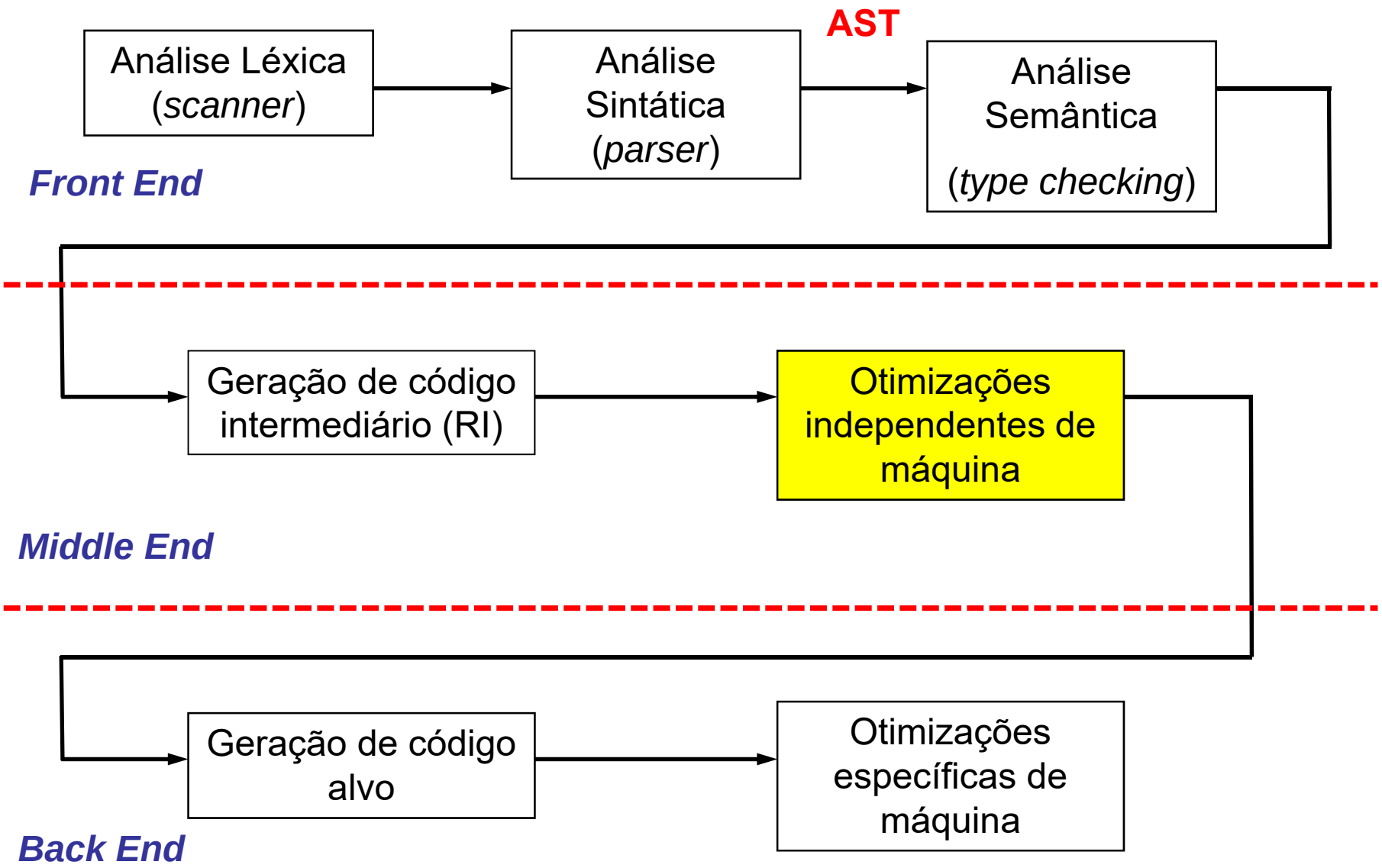
Fluxo do Compilador



Conceitos de Otimização de Código

Introdução

- **Melhorar o algoritmo é tarefa do programador**

- **O compilador pode ser útil para:**
  - Aplicar transformações que tornam o código gerado mais eficiente
  - Deixa o programador livre para escrever um código limpo

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);
}
```

Fig. 10.2. C code for quicksort.

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

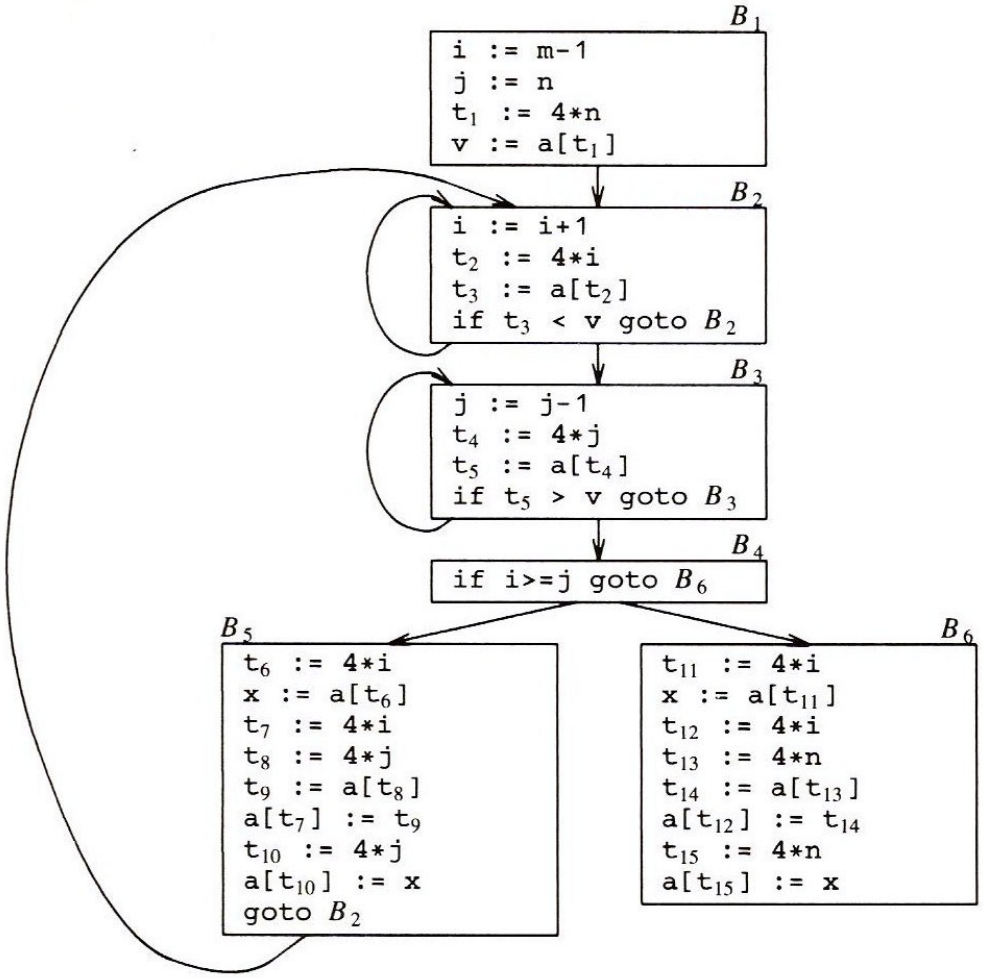


Fig. 10.5. Flow graph.

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

Local CSE

- *E* é sub-expressão comum se
  - *E* foi previamente computada
  - Os valores usados por E não sofreram alterações

B_5

```
t6 := 4*i
x := a[t6]
t7 := 4*i
t8 := 4*j
t9 := a[t8]
a[t7] := t9
t10 := 4*j
a[t10] := x
goto B2
```

(a) Before

B_5

```
t6 := 4*i
x := a[t6]
t8 := 4*j
t9 := a[t8]
a[t6] := t9
a[t8] := x
goto B2
```

(b) After

Fig. 10.6. Local common subexpression elimination.

Global CSE

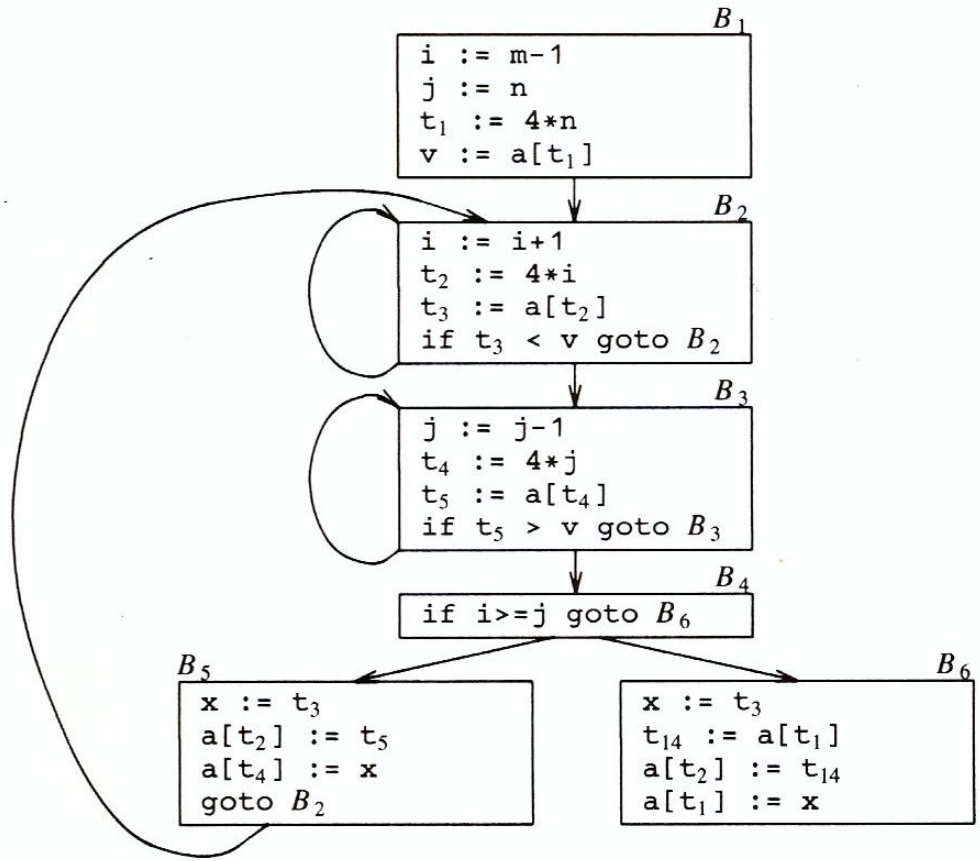


Fig. 10.7. B_5 and B_6 after common subexpression elimination.

Copy Propagation

- Voltemos ao bloco B5
  - Dá para melhorá-lo ainda mais?

Dead Code Elimination

- Código morto
  - Sentenças que computam valores que nunca são usados
- Pode ser inserido
  - Pelo programador
    - If(debug) {...}
  - Por outras transformações
    - Copy propagation
      - Bloco B5 do exemplo anterior

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

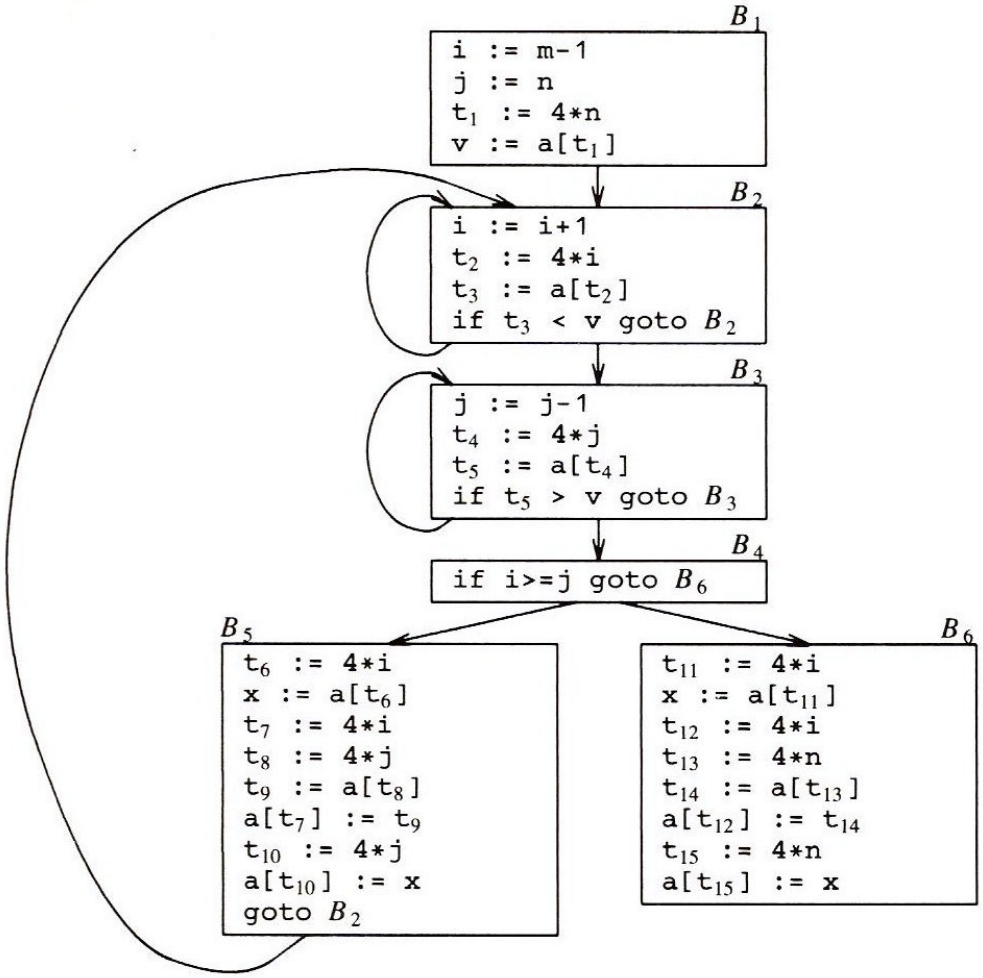


Fig. 10.5. Flow graph.

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

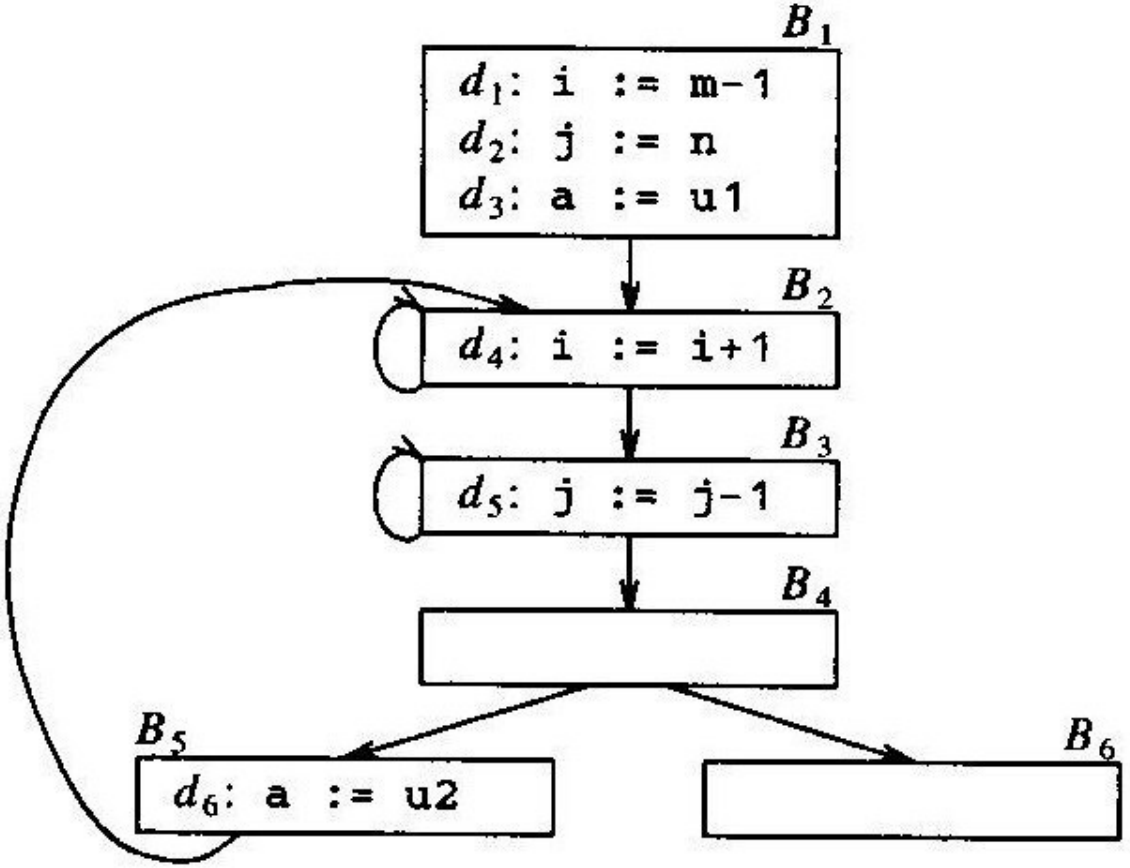


Fig. 10.19. A flow graph.

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.

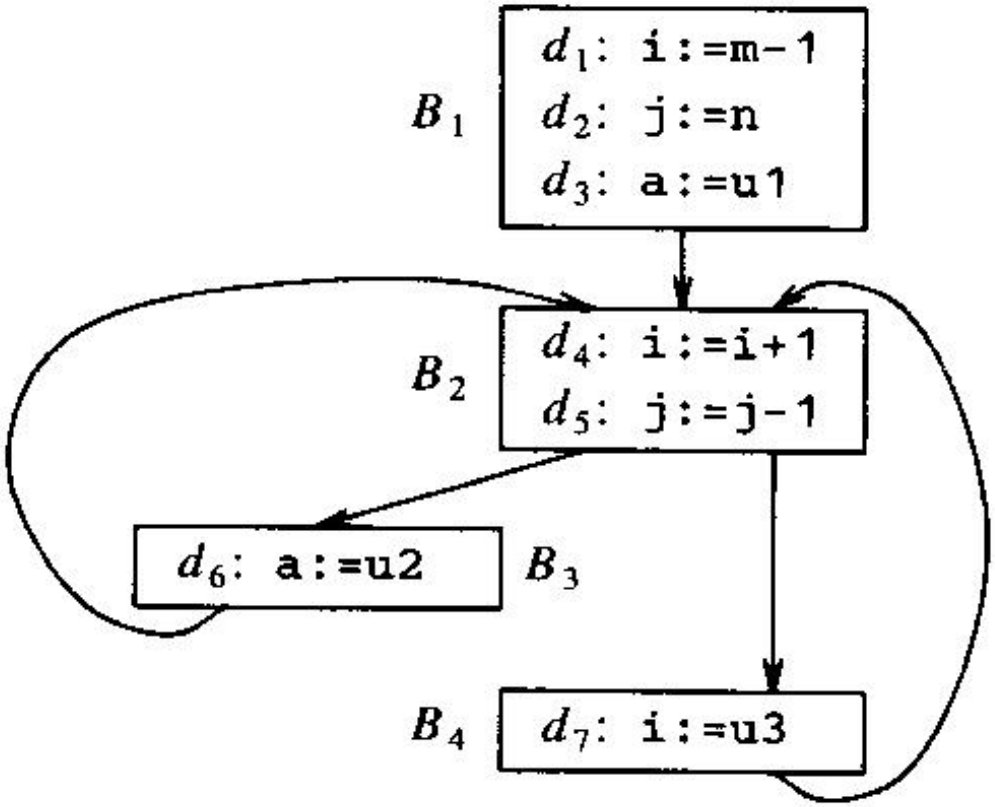
- **defs(t)** ou $\mathbf{D_t}$: conjunto de todas as definições de **t**

Reaching Definitions

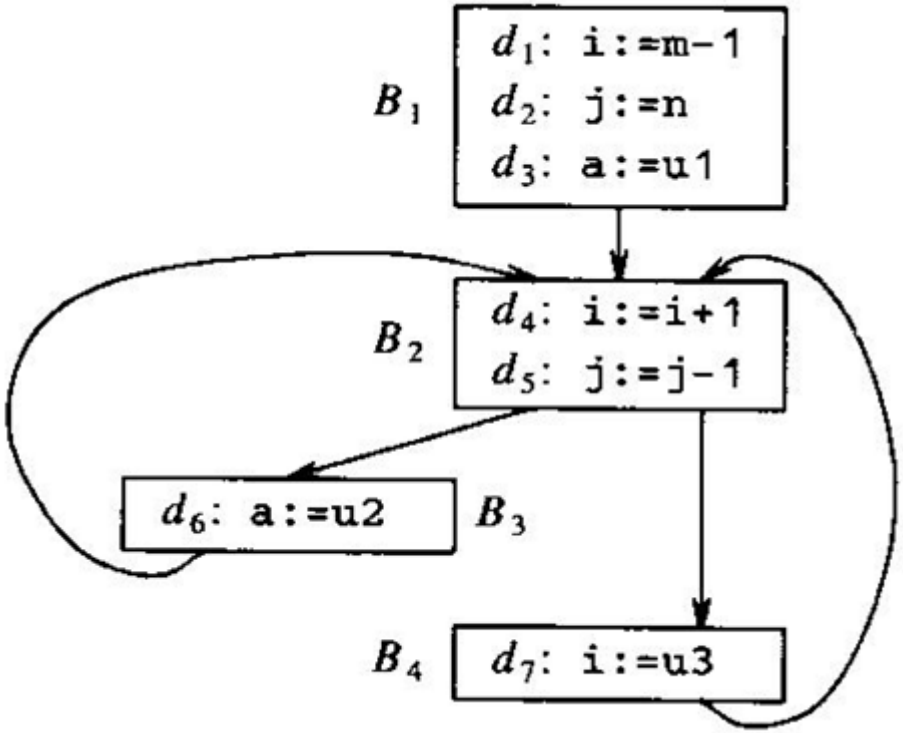
Principal uso:

- Dada uma variável $\mathbf{x}$ em um certo ponto $\boldsymbol{p}$ do programa, pode-se inferir que o valor de $\mathbf{x}$ 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:

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

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

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 predecessor 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
```

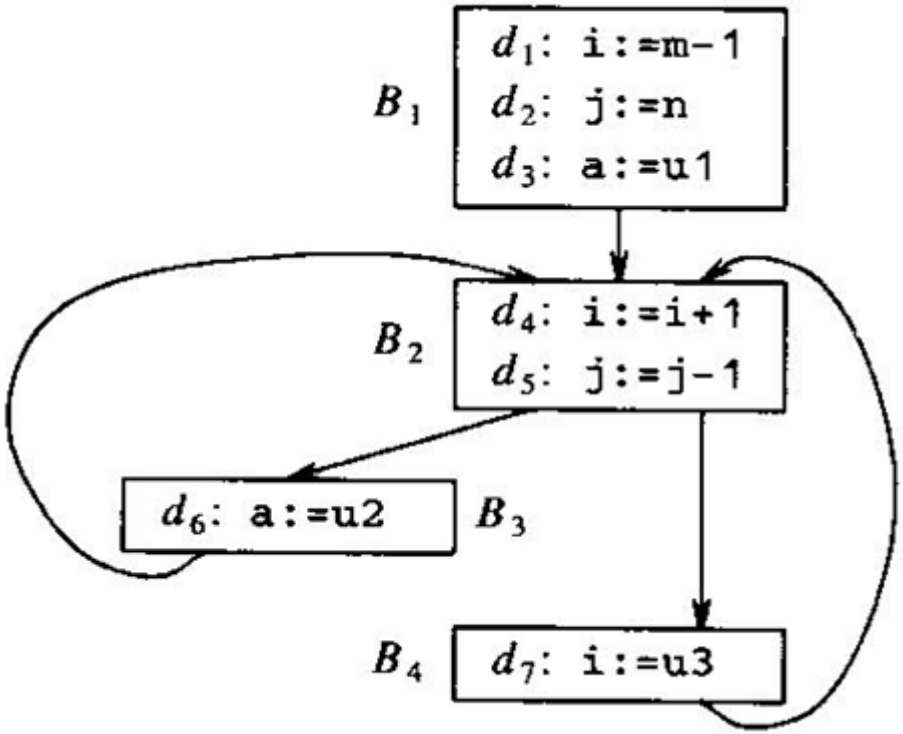
Fig. 10.26. Algorithm to compute *in* and *out*.

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



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

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

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

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