Compiladores

Aula 20: Introdução ao LLVM IR

Prof. Aléssio Miranda Júnior
alessio@cefetmg.br

CEFET-MG - Campus Timoteo
Dep. Engenharia de Computação

Maio de 2026

Sumário

Objetivos da Aula

- Conhecer a infraestrutura **LLVM** e seu papel na compilação moderna.
- Compreender a anatomia da **LLVM IR**: tipos, registradores, instruções.
- Ler e interpretar código LLVM IR (`.ll`).
- Utilizar as ferramentas `clang`, `opt` e `llc` do pipeline LLVM.
- Relacionar LLVM IR com os conceitos de 3AC e SSA vistos anteriormente.

O Ecossistema LLVM

LLVM (Low Level Virtual Machine) é a plataforma de referência para compiladores modernos:

Quem usa LLVM?

O Ecossistema LLVM

LLVM (Low Level Virtual Machine) é a plataforma de referência para compiladores modernos:

Quem usa LLVM?

- **Clang** (C/C++/Objective-C)
- **Rust** (rustc $\rightarrow$ LLVM)
- **Swift** (Apple)
- **Julia**, **Zig**, **Kotlin/Native**
- GPUs: **CUDA**, **ROCm**

O Ecossistema LLVM

LLVM (Low Level Virtual Machine) é a plataforma de referência para compiladores modernos:

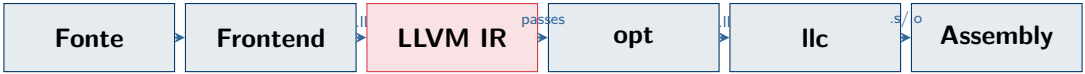
Quem usa LLVM?

- **Clang** (C/C++/Objective-C)
- **Rust** (rustc $\rightarrow$ LLVM)
- **Swift** (Apple)
- **Julia**, **Zig**, **Kotlin/Native**
- GPUs: **CUDA**, **ROCm**

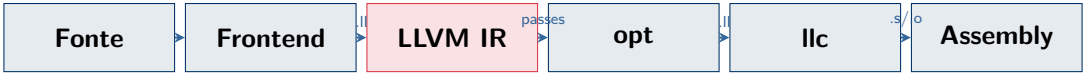
Por que é tão popular?

- Arquitetura **modular** (bibliotecas).
- IR bem definida, **SSA nativa**.
- Back-ends para x86, ARM, RISC-V, WebAssembly...
- Otimizações de ponta, atualizadas constantemente.

O Pipeline LLVM

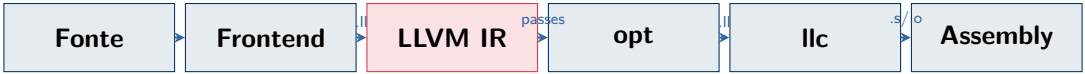


O Pipeline LLVM



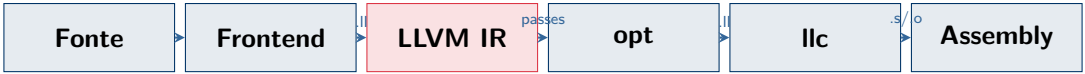
1. **Frontend** (Clang, rustc): código fonte → LLVM IR (`.ll` ou `.bc`).

O Pipeline LLVM



1. **Frontend** (Clang, rustc): código fonte → LLVM IR (`.ll` ou `.bc`).
2. **opt**: otimizações sobre a IR (passes como `-mem2reg`, `-instcombine`).

O Pipeline LLVM



1. **Frontend** (Clang, rustc): código fonte → LLVM IR (`.ll` ou `.bc`).
2. **opt**: otimizações sobre a IR (passes como `-mem2reg`, `-instcombine`).
3. **llc**: LLVM IR → assembly nativo (`.s`) ou objeto (`.o`).

Características Fundamentais

LLVM IR é um **assembly portável, tipado e em SSA**:

- **SSA nativa:** cada registrador virtual é atribuído uma vez.

Características Fundamentais

LLVM IR é um **assembly portável, tipado e em SSA**:

- **SSA nativa:** cada registrador virtual é atribuído uma vez.
- **Tipada:** toda instrução especifica tipos dos operandos.

Características Fundamentais

LLVM IR é um **assembly portável, tipado e em SSA**:

- **SSA nativa:** cada registrador virtual é atribuído uma vez.
- **Tipada:** toda instrução especifica tipos dos operandos.
- **Registradores virtuais infinitos:** `%0`, `%1`, `%result`, ...

Características Fundamentais

LLVM IR é um **assembly portável, tipado e em SSA**:

- **SSA nativa:** cada registrador virtual é atribuído uma vez.
- **Tipada:** toda instrução especifica tipos dos operandos.
- **Registradores virtuais infinitos:** `%0`, `%1`, `%result`, ...
- **Três formas:** texto (`.ll`), bitcode (`.bc`), em memória (API C++).

Características Fundamentais

LLVM IR é um **assembly portável, tipado e em SSA**:

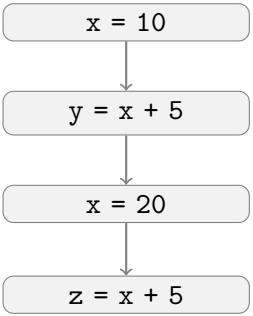
- **SSA nativa:** cada registrador virtual é atribuído uma vez.
- **Tipada:** toda instrução especifica tipos dos operandos.
- **Registradores virtuais infinitos:** `%0`, `%1`, `%result`, ...
- **Três formas:** texto (`.ll`), bitcode (`.bc`), em memória (API C++).

Relação com 3AC e SSA

LLVM IR é essencialmente **3AC em SSA**: instruções simples, um resultado por instrução, ϕ para junções.

Entendendo a SSA (Static Single Assignment)

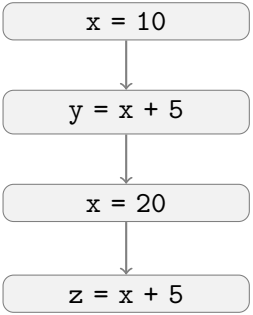
Sem SSA (Código Normal)



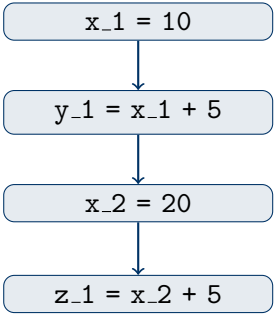
Qual `x` é usado em `z = x + 5`? O compilador precisa rastrear todo o caminho para trás.

Entendendo a SSA (Static Single Assignment)

Sem SSA (Código Normal)



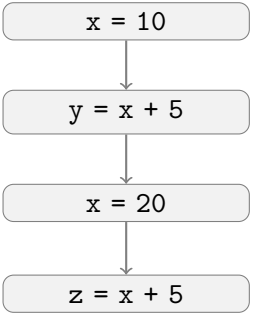
Com SSA (Atribuição Única)



Qual `x` é usado em `z = x + 5`? O compilador precisa rastrear todo o caminho para trás.

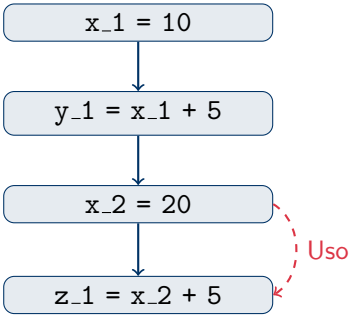
Entendendo a SSA (Static Single Assignment)

Sem SSA (Código Normal)



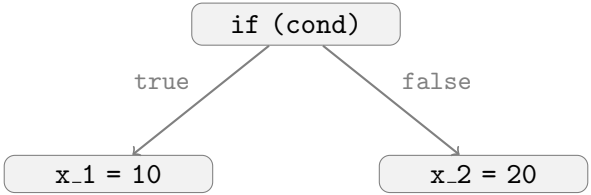
Qual `x` é usado em `z = x + 5`? O compilador precisa rastrear todo o caminho para trás.

Com SSA (Atribuição Única)

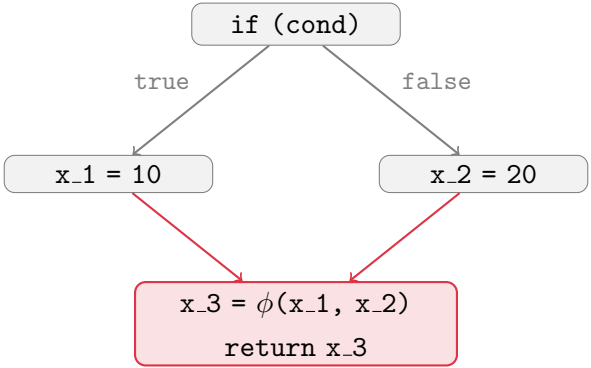


É **óbvio** que `z_1` depende de `x_2`. O versionamento cria ligações **Uso-Def** diretas e de complexidade $O(1)$.

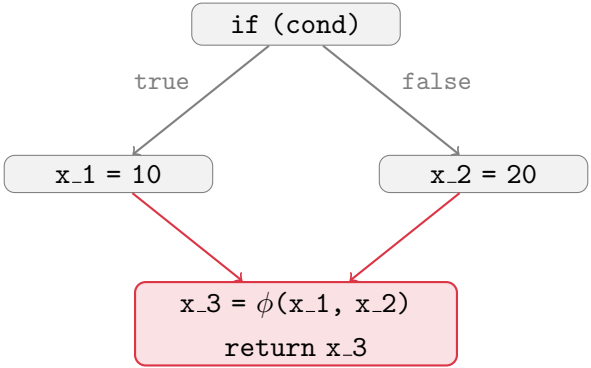
E quando o fluxo bifurca? A função ϕ (Phi)



E quando o fluxo bifurca? A função ϕ (Phi)



E quando o fluxo bifurca? A função ϕ (Phi)



A instrução ϕ (*Phi*) seleciona "magicamente" o valor correto de `x` dependendo de **qual bloco básico** o fluxo de execução veio.

Hierarquia da LLVM IR

O código em LLVM é estruturado em uma hierarquia de containers bem definida:

- **Module (Módulo):** Representa uma unidade de compilação (ex: `programa.c`). Contém funções, variáveis globais e metadados.

Hierarquia da LLVM IR

O código em LLVM é estruturado em uma hierarquia de containers bem definida:

- **Module (Módulo):** Representa uma unidade de compilação (ex: `programa.c`). Contém funções, variáveis globais e metadados.
- **Function (Função):** Composta por argumentos e um grafo de *Blocos Básicos*.

Hierarquia da LLVM IR

O código em LLVM é estruturado em uma hierarquia de containers bem definida:

- **Module (Módulo):** Representa uma unidade de compilação (ex: `programa.c`). Contém funções, variáveis globais e metadados.
- **Function (Função):** Composta por argumentos e um grafo de *Blocos Básicos*.
- **Basic Block (Bloco Básico):** Sequência de instruções de execução linear. Termina **sempre** com uma instrução terminadora (ex: `br`, `ret`).

Hierarquia da LLVM IR

O código em LLVM é estruturado em uma hierarquia de containers bem definida:

- **Module (Módulo):** Representa uma unidade de compilação (ex: `programa.c`). Contém funções, variáveis globais e metadados.
- **Function (Função):** Composta por argumentos e um grafo de *Blocos Básicos*.
- **Basic Block (Bloco Básico):** Sequência de instruções de execução linear. Termina **sempre** com uma instrução terminadora (ex: `br`, `ret`).
- **Instruction (Instrução):** A unidade mais granular (ex: `add`, `load`), que realiza uma única operação.

Hierarquia da LLVM IR

O código em LLVM é estruturado em uma hierarquia de containers bem definida:

- **Module (Módulo):** Representa uma unidade de compilação (ex: `programa.c`). Contém funções, variáveis globais e metadados.
- **Function (Função):** Composta por argumentos e um grafo de *Blocos Básicos*.
- **Basic Block (Bloco Básico):** Sequência de instruções de execução linear. Termina **sempre** com uma instrução terminadora (ex: `br`, `ret`).
- **Instruction (Instrução):** A unidade mais granular (ex: `add`, `load`), que realiza uma única operação.

Module → Function → Basic Block → Instruction

Sistema de Tipos da LLVM IR

Tipo	Descrição	Exemplo
`i1`	Booleano (1 bit)	condições
`i8`	Inteiro 8 bits (byte)	char
`i32`	Inteiro 32 bits	int
`i64`	Inteiro 64 bits	long
`float`	Ponto flutuante 32 bits	float
`double`	Ponto flutuante 64 bits	double
`ptr`	Ponteiro (opaco)	ponteiros
`void`	Sem retorno	procedimentos
`<4 x float>`	Vetor SIMD	SSE/AVX

Sistema de Tipos da LLVM IR

Tipo	Descrição	Exemplo
`i1`	Booleano (1 bit)	condições
`i8`	Inteiro 8 bits (byte)	char
`i32`	Inteiro 32 bits	int
`i64`	Inteiro 64 bits	long
`float`	Ponto flutuante 32 bits	float
`double`	Ponto flutuante 64 bits	double
`ptr`	Ponteiro (opaco)	ponteiros
`void`	Sem retorno	procedimentos
`<4 x float>`	Vetor SIMD	SSE/AVX

LLVM IR é **fortemente tipada**: `add i32 %a, %b` — o tipo é explícito em cada instrução.

Instruções Principais

Aritmética:

- `add`, `sub`, `mul`, `sdiv`
- `fadd`, `fsub`, `fmul`, `fdiv`

Comparação:

- `icmp eq/ne/slt/sgt/sle/sge`
- `fcmp oeq/ogt/olt/...`

Memória:

- `alloca` — alocar na pilha
- `load` — ler da memória
- `store` — escrever na memória

Controle:

- `br` — salto (cond/incond)
- `ret` — retorno
- `call` — chamada de função
- `phi` — junção SSA

Entendendo o GEP: A Analogia do Carteiro

A instrução `getelementptr` (**GEP**) é a mais confusa da LLVM. Pense nela como um **GPS para o carteiro**:

Endereço Base (`%A`)
"Vá para o Condomínio X"

Índices (`0, 3`)
"Bloco 0, Apartamento 3"

Resultado do GEP
Coordenadas exatas da porta do Apt 3.

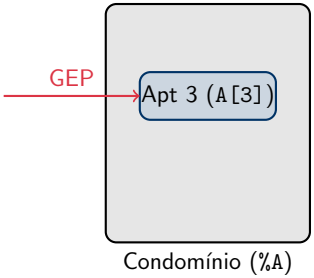
Entendendo o GEP: A Analogia do Carteiro

A instrução `getelementptr` (**GEP**) é a mais confusa da LLVM. Pense nela como um **GPS para o carteiro**:

Endereço Base (`%A`)
"Vá para o Condomínio X"

Índices (`0, 3`)
"Bloco 0, Apartamento 3"

Resultado do GEP
Coordenadas exatas da porta do Apt 3.



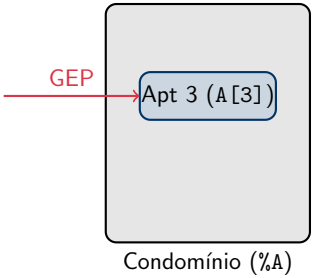
Entendendo o GEP: A Analogia do Carteiro

A instrução `getelementptr` (**GEP**) é a mais confusa da LLVM. Pense nela como um **GPS para o carteiro**:

Endereço Base (`%A`)
"Vá para o Condomínio X"

Índices (`0, 3`)
"Bloco 0, Apartamento 3"

Resultado do GEP
Coordenadas exatas da porta do Apt 3.



Importante!

O GEP **não entra no apartamento** (não faz `load`) e **não deixa o pacote** (não faz `store`). Ele **apenas calcula o endereço** da porta!

Dissecando o GEP (Exemplo com Array)

Por que o GEP usa **dois** índices para acessar um vetor simples?

Código em C

```
1 int A[10];
2 A[3] = 42;
```

Dissecando o GEP (Exemplo com Array)

Por que o GEP usa **dois** índices para acessar um vetor simples?

Código em C

```
1 int A[10];
2 A[3] = 42;
```

Código em LLVM IR

```
%ptr = getelementptr [10 x i32],
                     ptr %A,
                     i32 0, i32 3
store i32 42, ptr %ptr
```

Dissecando o GEP (Exemplo com Array)

Por que o GEP usa **dois** índices para acessar um vetor simples?

Código em C

```
1 int A[10];
2 A[3] = 42;
```

Código em LLVM IR

```
%ptr = getelementptr [10 x i32],
                     ptr %A,
                     i32 0, i32 3
store i32 42, ptr %ptr
```

Anatomia dos Parâmetros:

- `[10 x i32]`: É o tipo base. A "planta" da estrutura.
- `ptr %A`: O ponteiro inicial (endereço do Condomínio).
- **1º Índice** (`i32 0`): "Ande zero blocos inteiros do tipo `[10 x i32]`". Na prática, significa: *Acesse o próprio array para o qual o ponteiro aponta, não o próximo na memória.*
- **2º Índice** (`i32 3`): "Agora, dentro do array, ande 3 elementos do tipo `i32` (acesse o Apartamento 3)".

A Instrução GEP na Prática

Por que não usar apenas aritmética de ponteiros normal em bytes (ex: `ptr + 12`)?

```
; Exemplo C: int A[10]; A[3] = 42;
; %A eh o ponteiro para o inicio do array:

%ptr = getelementptr [10 x i32], ptr %A, i32 0, i32 3
store i32 42, ptr %ptr
```

- O GEP sabe o **tipo arquitetural** (`[10 x i32]`). Ele sabe que cada "apartamento" possui 4 bytes.

A Instrução GEP na Prática

Por que não usar apenas aritmética de ponteiros normal em bytes (ex: `ptr + 12`)?

```
; Exemplo C: int A[10]; A[3] = 42;
; %A eh o ponteiro para o inicio do array:

%ptr = getelementptr [10 x i32], ptr %A, i32 0, i32 3
store i32 42, ptr %ptr
```

- O GEP sabe o **tipo arquitetural** (`[10 x i32]`). Ele sabe que cada "apartamento" possui 4 bytes.
- **A Mágica da Otimização:** Se o otimizador vê `GEP A, 3` e `GEP A, 4`, ele tem **certeza matemática** de que são endereços diferentes (não há *aliasing*).

A Instrução GEP na Prática

Por que não usar apenas aritmética de ponteiros normal em bytes (ex: `ptr + 12`)?

```
; Exemplo C: int A[10]; A[3] = 42;
; %A eh o ponteiro para o inicio do array:

%ptr = getelementptr [10 x i32], ptr %A, i32 0, i32 3
store i32 42, ptr %ptr
```

- O GEP sabe o **tipo arquitetural** (`[10 x i32]`). Ele sabe que cada "apartamento" possui 4 bytes.
- **A Mágica da Otimização:** Se o otimizador vê `GEP A, 3` e `GEP A, 4`, ele tem **certeza matemática** de que são endereços diferentes (não há *aliasing*).
- Em aritmética nua (`A + x` vs `A + y`), provar que ponteiros não colidem é extremamente complexo para a máquina.

Exemplo 0.1: A Jornada de uma Expressão

Como o frontend transforma uma expressão complexa na representação intermediária?

1. Fonte

```
x := a+b*c;
```

Exemplo 0.1: A Jornada de uma Expressão

Como o frontend transforma uma expressão complexa na representação intermediária?

1. Fonte

```
x := a+b*c;
```

2. AST

```
Assign 'x'
 `-BinaryOp '+'
    |-Ref 'a'
    `-BinaryOp '*'
      |-Ref 'b'
      |-Ref 'c'
```

Exemplo 0.1: A Jornada de uma Expressão

Como o frontend transforma uma expressão complexa na representação intermediária?

1. Fonte

```
x := a+b*c;
```

2. AST

```
Assign 'x'
 `-BinaryOp '+'
    |-Ref 'a'
    `-BinaryOp '*'
      |-Ref 'b'
      |-Ref 'c'
```

3. 3AC Genérico

```
t1 = b * c
t2 = a + t1
x  = t2
```

Exemplo 0.1: A Jornada de uma Expressão

Como o frontend transforma uma expressão complexa na representação intermediária?

1. Fonte

```
x := a+b*c;
```

2. AST

```
Assign 'x'
 '-BinaryOp '+'
   |-Ref 'a'
   '-BinaryOp '*'
     |-Ref 'b'
     |-Ref 'c'
```

3. 3AC Genérico

```
t1 = b * c
t2 = a + t1
x  = t2
```

4. LLVM IR

```
%1 = mul i32 %b, %c
%2 = add i32 %a, %1
store i32 %2, ptr %x
```

Exemplo 0.1: A Jornada de uma Expressão

Como o frontend transforma uma expressão complexa na representação intermediária?

1. Fonte

```
x := a+b*c;
```

2. AST

```
Assign 'x'
 `-BinaryOp '+'
    |-Ref 'a'
    `-BinaryOp '*'
      |-Ref 'b'
      |-Ref 'c'
```

3. 3AC Genérico

```
t1 = b * c
t2 = a + t1
x  = t2
```

4. LLVM IR

```
%1 = mul i32 %b, %c
%2 = add i32 %a, %1
store i32 %2, ptr %x
```

O princípio do 3AC: O compilador "achata" a árvore de expressões (Passo 2) em um formato intermediário livre (Passo 3), para então gerar a IR tipada e estrita (Passo 4).

Exemplo 0.2: A Jornada de uma Função

Elegância e Universalidade: abstraindo os detalhes de cada linguagem.

1. Código Fonte (Pascal)

```
function soma(a, b:
    integer): integer;
begin
  soma := a + b;
end;
```

Exemplo 0.2: A Jornada de uma Função

Elegância e Universalidade: abstraindo os detalhes de cada linguagem.

1. Código Fonte (Pascal)

```
function soma(a, b:
    integer): integer;
begin
  soma := a + b;
end;
```

2. Árvore Sintática (AST)

```
1 FunctionDecl 'soma'
2  |-Param 'a' Integer
3  |-Param 'b' Integer
4  `-Block
5    `-Assign 'soma'
6      `-BinaryOp '+' ...
```

Exemplo 0.2: A Jornada de uma Função

Elegância e Universalidade: abstraindo os detalhes de cada linguagem.

1. Código Fonte (Pascal)

```
function soma(a, b:
    integer): integer;
begin
  soma := a + b;
end;
```

2. Árvore Sintática (AST)

```
1 FunctionDecl 'soma'
2  |-Param 'a' Integer
3  |-Param 'b' Integer
4  `-Block
5    `-Assign 'soma'
6      `-BinaryOp '+' ...
```

3. Geração de LLVM IR

```
define i32 soma(i32 entry:ret
    i32
```

Exemplo 0.2: A Jornada de uma Função

Elegância e Universalidade: abstraindo os detalhes de cada linguagem.

1. Código Fonte (Pascal)

```
function soma(a, b:
    integer): integer;
begin
  soma := a + b;
end;
```

2. Árvore Sintática (AST)

```
1 FunctionDecl 'soma'
2  |-Param 'a' Integer
3  |-Param 'b' Integer
4  `-Block
5    `-Assign 'soma'
6      `-BinaryOp '+' ...
```

3. Geração de LLVM IR

```
define i32 soma(i32 entry:ret
    i32
```

O **Frontend** abstrai as idiossincrasias do Pascal (como usar o nome da função como retorno) e gera um bloco básico universal, tipado e estrito.

Exemplo 0.3: A Jornada do Controle de Fluxo

Como o compilador lida com bifurcações no formato SSA?

1. Código Fonte (Pascal)

```
if a > b then
  max := a
else
  max := b;
```

Exemplo 0.3: A Jornada do Controle de Fluxo

Como o compilador lida com bifurcações no formato SSA?

1. Código Fonte (Pascal)

```
if a > b then
  max := a
else
  max := b;
```

2. Árvore Sintática (AST)

```
IfStmt
 |-BinaryOp '>'
 | |-Ref 'a'
 | `-Ref 'b'
 |-Then: Assign 'max'
 `-Else: Assign 'max'
```

Exemplo 0.3: A Jornada do Controle de Fluxo

Como o compilador lida com bifurcações no formato SSA?

1. Código Fonte (Pascal)

```
if a > b then
  max := a
else
  max := b;
```

2. Árvore Sintática (AST)

```
IfStmt
 |-BinaryOp '>'
 | |-Ref 'a'
 | `-Ref 'b'
 |-Then: Assign 'max'
 `-Else: Assign 'max'
```

3. Geração de LLVM IR

```
  %cmp = icmp sgt i32 %a, %b
  br i1 %cmp, label %thn,
      label %els
thn: br label %end
els: br label %end
end:
  %max = phi i32 [%a, %thn],
      [%b, %els]
```

Exemplo 0.3: A Jornada do Controle de Fluxo

Como o compilador lida com bifurcações no formato SSA?

1. Código Fonte (Pascal)

```
if a > b then
  max := a
else
  max := b;
```

2. Árvore Sintática (AST)

```
1 IfStmt
2  |-BinaryOp '>'
3  | |-Ref 'a'
4  | `-Ref 'b'
5  |-Then: Assign 'max'
6  `-Else: Assign 'max'
```

3. Geração de LLVM IR

```
  %cmp = icmp sgt i32 %a, %b
  br i1 %cmp, label %thn,
      label %els
thn: br label %end
els: br label %end
end:
  %max = phi i32 [%a, %thn],
      [%b, %els]
```

A AST condicional é transformada num **Grafo de Controle de Fluxo** (com nós `thn` e `els`).
A magia matemática da **função** ϕ garante a atribuição única (SSA)!

Exemplo 1: Função Simples

```
define i32 soma(i32 entry:ret i32
```

Exemplo 1: Função Simples

```
define i32 soma(i32 entry:ret i32
```

Anatomia (Por que essas instruções?)

- `define ... @soma(...)`: Declara a função global (o `@` indica escopo global).
- `entry:`: Label que cria o **Bloco Básico inicial**. O LLVM exige que toda função tenha pelo menos um bloco básico.
- `add i32 ...`: Operação matemática restrita a no máximo 3 endereços (3AC).
- `ret i32 ...`: Instrução **terminadora**. Todo bloco básico no LLVM é obrigado a terminar com um desvio (`br`) ou um retorno (`ret`).

Exemplo 2: If/Else com ϕ

```
define i32 max(i32 entry:br i1 then:br label else:br label merge:ret i32
```

Exemplo 2: If/Else com ϕ

```
define i32 max(i32 entry:br i1 then:br label else:br label merge:ret i32
```

Anatomia (Bifurcações e ϕ):

- `icmp sgt ...`: Faz a comparação (*Signed Greater Than*) e gera um booleano de 1 bit (`i1`).
- `br i1 %cmp, ...`: Desvio condicional. Encerra o bloco `entry` e divide o fluxo em dois caminhos.
- `phi i32 ...`: O nó mágico que consolida o SSA. Ele diz: *"Assuma o valor `%a` se viermos do bloco `then`, ou `%b` se do `else`"*.

Exemplo 3: Loop com ϕ

```
define i32 somaAteN(i32 entry:br label loop:br i1 exit:ret i32
```

Exemplo 3: Loop com ϕ

```
define i32 somaAteN(i32 entry:br label loop:br i1 exit:ret i32
```

Anatomia (Loops em SSA):

- `br label %loop`: Desvio incondicional para mergulhar no loop.
- O bloco `loop:` precisa de dois `phi` para atualizar o estado:
  - `%i`: Inicia com `0` (do `entry`) e recebe `%next` nas próximas voltas.
  - `%sum`: Inicia com `0` (do `entry`) e acumula `%new`.
- `icmp slt ...`: Verifica se a condição de parada foi atingida (*Signed Less Than*).

Usando Clang, opt e llc

```
# 1. Gerar LLVM IR a partir de C
clang -S -emit-llvm programa.c -o programa.ll

# 2. Aplicar otimizacoes
opt -S -mem2reg -instcombine programa.ll -o opt.ll

# 3. Gerar assembly nativo
llc opt.ll -o programa.s

# 4. Ou compilar direto para executavel
clang programa.c -O2 -o programa
```

Usando Clang, opt e llc

```
# 1. Gerar LLVM IR a partir de C
clang -S -emit-llvm programa.c -o programa.ll

# 2. Aplicar otimizacoes
opt -S -mem2reg -instcombine programa.ll -o opt.ll

# 3. Gerar assembly nativo
llc opt.ll -o programa.s

# 4. Ou compilar direto para executavel
clang programa.c -O2 -o programa
```

Dica

Use `clang -S -emit-llvm -O0` para ver a IR sem otimizações (mais legível). Com `-O2`, a IR já vem otimizada.

Passes de Otimização Comuns

Pass	O que faz
`-mem2reg`	Promove `alloca/load/store` para registradores SSA
`-instcombine`	Simplifica instruções (constant folding, strength reduction)
`-dce`	Dead Code Elimination
`-simplifycfg`	Simplifica o grafo de controle (remove blocos vazios)
`-inline`	Inlining de funções pequenas
`-gvn`	Global Value Numbering (elimina subexpressões comuns)

Passes de Otimização Comuns

Pass	O que faz
`-mem2reg`	Promove `alloca/load/store` para registradores SSA
`-instcombine`	Simplifica instruções (constant folding, strength reduction)
`-dce`	Dead Code Elimination
`-simplifycfg`	Simplifica o grafo de controle (remove blocos vazios)
`-inline`	Inlining de funções pequenas
`-gvn`	Global Value Numbering (elimina subexpressões comuns)

Cada pass transforma LLVM IR $\rightarrow$ LLVM IR. O pipeline `-O2` aplica dezenas de passes encadeadas.

Visualizando o Grafo de Controle de Fluxo (CFG)

A infraestrutura LLVM permite visualizar a estrutura de blocos básicos facilmente:

```
# 1. Gera um arquivo .dot para cada funcao da IR
opt -dot-cfg programa.ll

# 2. Converte o grafo .dot para PDF usando o Graphviz
dot -Tpdf .soma.dot -o cfg_soma.pdf
```

- Excelente ferramenta para debugar compiladores em desenvolvimento.
- Mostra visualmente os `Basic Blocks` e as arestas de salto (`br`), indicando caminhos de `true`/`false`.

Exercício: Leitura de LLVM IR

Dado o seguinte código C:

Código C

```
int abs(int x) {
  if (x < 0)
    return -x;
  else
    return x;
}
```

Perguntas:

1. Quantos blocos básicos terá a LLVM IR?
2. Qual instrução LLVM faz a comparação `x < 0`?
3. Haverá ϕ? Se sim, onde?
4. Como é representada a negação `-x`?

Solução do Exercício

```
define i32 abs(i32 entry:br i1 neg:br label pos:br label merge:ret i32
```

4 blocos, `icmp slt` para <, ϕ no merge, negação via `sub i32 0, %x`.

Resumo da Aula

- **LLVM** é a plataforma de compilação mais influente da atualidade.
- **LLVM IR:** assembly portável, tipado, SSA. Registradores virtuais (%).
- Instruções: aritmética, comparação, memória, controle, ϕ.
- Pipeline: `frontend` $\rightarrow$ `.ll` $\rightarrow$ `opt` $\rightarrow$ `llc` $\rightarrow$ `.s/.o`.
- Passes de otimização transformam IR $\rightarrow$ IR.

Conexão com o curso

LLVM IR materializa os conceitos de **3AC** (Aula 18) e **SSA** (Aula 19) em uma ferramenta industrial real.

Próxima Aula: Gerenciamento de Memória — pilha de ativação, heap, garbage collection.

Referências

- **AHO, A. V. et al.** *Compiladores: Princípios, Técnicas e Ferramentas*. 2ª Edição.
- **COOPER, K.; TORCZON, L.** *Engineering a Compiler*. 2ª Edição.
- **LLVM Language Reference:** `https://llvm.org/docs/LangRef.html`
- **LLVM Tutorial:** `https://llvm.org/docs/tutorial/`
- **LATTNER, C.** "LLVM: An Infrastructure for Multi-Stage Optimization." Tese de MS, UIUC, 2002.

Dúvidas?

Obrigado!

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