Compiladores

Aula 19: Representação SSA
(Static Single Assignment)

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:

- Compreender o princípio da **Static Single Assignment (SSA)** e sua motivação dentro do pipeline de um compilador.
- Entender por que SSA simplifica **análises e otimizações** de código.
- Dominar o conceito da **função** ϕ e as regras que determinam onde ela deve ser inserida.
- Conhecer o fluxo de conversão de IR convencional para SSA: dominância $\rightarrow$ inserção de ϕ $\rightarrow$ renomeação de variáveis.
- Relacionar SSA com compiladores reais (GCC, LLVM, HotSpot JVM).

2 Motivação: O Problema da Ambiguidade na IR Convencional

Depois que o compilador realiza a análise semântica e gera uma **Representação Intermediária (IR)** de baixo nível — tipicamente o *código de três endereços (3AC)* — ele precisa realizar **análises de fluxo de dados** para habilitar otimizações como propagação de constantes, eliminação de código morto e eliminação de subexpressões comuns.

O grande obstáculo do 3AC convencional é que uma mesma variável pode receber **múltiplas definições** ao longo do programa. Considere o trecho abaixo:

Exemplo: Ambiguidade na IR Convencional

```
1 x = 1
2 x = 2
3 y = x        // qual definicao de x chega aqui?
```

Para saber *qual* definição de `x` chega até o uso em `y = x`, o compilador precisa executar uma análise chamada ***reaching definitions*** (definições ativas). Essa análise itera em ponto-fixo sobre o Grafo de Fluxo de Controle (CFG), o que tem custo proporcional ao tamanho do grafo multiplicado pelo número de iterações até convergir.

△ Importante

Custo da análise tradicional: para cada *uso* de uma variável, o compilador pode precisar rastrear todas as definições que *possivelmente* chegam até aquele ponto, o que torna cada transformação complexa e propensa a erros de implementação.

A pergunta natural é: *e se eliminássemos essa ambiguidade por design?* Essa é exatamente a intuição por trás de SSA.

3 O Princípio SSA

Definição: Static Single Assignment (SSA)

Uma IR está em **forma SSA** se e somente se cada variável for **definida exatamente uma vez** no texto (estático) do programa, e cada uso de uma variável se refere à **única definição** que a domina.

A palavra *estática* é importante: em SSA, contamos definições no texto do programa, não em tempo de execução. Um laço que executa mil vezes ainda possui uma única definição estática no cabeçalho do laço.

3.1 Versionamento de Variáveis

Para transformar IR convencional em SSA, **versionamos** as variáveis: toda vez que uma variável receberia uma nova atribuição, criamos uma nova *versão* (subescrito), que é tratada como uma variável completamente distinta.

Versionamento em Código Linear

IR convencional

```
x = 1
x = 2
y = x
```

IR em SSA

```
x1 = 1
x2 = 2
y1 = x2     // sem ambiguidade
```

Com SSA, y_1 usa *definitivamente* x_2. A análise de *reaching definitions* é trivial: cada uso aponta diretamente para sua única definição.

3.2 Exemplo Mais Complexo

Sequência de Operações

IR convencional

```
a = 5
b = a + 1
a = b * 2
c = a + b
```

IR em SSA

```
a1 = 5
b1 = a1 + 1
a2 = b1 * 2
c1 = a2 + b1
```

Observações:

- `a1` e `a2` são variáveis **independentes**, com intervalos de vida não sobrepostos.
- `c1` usa `a2` (a definição dominante), nunca `a1`.
- Se `a1` não fosse usada após `b1 = a1 + 1`, o compilador poderia imediatamente propagar o valor `5` e eliminar `a1`.

4 A Função ϕ (Phi)

SSA funciona perfeitamente para código **linear**. O desafio surge quando existem **junções de fluxo de controle** (convergência de dois ou mais caminhos no CFG), como ocorre após um `if-else` ou na entrada de um laço.

4.1 O Problema das Junções

Considere o seguinte código com ramificação:

```
if (cond)
    x = 1;
else
    x = 2;
print(x);       // de onde vem x?
```

Ao tentar converter para SSA, criamos:

- No bloco *then*: `x₁ = 1`
- No bloco *else*: `x₂ = 2`
- No bloco de junção: `print(???)` — qual versão usar?

Não podemos escolher estaticamente uma das duas, pois isso dependeria do valor de `cond` em tempo de execução.

Solução: Função ϕ

Uma **função ϕ** (lê-se "phi") é uma instrução especial inserida nos pontos de junção do CFG. Ela aceita **um argumento por predecessor** do bloco e seleciona, em tempo de execução, o valor proveniente do caminho que efetivamente foi executado:

$$x_k = \phi(x_i : B_i,\ x_j : B_j,\ \ldots)$$

onde cada par (x_i, B_i) indica "se o fluxo veio do bloco B_i, usar x_i".

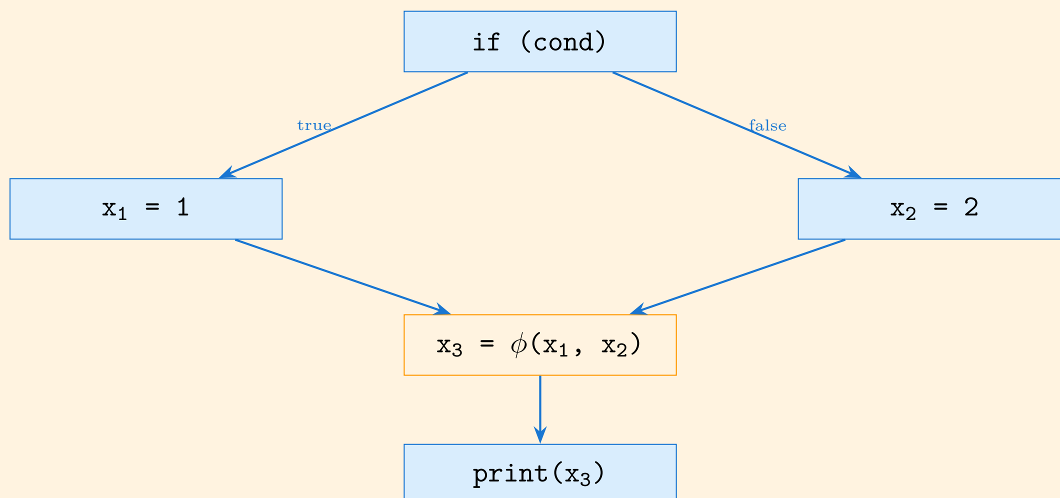
4.2 Função ϕ em Ação

Aplicando a função ϕ ao exemplo anterior:

if-else com Função ϕ

```
1                if (cond) goto THEN else goto ELSE
2 THEN:          x1 = 1
3                goto MERGE
4 ELSE:          x2 = 2
5                goto MERGE
6 MERGE:         x3 = phi(x1:THEN, x2:ELSE)   // seleciona
    conforme o predecessor
7                print(x3)
```

O diagrama de fluxo correspondente:



4.3 Propriedades da Função ϕ

- ϕ possui **exatamente um argumento por predecessor** do bloco onde é inserida.
- ϕ é uma instrução **fictícia** (não existe em nenhuma ISA real): ela é resolvida durante a saída de SSA (*SSA deconstruction*).
- Na geração de código real, cada ϕ é implementada via **movimentação de registradores** nos predecessores, e o alocador de registradores pode eliminar essas movimentações.
- ϕ aparece **apenas no início** de blocos com dois ou mais predecessores.
- Um mesmo bloco pode conter **múltiplas** funções ϕ (uma para cada variável que necessita de mesclagem).

4.4 Função ϕ em Laços

Laços também geram funções ϕ: o cabeçalho do laço tem dois predecessores — a entrada do laço e a aresta de *back-edge* (iteração seguinte).

Laço com Função ϕ

Código original:

```
1 i = 0
2 while (i < 10)
3     i = i + 1
```

Após conversão para SSA:

```
1         i1 = 0
2         goto L1
3 L1:     i2 = phi(i1:entry, i3:L2)    // i1 na 1a iteracao, i3
      nas seguintes
4         if i2 < 10 goto L2
5         goto L3
6 L2:     i3 = i2 + 1
7         goto L1
8 L3:     ...                           // i2 contem o valor
     final
```

i_2 recebe i_1 na primeira execução do cabeçalho e i_3 em todas as iterações subsequentes.

◇ Informação

Por que a ϕ de laço é importante? Ela torna o **grafo de uso–definição** de variáveis de indução explícito: i_2 depende de i_1 (inicialização) e de i_3 (passo). Isso facilita a *análise de variáveis de indução* e a *vetorização automática*.

5 Conversão para SSA

A transformação de IR convencional para SSA é realizada em três etapas principais, que dependem da estrutura de dominância do CFG.

Pipeline de Conversão

IR convencional $\xrightarrow{1}$ Calcular dominância $\xrightarrow{2}$ Inserir ϕ $\xrightarrow{3}$ Renomear variáveis $\Rightarrow$ SSA

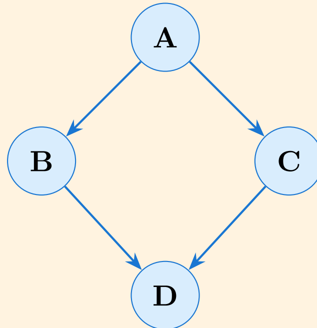
5.1 Passo 1: Calcular Dominância

Definição: Dominância

Um bloco B **domina** um bloco B' (escrito BB') se e somente se **todo caminho** da entrada do programa até B' passa por B.

Cada bloco domina a si mesmo. A relação de dominância define uma **árvore de dominância**, onde o pai de B' é o seu **dominador imediato** (*idom*).

Exemplo de Dominância



- **A** domina todos os blocos (é o único ponto de entrada).
- **B** *não* domina D (existe o caminho $A \to C \to D$).
- **C** *não* domina D (existe o caminho $A \to B \to D$).
- **D** domina apenas a si mesmo.

5.1.1 Fronteira de Dominância

Definição: Fronteira de Dominância

A **fronteira de dominância** de um bloco B ($\text{DF}(B)$) é o conjunto de blocos Y tais que:

- B domina algum predecessor de Y, **e**
- B *não* domina Y de forma estrita.

Em outras palavras, $\text{DF}(B)$ são os primeiros blocos "além do alcance" direto de B — exatamente os pontos onde o controle pode vir de outro caminho.

No exemplo acima, $\text{DF}(B) = \{D\}$ e $\text{DF}(C) = \{D\}$. Isso significa que qualquer variável definida em B ou C que seja usada após a junção em D precisará de uma função ϕ em D.

5.2 Passo 2: Inserção das Funções ϕ

O algoritmo clássico de Cytron et al. (1991) insere funções ϕ usando o conceito de **fronteira de dominância iterated** (DF^+):

1. Para cada variável v com definições nos blocos $\{D_1, D_2, \ldots\}$, calcule $\text{DF}^+(D_1 \cup D_2 \cup \cdots)$.
2. Insira uma instrução $v = \phi(v, v, \ldots)$ no início de cada bloco em DF^+ que ainda não possua tal ϕ.

3 A inserção de uma ϕ cria uma nova definição de v, o que pode expandir DF^+ — repita até convergir.

◇ Informação

Eficiência: Cytron et al. demonstraram que esse algoritmo é linear no tamanho do programa (número de instruções mais número de arestas do CFG), tornando a construção de SSA eficiente mesmo para programas grandes.

5.3 Passo 3: Renomeação das Variáveis

Com as funções ϕ inseridas, percorremos a **árvore de dominância** em pré-ordem e renomeamos cada definição de variável de acordo com uma pilha de versões correntes:

1 Para cada instrução do bloco corrente:

- Substitui os **usos** de v pelo topo da pilha de v.
- Substitui cada **definição** de v por uma nova versão v_k e empilha v_k.

2 Para cada **sucessor** no CFG, preenche os argumentos das funções ϕ correspondentes ao bloco corrente.

3 Recursa para os filhos na **árvore de dominância**.

4 Ao retornar, **desempilha** as versões criadas neste bloco.

5.4 Exemplo Passo a Passo

Considere o seguinte código 3AC:

```
1         a = 5
2         b = a + 1
3         if b > 3 goto L1
4         a = 10
5         goto L2
6   L1:   a = 20
7   L2:   c = a + b
```

Análise de dominância

O CFG possui quatro blocos: `ENTRY` (instruções 1–3), `THEN` (`a = 10`), `ELSE` (`a = 20`) e `MERGE` (`c = a + b`). `ENTRY` domina todos; `MERGE` é fronteira de dominância de `THEN` e `ELSE`.

Inserção de ϕ

`a` é definida em `THEN` e `ELSE`: inserimos ϕ em `MERGE`. `b` é definida apenas em `ENTRY`, que domina todo o CFG: **não** precisa de ϕ.

Renomeação

Resultado Final em SSA

```
1          a1 = 5
2          b1 = a1 + 1
3          if b1 > 3 goto L1
4          a2 = 10
5          goto L2
6  L1:     a3 = 20
7  L2:     a4 = phi(a2:THEN, a3:ELSE)
8          c1 = a4 + b1
```

Respostas às perguntas de análise:

- **3 versões** de `a`: a_1, a_2, a_3 (mais a_4 produzida pela ϕ).
- ϕ em `L2` porque `a` é definida em ambos os caminhos que chegam a `L2`.
- **Não** é necessária ϕ para `b`: existe apenas uma definição (b_1) que alcança todos os usos.

6 Vantagens para Otimização

SSA é amplamente adotada porque torna várias otimizações **triviais** ou **significativamente mais baratas**. A seguir, vemos as principais.

6.1 Propagação de Constantes

Em IR convencional, para saber se um uso de `x` pode ser substituído por uma constante, precisamos de análise de *reaching definitions* — potencialmente complexa. Em SSA:

Propagação de Constantes em SSA

Se a **única definição** que alcança um uso é x_k `= c` (com c constante), então substituímos diretamente o uso por c. Como cada uso aponta para **exatamente uma** definição, a verificação é $O(1)$ por uso.

6.2 Eliminação de Código Morto

Uma definição v_k é **morta** se não possui nenhum uso (o conjunto de usos é vazio). Em SSA, essa verificação é imediata — basta checar a lista de usos associada a v_k.

6.3 Propagação de Cópias

Se existe uma instrução y_1 `=` x_2, podemos substituir todos os usos de y_1 por x_2 diretamente. A identidade `use-def` única garante que isso é seguro sem nenhuma análise

adicional.

6.4 Exemplo Encadeado de Otimizações

Sequência de Otimizações sobre SSA

IR inicial em SSA:

```
x1 = 10
y1 = x1          // copia de x1
z1 = y1 + 5    // usa y1
```

Passo 1 — Propagação de Cópias: substituímos y_1 por x_1 em z_1:

```
x1 = 10
y1 = x1          // ainda existe, mas y1 pode ficar morta
z1 = x1 + 5
```

Passo 2 — Propagação de Constantes: $x_1 = 10$ é uma constante; substituímos:

```
x1 = 10
y1 = x1
z1 = 10 + 5
```

Passo 3 — Constant Folding: avaliamos `10 + 5 = 15`:

```
x1 = 10
y1 = x1
z1 = 15
```

Passo 4 — Eliminação de Código Morto: se x_1 e y_1 não têm mais usos, são eliminadas:

```
z1 = 15          // resultado final
```

6.5 Alocação de Registradores

Em SSA, os intervalos de vida das variáveis têm propriedades especiais (*liveness* estruturada pela dominância) que simplificam a construção de **grafos de interferência**. O LLVM aloca registradores diretamente sobre nomes SSA, sem precisar computar o grafo de interferência completo em muitos casos.

7 SSA em Compiladores Reais

Compilador / VM	Uso de SSA	Desde
GCC	GIMPLE em SSA (*middle-end*)	4.0 (2005)
LLVM	LLVM IR é **nativamente** SSA	2003
HotSpot JVM	C2 (*Sea of Nodes*, SSA)	1999
V8 (JavaScript)	Turbofan (SSA)	2016
Rust (rustc)	Via LLVM IR (SSA)	2010
Swift	Via LLVM IR + SIL (SSA)	2014

◇ Informação

SSA é o estado da arte. Praticamente todo compilador moderno de alto desempenho opera sobre SSA, pois ela fornece a estrutura mais eficiente para análises e otimizações avançadas.

7.1 LLVM IR: SSA Nativo

A LLVM IR exemplifica SSA ao extremo: cada instrução que produz um valor define um *virtual register* que é imutável. O frontend (Clang, por exemplo) não precisa gerar SSA diretamente; ele gera alocações na pilha (`alloca`) e o passe **`mem2reg`** do LLVM converte automaticamente para SSA.

LLVM IR em SSA

```
1  define i32 @max(i32 %a, i32 %b) {
2  entry:
3    %cmp = icmp sgt i32 %a, %b
4    br i1 %cmp, label %then, label %else
5  then:
6    br label %merge
7  else:
8    br label %merge
9  merge:
10   %result = phi i32 [ %a, %then ], [ %b, %else ]
11   ret i32 %result
12 }
```

Observe a instrução `phi` em `merge`: é literalmente a função ϕ de SSA.

8 Saída de SSA (*SSA Deconstruction*)

Antes de gerar código de máquina, o compilador precisa **eliminar as funções** ϕ, pois não há instrução equivalente em nenhuma ISA real.

O processo clássico substitui cada ϕ por **cópias** nos predecessores: para `x₃ = φ(x₁:B1, x₂:B2)`, inserimos $x_3 = x_1$ ao final de B1 e $x_3 = x_2$ ao final de B2.

⚠ Importante

Problema da Troca Simultânea (*Lost Copy / Swap Problem*): quando múltiplas ϕ em um mesmo bloco envolvem as mesmas variáveis, a inserção ingênua de cópias pode criar dependências circulares. Algoritmos corretos de eliminação de ϕ (Boissinot et al., 2009) resolvem isso verificando a necessidade de variáveis temporárias.

9 Exercício Guiado

Exercício: Conversão para SSA

Converta o seguinte código 3AC para a forma SSA:

```
1       a = 5
2       b = a + 1
3       if b > 3 goto L1
4       a = 10
5       goto L2
6  L1:  a = 20
7  L2:  c = a + b
```

Questões:

1 Quantas versões distintas de `a` existem?

2 Em qual bloco deve ser inserida a função ϕ para `a`? Por quê?

3 É necessária uma ϕ para `b`? Justifique em termos de dominância.

4 Escreva o código completo em SSA.

Solução

```
1       a1 = 5
2       b1 = a1 + 1
3       if b1 > 3 goto L1
4       a2 = 10
5       goto L2
6 L1:   a3 = 20
7 L2:   a4 = phi(a2:THEN, a3:L1)
8       c1 = a4 + b1
```

Respostas:

1 **4 versões:** a_1, a_2, a_3 e a_4 (produzida pela ϕ).

2 ϕ em **L2**, pois este é o ponto de junção onde chegam tanto o caminho que passa por `a = 10` quanto o caminho que passa por `a = 20`. L2 está na fronteira de dominância de ambos os blocos predecessores.

3 **Não** é necessária ϕ para `b`: `b` possui apenas uma definição ($b_1 = a_1 + 1$) no bloco de entrada, que domina todos os blocos do CFG. Portanto, não há junção de definições distintas.

10 Perguntas de Reflexão

1 Se SSA exige funções ϕ em junções, e laços aninhados criam múltiplas junções, um programa com muitos laços aninhados terá muitas ϕ. Isso é um problema prático?

2 Como funciona a "saída de SSA" (*SSA deconstruction*) antes da geração de código final? O que acontece com as funções ϕ?

3 A LLVM IR é nativamente SSA. Isso significa que o frontend (Clang) precisa gerar código diretamente em SSA? Pesquise sobre o passe `mem2reg`.

4 Qual é a relação entre SSA e **Continuous Passing Style** (CPS), a forma funcional equivalente usada em compiladores de linguagens funcionais?

11 Resumo

✓ title=Pontos-Chave

- **SSA:** cada variável é definida **exatamente uma vez**. Versões $(x_1, x_2, \ldots)$ eliminam ambiguidade de forma definitiva.
- **Função ϕ:** inserida nos pontos de junção do CFG, seleciona o valor correto conforme o predecessor de onde o fluxo veio.
- **Conversão:**
  1. Calcular dominância e fronteiras de dominância.
  2. Inserir ϕ nas fronteiras de dominância (algoritmo de Cytron et al.).
  3. Renomear variáveis em pré-ordem da árvore de dominância.
- **Otimizações facilitadas:** propagação de constantes, eliminação de código morto, propagação de cópias, *constant folding*, alocação de registradores.
- **Uso industrial:** GCC (GIMPLE SSA), LLVM (nativamente SSA), HotSpot JVM, V8, Rust, Swift.
- **Saída de SSA:** funções ϕ são eliminadas antes da geração de código de máquina, substituídas por cópias nos predecessores.

12 Referências

- **AHO, A. V.; LAM, M. S.; SETHI, R.; ULLMAN, J. D.** *Compiladores: Princípios, Técnicas e Ferramentas.* 2ª ed. Pearson, 2008. Cap. 9 (Otimizações de Código).
- **COOPER, K.; TORCZON, L.** *Engineering a Compiler.* 2ª ed. Morgan Kaufmann, 2011. Cap. 9 (Data-Flow Analysis), Cap. 10 (SSA Form).
- **CYTRON, R.; FERRANTE, J.; ROSEN, B. K.; WEGMAN, M. N.; ZADECK, F. K.** "Efficiently Computing Static Single Assignment Form and the Control Dependence Graph." *ACM Transactions on Programming Languages and Systems (TOPLAS)*, v. 13, n. 4, p. 451–490, 1991.
- **BOISSINOT, B.; DARTE, A.; RASTELLO, F.; DE DINECHIN, B. D.; GUILLON, C.** "Revisiting Out-of-SSA Translation for Correctness, Code Quality, and Efficiency." *Proceedings of CGO 2009.*
- **Documentação LLVM:** `https://llvm.org/docs/LangRef.html` — Referência completa da LLVM IR, incluindo a instrução `phi`.