Compiladores
(IF688)

Leopoldo Teixeira
lmt@cin.ufpe.br | @leopoldomt

Exemplo

```
a := b + c
b := a - d
c := b + c
d := a - d
```

Exemplo

```
a := b + c
b := a - d
c := b + c
d := b
```

Safety e Profitability

- Só vamos aplicar a otimização, se não for necessário avaliar novamente (safety)
- É interessante substituir avaliações redundantes com referências a valores computados anteriormente (profitability)
- porém, importante perceber que ao reescrever, aumentamos o “tempo de vida” de **b**, e potencialmente reduzimos o tempo de vida de **a** ou **d**.

Local Value

Numbering

Local Value Numbering

- Fazer uma travessia no bloco básico e assinalar números distintos a cada valor que o bloco computa

- Chave: Escolher números de tal forma que duas expressões e_i e e_j tem o mesmo valor *sse* os valores dos operandos são comprovadamente iguais

- Hashing de operações, para armazenar expressões já calculadas

Exemplo

$$a^2 := b^0 + c^1$$
$$b^4 := a^2 - d^3$$
$$c^5 := b^4 + c^1$$
$$d^4 := a^2 - d^3$$

Extensões

- Operações comutativas
- *Constant folding*
- Simplificações algébricas

Importância do esquema de nomes de variáveis...

Otimizações locais

- Código intermediário facilita
  - blocos básicos (identificar entrada e saída)
  - *single assignment* (uma definição por variável)
- Otimização nem sempre dá o sentido correto
  - código produzido não é necessariamente ‘ótimo’
  - Melhorias no programa (‘*Program improvement*’) seria termo mais adequado

https://xkcd.com/292/

Compiladores
(IF688)

Leopoldo Teixeira
lmt@cin.ufpe.br | @leopoldomt

Otimizações globais

- Operam em um procedimento ou método inteiro
  - ou seja, um CFG inteiro ==> múltiplos blocos básicos
- Pelo fato de tipicamente incluir construções como loops e if-then-else, estas otimizações normalmente envolvem uma fase de **análise**

Outline

- Otimizações globais modificam métodos inteiros
  - múltiplos blocos básicos
- O conceito fundamental em otimização é corretude, preservar semântica original do programa
- Utilizamos *data-flow analysis* para determinar se uma otimização pode ser aplicada

Data-flow Analysis

- Antes de aplicar uma otimização, é necessário localizar pontos onde o programa pode ser modificado *para melhor*

- Para coletar esta informação, o compilador normalmente usa algum tipo de análise estática

- Geralmente inicia-se com algum tipo de análise do fluxo de controle, para montar um CFG

- A partir do CFG, podemos analisar como os valores fluem por meio do código (*data-flow analysis*)

Iterative Data-flow Analysis

- Geralmente construídas a partir de um conjunto de equações definidas a partir de conjuntos
- Estas equações definem como dados são transferidos entre blocos básicos (funções de transferência)
- A solução para estas equações é um algoritmo de ponto fixo, simples e robusto

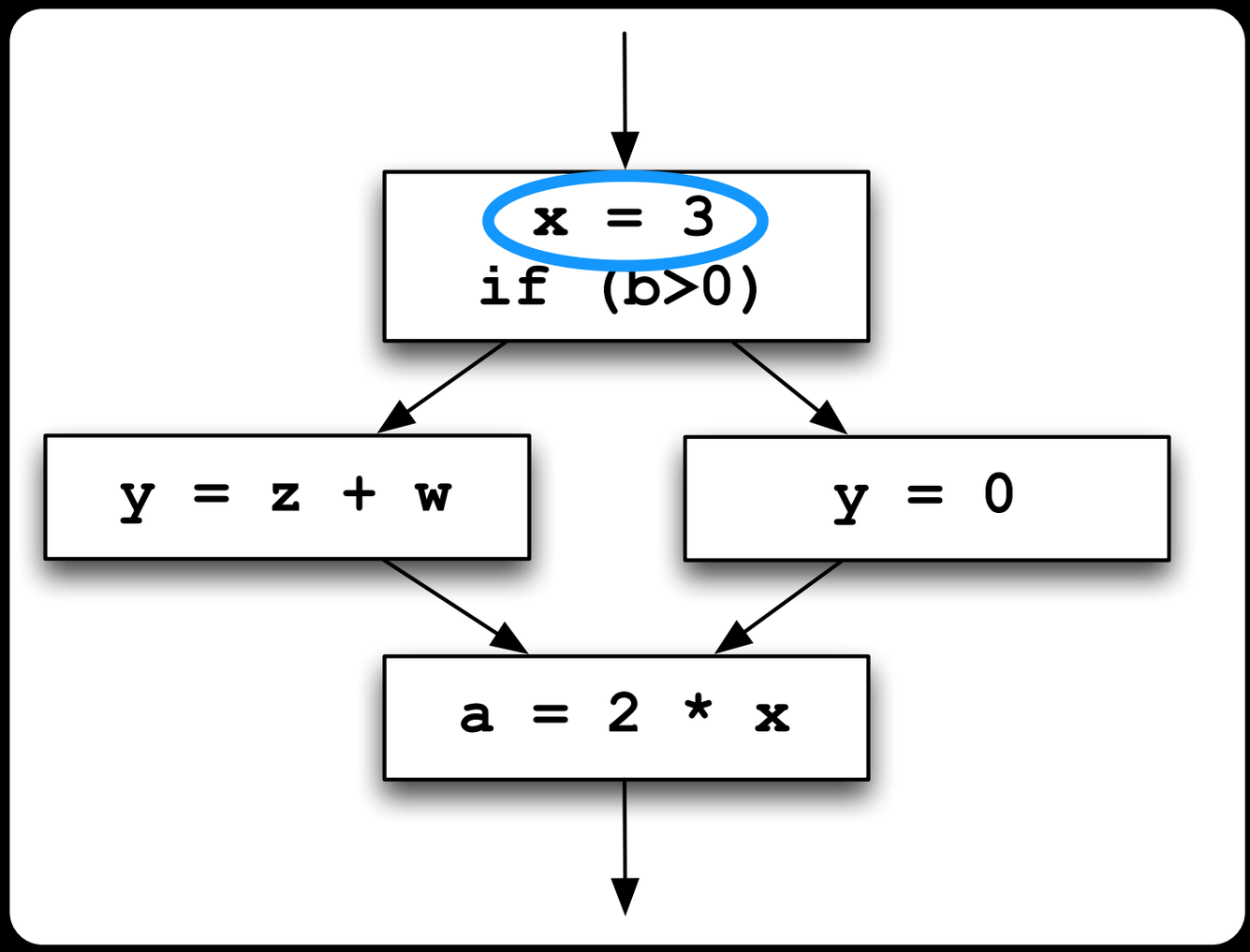
Otimizações locais

- Aplicadas a blocos básicos
  - *constant propagation*
  - *dead code elimination* - nem sempre possível

```
x = 3          x = 3
y = z * w      y = z * w      y = z * w
q = x + y      q = 3 + y      q = 3 + y
```

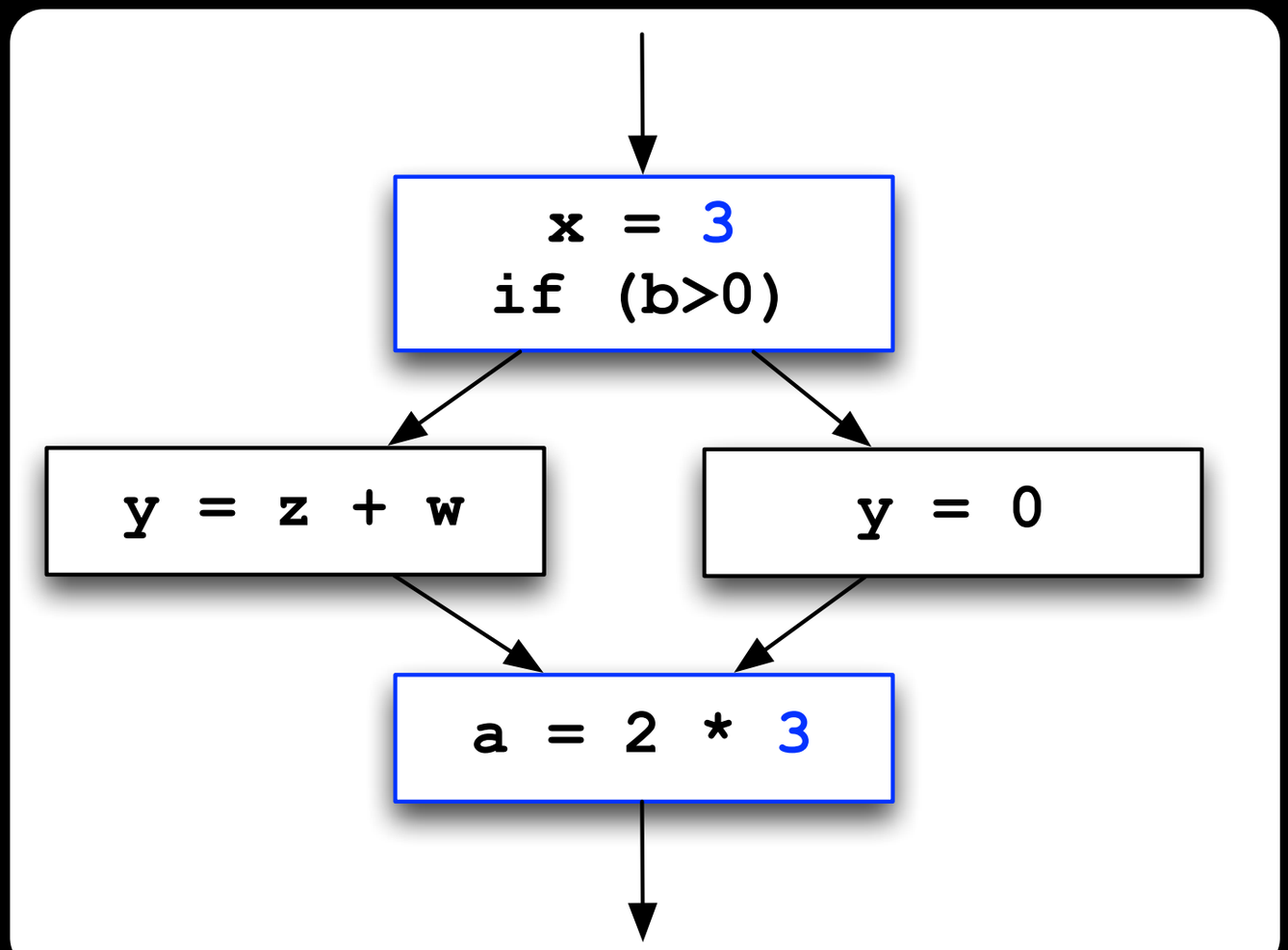
Otimizações globais

- Aplicadas a um CFG inteiro



Otimizações globais

- Aplicadas a um CFG inteiro

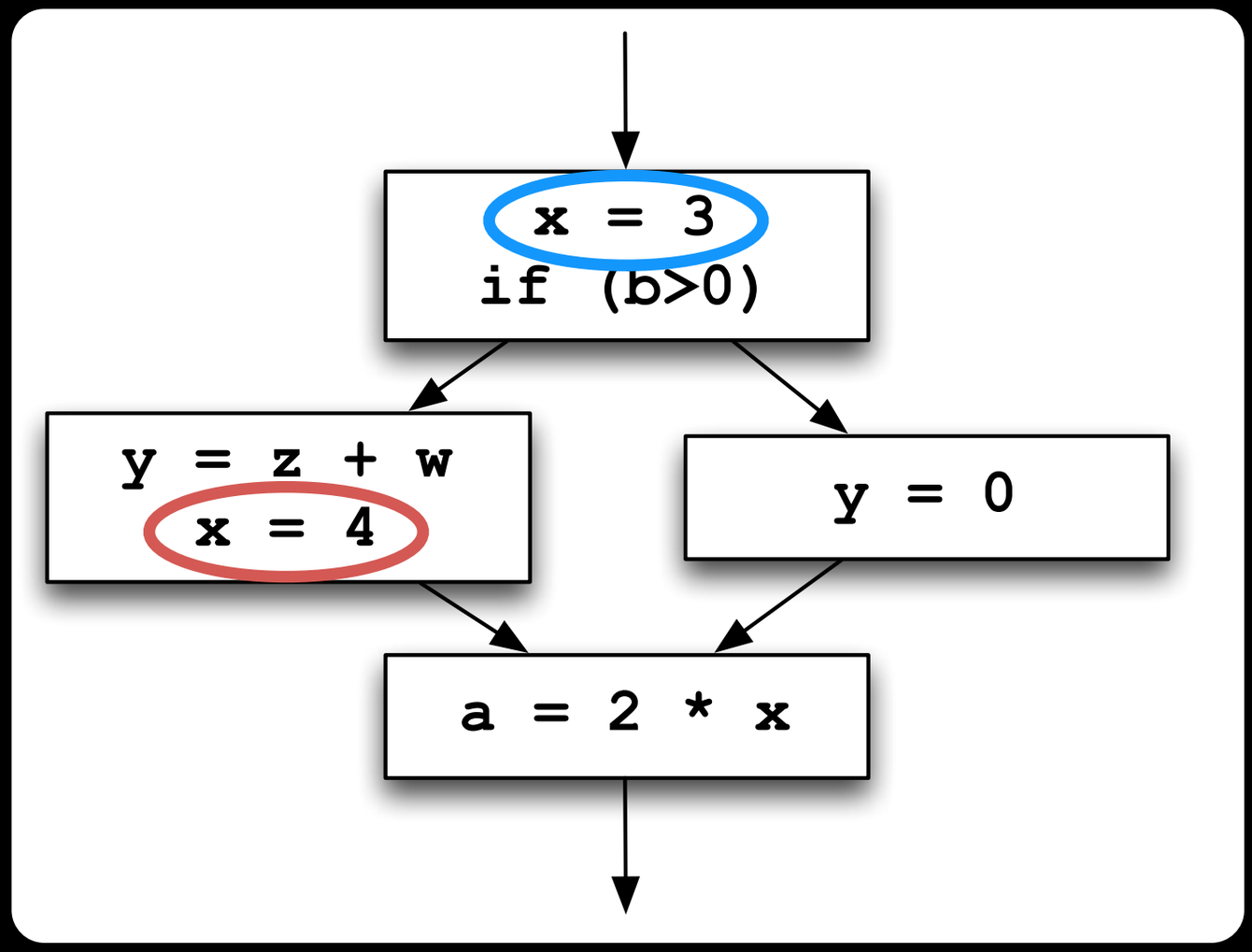


Corretude!

Como saber que está tudo ok com a propagação de uma constante?

Otimizações globais

- Existem situações que podem não ser ok



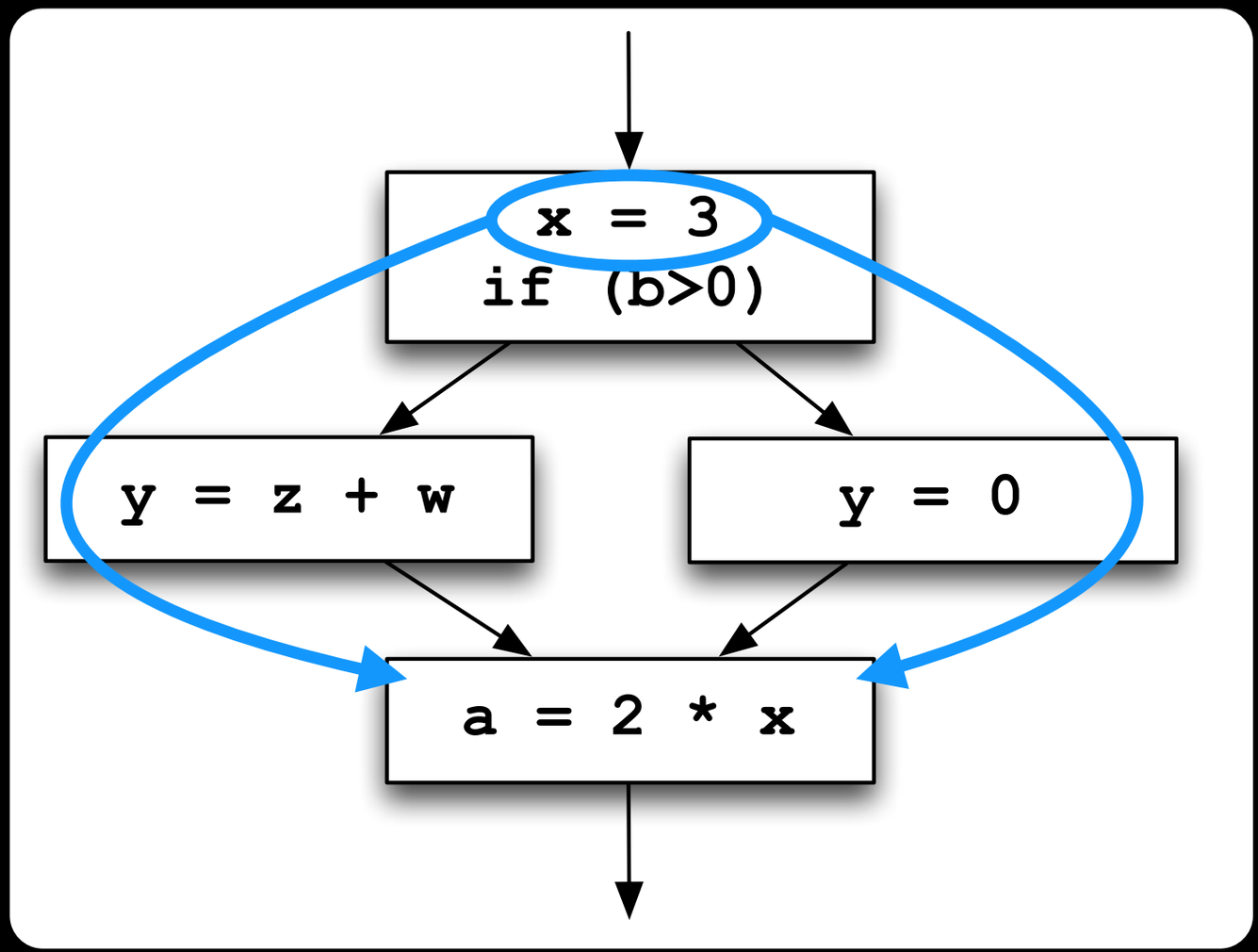
Qual a condição de corretude para propagar uma constante?

Expressada em termos de CFGs...

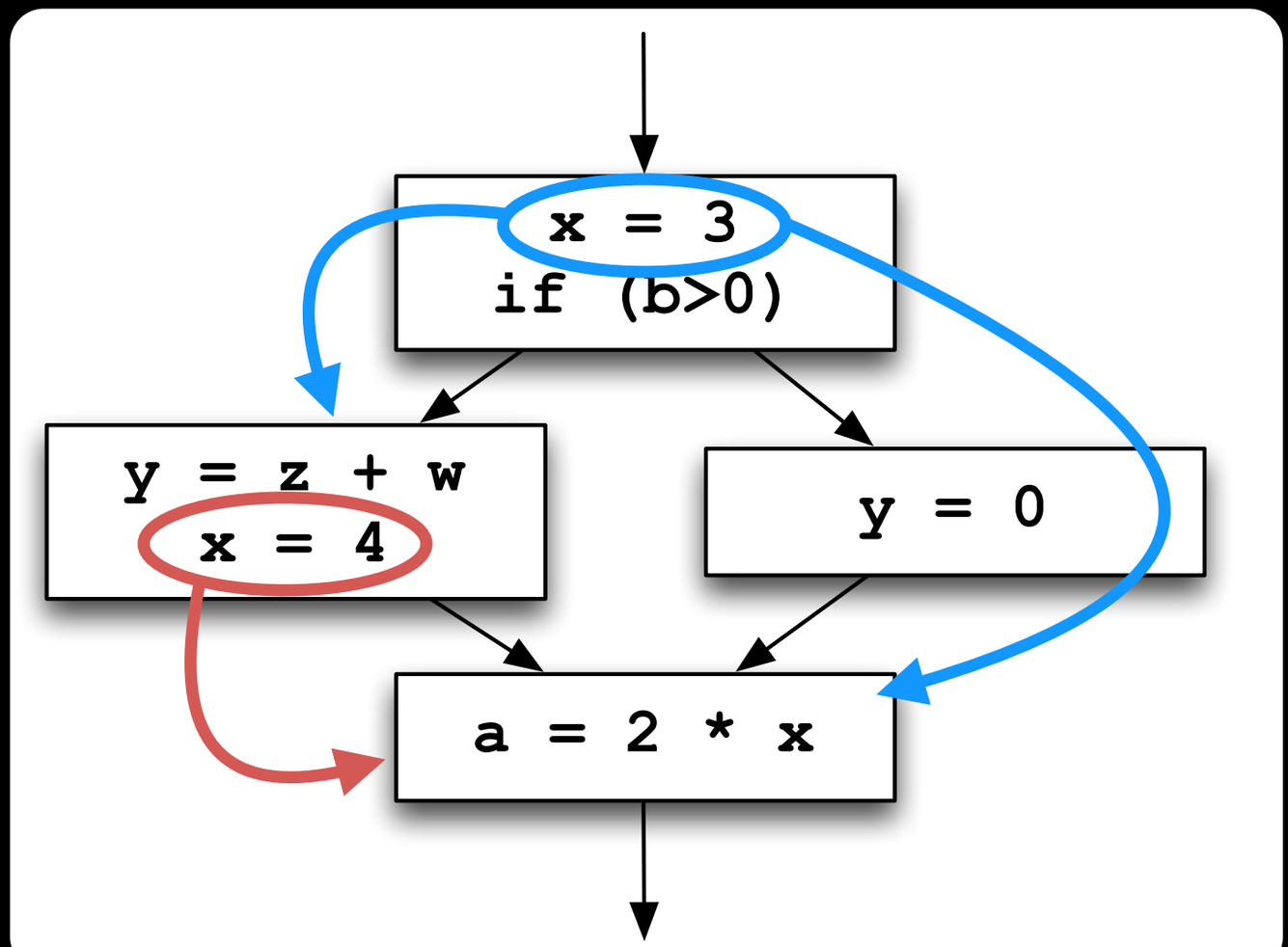
Corretude

- Para substituir o uso de **`x`** por uma constante **`k`**, devemos garantir a seguinte condição
  - Em todos os caminhos onde há um uso de **`x`**, a última atribuição a **`x`** é **`x = k`**
    - chamaremos esta condição de Φ

Revisitando…



Revisitando…



Quais os desafios para checar esta condição de corretude?

Checar Corretude (Φ)

- Não é trivial
- Ao quantificarmos *todos os caminhos*, precisamos incluir loops e branches de condicionais
- Checar esta condição requer análise global
  - global = análise do CFG para um corpo de método

Análise Global

- Tarefas de otimização global compartilham diversas características
  - a otimização depende do conhecimento de uma propriedade **P** em um ponto particular da execução do programa
  - Provar **P** em qualquer ponto requer conhecimento do corpo inteiro do método

Indecidibilidade

- Teorema de Rice: propriedades não triviais de uma linguagem são indecidíveis
  - halting problem
  - função F sempre retorna positivo?
- Propriedades sintáticas são decidíveis
  - quantas ocorrências de `x` existem?

Análise

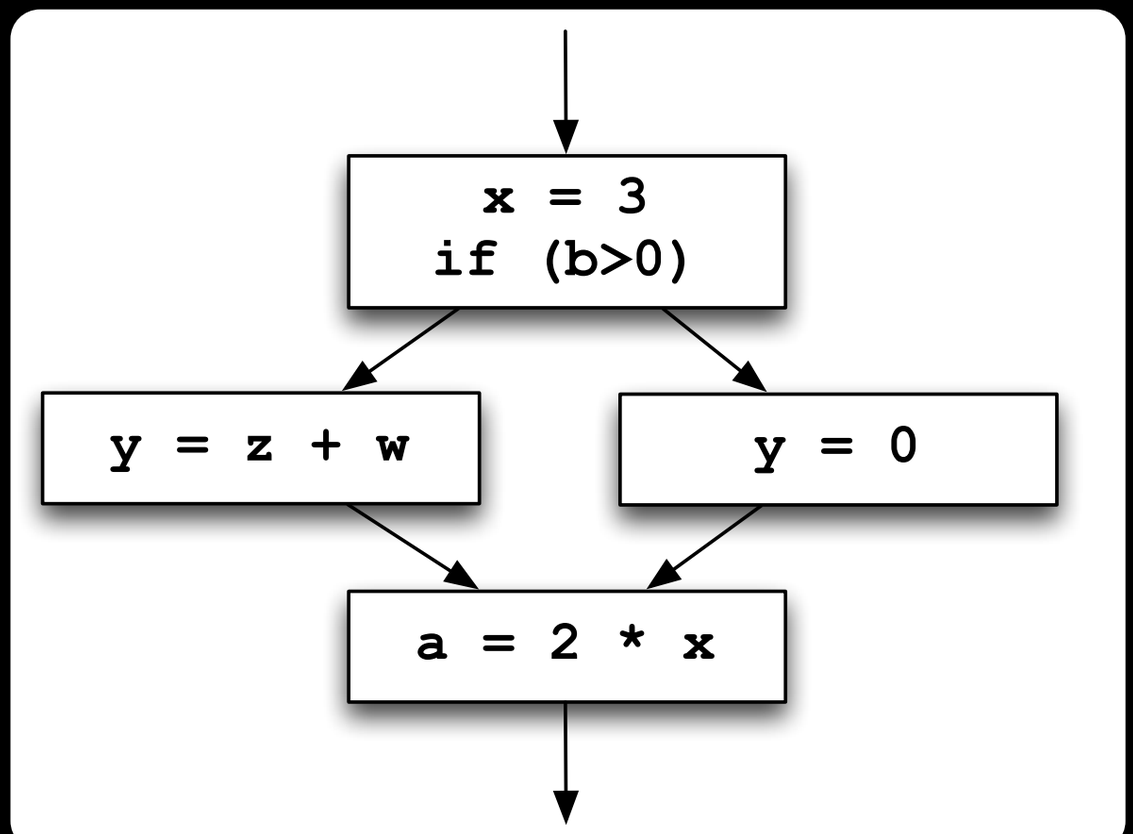
Conservadora!

Análise Conservadora

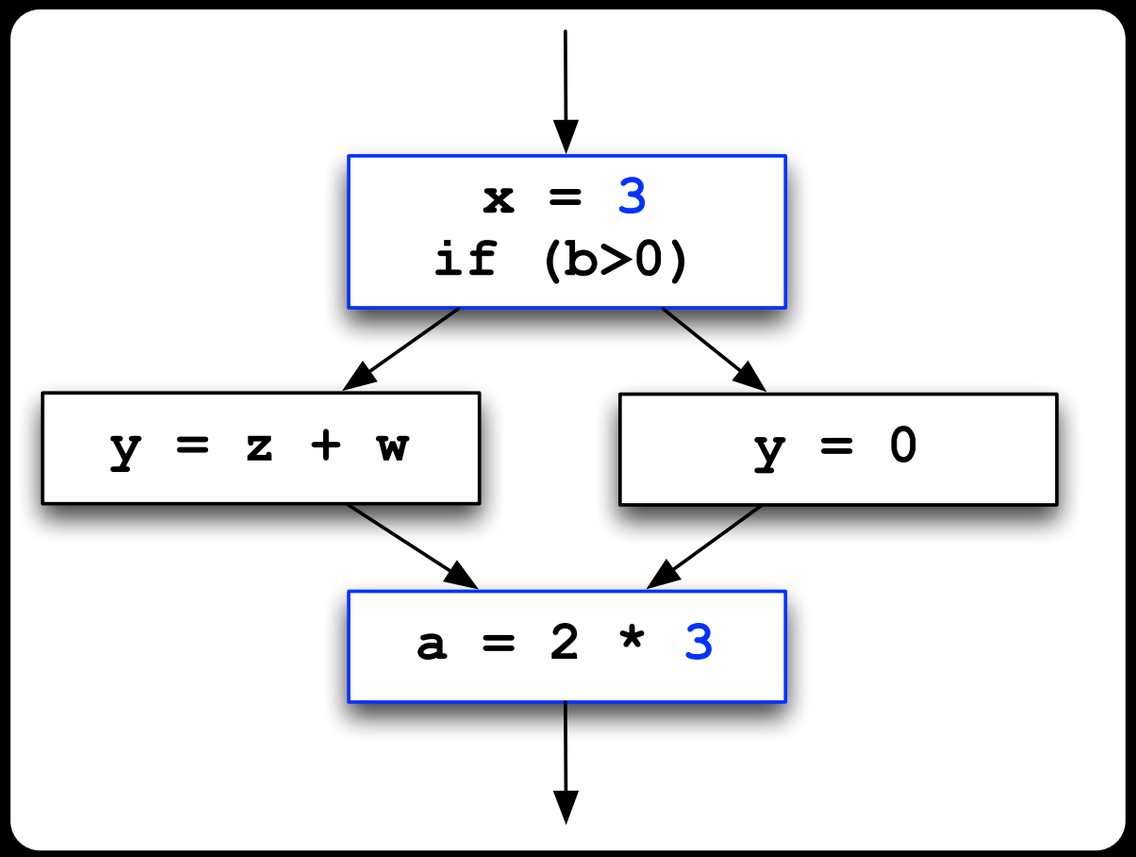
- Não podemos afirmar com certeza que `x` sempre vai ser 3
- Então, como aplicar *constant propagation*?
- Precisamos nos tornar conservadores…
  - Se otimização requer que **P** seja *true*, desejamos então saber se
    - **P** é definitivamente *true*; ou
    - Não temos certeza se **P** é *true*.

Tudo bem em dizer:
não sei
embora o alvo seja tentar dizer *não sei*
o mais raramente possível

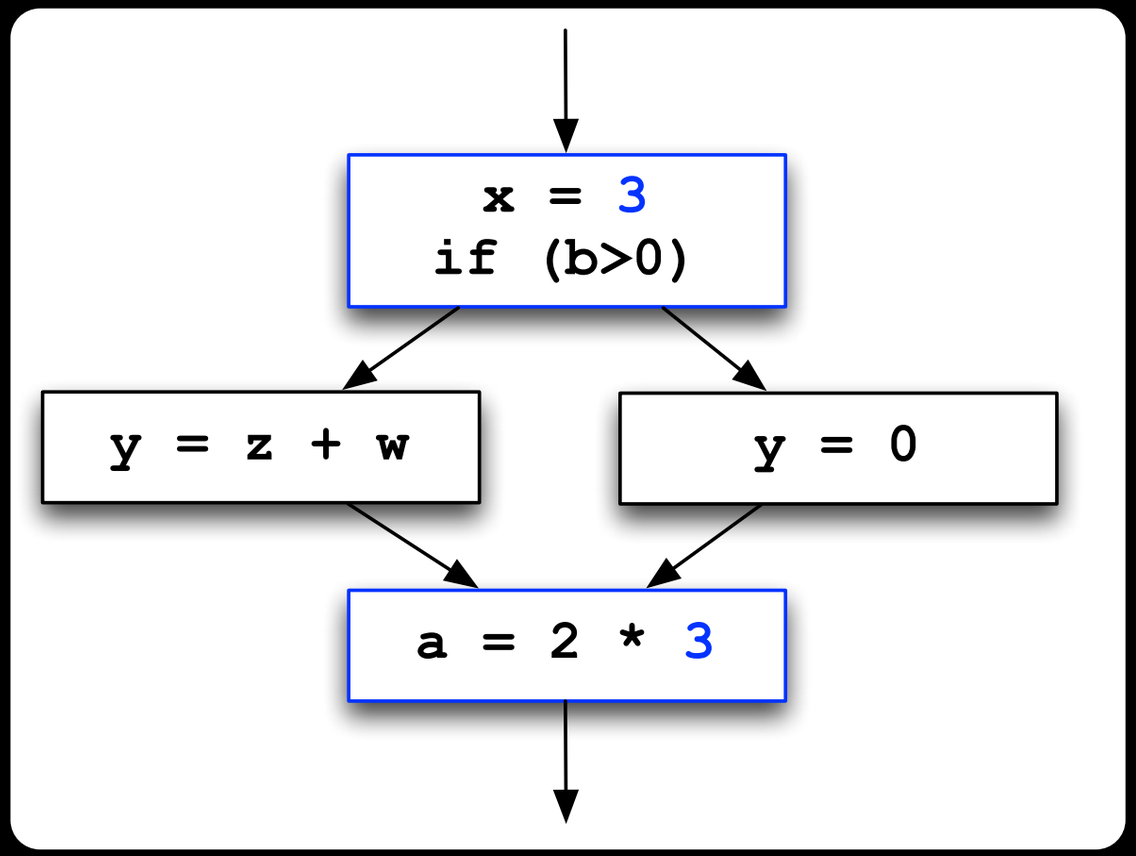
Otimização global

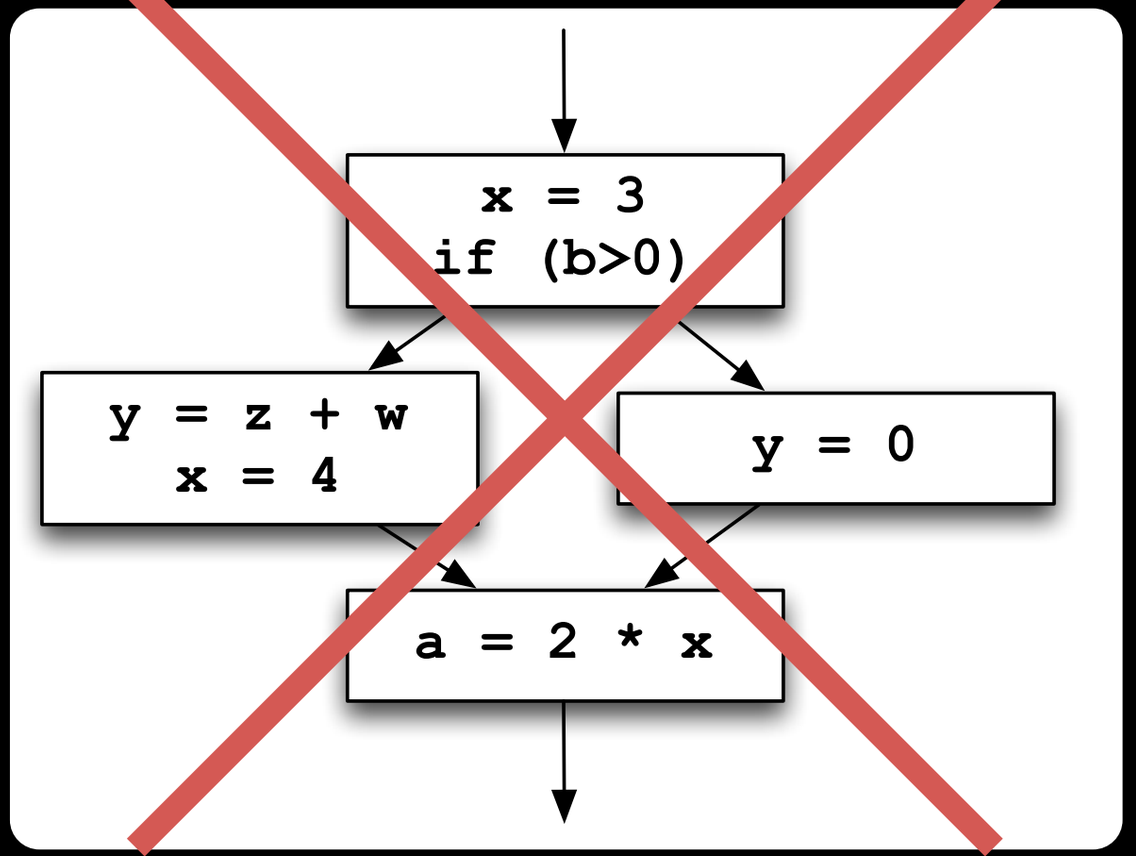


Otimização global



Otimização global





Análise de fluxo de dados

- As condições de corretude podem ser difíceis de checar
- Requerem análises globais (CFG inteiro)
- Análise de fluxo de dados global é uma técnica para resolver problemas deste tipo
- *Global constant propagation* é um exemplo de otimização que requer uma análise global

Global constant propagation

- Podemos propagar constantes em qualquer ponto onde Φ é verdade

- Considere o caso de computar Φ para uma variável x em todos os pontos do programa

- É possível?

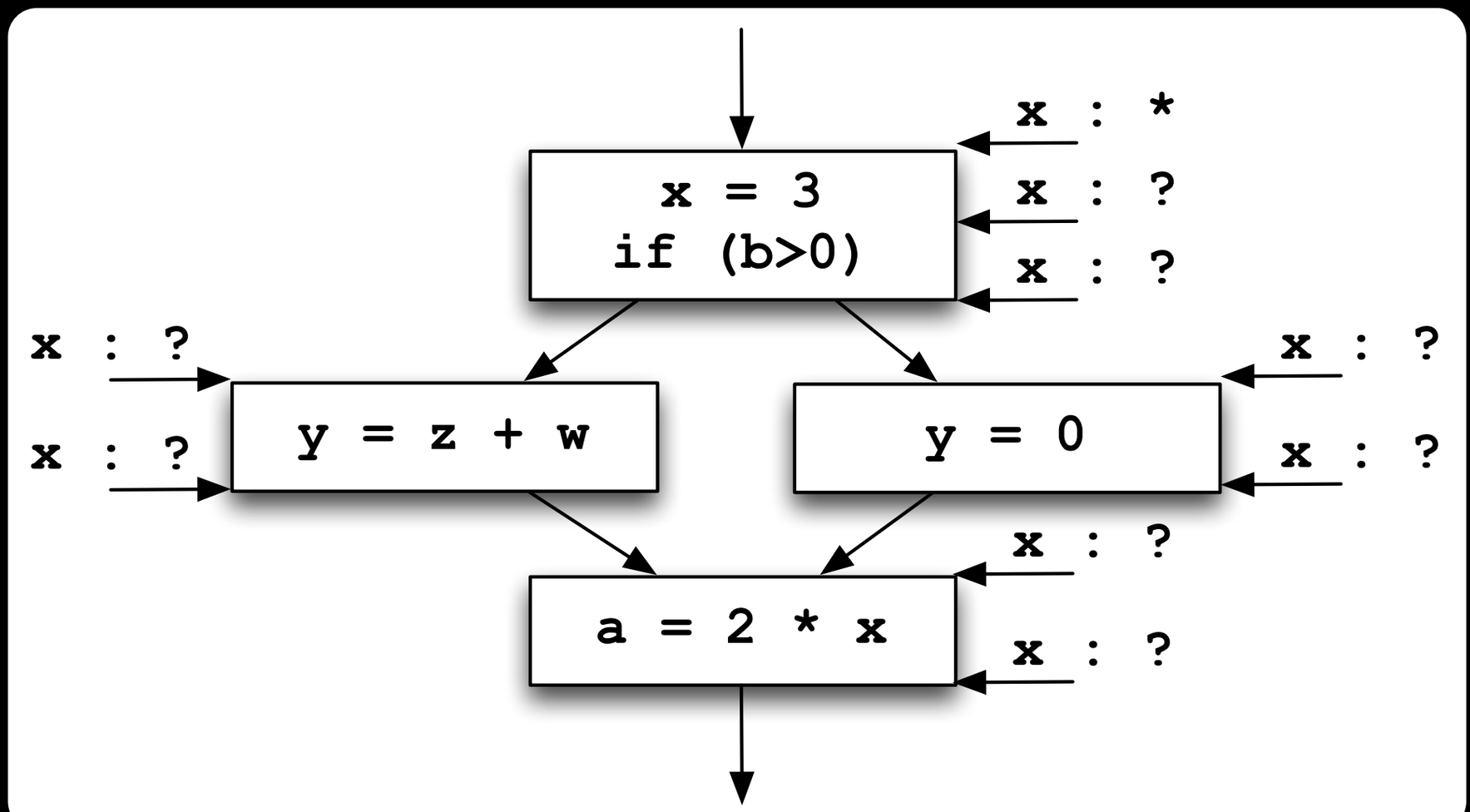
Global constant propagation

- Para tornar o problema preciso, associamos um dos seguintes valores com **`x`** em todos os pontos do programa

valor	interpretação
#	instrução não alcançada (ou não alcançável)
c	**`x = constante C`**
*	não sabemos se **`x`** é constante

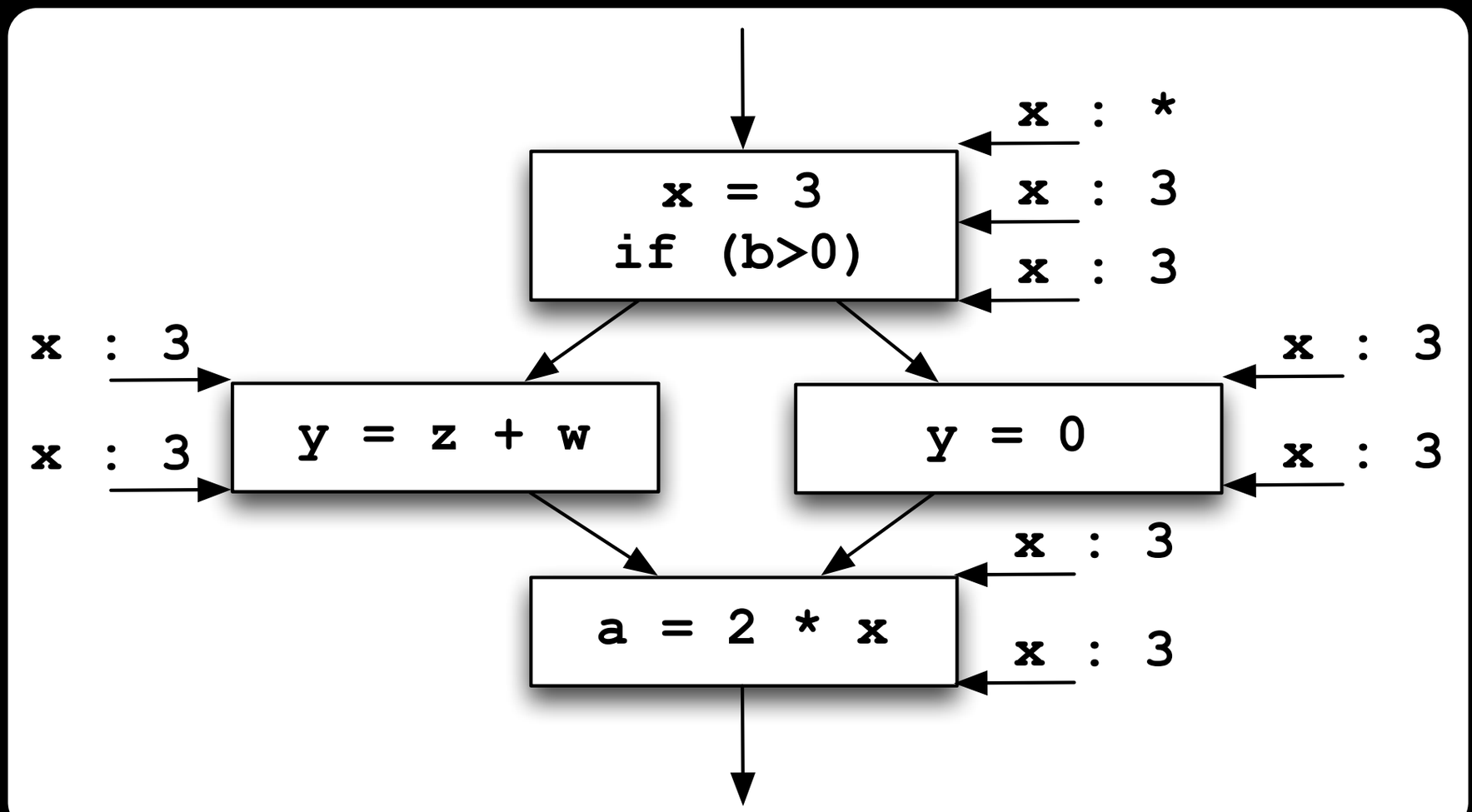
Exemplo

Quais os valores nos pontos **?**

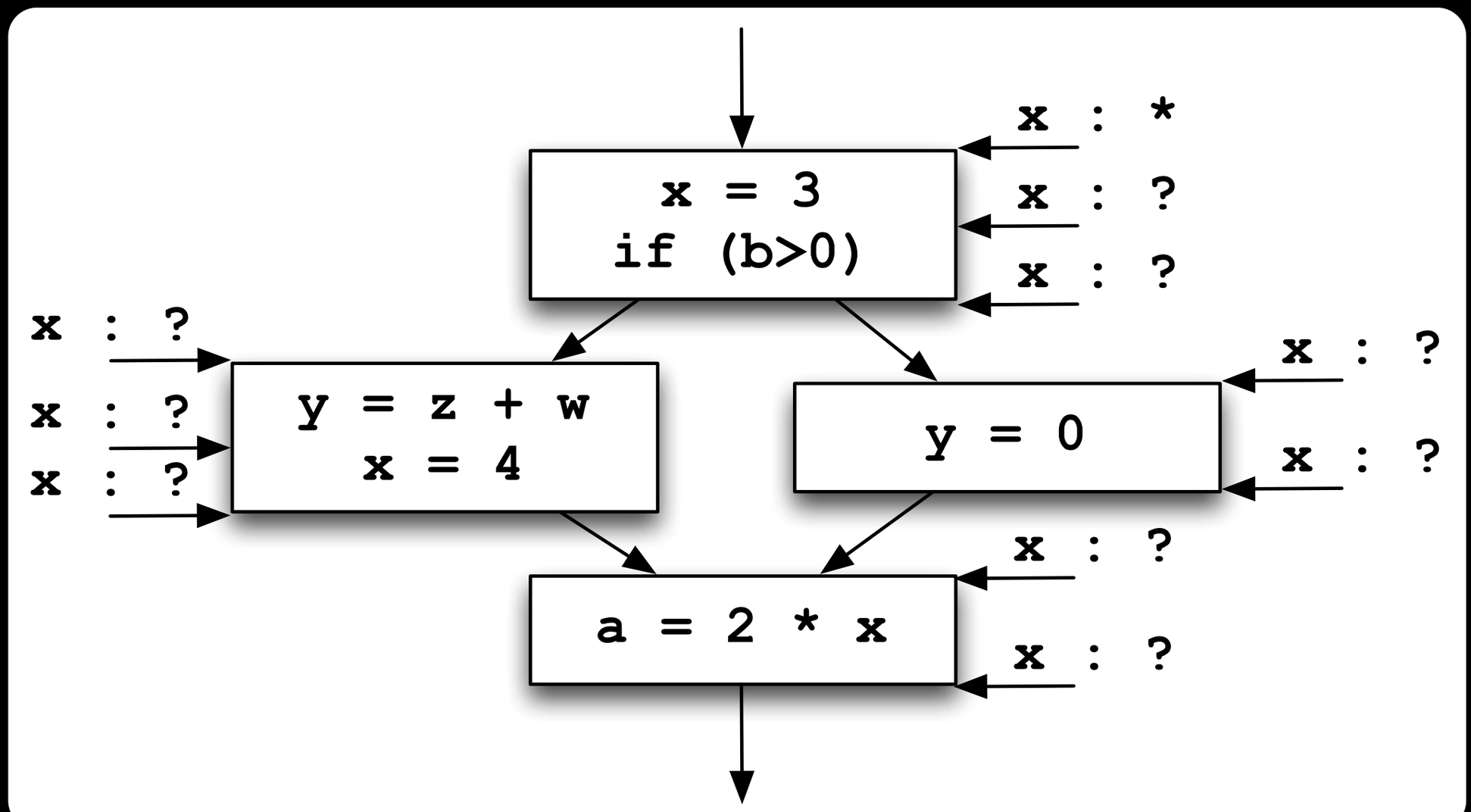


Exemplo

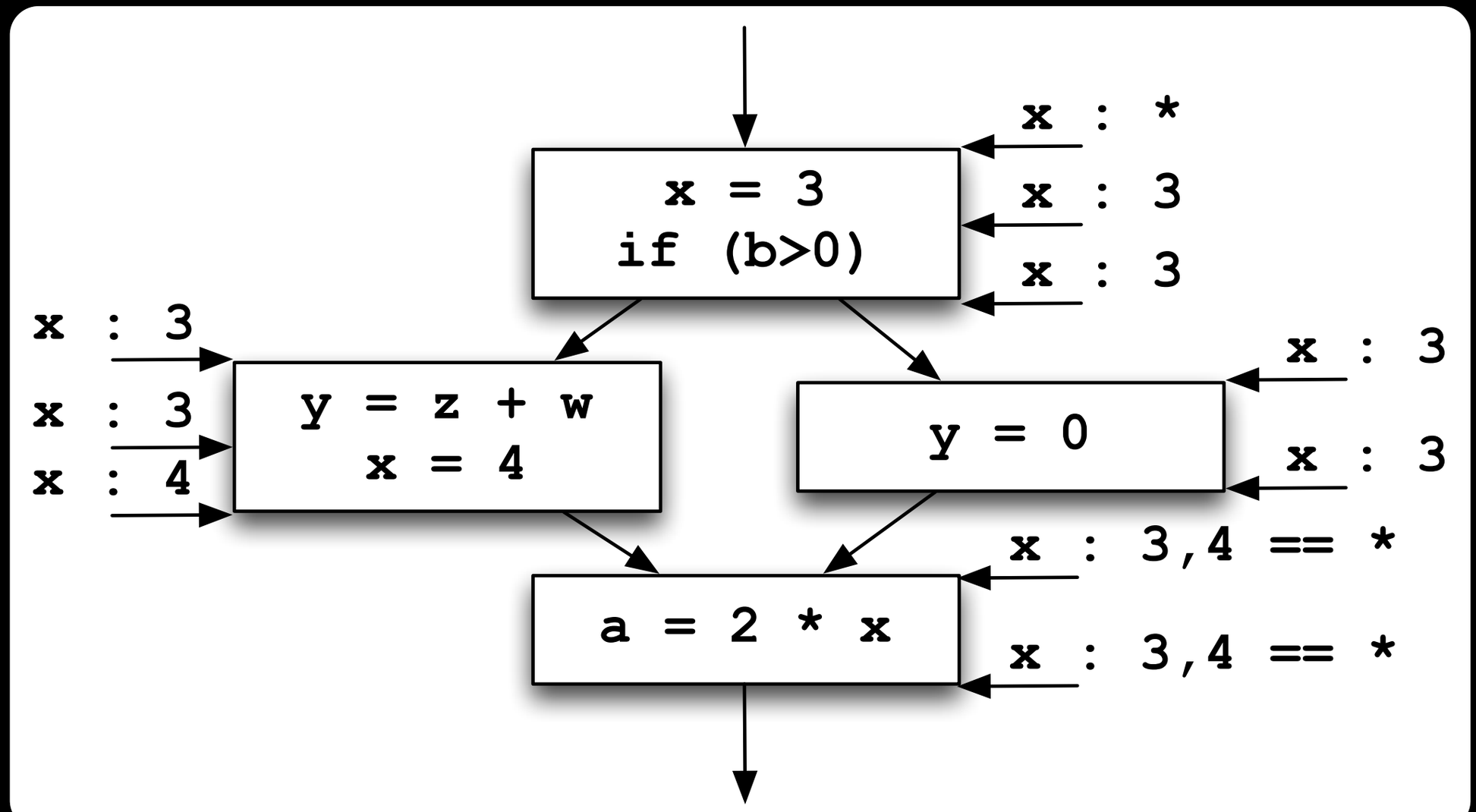
Quais os valores nos pontos **?**



Exemplo



Exemplo



Como realizar a otimização então?

Usando a Informação

- Dado que temos informações globais, é fácil de realizar a otimização
  - basta inspecionar as propriedades **`x = ?`** associadas com instruções que usam **`x`**
  - se **`x`** for constante naquele ponto, substitua o uso de **`x`** pela constante
- Mas como computar as propriedades **`x = ?`**

A análise de um programa pode ser expressa como a combinação de ***regras simples***, relacionando a mudança de informação entre ***instruções adjacentes***

Detalhando

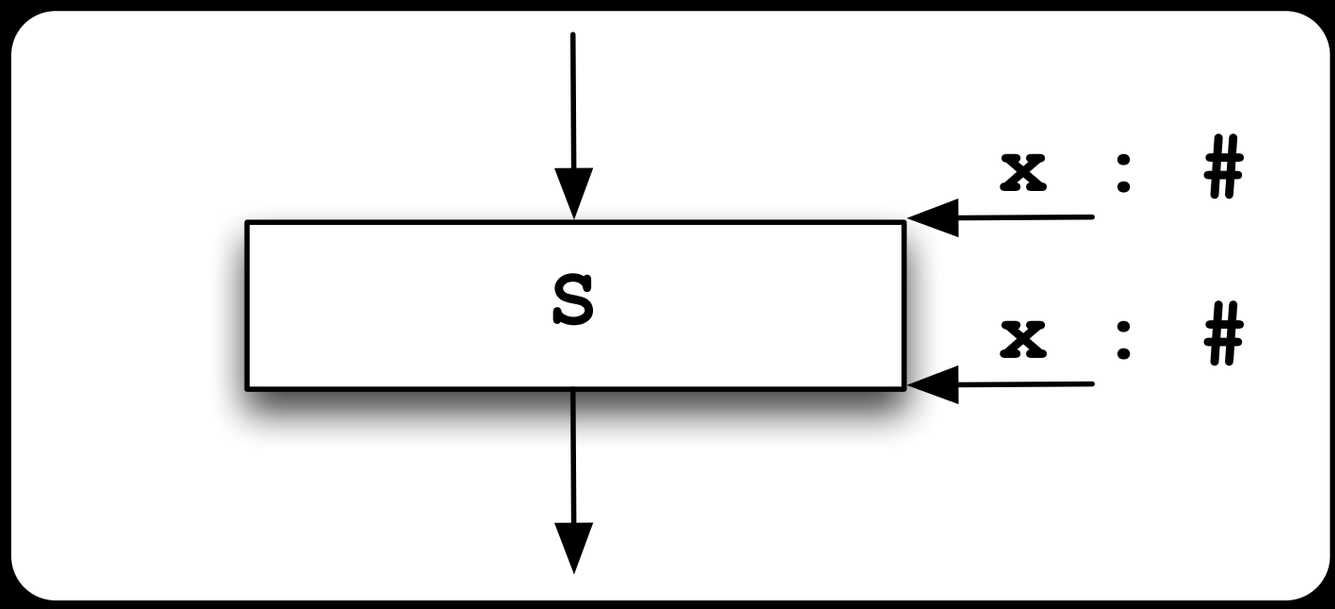
- A ideia é transferir informação de uma instrução para a próxima
- Para cada instrução **`s`**, computamos a informação sobre o valor de **`x`** imediatamente antes e depois de **`s`**
  - **C_{in}`(x,s) =`** `valor de` **`x`** `antes de` **`s`**
  - **C_{out}`(x,s) =`** `valor de` **`x`** `após` **`s`**

Funções de transferência

- transferir informação entre instruções

- definidas por meio de regras

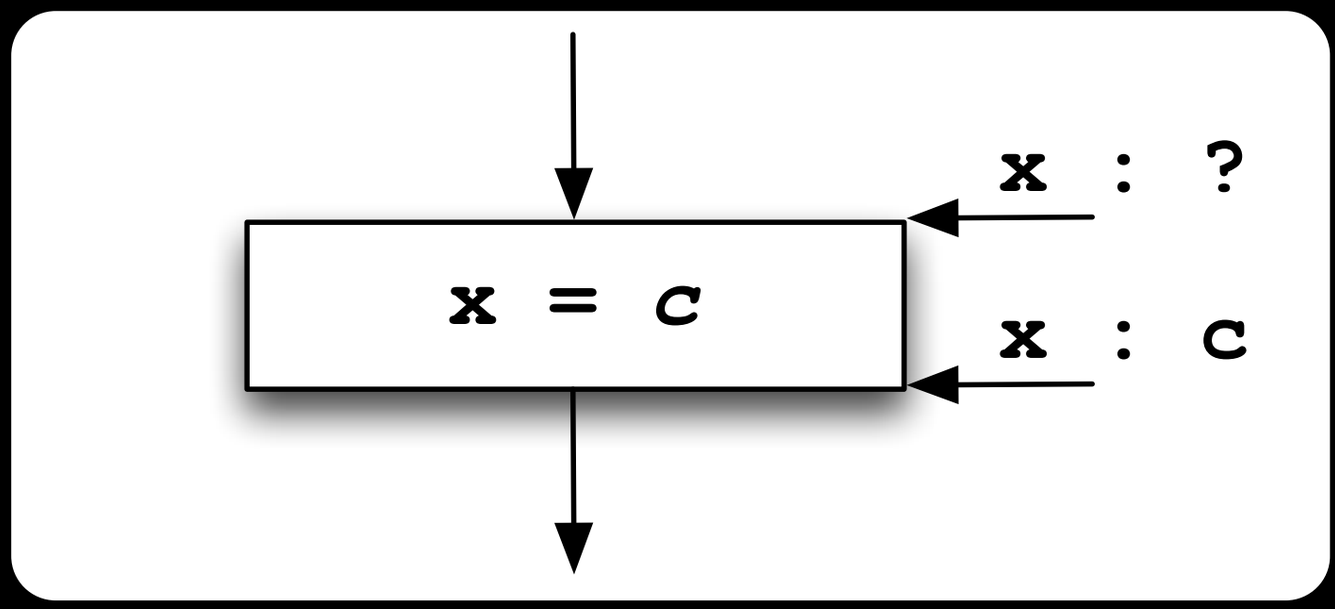
Regra 1



$C_{out}(x,s) = \#,\ \text{se}\ C_{in}(x,s) = \#$

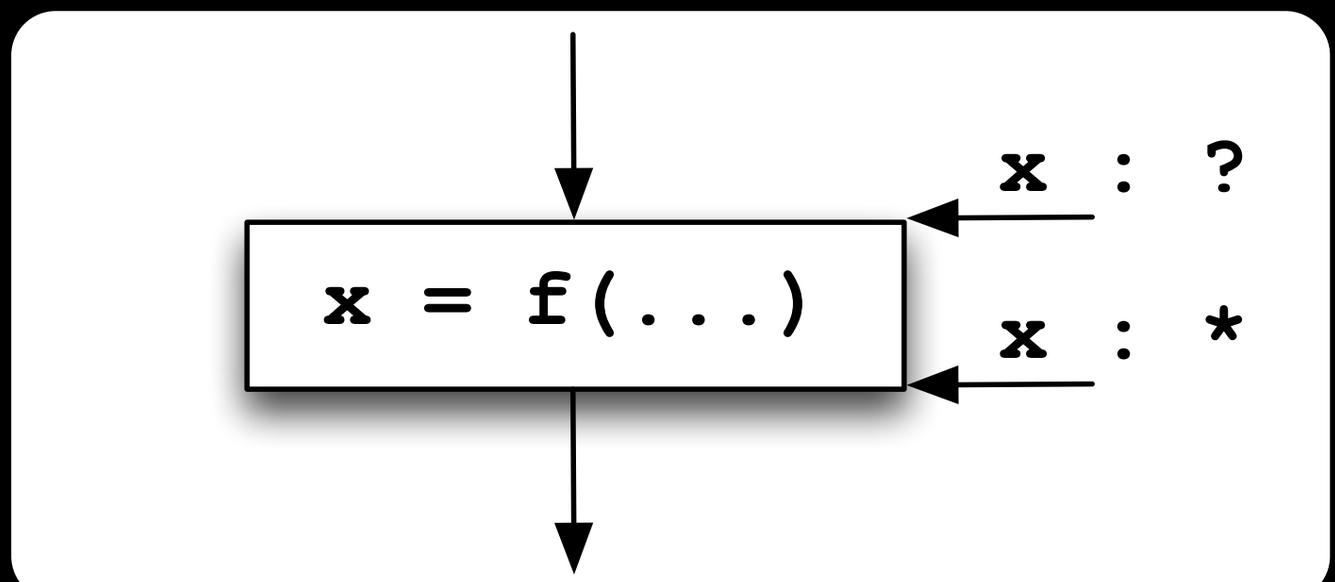
lembre que # é código não alcançado

Regra 2



$C_{out}(x, x:=c) = c$, se c é uma constante

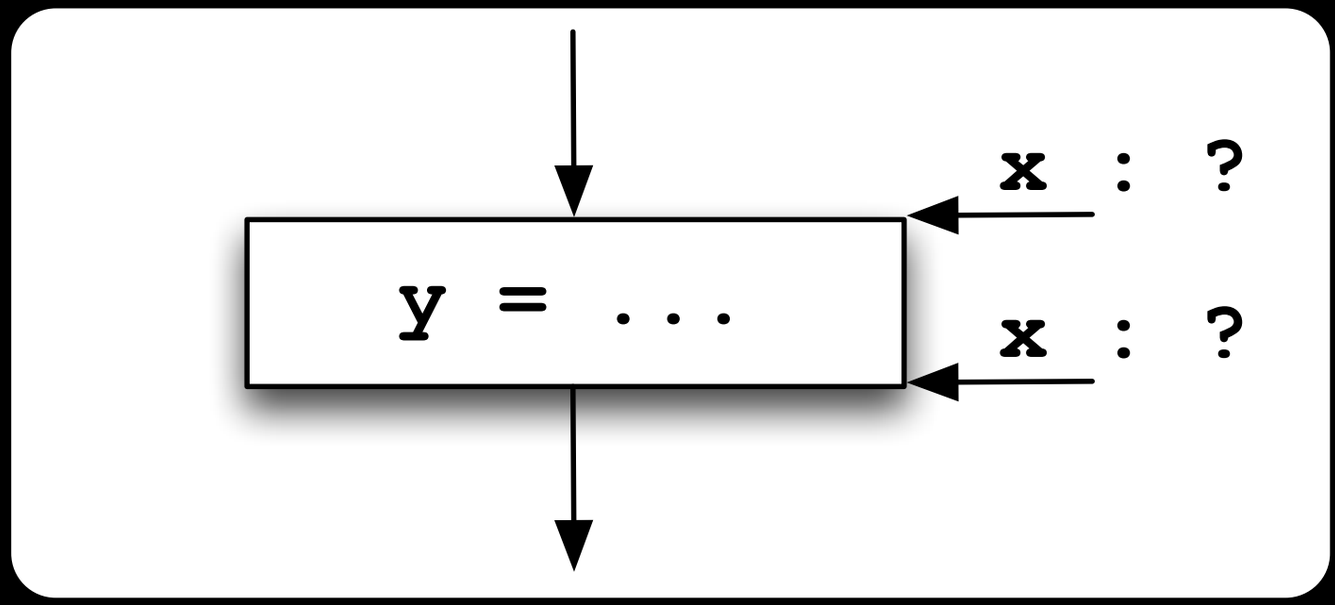
Regra 3



$C_{out}(x, x := f(\ldots)) = *$

*lembre que * é não sei*
aproximação conservadora!

Regra 4



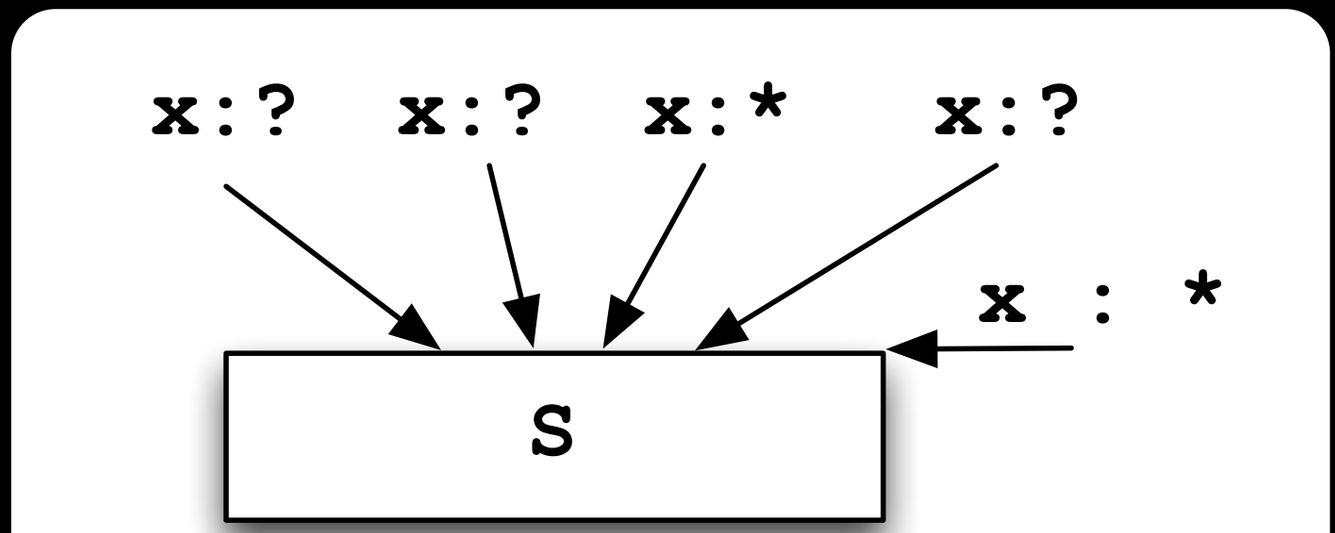
$C_{out}(x, y := \ldots) = C_{in}(x, y := \ldots)$ se $x \neq y$

Resolvido?

Regras 1 a 4

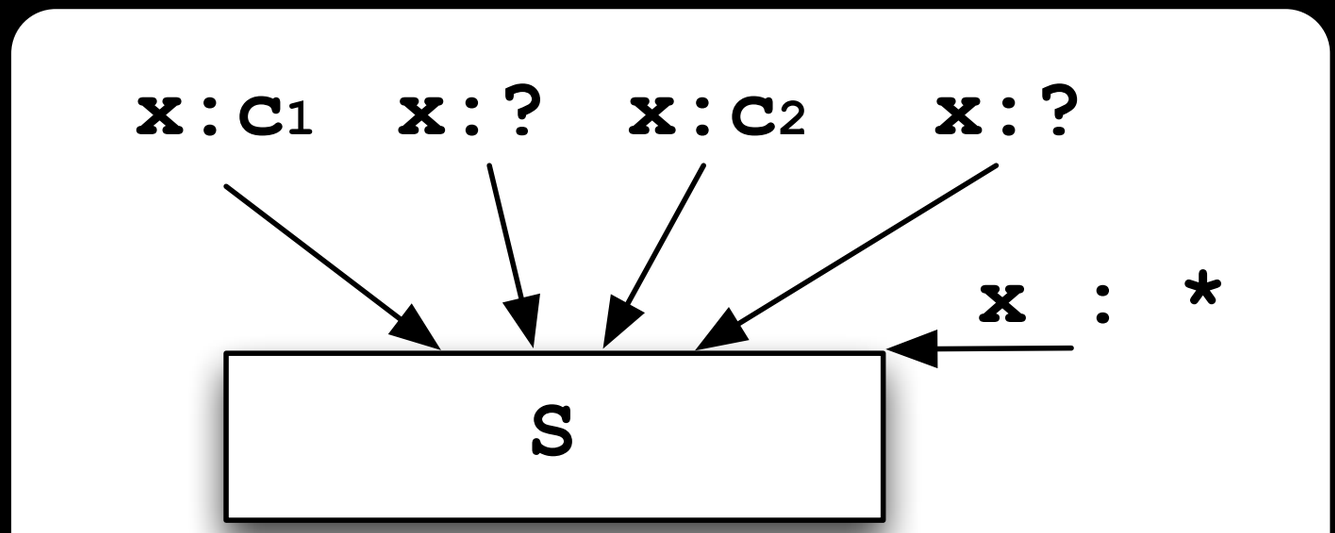
- Relacionam o *in* de um *statement* com o *out* do mesmo *statement*
  - propagam informação entre *statements*
- Precisamos de regras relacionando o *out* de um *statement* com o *in* do *statement* seguinte
  - para propagar informação entre nós do CFG
    - *forward-propagation*
- Nas regras a seguir, *statement s* tem como *statements* imediatamente predecessores p_1, … p_n

Regra 5



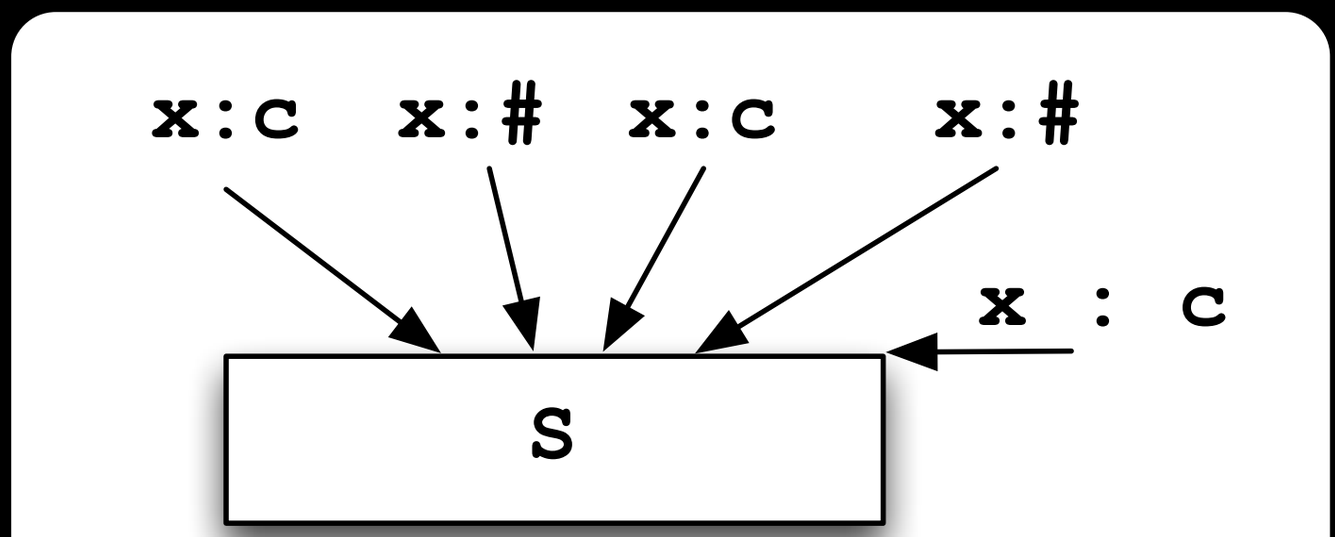
se $C_{out}(x,p_i)=*$ para algum i, $C_{in}(x,s)=*$

Regra 6



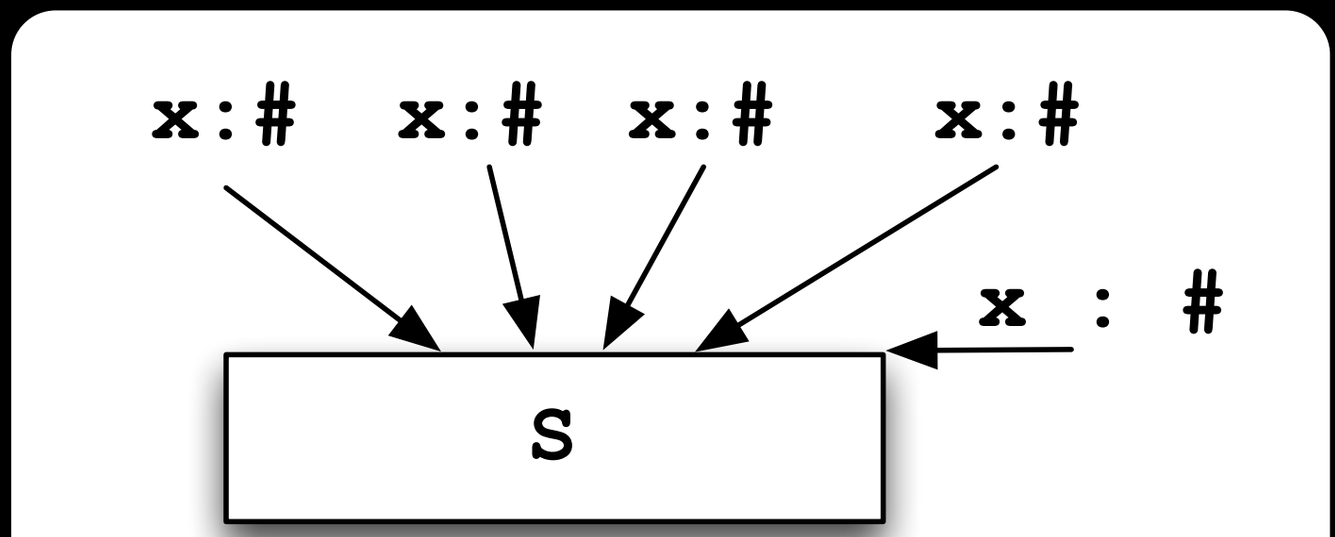
se $C_{out}(x,p_i)=c_1$ e $C_{out}(x,p_j)=c_2$ e $c_1 \neq c_2$,
$C_{in}(x,s)=*$

Regra 7



se $C_{out}(x,p_i)=c$ ou # para todo i,
$C_{in}(x,s)=c$

Regra 8



se $C_{out}(x,p_i)=\#$ para todo i,
$C_{in}(x,s)=\#$

Algoritmo

- Para todo nó inicial (*statement*) s do programa, defina $\mathbf{C_{in}(x,s)=*}$
- Em todos os demais pontos do programa, defina $\mathbf{C_{in}(x,s) = C_{out}(x,s) = \#}$
- Repita o processo abaixo até que a aplicação das regras 1-8 não produza alteração
  - dado um *statement* que não satisfaça 1-8, atualize usando a regra apropriada

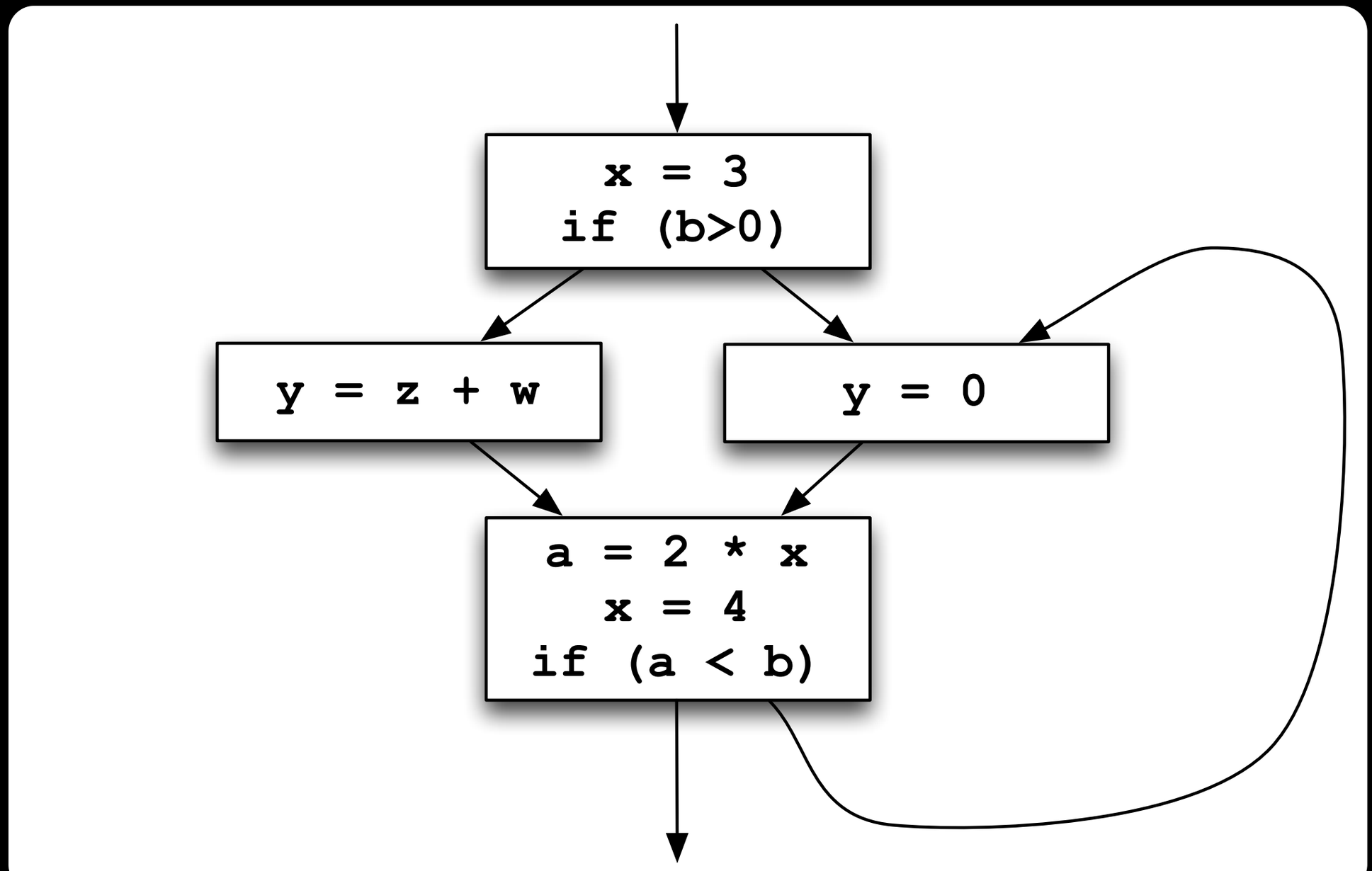
O valor inicial

- Por conta de ciclos, todos os pontos devem conter valores em todos os momentos durante a análise

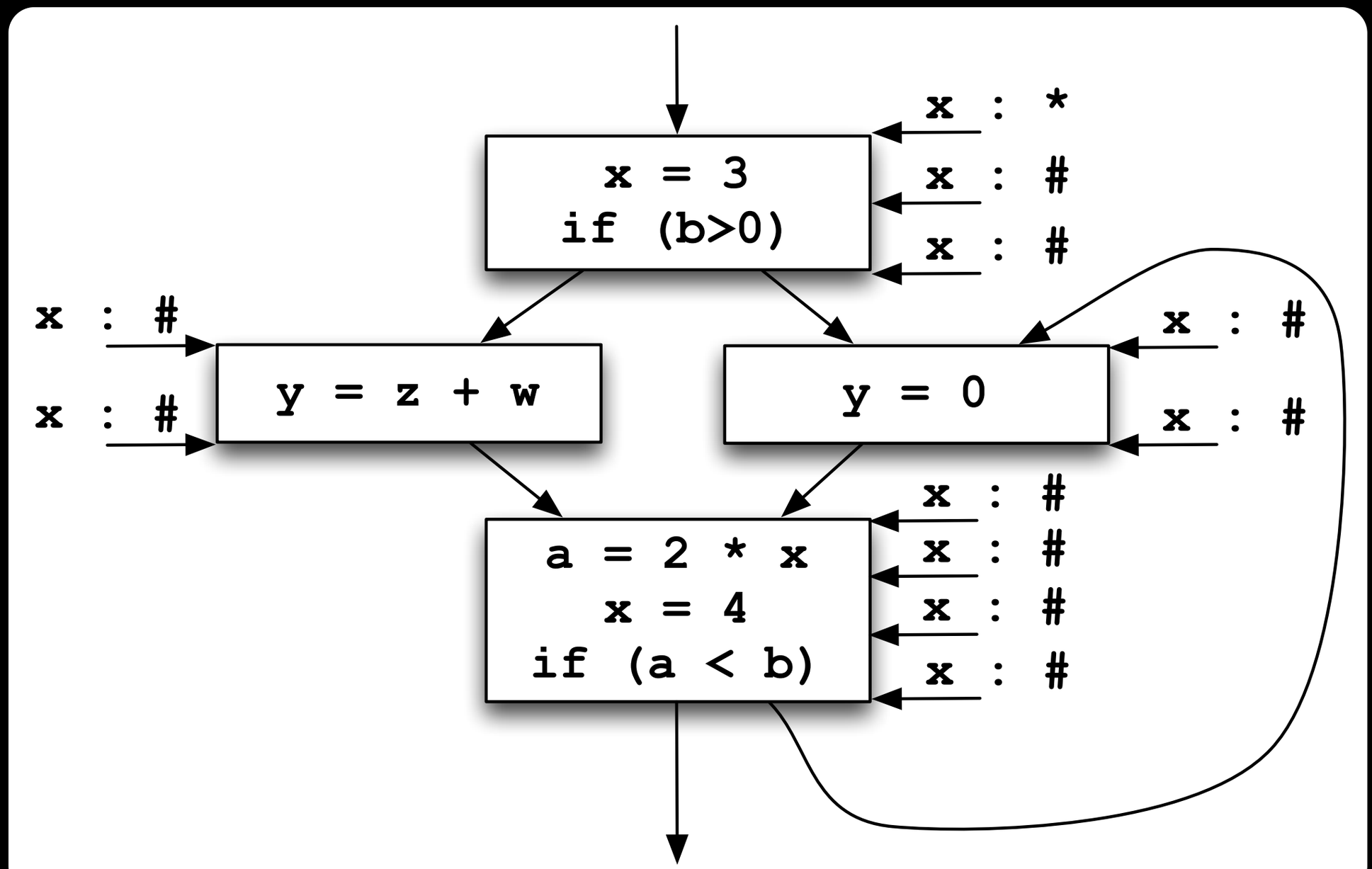
- Definir valores iniciais permite que a análise alcance um ponto fixo

- O valor inicial # significa "*até o momento, com o que sabemos, controle não alcança este ponto*"

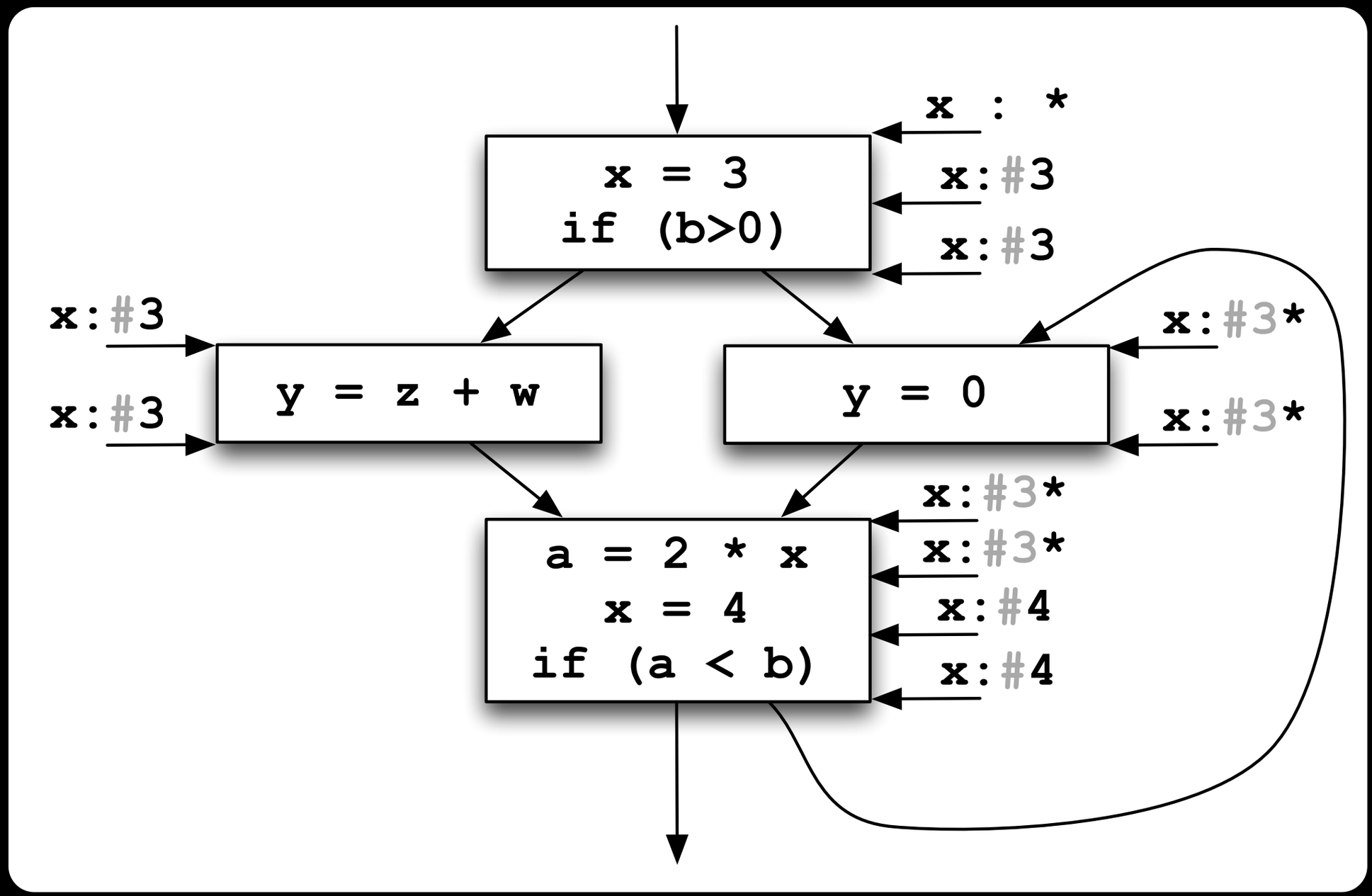
Exemplo



Exemplo



Exemplo



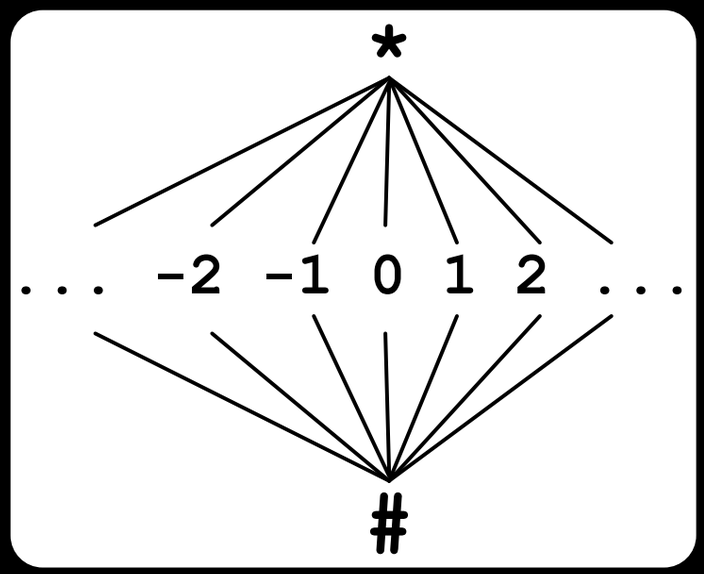
continuamos até que as regras estejam satisfeitas

Relações de ordem

- Podemos simplificar a apresentação da análise ordenando os valores

$$\# < c < *$$

- Podemos ilustrar estes valores e sua relação de ordem, por meio de reticulados (*lattices*):



Relações de ordem

- ***** é o maior valor, **#** é o menor valor
  - todas as constantes entre estes dois valores são incomparáveis (para efeitos desta análise)
- Seja **lub** o *least-upper bound* nesta relação de ordem
- As regras 5-8 podem ser escritas como:

$$\mathbf{C_{in}(x,s) = lub\{C_{out}(x,p) \mid p}\ \textit{é um predecessor de}\ \mathbf{s\}}$$

Termination

- Simplesmente afirmar ‘repita até que nada mude’ não garante que eventualmente nada realmente irá mudar
- O uso de **`lub`** explica a razão do algoritmo eventualmente terminar
  - os valores iniciam em **`#`** e apenas *aumentam*
  - **`#`** pode virar constante, e constante vira **`*`**
  - Portanto, **`C_(x,s)`** muda no máximo duas vezes