MO615B - Implementação de Linguagens II
e
MC900A - Tópicos Especiais em Linguagem de Programação

Prof. Sandro Rigo

www.ic.unicamp.br/~sandro

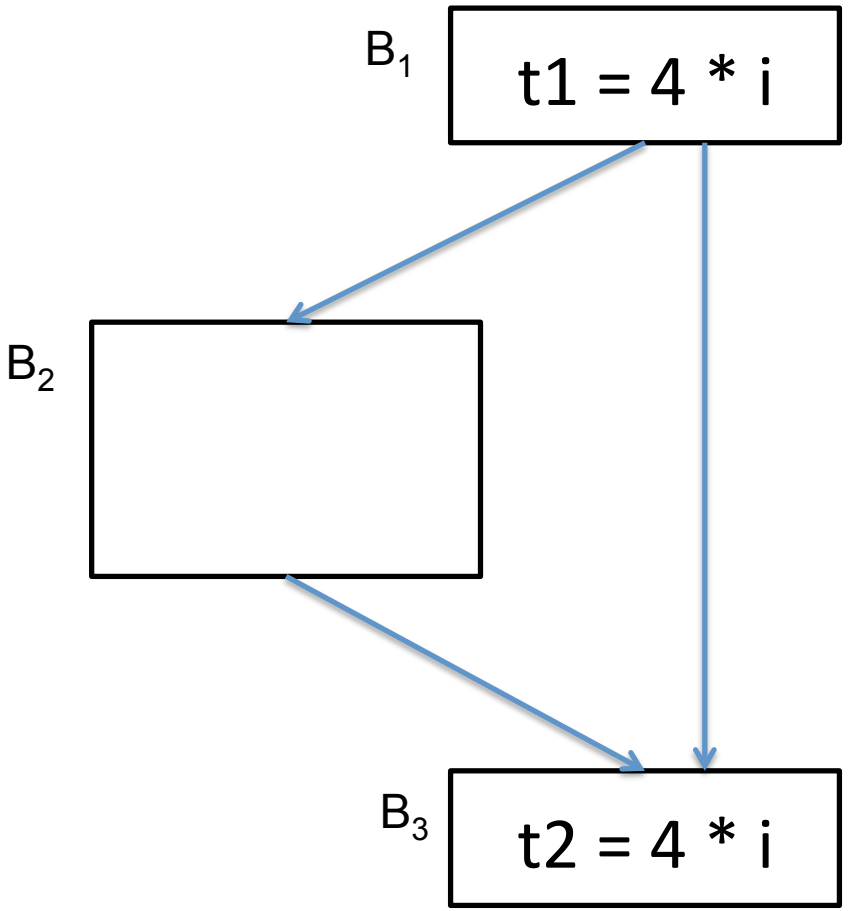
Available Expressions

- 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

Available Expressions

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

Available Expressions



Available Expressions

- *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.

Equações da DFA

- Computamos *gen* e *kill* para cada B como visto anteriormente
- Temos: $out[ENTRY] = \varnothing$

Para todo bloco B diferente de ENTRY

$out[B] =$

$in[B] =$

Equações da DFA

- Computamos *gen* e *kill* para cada B como visto anteriormente
- Temos: $out[ENTRY] = \varnothing$

Para todo bloco B diferente de ENTRY

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

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

Available Expressions

$$out[ENTRY] = \varnothing$$

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

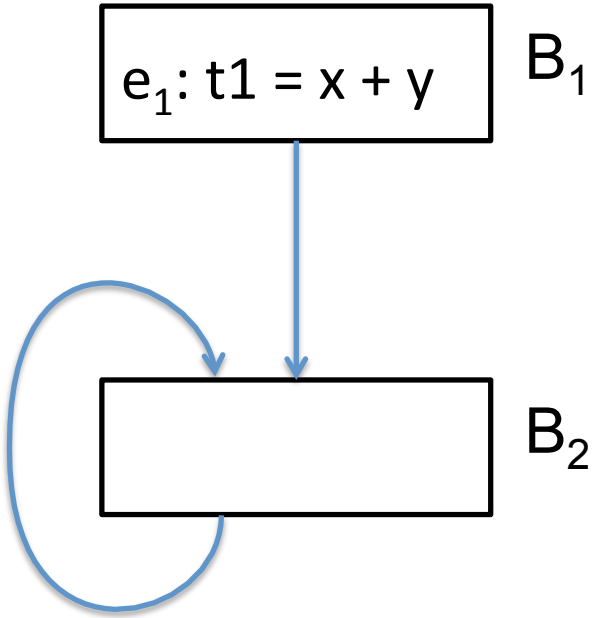
$$in[B] = \bigcap_{P \in predecessores\ de\ B} out[P]$$

in[B_1] =

out[B_1] =

in[B_2] =

out[B_2] =



Available Expressions

$$out[ENTRY] = \varnothing$$

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

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

$in[B_1]$ = {}

$out[B_1]$ = $\{e_1\}$

$in[B_2]$ = $out[B_1] \cap out[B_2]$

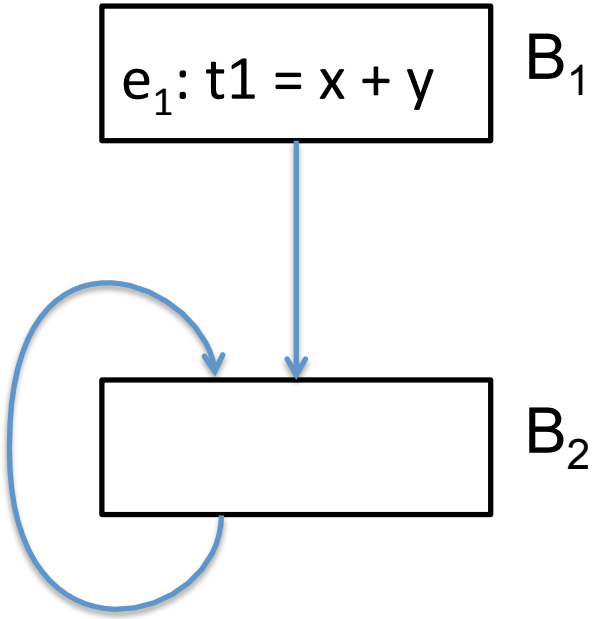
= $\{e_1\} \cap \{\}$

= {}

$out[B_2]$ = $gen[B_2] \cup (in[B_2] - kill[B_2])$

= {} U ({} – {})

= {}



Available Expressions

$$out[ENTRY] = \varnothing$$

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

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

// Iniciar out[B] com todas as expressões

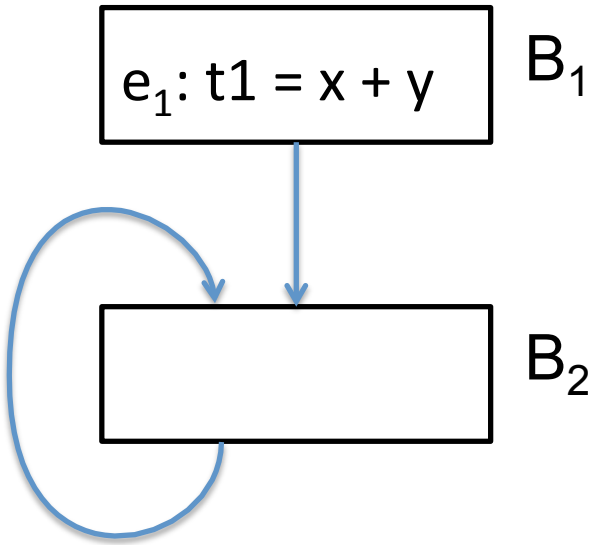
out[B_1] = {e_1}

out[B_2] = {e_1}

in[B_1] =

in[B_2] =

out[B_2] =



Diferenças para *Reaching Defs*

- O operador de confluência é intersecção
  - Tem que vir por todos os caminhos
- Estimativa inicial é grande
  - Intersecção vai diminuindo os conjuntos até chegar ao maior ponto fixo

Solução Iterativa

OUT[ENTRY] = {} // Conjunto vazio

for (each basic block *B* other than *ENTRY*)

  OUT[*B*] = {e_1, e_2, e_3, ..., e_N} // Todas as expressões

while (changes to any OUT occur)

  for (each basic block *B* other than *ENTRY*) {

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

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

}

Exemplo

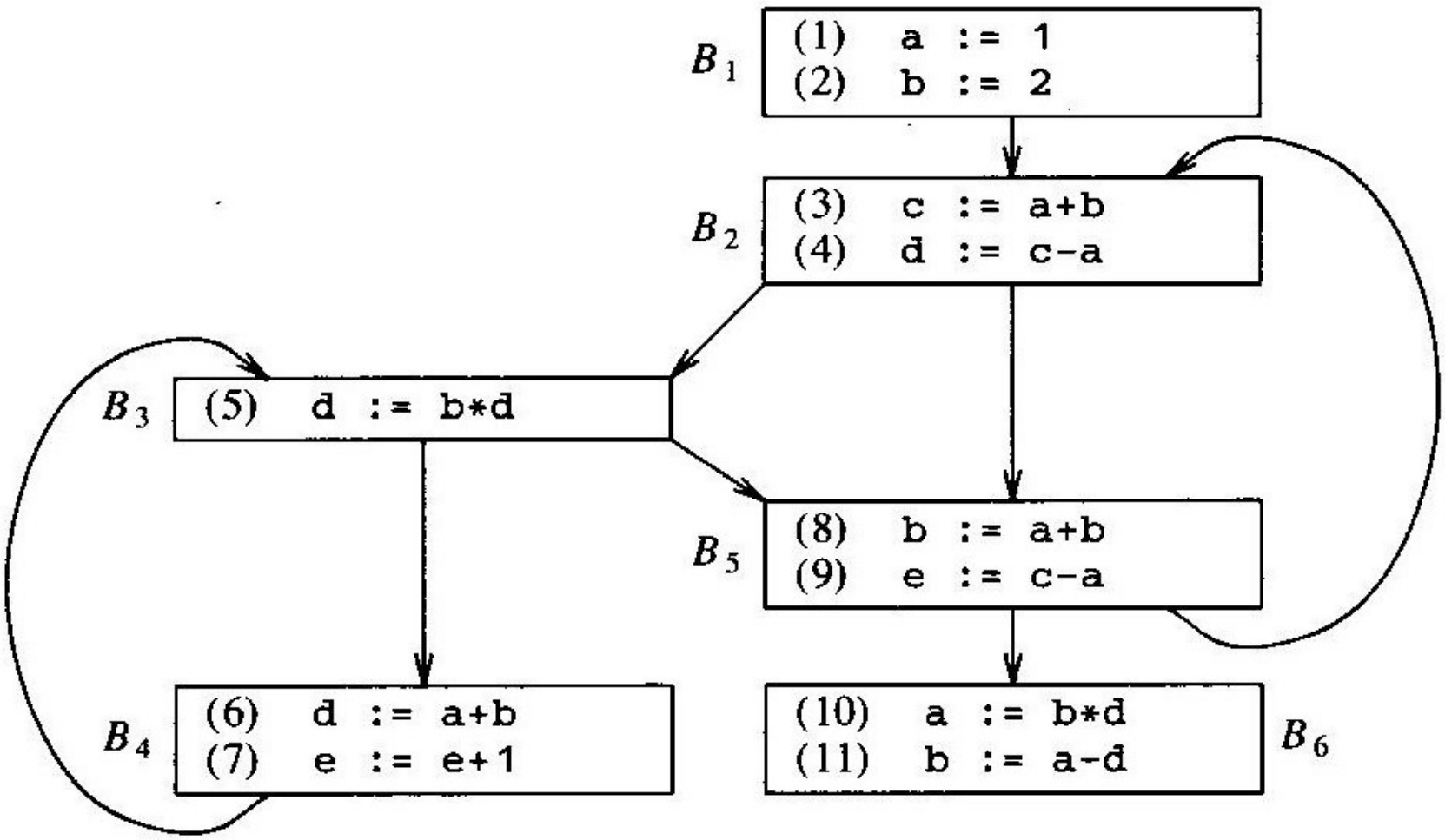


Fig. 10.74. Flow graph.

Live-Variable Analysis

- Linguagem intermediária
  - Gerada pelo *front-end* considerando número infinito de registradores para temporários
- Máquinas reais têm um número finito de registradores
  - 32 é um número típico para máquinas RISC
- 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*)

Live-Variable Analysis

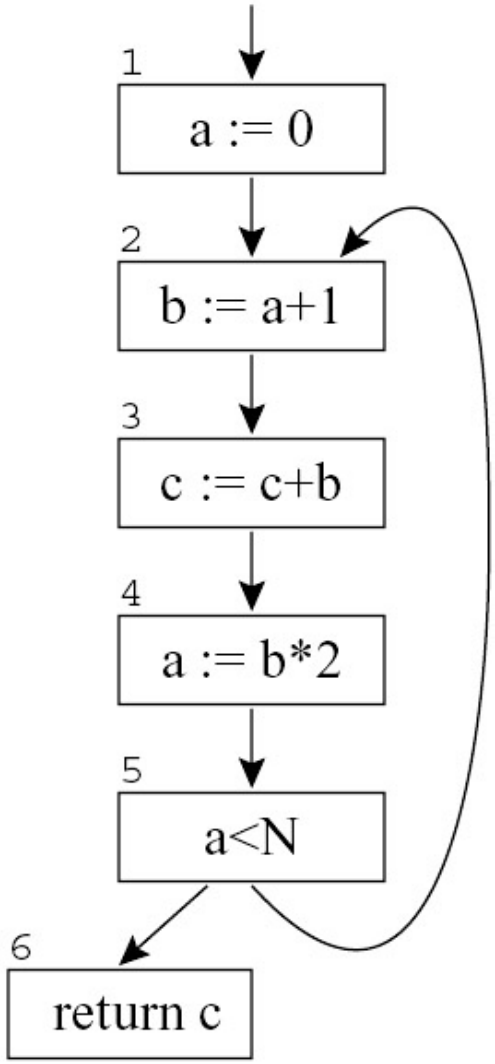
- Dizemos que uma variável está *viva* em um ponto *p* se ela pode vir a ser usada no futuro em um caminho a partir de *p*.

- O compilador analisa a RI para saber quais variáveis estão *vivas* ao mesmo tempo.

- Esta tarefa então, é conhecida como *live-variable analysis, ou liveness analysis* (análise de longevidade).

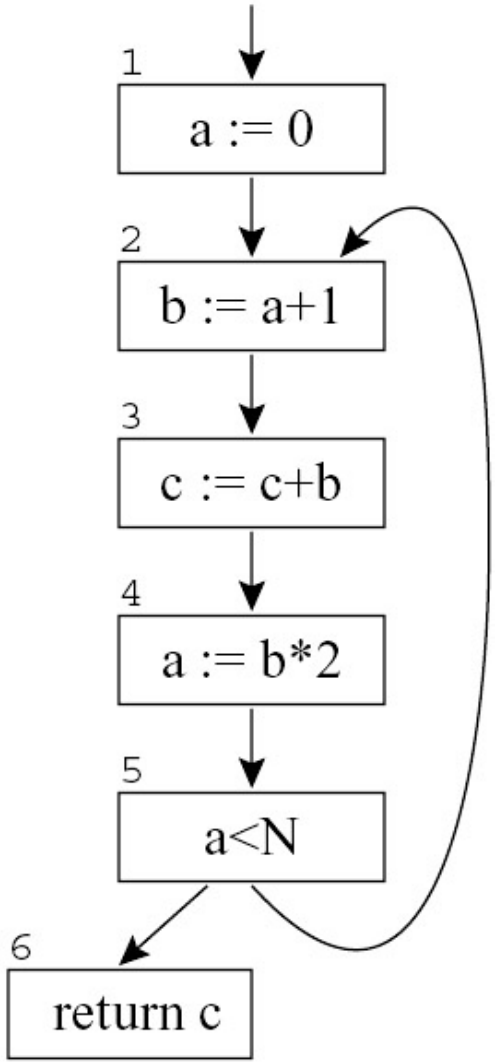
Exemplo

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



Exemplo

Em quais pontos do CFG a variável b está viva?



Exemplo

- Quantos registradores precisamos para alocar todas as variáveis do programa anterior?

Live-Variable Analysis

- IN[B]: conjunto de variáveis vivas no início de B
- OUT[B]: conjunto de variáveis vivas no final de B
- def_B: conjunto de variáveis atribuídas em B antes de qualquer uso daquela variável em B.
- use_B: conjunto de variáveis utilizadas em B antes de qualquer atribuição à variável em B.

B_2

```
b = a + 1
c = c + b
a = b + 2
if a < N goto B2
```

def_{B2} =

use_{B2} =

Computando *Liveness*

1. Se a variável *v* está em use_B, então *v* é *live-in* em *B e pertence a in[B]*.

2. Se a variável *v* é *live-in* no bloco *B*, então ela é *live-out* para todo bloco P que precede B.

3. Se a variável *v* é *live-out* no bloco *B*, e não está em def_B, então *v* é também *live-in* em *B.*

Equações da DFA

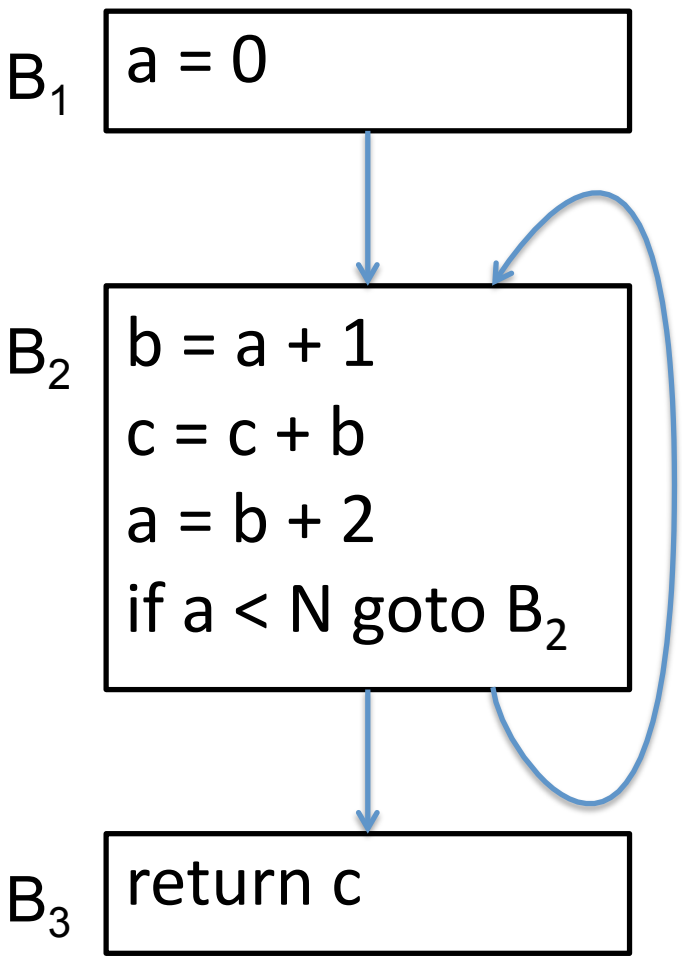
- Computamos *use (gen)* e *def (kill)* para cada B como visto anteriormente
- Neste caso, IN[B] é definido em função de OUT[B]
- Temos: $in[EXIT] = \varnothing$

Para todo bloco B diferente de EXIT

$$in[B] = use_B \cup \left(out[B] - def_B \right)$$

$$out[B] = \bigcup_{S \in sucessores\ de\ B} in[S]$$

Exemplo



$in[B_1]$ =

def_{B1} =

use_{B1} =

$out[B_1]$ =

$in[B_2]$ =

def_{B2} =

use_{B2} =

$out[B_2]$ =

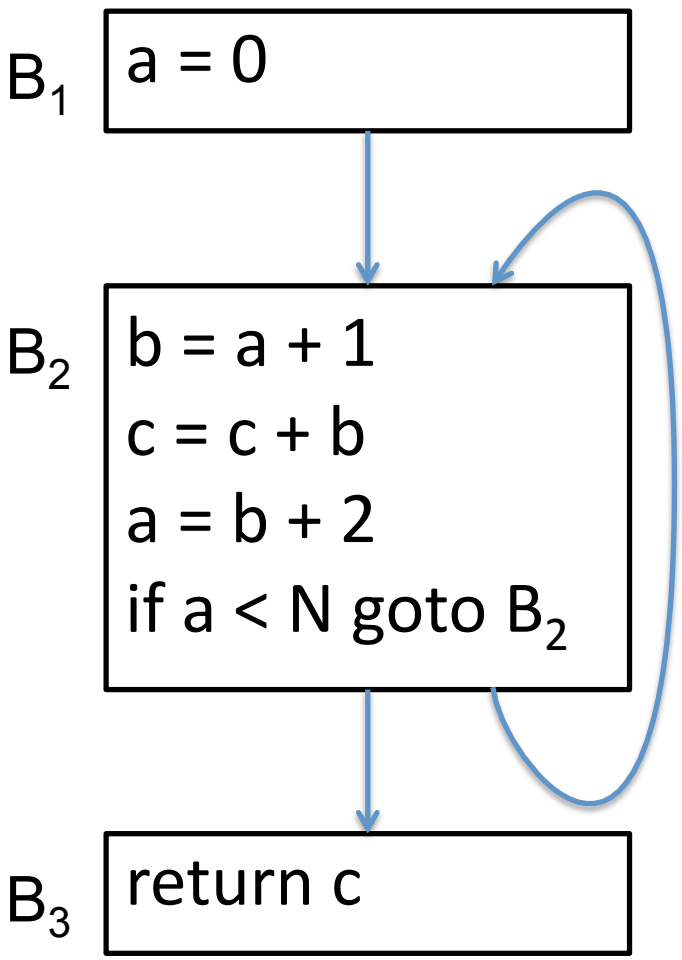
$in[B_3]$ =

def_{B3} =

use_{B3} =

$out[B_3]$ =

Exemplo



$in[B_1] = \{c\}$

$def_{B1} = \{a\}$

$use_{B1} = \{\}$

$out[B_1] = \{a,c\}$

$in[B_2] = \{a,c\}$

$def_{B2} = \{b\}$

$use_{B2} = \{a,c\}$

$out[B_2] = \{a,c\}$

$in[B_3] = \{c\}$

$def_{B3} = \{\}$

$use_{B3} = \{c\}$

$out[B_3] = \{\}$

Diferenças para as anteriores

- Direção da análise
  - O conjunto IN[B] é definido em função de OUT[B]
  - *Backward*: $in = f_s (out)$

Solução Iterativa

IN[*EXIT*] = {}; // Conjunto vazio

for (each basic block *B* other than *EXIT*)

  IN[*B*] = {};

while (changes to any IN occur)

  for (each basic block *B* other than *EXIT*) {

$$out[B] = \bigcup_{S \in sucessores\ de\ B} in[S]$$

$$in[B] = use_B \cup \left(out[B] - def_B \right)$$

}

Def-Use Chain

- Liga cada *definição* aos *usos* que alcança
- Pode ser calculada a partir das *use-def chains* (mais comum). *Use-def chains* são calculadas a partir de *reaching definitions*.
- Pode ser definida como um DFA com as mesmas equações de *live-variable analysis*

Def-Use Chain

- OUT[B]: usos alcançáveis a partir do final de B
- use_B: usos $u_i(x)$ que aparecem antes de qualquer definição não ambigua à variável x.
- def_B: todos os usos $u_i(x)$ para as variáveis x que forem definidas no bloco básico.
- Equações e algoritmo são idênticos.

Grafo de Interferência

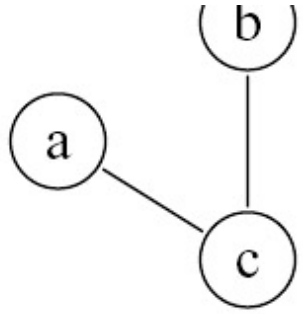
- A informação de liveness é usada para otimização
  - Alocação de registradores
- Interferência: ocorre quando a e b não podem ocupar o mesmo registrador
  - *Live ranges* com sobreposição
  - a não pode ser alocada a r1

Grafo de Interferência

- Representação:

	a	b	c
a			**x**
b			**x**
c	**x**	**x**	

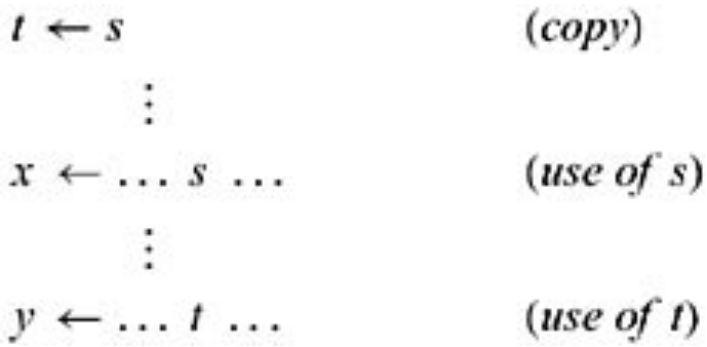
(a) Matrix



(b) Graph

Grafo de Interferência

- MOVE: é importante não criar falsas interferências entre a fonte e destino



- S e t estariam vivas após a instrução de cópia
- Devemos aproveitar o mesmo registrador

Grafo de Interferência

1. Definição de a que não seja move:
    1. Live-out = b1,…, bj
        1. Adicione as arestas (a, b1),…,(a, bj).

2. Moves a ← c:
    1. Live-out = b1,…, bj
        1. Adicione as arestas (a, b1),…,(a, bj) para os bi's que não são o mesmo que c