t6
```

Análise da Solução:

- **Temporários:** 6 variáveis locais extras foram exigidas (`t1` a `t6`).
- **Labels:** 3 rótulos estruturais controlam a junção (`L1`, `L2`, `L3`).

8 IR na Prática: LLVM vs JVM

Duas das maiores infraestruturas de compilação da atualidade refletem abordagens diferentes para a IR:

8.1 A LLVM IR (Máquina de Registradores Infinita)

A IR do ecossistema LLVM (base do clang para C/C++, 'rustc' para Rust, etc.) comporta-se exatamente como o Código de Três Endereços, suportando um número ilimitado de "temporários" locais que no LLVM são virtualizados com prefixo % (ex.: `%t1`, `%t2`). Ela também é nativamente formatada em SSA (*Static Single Assignment*).

8.2 Bytecode da Java Virtual Machine (Máquina de Pilha)

A IR do ecossistema Java (e do Kotlin, Scala, etc.) abstrai a CPU como uma máquina operada por uma **pilha de operandos**. Operações não têm destinos nomeados: elas simplesmente desempilham valores do topo da pilha, executam a matemática, e empilham o resultado de volta. Essa é uma IR muito eficiente para ser armazenada (bytecode super denso) e será nosso foco na construção final do projeto prático Mini-Pascal.

9 Resumo

✓ title=Pontos-Chave

- A inserção da **Representação Intermediária (IR)** desvincula front-end e back-end, reduzindo a complexidade de desenvolvimento a sistemas do tipo $N + M$ tradutores.
- **Código de Três Endereços (3AC):** Consiste na decomposição de computações de linguagem de alto nível em operações primitivas (operação binária, unária, salto).
- A construção em **Quádruplas** suporta otimizações seguras graças à nomeação explícita do `result`.
- A geração ocorre de modo prático mediante uma travessia baseada no padrão **Visitor** pela Árvore Sintática Abstrata (AST), criando temporários conforme cálculos e alocando rótulos ao processar o fluxo.

10 Referências

- **AHO, A. V.; LAM, M. S.; SETHI, R.; ULLMAN, J. D.** *Compiladores: Princípios, Técnicas e Ferramentas*. 2ª ed. Pearson, 2008. Cap. 6 (Geração de Código Intermediário).
- **COOPER, K.; TORCZON, L.** *Engineering a Compiler*. 2ª ed. Morgan Kaufmann, 2011. Cap. 5 (Intermediate Representations).
- **APPEL, A. W.** *Modern Compiler Implementation in Java*. 2ª ed. Cambridge University Press, 2002. Cap. 7 (Translation to Intermediate Code).
- **NYSTROM, R.** *Crafting Interpreters*. Genever Benning, 2021. Cap. 15 (A Virtual Machine).