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

Prof. Sandro Rigo

www.ic.unicamp.br/~sandro

Partial Redundancy Elimination

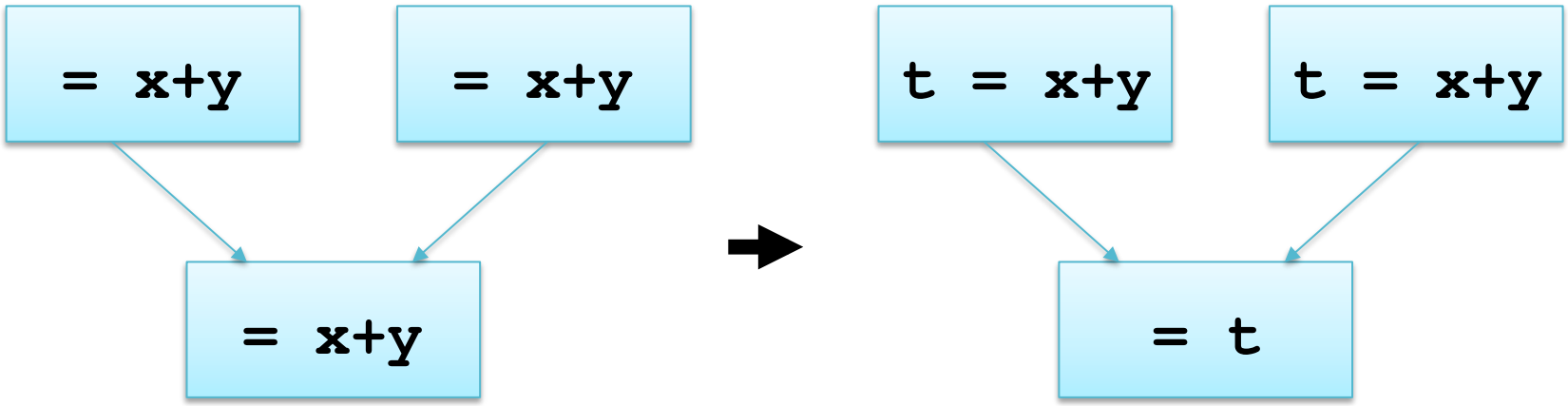
Fontes de Redundância

- Subexpressões comuns (GCSE)
- Expressões invariantes de laço
- Expressões parcialmente redundantes

GCSE

- A expressão b+c é (completamente) redundante no ponto p se ela é uma expressão disponível (como visto antes).
  - Ou seja, ela é computada em todo caminho que alcança p e b ou c não é modificado após a computação.

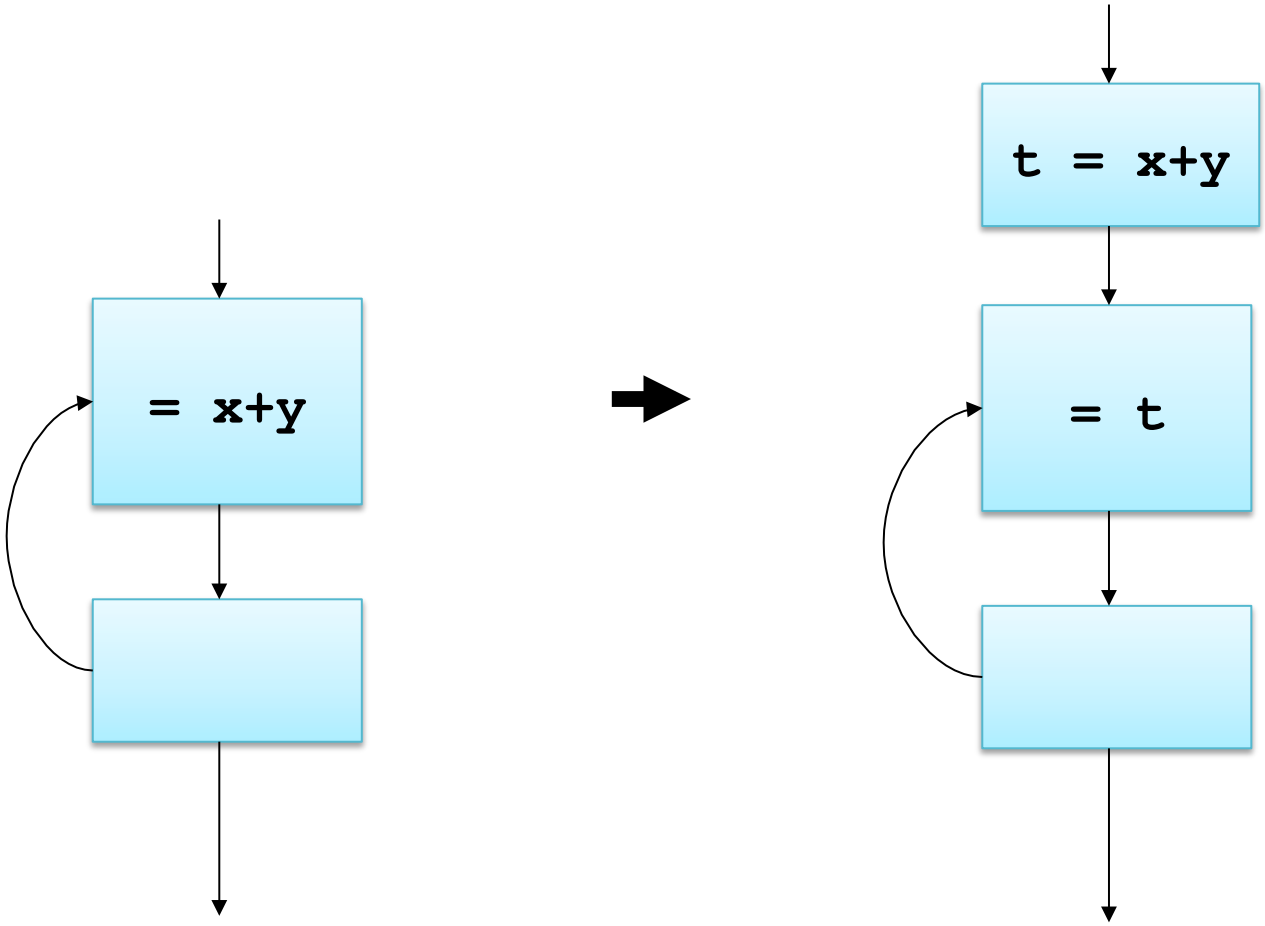
Exemplo: *Common Subexpression Elimination*



LICM – *Loop-invariant code motion*

- Expressão invariante de laço: computa sempre o mesmo valor dentro do laço, pois os operandos não são modificados no laço.
  - Podemos mover a expressão para fora do laço para reduzir a frequência com que é executada.
- Diferente de CSE, a expressão não pode ser simplesmente removida, ela tem que ser movida para fora do laço.

Exemplo: *Loop-Invariant Code Motion*



LICM – *Loop-invariant code motion*

- Cuidado para não executar uma instrução que não seria executada. Pode modificar o comportamento do programa.

```
while (c) {
  S;
}
```

LICM – *Loop-invariant code motion*

- Cuidado para não executar uma instrução que não seria executada. Pode modificar o comportamento do programa.

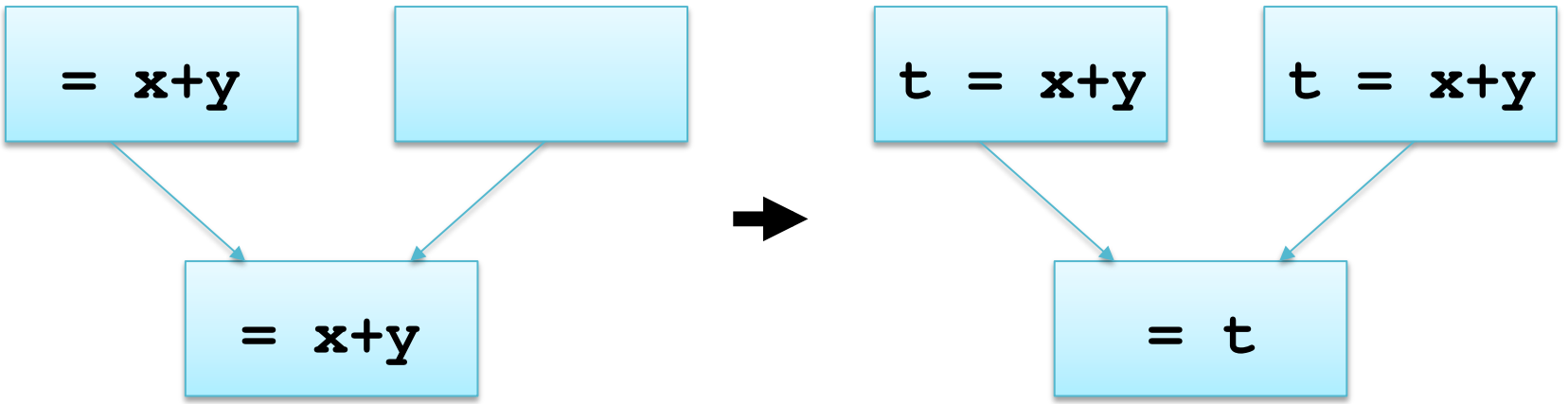
```
while (c) {
  S;
}
```

Loop Inversion ➔

```
if (c) {
    do {
        S;
    } while (c);
}
```

Redundância Parcial

- Redundância parcial = redundância em apenas alguns caminhos do CFG

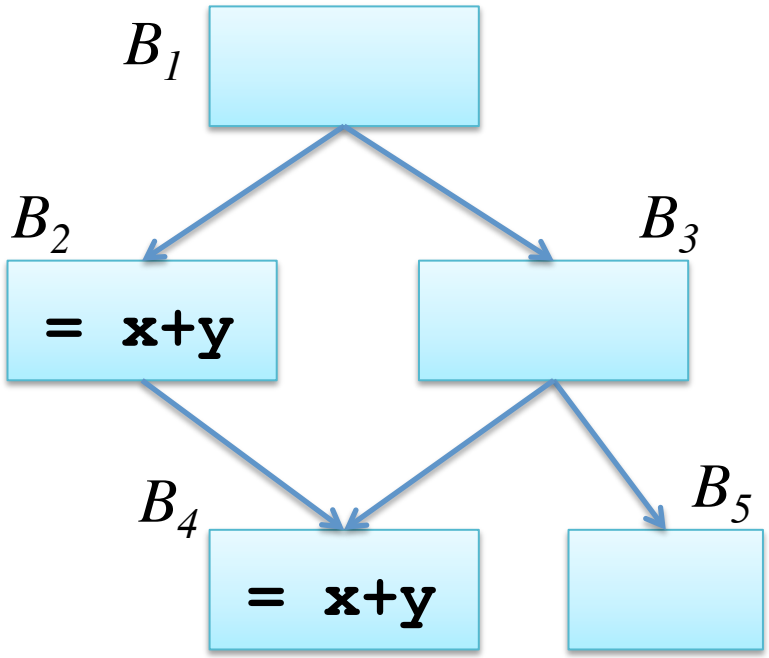


PRE – *Partial Redundancy Elimination*

- Remove redundância parcial.
- Remove redundância total também. GCSE e LICM são casos especiais de PRE.
  - PRE realiza ambos e mais um pouco...
- Assim como LICM, move a computação de expressões para outros lugares no CFG para diminuir o número de vezes que a computação é realizada.
- Otimização comum em compiladores modernos.

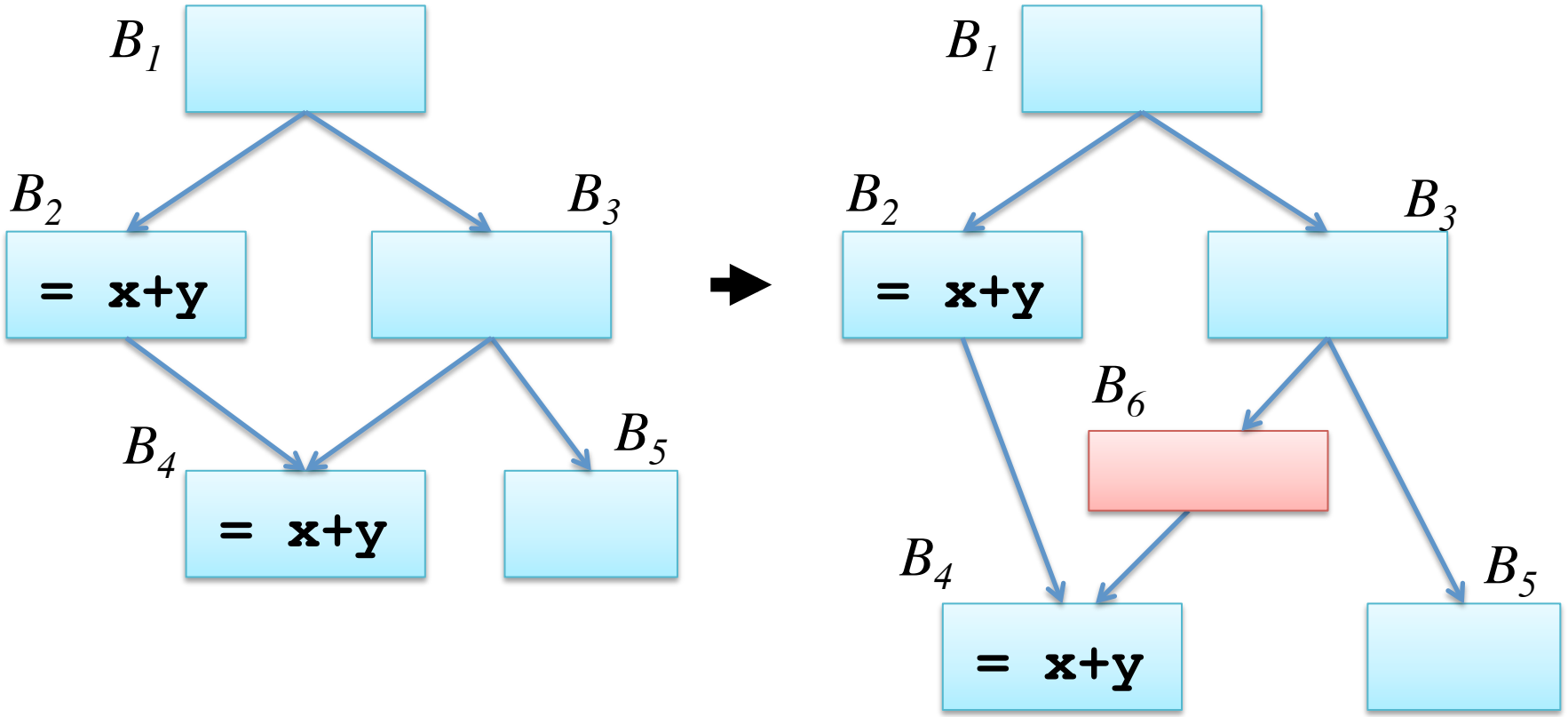
Redundância Parcial

- Qualquer tipo de redundância parcial pode ser removida?

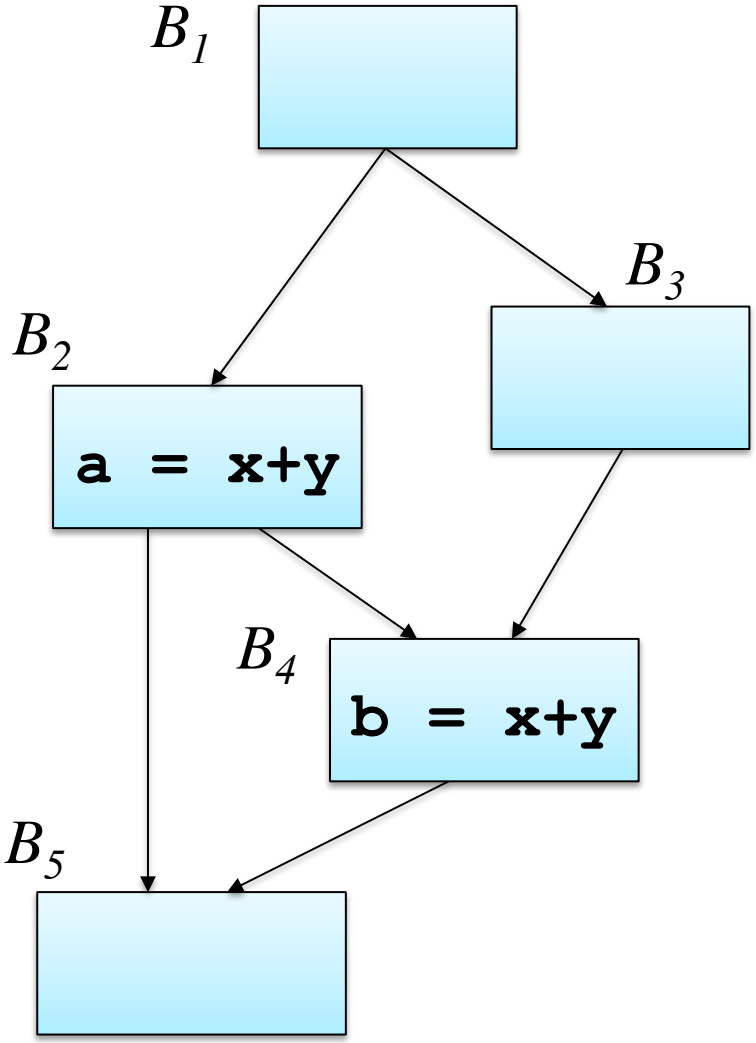


Redundância Parcial

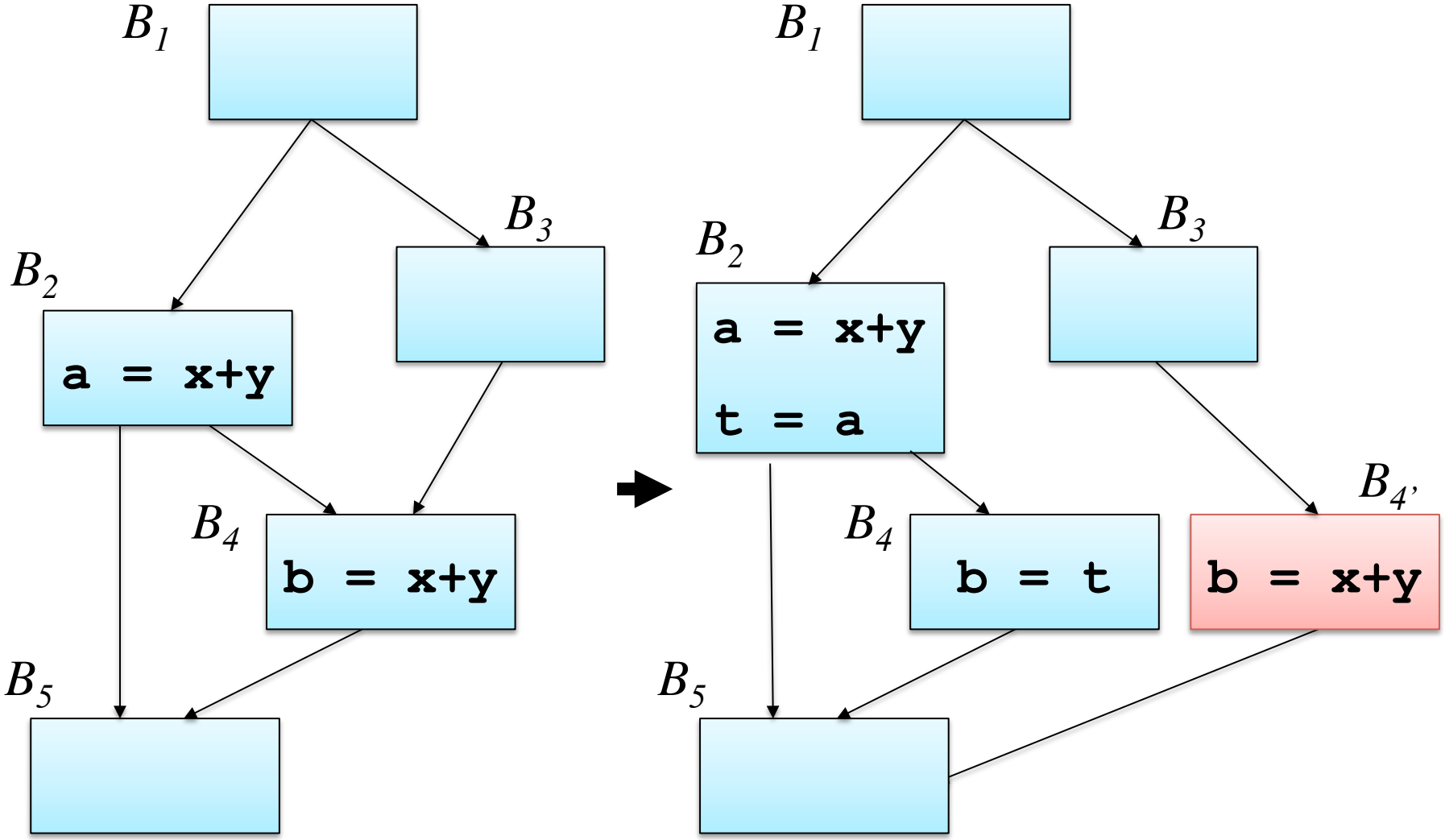
- Não, a não ser que possamos modificar o CFG.



Exemplo 2



Exemplo 2: Duplicação de blocos



Modificando o CFG

1. Adicionar um bloco novo em uma aresta.
   - Só é necessário se for uma *aresta crítica*: Bloco origem possui múltiplos sucessores e o bloco alvo múltiplos predecessores.

2. Duplicar blocos de forma que a expressão `x +y` seja computada apenas nos caminhos onde é usada.

Problemas com Duplicação de blocos

- Número de caminhos no CFG é exponencial no número de saltos condicionais: Duplicação de caminhos/Blocos pode aumentar o número de blocos exponencialmente.

- Vamos permitir adicionar blocos nas arestas críticas, mas não vamos duplicar blocos para permitir a remoção de computações redundantes.

Lazy-Code-Motion

- Propriedades desejáveis de um algoritmo de PRE:

1. Remover toda computação redundante sem a necessidade de duplicar código.
2. O código resultante da otimização não realiza nenhuma computação que não está na execução do programa original.
3. Expressões são computadas no último momento possível (*Lazy*).

Lazy-Code-Motion

- Propriedades desejáveis de um algoritmo de PRE:

1. Remover toda computação redundante sem a necessidade de duplicar código.
2. O código resultante da otimização não realiza nenhuma computação que não está na execução do programa original.
3. Expressões são computadas no último momento possível (*Lazy*).
   - *O objetivo desta propriedade é encurtar o tempo de vida dos temporários para reduzir a pressão no alocador de registradores*.

Lazy-Code-Motion

- **Redundância total**:
  - Uma expressão ***e*** é redundante em B se foi computada por todos os caminhos alcançando B, e seus operandos não foram redefinidos

  Em outras palavras ...

  - Seja S o conjunto de blocos que contem as expressões redundantes à expressão ***e.***
    - As arestas saindo de S formam um conjunto de corte entre a entrada do CFG e a expressão redundante ***e***.

Lazy-Code-Motion

- **Redundância parcial**:
  - O conjunto de blocos que contem as expressões redundantes à expressão ***e*** não é suficiente para formar um conjunto de corte entre a entrada do CFG e a expressão redundante ***e***.
  - Existem caminhos entre a entrada e a expressão ***e*** que não computam ***e***.
  - Tentaremos tornar a expressão ***e*** totalmente redundante ...

Lazy-Code-Motion

- O algoritmo *Lazy-Code-Motion* tenta introduzir cópias de *e* no CFG para tornar a expressão *e* totalmente redundante.

1. Descobrir os pontos próximo ao início do CFG para onde podemos mover as expressões (*Anticipated Expressions*)
2. Verificar se a expressão já está disponível nos pontos onde ela é "anticipável". Se já estiver disponível não há necessidade de introduzir uma cópia.
3. Computar os pontos nos quais as expressões são "postergáveis" (*postponable*). E introduzir as cópias das expressões o mais tarde possível.
4. Eliminar atribuições desnecessárias à variáveis temporárias.

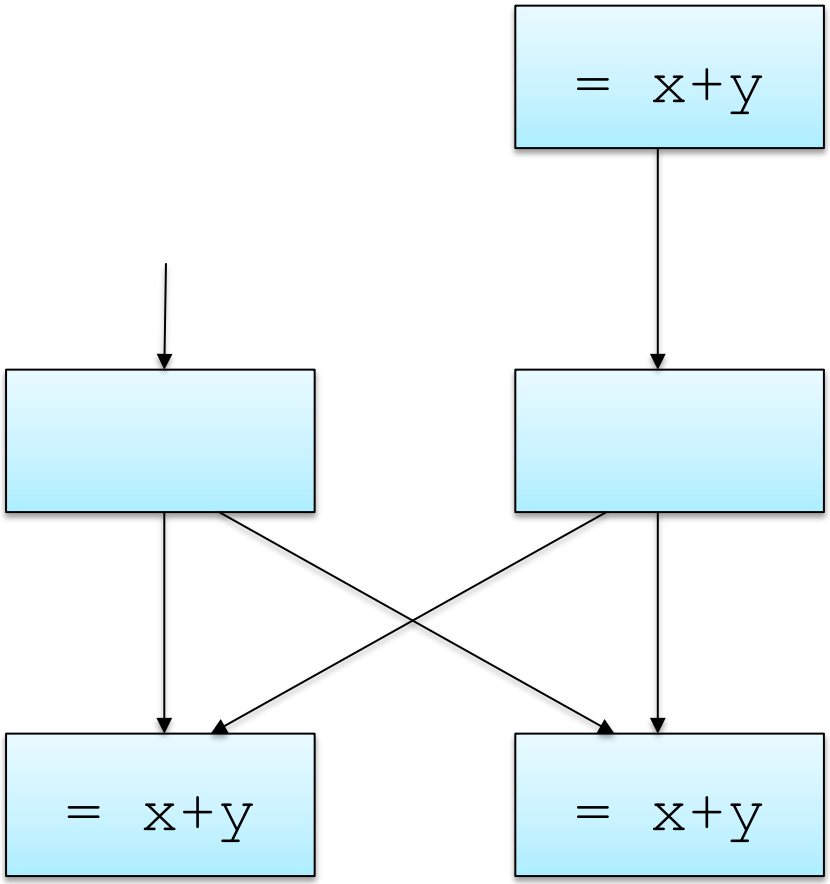
Lazy-Code-Motion

- Vamos usar 4 análises de fluxo de dados, uma para cada passo.
- Cada análise utiliza o resultado da anterior.

1- *Anticipated Expressions*

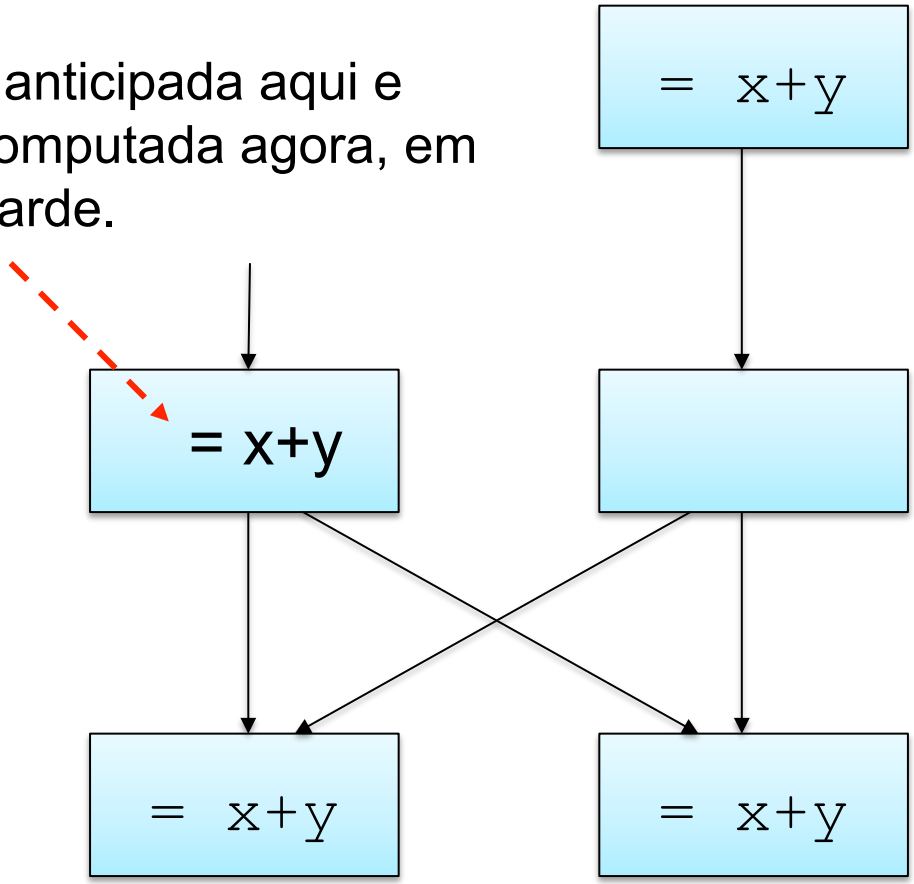
- A expressão `x+y` é anticipável no ponto **p** se `x+y` será certamente computada antes de x ou y serem modificados em qualquer caminho partindo de p.

Exemplo: *Anticipated Expressions*



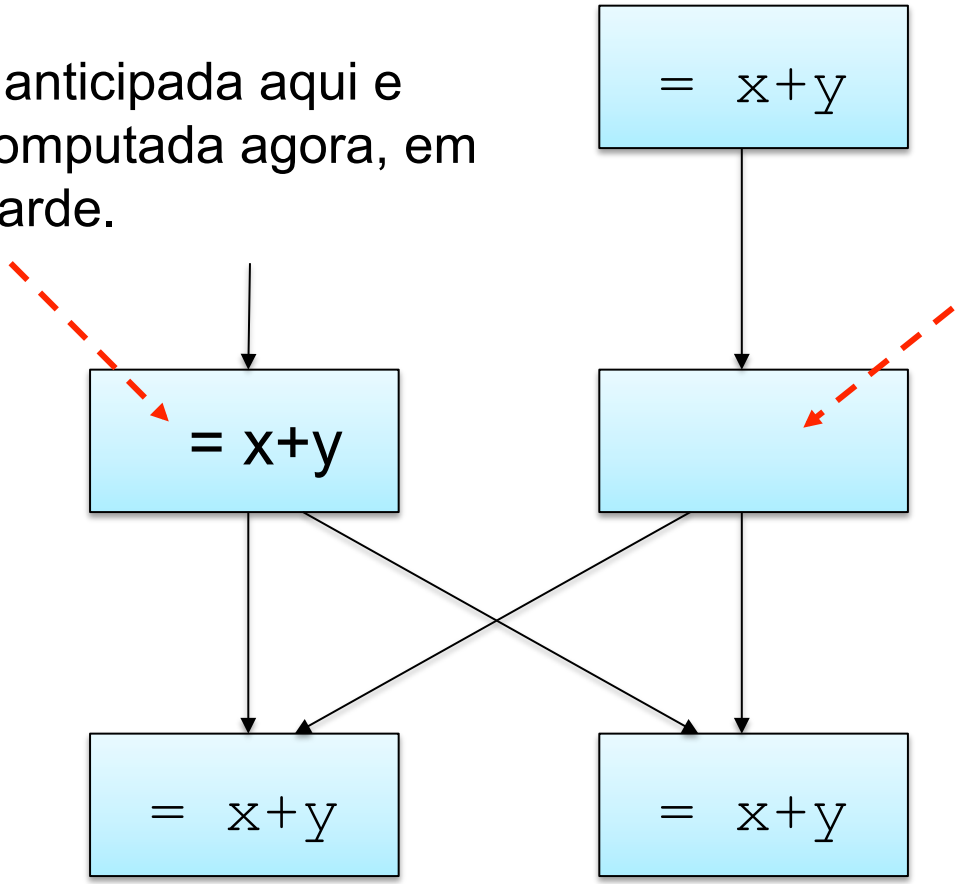
Exemplo: *Anticipated Expressions*

x+y pode ser anticipada aqui e poderia ser computada agora, em vez de mais tarde.



Exemplo: *Anticipated Expressions*

x+y pode ser anticipada aqui e poderia ser computada agora, em vez de mais tarde.



x+y pode ser anticipada aqui também, mas como ela já está disponível aqui, nenhuma computação é necessária.

Computando *Anticipated Expressions*

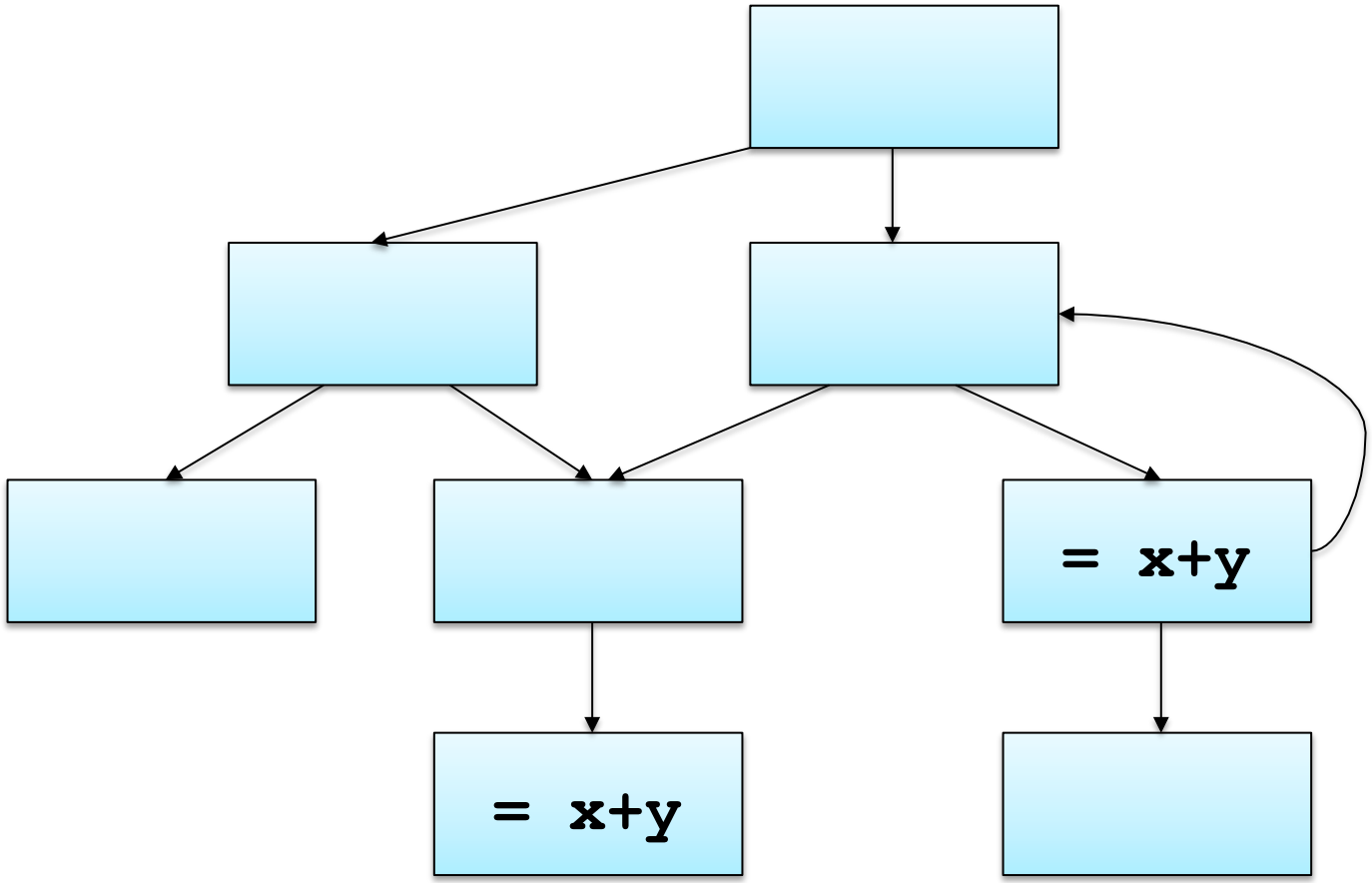
- e_use_B = conjunto de expressões x+y computadas em B antes de qualquer atribuição a x ou y.

- e_kill_B= conjunto de expressões que usem variáveis moficadas no bloco B.

Computando *Anticipated Expressions*

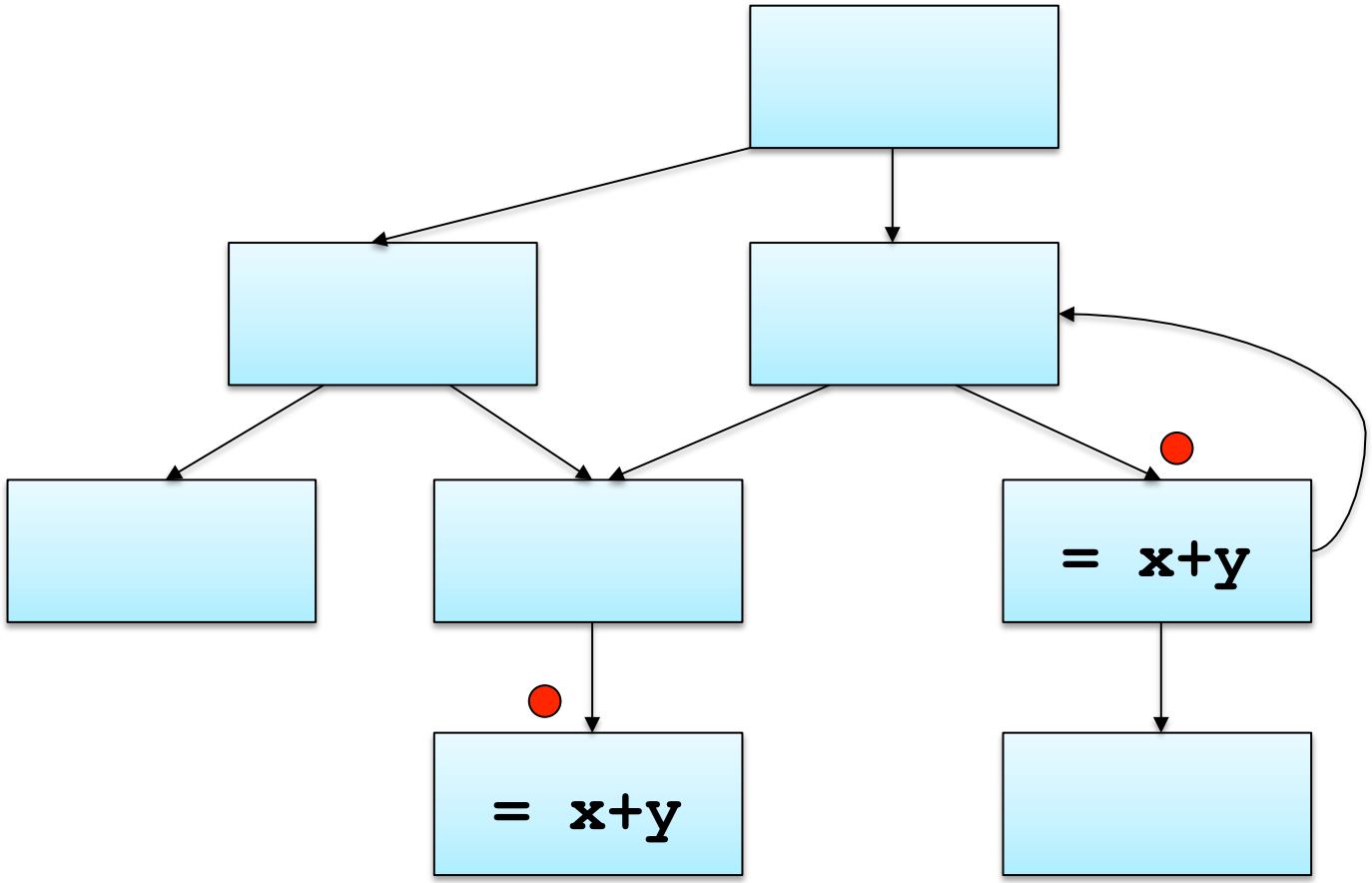
- DFA
  - Direção: *backwards*.
  - Inicialização:
    - IN[EXIT] = {}
    - IN[B] = $\{e_1, e_2, \ldots, e_N\}$
  - Equações
    - IN[B] = $e\_use_B \cup (\text{OUT[B]} - e\_kill_B)$
    - OUT[B] = $\cap_{S,\ succ(B)}$ IN[S]

Exemplo: *Anticipated Expressions*



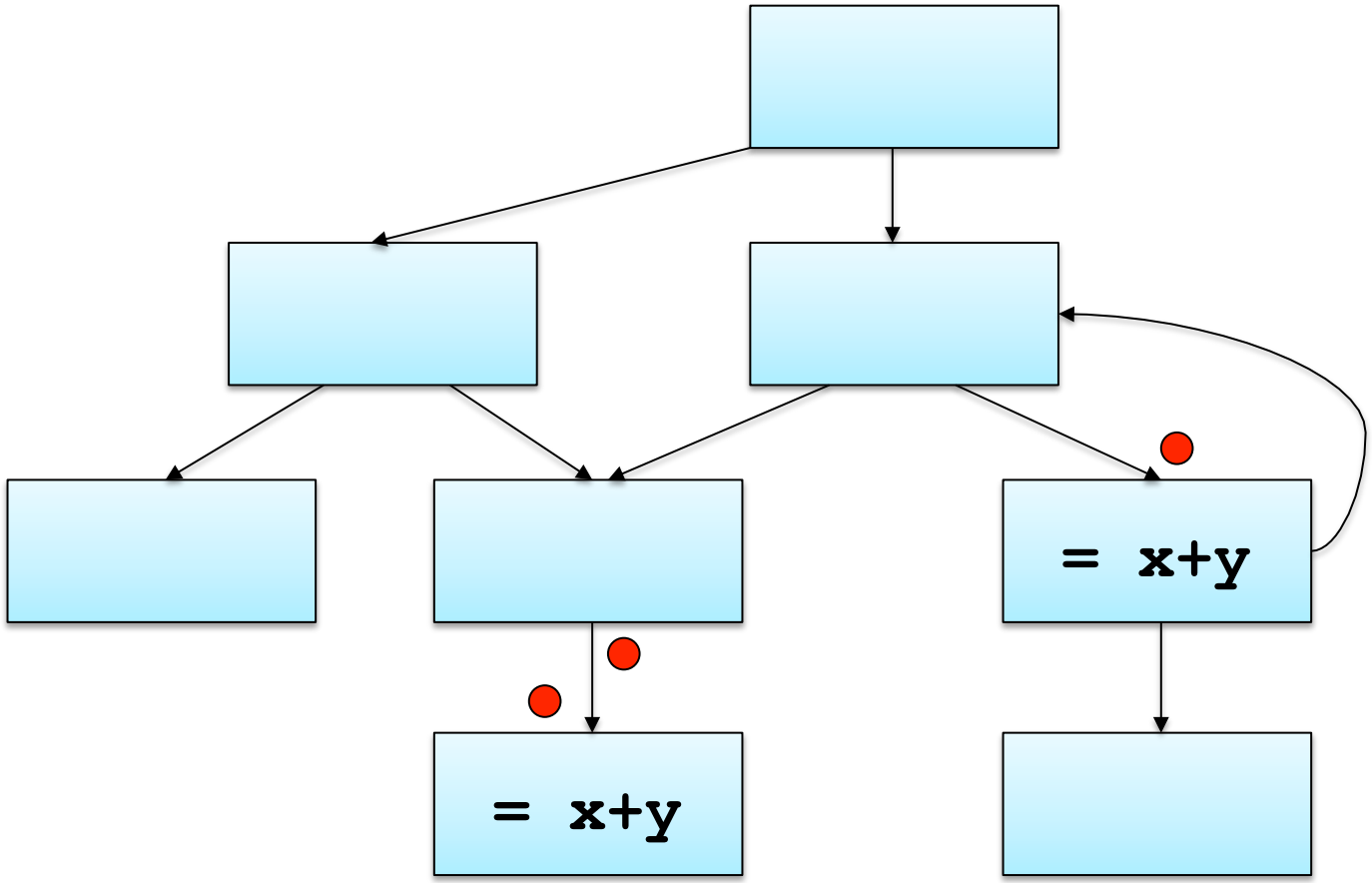
Exemplo: *Anticipated Expressions*

● Anticipated



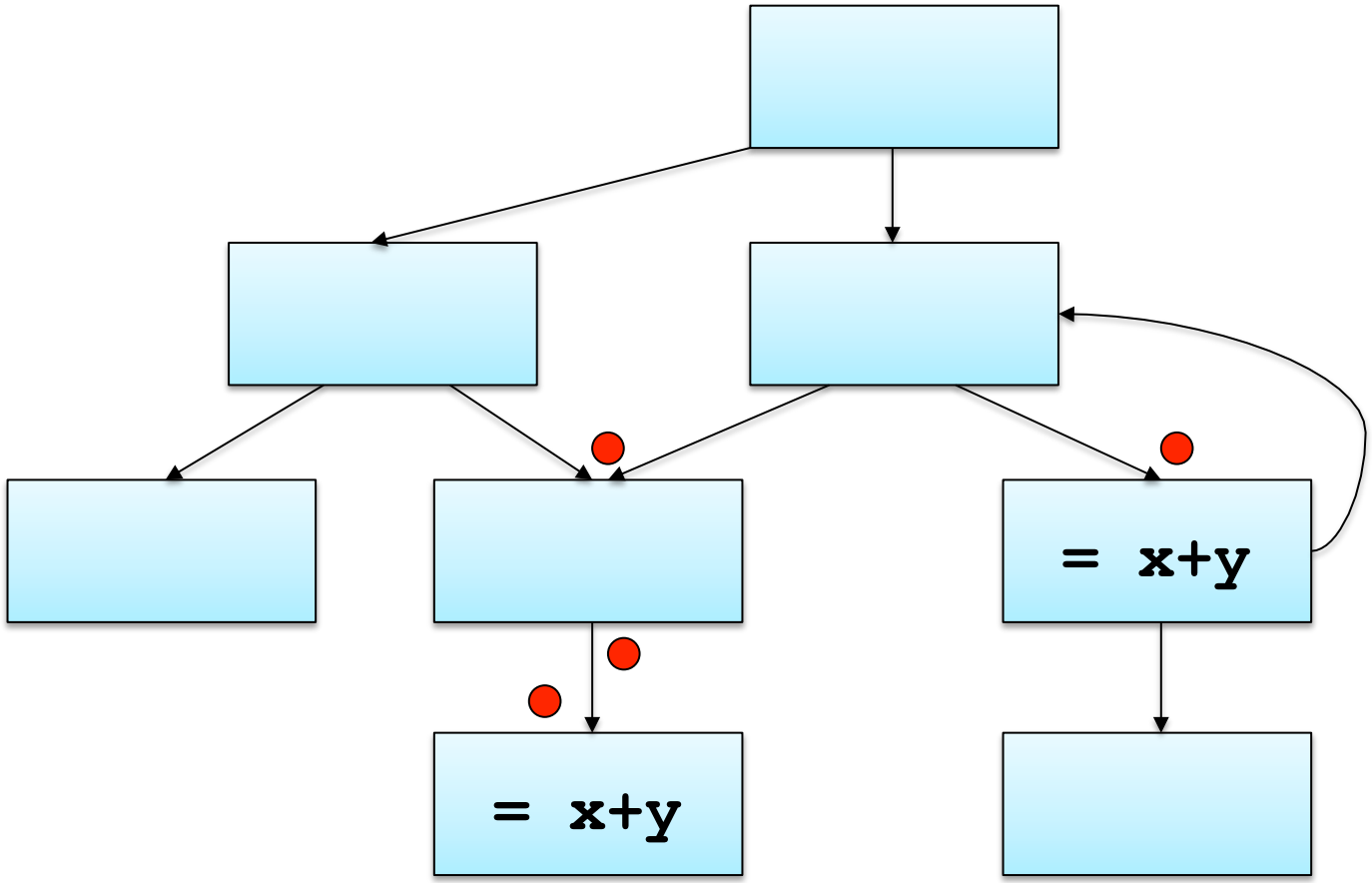
Exemplo: *Anticipated Expressions*

● Anticipated



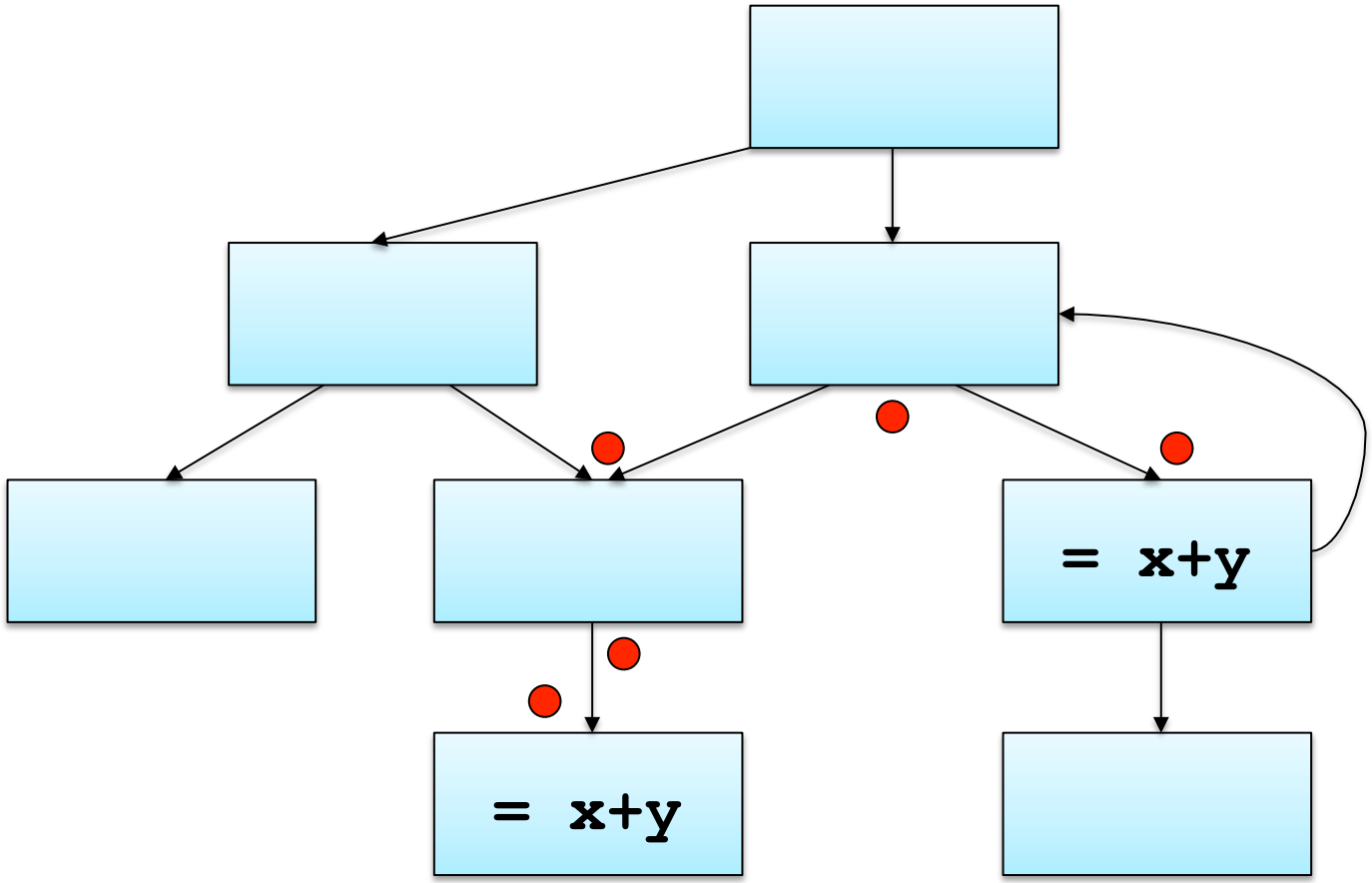
Exemplo: *Anticipated Expressions*

● Anticipated



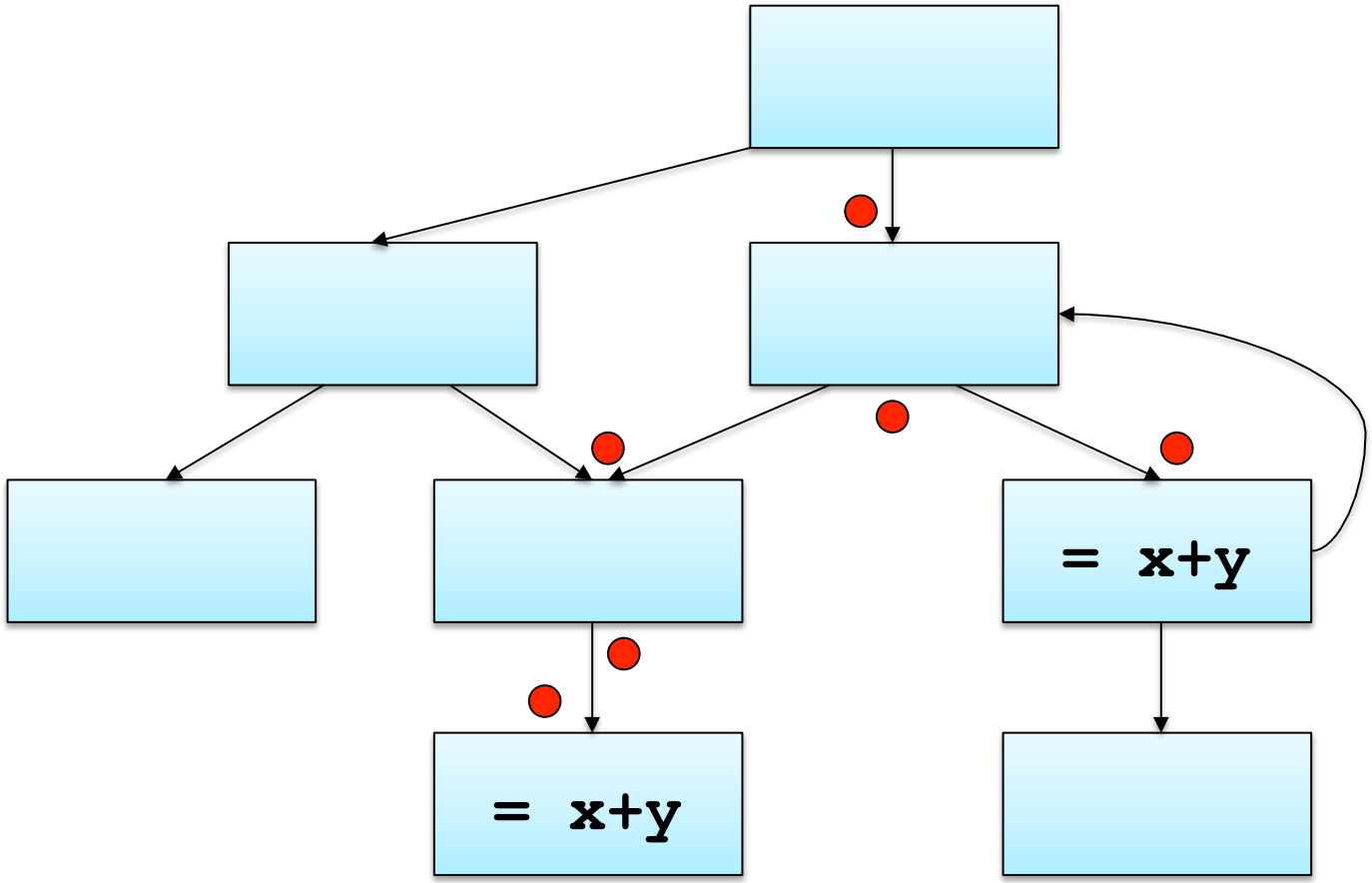
Exemplo: *Anticipated Expressions*

● Anticipated



Exemplo: *Anticipated Expressions*

● Anticipated



2-Expressões “Disponíveis”

- Modificação da análise de expressões disponíveis vista anteriormente.
- x+y está disponível na saída de B se:

1. Se:
    1. Está disponível como visto antes. Ou seja, está em in[B] e não foi morta; ou
    2. Ela é anticipável em *in[B]*. Ou seja, ela “poderia” estar disponível se nós decidirmos computá-la aqui.
2. E não foi morta em B

2-Expressões "Disponíveis"

- x+y está em e_kill_B se x ou y for modificada e x+y não é recomputada após a modificação.
- DFA
  - Direção: *forwards*.
  - Inicialização:
    - OUT[ENTRY] = {}
    - OUT[B] = $\{e_1, e_2, \ldots, e_N\}$
  - Equações
    - OUT[B] = $(anticipated[B].in \cup IN[B]) - e\_kill_B$
    - IN[B] = $\cap_{P,\ pred(B)}$ OUT[P]

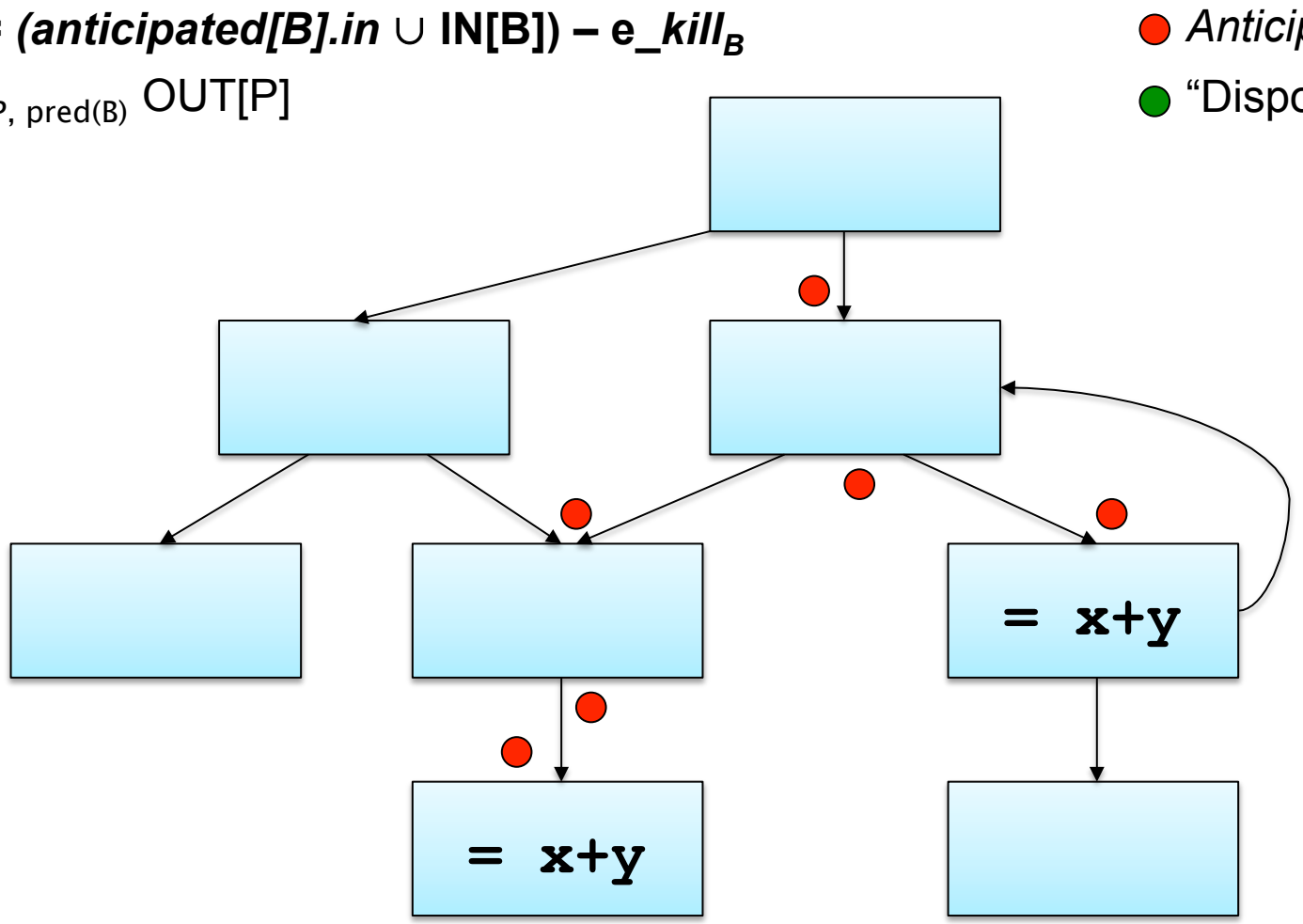
Posicionamento mais Cedo

- A expressão x+y está em Earliest[B] se ela é anticipável no início de B mas não está "disponível" lá.
  - Ou seja: quando x+y está em IN[B] na computação de *anticipated expressions* (anticipated[B].in), mas não está em IN[B] na computação de expressões disponíveis (available[B].in).
- Earliest[B] = anticipated[B].in – available[B].in

Exemplo: Posicionamento mais cedo

OUT[B] = *(anticipated[B].in* ∪ IN[B]) – e_$kill_B$

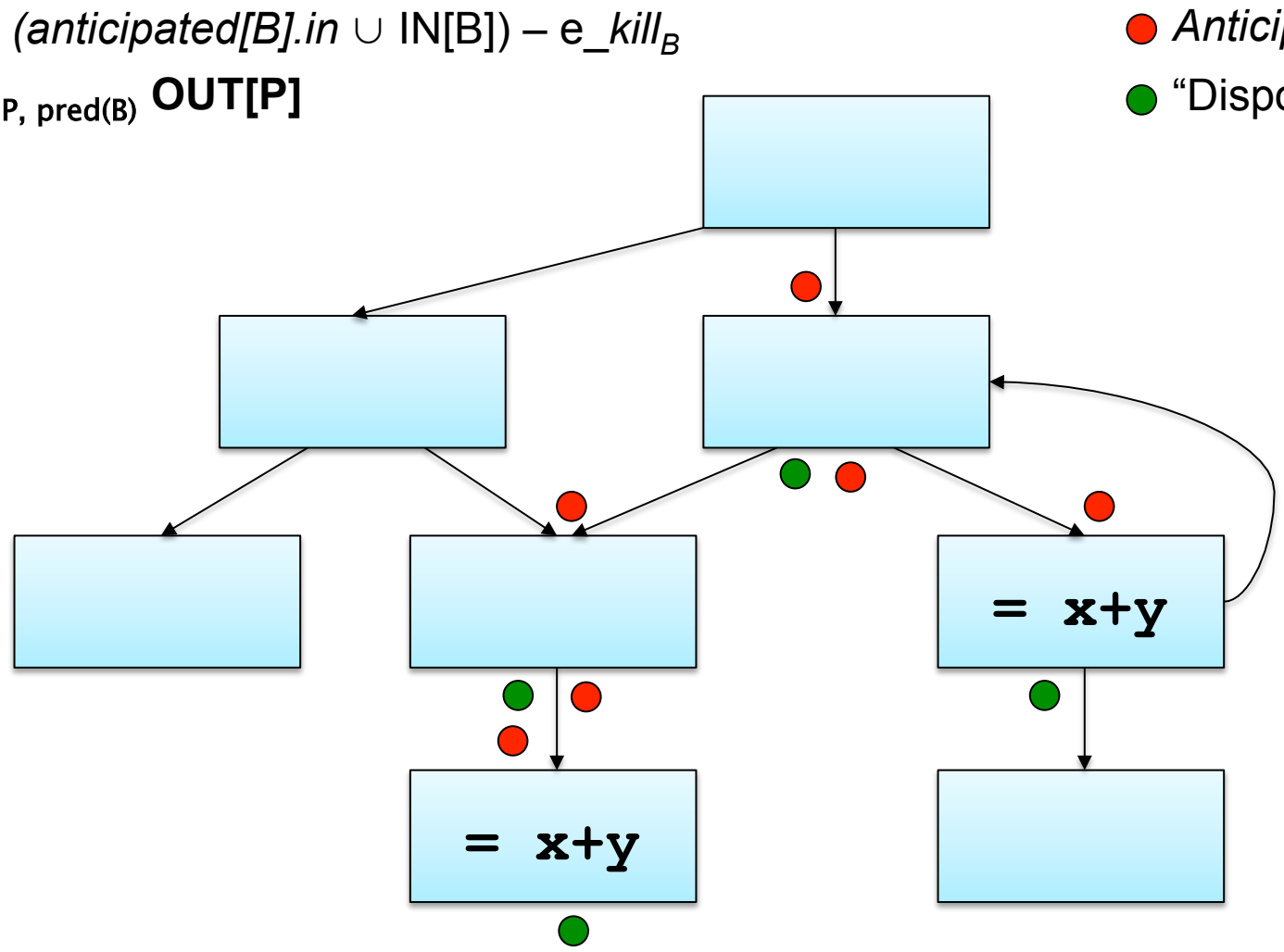
IN[B] = $\cap_{P,\ pred(B)}$ OUT[P]



Exemplo: Posicionamento mais dedo

OUT[B] = *(anticipated[B].in* ∪ IN[B]) – e_$kill_B$

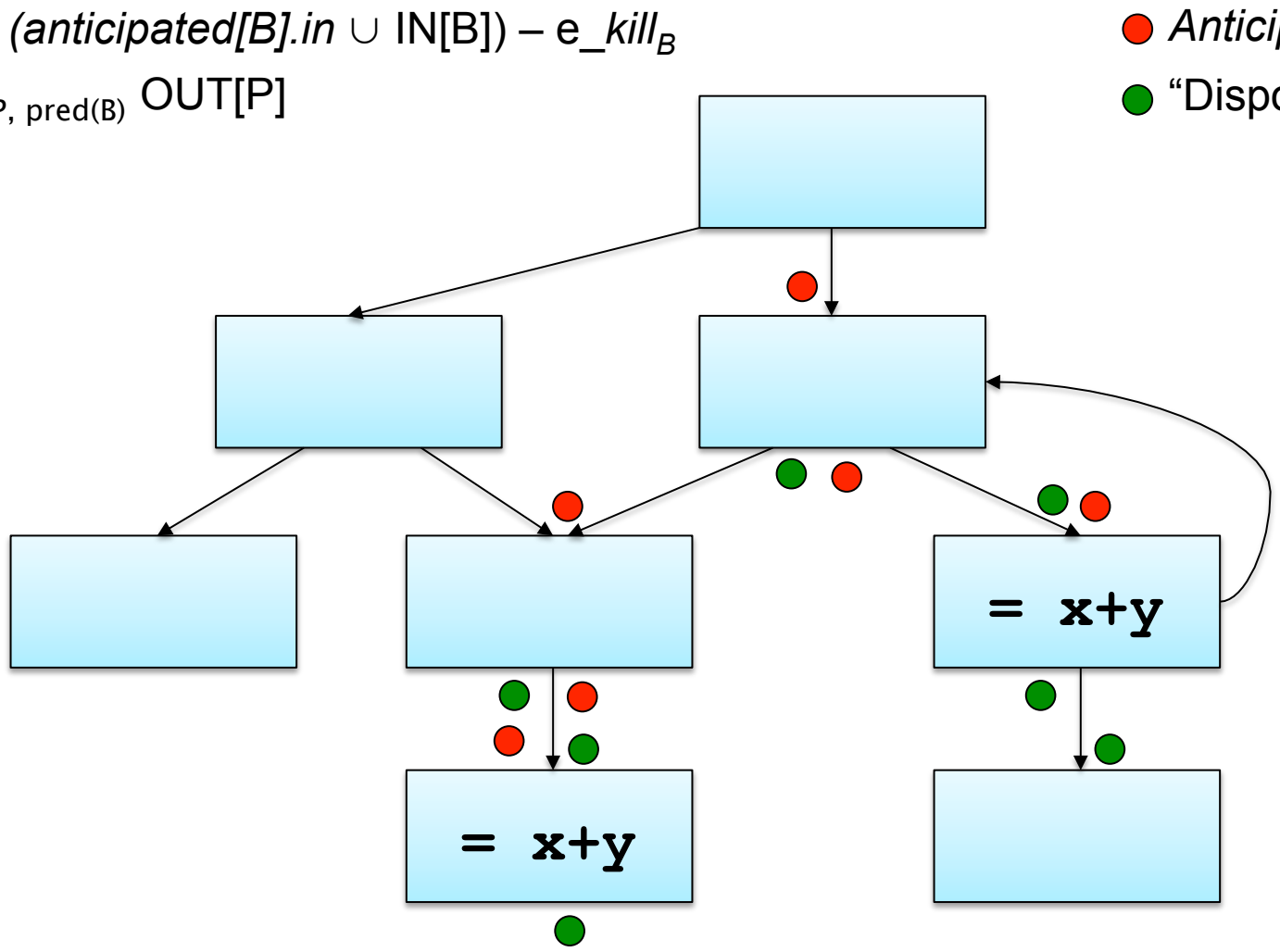
IN[B] = $\cap_{P,\ pred(B)}$ OUT[P]



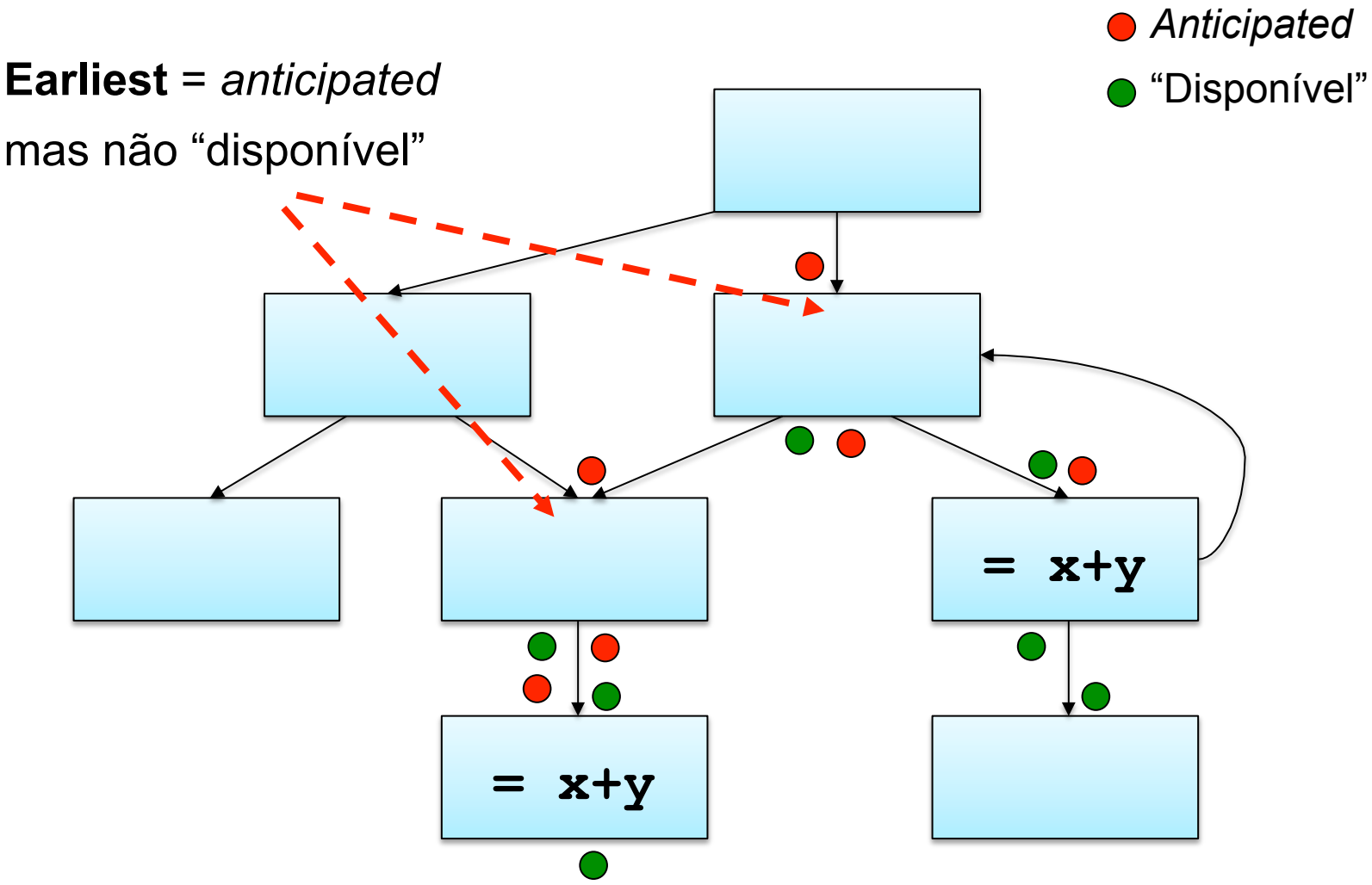
Exemplo: Posicionamento mais dedo

OUT[B] = *(anticipated[B].in* ∪ IN[B]) – e_$kill_B$

IN[B] = $\cap_{P,\ pred(B)}$ OUT[P]



Exemplo: Posicionamento mais dedo



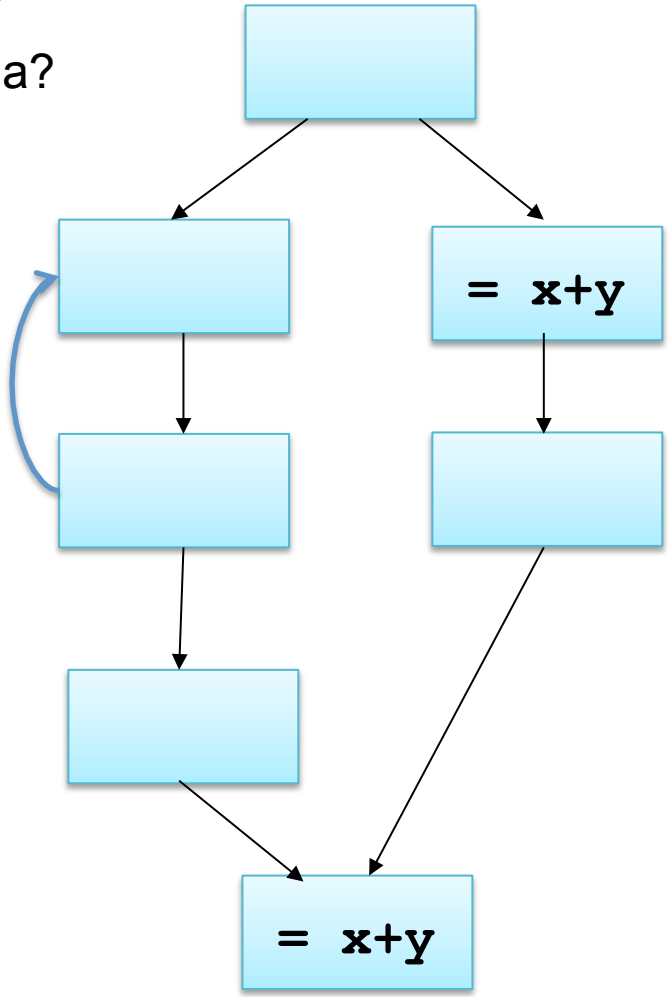
3 - Postponable Expressions

- Computar os pontos nos quais as expressões são “postergáveis” (*postponable*). E introduzir as cópias das expressões o mais tarde possível.
  1. Não pode ser após o uso da expressão.
  2. Não pode introduzir redundância. Ou seja, computar a expressão em pontos onde o valor já está disponível.
  3. É aceitável aumentar o tamanho do código (maior número de expressões) se nós reduzirmos a pressão no alocador de registradores, ou seja, se diminuirmos o tempo de vida dos temporários.

Exemplo

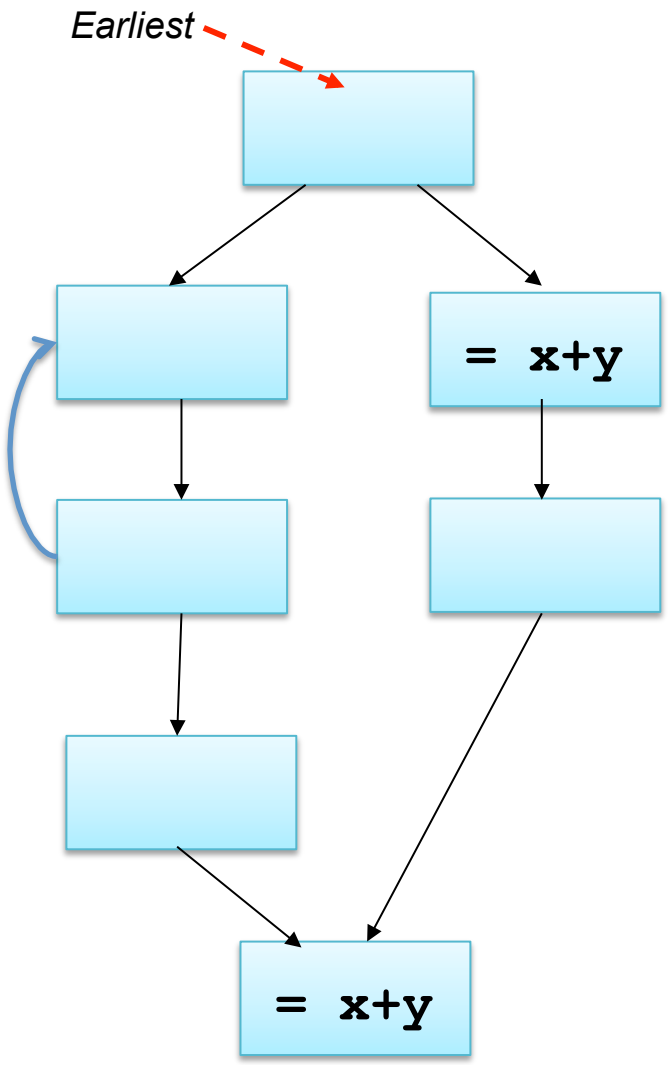
Qual o primeiro ponto

onde x+y é antecipada?



Exemplo

Por que você não quer colocar a expressão aqui?



3 - Postponable Expressions

- x+y é postergável para um ponto p se em todo caminho da entrada até p:
  1. Existe um bloco B tal que x+y está no cunjunto Earliest[B], e
  2. Após este bloco, não há mais usos de x+y.

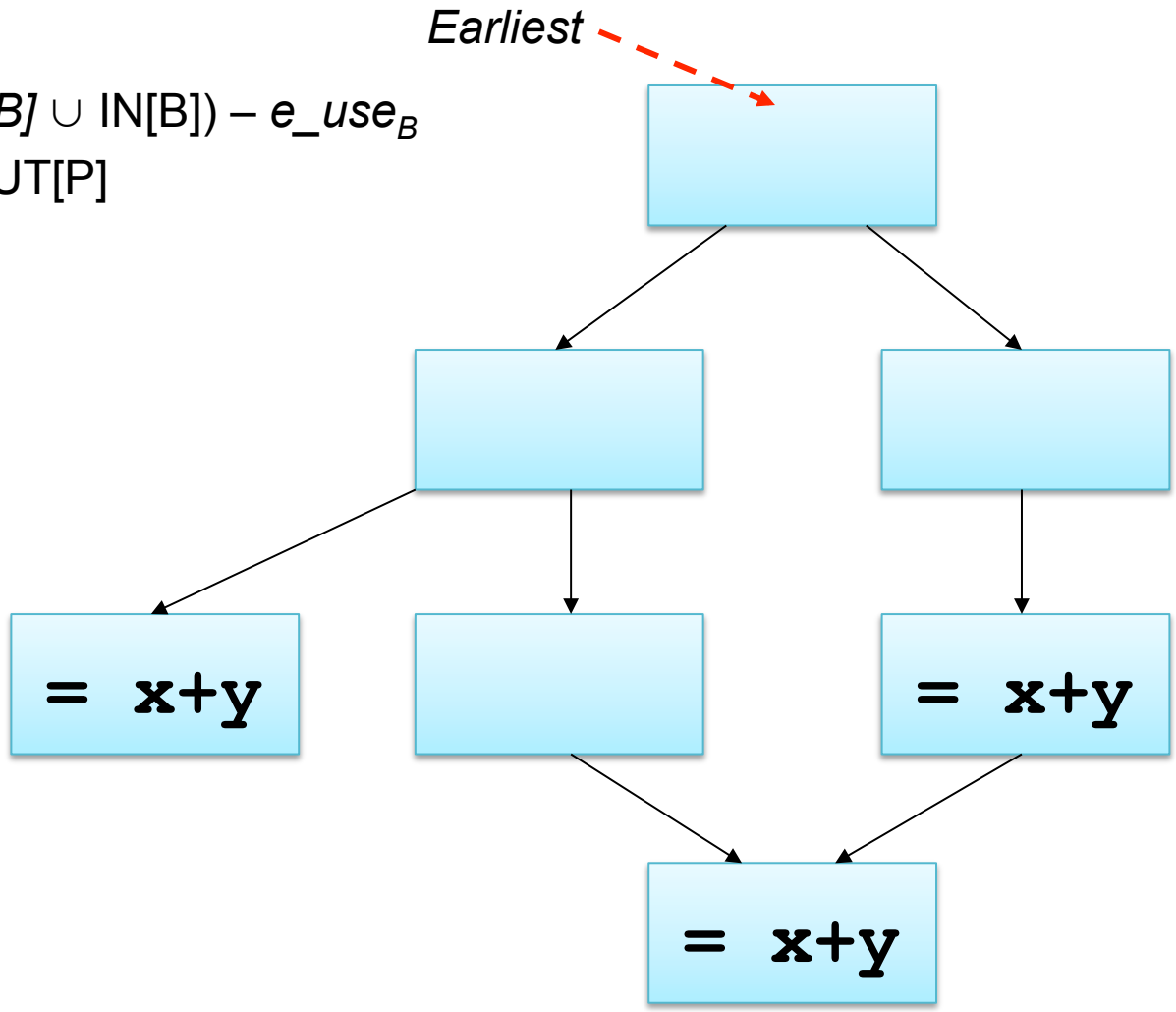
3 - Postponable Expressions

- DFA
  - Direção: *forwards*.
  - Inicialização:
    - OUT[ENTRY] = {}
    - OUT[B] = $\{e_1, e_2, ..., e_N\}$
  - Equações
    - OUT[B] = $(earliest[B] \cup IN[B]) - e\_use_B$
    - IN[B] = $\bigcap_{P,\ pred(B)} OUT[P]$

Exemplo: *Postponable Expressions*

OUT[B] = *(earliest[B]* ∪ IN[B]) – e_use_B

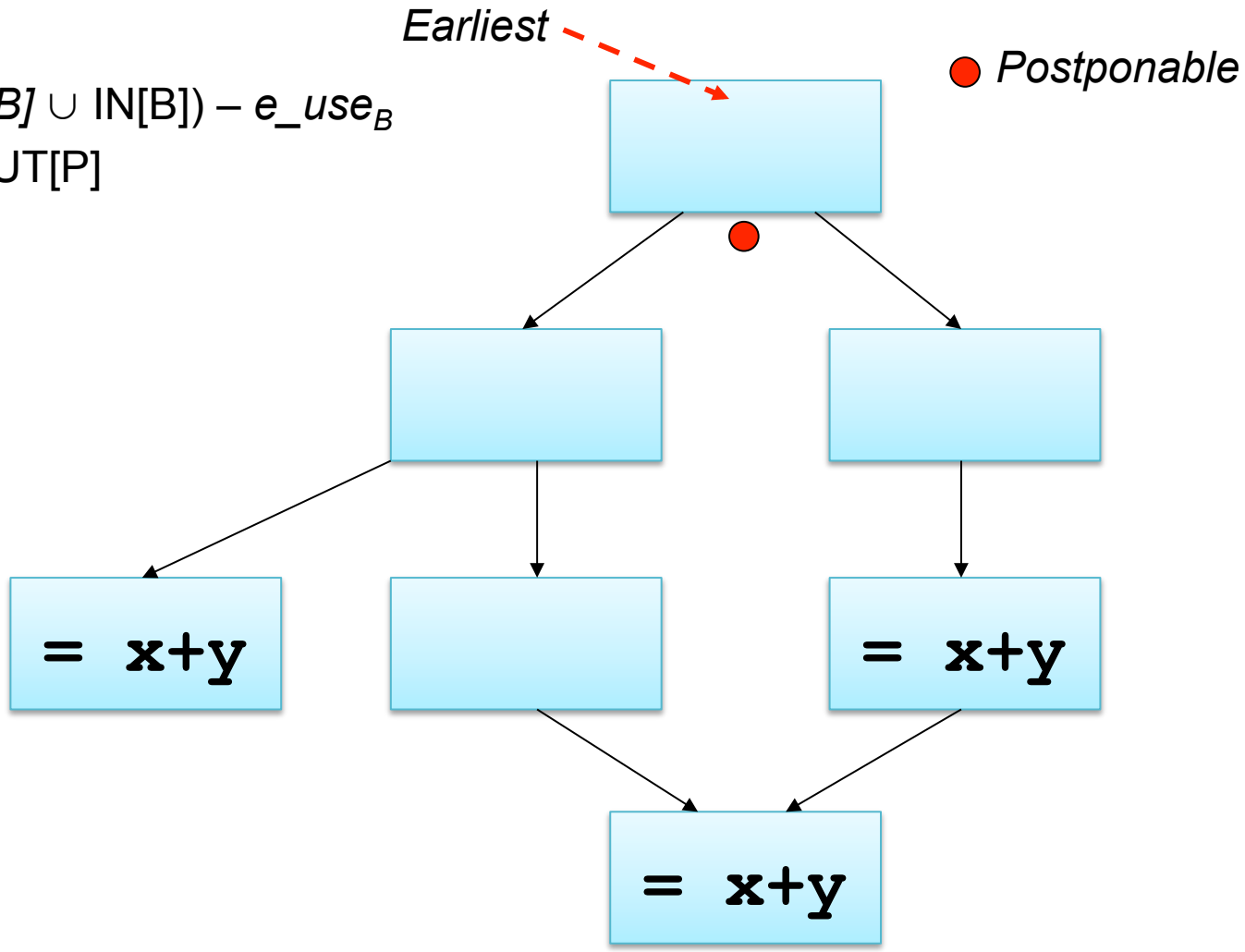
IN[B] = $\cap_{P,\ pred(B)}$ OUT[P]



Exemplo: *Postponable Expressions*

OUT[B] = *(earliest[B]* ∪ IN[B]) – e_use_B

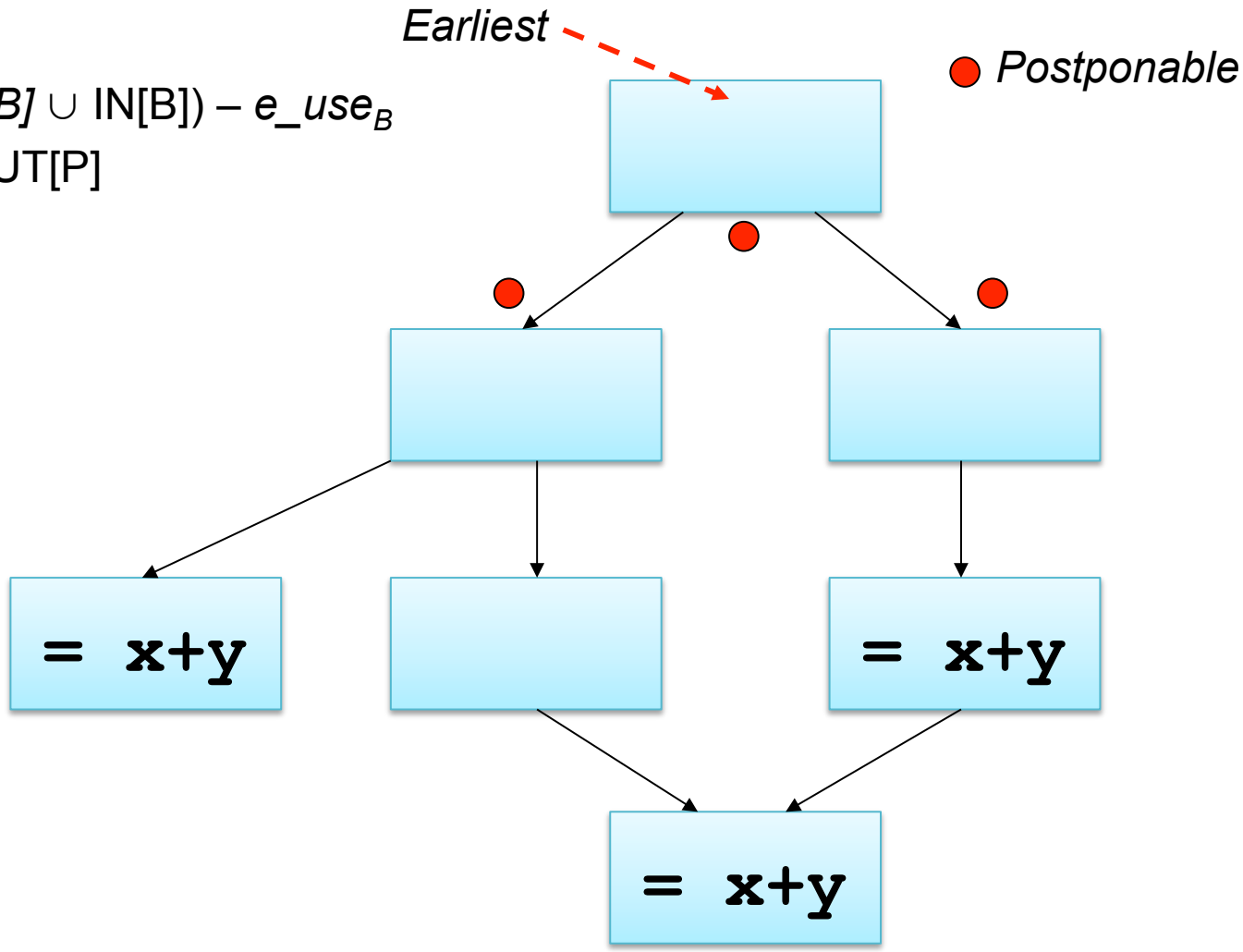
IN[B] = $\cap_{P,\ pred(B)}$ OUT[P]



Exemplo: *Postponable Expressions*

OUT[B] = *(earliest[B]* ∪ IN[B]) – e_use_B

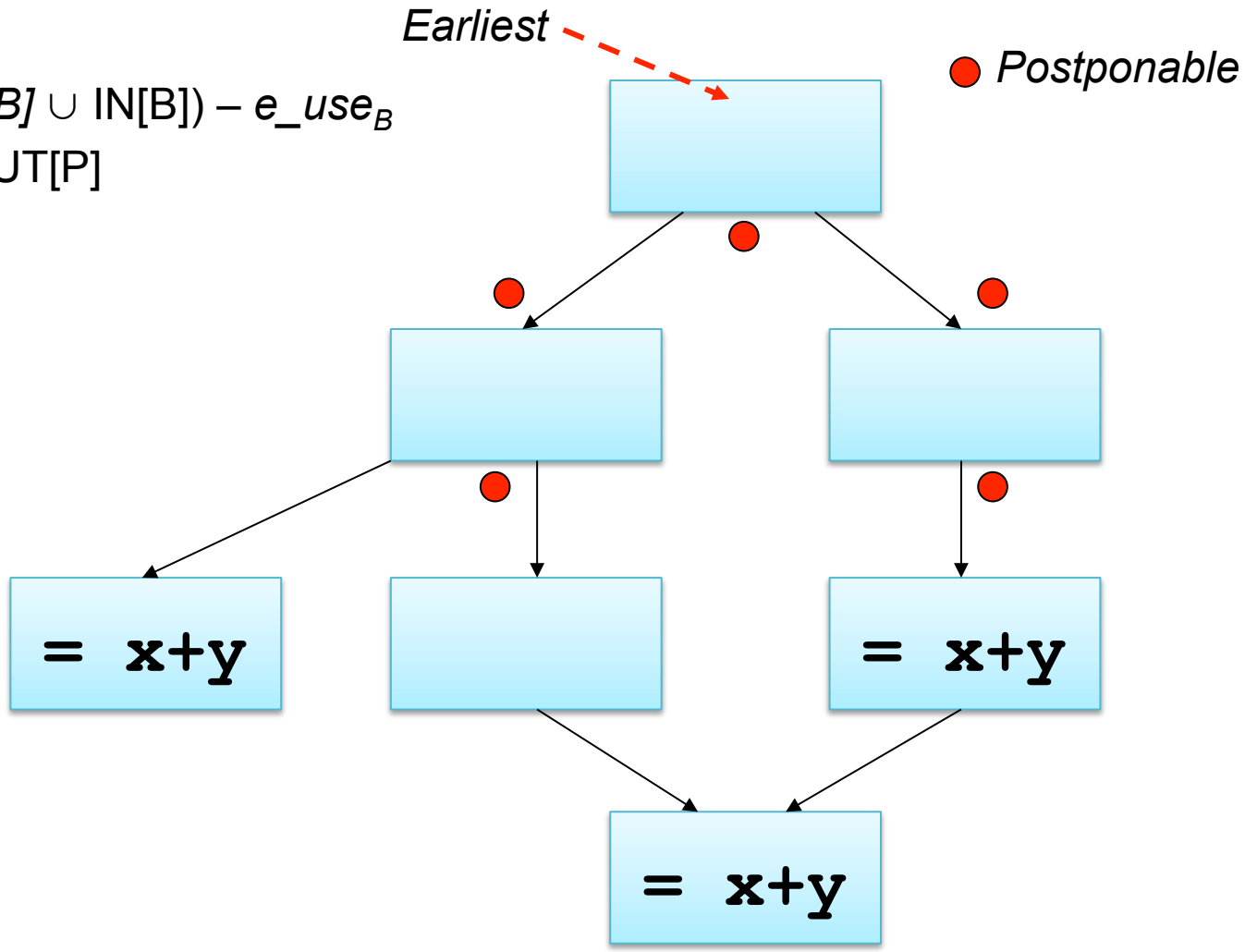
IN[B] = $\cap_{P,\ pred(B)}$ OUT[P]



Exemplo: *Postponable Expressions*

OUT[B] = *(earliest[B]* ∪ IN[B]) – e_use_B

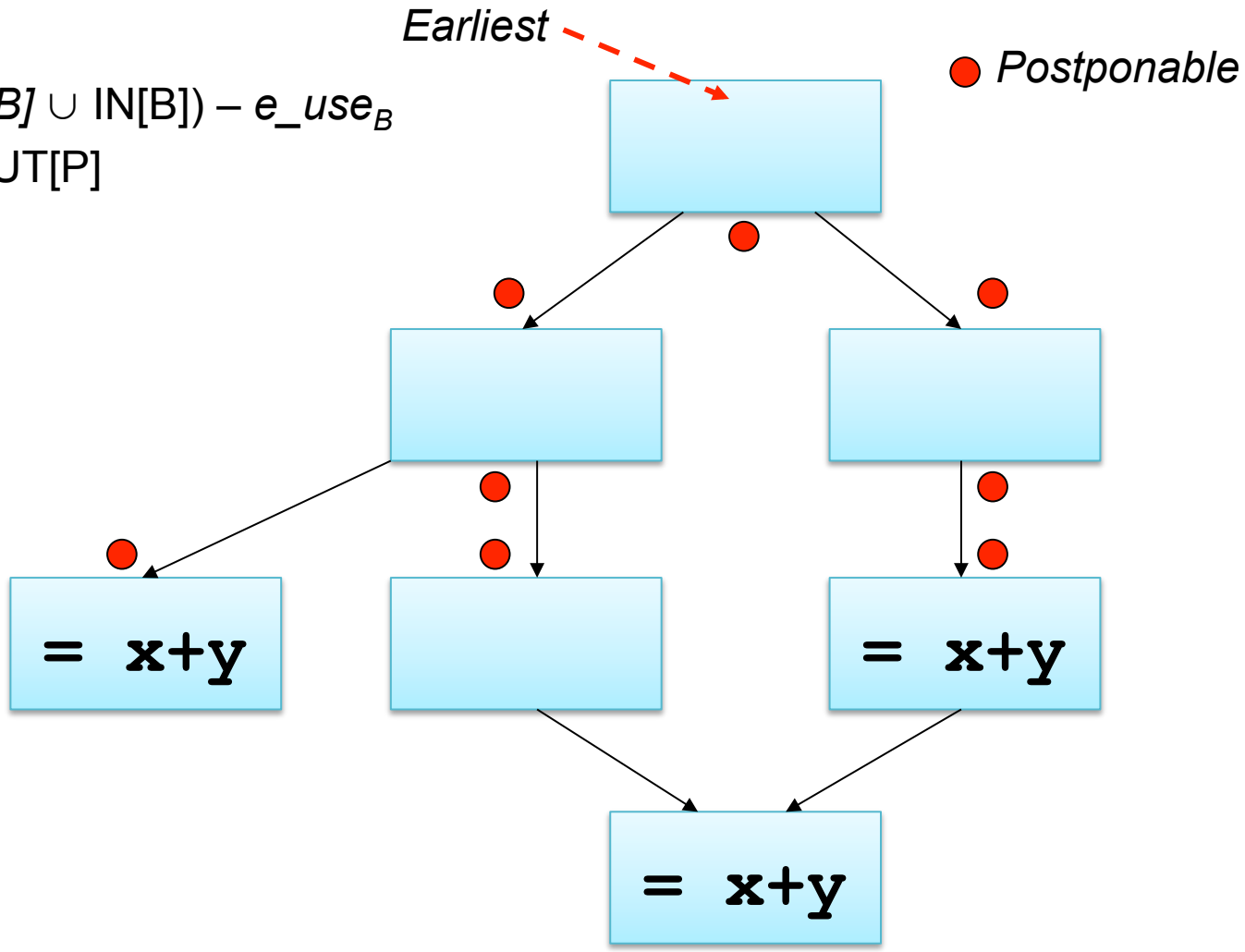
IN[B] = $\cap_{P,\ pred(B)}$ OUT[P]



Exemplo: *Postponable Expressions*

OUT[B] = *(earliest[B]* ∪ IN[B]) – e_use_B

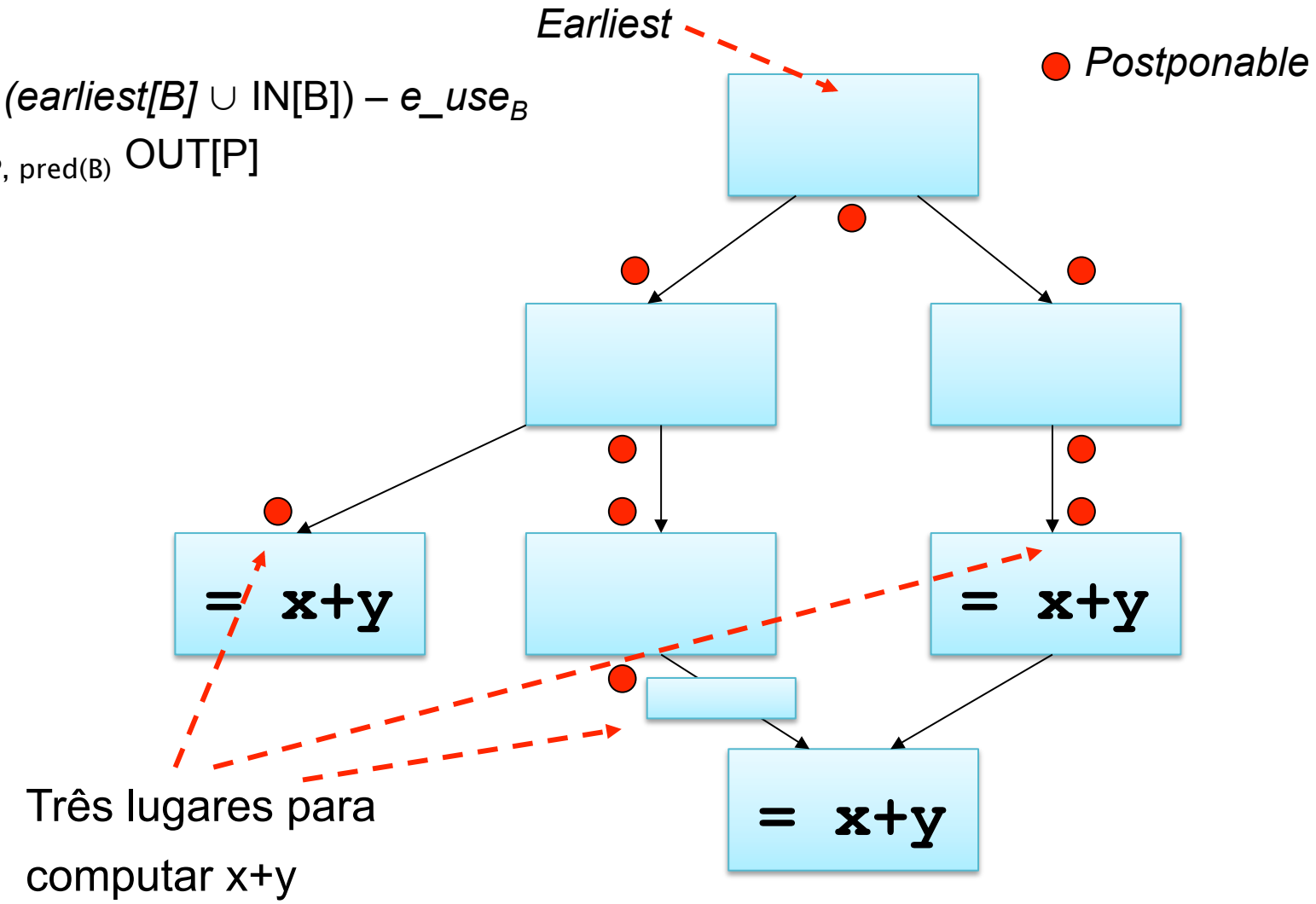
IN[B] = $\cap_{P,\ pred(B)}$ OUT[P]



Exemplo: *Postponable Expressions*

OUT[B] = *(earliest[B]* ∪ IN[B]) – e_use_B

IN[B] = $\cap_{P,\ pred(B)}$ OUT[P]



Latest[B]

- Queremos postergar o máximo possível.
- Quais são os pontos do programa onde não podemos postergar mais?
- Não podemos postergar a computação de uma expressão após o uso da mesma ou para um bloco básico com outro predecessor tal que x+y não é postergável.

Latest[B]

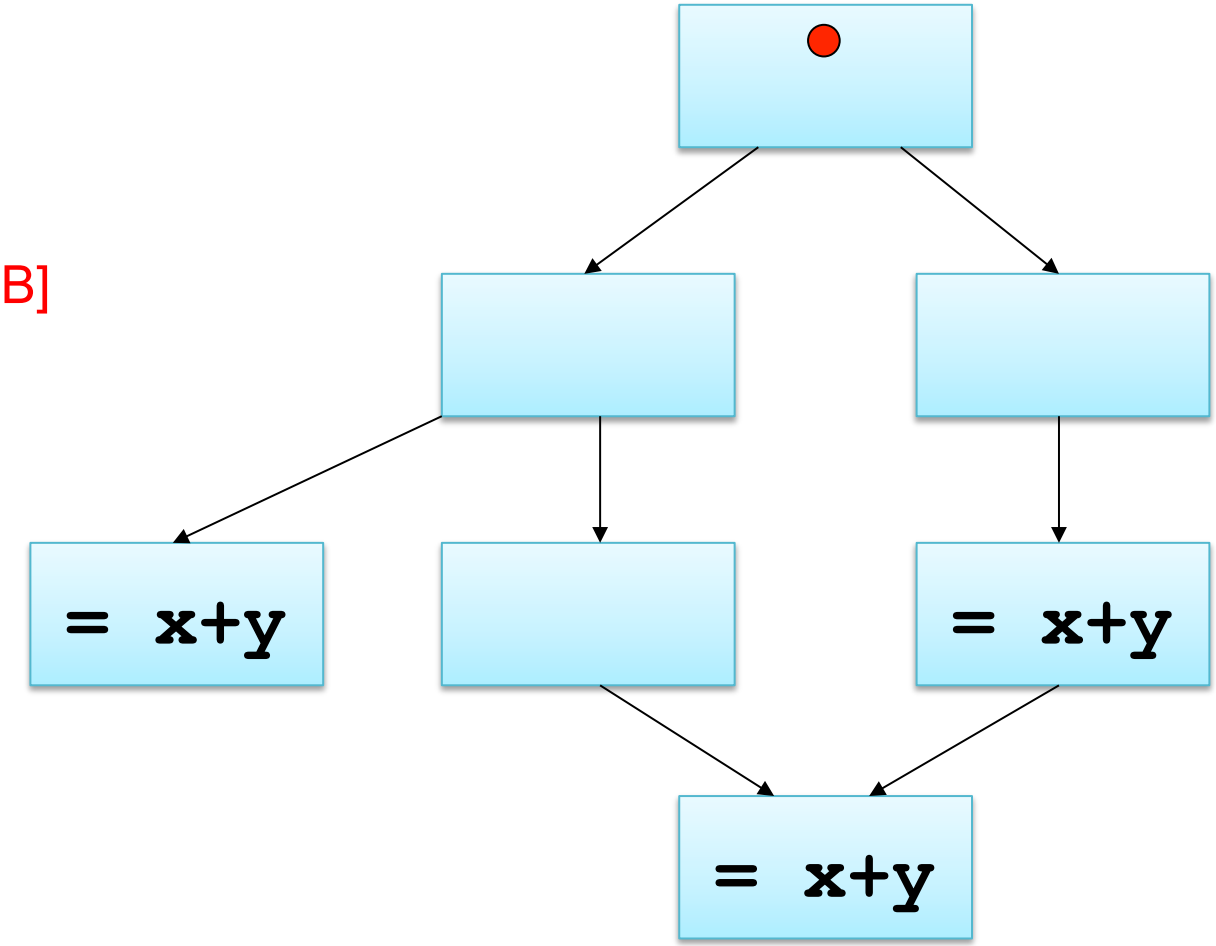
- Para x+y estar em Latest[B]:
  1. x+y tem que estar em Earliest[B] ou em postponable[B].in; e
  2. x+y é usada em B ou existe um sucessor de B tal que (1) não é válido.
     - Ou seja, não podemos postergar mais.

$$latest[B] = (earliest[B] \cup postponable[B].in) \cap (e\_useB \cup \neg(\bigcap_{S \in succ(B)} (earliest[S] \cup postponable[S].in)))$$

Exemplo: *Latest*

$$latest[B] = (earliest[B] \cup postponable[B].in) \cap (e\_useB \cup \neg(\bigcap_{S \in succ(B)} (earliest[S] \cup postponable[S].in)))$$

Earliest[B]

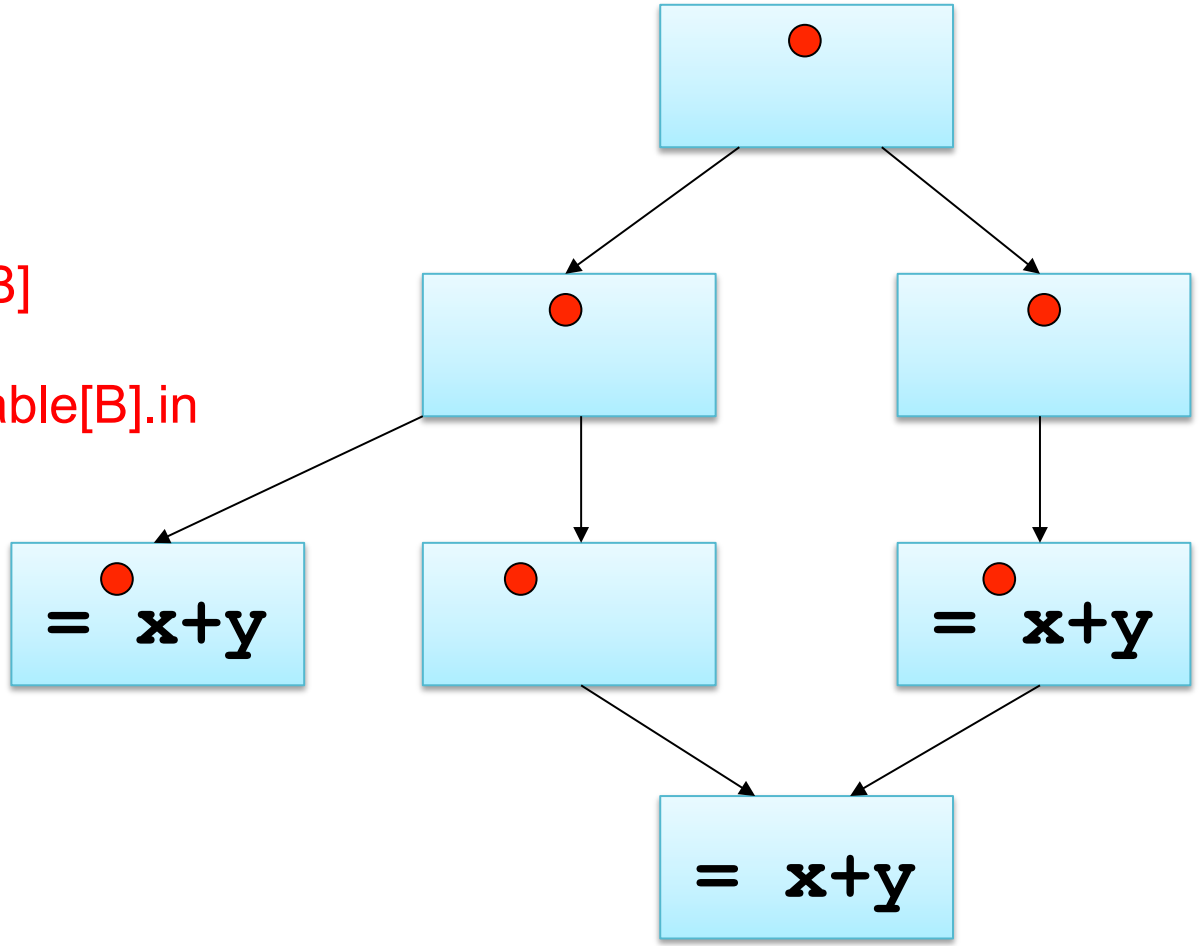


Exemplo: *Latest*

$$latest[B] = \begin{array}{l}(earliest[B] \cup postponable[B].in) \quad \cap \\ (e\_useB \cup \neg(\bigcap_{S \in succ(B)} (earliest[S] \cup postponable[S].in)))\end{array}$$

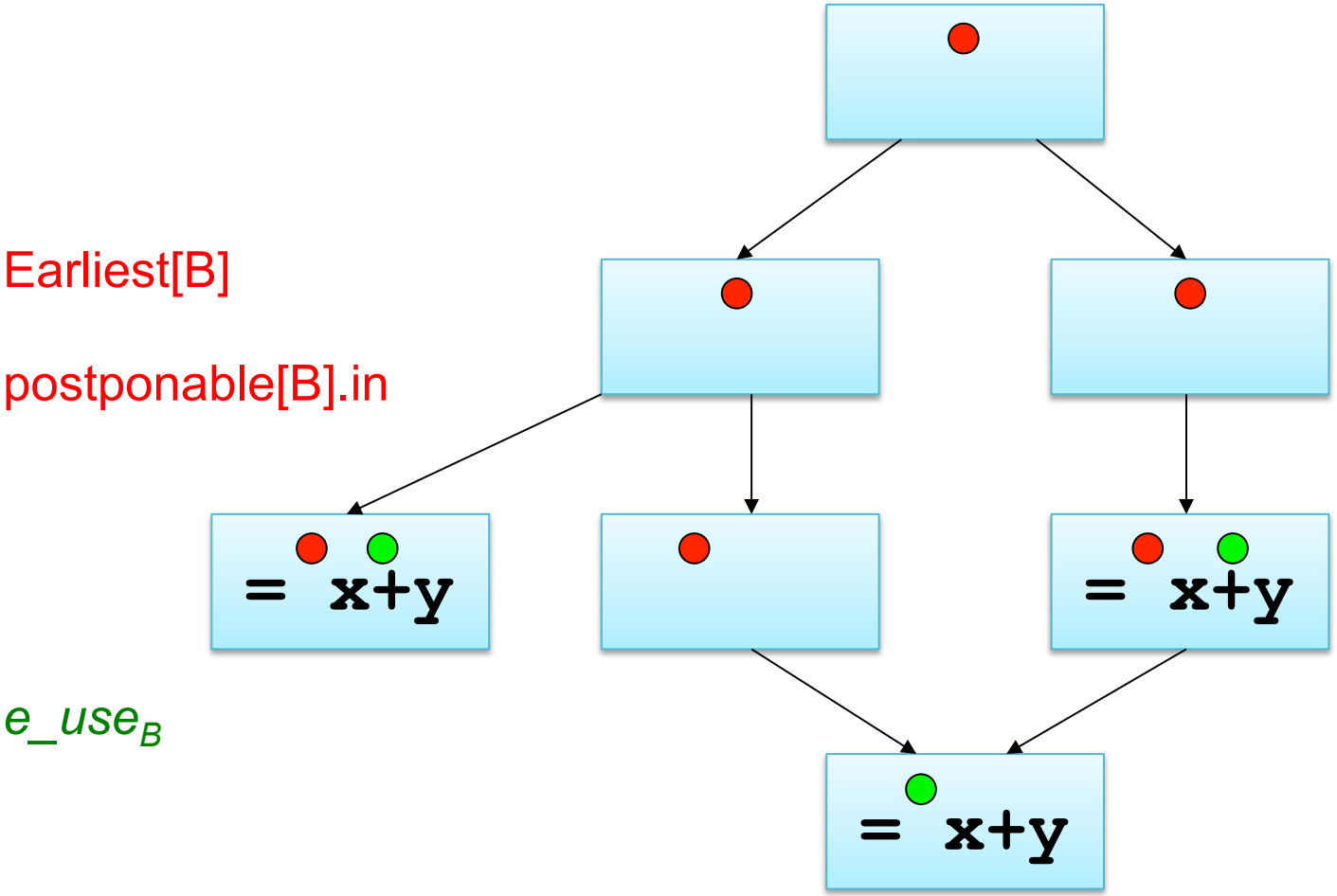
Earliest[B]

postponable[B].in



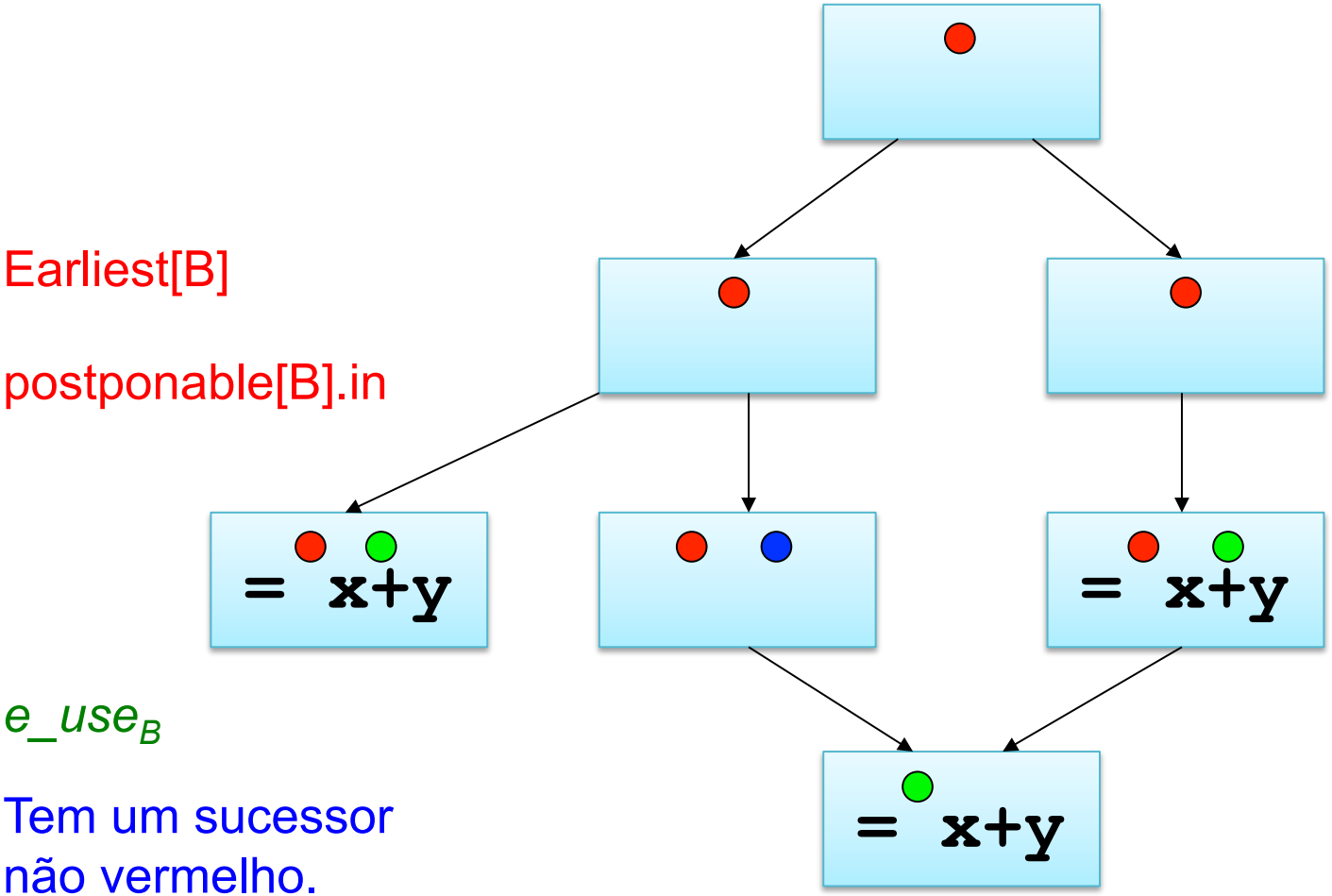
Exemplo: *Latest*

$$latest[B] = \begin{array}{l}(earliest[B] \cup postponable[B].in) \quad \cap \\ (e\_useB \cup \neg(\bigcap_{S \in succ(B)} (earliest[S] \cup postponable[S].in)))\end{array}$$



Exemplo: *Latest*

$$latest[B] = (earliest[B] \cup postponable[B].in) \cap (e\_useB \cup \neg(\bigcap_{S \in succ(B)} (earliest[S] \cup postponable[S].in)))$$



Exemplo: *Latest*

$$latest[B] = \begin{array}{l}(earliest[B] \cup postponable[B].in) \quad \cap \\ (e\_useB \cup \neg(\bigcap_{S \in succ(B)} (earliest[S] \cup postponable[S].in)))\end{array}$$

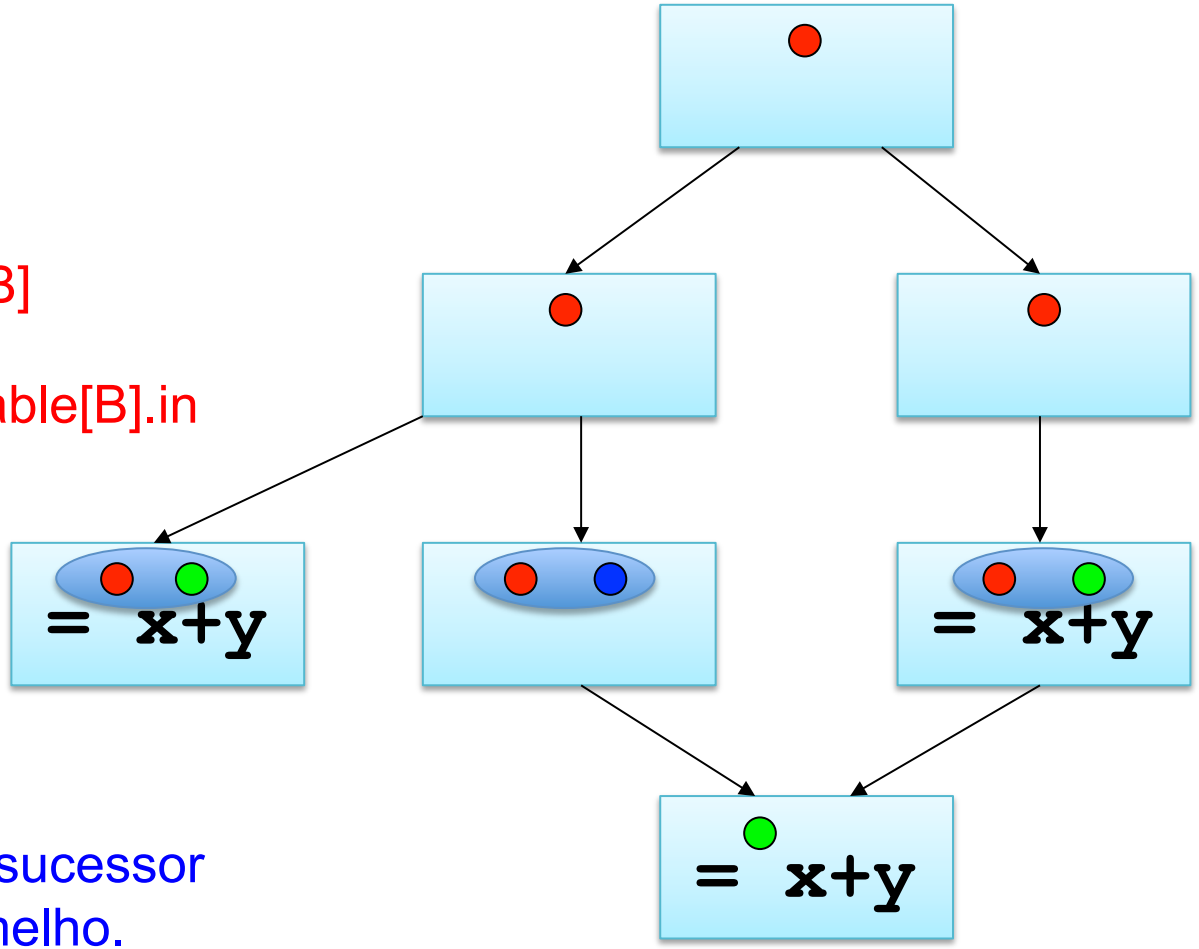
Earliest[B]

postponable[B].in

$\cap$

e_use_B

Tem um sucessor não vermelho.



4 - Expressões Usadas

- Podemos introduzir o temporário t para armazenar o valor computado pela expressão x+y.
- Mas nem sempre é necessário.
  - P.e, x+y é computado em exatamente um lugar.

4 - Expressões Usadas

- Dizemos que uma expressão e está em uso em um ponto p se existe um caminho saindo de p que usa a expressão antes da mesma ser reavaliada.
- *Análise de longevidade para expressões!!*

4 - Expressões Usadas

- DFA
  - Direção: *backwards*.
  - Inicialização:
    - IN[EXIT] = {}
    - IN[B] = {}
  - Equações
    - IN[B] = $(e\_use_B \cup \text{IN[B]}) - \text{Latest[B]}$
    - OUT[B] = $\bigcup_{S,\ succ\ (B)} \text{IN[S]}$

Lazy-Code-Motion

- O algoritmo *Lazy-Code-Motion* tenta introduzir cópias de *e* no CFG para tornar a expressão *e* totalmente redundante.

1. Descobrir os pontos próximo ao início do CFG para onde podemos mover as expressões (*Anticipated Expressions*)
2. Verificar se a expressão já está disponível nos pontos onde ela é "anticipável". Se já estiver disponível não há necessidade de introduzir uma cópia.
3. Computar os pontos nos quais as expressões são "postergáveis" (*postponable*). E introduzir as cópias das expressões o mais tarde possível.
4. Eliminar atribuições desnecessárias à variáveis temporárias.

Algoritmo Lazy Code Motion

- Pré-processamento
  - Assume blocos de uma instrução
  - Só inserimos novas sentenças no início de blocos
  - Insere um bloco vazio em todas as arestas que chegam a um bloco com mais de um predecessor
    - Resolve arestas críticas

Algoritmo Lazy Code Motion

1. Compute antecipated[B].in para todo B
2. Compute available[B].in para todo B
3. Compute earliest[B] para todo B
4. Compute postponable[B].in para todo B
5. Compute latest[B] para todo B
6. Copmute used[B].out para todo B

Algoritmo Lazy Code Motion

1. Para cada expressão x+y do programa:
   1. Crie um novo temporário t
   2. Para todo bloco B tal que x+y pertence a latest[B] ∩ used[B].out
      1. Adicione t=x+y ao inicio de B
   3. Para todo B tal que x+y pertence a:

$$e\_use_B \cup (\neg latest[B] \bigcup used.out[B])$$

      1. Substitua cada x+y original por t