Análise de Fluxo de Dados

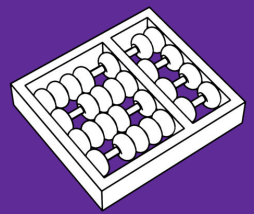
e Otimizações

Hervé Yviquel
herve@ic.unicamp.br

Rev. Sandro Rigo

Universidade Estadual de Campinas (Unicamp)
Instituto de Computação (IC)
Laboratório de Sistemas Computacionais (LSC)

MC921 • Projeto e Construção de Compiladores • 2024 S1



Aula Anterior

Plano

- Arquitetura e Memória
- Alocação de Registro
- Primeiro Exemplo
- Algoritmo de Sethi-Ullman

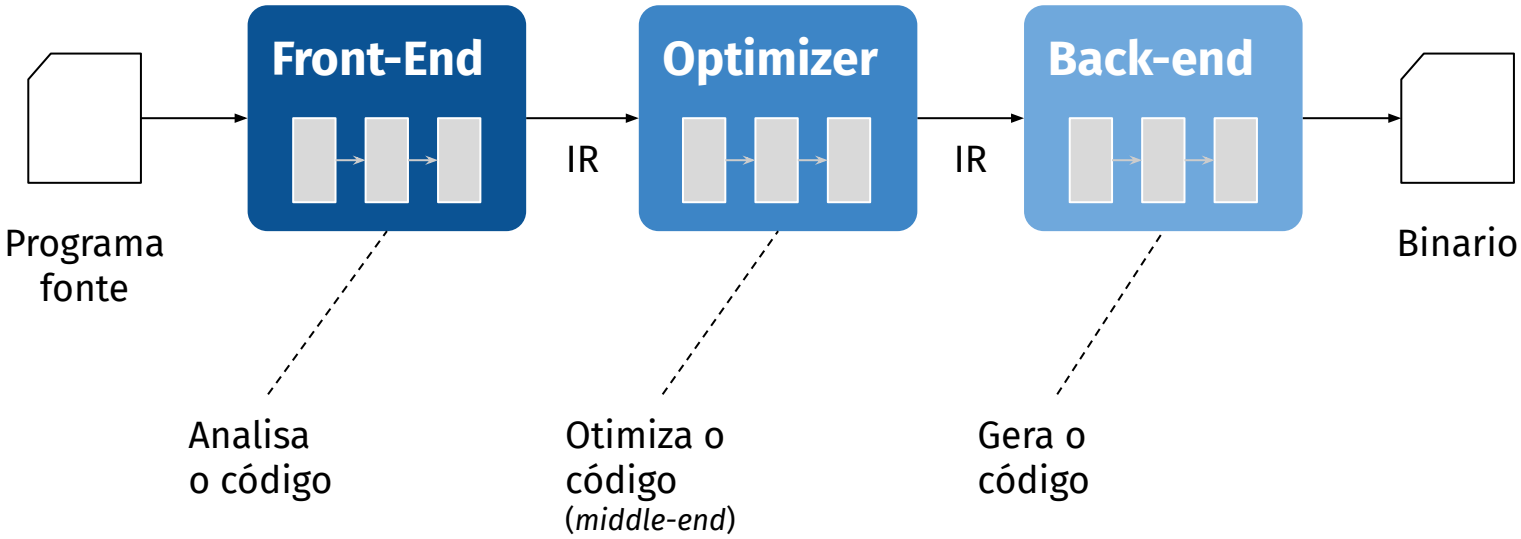
Aula de Hoje

Resumo

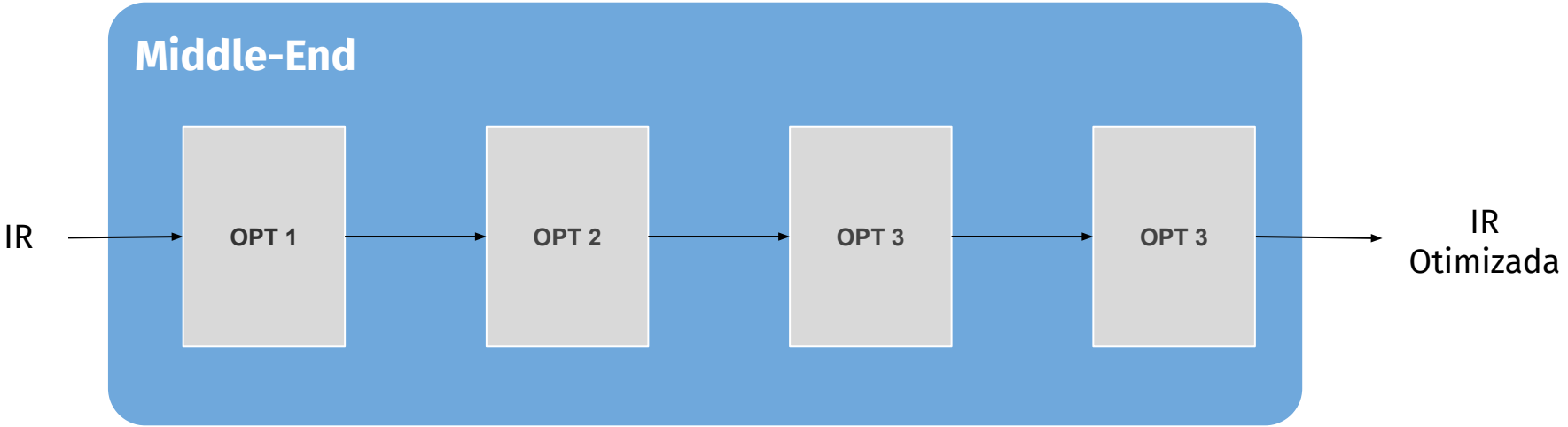
- Introdução
- Primeiros Exemplos de Otimização
- Um Exemplo em Código Real
- Como Fazer Otimizações?
- Reaching Definition
- DU-chain
- Propagação de Constante

Introdução

Estrutura Moderna do Compilador



Middle-End do Compilador



Programador versus Compilador

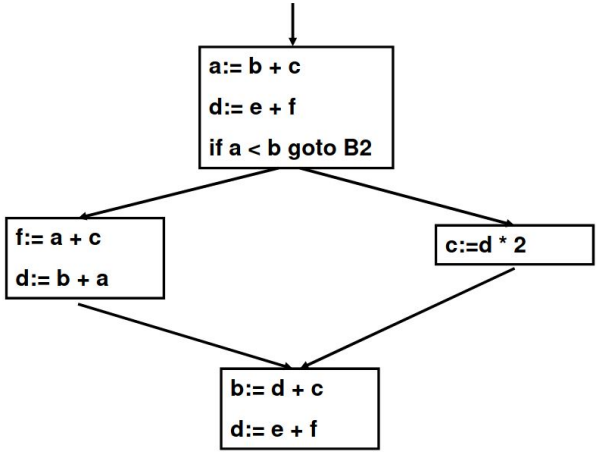
- Melhorar o algoritmo é tarefa do programador
- O compilador pode ser útil para
  - Aplicar transformações que tornam o código gerado mais eficiente
  - Deixar o programador livre para escrever um código limpo

Otimizações

- Transformações para ganho de eficiência
  - Não podem alterar a saída do programa
- Alguns exemplos:
  - **Dead Code Elimination**
    - ➢ Apaga uma computação cujo resultado nunca será usado
  - **Common-subexpression Elimination**
    - ➢ Se uma expressão é computada mais de uma vez, elimine uma das computações
  - **Constant Folding**
    - ➢ Se os operandos são constantes, calcule a expressão em tempo de compilação
  - **Register Allocation**
    - ➢ Reaproveitamento de registradores

Control-Flow Graph (CFG)

- Representação gráfica do código de 3 endereços
  - Nós -> Computação
    - Blocos básicos
  - Arestas -> Fluxo de controle
    - if/else, while/for, break, etc)
- Muito usado em coletas de informações sobre o programa (Análise estática)
  - para algoritmos de otimização de código



Principais Fontes de Otimização

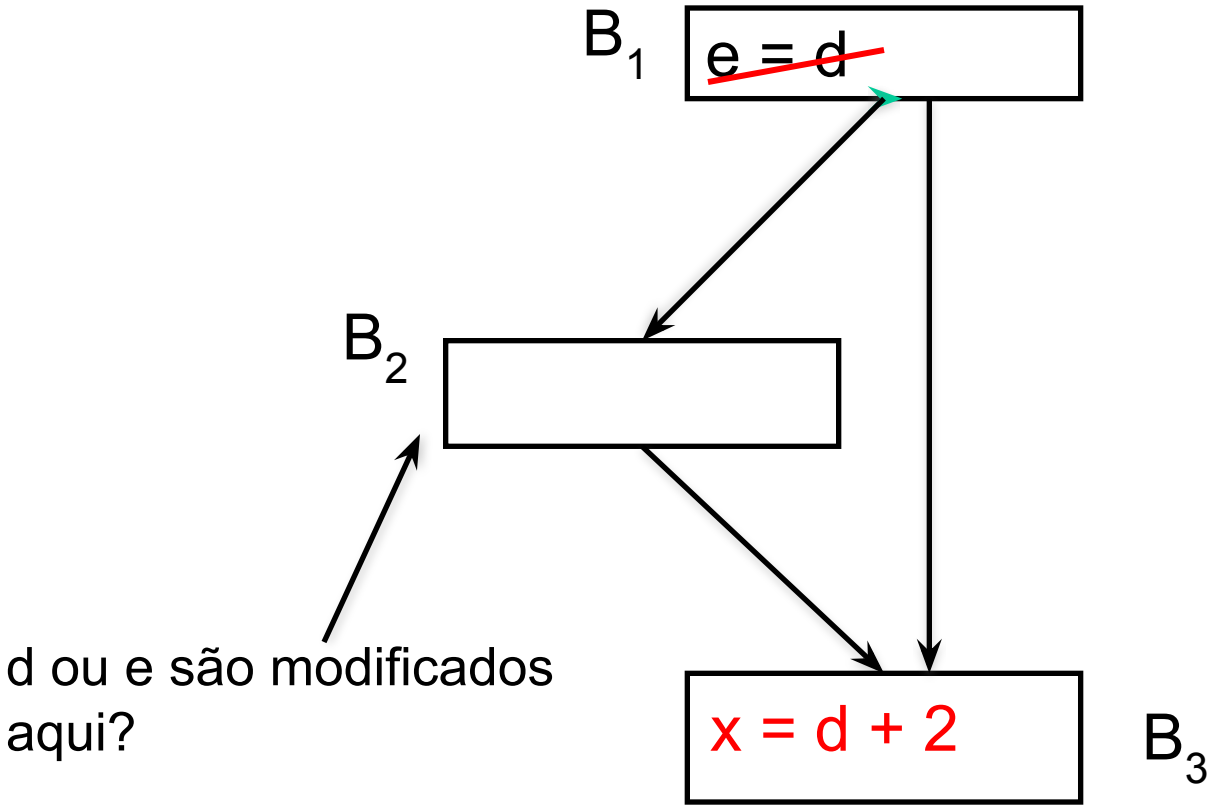
- Transformações que preservam a funcionalidade
  - Eliminação de Sub-expressões comuns (CSE)
  - Propagação de Cópias
  - Eliminação de código morto
  - Constant folding
- Transformações Locais
  - Dentro de um bloco básico
- Transformações Globais
  - Envolve mais de um bloco básico

Primeiros Exemplos de Otimização

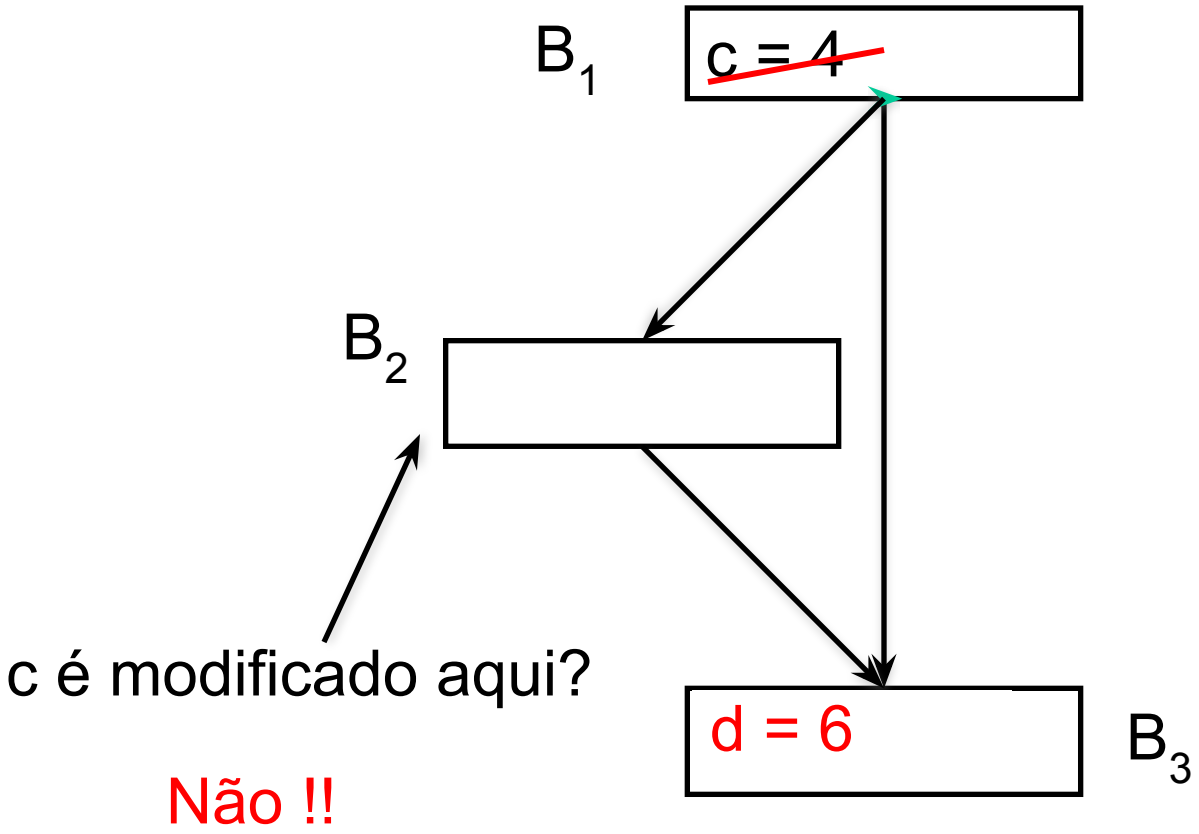
Exemplos de Otimização

- Otimizações de código
  - Usar as informações coletadas pelas análises
  - Tornar o código mais eficiente
- Vamos começar olhando:
  - Copy Propagation
  - Dead Code Elimination
  - Common Sub-expression Elimination (CSE)

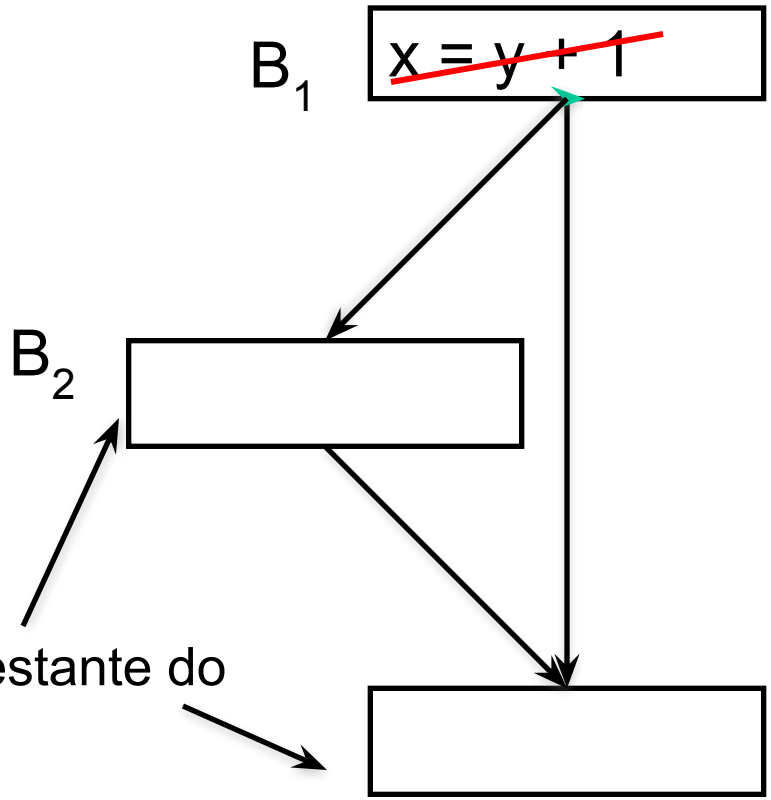
Copy Propagation



Constant Propagation



Dead Code Elimination

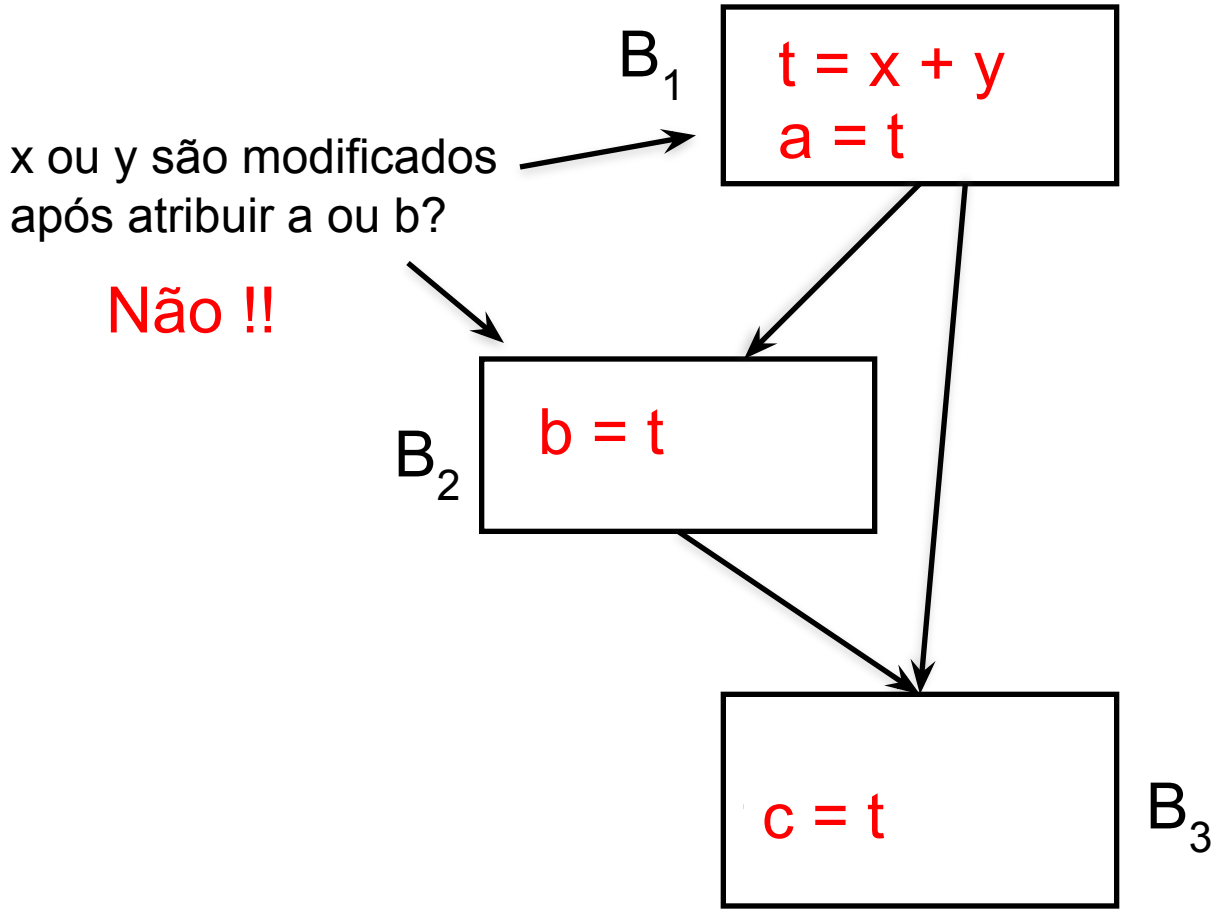


x é usado no restante do programa?

Não !!

Notem que **código morto** pode ser **gerado** pelas próprias **otimizações** do compilador

Common Subexpression Elimination



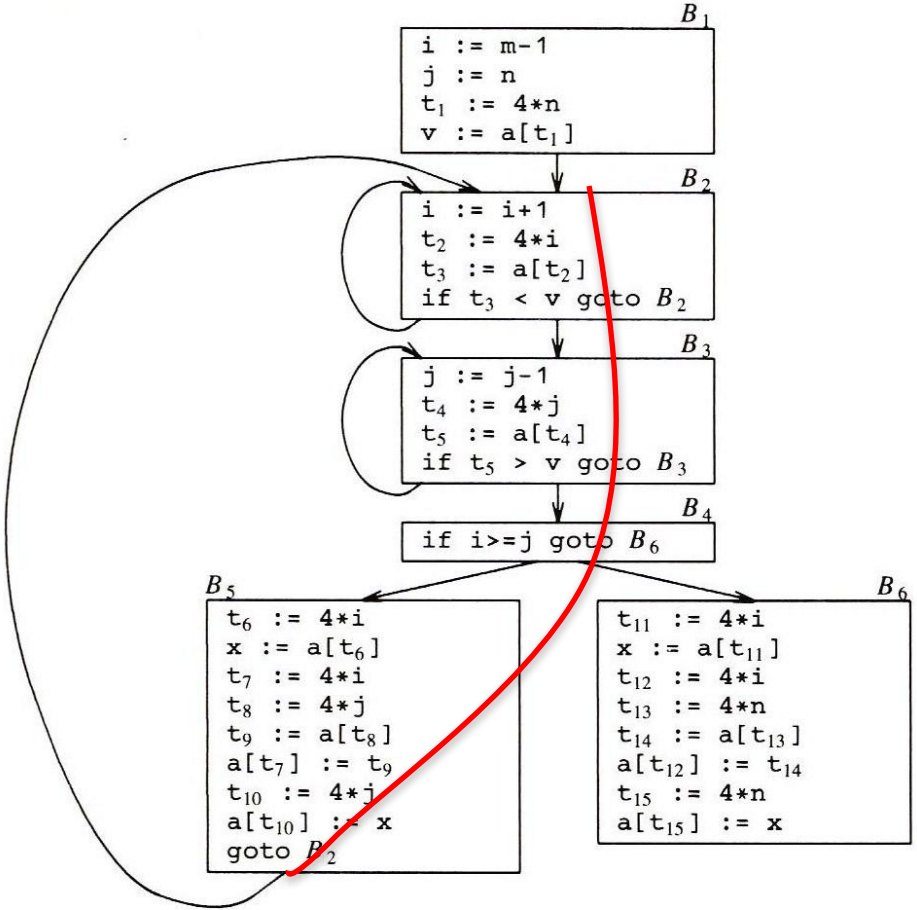
Notem que o **código resultante** pode ser **otimizado** com **propagação de cópia**

Um Exemplo em Código Real

Quick Sort

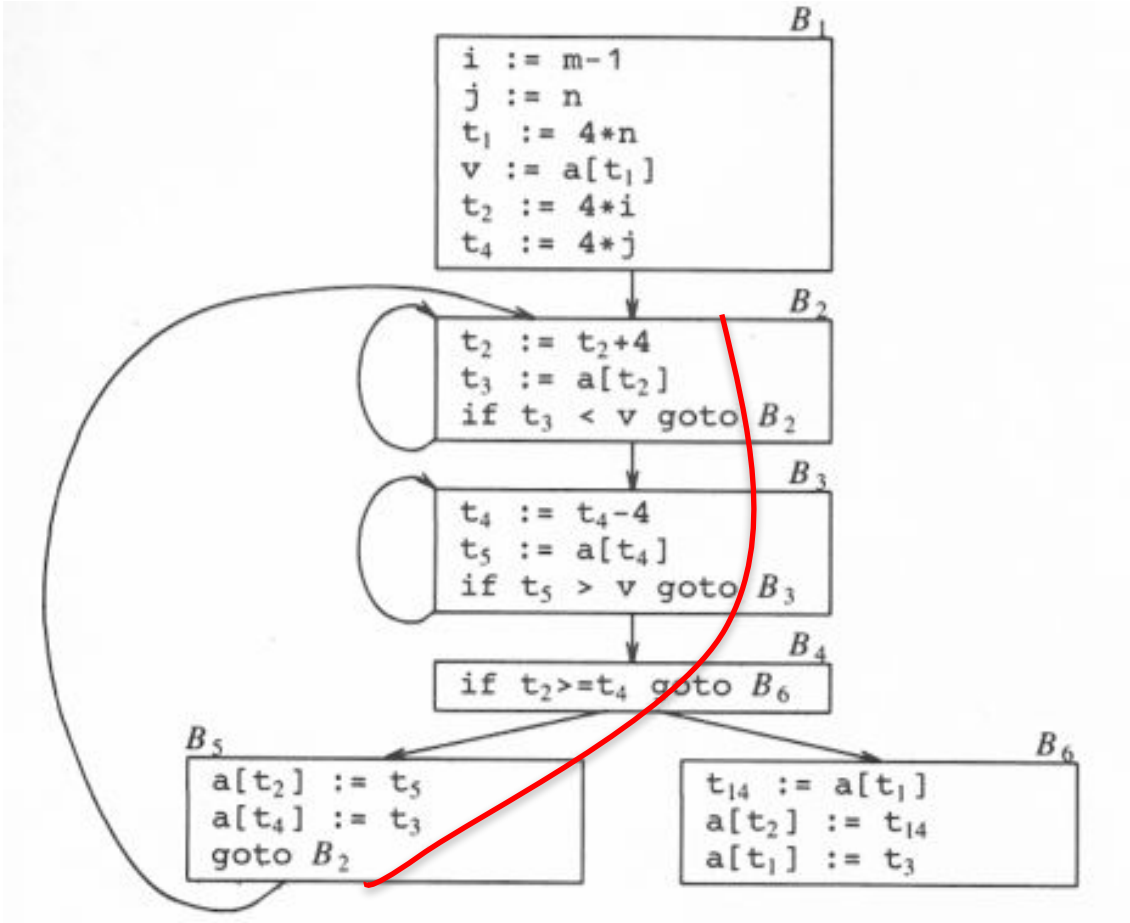

```
void quicksort(m,n)
int m,n;
{
     int i,j;
     int v,x;
     if ( n <= m ) return;
     /* fragment begins here */
     i = m-1; j = n; v = a[n];
     while(1) {
          do i = i+1; while ( a[i] < v );
          do j = j-1; while ( a[j] > v );
          if (i >= j ) break;
          x = a[i]; a[i] = a[j]; a[j] = x;
     }
     x = a[i]; a[i] = a[n]; a[n] = x;
     /* fragment ends here */
     quicksort(m,j); quicksort(i+1,n);
}
```

Quick Sort (original)



Código original
18 instruções

Quick Sort (otimizado)



Código otimizado
10 instruções

Aceleração* = 18/10
= 1.8x

*Sem contar o tempo maior de load/store

Como Fazer Otimizações?

Análise de Fluxo de Dados

- Essas transformações são feitas com base em informações coletadas do programa
- Esse é o trabalho de análise de fluxo de dados
  - Data-Flow Analysis (DFA)
- Otimização global intraprocedural
  - Interna a um procedimento ou uma função
  - Engloba todos os blocos básico

Idéia Básica

- Atravesse o grafo de fluxo do programa (CFG) coletando informações sobre a execução
- Conservativamente!
  - Ou seja, é verdade para qualquer caminho de execução do CFG
- Modifique o programa para torná-lo mais eficiente em algum aspecto
  - Desempenho (tempo de execução)
  - Tamanho do binário (importante por exemplo para sistemas embarcados)

Maioria das **análises** podem ser **descritas** através de **equações de fluxo de dados**

Análise de Fluxo de Dados

- Veremos análises baseadas no CFG de quádruplas (3AC)
  - a <- b op c é representada como (a, b, c, op)
- Veremos 3 análises
  - Reaching Definitions
  - Available Expressions
  - Liveness Analysis

Reaching Definitions

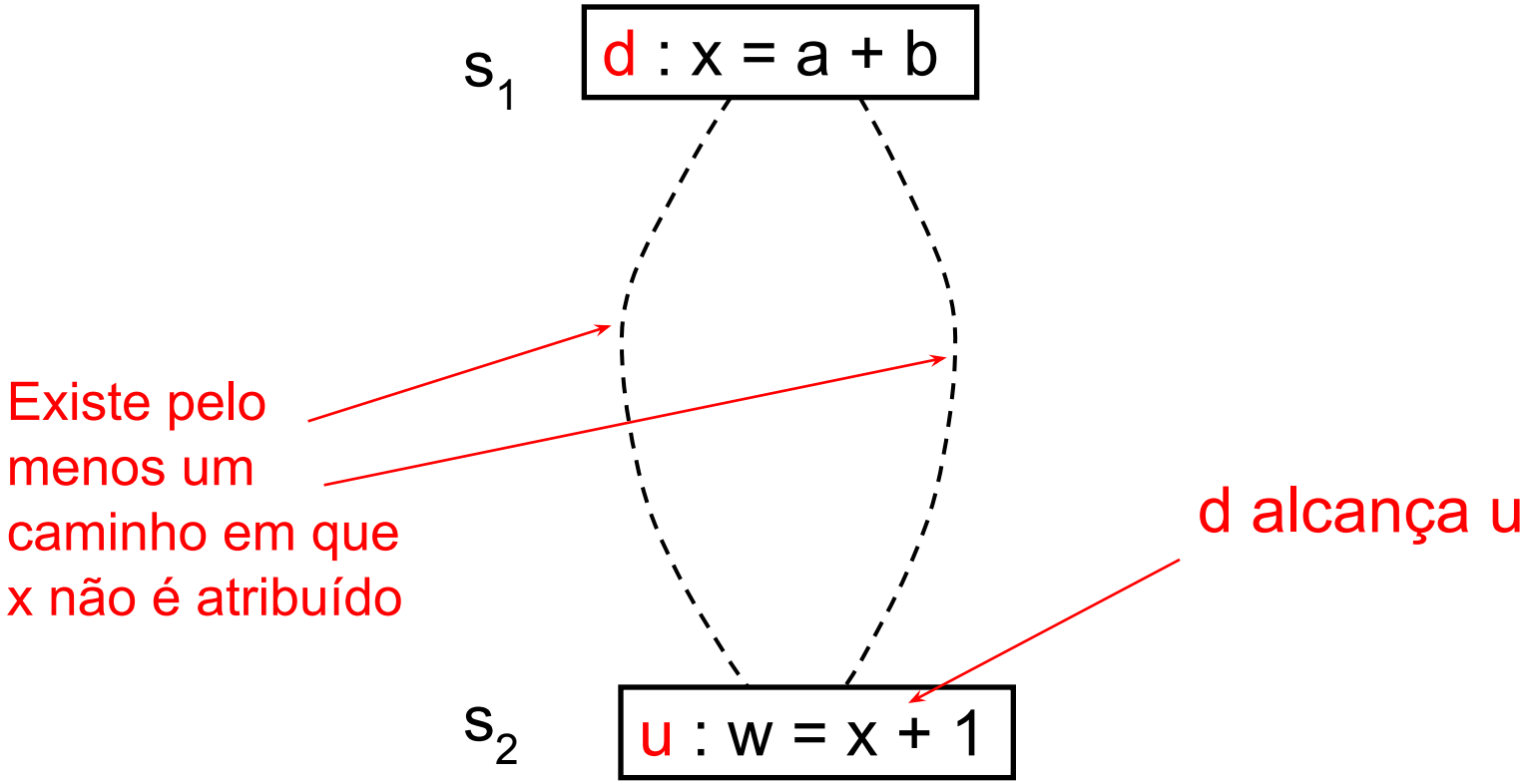
Uso Principal

- Dada uma variável x em um certo ponto do programa
- Inferimos que o valor de x é limitado a um determinado grupo de possibilidades

Definição

- Definição não ambígua de t
  - *d*: t ← a op b
  - *d*: t ← M[a]
- Definição ambígua
  - Uma sentença que pode ou não atribuir um valor a t
  - Exemplos:
    - ➢ Chamada de função
    - ➢ Atribuição a ponteiros
    - ➢ Instruções com predicados
- Vamos dizer que “*d* alcança uma sentença *u*”
  - Se existe pelo menos um caminho no CFG de d para u
  - Esse caminho não contém outra definição não ambígua de t

Fluxo de Execução



Notação

- Criamos IDs para as definições
  - d1: t ← x op y
    - **Gera** d1
    - **Mata** todas as outras definições de t, pois não alcançam o final desta instrução
- defs(t): conjunto de todas as definições de t

Conjuntos Gen e Kill

Table 17.2: Gen and kill for reaching definitions.

Statement *s*	*gen*[s]	*kill*[s]
$d : t \leftarrow b \ominus c$	{*d*}	$defs(t) - \{d\}$
$d : t \leftarrow M[b]$	{*d*}	$defs(t) - \{d\}$
$M[a] \leftarrow b$	{}	{}
if *a* relop *b* goto L_1 else goto L_2	{}	{}
goto *L*	{}	{}
L :	{}	{}
$f(a_1, \ldots, a_n)$	{}	{}
$d : t \leftarrow f(a_1, \ldots, a_n)$	{*d*}	$defs(t) - \{d\}$

Equações de Fluxo de Dados

In e Out inicializados como vazios.

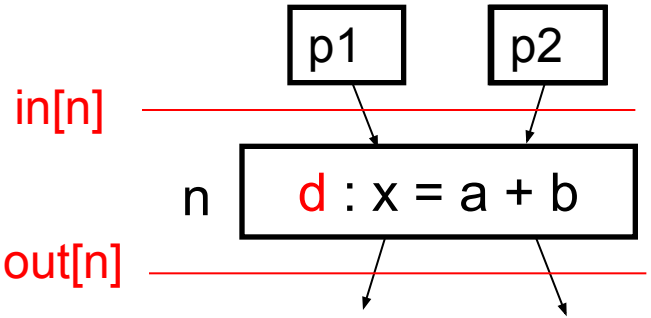
- Usando gen e kill computamos:
  - In[n]: conjunto de definições que alcançam o início de n
  - Out[n]: conjunto de definições que alcançam o final de n

$$in[n] = \bigcup_{p \in pred[n]} out[p]$$

$$out[n] = gen[n] \cup (in[n] - kill[n])$$

{d}

defs(x) – {d}



Reaching Definitions


```
1 :    a ← 5
2 :    c ← 1
3 : L1 : if c > a goto L2
4 :    c ← c + c
5 :    goto L1
6 : L2 : a ← c - a
7 :    c ← 0
```

$$in[n] = \bigcup_{p \in pred[n]} out[p]$$

$$out[n] = gen[n] \cup (in[n] - kill[n])$$

Iter. 1

n	$gen[n]$	$kill[n]$	$in[n]$	$out[n]$
1	1	6		
2	2	4,7		
3				
4	4	2,7		
5				
6	6	1		
7	7	2,4		

1 : a ← 5

2 : c ← 1

3 : L1 : if c > a goto L2

4 : c ← c + c

5 : goto L1

6 : L2 : a ← c - a

7 : c ← 0

$$in[n] = \bigcup_{p \in pred[n]} out[p]$$

$$out[n] = gen[n] \cup (in[n] - kill[n])$$

			Iter. 1		Iter. 2		Iter. 3	
n	$gen[n]$	$kill[n]$	$in[n]$	$out[n]$	$in[n]$	$out[n]$	$in[n]$	$out[n]$
1	1	6		1				
2	2	4,7	1	1,2				
3			1,2	1,2				
4	4	2,7	1,2	1,4				
5			1,4	1,4				
6	6	1	1,2	2,6				
7	7	2,4	2,6	6,7				

Algoritmo

$W \leftarrow$ the set of all nodes
while W is not empty
    remove a node n from W
    $old \leftarrow out[n]$
    $in \leftarrow \bigcup_{p \in pred[n]} out[p]$
    $out[n] \leftarrow gen[n] \cup (in - kill[n])$
    if $old \neq out[n]$
        for each successor s of n
            if $s \notin W$
                put s into W

Observações

- O algoritmo propaga as definições
  - Até onde elas podem chegar sem serem mortas
  - Até que não haja mais modificações (todas as restrições são satisfeitas)
- O algoritmo sempre termina:
  - OUT[B] nunca diminui de tamanho
  - O número de definições é finito
  - Se OUT não muda, IN não muda nó próximo passo
- Limitante superior para número de iterações
  - Número de nós no CFG
  - Pode ser melhorado de acordo com a ordem de avaliação dos nós

Use-Definition Chain

ud-chain

- Forma de armazenar reaching definitions
  - para acesso mais rápido
- A cada sentença s: t := x op y
  - armazena-se para cada variável x e y
  - uma lista das sentenças que definem x e y e alcançam s.

Exemplo

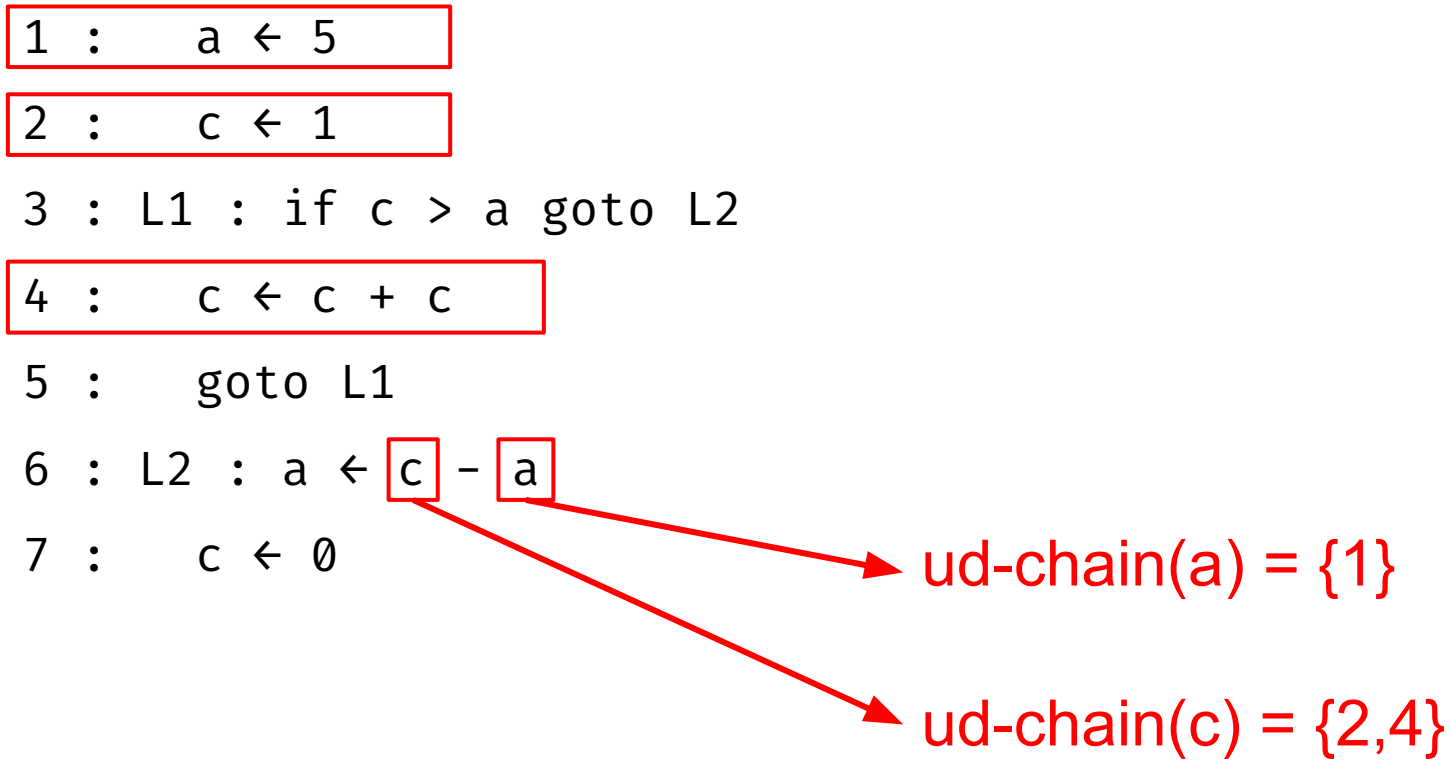

```
1 :    a ← 5
2 :    c ← 1
3 : L1 : if c > a goto L2
4 :    c ← c + c
5 :    goto L1
6 : L2 : a ← c - a
7 :    c ← 0
```

1,2,4

Iter. 3

n	*in*[*n*]	*out*[*n*]
1		1
2	1	1,2
3	1,2,4	1,2,4
4	1,2,4	1,4
5	1,4	1,4
6	1,2,4	2,4,6
7	2,4,6	6,7

Exemplo


1 : a ← 5
2 : c ← 1
3 : L1 : if c > a goto L2
4 : c ← c + c
5 : goto L1
6 : L2 : a ← c - a
7 : c ← 0
ud-chain(a) = {1}
ud-chain(c) = {2,4}

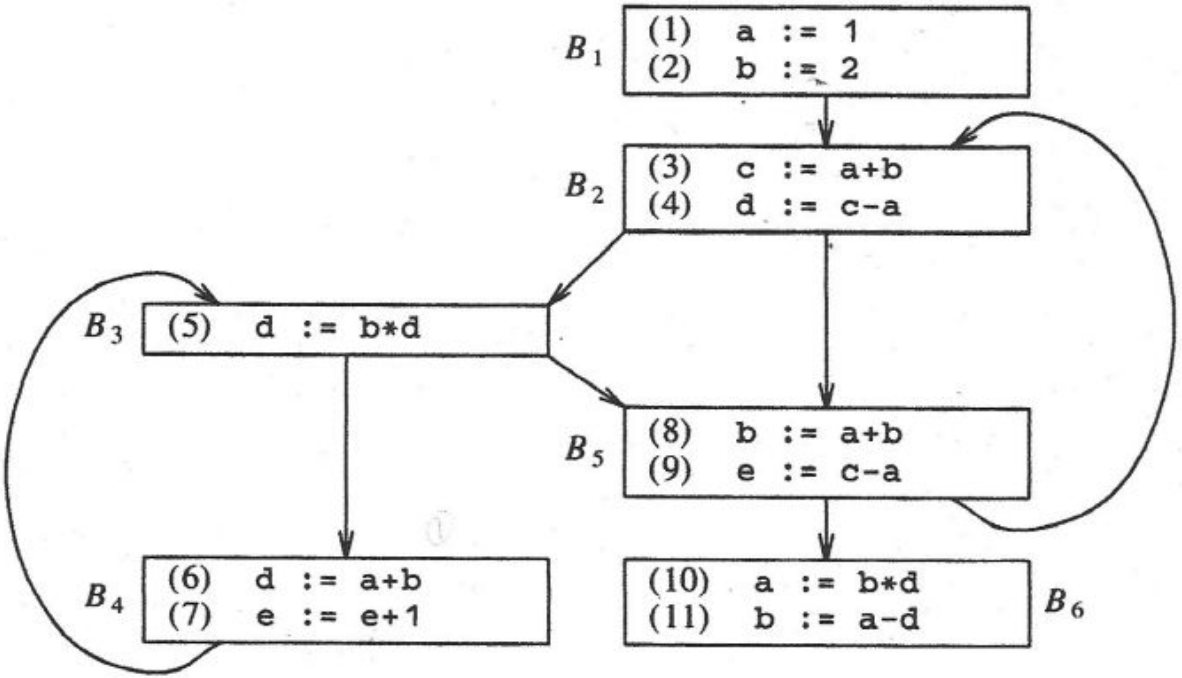
Exercício

- Cria o CFG
- Calcule o Reaching Definition
- Cria a UD-chain

```
(1)     x : =1;
(2)     y := 1;
(3)     if z <> 0
(4)       then x := 2
(5)       else y := 2;
(6)     w := x + y;
```

Exercício

- Calcule o Reaching Definition
- Cria a UD-chain



Otimizações com UD-chain

Como propagar constantes usando essa análise?

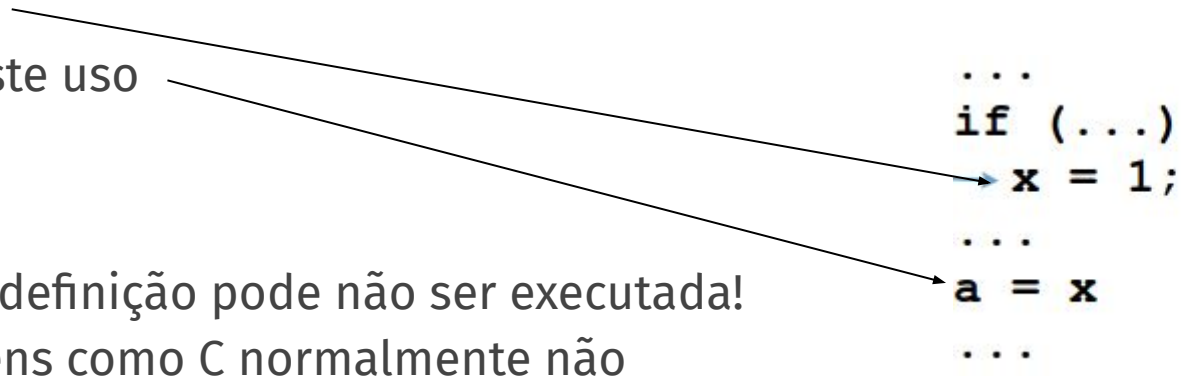
Constant propagation

- Para um uso da variável v na instrução n,
  - n: x = ... v ...
- Se as definições de v que chegam a n são todas da forma
  - d: v = c [c uma constante]
- Então substitua o uso de v em n por c

Pode também simplificar branch if(true)

Variáveis não inicializadas?

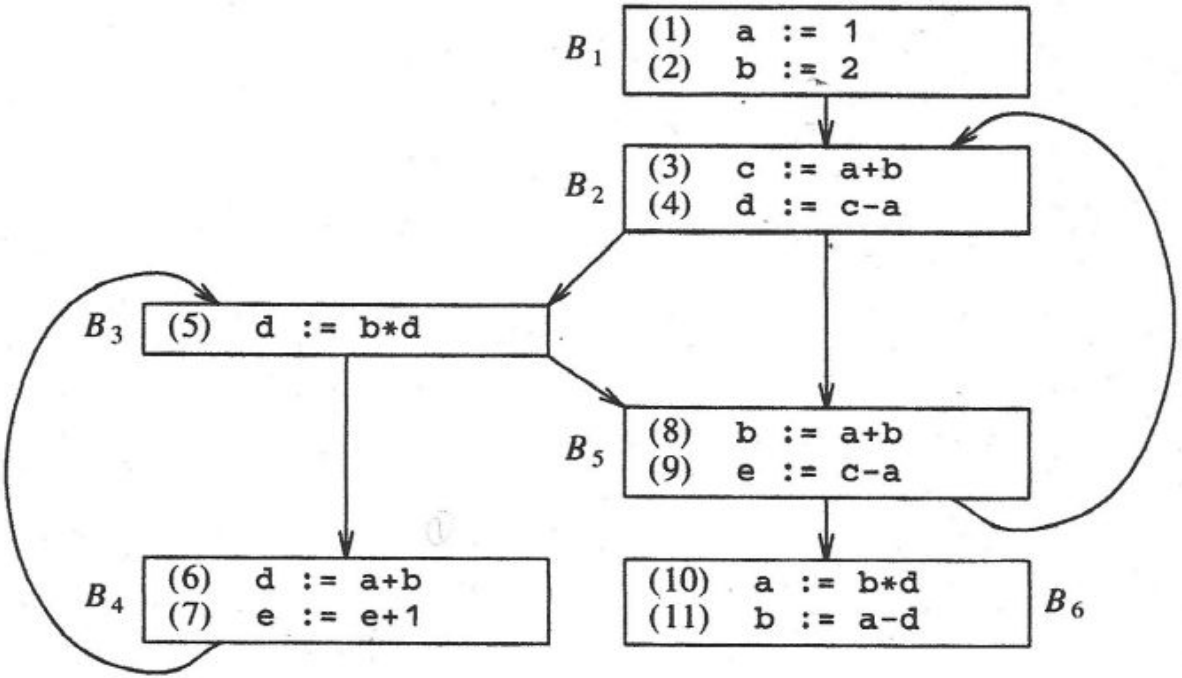
- Esta def
- atinge este uso



- ... mas a definição pode não ser executada!
- Linguagens como C normalmente não definem o comportamento de programas com variáveis não inicializadas
  - então propagação constante simples é válida

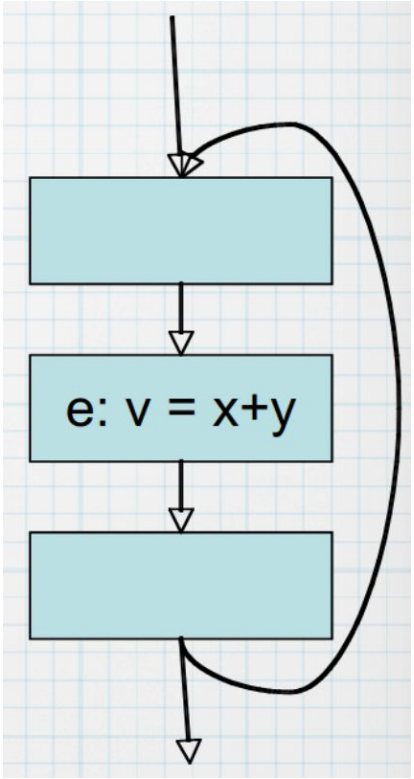
Exercício

- Usando a UD-chain criada previamente
- Qual constante pode ser propagada?



Outras Aplicações

- Há várias aplicações importantes da análise Reaching Definitions
- Considere um loop contendo uma expr e
  - se todas as definições do operandos de e estão fora do loop
  - então e pode ser movido para fora do loop
- Esta otimização é chamada de *loop-invariant code motion*



Available Expressions

Expressões Disponíveis (Available Expressions)

- Expressão disponível:
  - x+y está disponível em p se:
    - todo caminho do nó inicial até p calcula x+y
    - após a última computação de x+y, nem x nem y sofrem atribuições
- Kill:
  - Um bloco B mata, ou pode matar, x+y se ele atribui a x e/ou y, e não recomputa x+y
- Gen:
  - Um bloco B gera x+y se ele certamente computa x+y, e não redefine x ou y.

Expressões Disponíveis (Available Expressions)

Instruções	Expressões disponíveis
a = b + c	
b = a - d	
c = b + c	
d = a - d	

STATEMENTS	AVAILABLE EXPRESSIONS
	..... none
a := b+c	
	..... only b+c
b := a-d	
	..... only a-d
c := b+c	
	..... only a-d
d := a-d	
	..... none

Fig. 10.30. Computation of available expressions.

Expressões Disponíveis (Available Expressions)

- Computamos gen e kill para cada B como visto anteriormente
- Temos:

$$out[B] = gen[B] \bigcup (in[B] - kill[B])$$

$$in[B] = \bigcap_{P \in predecessores\ de\ B} out[P]$$

Expressões Disponíveis (Available Expressions)

- O *in* do nó inicial é sempre vazio
  - Nada está disponível antes do início do programa
- O operador de confluência é intersecção
  - Tem que vir por todos os caminhos
- Estimativa inicial é muito grande
  - Intersecção vai diminuindo os conjuntos a chegar ao maior ponto fixo

Expressões Disponíveis (Available Expressions)

Algorithm 10.3. Available expressions.

Input. A flow graph G with $e\_kill[B]$ and $e\_gen[B]$ computed for each block B. The initial block is B_1.

Output. The set $in[B]$ for each block B.

Method. Execute the algorithm of Fig. 10.32. The explanation of the steps is similar to that for Fig. 10.26. □

```
in[B1] := ∅;
out[B1] := e_gen[B1];  /* in and out never change for the initial node, B1 */
for B ≠ B1 do out[B] := U − e_kill[B];  /* initial estimate is too large */
change := true;
while change do begin
        change := false;
        for B ≠ B1 do begin
                in[B] := ⋂ out[P];
                         P a predecessor of B
                oldout := out[B];
                out[B] := e_gen[B] ∪ (in[B] − e_kill[B]);
                if out[B] ≠ oldout then change := true
        end
end
```

Fig. 10.32. Available expressions computation.

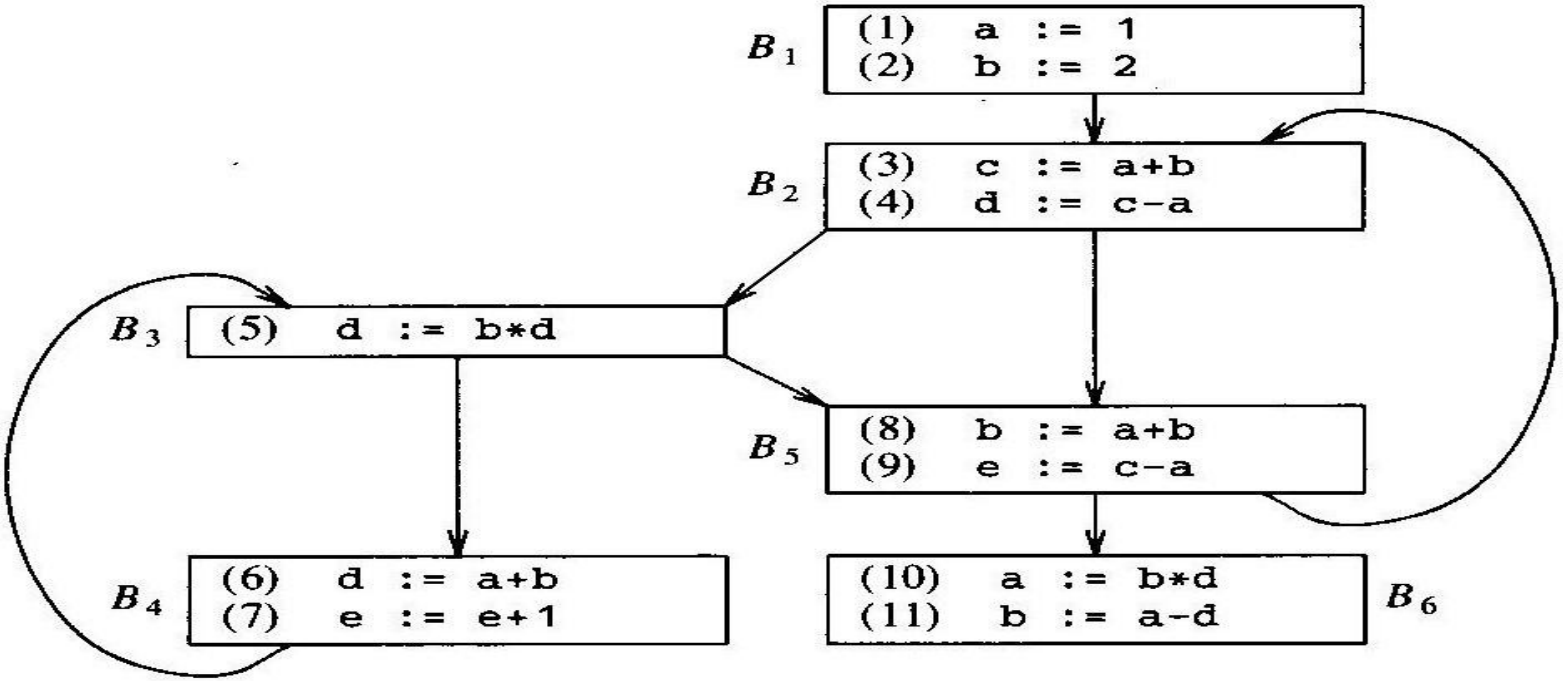


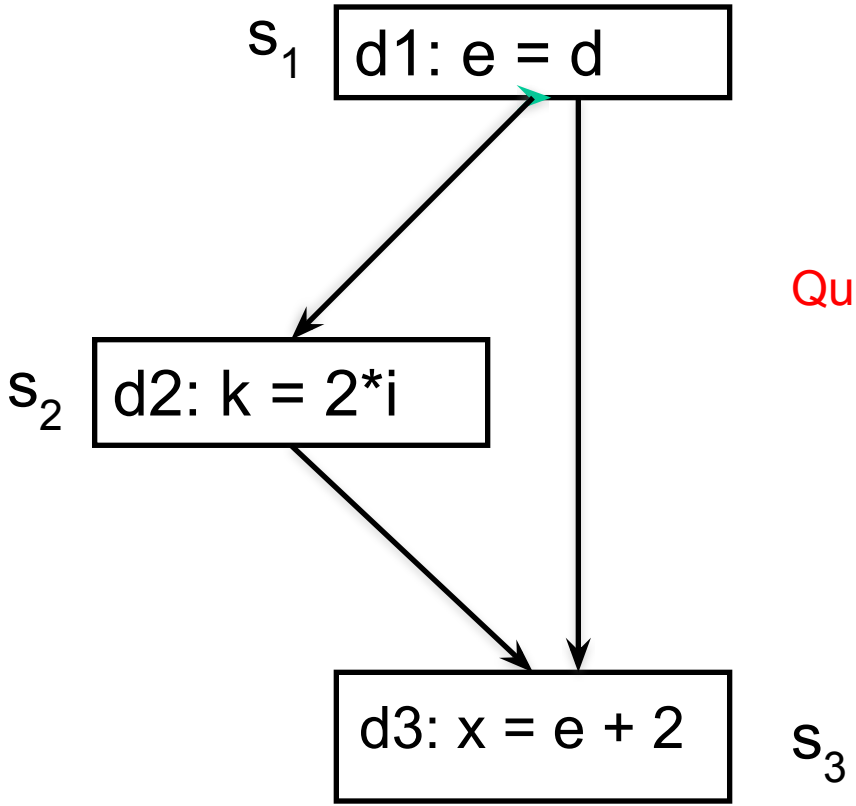
Fig. 10.74. Flow graph.

Copy Propagation

Copy Propagation

- Elimina cópias desnecessárias
- Seja d: t ← z
- Seja n: y ← t op x
- Quando t será uma cópia em n?
  - Neste caso, podemos reescrever n da forma
    - n: y ← z op x

Copy Propagation



Que análise é necessária?

Available Copies Analysis

- Usando gen e kill computamos
  - In[n]: conjunto de cópias disponíveis no início de n
  - Out[n]: conjunto de cópias disponíveis no final de n

$$in[n] = \bigcap_{p \in pred[n]} out[p]$$

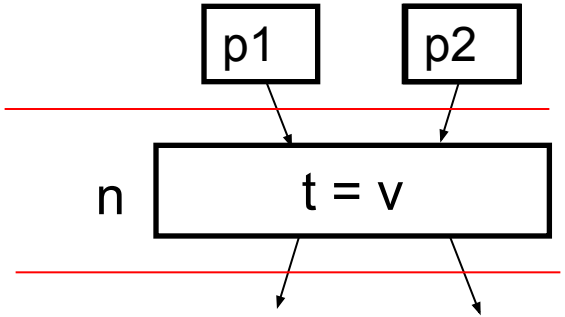
$$out[n] = gen[n] \cup (in[n] - kill[n])$$

{t = v}

copies(t)

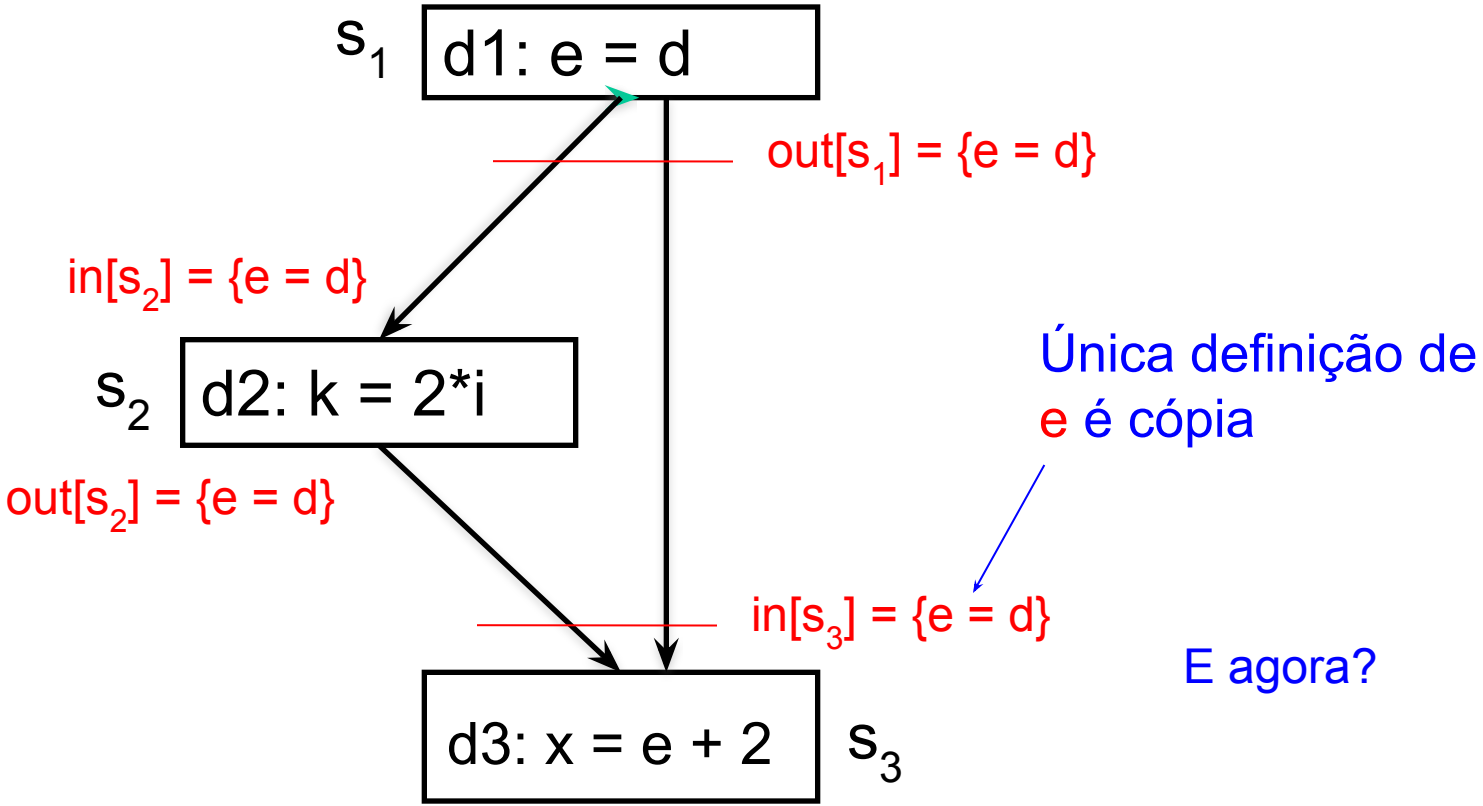
in[n]

out[n]

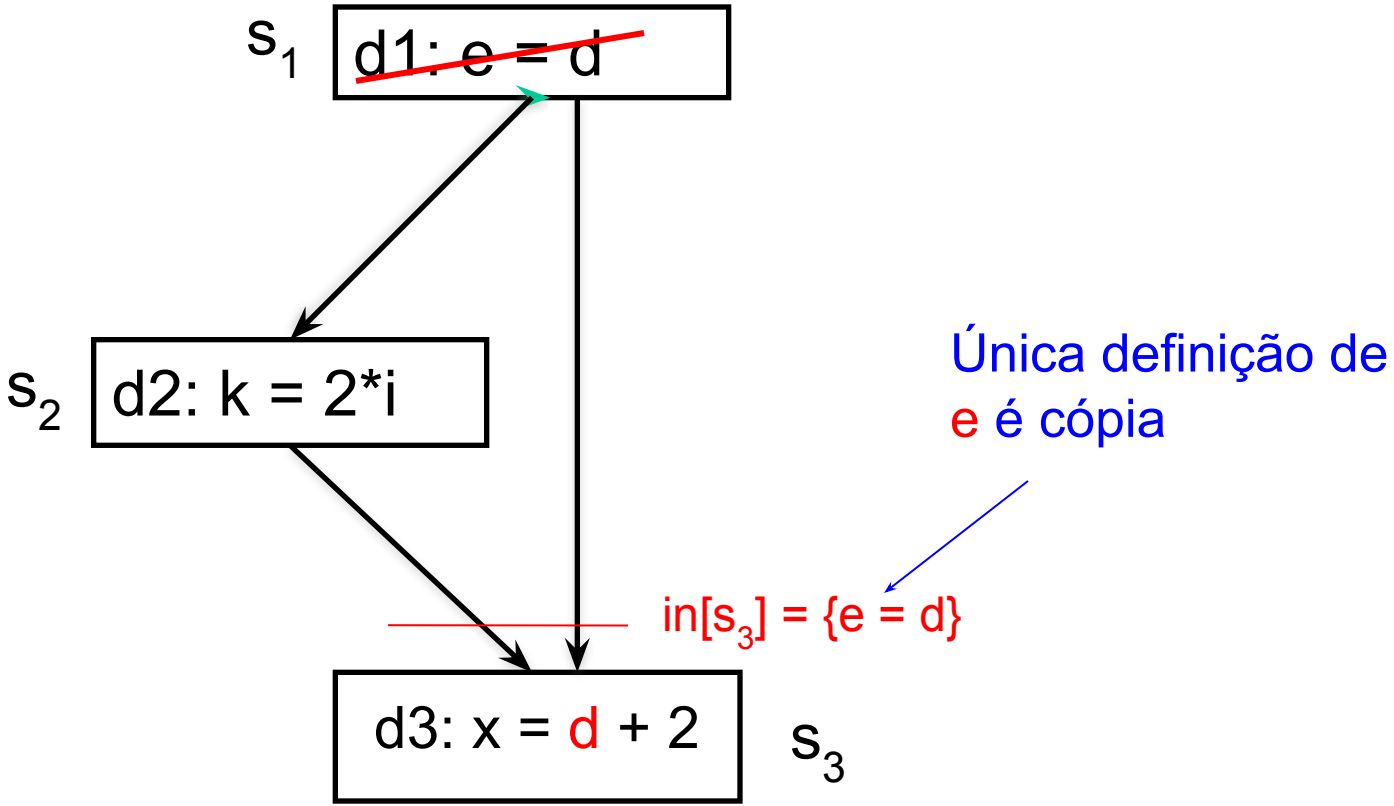


  - in e out inicializados com “cheio”, exceto in[START]
  - copies(t) são todas as cópias do tipo t = x, x = t
  - O mesmo kill vale para outras sentenças n em que t ou v são atribuídos

Exemplo


s_1
d1: e = d
out[s_1] = {e = d}
in[s_2] = {e = d}
s_2
d2: k = 2*i
out[s_2] = {e = d}
Única definição de
e é cópia
in[s_3] = {e = d}
E agora?
d3: x = e + 2
s_3

Exemplo



Constant Folding

Constant Folding

- Seja d: t ← c (constante)
- Seja n: y ← t op x
- Quando t será constante em n?
  - Neste caso, podemos reescrever n da forma
    - n: y ← c op x

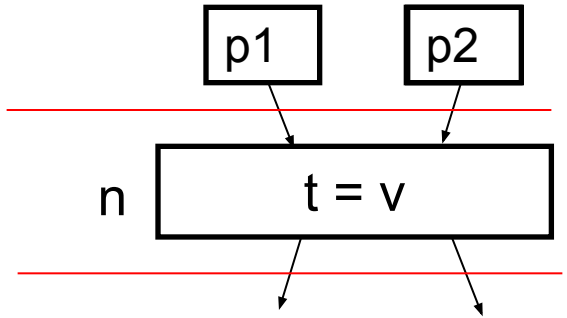
Variation of Available Copies Analysis

- Usando gen e kill computamos
  - In[n]: conjunto de cópias constantes disponíveis no início de n
  - Out[n]: conjunto de cópias constantes disponíveis no final de n

$$in[n] = \bigcap_{p \in pred[n]} out[p]$$

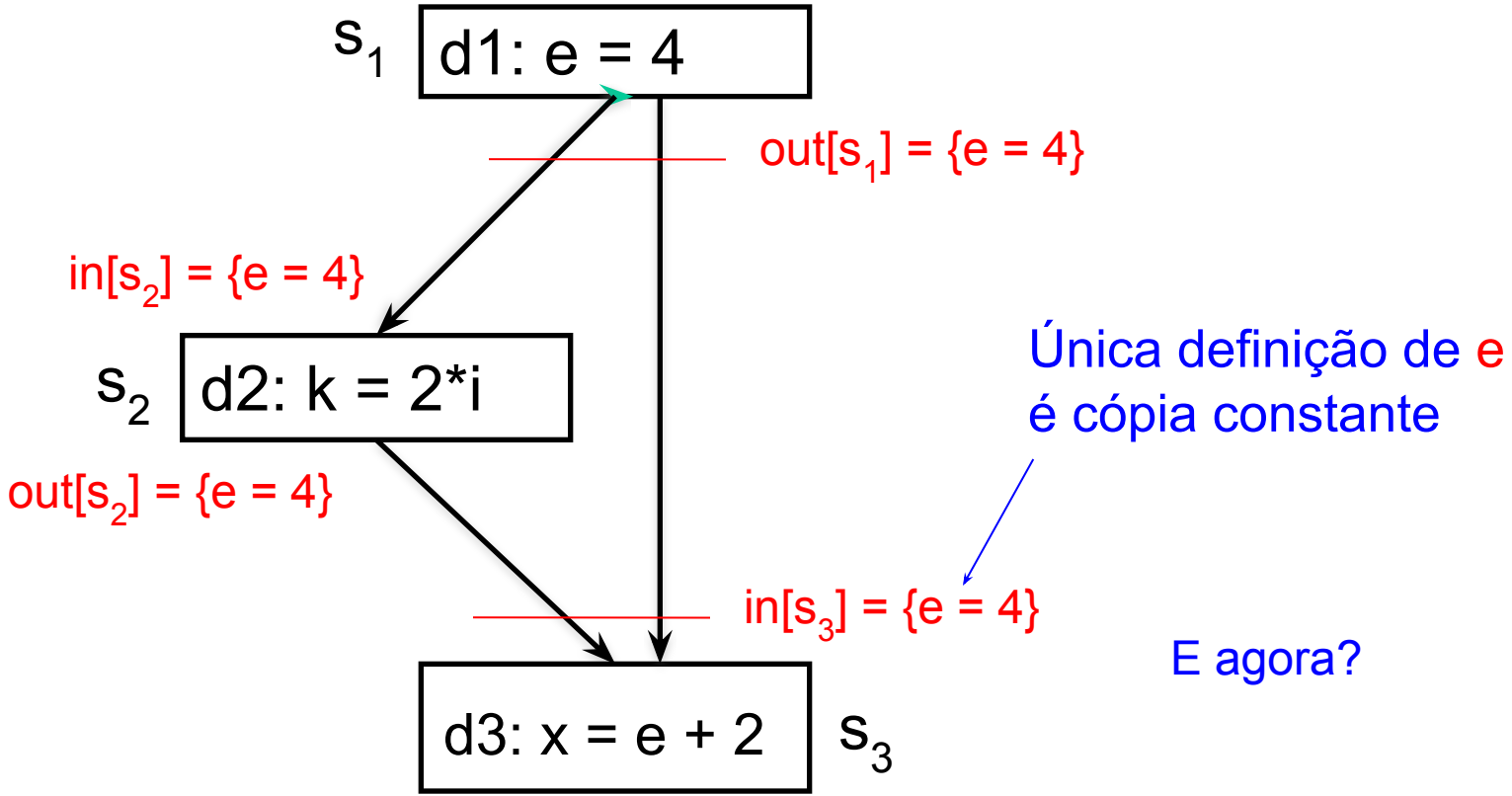
$$out[n] = gen[n] \cup (in[n] - kill[n])$$

{t = v} const(t)

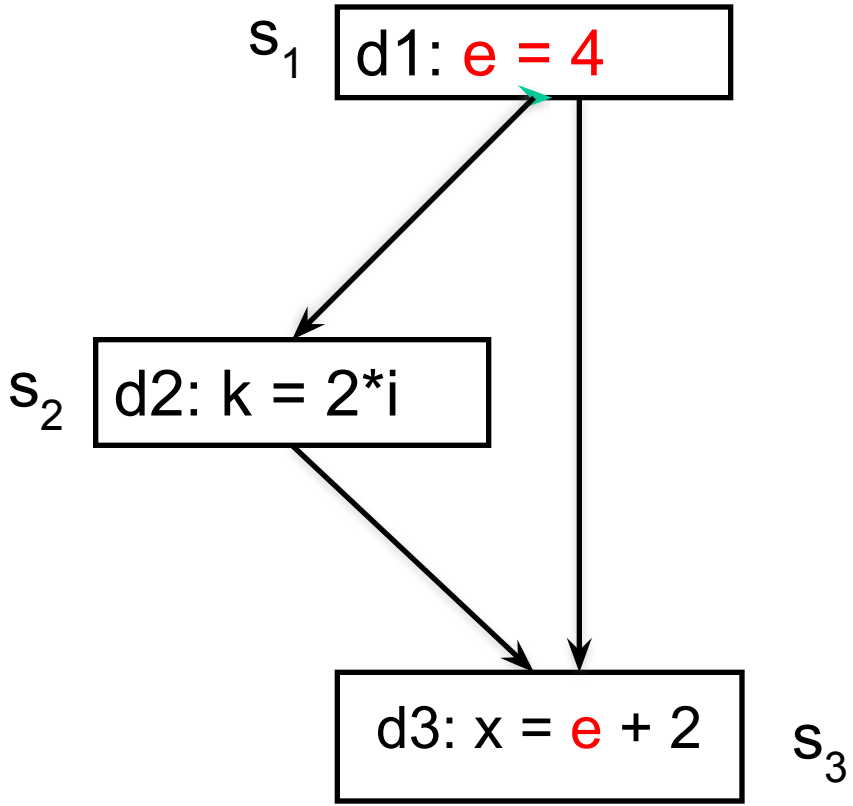


  - in e out inicializados com “cheio”, exceto in[START]
  - const(t) são todas as cópias do tipo t = k, onde k é constante
  - O mesmo kill vale para sentenças n em que t for atribuído (ex. t = x op y)

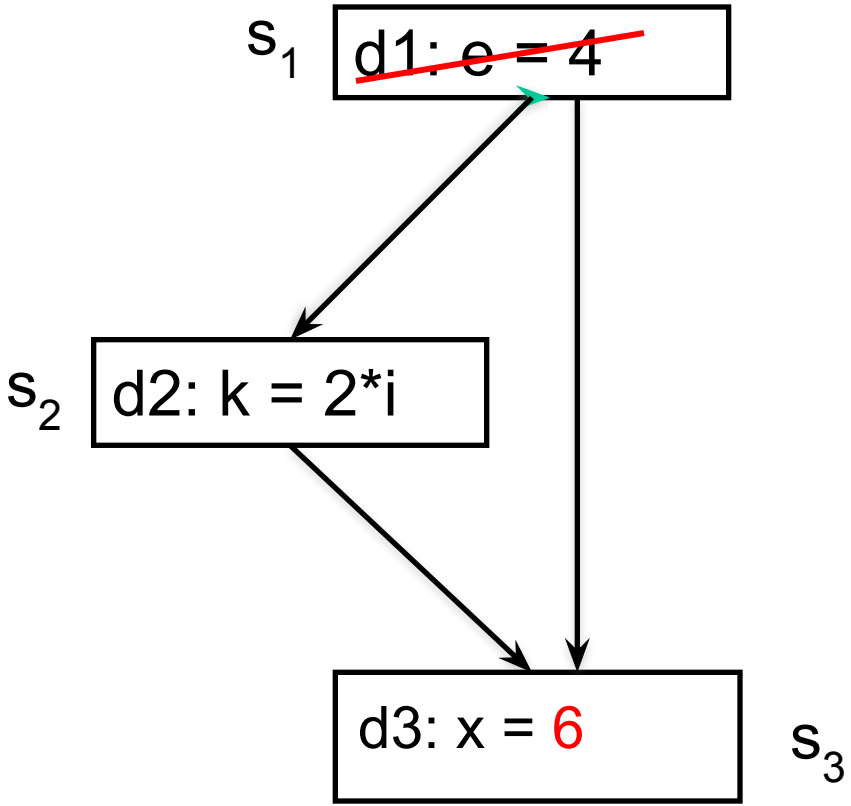
Variation of Available Copies Analysis



Constant Folding



Constant Folding

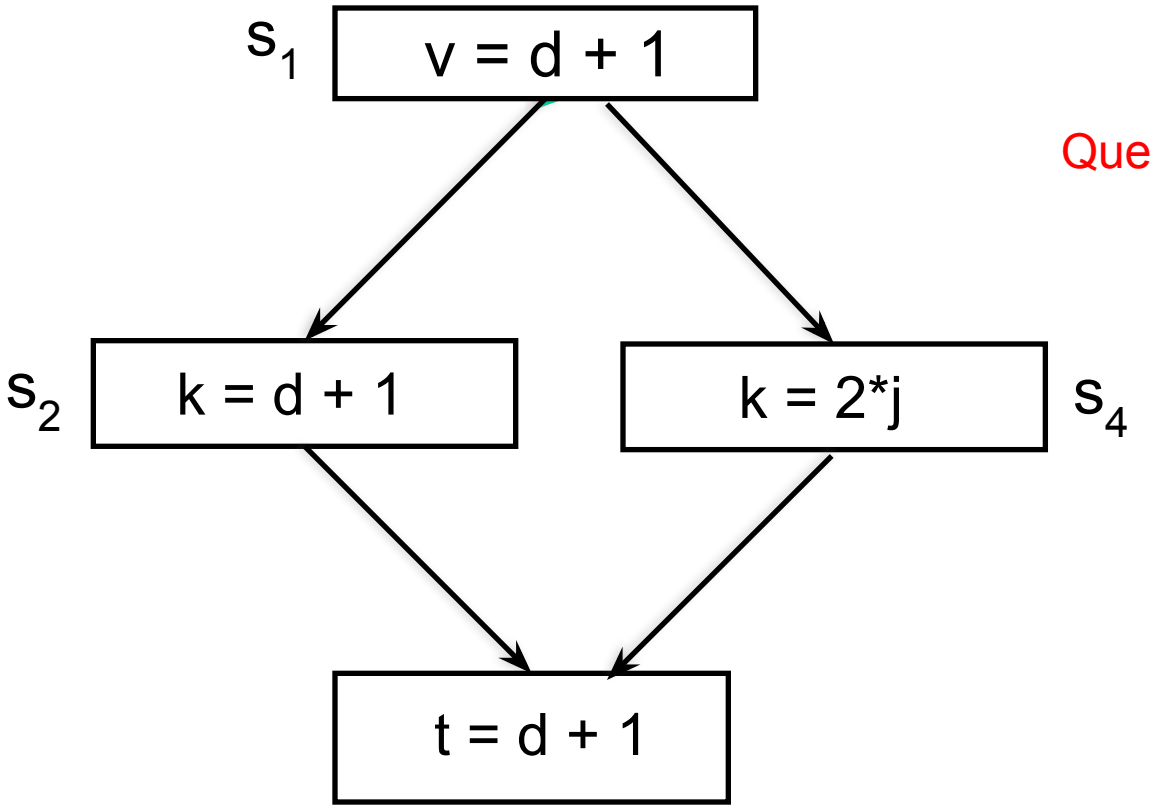


Common Subexpression Elimination

Common-subexpression Elimination

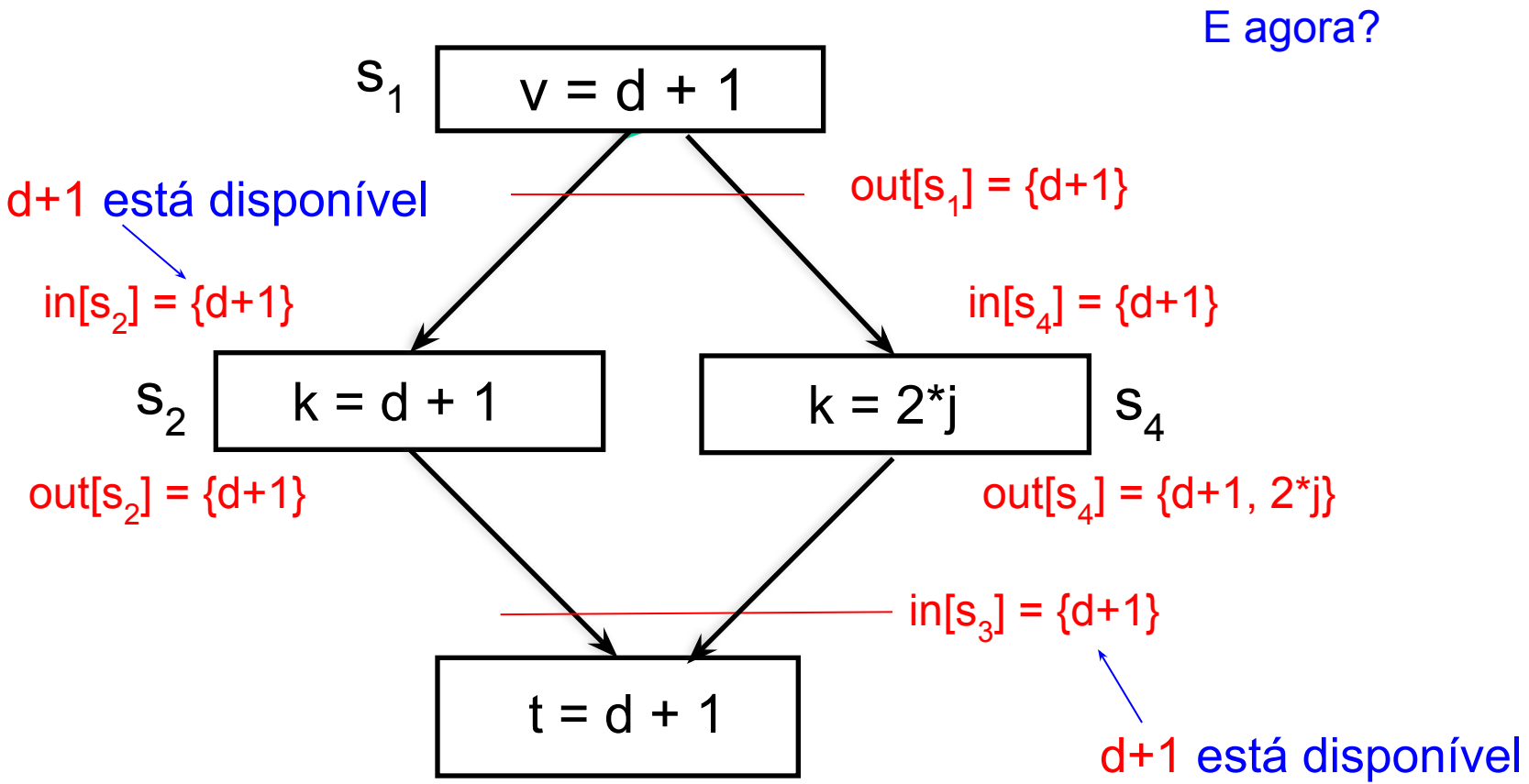
- Seja s: t ← x op y
- Se x op y está disponível em s
  - Elimine o cálculo de x op y de s
- Algoritmo
  - Usa informação das expressões disponíveis em s
  - Compute available expressions, encontrando expressões da forma x op y em sentenças n: v ← x op y que alcançam s
  - Em n crie um novo temporário w e reescreva n da forma
    - ➢ n: w ← x op y
    - ➢ n': v ← w
  - Modifique s para:
    - ➢ s: t ← w

Common Subexpression Elimination

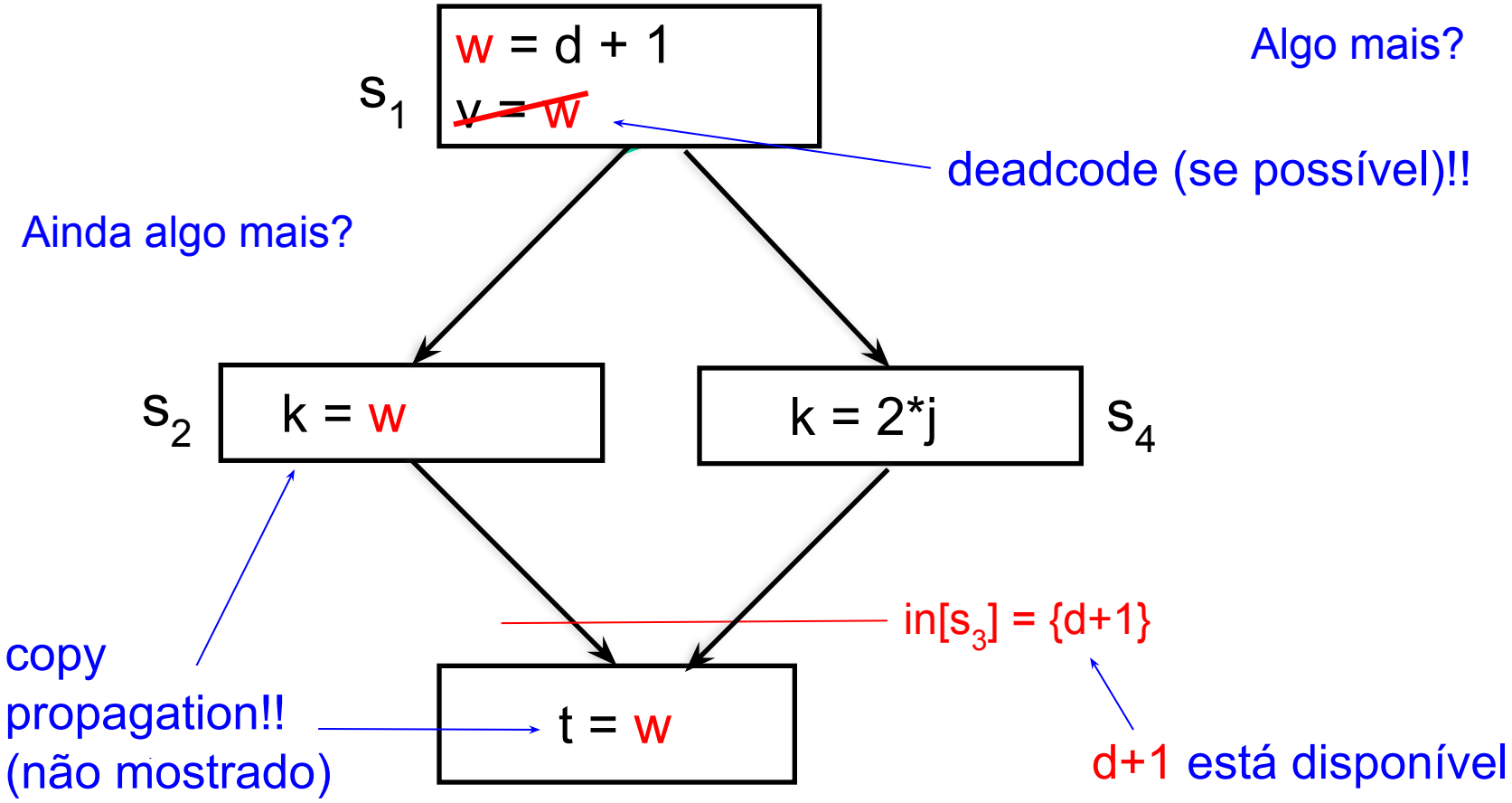


Que análise é necessária?

Using Available Expressions Analysis



Common Sub-expression Elimination



Liveness Analysis

Introdução

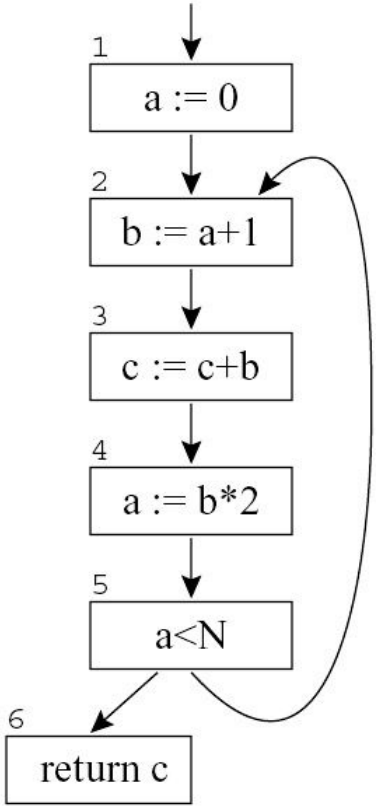
- Linguagem intermediária
  - Gerada pelo front-end considerando número infinito de registradores para temporários
- Máquinas reais têm finitos registradores
  - Para máquinas RISC, 32 é um número típico
- Dois valores temporários podem ocupar o mesmo registrador se não estão “em uso” ao mesmo tempo
  - Muitos temporários podem caber em poucos registradores
  - Os que não couberem vão para a memória (spill)

Introdução

- O compilador analisa a IR para saber quais valores estão em uso ao mesmo tempo
- Chamamos de *viva* uma variável que pode vir a ser usada no futuro
- Esta tarefa então, é conhecida como *liveness analysis*

Control Flow Graph (CFG)

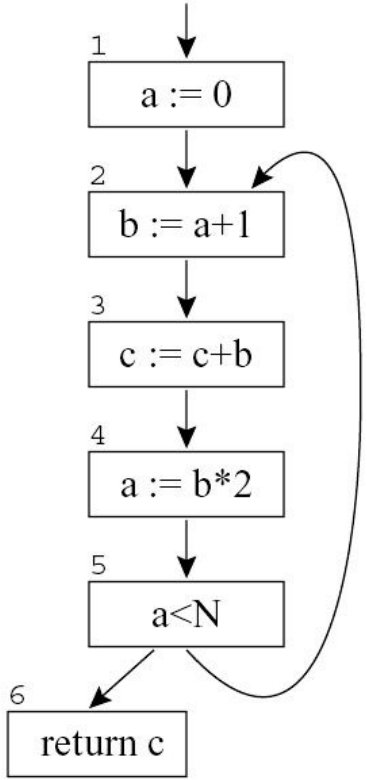
$$
\begin{aligned}
& a \leftarrow 0 \\
L_1 : \; & b \leftarrow a + 1 \\
& c \leftarrow c + b \\
& a \leftarrow b * 2 \\
& \text{if } a < N \text{ goto } L_1 \\
& \text{return } c
\end{aligned}
$$



Análise de Longevidade

- *b* é usada em 4
  - Precisa estar viva na aresta 3 → 4
- *b* não é definida no nó 3
  - Logo, deve estar viva na aresta 2 → 3
- *b* é definida em 2
  - Logo, *b* está morta na aresta 1 → 2
  - Seu valor nesse ponto não será mais útil a ninguém
- Live range de b:
  - {2 → 3 , 3 → 4}

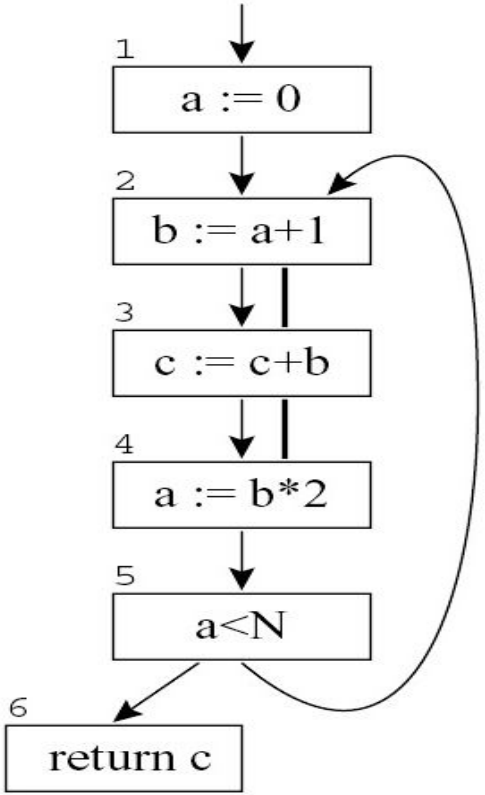
$$
\begin{aligned}
& a \leftarrow 0 \\
L_1 : \; & b \leftarrow a + 1 \\
& c \leftarrow c + b \\
& a \leftarrow b * 2 \\
& \text{if } a < N \text{ goto } L_1 \\
& \text{return } c
\end{aligned}
$$



Análise de Longevidade

b: {2 → 3 , 3 → 4,}

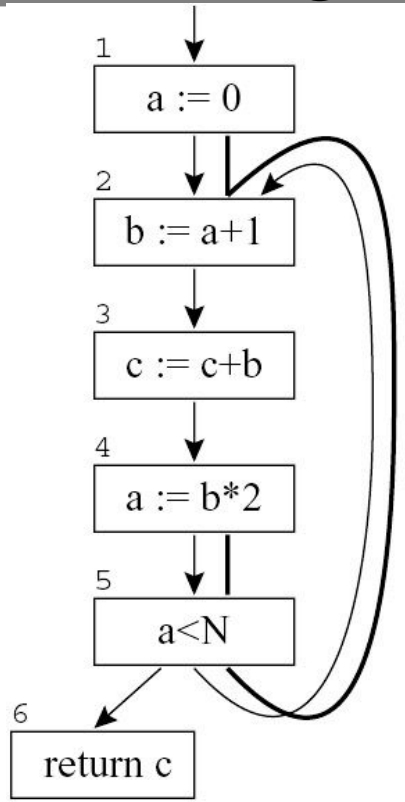
Como seria para a e c?



Live range

Análise de Longevidade

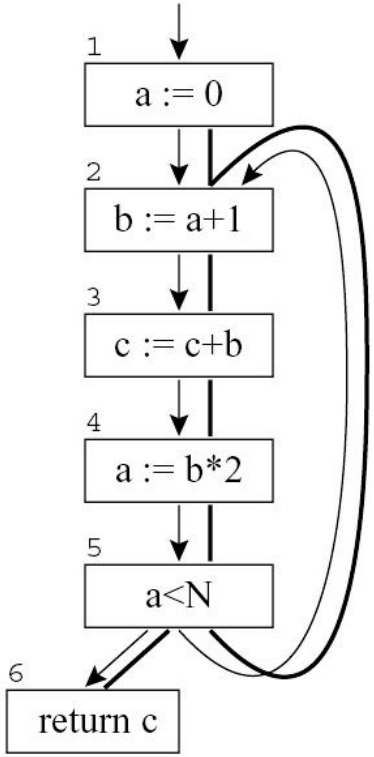
a: {1 → 2 , 4 → 5, 5 → 2}



Análise de Longevidade

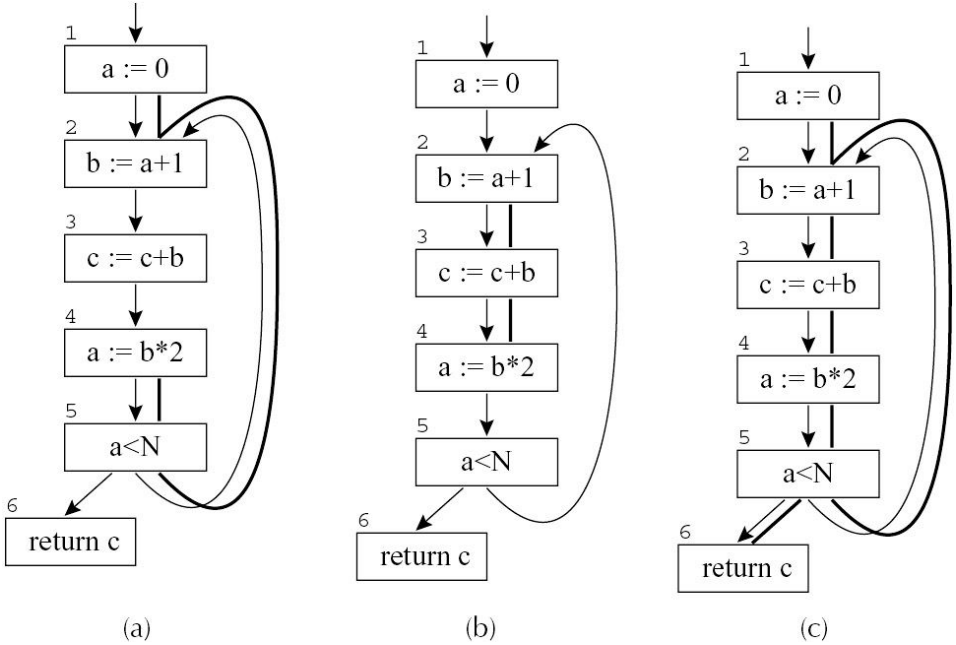
c: {1 → 2 , 2 → 3, 3 → 4,
4 → 5, 5 → 2, 5 → 6}

Alguma coisa especial sobre c?



Análise de Longevidade

- De quantos registradores preciso?



Análise de Longevidade

- É um exemplo de análise de fluxo de dados
- Terminologia:
  - Succ[n]: conjunto de nós sucessores a n
  - Pred[n]: conjunto de predecessores de n
  - Out-edges: saem para os sucessores
  - In-edges: chegam dos predecessores
  - Uma atribuição a uma variável define a mesma
  - Uma ocorrência do lado direito de uma instrução é um uso da variável

Análise de Longevidade

- Terminologia:
  - Def de uma variável é o conjunto de nós do grafo que a definem
  - Def de um nó é o conjunto de variáveis que ele define
  - Analogamente para use
- Longevidade:
  - Uma variável v está viva em uma aresta se existe uma caminho direcionado desta aresta até um uso de v, que não passa por alguma definição de v
  - Live-in: v é live-in em um nó n se v está viva em alguma in-edge de n
  - Live-out: v é live-out em n se v está viva em alguma out-edge de n

Computando Liveness

1. Se v está em *use*[*n*], então v é *live-in* em *n*.

2. Se v é *live-in* no nó *n*, então ela é *live-out* para todo *m* em *pred*[*n*].

3. Se v é *live-out* no nó *n*, e não está em *def* [*n*], então v é também *live-in* em *n*

Algoritmo

$$in[n] = use[n] \cup (out[n] - def[n])$$

$$out[n] = \bigcup_{s \in succ[n]} in[s]$$

for each n

$in[n] \leftarrow \{\}$; $out[n] \leftarrow \{\}$

repeat

for each n

$in'[n] \leftarrow in[n]$; $out'[n] \leftarrow out[n]$

$in[n] \leftarrow use[n] \cup (out[n] - def[n])$

$out[n] \leftarrow \bigcup_{s \in succ[n]} in[s]$

until $in'[n] = in[n]$ and $out'[n] = out[n]$ for all n

Algoritmo

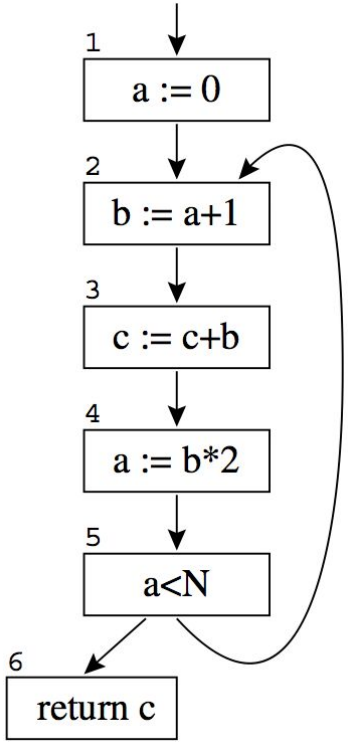
- Execute o algoritmo para o grafo do exemplo anterior
- Temos como melhorar o desempenho?
- Sim:
  - Usando uma ordem melhor para os nós
  - Repare que in[i] é calculado a partir de out[i] e out[i-1] é computado a partir de in[i]
  - A convergência ocorre antes de computarmos
    - out[i], in[i], out[i-1], ...
    - Invertendo a ordem dos nós aproveitamos mais cedo as informações!

Algoritmo

- O fluxo da análise deve seguir o fluxo do liveness: backwards
- A ordenação pode ser obtida através de uma busca em profundidade
- Complexidade:
  - Pior caso: $O(N^4)$
  - Com a ordenação, na prática roda tipicamente entre $O(N)$ e $O(N^2)$

Control Flow Graph (CFG)

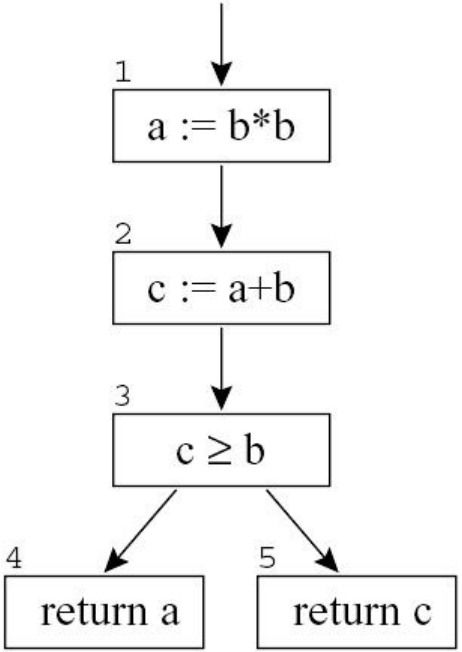
			1st		2nd		3rd	
	use	*def*	*out*	*in*	*out*	*in*	*out*	*in*
6	c			c		c		c
5	a		c	ac	ac	ac	ac	ac
4	b	a	ac	bc	ac	bc	ac	bc
3	bc	c	bc	bc	bc	bc	bc	bc
2	a	b	bc	ac	bc	ac	bc	ac
1		a	ac	c	ac	c	ac	c



Algoritmo

- É conservativo:
  - Se uma variável pode estar viva em algum nó n, ela estará no out[n] para todo m em pred[n]
  - Pode haver alguma variável em out[n] que na verdade não seja realmente usada adiante
- Deve ser dessa maneira para prevenir o compilador de tornar o programa errado!

Exemplo



- Qual seria o conjunto in[4]?
- E o out[3]?
- Algo estranho?

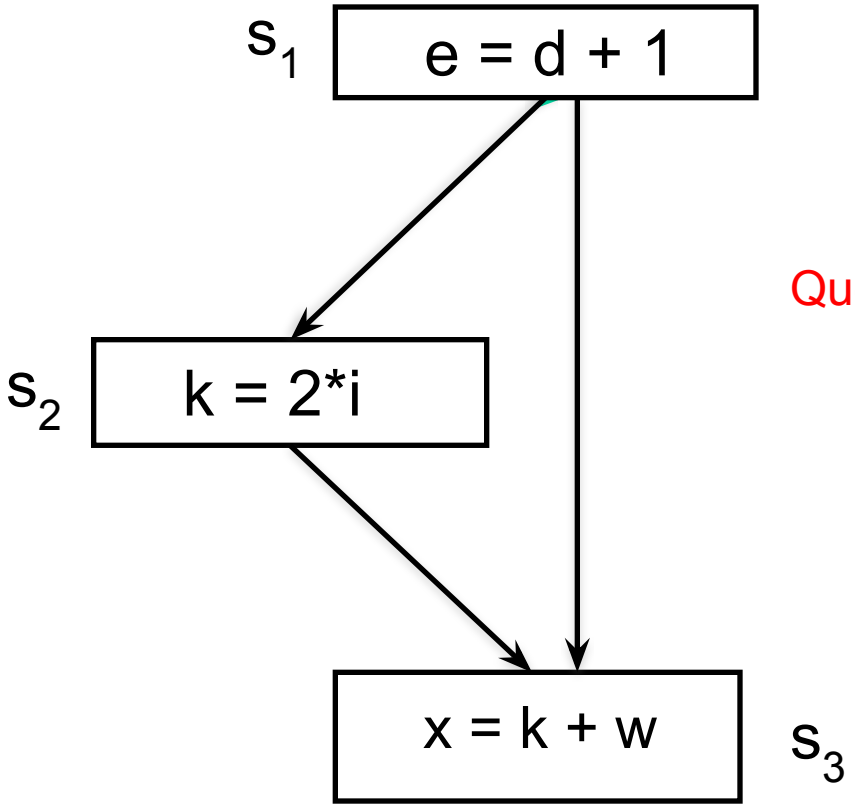
b*b + b >= b
sempre
verdadeiro

Dead Code Elimination

Dead Code Elimination

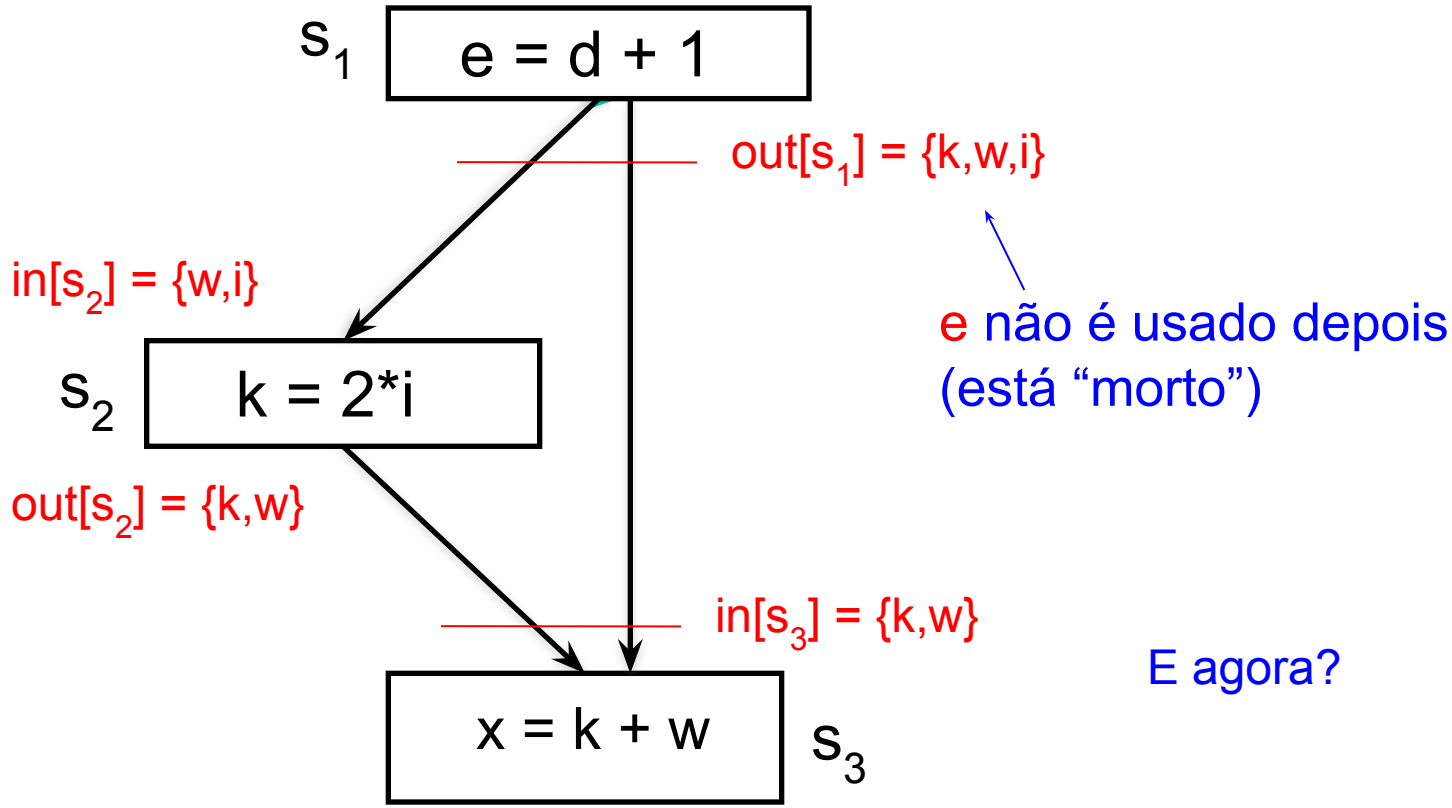
- Se a não está viva em out[s] em:
  - s: a ← t op x
  - s: a ← M[x]
- Podemos remover s
- Qual análise é necessária?
- Tomar cuidado com efeitos colaterais

Dead Code Elimination



Que análise é necessária?

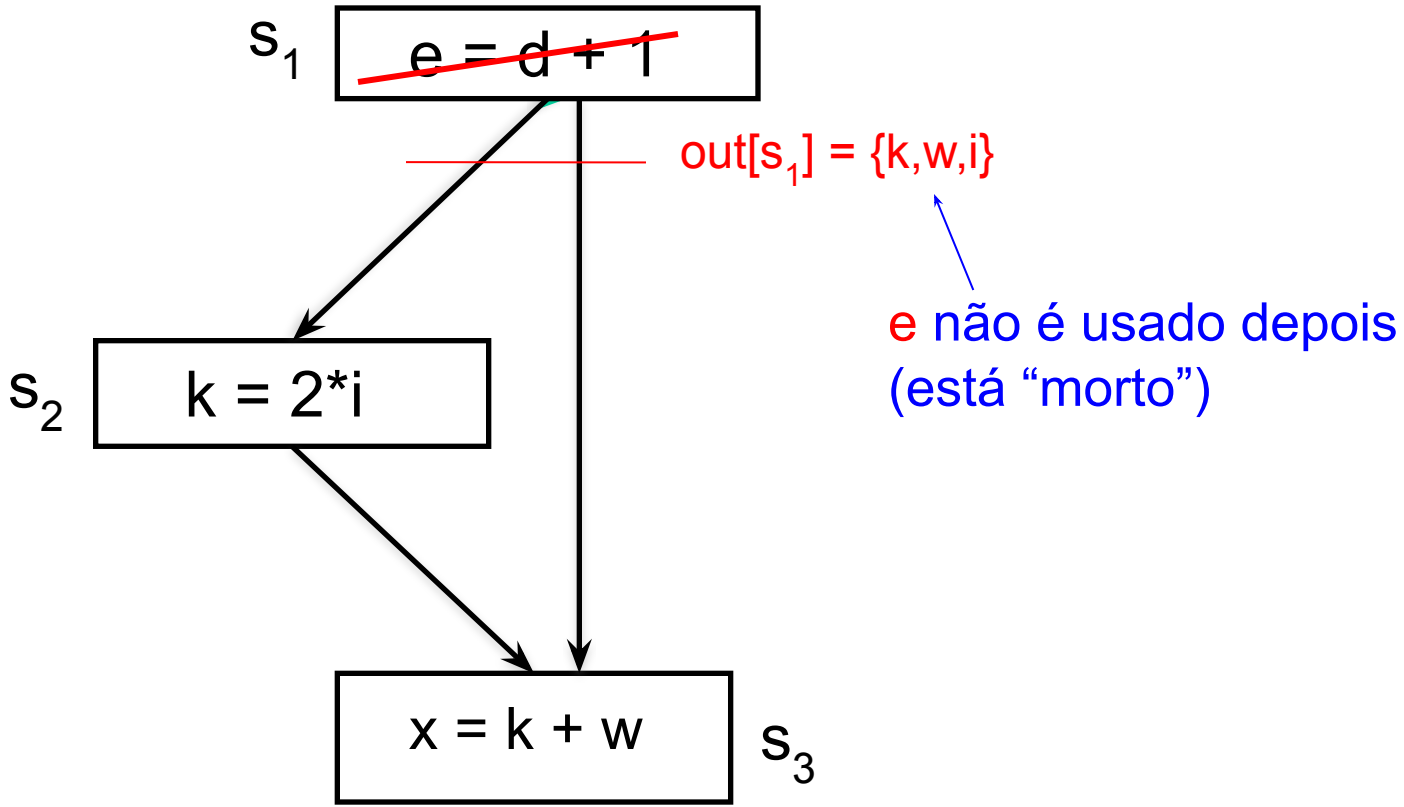
Using Liveness Analysis



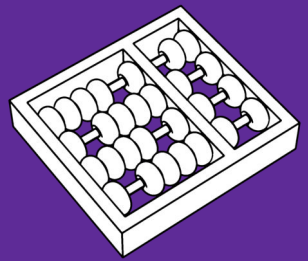
e não é usado depois
(está "morto")

E agora?

Dead Code Elimination



Obrigado!
Merci!



UNICAMP

Pallete

Bubbles

BUBBLE
Lorem ipsum dolor sit amet, consectetur adipiscing elit.

BUBBLE
Lorem ipsum dolor sit amet, consectetur adipiscing elit.

BUBBLE
Lorem ipsum dolor sit amet, consectetur adipiscing elit.

BUBBLE
Lorem ipsum dolor sit amet, consectetur adipiscing elit.

BUBBLE
Lorem ipsum dolor sit amet, consectetur adipiscing elit.

BUBBLE
Lorem ipsum dolor sit amet, consectetur adipiscing elit.

BUBBLE
Lorem ipsum dolor sit amet, consectetur adipiscing elit.

BUBBLE
Lorem ipsum dolor sit amet, consectetur adipiscing elit.

BUBBLE
Lorem ipsum dolor sit amet, consectetur adipiscing elit.

BUBBLE
Lorem ipsum dolor sit amet, consectetur adipiscing elit.

BUBBLE
Lorem ipsum dolor sit amet, consectetur adipiscing elit.

DRACULA

Tables

Table Title	
Column 1	**Column 2**
One	Two
Three	Four

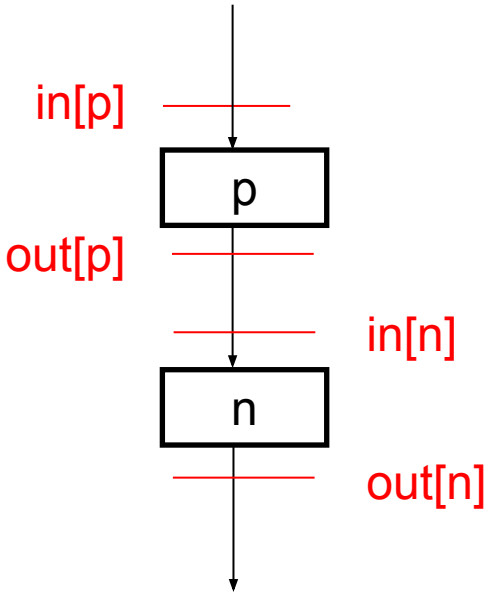
Table Title	
Column 1	**Column 2**
One	Two
Three	Four

Table Title	
Column 1	**Column 2**
One	Two
Three	Four

Table Title	
Column 1	**Column 2**
One	Two
Three	Four

Emojis

- Suponha dois nós no CFG n e p
  - p é o único predecessor de n
- Neste caso, podemos combinar os efeitos gen e kill de n e p
- Teremos apenas um nó no grafo
- Podemos repetir para todas as instruções de um bloco básico!



- Bloco básico: Apenas uma entrada, uma saída e nenhum desvio contido nele
- Pense em Reaching Definitions
- Como combinar gen e kill para um bloco básico?
  - out[n] = gen[n] U (in[n] – kill[n]).
  - in[n] = out[p].