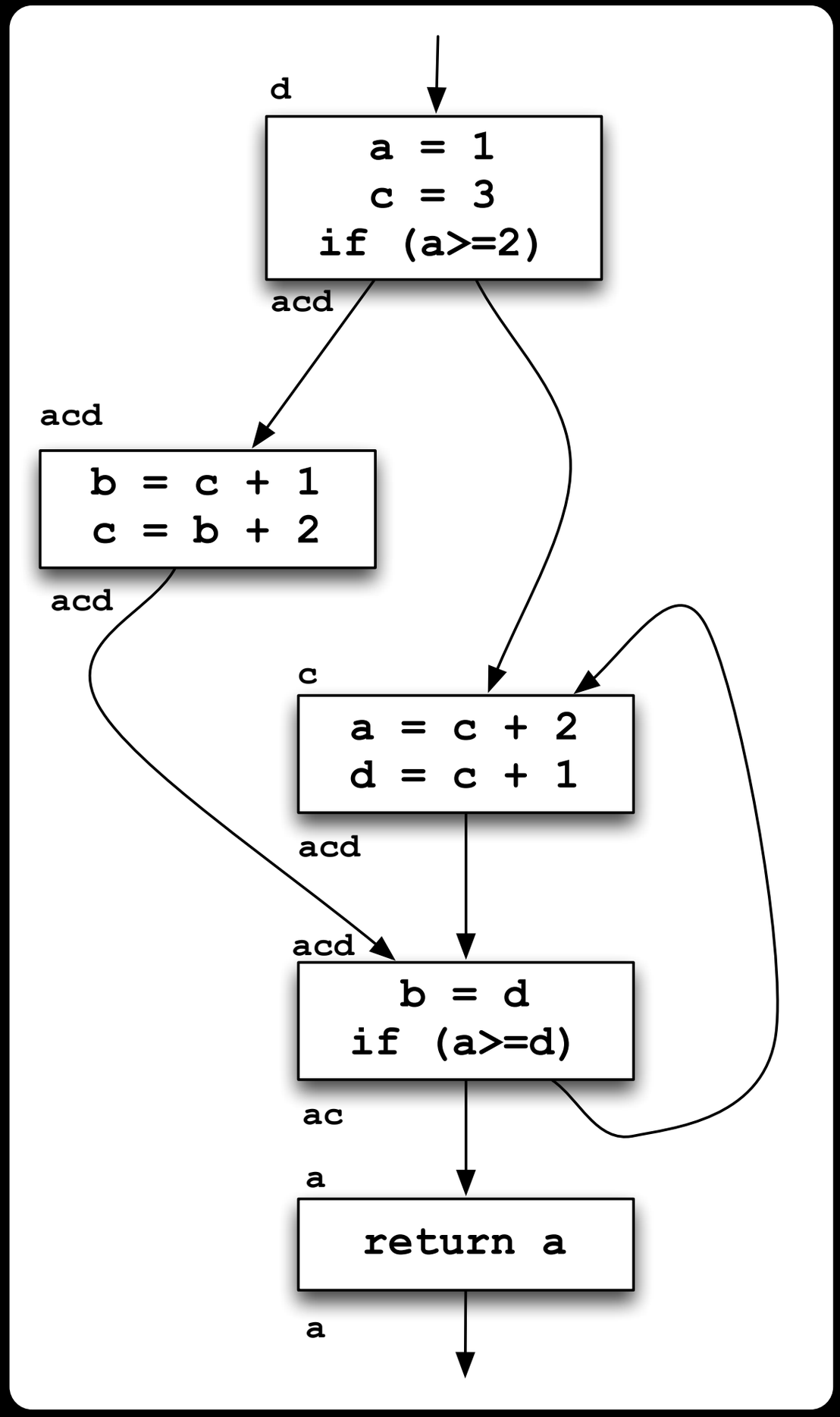
Compiladores
(IF688)

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

Considere o seguinte fragmento de código de três endereços:

- Construa o CFG usando as regras para criar blocos básicos;
- Informe o conjunto de variáveis vivas na entrada e saída de cada bloco básico do CFG, assumindo que apenas **a** é viva na saída do CFG.

```
     a = 1
     c = 3
     if a >= 2 goto L2
     b = c+1
     c = b+2
L1:  b = d
     if a >= d goto L2
     return a
L2:  a = c+2
     d = c+1
     goto L1
```


d
a = 1
c = 3
if (a>=2)
acd
acd
b = c + 1
c = b + 2
acd
c
a = c + 2
d = c + 1
acd
acd
b = d
if (a>=d)
ac
a
return a
a

Como encontrar variáveis não inicializadas com *liveness*?

Unitialized variables

- Considere uma variável v. Se $v \in \text{Live}_{in}[B]$,
  - onde B é o nó de entrada do CFG
- Existe um caminho de B até um uso de v em que v não é definida
- Este uso pode receber um valor não inicializado

Pode gerar falsos positivos…

Falsos positivos

- Se ***v*** já existe antes da entrada no CFG, pode ter sido inicializada antes
- A análise não vai conseguir detectar isto
- Isso acontece com variáveis estáticas ou variáveis globais declaradas fora do escopo atual

Falsos positivos

- Se ***v*** é acessível por outro nome e inicializada por meio deste nome, análise não captura esta informação
- O uso de ponteiros, como no código ao lado, pode gerar uma situação destas

```
…
p = &x;
*p = 0;
…
x = x + 1;
…
```

Falsos positivos

- As equações podem descobrir um caminho da entrada do procedimento, até um uso de v, onde v não é inicializada

- Ainda que este caminho não seja possível de ser executado, a variável v aparecerá em $\mathrm{Live}_{in}[B]$.

- A análise considera todos os possíveis caminhos…

Falsos positivos

```
main() {
   int i, n, s;
   scanf(“%d”, &n);  i = 1;
   while (i<=n) {
      if (i==1) {
         s = 0;
      }
      s = s + i++;
   }
}
```

eficiência

vs.

precisão

Outros usos para *Liveness*

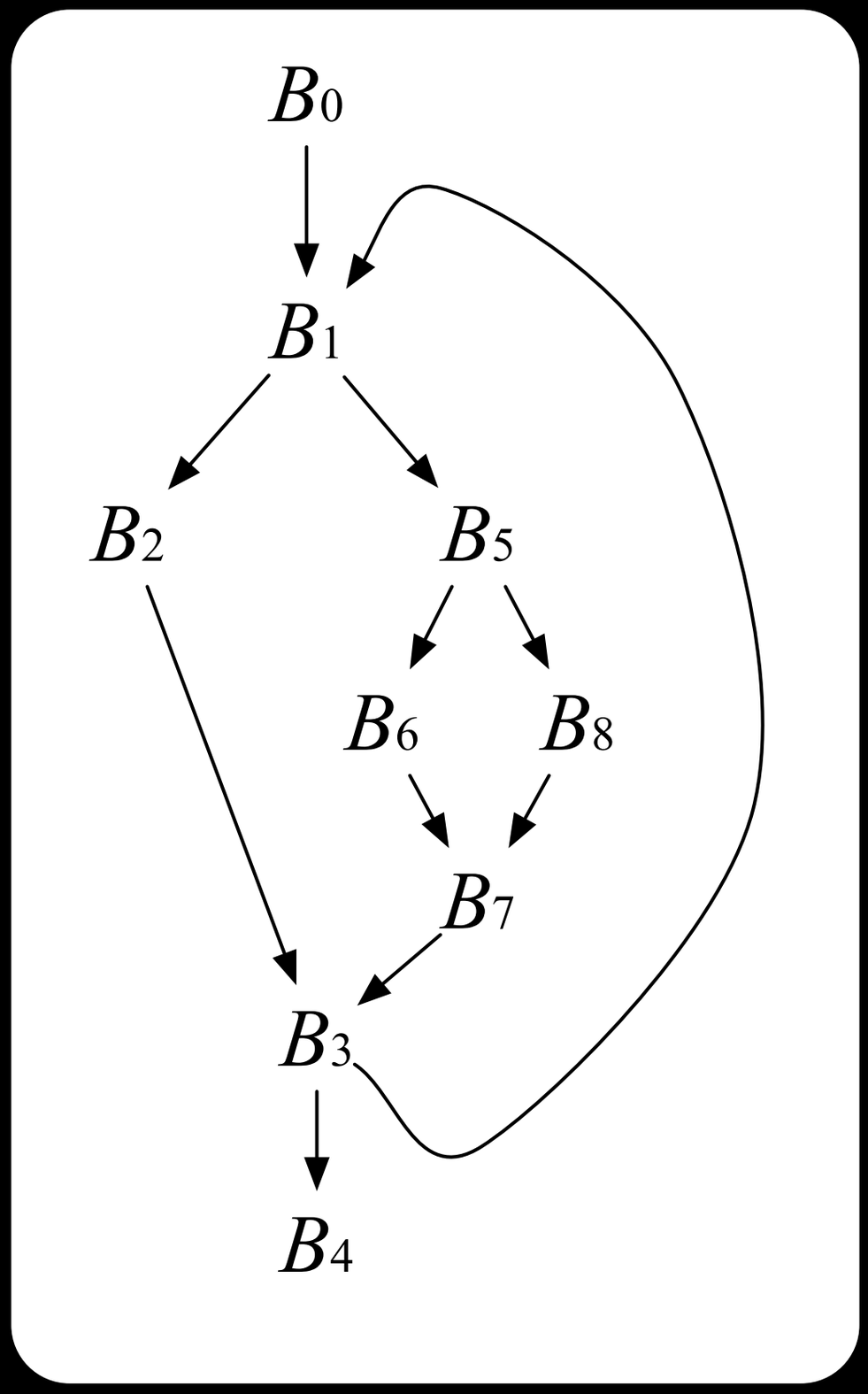
- Alocação global de registradores
  - não precisa manter valores em registradores se não estão live
    - ao transicionar de live para dead, pode liberar registrador
- Construção de SSA
- Stores desnecessários

Dominance

Dominance

- Muitas técnicas de otimização raciocinam sobre propriedades estruturais do CFG
- A noção de *dominators* é importante neste caso
- Podemos calcular informação de *dominance* usando análise de fluxo de dados
- $\mathbf{Dom}[\mathbf{B}_1]$ é o conjunto de todos os nós que dominam $\mathbf{B}_1$

Conjunto de nós que estão presentes em qualquer caminho a partir do nó inicial até o nó B



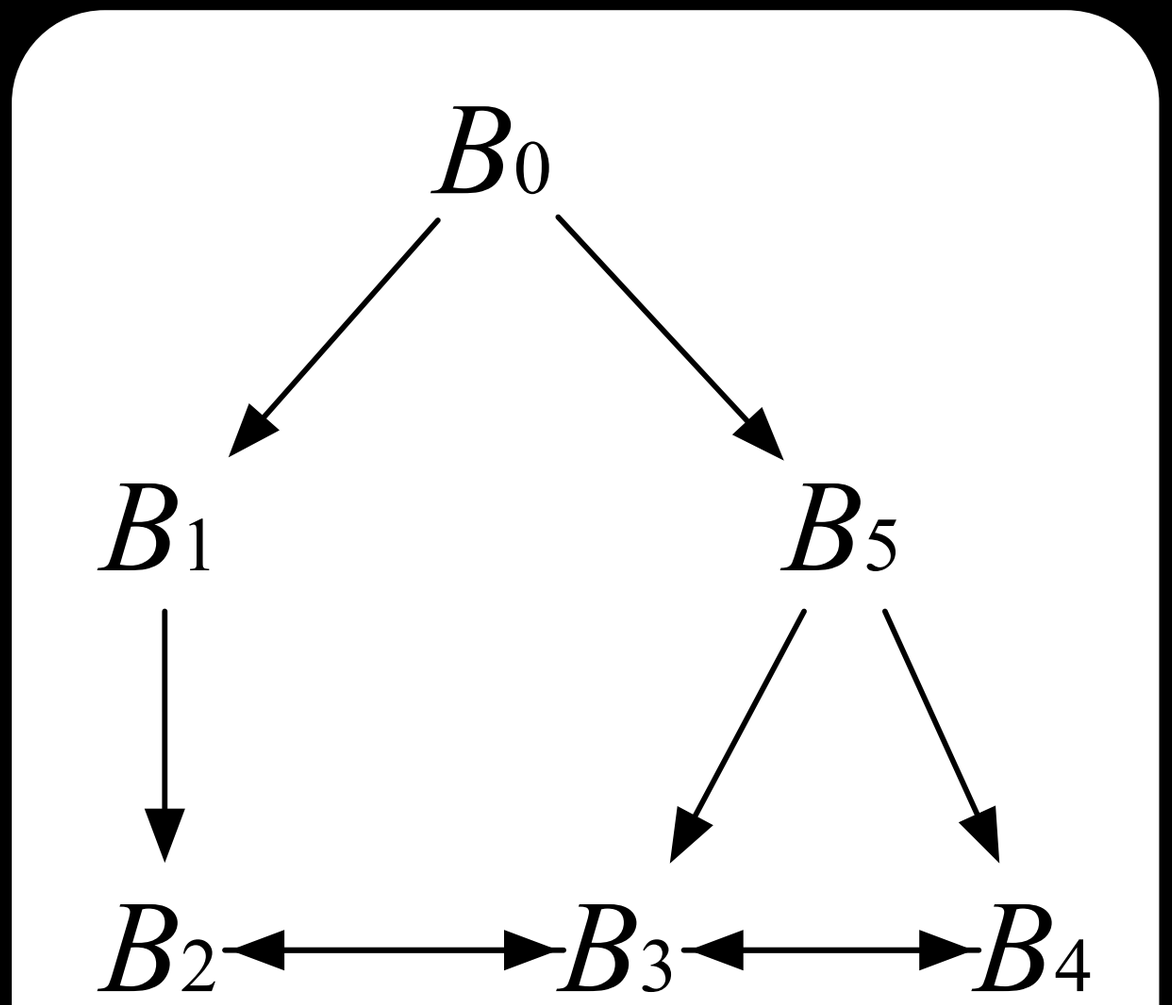
$\text{Dom}[B_6] = \{B_0, B_1, B_5, B_6\}$

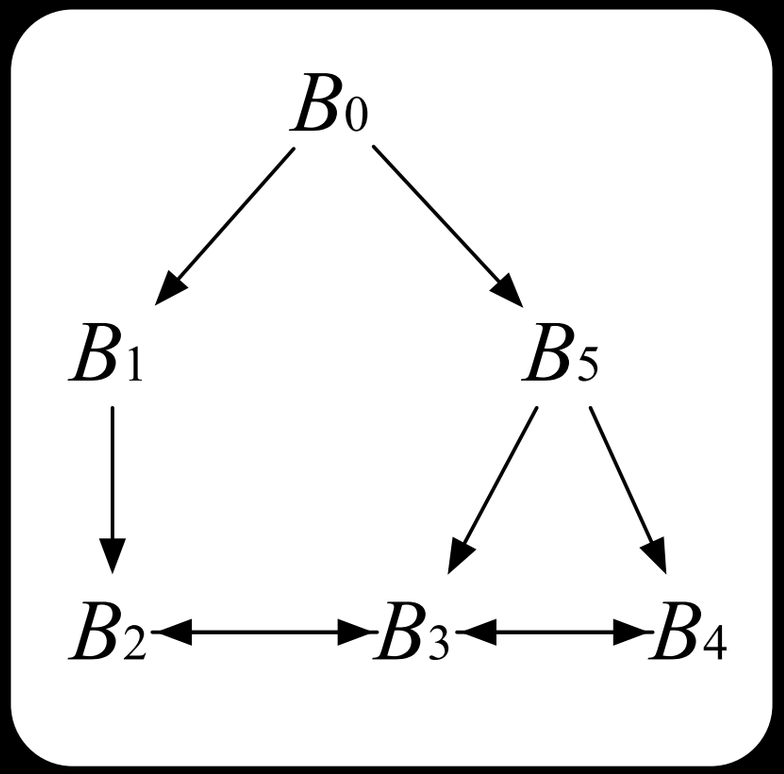
Calculando…

- $\mathrm{Dom}[B] = \{B\} \cup (\cap_{S \in preds(B)} \mathrm{Dom}[S])$
  - condições iniciais são:
    - $\mathrm{Dom}[ENTRY] = \{ENTRY\}$
    - $\forall\ B \cdot B \neq ENTRY \rightarrow \mathrm{Dom}[B] = N$
      - *onde N é o conjunto de todos os nós do CFG*
- Cálculo de DOM é feito com função nos predecessores, *forward-flow*

Termination

- O algoritmo encerra pois os conjuntos diminuem monotonicamente durante a computação
- Um conjunto DOM não pode ser menor do que um elemento (próprio nó) e não pode ter mais nós que o total do CFG
- O conjunto só diminui, não tem como aumentar


B_0
B_1
B_5
B_2
B_3
B_4



	B_0	B_1	B_5	B_2	B_3	B_4
—	{0}	N	N	N	N	N
1	{0}	{0,1}	{0,5}	{0,1,2}	{0,3}	{0,4}
2	{0}	{0,1}	{0,5}	{0,2}	{0,3}	{0,4}
3	{0}	{0,1}	{0,5}	{0,2}	{0,3}	{0,4}

Limitações de Análise Estática

- Maioria se resume a questões de precisão
  - Assume que todos os caminhos são possíveis
  - Referências a *Arrays*: Qual elemento estamos acessando ao referenciar **`A[i]`**?
  - Uso de Ponteiros, principalmente quando código envolve aritmética de ponteiros
  - Chamadas de procedimentos

Forward vs. Backward

- *Constant propagation* é uma *forward analysis*
  - informação foi passada de *in* para *out*
- *Liveness* é uma *backward analysis*
  - informação foi passada de *out* para *in*

Outras análises

- Existem outras *global flow analyses*, que habilitam outras otimizações
  - *Available Expressions: Common subexpression elimination*
  - *Reaching definitions: loop invariant code motion*
  - *Definite assignment: null check elimination*
- A maioria pode ser classificada como *forward* ou *backward*
- Também devemos seguir a metodologia de regras simples relacionando informação entre pontos adjacentes do programa
  - ***daí a necessidade de uma IR simples…***