Blocos Básicos e Grafos de Fluxo de Controle

Introdução

- Representação gráfica do código de 3 endereços é útil para entender os algoritmos de otimização
- **Nós**: computação
- **Arestas**: fluxo de controle
- Muito usado em coletas de informações sobre o programa

Blocos Básicos

- Seqüência de instruções consecutivas
- Fluxo de Controle:
  - Entra no início
  - Sai pelo final
  - Não existem saltos para dentro ou do meio para fora da seqüência

```
t1 = a * a
t2 = a * b
t3 = b * 3
t4 = t2 - t3
```

Algoritmo para Quebra em BBs

- **Entrada:** seqüência de código de 3 endereços

- **Defina os líderes** (iniciam os BBs):
  - Primeira Sentença é um líder
  - Todo alvo de um **goto**, **condicional** ou **incondicional**, é um líder
  - Toda sentença que sucede imediatamente um **goto**, **condicional** ou **incondicional**, é um líder

- Os BBs são compostos pelos líderes e todas as instruções subsequentes até o próximo líder (exclusive)

Quick Sort

```
(1)   i := m-1
(2)   j := n
(3)   t1 := 4*n
(4)   v := a[t1]
(5)   i := i+1
(6)   t2 := 4*i
(7)   t3 := a[t2]
(8)   if t3 < v goto (5)
(9)   j := j-1
(10)  t4 := 4*j
(11)  t5 := a[t4]
(12)  if t5 > v goto (9)
(13)  if i >= j goto (23)
(14)  t6 := 4*i
(15)  x := a[t6]
(16)  t7 := 4*i
(17)  t8 := 4*j
(18)  t9 := a[t8]
(19)  a[t7] := t9
(20)  t10 := 4*j
(21)  a[t10] := x
(22)  goto (5)
(23)  t11 := 4*i
(24)  x := a[t11]
(25)  t12 := 4*i
(26)  t13 := 4*n
(27)  t14 := a[t13]
(28)  a[t12] := t14
(29)  t15 := 4*n
(30)  a[t15] := x
```

Fig. 10.4. Three-address code for fragment in Fig. 10.2.

Quick Sort

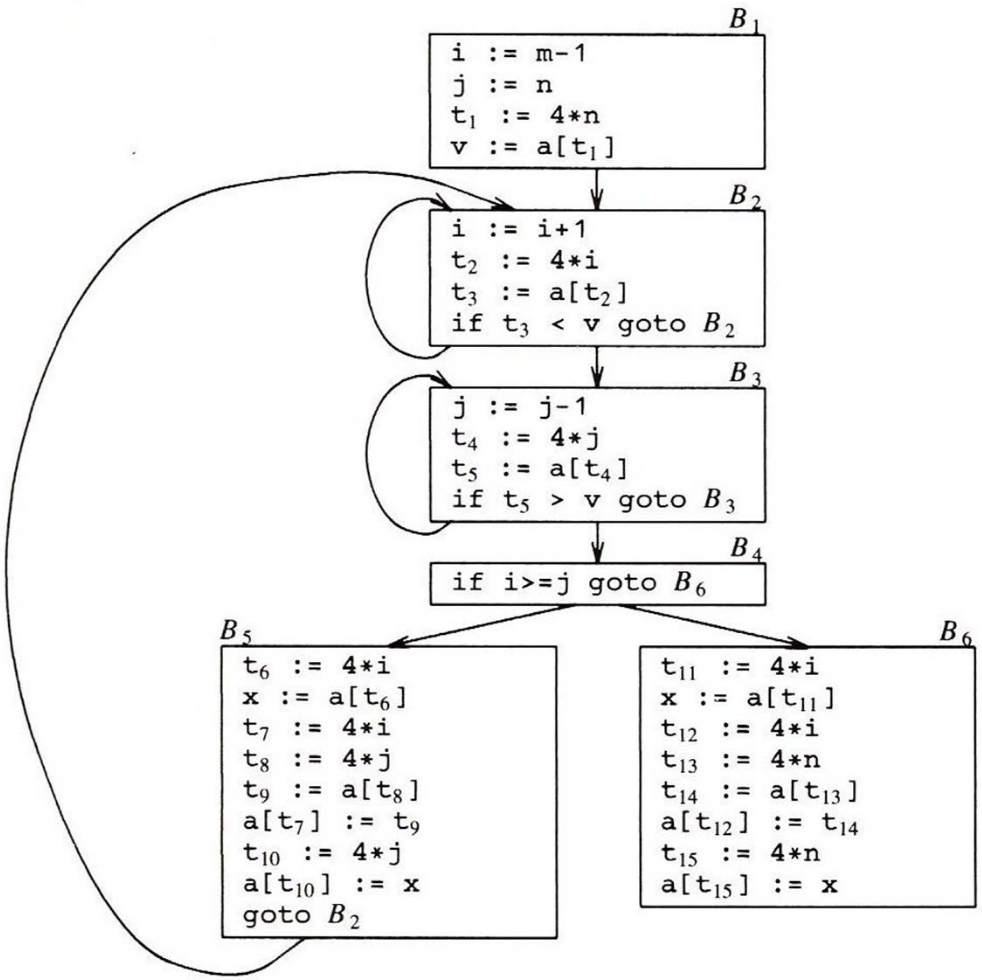


Fig. 10.5. Flow graph.

Análise de Fluxo de Dados

Introdução

- **Otimização**
  - Transformações para ganho de eficiência
  - Não podem alterar a saída do programa

- **Exemplos:**
  - ***Dead Code Elimination***: Apaga uma computação cujo resultado nunca será usado
  - ***Register Allocation***: Reaproveitamento de registradores
  - ***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

Introdução

- Otimizações são transformações feitas com base em informações coletadas do programa
- Coletar informações é trabalho da análise de fluxo de dados.
- Intraprocedural global optimization
  - Interna a um procedimento ou função
  - Engloba todos os blocos básicos

Introdução

- **Idéia básica**
  - Atravessar o grafo de fluxo de controle do programa coletando informações sobre a execução
  - Conservativamente!
  - Modificar o programa para torná-lo mais eficiente em algum aspecto:
    - Desempenho
    - Tamanho

- **Análises são descritas através de equações de fluxo de dados:**

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

Introdução

As equações podem mudar de acordo com a análise:

- As noções de *gen* e *kill* dependem da informação desejada
- Podem seguir o fluxo de controle ou não
  - Forward
  - Backward

- Chamadas de procedimentos, atribuição a ponteiros
  e a arrays não serão consideradas em um primeiro momento.

Pontos e Caminhos

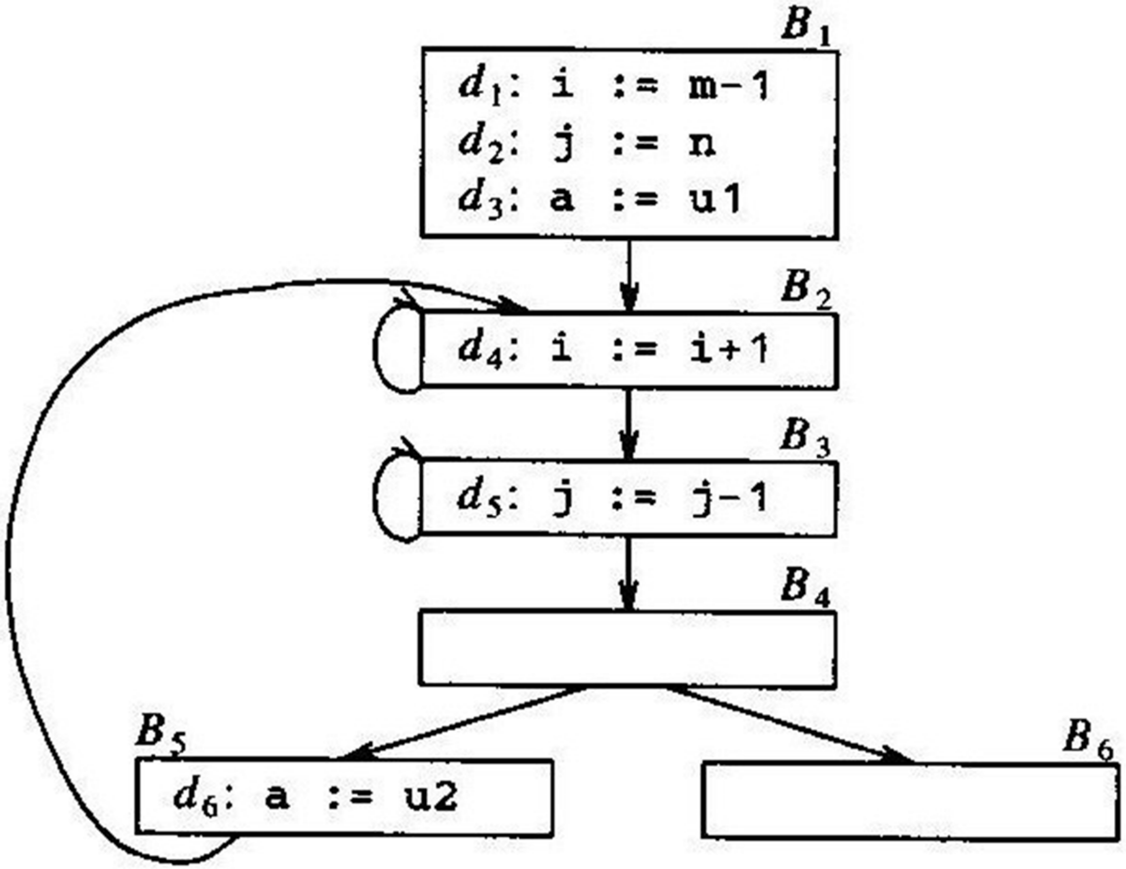


Fig. 10.19. A flow graph.

Análise de Fluxo de Dados: Reaching Definitions

Reaching Definitions

- **Definição não ambígua de uma variável t:**

  d: `t := a op b`

  d: `t := M[a]`

- *d* alcança um uso na sentença *u* se:
  - Se existe um caminho no CFG de *d* para *u*
  - Esse caminho não contém outra definição não ambígua de t

- **Definição ambígua**
  - Uma sentença que pode ou não atribuir um valor a t
    - CALL
    - Atribuição a ponteiros

Reaching Definitions

- Criam-se IDs para as definições:

  `d1: t ← x op y`

  - Gera a definição `d1`
  - Mata todas as outras definições da variável `t`, pois não alcançam o final dessa instrução.

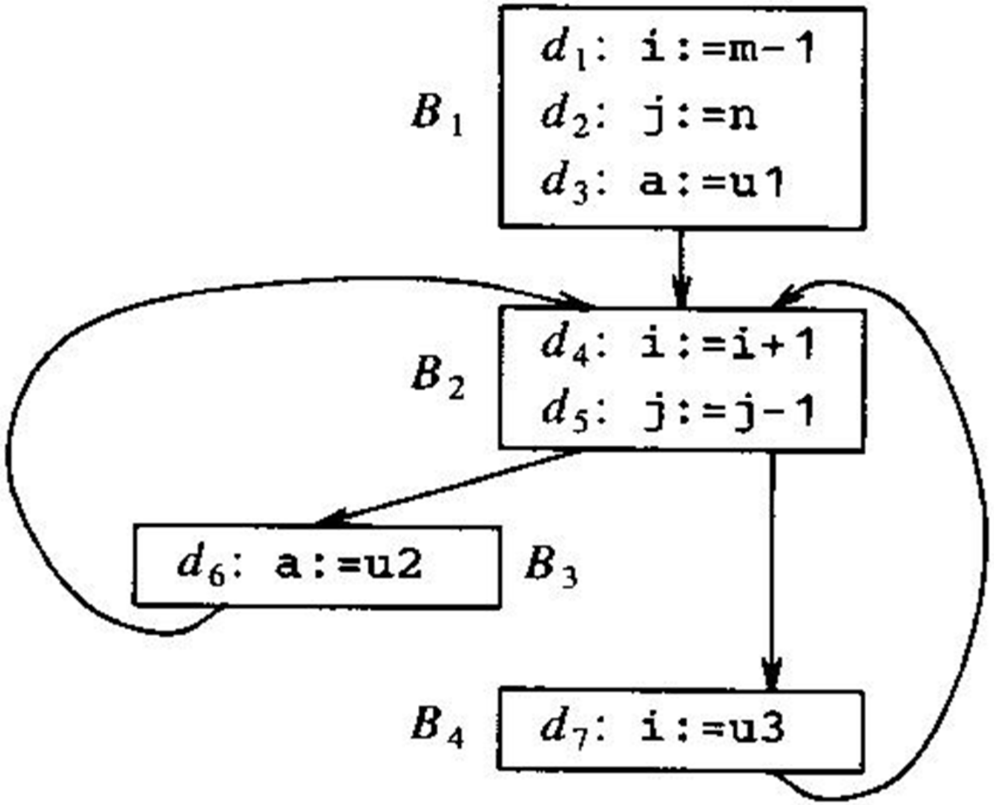
- **defs(t)** ou **D_t**: conjunto de todas as definições de `t`

Reaching Definitions

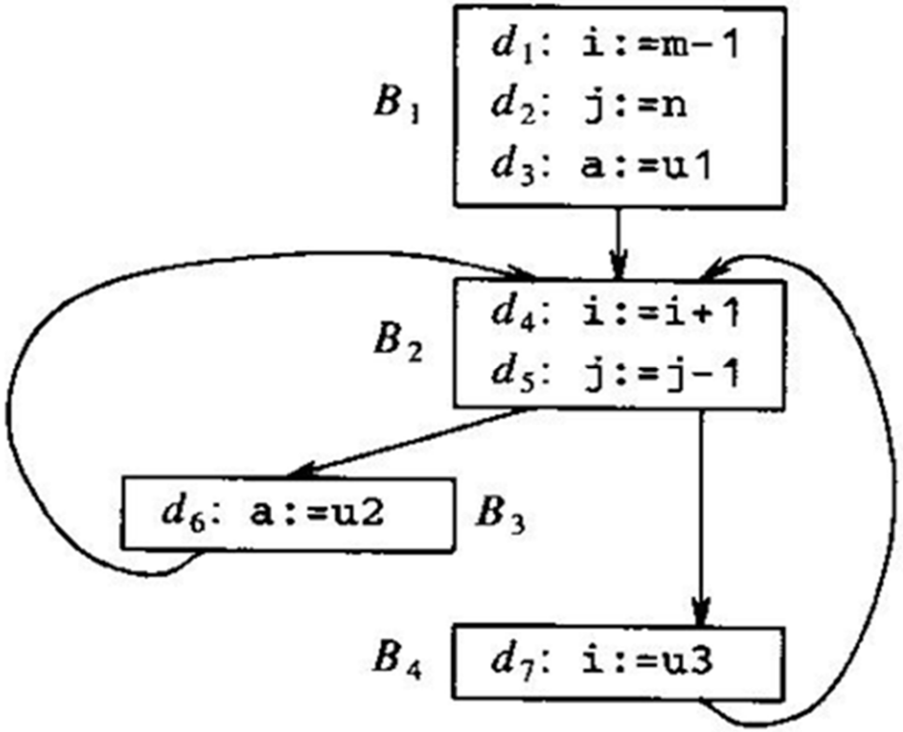
Principal uso:

- Dada uma variável **x** em um certo ponto ***p*** do programa, pode-se inferir que o valor de **x** é limitado a um determinado grupo de possibilidades.

Reaching Definitions: gen e kill



Reaching Definitions: gen e kill



$$gen[B_1] = \{d_1, d_2, d_3\}$$
$$kill[B_1] = \{d_4, d_5, d_6, d_7\}$$

$$gen[B_2] = \{d_4, d_5\}$$
$$kill[B_2] = \{d_1, d_2, d_7\}$$

$$gen[B_3] = \{d_6\}$$
$$kill[B_3] = \{d_3\}$$

$$gen[B_4] = \{d_7\}$$
$$kill[B_4] = \{d_1, d_4\}$$

Reaching Definitions: Equações de DFA

- Vendo B como uma sequência de uma ou mais sentenças
  - Pode-se definir
    - in[B], out[B], gen[B], kill[B]
  - Computando gen e kill para cada B como visto anteriormente
- Tem-se:

$$in[B] = \bigcup_{P \in Pred(B)} out[P]$$

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

Reaching Definitions: Solução Iterativa

```
      /* initialize out on the assumption in[B] = ∅ for all B */
(1)   for each block B do out[B] := gen[B];
(2)   change := true;        /* to get the while-loop going */
(3)   while change do begin
(4)       change := false;
(5)       for each block B do begin
(6)           in[B] :=        ∪          out[P];
                        P a prede-
                        cessor of B
(7)           oldout := out[B];
(8)           out[B] := gen[B] ∪ (in[B] - kill[B]);
(9)           if out[B] ≠ oldout then change := true
          end
      end
```

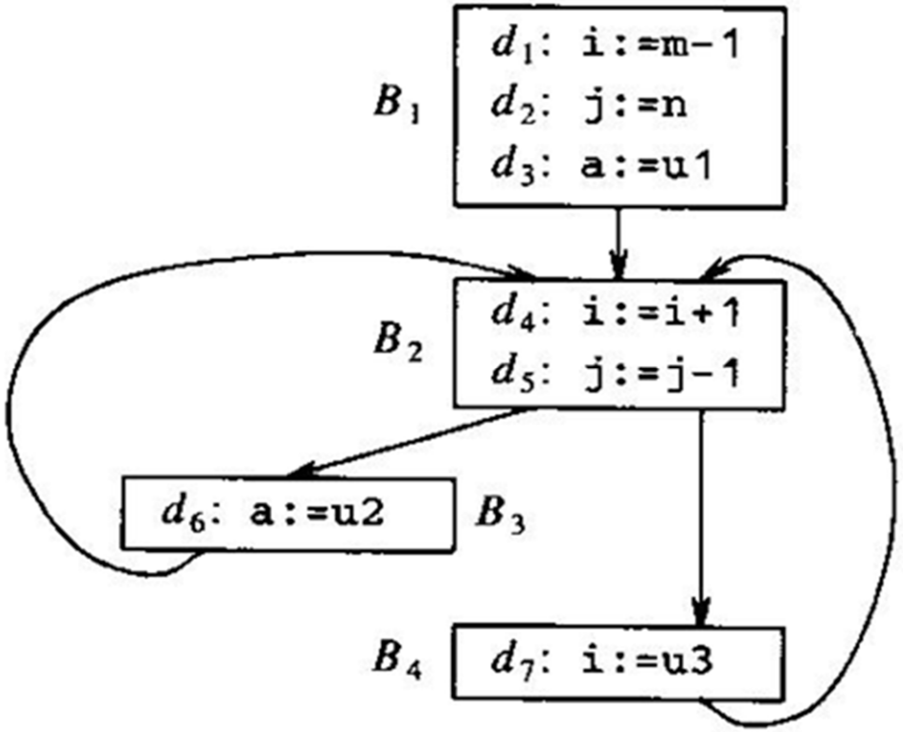
Fig. 10.26. Algorithm to compute *in* and *out*.

Reaching Definitions: Observações

- O algoritmo propaga as definições
  - Até onde elas podem chegar sem serem mortas
  - “Simula” todos os caminhos de execução

- O algoritmo sempre termina:
  - out[B] nunca diminui de tamanho
  - o número de definições é finito
  - se out[B] não muda, in[B] 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

Reaching Definitions: Computar in/out



$$gen[B_1] = \{d_1, d_2, d_3\}$$
$$kill[B_1] = \{d_4, d_5, d_6, d_7\}$$

$$gen[B_2] = \{d_4, d_5\}$$
$$kill[B_2] = \{d_1, d_2, d_7\}$$

$$gen[B_3] = \{d_6\}$$
$$kill[B_3] = \{d_3\}$$

$$gen[B_4] = \{d_7\}$$
$$kill[B_4] = \{d_1, d_4\}$$

Transformações para Otimização de Código

Introdução

- Usar as informações coletadas pelas análises de fluxo de dados

- Tornar o código mais eficiente

- Inicialmente serão vistas:
  - Copy Propagation
  - Constant Propagation
  - CSE
  - Dead Code Elimination

Constant Propagation

Dadas as instruções:

i_1: $t = c$, onde c é constante

i_2: $y = t + x$

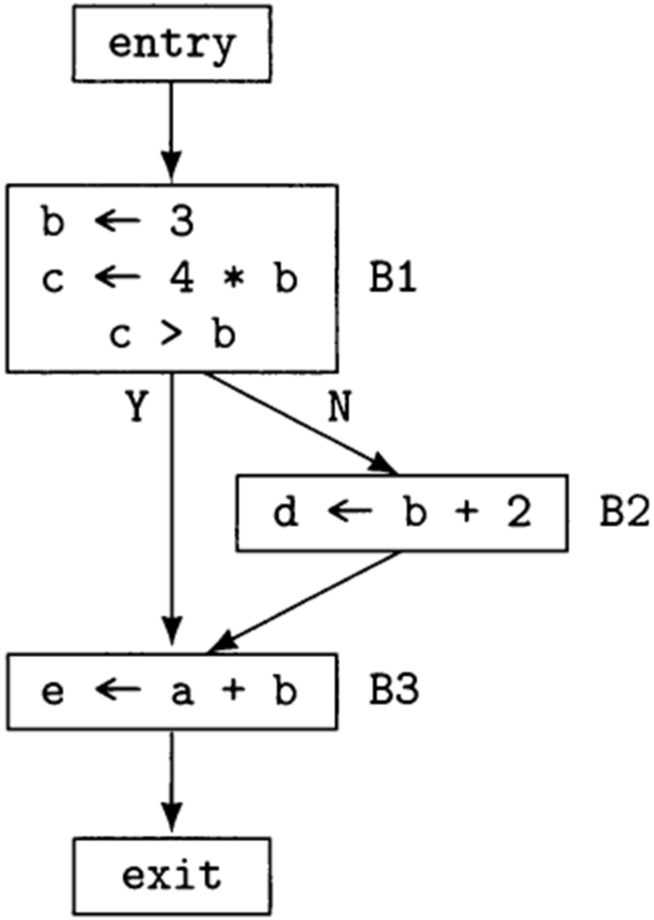
Pode-se afirmar que t é constante em i_2 se:

- i_1 alcança i_2 tal que **todo caminho** do início até i_2 contém i_1
- nenhuma outra definição de t alcança i_2

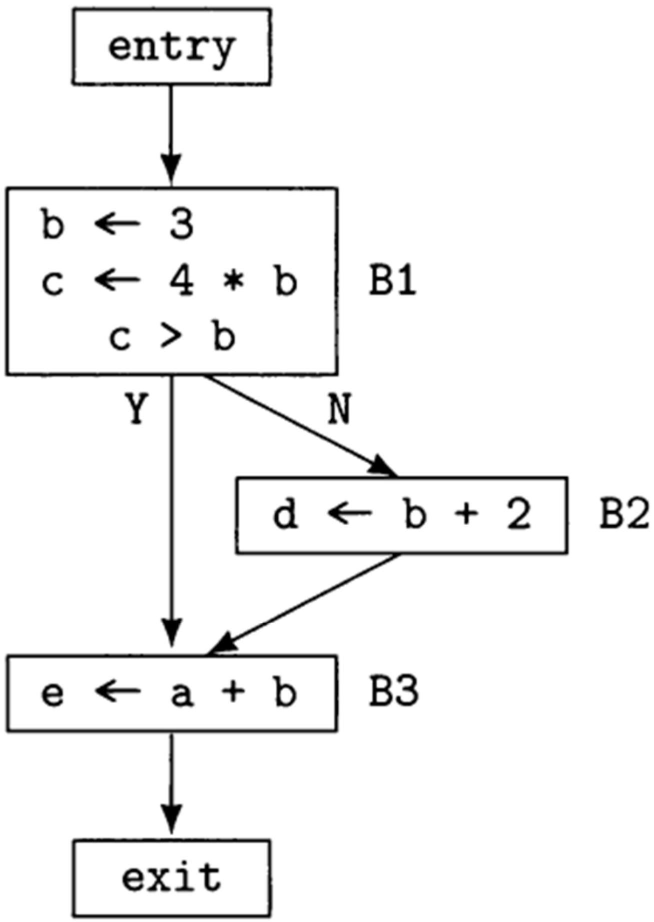
Pode-se reescrever:

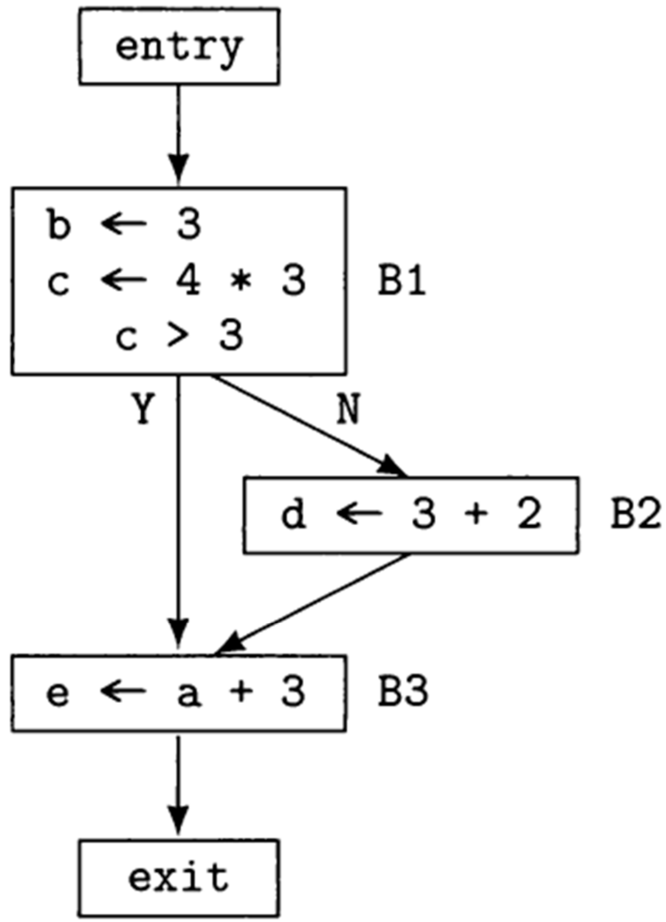
i_2: $y = c + x$

Constant Propagation



Constant Propagation





Dead Code Elimination

Se existem instruções do tipo:

```
i:   t   =   x + y
i:   t   =   M[x]
```

De maneira que `t` não está vivo após `i`,
então `i` pode ser eliminada.

Instruções com efeito colateral:

- Podem provocar alteração no resultado do programa se forem removidas
- O código pode funcionar com o otimizador desligado e não funcionar com ele ligado.

Dead Code Elimination

$$
\begin{aligned}
x &\leftarrow b + c \\
y &\leftarrow a + x \\
u &\leftarrow b + c \\
v &\leftarrow a + u \\
&f(v)
\end{aligned}
$$

Copy Propagation

- Dadas as instruções:

  i_1: **t** = z , onde z é variável

  i_2: y = **t** + x

- A variável t será “constante” em i_2 se:

  1. i_1 alcança i_2
  2. nenhuma outra definição de **t** alcança i_2
  3. não existe definição de **z** em qualquer caminho de i_1 a i_2, incluindo passagens sobre i_2 uma ou mais vezes

- Pode-se reescrever:

  i_2: y = **z** + x

Copy Propagation

- As Condições 1 e 2 podem ser checadas utilizando-se *ud-chains*

- Condição 3:
  - Nova *dataflow analysis*
    - c_in[B]: conjunto de cópias **s: x = y** tais que todo caminho do início até o nó B contém a sentença ***s***, e após a última ocorrência de ***s*** não há atribuições a ***y***
    - c_out[B]: idem para o final de B

Copy Propagation

- Condição 3
  - Nova *dataflow analysis*

    - c_gen[B]: ***s*** ocorre em B e não há atribuição a ***y*** após ***s***

    - c_kill[B]: ***s*** é morta em B se ***x*** ou ***y*** são atribuídos em B e ***s*** não está em B
      - Nota-se que diferentes atribuições ***x = y*** matam umas as outras
      - c_in[B] só pode conter uma sentença ***x = y*** com ***x*** à esquerda

Copy Propagation

- **Condição 3**
  - Equações: as mesmas de *available expressions*!
  - É chamada de Cópias Disponíveis

$$in[B] = \bigcap_{P \in Pred(B)} out[P]$$

$$in[B1] = \emptyset$$

$$out[B] = c\_gen[B] \cup (in[B] - c\_kill[B])$$

Copy Propagation

Algorithm 10.6. Copy propagation.

Input. A flow graph G, with ud-chains giving the definitions reaching block B, and with $c\_in[B]$ representing the solution to Equations 10.12, that is, the set of copies **x:=y** that reach block B along every path, with no assignment to

chains giving the uses of each definition.

Output. A revised flow graph.

Method. For each copy s: **x:=y** do the following.

1. Determine those uses of **x** that are reached by this definition of **x**, namely, s: **x:=y**.
2. Determine whether for every use of **x** found in (1), s is in $c\_in[B]$, where B is the block of this particular use, and moreover, no definitions of **x** or **y** occur prior to this use of **x** within B. Recall that if s is in $c\_in[B]$, then s is the only definition of **x** that reaches B.
3. If s meets the conditions of (2), then remove s and replace all uses of **x** found in (1) by **y**. □

Copy Propagation

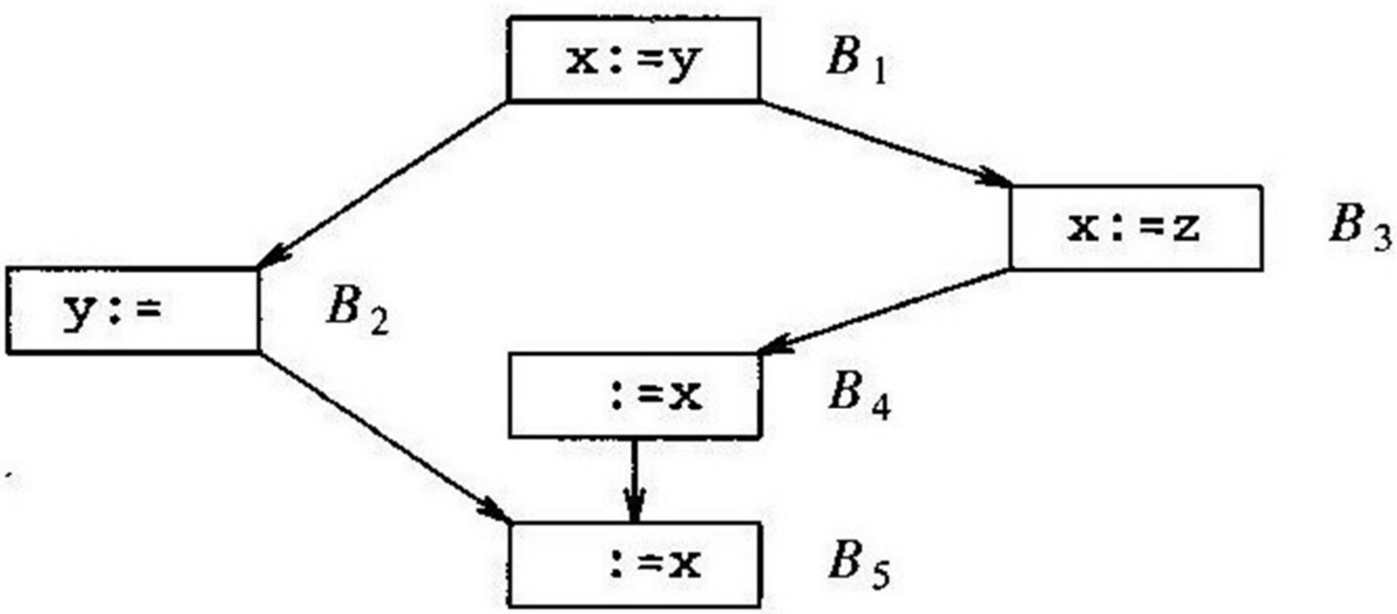


Fig. 10.35. Example flow graph.

Copy Propagation

c_gen/c_kill:

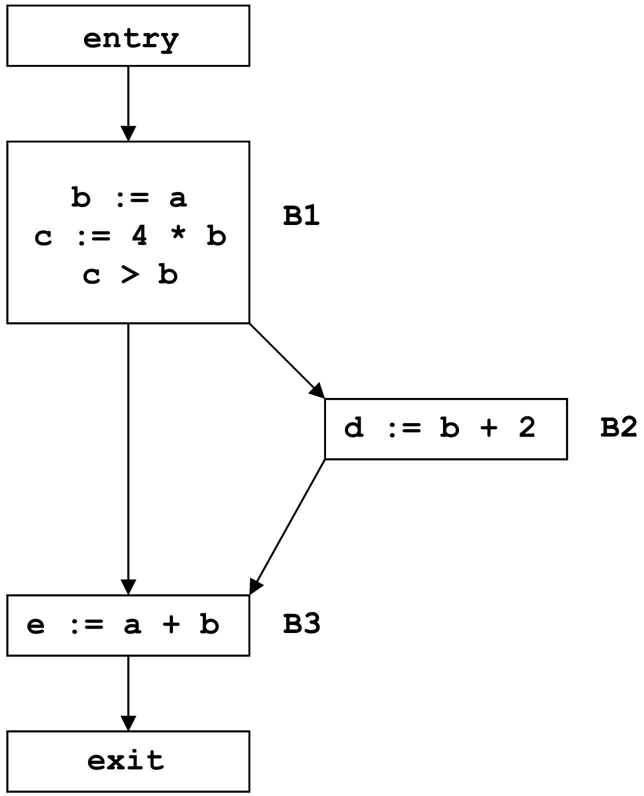
```
c_gen[B1] = {x=y}          c_kill[B1] = {x=z}
c_gen[B2] = {}             c_kill[B2] = {x=y}
c_gen[B3] = {x=z}          c_kill[B3] = {x=y}
```

- Os outros são vazios

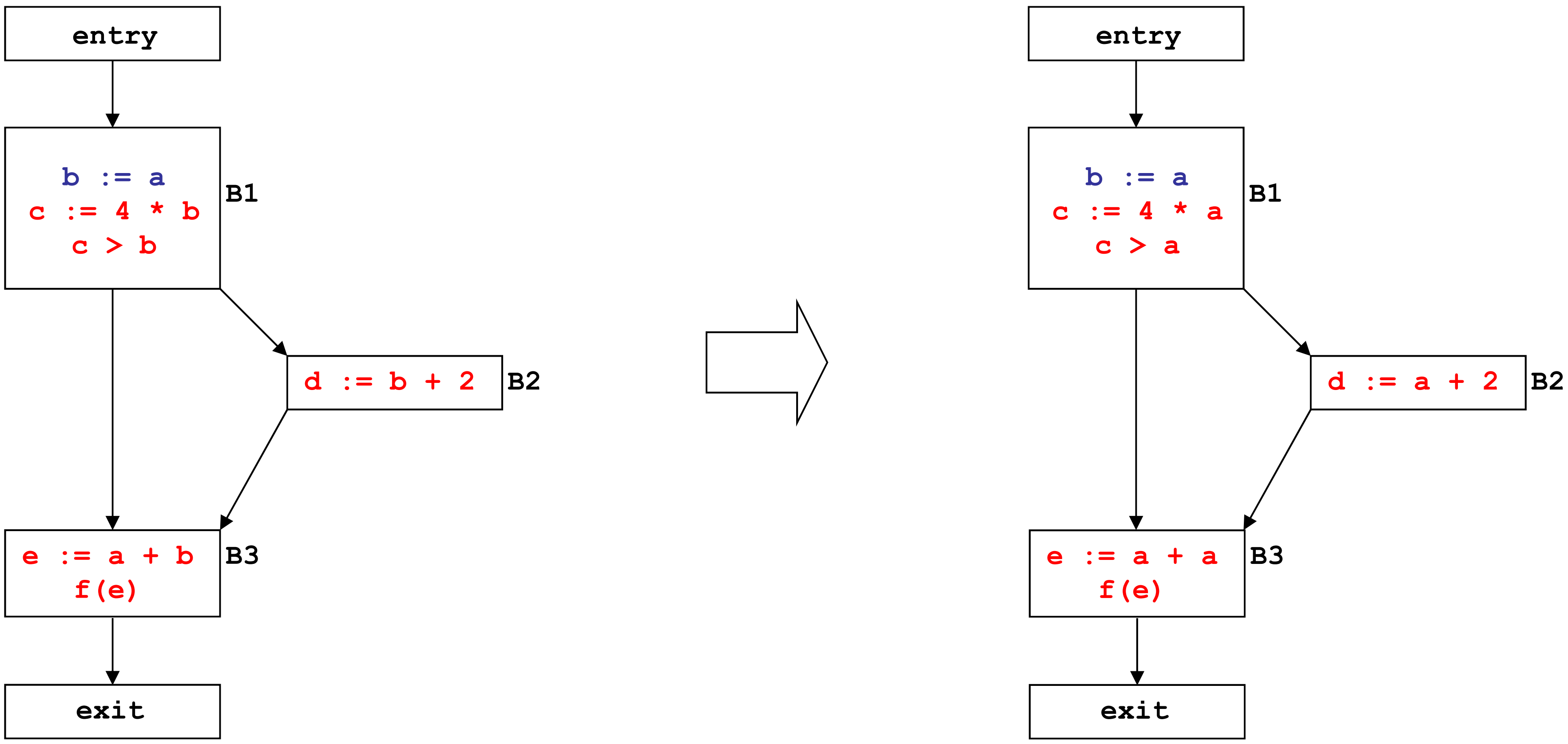
in/out:

```
in[B1] = {}                out[B1] = {x=y}
in[B2] = {x=y}             out[B2] = {}
in[B3] = {x=y}             out[B3] = {x=z}
in[B4] = {x=z}             out[B4] = {x=z}
in[B5] = {}                out[B5] = {}
```

Copy Propagation



Copy Propagation



Dead Code Elimination