Compiladores

JEAN LUCA BEZ

LUCAS MELLO SCHNORR

Otimização de Código I

INF01147

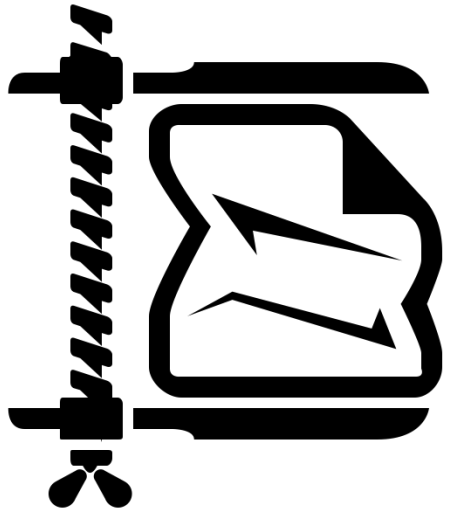
UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL

Introdução

JEAN LUCA BEZ
LUCAS MELLO SCHNORR

- Código gerado pelos algoritmos de compilação pode ser **transformado** para:
  - Executar mais rapidamente
  - Ocupar menos espaço

- Otimização do código ou melhoria do código
  - Permite eliminar instruções desnecessárias
  - Substituir sequências de instruções por outras mais rápidas



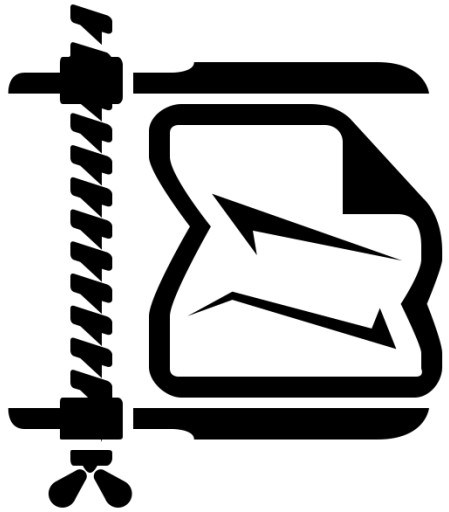
Quais são as melhores transformação (otimização) que podemos aplicar?

R:

JEAN LUCA BEZ
LUCAS MELLO SCHNORR

- Código gerado pelos algoritmos de compilação pode ser **transformado** para:
  - Executar mais rapidamente
  - Ocupar menos espaço

- Otimização do código ou melhoria do código
  - Permite eliminar instruções desnecessárias
  - Substituir sequências de instruções por outras mais rápidas

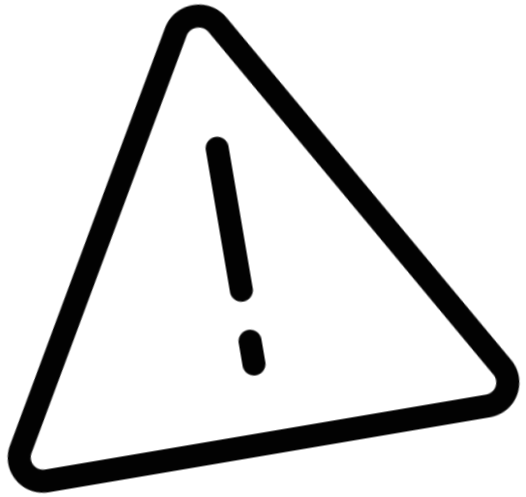


Quais são as melhores transformação (otimização) que podemos aplicar?

R: As que produzem o maior benefício com o menor esforço

Código “Otimizado”

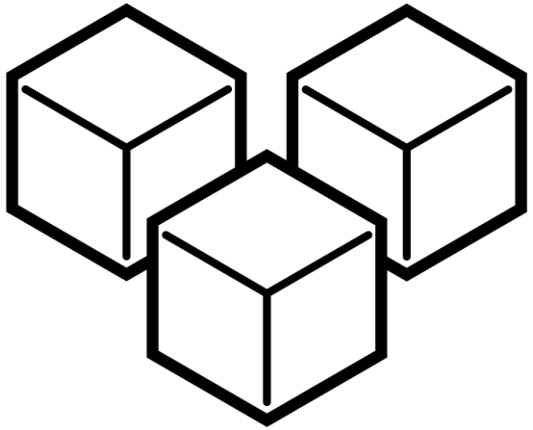
JEAN LUCA BEZ
LUCAS MELLO SCHNORR



- O termo **otimizador de código** ou **código otimizado** não é representativo
- Não há garantias matemáticas que o código será de fato ótimo
- Ao aplicar otimizações o código será **melhorado**
- Podemos melhorar significativamente:
  - Tempo de execução
  - Exigências de espaço
  - Tamanho do programa alvo

Propriedades

JEAN LUCA BEZ
LUCAS MELLO SCHNORR



1. Uma transformação deve **preservar** o significado dos programas
2. Uma transformação precisa **acelerar** os programas por um fator mensurável
3. Uma transformação precisa **valer** o esforço

Redundância

JEAN LUCA BEZ
LUCAS MELLO SCHNORR

- Existem muitas operações redundantes (repedidas) em um programa
- Redundância pode ter **duas causas**

1. Culpa do programador
   - Recalcular algum resultado
   - Deixar que o compilador identifique e otimize (removendo cálculos desnecessários)
2. Efeito colateral de linguagem de alto nível
   - Acessos do tipo `A[i][j]` ou `X->f1` (arranjos ou campos de estruturas)
   - Na compilação estes acessos se expandem em uma série de operações aritméticas de baixo nível
   - Acesso a uma mesma estrutura compartilha muitas operações
   - Fora do controle do programador

Organização para um Compilador Otimizante

JEAN LUCA BEZ
LUCAS MELLO SCHNORR

- Há **vários níveis** onde um programa pode ser melhorado

"A fase de melhoramento de código consiste nas análises de fluxo de controle e fluxo de dados, seguido pela aplicação de transformações"

Código Intermediário e Otimização

JEAN LUCA BEZ
LUCAS MELLO SCHNORR

- Operações necessárias para implementar as construções de alto nível ficam **explícitas** no código intermediário
- É possível então aplicar otimizações
- Código intermediário pode ser relativamente independente de máquina
- Otimizador não tem que mudar muito se o gerador de código for substituído por outro de uma máquina diferente

Quick Sort

JEAN LUCA BEZ
LUCAS MELLO SCHNORR

QUICKSORT

```
void quicksort (int m, int n)
{
   int i, j;
   int v, x;
   if (n <= m) return;
   /* o fragmento comeca aqui */
   i = m-1; j = n; v = a[n];
   while (1) {
      do i = j + 1; while (a[i] < v);
      do j = j - 1; while (a[j] > v);
      x = a[i]; a[i] = a[j]; a[j] = x; /* troca a[i], a[j] */
   }
   x = a[i]; a[i] = a[n]; a[n] = x; /* troca a[i], a[n] */
   /* fragmento termina aqui */
   quicksort(m, j); quicksort(i+1, n);
}
```

CÓDIGO DE 3 ENDEREÇOS

```
// int(4 bytes)

// x = a[i]
t6 = 4 * i
x = a[t6]

// a[j] = x
t10 = 4 * j
a[t10] = x
```

Código de 3 Endereços

JEAN LUCA BEZ
LUCAS MELLO SCHNORR

CÓDIGO DE 3 ENDEREÇOS PARA QUICKSORT

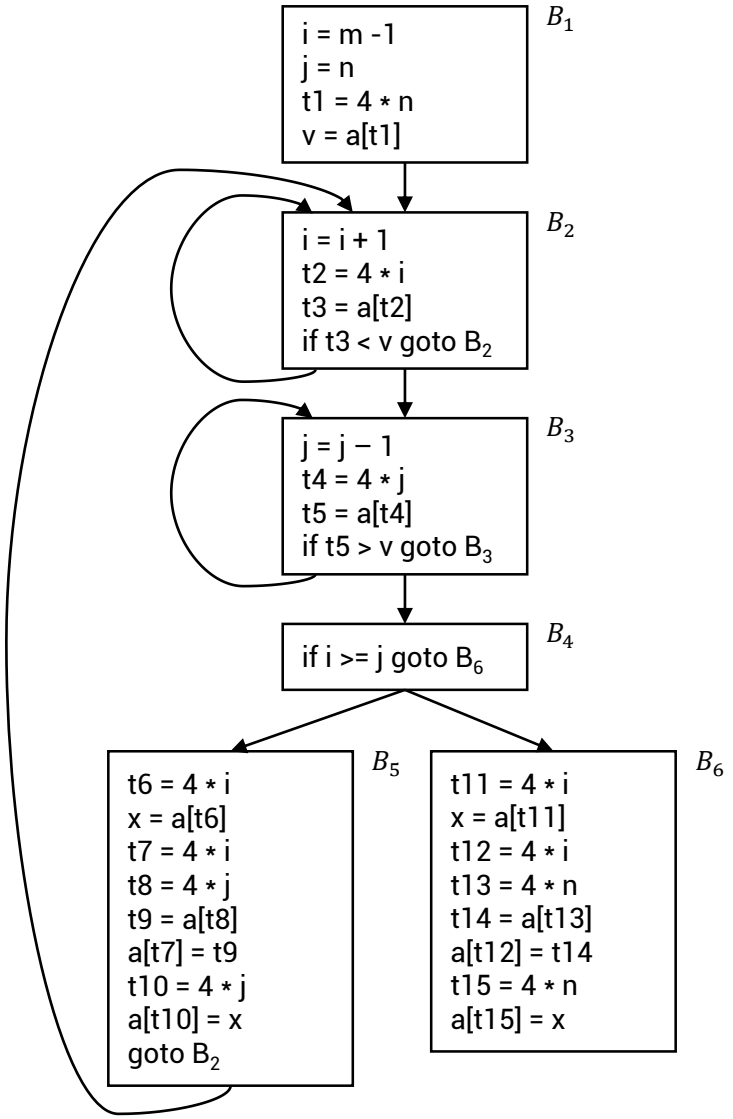
```
 (1) i = m - 1
 (2) j = n
 (3) t1 = 4 * n
 (4) v = a[t1]
 (5) i = i + 1
 (6) t2 = 4 * i
 (7) t3 = a[t2]
 (8) if t3 < v goto (5)
 (9) j = j - 1
(10) t4 = 4 * j
(11) t5 = a[t4]
(12) if t5 > v goto (9)
(13) if i >= j goto (23)
(14) t6 = 4 * i
(15) x = a[t6]
```

CÓDIGO DE 3 ENDEREÇOS PARA QUICKSORT (CONTINUAÇÃO)

```
(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
```

Grafo de Fluxo

JEAN LUCA BEZ
LUCAS MELLO SCHNORR



LOCAL
ESCOPO DOS BLOCOS

GLOBAL
UTILIZADO POR SEGUNDO

Otimizações Locais

JEAN LUCA BEZ
LUCAS MELLO SCHNORR

- É a forma mais simples de otimização
- Objetivo é otimizar **um único** bloco básico
  - Não é necessário analisar todo o corpo do procedimento
- Algumas instruções podem ser **removidas**
- Algumas instruções podem ser **simplificadas**

INSTRUÇÕES PARA REMOVER

```
x := x + 0
x := x * 1
```

INSTRUÇÕES PARA SIMPLIFICAR

```
x + 0          0 + x          => x
1 * x          x * 1          => x
2 * x          x * 2          => x + x
x ** x         pow(x,2)       => x * x
a * (1/b)      (1/b) * a      => a / b
x > y                         => x - y > 0
```

Otimizações Locais

JEAN LUCA BEZ
LUCAS MELLO SCHNORR

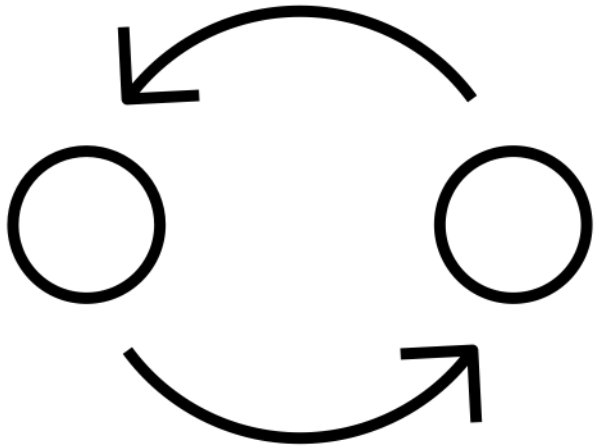
- Operações em **constantes** podem ser computadas durante a compilação
- Exemplo de uma instrução `x := y op z`
  - Onde `y` e `z` são constantes
  - Podemos computar o valor de `x` durante a compilação
- Conhecido por **constant folding**

INSTRUÇÕES PARA COMPUTAR

```
x := 2 + 2          => x := 4
if 2 < 0 jump L     => if false jump L
```

Transformações com Preservação de Semântica

JEAN LUCA BEZ
LUCAS MELLO SCHNORR



- Melhorar o programa sem mudar a forma como ele calcula

- **Eliminação** de subexpressões comuns
- **Propagação** de cópias
- **Eliminação** de código morto
- **Transposição** para constantes

Eliminação de Subexpressões Comuns Locais

JEAN LUCA BEZ
LUCAS MELLO SCHNORR

- Frequentemente programa incluirá cálculos duplicados
- Abaixo do nível de detalhe do programador

EXERCÍCIO 1

Como podemos otimizar o bloco 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
```

B_5

B_5

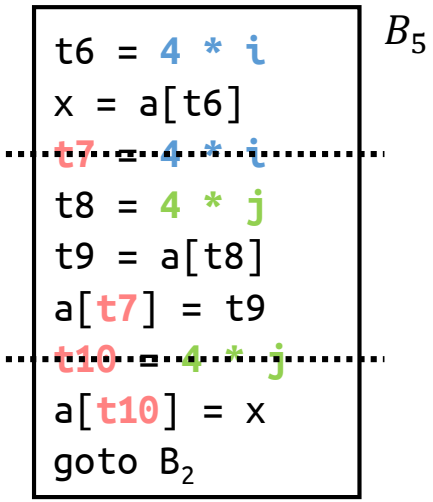
Eliminação de Subexpressões Comuns Locais

JEAN LUCA BEZ
LUCAS MELLO SCHNORR

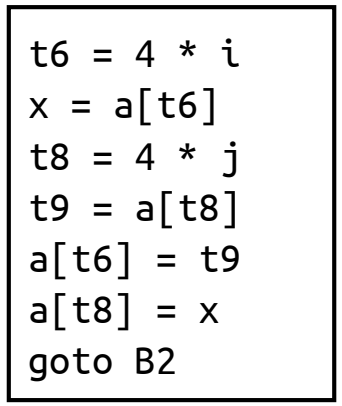
- Frequentemente programa incluirá cálculos duplicados
- Abaixo do nível de detalhe do programador

EXERCÍCIO 1

Como podemos otimizar o bloco B_5?



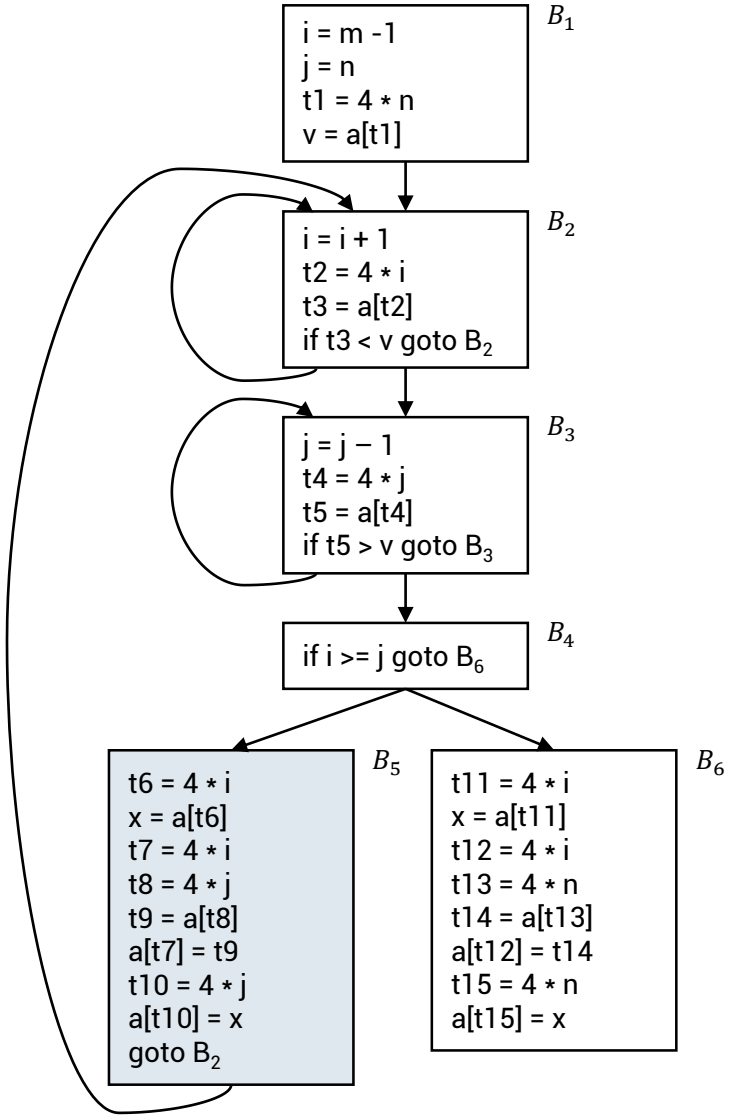
B_5

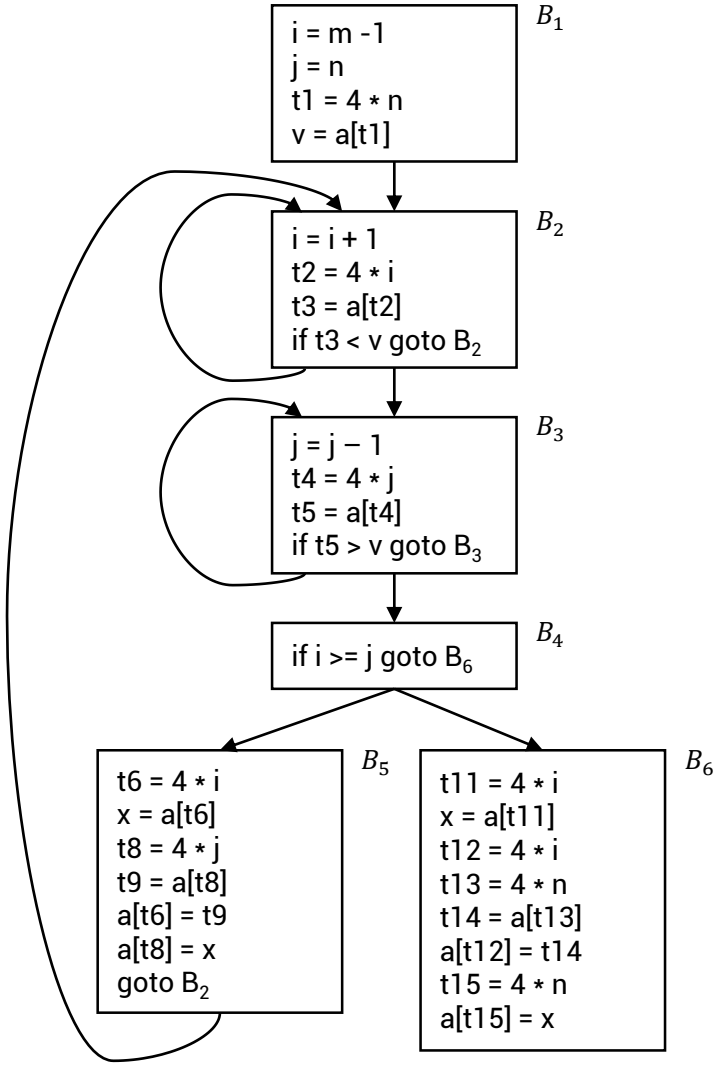


B_5

Eliminação de Subexpressões Comuns Locais

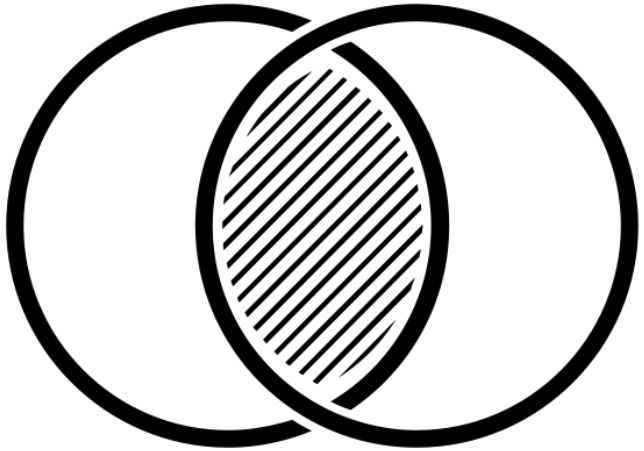
JEAN LUCA BEZ
LUCAS MELLO SCHNORR





Eliminação de Subexpressões Comuns Globais

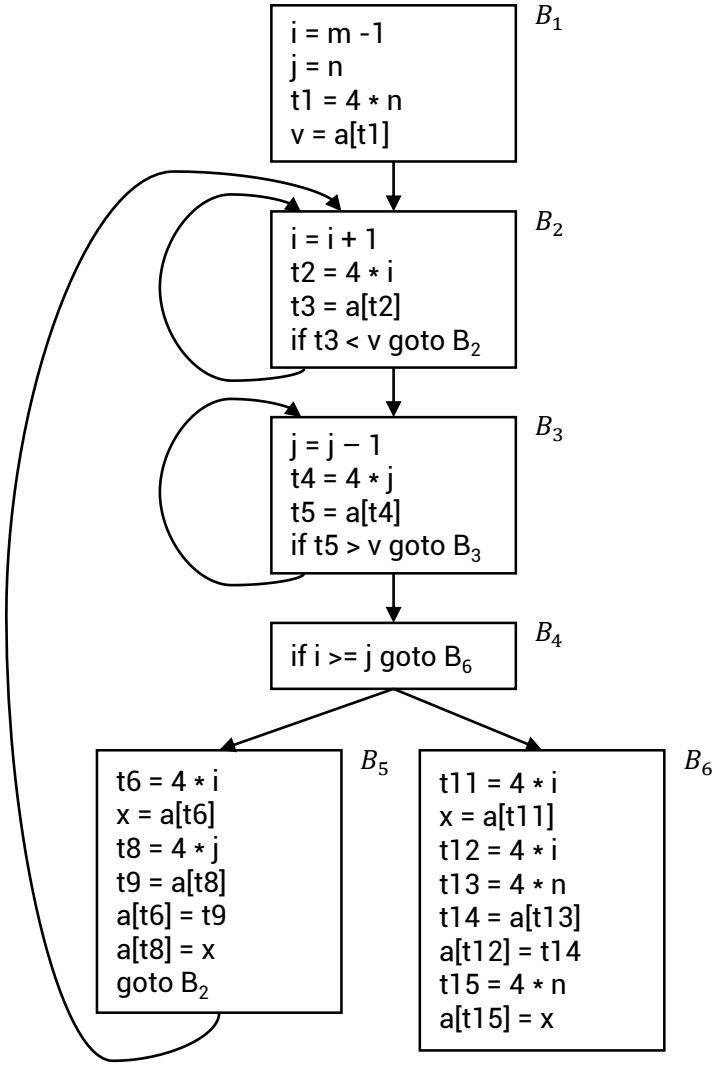
JEAN LUCA BEZ
LUCAS MELLO SCHNORR

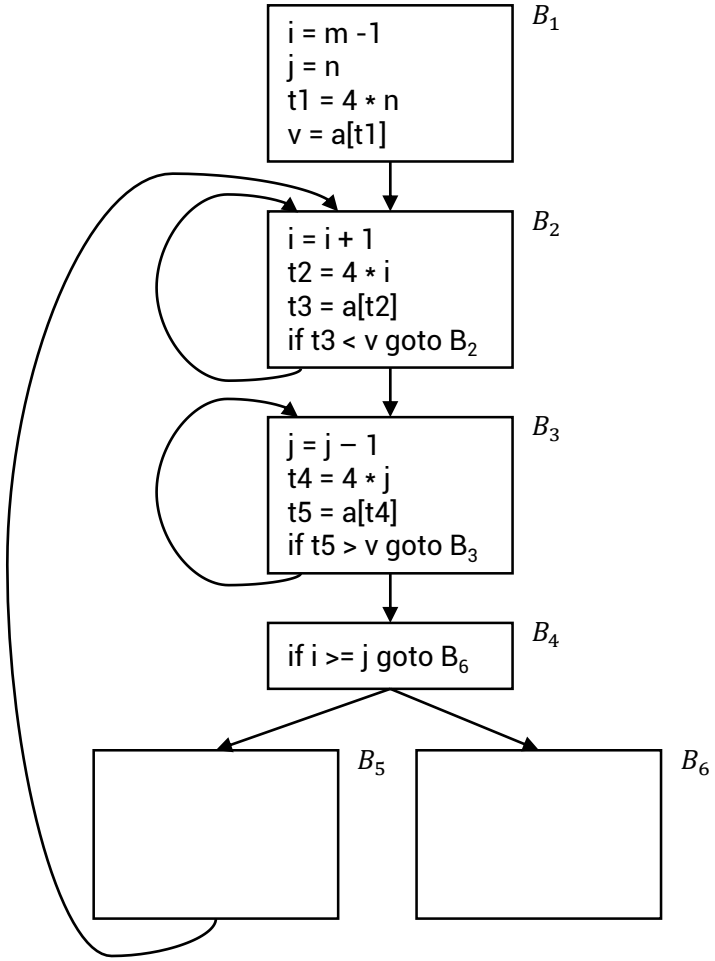


A ocorrência de uma expressão E é chamada de uma **subexpressão comum** se E foi computada previamente e os valores das variáveis em E não mudaram desde a computação anterior.

Eliminação de Subexpressões Comuns Globais

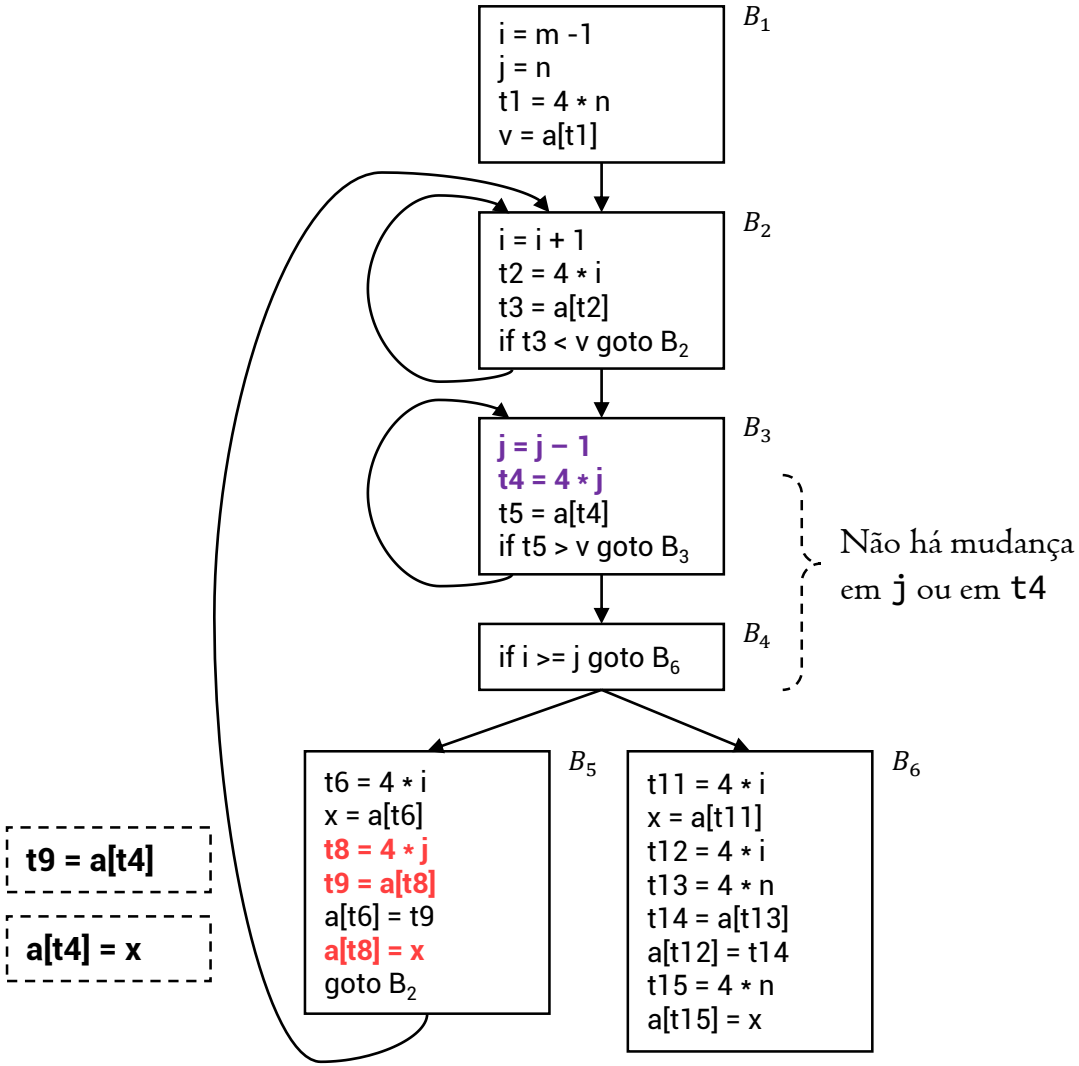
JEAN LUCA BEZ
LUCAS MELLO SCHNORR

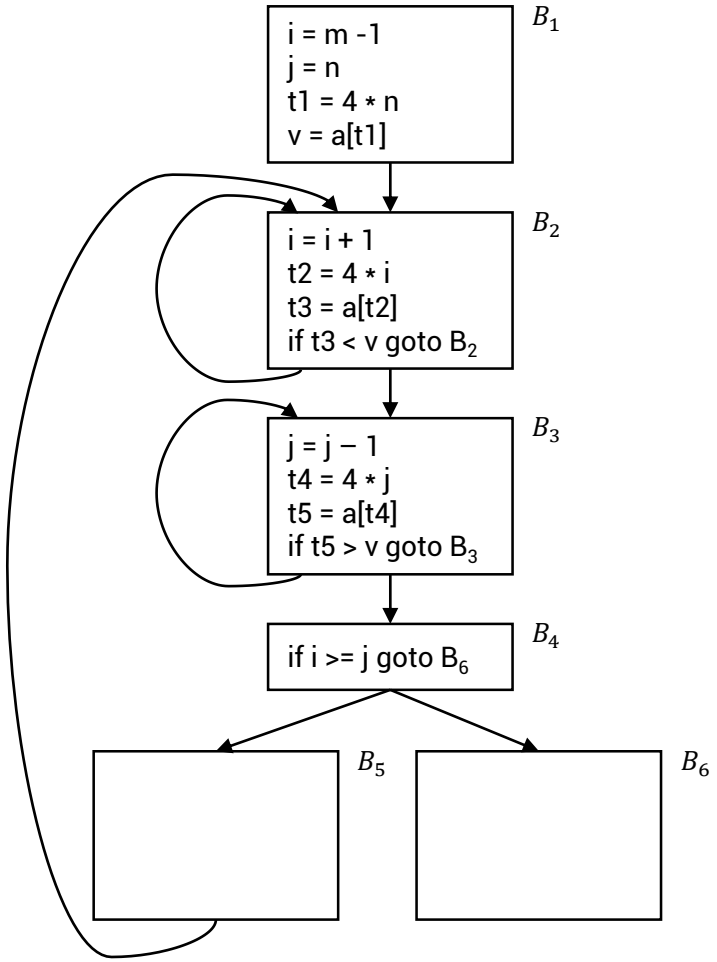




Eliminação de Subexpressões Comuns Globais

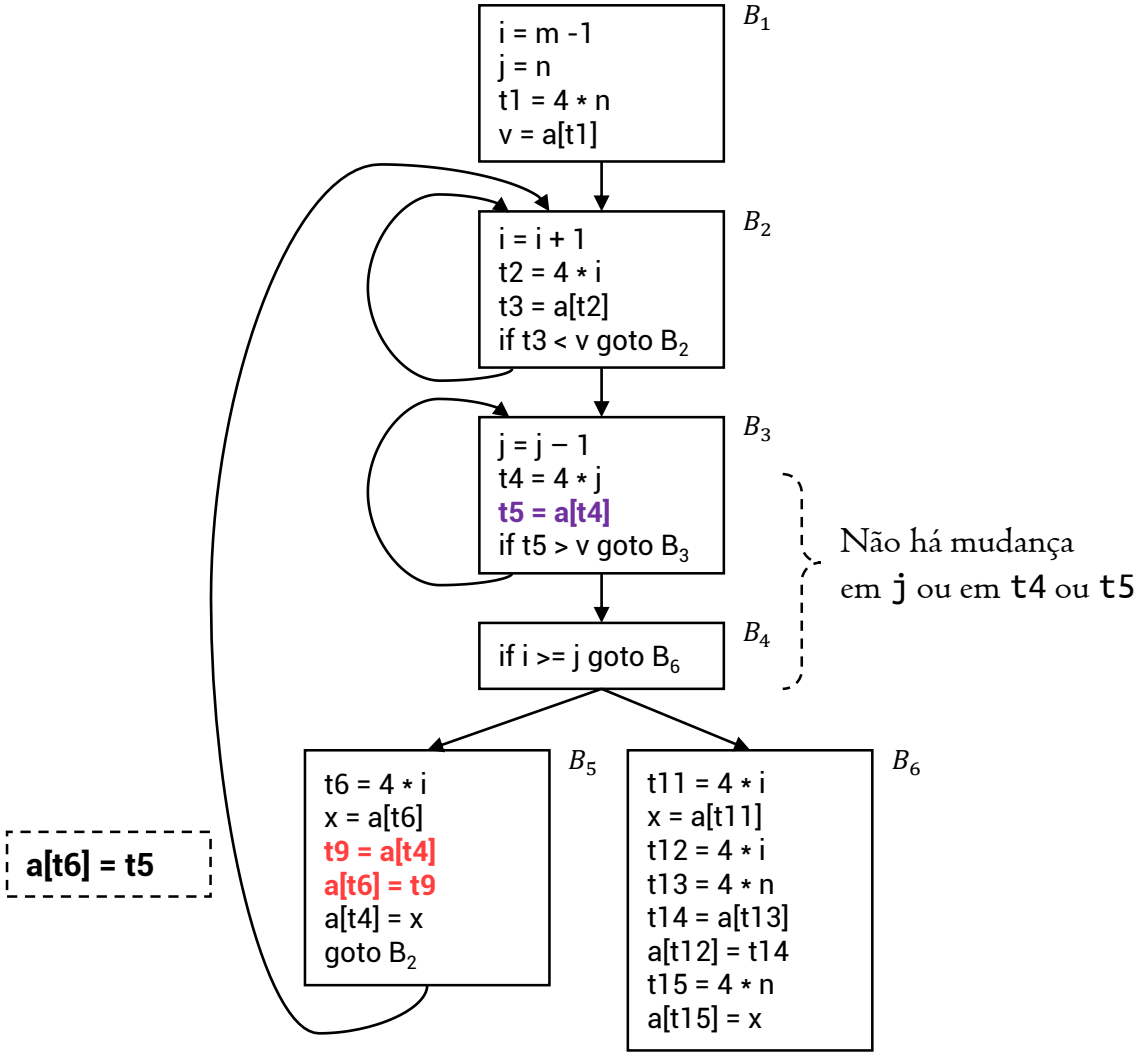
JEAN LUCA BEZ
LUCAS MELLO SCHNORR

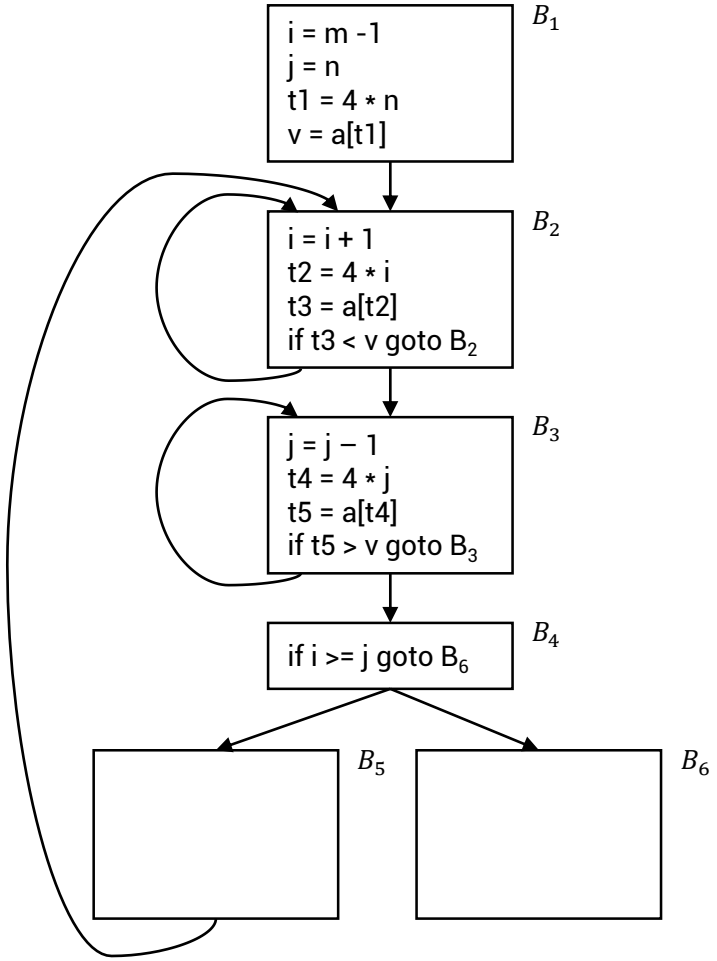




Eliminação de Subexpressões Comuns Globais

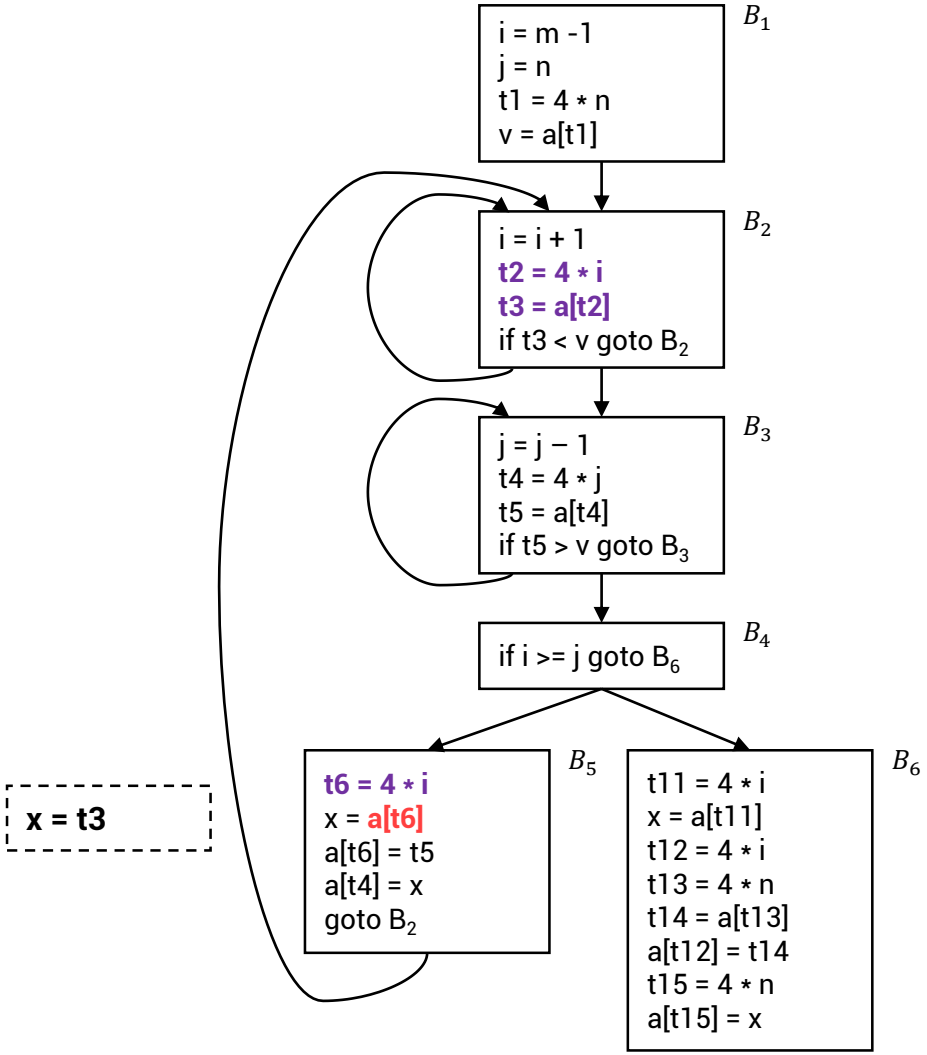
JEAN LUCA BEZ
LUCAS MELLO SCHNORR

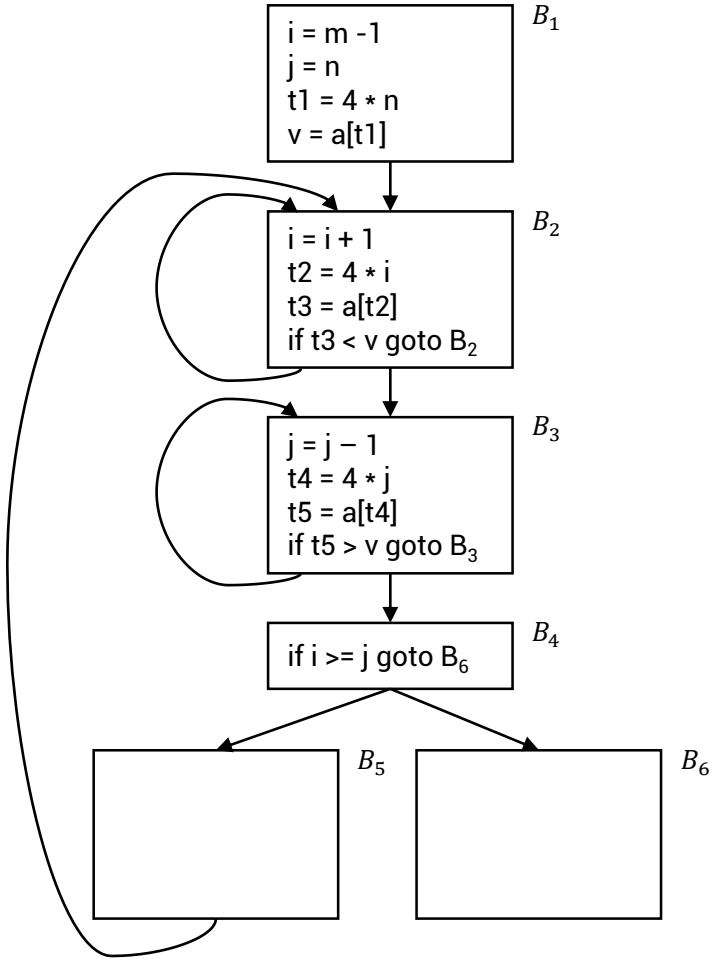




Eliminação de Subexpressões Comuns Globais

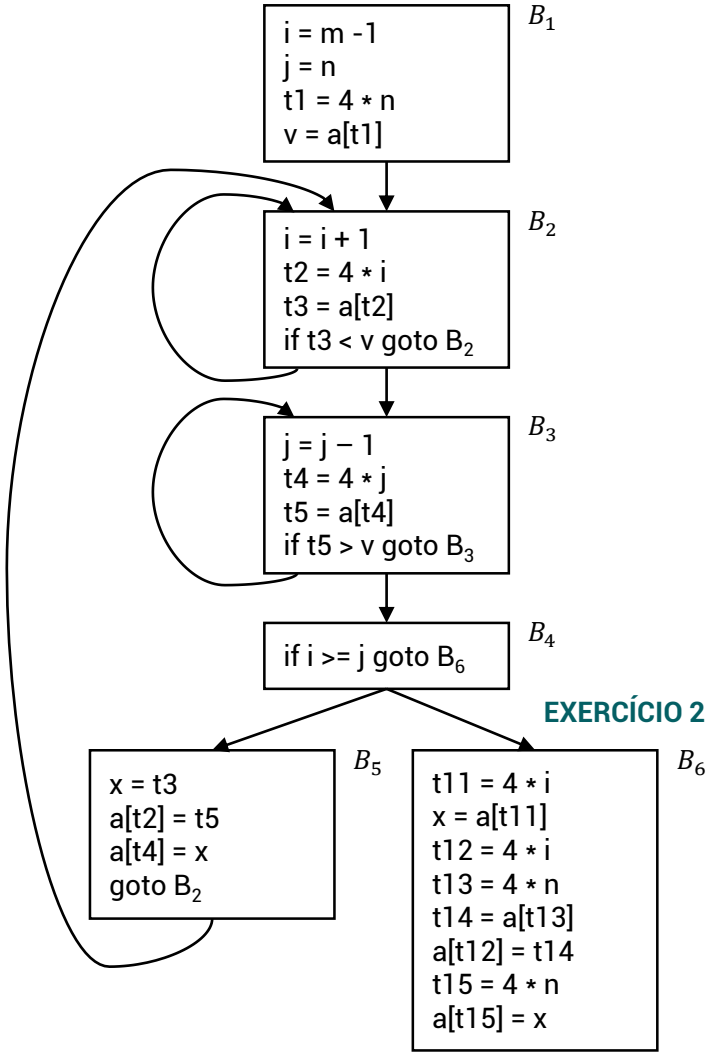
JEAN LUCA BEZ
LUCAS MELLO SCHNORR



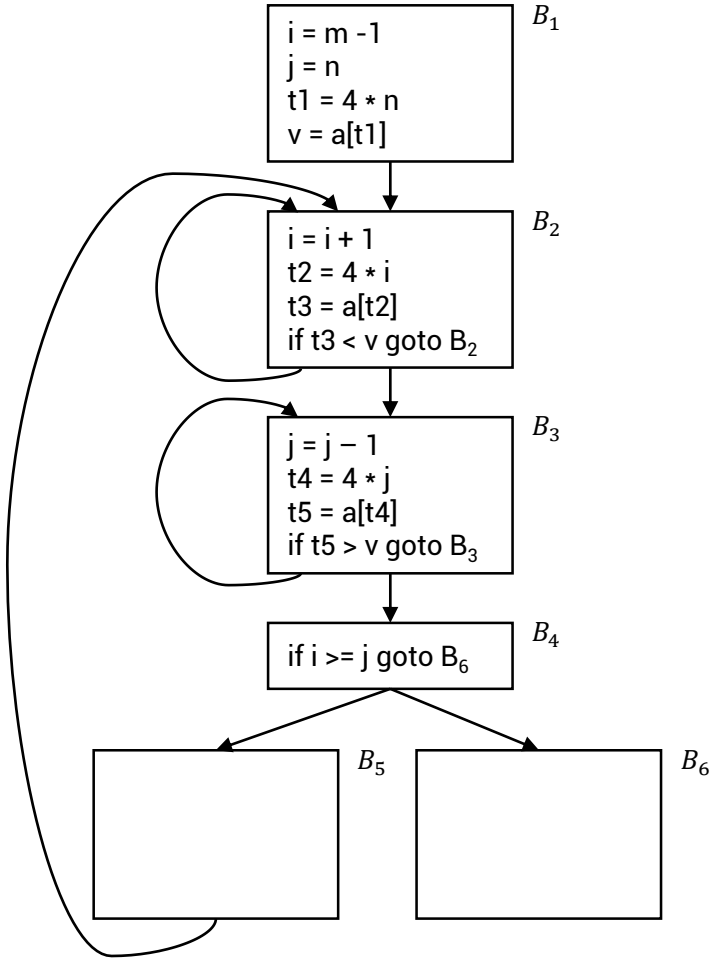


Eliminação de Subexpressões Comuns Globais

JEAN LUCA BEZ
LUCAS MELLO SCHNORR

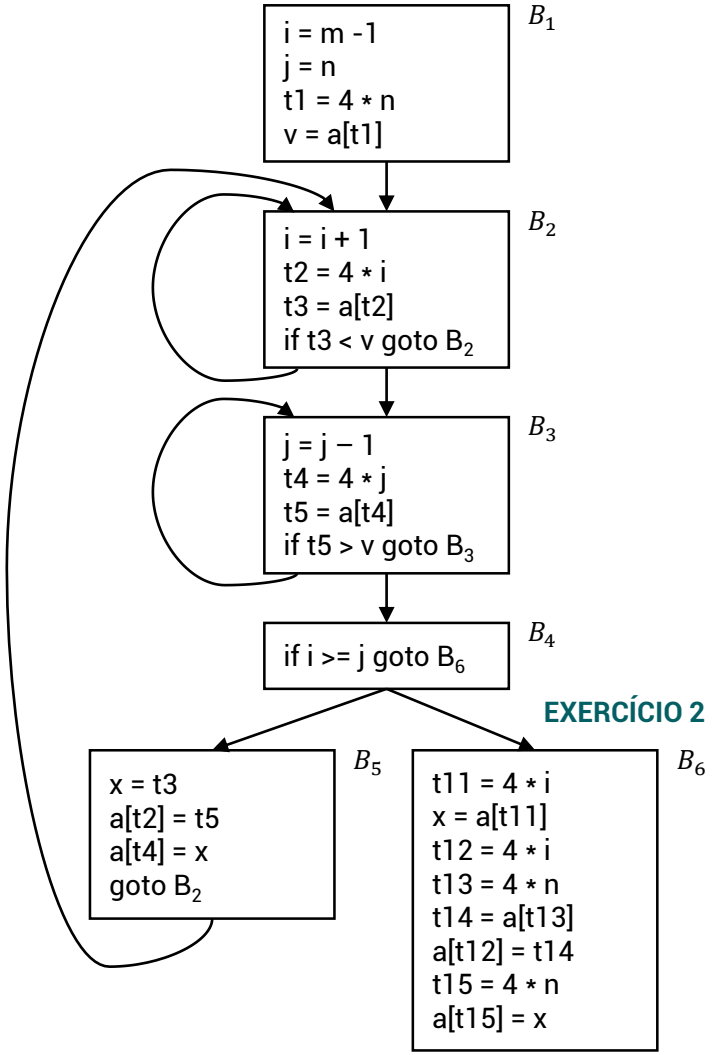


EXERCÍCIO 2

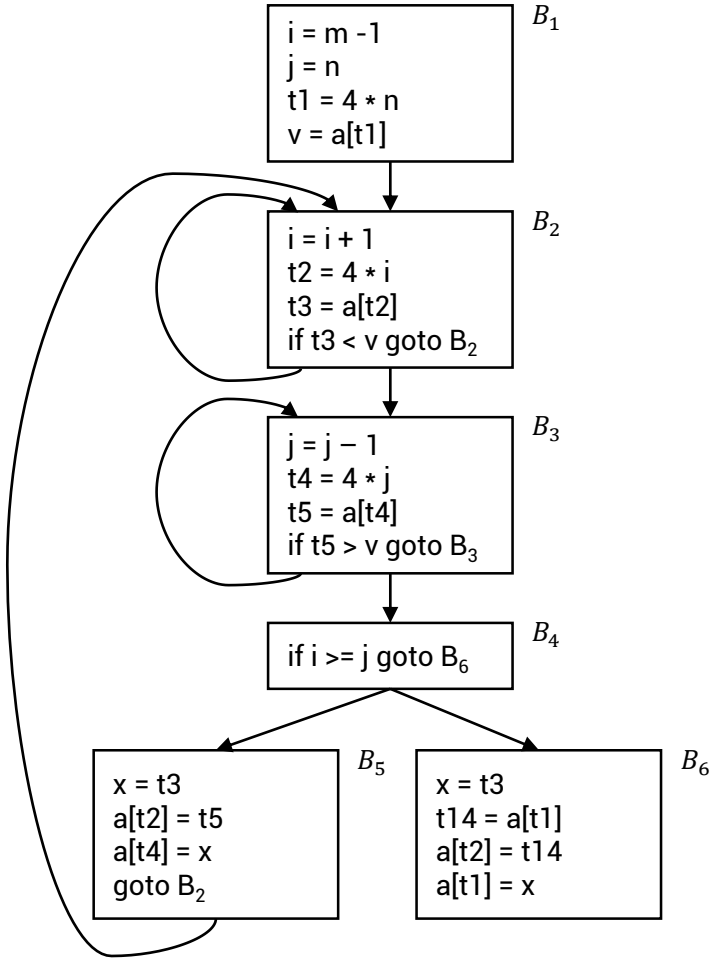


Eliminação de Subexpressões Comuns Globais

JEAN LUCA BEZ
LUCAS MELLO SCHNORR

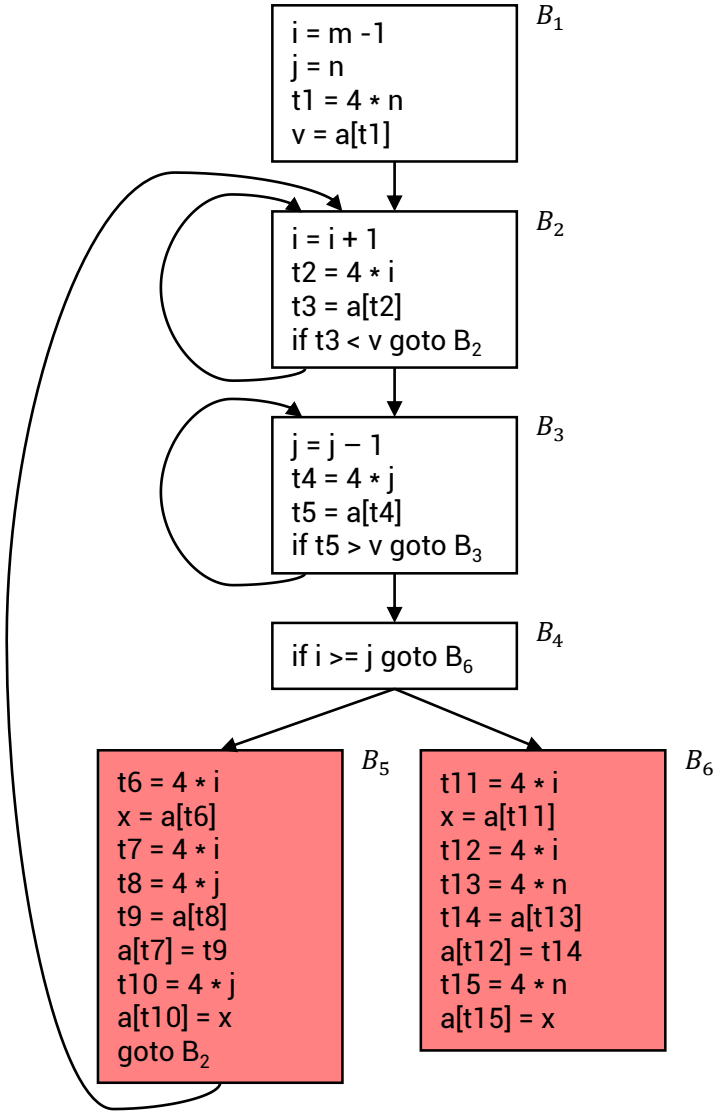


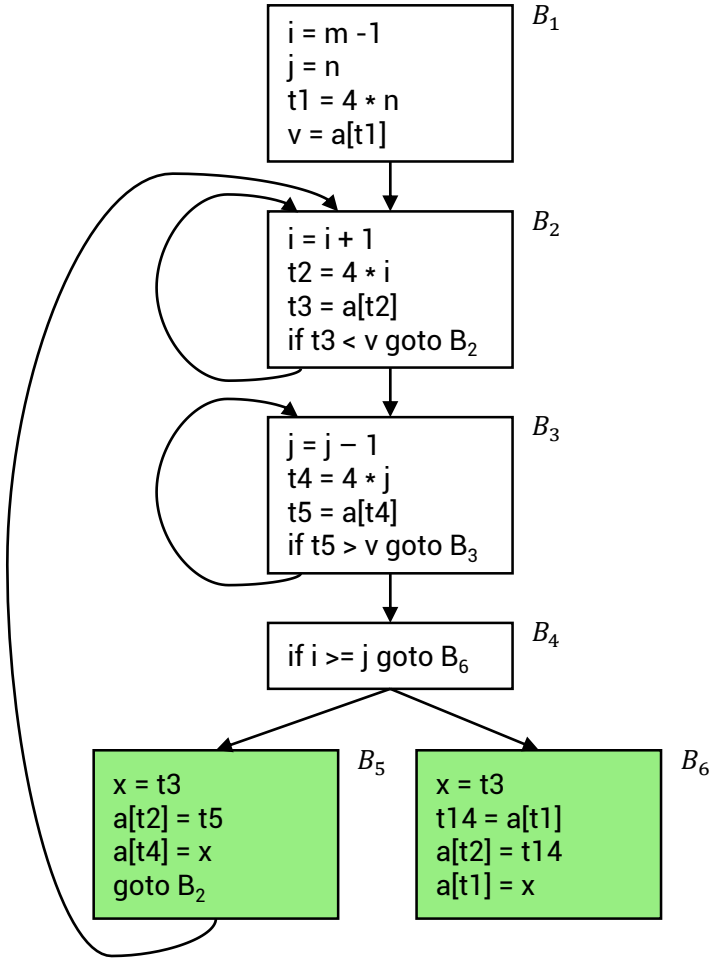
EXERCÍCIO 2



Eliminação de Subexpressões Comuns Globais

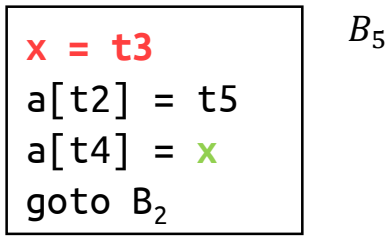
JEAN LUCA BEZ
LUCAS MELLO SCHNORR



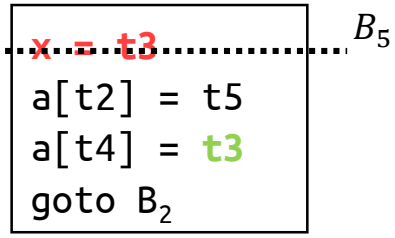


Propagação de Cópia

JEAN LUCA BEZ
LUCAS MELLO SCHNORR



B_5



B_5

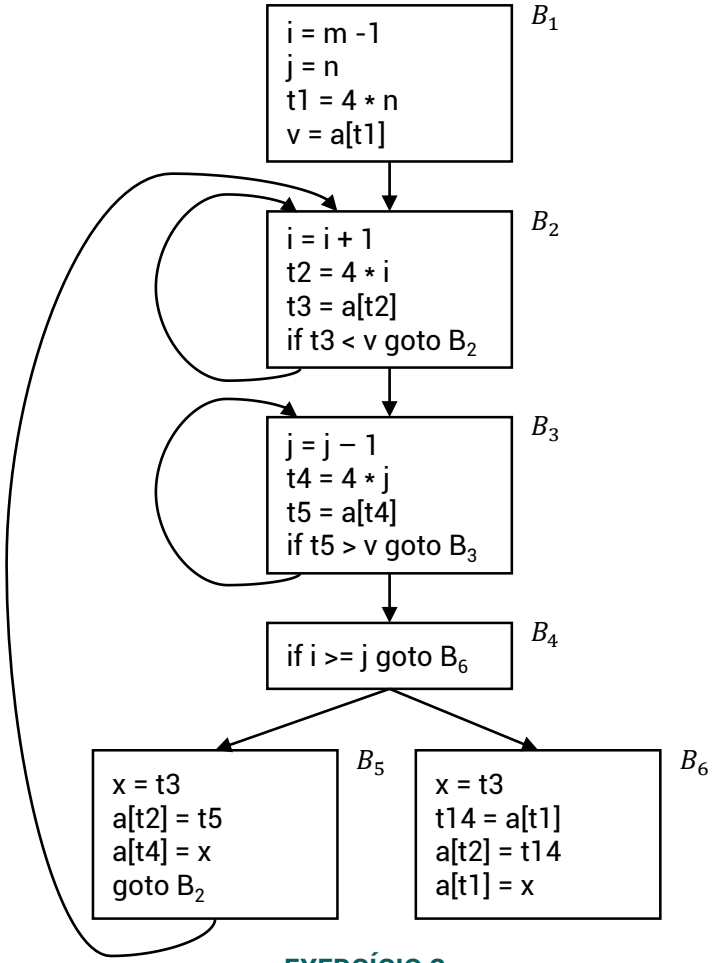
- Otimizar B_5 **eliminando** x
- Instruções de cópia (u = v)
- Introduzidas pelo algoritmo
  - Eliminar subexpressões comuns
  - Outras otimizações

Ideia:

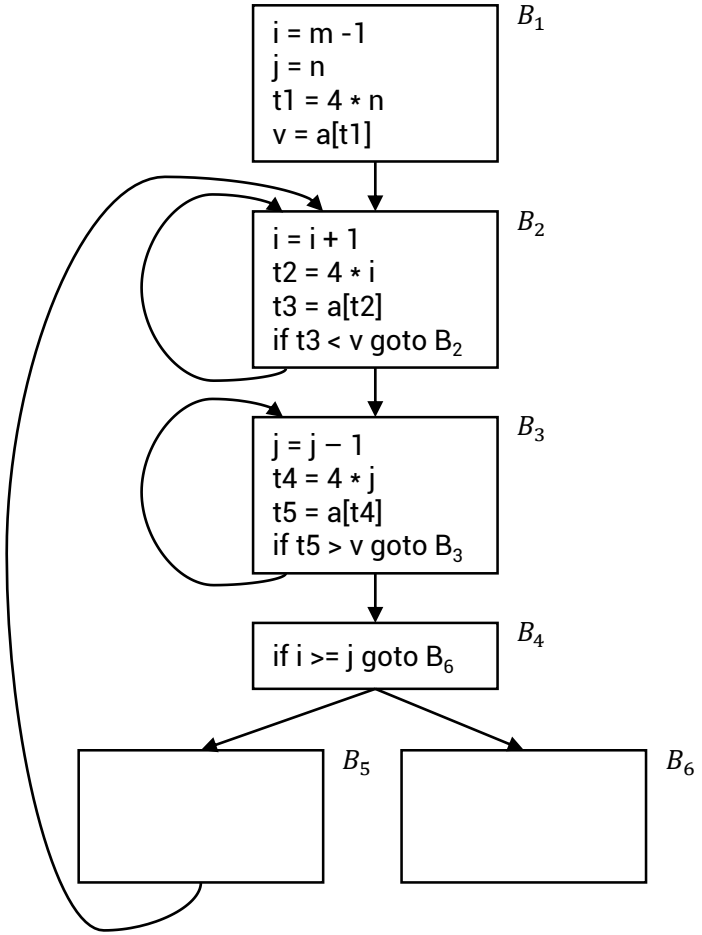
Utilizar, se possível, v por u após o comando de cópia u = v

Propagação de Cópia

JEAN LUCA BEZ
LUCAS MELLO SCHNORR

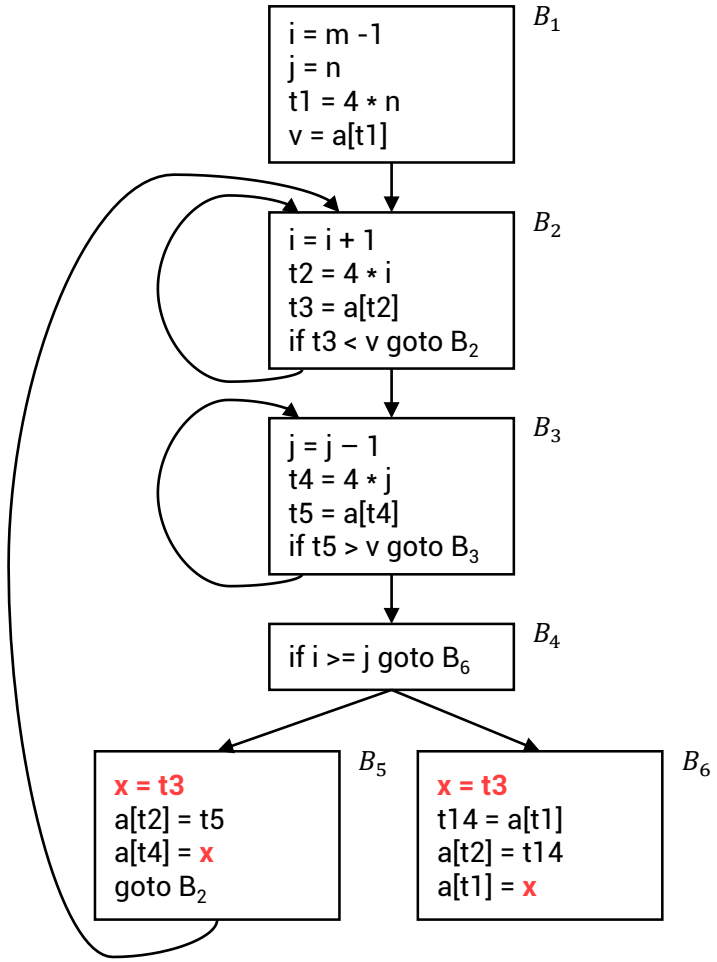


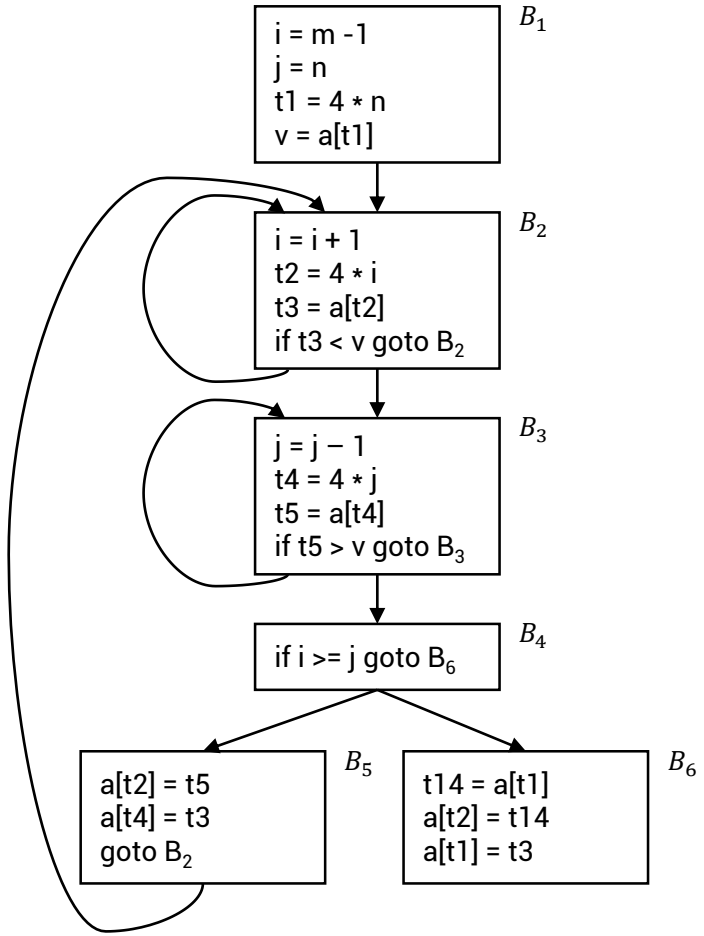
EXERCÍCIO 3



Propagação de Cópia

JEAN LUCA BEZ
LUCAS MELLO SCHNORR





Exercício

JEAN LUCA BEZ
LUCAS MELLO SCHNORR

EXERCÍCIO 4

Como podemos otimizar o seguinte trecho de código utilizando as técnicas vistas até o momento?

CÓDIGO PARA OTIMIZAR

```
b = a * a;
c = a * a;
d = b + c;
e = b + b;
```

Exercício

JEAN LUCA BEZ
LUCAS MELLO SCHNORR

EXERCÍCIO 4

Como podemos otimizar o seguinte trecho de código utilizando as técnicas vistas até o momento?

CÓDIGO PARA OTIMIZAR

```
b = a * a;
c = a * a;
d = b + c;
e = b + b;
```

RESPOSTA

```
b = a * a;
c = b;
d = b + b;
e = d;
```

Eliminação de Código Morto

JEAN LUCA BEZ
LUCAS MELLO SCHNORR

"Variável está **viva** em um ponto se seu valor puder ser usado posteriormente"

- Programador não introduz (intencionalmente) código morto
- Resultado de transformações
  - Outras otimizações

Eliminação de Código Morto

JEAN LUCA BEZ
LUCAS MELLO SCHNORR

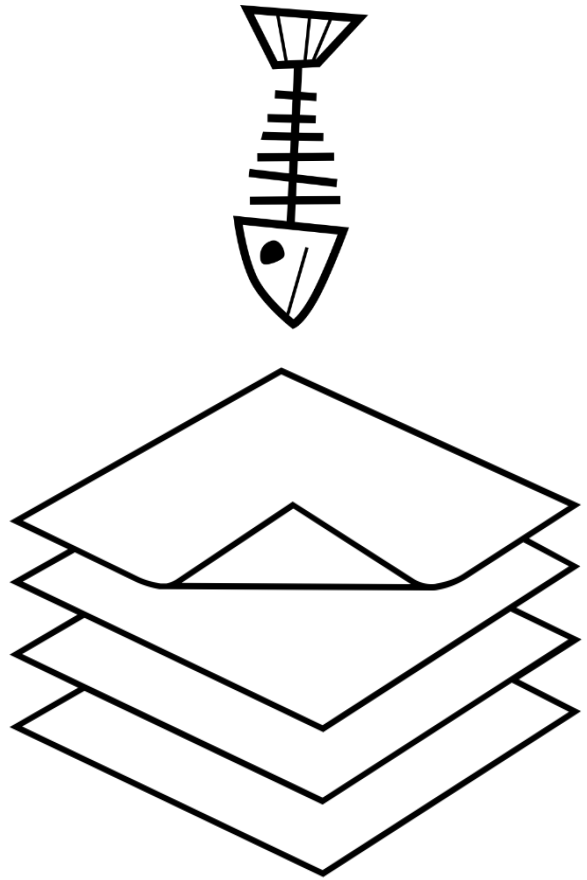
EXEMPLO DE CÓDIGO

```
debug = FALSE

....

if (debug) print ...
```

- Compilador pode deduzir que o valor é sempre **FALSE**
- Utilizando propagação de cópias
- Todo conteúdo do **if** nunca será executado (código morto)
- Possível eliminar o teste e o comando



Exercício

JEAN LUCA BEZ
LUCAS MELLO SCHNORR

EXERCÍCIO 5

Como podemos otimizar o seguinte trecho de código utilizando as técnicas vistas até o momento? Assuma que somente **g** e **x** são referenciadas fora deste bloco básico.

CÓDIGO PARA OTIMIZAR

```
1 a := 1
2 b := 3
3 c := a + x
4 d := a * 3
5 e := b * 3
6 f := a + b
7 g := e - f
```

Exercício

JEAN LUCA BEZ
LUCAS MELLO SCHNORR

EXERCÍCIO 5

Como podemos otimizar o seguinte trecho de código utilizando as técnicas vistas até o momento? Assuma que somente **g** e **x** são referenciadas fora deste bloco básico.

CÓDIGO PARA OTIMIZAR

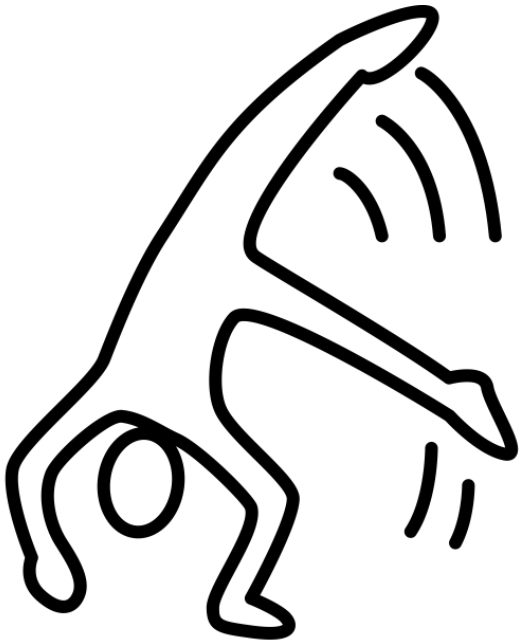
```
1 a := 1
2 b := 3
3 c := a + x
4 d := a * 3
5 e := b * 3
6 f := a + b
7 g := e - f
```

RESPOSTA

```
1 g := 5
```

Movimentação de Código

JEAN LUCA BEZ
LUCAS MELLO SCHNORR

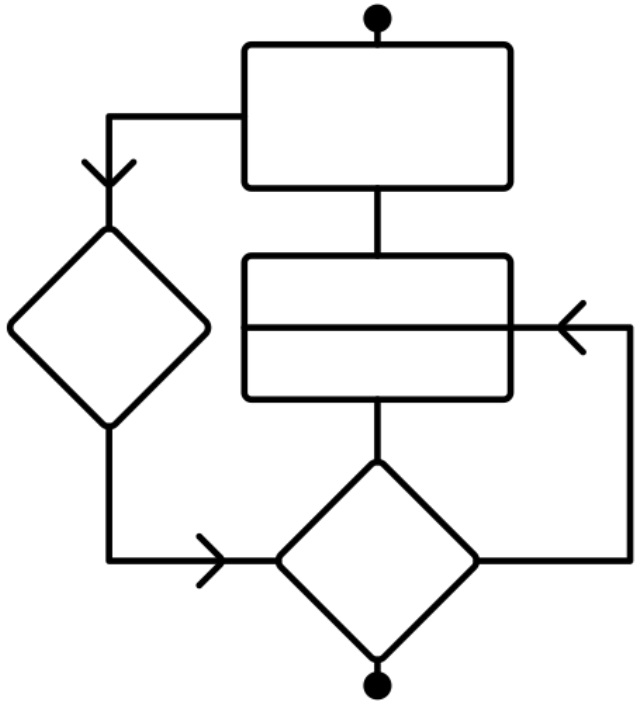


- Loops são **ótimos** candidatos a otimização
- Principalmente loops **internos**
- Podemos diminuir tempo de execução:
  - Diminuirmos o número de instruções em um loop interno
  - Mesmo se aumentar a quantidade de instruções fora deste loop

- Pega expressão que gera o mesmo resultado
- Independente do número de vezes que o loop é executado
- Avalia **antes** do loop (a.k.a. Cálculo do Invariante do Loop)

Movimentação de Código

JEAN LUCA BEZ
LUCAS MELLO SCHNORR



- $limit - 2$ é **invariante** ao loop
- Cálculo é feito $n + 1$ vezes

```
while (i <= limit-2)
```

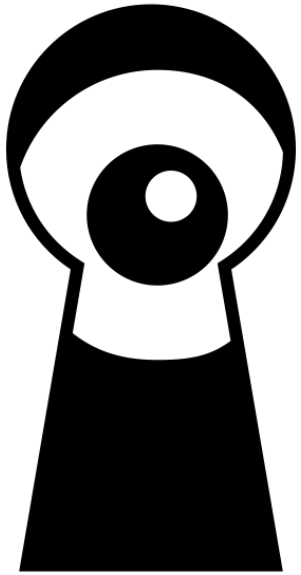
- Cálculo é feito uma única vez

```
t = limit-2
while (i <= t)
```

- Assumindo que $limit$ não seja alterado dentro do loop

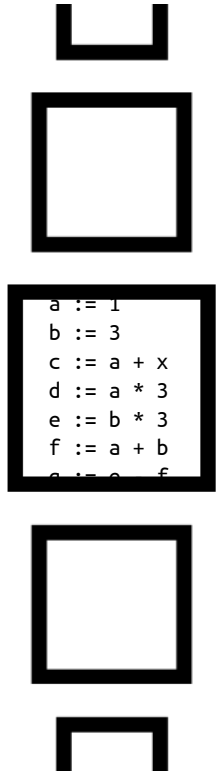
Peephole

JEAN LUCA BEZ
LUCAS MELLO SCHNORR



- Otimização **peephole** é uma variação da otimização local
- Aplicada diretamente em código assembly
- Considera uma **janela deslizante** no código
- Sequência de instruções (geralmente contíguas)
- Otimizador substitui a sequência por outra equivalente (mais rápida)
- Característica:
  - Cada melhoria possa gerar oportunidade para melhorias adicionais

JEAN LUCA BEZ
LUCAS MELLO SCHNORR



- Escritas como **regras** de substituição no formato:

$$i_1 , \dots , i_n \rightarrow j_1 , \dots , j_m$$

- Onde o lado direito representa a versão otimizada do lado esquerdo
- Várias das otimizações **locais** podem ser aplicadas como otimização **peephole**
- Devem ser aplicadas várias vezes para obter o efeito máximo