Compiladores
(IF688)

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

Alocação e Atribuição de Registradores

- Instruções envolvendo registradores são mais rápidas do que as que envolvem operandos na memória
  - pelo menos uma ordem de magnitude
- Utilização eficiente dos registradores é vital para geração de código bom

Abordagem Simplista

- Separar registradores de acordo com alguns tipos de valores específicos
- Por exemplo, computações aritméticas são sempre realizadas em um grupo específico de registradores, outro grupo para armazenar outros tipos de valores, topo da pilha, etc…
- Vantagens vs. desvantagens?

De qualquer forma, é comum ter um grupo limitado de registradores para fins específicos

Alocação (*global*) de Registradores

- Na aula passada, vimos como usar registradores para guardar valores no contexto de um único bloco básico

- Ao fim do bloco, todas as variáveis *vivas*, eram armazenadas na memória novamente

- Podemos evitar alguns destes *stores* (e os *loads* correspondentes) mantendo variáveis frequentemente utilizadas em registradores de forma consistente entre blocos (*globalmente*)

Alocação (*global*) de Registradores

- Dado que programas passam boa parte de seu tempo em *inner loops*, uma abordagem natural é tentar manter um valor usado frequentemente em um registrador fixo durante o laço

- A partir do código de três endereços, podemos representar um *flow graph* e descobrir (usando análise estática) que variáveis computadas em um bloco básico são usadas em outro(s) blocos

Possível Estratégia

- Assinalar número fixo de registradores para valores mais ativos em cada *inner loop*
- Os valores selecionados podem ser diferentes em *loops* diferentes
- Registradores livres podem ser utilizados da forma já discutida em blocos básicos
- Desvantagem é fixar um número, nem sempre será o mais correto/vantajoso

Contagem de Usos

- A ideia é estimar o quanto uma variável **x** é utilizada no programa, para calcular a economia (aproximada) ao mantê-la em um registrador durante um *loop* L

- A economia se dá em cada referência (uso) a **x**, se **x** já estiver em um registrador

- A estratégia discutida anteriormente visa manter valores nos registradores se forem usados novamente no mesmo bloco básico

‘crédito’

- Economizamos uma unidade para cada uso de **`x`** em um loop **`L`** que não tenha sido precedido por uma atribuição a **`x`**, no mesmo bloco
- Economizamos duas unidades se podemos evitar um *store* de **`x`** ao final do bloco básico
  - ou seja, se **`x`** estiver *live* na saída de cada bloco do loop **`L`** onde um valor é assinalado

‘débito’

- Se **x** estiver *live* na entrada do *loop*, teremos de fazer um *load* de **x** no registrador, a um custo de duas unidades, antes de iniciar a execução do *loop* **L**

- Da mesma forma, para cada bloco de saída **B** do loop **L**, onde **x** estiver *live* para o sucessor do bloco **B**, devemos realizar o store de **x** a um custo de duas unidades

‘débito’

- No entanto, assumindo que teremos várias iterações do *loop* `L`, estes ‘débitos’ são negligenciáveis
- Ocorrem apenas uma vez em toda a execução do *loop*

Fórmula Aproximada

$$\sum_{B \in L} use(x, B) + 2 * live(x, B)$$

- *use(x,B)*: quantidade de vezes em que x é utilizada em B antes de qualquer definição de x

- *live(x,B)*: 1, se x for *live* na saída de B e há alguma atribuição a x em B; 0 do contrário

- aproximação, pois nem todos os blocos em um *loop* são executados com a mesma frequência e assumimos várias iterações do *loop*

Liveness

- Utilizamos a análise de live variables para saber se em um dado ponto **p**, uma variável **x** pode ser usada em um dos caminhos a partir de **p**
  - dependendo da resposta **x** é *live* ou *dead*
  - análise é *backwards*

Uso de *Liveness*

- Após um valor ser computado em um registrador, e, presumidamente, usado no bloco, não é necessário fazer um *store* se o valor for dead ao fim do bloco

- Se não há nenhum registrador vazio e é preciso utilizar um registrador, podemos favorecer o uso de um registrador armazenando valor *dead*

Calculando *Liveness*

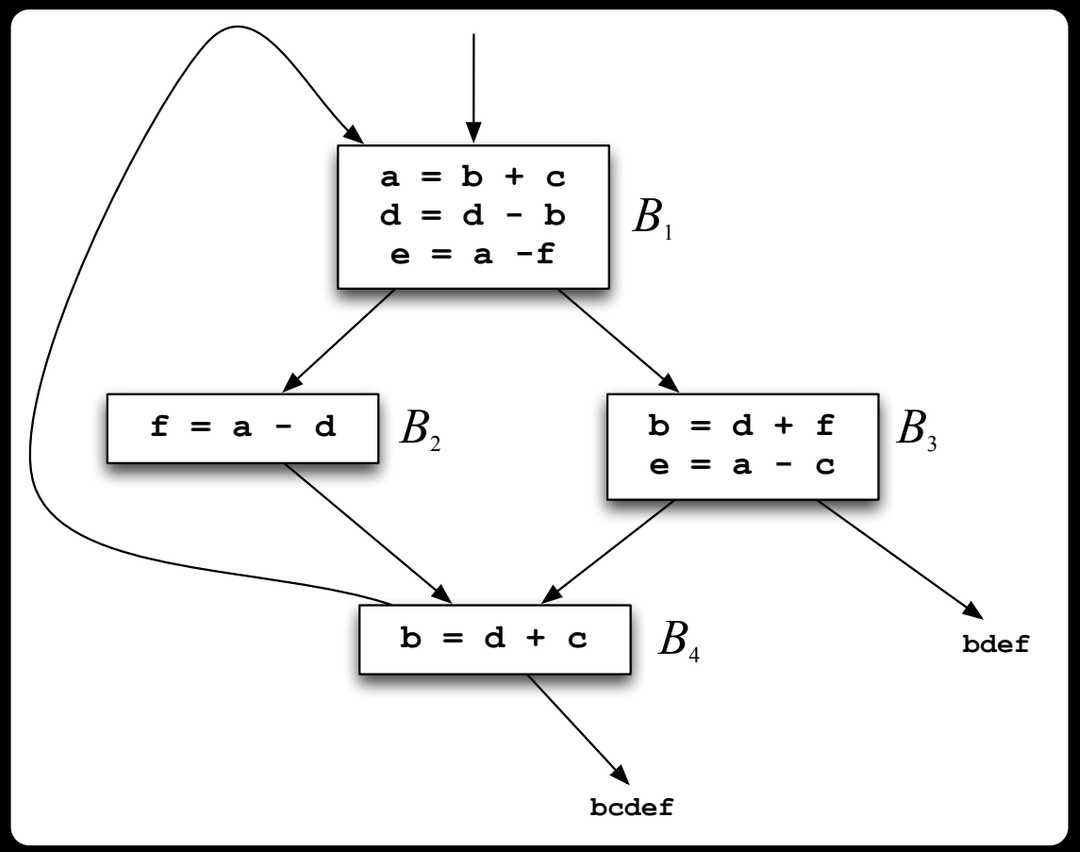
- Definimos as equações em termos das entradas e saídas de um bloco, isto é IN[B] e OUT[B]
  - conjunto de variáveis vivas nos pontos imediatamente antes e depois do bloco
- def_B é o conjunto de variáveis definidas em B antes de qualquer uso da variável em B
- use_B é o conjunto de variáveis cujos valores podem ser usados em B antes de qualquer definição

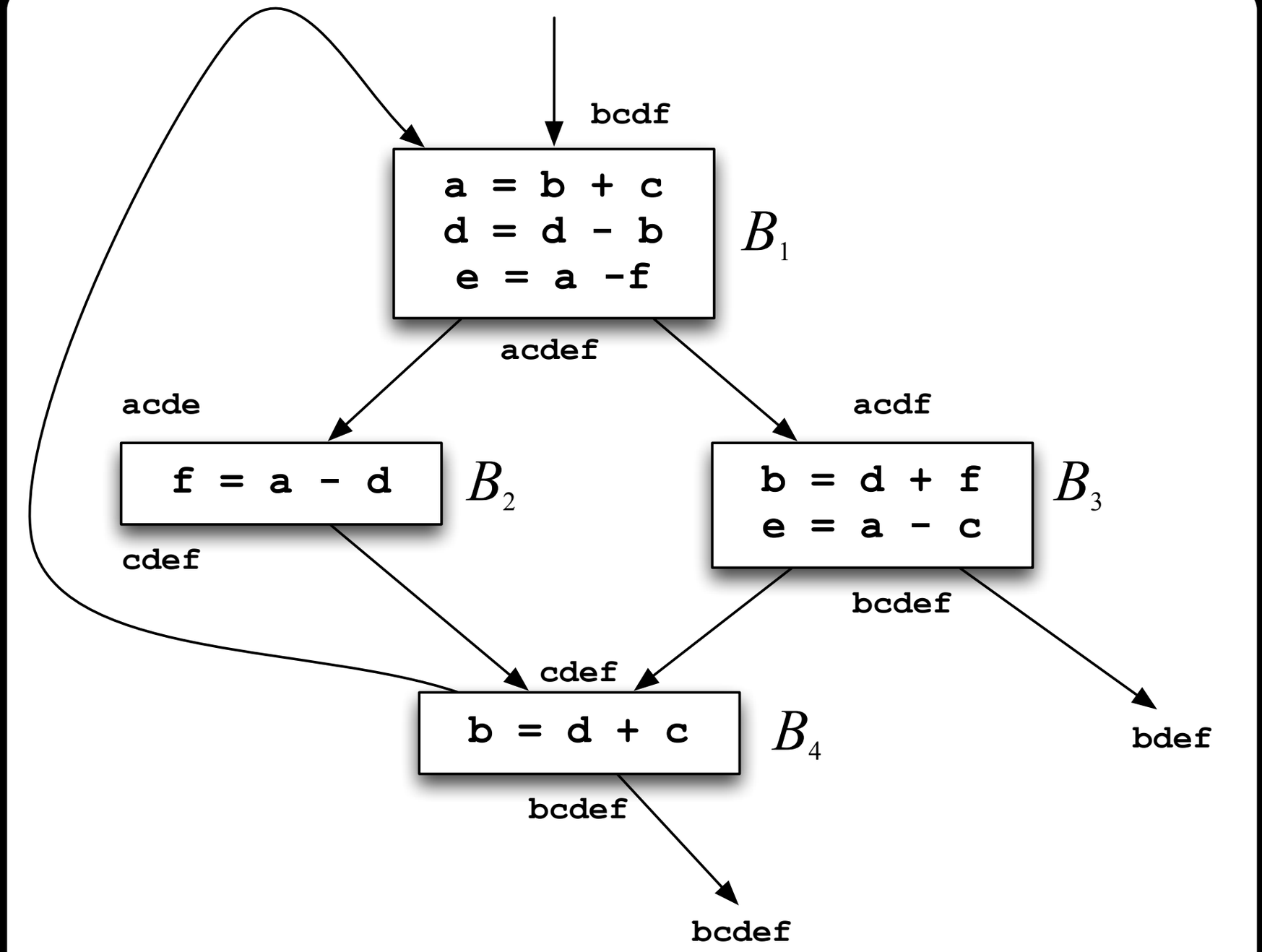
Usando as definições

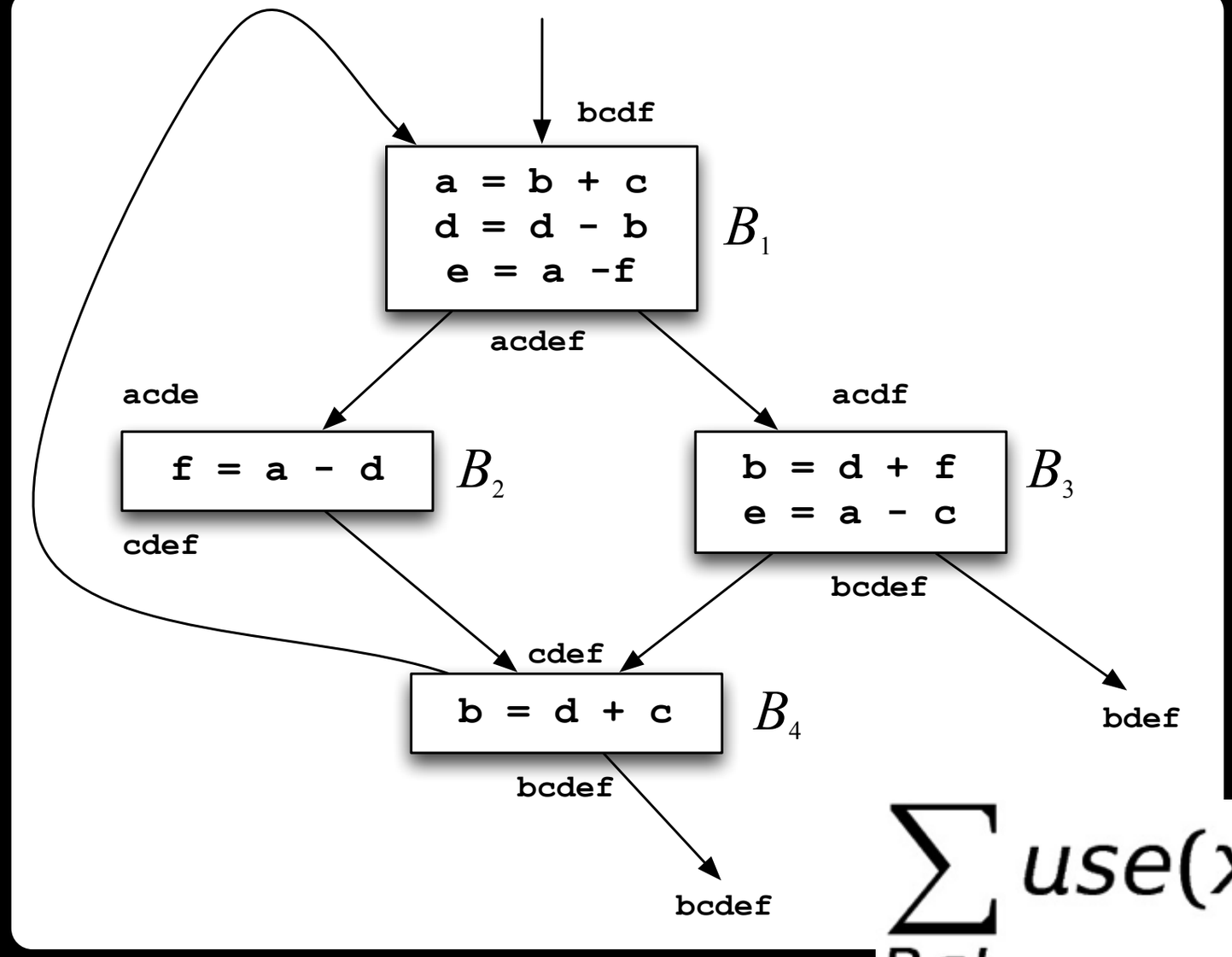
- Qualquer variável em use_B deve ser considerada *live* na entrada do bloco B
- Qualquer variável em def_B deve ser considerada *dead* na entrada do bloco B
  - estar em def_B mata qualquer oportunidade da variável ser considerada *live* em caminhos iniciados a partir do bloco B

Equações

- $\text{IN}[\text{EXIT}] = \varnothing$
  - condição de fronteira, nenhuma variável é *live* ao encerrarmos o programa
- $\text{IN}[B] = use_B \cup (\text{OUT}[B] - def_B)$
  - variável é *live* na entrada do bloco se é usada antes de ser redefinida, ou se está viva na saída e não é redefinida
- $\text{OUT}[B] = \cup_{S \text{ é sucessor de } B} \text{IN}[S]$
  - variável é *live* na saída do bloco se e somente se é *live* na entrada de seus sucessores


a = b + c
d = d - b
e = a -f
B_1
f = a - d
B_2
b = d + f
e = a - c
B_3
b = d + c
B_4
bdef
bcdef


bcdf
a = b + c
d = d - b
e = a -f
B_1
acdef
acde
acdf
f = a - d
B_2
b = d + f
e = a - c
B_3
cdef
bcdef
cdef
b = d + c
B_4
bdef
bcdef
bcdef



$$\sum_{B \in L} use(x, B) + 2 * live(x, B)$$

- a = ?
- b = ?
- c = ?
- d = ?
- e = ?
- f = ?

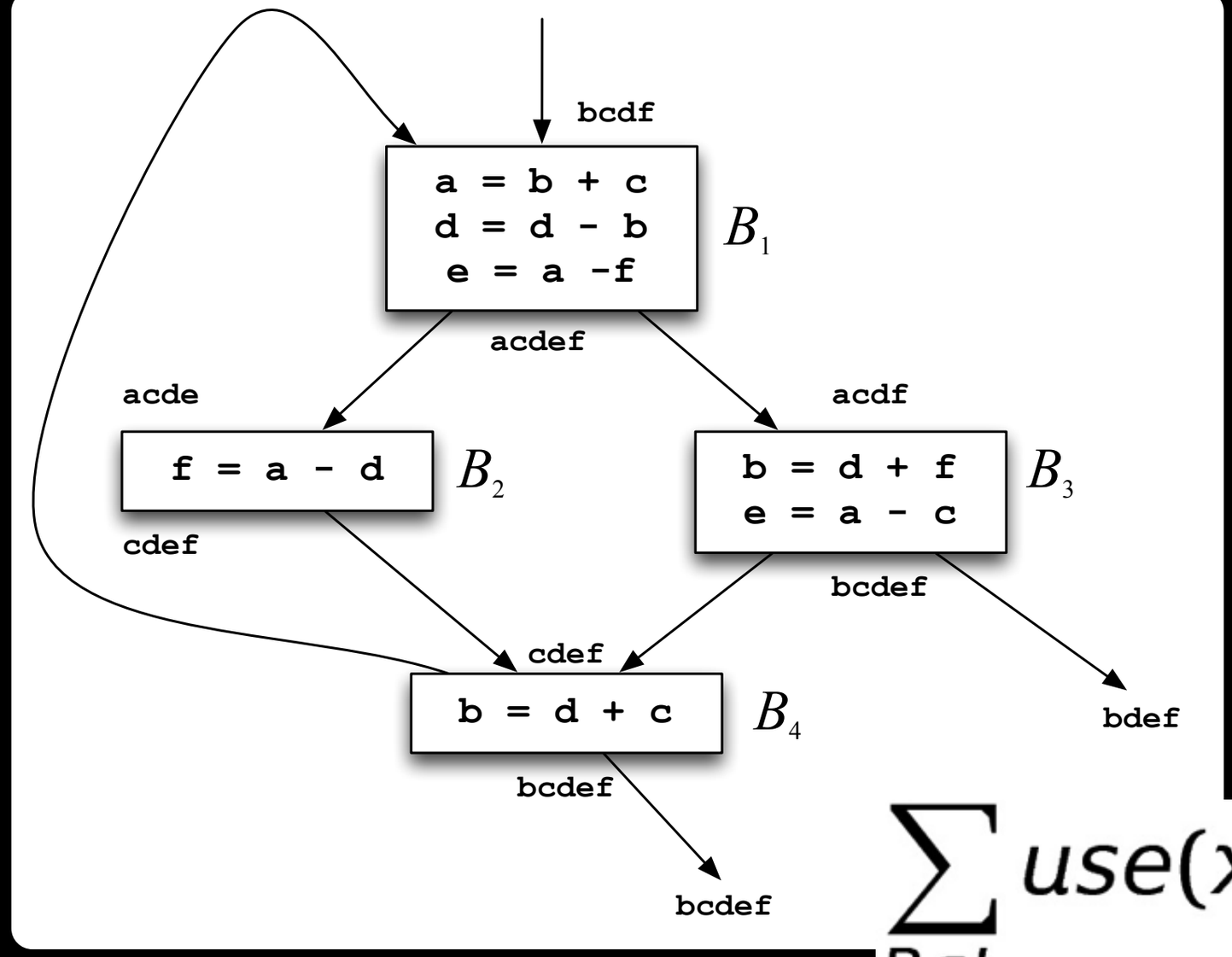
Fórmula Aproximada

$$\sum_{B \in L} use(x, B) + 2 * live(x, B)$$

- *use(x,B)*: quantidade de vezes em que x é utilizada em B antes de qualquer definição de x

- *live(x,B)*: 1, se x for *live* na saída de B e há alguma atribuição a x em B; 0 do contrário

- aproximação, pois nem todos os blocos em um *loop* são executados com a mesma frequência e assumimos várias iterações do *loop*



$$\sum_{B \in L} use(x, B) + 2 * live(x, B)$$

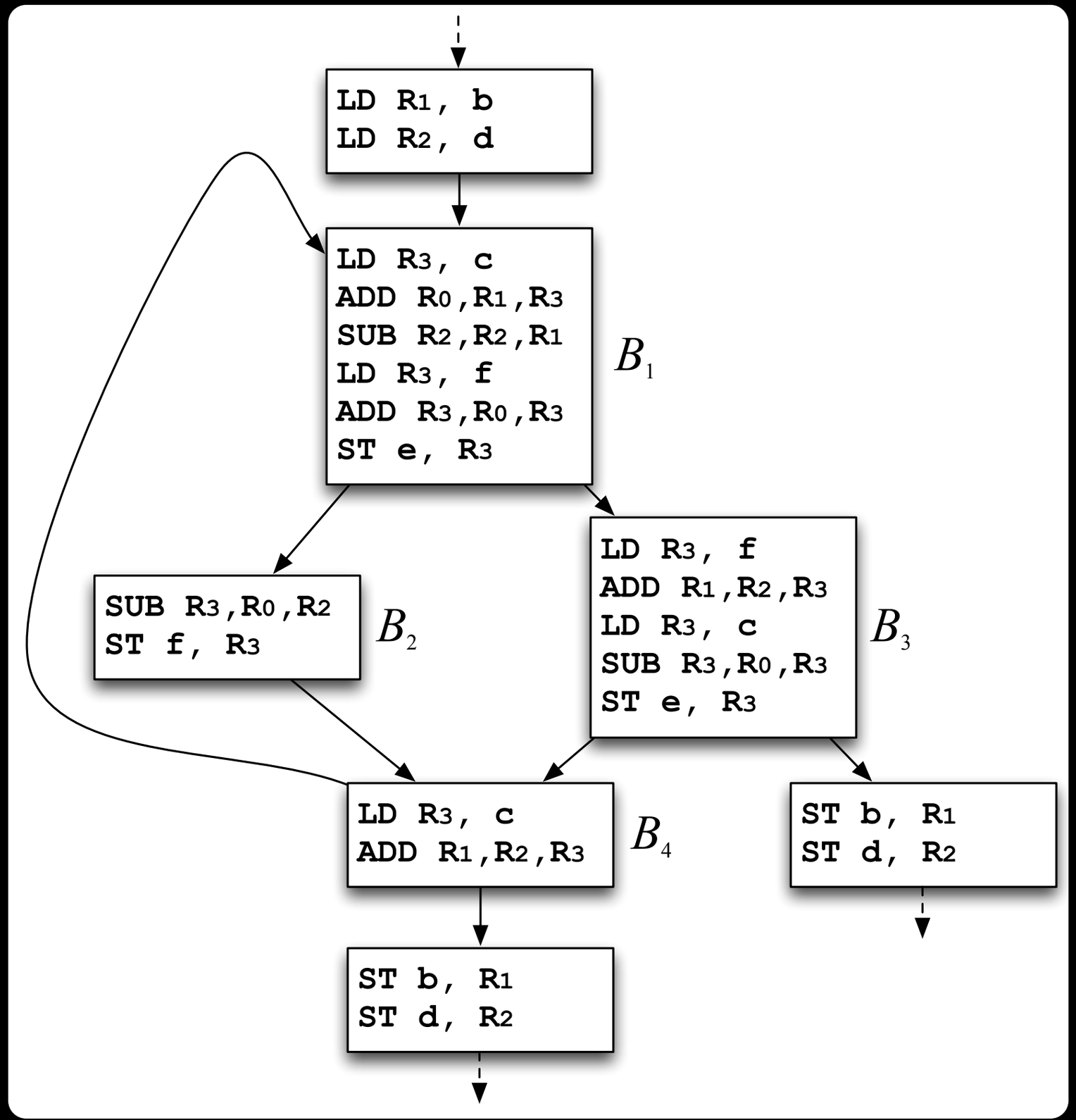
- a = 4
- b = 6
- c = 3
- d = 6
- e = 4
- f = 4

Usando estes valores

- O valor de uma variável **x** corresponde a uma estimativa de economia de custos ao guardar x *em um registrador durante o loop*

- No caso de **a**, quatro unidades de custo são economizadas mantendo **a** em registrador

- Assumindo que só temos três registradores, guardamos **b** e **d** durante todo o loop e utilizamos o registrador restante para demais variáveis


LD R1, b
LD R2, d
LD R3, c
ADD R0,R1,R3
SUB R2,R2,R1
LD R3, f
ADD R3,R0,R3
ST e, R3
B1
SUB R3,R0,R2
ST f, R3
B2
LD R3, f
ADD R1,R2,R3
LD R3, c
SUB R3,R0,R3
ST e, R3
B3
LD R3, c
ADD R1,R2,R3
B4
ST b, R1
ST d, R2
ST b, R1
ST d, R2

Atribuição em *outer loops*

- A mesma ideia aplicada para *inner loops* é utilizada para *loops* progressivamente maiores
- Se uma variável **x** for alocada em registrador para um loop L_2, contido em um loop L_1, devemos fazer o *load* de **x** na entrada de L_2 e o *store* na saída

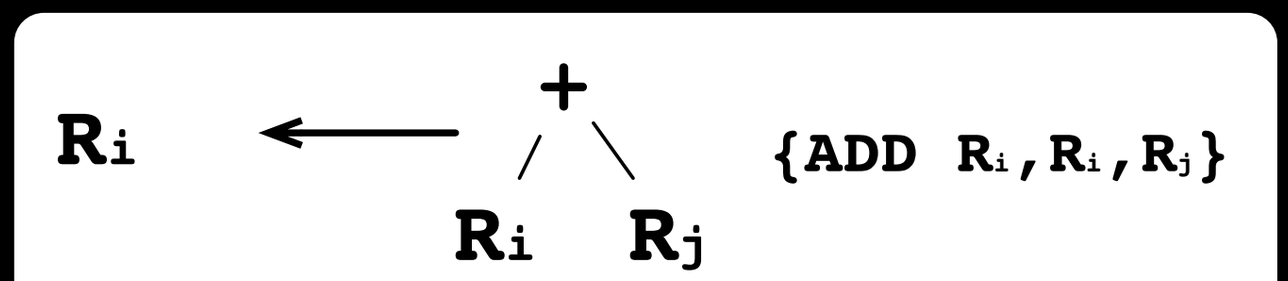
Alocação de registradores por meio de coloração de grafos

- A ideia é usar a técnica de coloração para gerenciar a alocação e *spilling* de registradores
- Se baseia na construção de um grafo de interferência, em que nós são registradores simbólicos e arestas conectam nós se o nó está live no ponto que o outro é definido
- O grafo é colorido com **k** cores, onde **k** é o número de registradores assinaláveis

Outras técnicas

- Seleção de instruções por meio de reescrita de árvores

replacement ⟵ *template* { *action* }



Outras técnicas

- Geração de código ótimo para expressões
- Ershov *numbers*
  - construir uma árvore para a expressão
  - associar *labels* aos nós
  - permite evitar stores de resultados temporários (nós internos)

Ershov numbers

$t_1 = a - b$

$t_2 = c + d$

$t_3 = e * t_2$

$t_4 = t_1 + t_3$

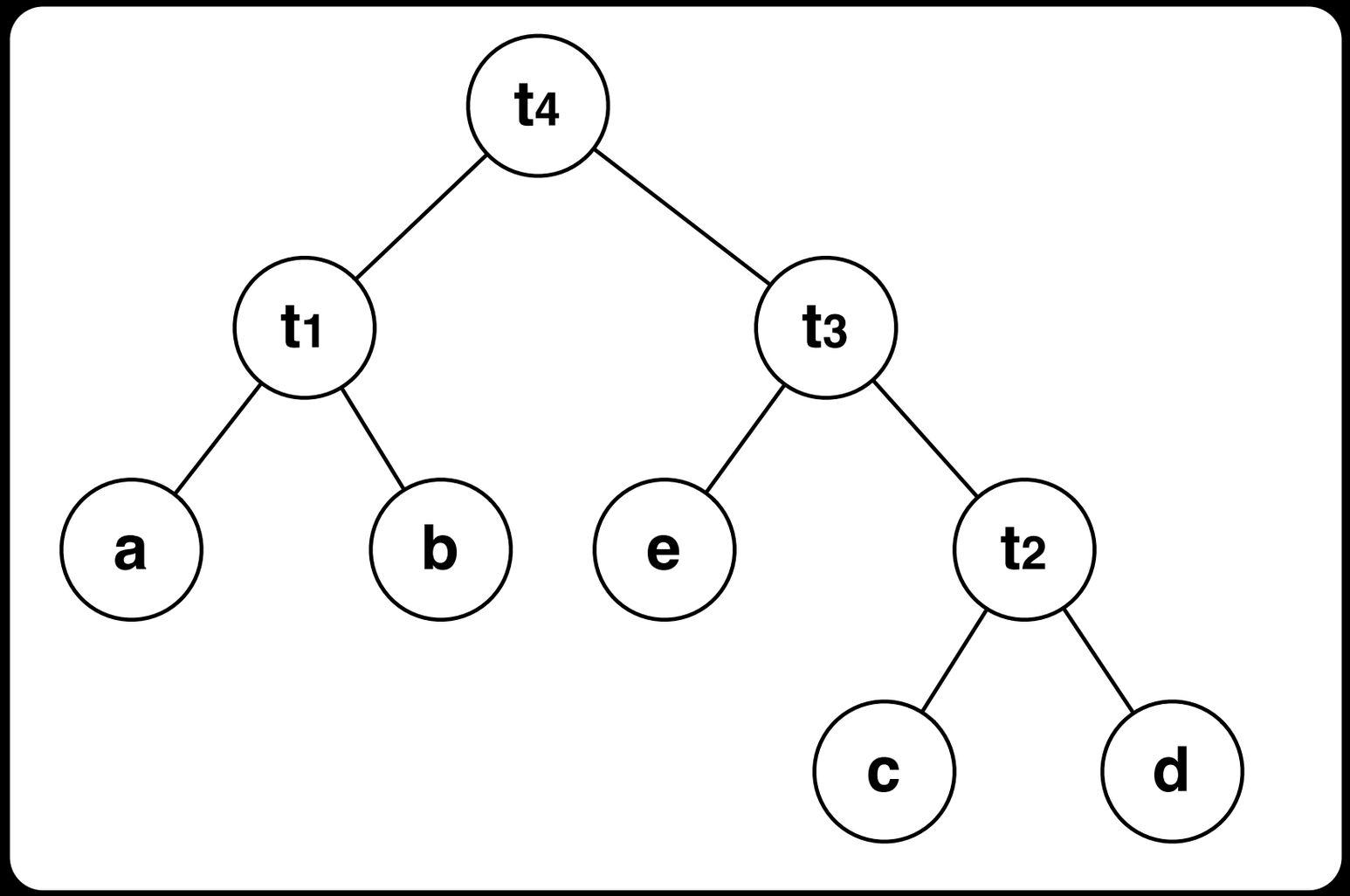
Ershov numbers

$t_1 = a - b$

$t_2 = c + d$

$t_3 = e * t_2$

$t_4 = t_1 + t_3$



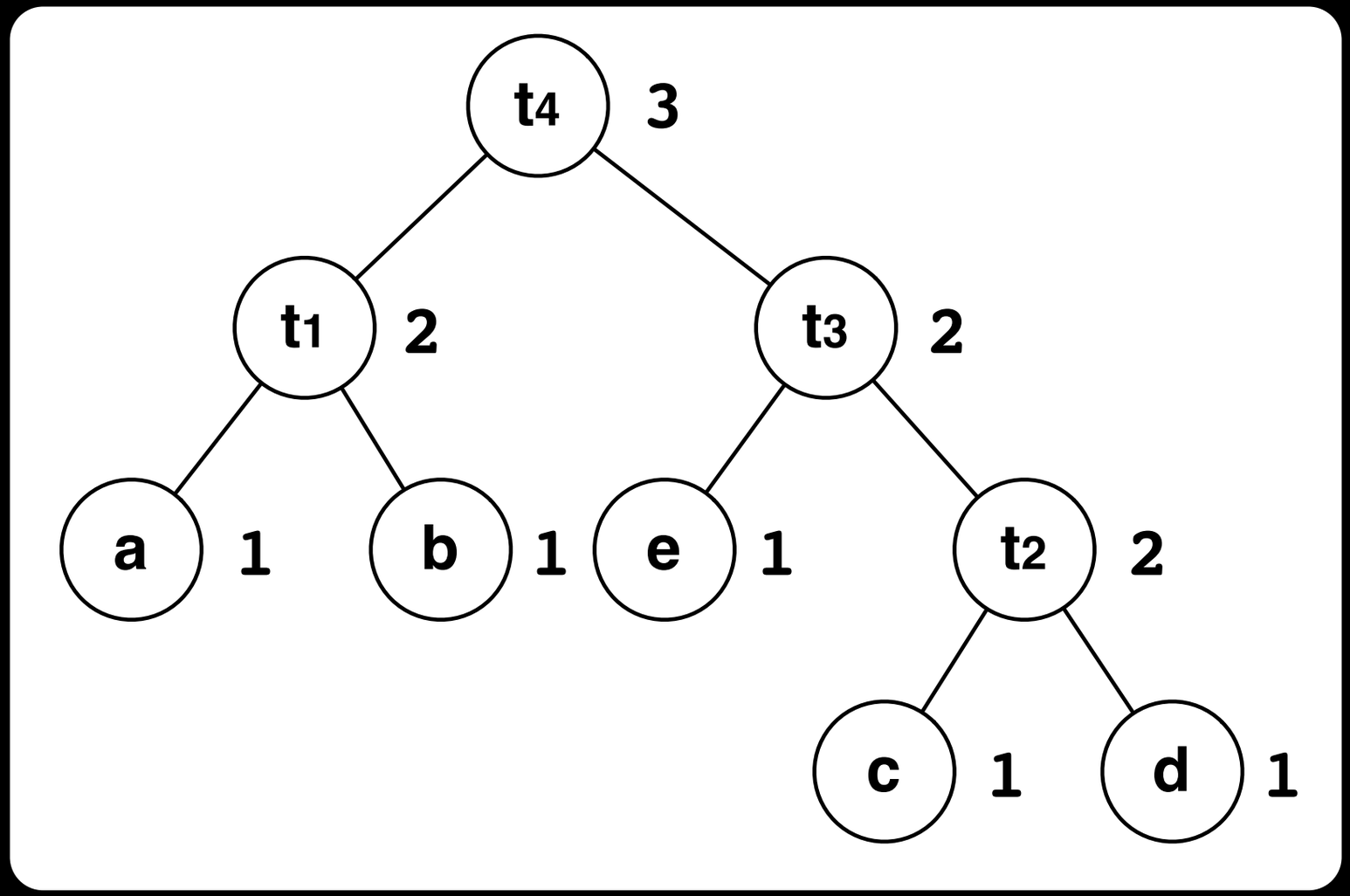
Ershov numbers

$t_1 = a - b$

$t_2 = c + d$

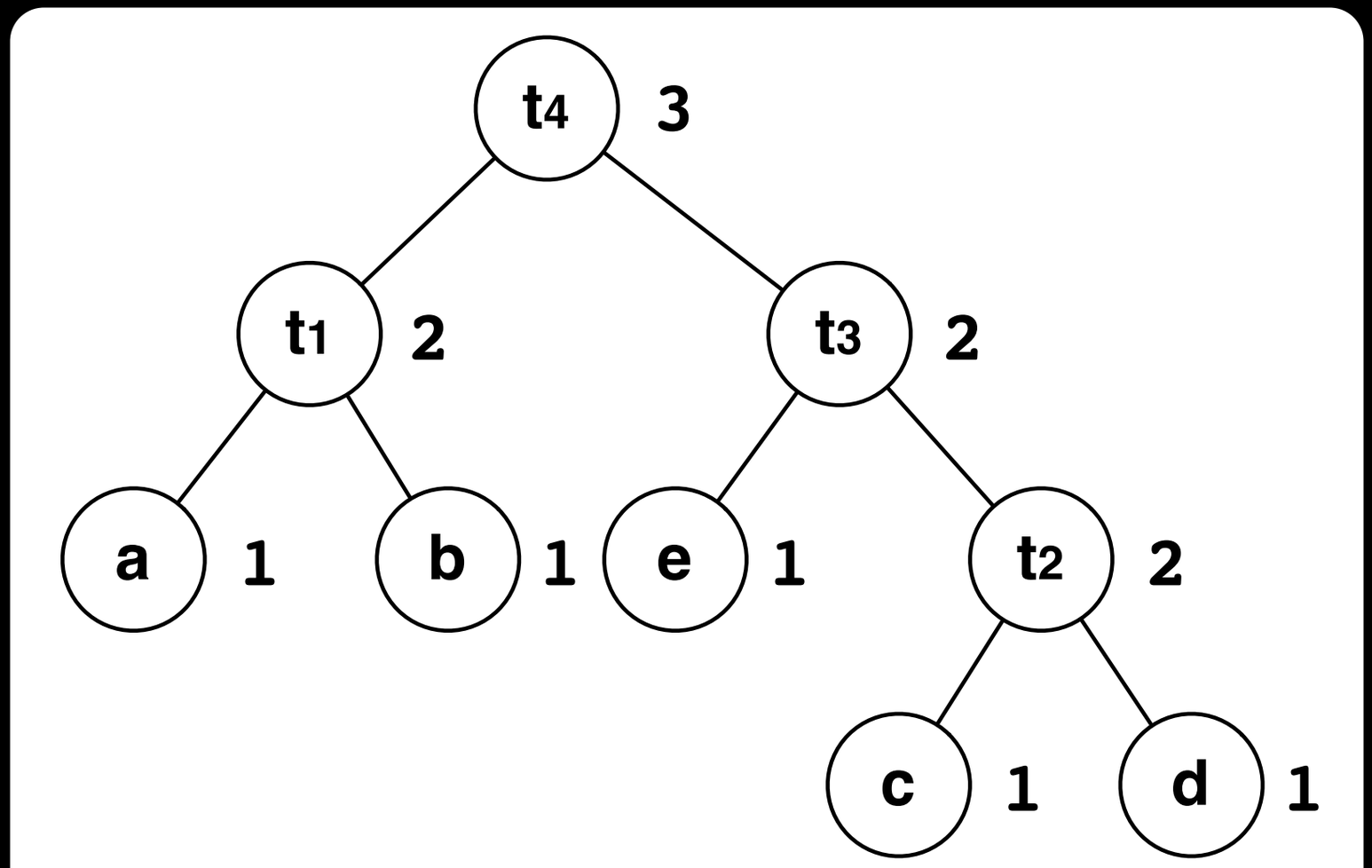
$t_3 = e * t_2$

$t_4 = t_1 + t_3$



Ershov numbers

```
LD   R3, d
LD   R2, c
ADD  R3, R2, R3
LD   R2, e
MUL  R3, R2, R3
LD   R2, b
LD   R1, a
SUB  R2, R1, R2
ADD  R3, R2, R3
```



Peephole optimizations

- Embora muitos compiladores produzam bom código de acordo com seleção cuidadosa de instruções, alguns usam estratégias alternativas
- Produzem *naive code*, e melhoram a qualidade introduzindo transformações de otimização ao programa gerado
- Novamente, otimização não significa código ótimo sob nenhuma medida matemática

Peephole optimizations

- Uma técnica simples e eficiente para melhorar localmente o código alvo é o exame de uma janela deslizante de instruções alvo (*peephole*)
- Busca oportunidades de substituir sequências de instruções dentro da janela, com uma sequência mais curta ou mais rápida
- Pode ser aplicado também à representação intermediária de código

Exemplos

- *Redundant-instruction elimination*
- *Flow-of-control optimizations*
- *Algebraic simplifications*
- *Use of machine idioms*

```
LD R0, a
ST a, R0
```

```
if debug == 1 goto L1
goto L2
L1: print debug info
L2: …
```

```
    if debug != 1 goto L2
        print debug info
L2: …
```

```
    if 0 != 1 goto L2
    print debug info
L2: …
```

```
    goto L2
    print debug info //unreachable
L2: …
```

```
goto L1
…
L1: goto L2
    …
L2: …
```

```
goto L2
…
L1: goto L2
    …
L2: …
```

```
if a<b goto L1
…
L1: goto L2
    …
L2: …
```

```
if a<b goto L2
…
L1: goto L2
    …
L2: …
```

```
goto L1
…
L1: if a<b goto L2
L3: …
```

```
    if a<b goto L2
    goto L3
        …
L3: …
```

Compiladores
(IF688)

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