Compiladores

CIÊNCIA DA COMPUTAÇÃO

Prof. Dr. João Paulo Aramuni

UNIVERSIDADE FUMEC

Sumário

* **Otimização de Código**
  * Introdução
  * Otimização de registros e de blocos sequenciais de código
  * Ciclos de controle de fluxo
  * Análise de fluxo de dados
  * Resolução de equações de fluxo de dados
  * Transformações de código gerado

Otimização de Código

* **Introdução**
  * Construções de linguagem de alto nível podem introduzir um custo substancial em tempo de execução se traduzirmos ingenuamente cada construção independente para código de máquina.
  * Esta aula discute como eliminar muitas dessas ineficiências.

Otimização de Código

* **Introdução**
  * A eliminação de instruções desnecessárias no código objeto, ou a substituição de uma sequência de instruções por uma sequência de instruções mais rápida, que efetua a mesma operação, usualmente é denominada ‘melhoria do código’ ou ‘otimização do código’.
  * Todas as otimizações que veremos são “otimizações independentes de máquina”.
  * Utilizaremos código de três endereços e grafos de fluxo para compreender as estratégias de otimização.

Otimização de Código

* **Introdução**
  * A otimização de código **local** (melhoria de código dentro de um bloco básico) pode ocorrer durante a geração de código.
  * A fase de otimização irá tratar da otimização de código **global,** no qual as melhorias levam em conta o que acontece entre os blocos básicos.
  * Começaremos discutindo as principais oportunidades para melhoria de código.

Otimização de Código

* **Introdução**
  * A maioria das otimizações globais é baseada na **análise de fluxo de dados**, que são algoritmos para colher informações sobre um programa.
  * Todos os resultados da análise de fluxo de dados têm o mesmo formato: para cada instrução do programa, especificam alguma propriedade que deve ser mantida toda vez que a instrução for executada.
  * As análises diferem nas propriedades que calculam.

Otimização de Código

* **Introdução**
  * Por exemplo, uma análise de propagação de constante calcula, para cada ponto no programa e para cada variável usada pelo programa, se essa variável possui um único valor constante nesse ponto.

  * Essa informação pode ser usada, por exemplo, para substituir referências de variável por valores constantes.

Otimização de Código

* **Introdução**
  * Outro exemplo, uma análise de tempo de vida (*liveness*) determina, para cada ponto no programa, se o valor mantido por uma variável particular nesse ponto com certeza será sobrescrito antes de ser lido.

  * Se for, não precisaremos preservar esse valor, seja em um registrador, seja em um endereço da memória.

Otimização de Código

* **Introdução**
  * Veremos agora as principais fontes de otimização:
    * Causas de redundância
    * Transformações com preservação semântica
    * Subexpressões comuns globais
    * Propagação de cópia
    * Eliminação de código morto
    * Movimentação de código
    * Variáveis de indução e redução de força

Otimização de Código

* **Otimização de registros e de blocos sequenciais de código**
* **As principais fontes de otimização**
  * Uma otimização de compilador deve preservar a semântica do programa original. Exceto em circunstâncias muito especiais, quando um programador escolhe e implementa um algoritmo particular, o compilador não pode saber o suficiente sobre o programa para substituí-lo por um algoritmo substancialmente diferente e mais eficaz.

Otimização de Código

* **Otimização de registros e de blocos sequenciais de código**
* **As principais fontes de otimização**
  * Um compilador sabe apenas como aplicar transformações semânticas relativamente de baixo nível, usando fatos gerais como identidades algébricas, do tipo $i + 0 = i$, ou a semântica do programa, como o fato de que realizar a mesma operação sobre os mesmos valores gera o mesmo resultado.

Otimização de Código

* **Otimização de registros e de blocos sequenciais de código**
* **As principais fontes de otimização**
  * **Causas de redundância**
    * Existem muitas operações redundantes de um programa típico. Às vezes, a redundância está disponível no nível de fonte.
    * Por exemplo, um programador pode achar mais direto e conveniente recalcular algum resultado, deixando para o compilador reconhecer que somente um desses cálculos é necessário.

Otimização de Código

- **Otimização de registros e de blocos sequenciais de código**
- **As principais fontes de otimização**
  - **Causas de redundância**
    - Porém, mais frequentemente, a redundância é um efeito colateral de o programa ter sido escrito em uma linguagem de alto nível.

    - Na maioria das linguagens (que não sejam C ou C++, em que a aritmética de apontador é permitida), os programadores não têm escolha, exceto referir-se aos elementos de um arranjo ou aos campos de uma estrutura por meio de acessos do tipo $A[i][j]$ por exemplo.

Otimização de Código

- **Otimização de registros e de blocos sequenciais de código**
- **As principais fontes de otimização**
  - **Causas de redundância**
    - Enquanto um programa é compilado, cada um desses acessos à estrutura de dados de alto nível se expande em uma série de operações aritméticas de baixo nível, como o cálculo do endereço do elemento (i, j)-ésimo de uma matriz A.
    - Os acessos à mesma estrutura de dados frequentemente compartilham muitas operações comuns de baixo nível. Os programadores não conhecem essas operações de baixo nível e não podem eliminar as redundâncias.

Otimização de Código

* **Otimização de registros e de blocos sequenciais de código**
* **As principais fontes de otimização**
  * **Causas de redundância**
    * Realmente é preferível, do ponto de vista da engenharia de software, que os programadores só acessem os elementos de dados por seus nomes em alto nível; os programas são mais fáceis de escrever e, principalmente, mais fáceis de entender e desenvolver.
    * Deixando que um compilador elimine as redundâncias, teremos o melhor dos dois mundos: programas mais eficientes e mais fáceis de administrar.

Otimização de Código

- **Otimização de registros e de blocos sequenciais de código**
- **As principais fontes de otimização**
  - **Um exemplo executável: Quicksort**
    - A seguir, usaremos um fragmento de um programa de ordenação, chamado *quicksort*, para ilustrar várias transformações importantes para melhoria de código.
    - A Fig. 1 a seguir demonstra o programa em C para o *quicksort*.

Otimização de Código

* Fig. 1, código em C para o *quicksort*

```
void quicksort(int m, int n)
     /* ordena recursivamente a[m] até a[n] */
{
     int i, j;
     int v, x;
     if (n <= m) return;
     /* o fragmento começa aqui */
     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; /* 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 C para o quicksort.

Otimização de Código

- **Otimização de registros e de blocos sequenciais de código**
- **As principais fontes de otimização**
  - **Um exemplo executável: Quicksort**
    - Antes que possamos otimizar para remover as redundâncias nos cálculos de endereço, as operações de endereço em um programa precisam ser desmembradas em operações aritméticas de baixo nível, para expor as redundâncias.
    - Iremos considerar que a representação intermediária consiste em comandos de três endereços, onde variáveis temporárias são usadas para manter todos os resultados intermediários das expressões.

Otimização de Código

* **Otimização de registros e de blocos sequenciais de código**
* **As principais fontes de otimização**
  * **Um exemplo executável: Quicksort**
    * O código intermediário para o fragmento marcado no programa da Fig. 1, através de comentários, é mostrado a seguir na Fig. 2.

Otimização de Código

* Fig. 2, código de três endereços do fragmento da Fig. 1

```
(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
```

Código de três endereços do fragmento da Figura 1.

Otimização de Código

* **Otimização de registros e de blocos sequenciais de código**
* **As principais fontes de otimização**
  * **Um exemplo executável: Quicksort**
    * Neste exemplo, consideramos que os inteiros ocupam quatro bytes. A atribuição `x = a[i]` é traduzida para as duas instruções de três endereços:

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

    * como mostram as etapas (14) e (15) da Fig. 2. De modo semelhante, `a[j] = x` torna-se:

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

    * como mostram as etapas (20) e (21).

Otimização de Código

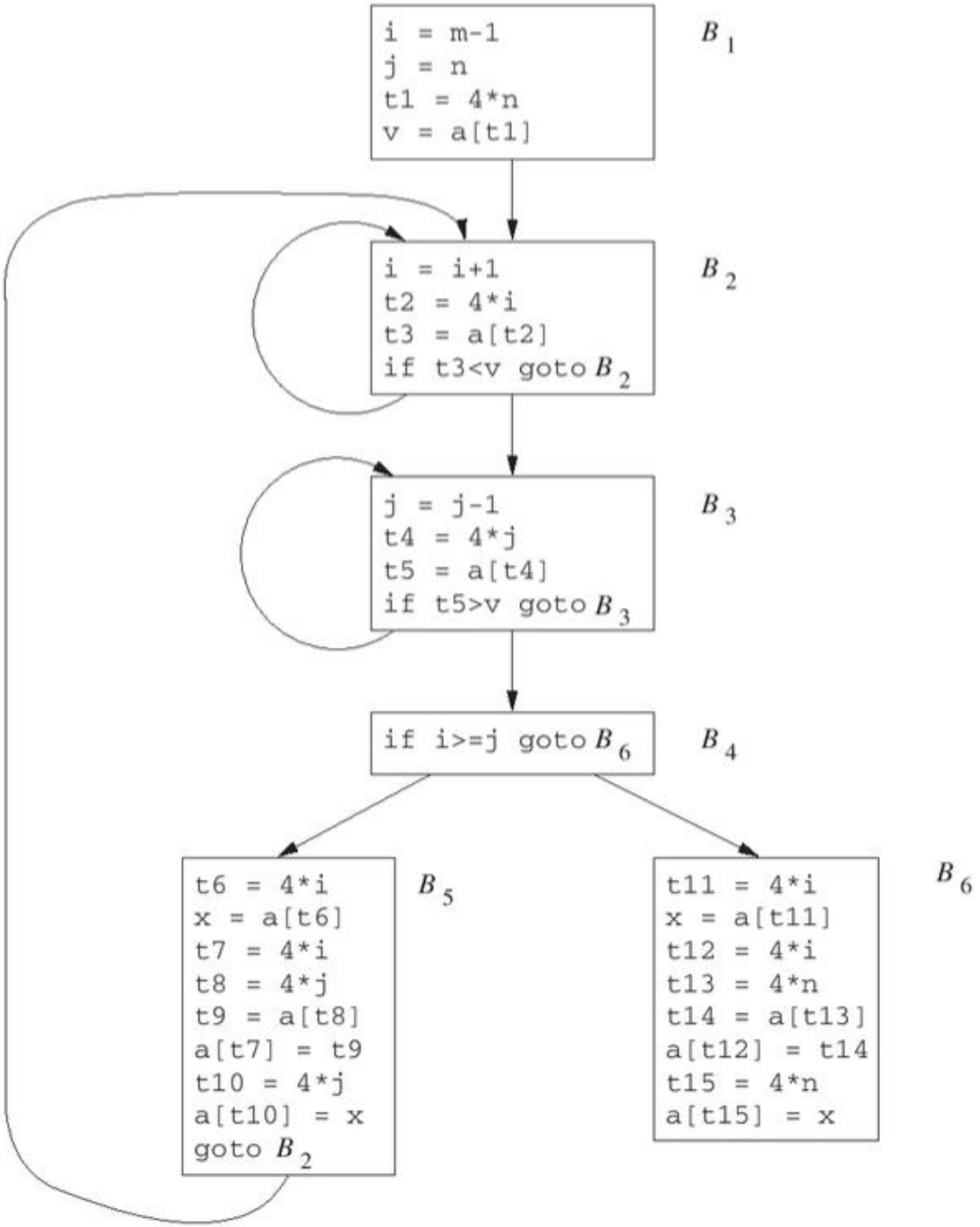
* **Otimização de registros e de blocos sequenciais de código**
* **As principais fontes de otimização**
  * **Um exemplo executável: Quicksort**
    * Observe que todo acesso a arranjo no programa original é traduzido em um par de passos, consistindo em uma multiplicação e uma operação de subscrito de arranjo.
    * Como resultado, esse pequeno fragmento de programa é traduzido para uma sequência um tanto longa de operações de três endereços.

Otimização de Código

- **Otimização de registros e de blocos sequenciais de código**
- **As principais fontes de otimização**
  - **Um exemplo executável: Quicksort**
    - A Fig. 3 mostrada a seguir é o grafo de fluxo para o programa da Fig. 2. O bloco *B1* é o nó de entrada. Todos os desvios condicionais e incondicionais para os comandos da Fig. 2 foram substituídos na Fig. 3 por desvios para o bloco no qual os comandos são líderes.

    - Na Fig. 3, existem três loops. Os blocos *B2* e *B3* são loops por si sós. Os blocos *B2*, *B3*, *B4* e *B5* juntos formam um loop, com um único ponto de entrada em *B2*.

* Fig. 3



Grafo de fluxo para o fragmento do quicksort.

Otimização de Código

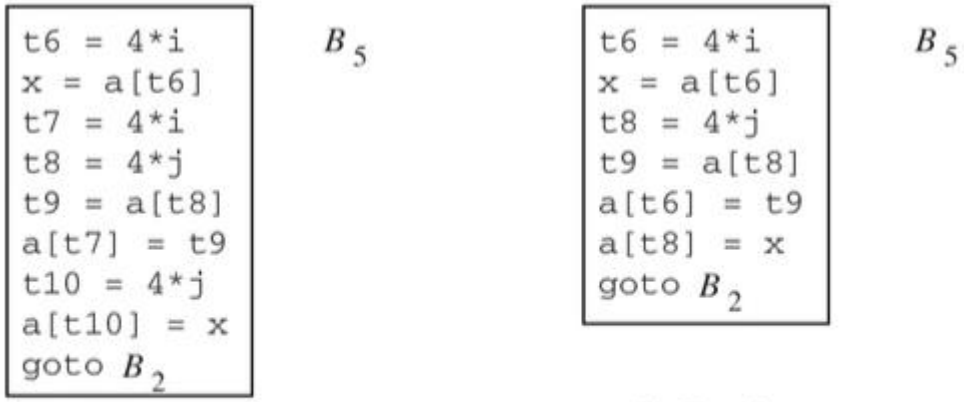
* **Otimização de registros e de blocos sequenciais de código**
* **As principais fontes de otimização**
  * **Transformações com preservação semântica**
    * Existem várias maneiras pelas quais um compilador pode melhorar um programa sem alterar a função que ele calcula.

    * Eliminação de subexpressão comum, propagação de cópia, eliminação de código morto e desdobramento de constantes são exemplos comuns dessas transformações de preservação de função (ou preservação de semântica); vamos examinar cada uma delas.

Otimização de Código

* **Otimização de registros e de blocos sequenciais de código**
* **As principais fontes de otimização**
  * **Transformações com preservação semântica**
    * Frequentemente, um programa incluirá vários cálculos do mesmo valor, como um deslocamento em um arranjo.
    * Conforme vimos anteriormente, alguns desses cálculos não podem ser evitados pelo programador, porque estão abaixo do nível de detalhe acessível em uma linguagem fonte.

* **Otimização de registros e de blocos sequenciais de código**
* **As principais fontes de otimização**
  * **Transformações com preservação semântica**
    * Por exemplo, o bloco *B5* mostrado na Fig. 4(a) recalcula `4 * i` e `4 * j`, embora nenhum desses cálculos fosse solicitado explicitamente pelo programador.



Eliminação de subexpressão comum local.

Otimização de Código

* **Otimização de registros e de blocos sequenciais de código**
* **As principais fontes de otimização**
  * **Subexpressões comuns globais**
    * Uma ocorrência de uma expressão *E* é chamada subexpressão comum se *E* tiver sido computado anteriormente e os valores das variáveis em *E* não tiverem mudado desde a computação anterior.
    * Evitamos recomputar *E* se pudermos usar seu valor computado anteriormente; ou seja, se a variável x à qual a computação anterior de *E* foi atribuída não tiver mudado nesse ínterim.

Otimização de Código

- **Otimização de registros e de blocos sequenciais de código**
- **As principais fontes de otimização**
  - **Subexpressões comuns globais**
    - Exemplo 1: As atribuições `a t7` e `t10` na Fig. 4(a) computam as subexpressões comuns `4 * i` e `4 * j`, respectivamente. Esses passos foram eliminados na Fig. 4(b), que usa `t6` no lugar de `t7` e `t8` no lugar de `t10`.

Otimização de Código

* **Otimização de registros e de blocos sequenciais de código**
* **As principais fontes de otimização**
  * **Subexpressões comuns globais**
    * Exemplo 2: A Fig. 5 mostra o resultado da eliminação de subexpressões globais e locais nos blocos *B5* e *B6* do grafo de fluxo da Fig. 3.
    * Primeiro, discutimos a transformação de *B5*, e depois mencionamos algumas sutilezas envolvendo arranjos.

* **Otimização de registros e de blocos sequenciais de código**
* **As principais fontes de otimização**
  * **Subexpressões comuns globais**
    * Depois que as subexpressões comuns forem eliminadas, *B5* ainda avalia `4 * i` e `4 * j`, como mostra a Fig. 4(b). Ambas são subexpressões comuns; em particular, os três comandos:

```
t8 = 4*j
t9 = a[t8]
a[t8] = x
```

    * em *B5* podem ser substituídos por:

```
t9 = a[t4]
a[t4] = x
```

    * usando *t4* computado no bloco *B3*.

Otimização de Código

* **Otimização de registros e de blocos sequenciais de código**
* **As principais fontes de otimização**
  * **Subexpressões comuns globais**
    * Na Fig. 5, observe que, quando o controle passa da avaliação de `4 * j` em *B3* para *B5*, não há mudança em *j* e nem em *t4*, assim *t4* pode ser usado se `4 * j` for necessário.

Otimização de Código

* **Otimização de registros e de blocos sequenciais de código**
* **As principais fontes de otimização**
  * **Subexpressões comuns globais**
    * Outra subexpressão comum aparece em *B5* após *t4* substituir *t8*. A nova expressão `a[t4]` corresponde ao valor de `a[j]` no nível de fonte.

    * Não apenas *j* retém seu valor quando o controle sai de *B3* e então entra em *B5*, mas também `a[j]`, um valor computado em um temporário *t5*, pois não existem atribuições aos elementos de arranjo `a` nesse ínterim.

Otimização de Código

- **Otimização de registros e de blocos sequenciais de código**
- **As principais fontes de otimização**
  - **Subexpressões comuns globais**
    - Os comandos:

```
t9 = a[t4]
a[t6] = t9
```

    - em *B5*, portanto, podem ser substituídos por:

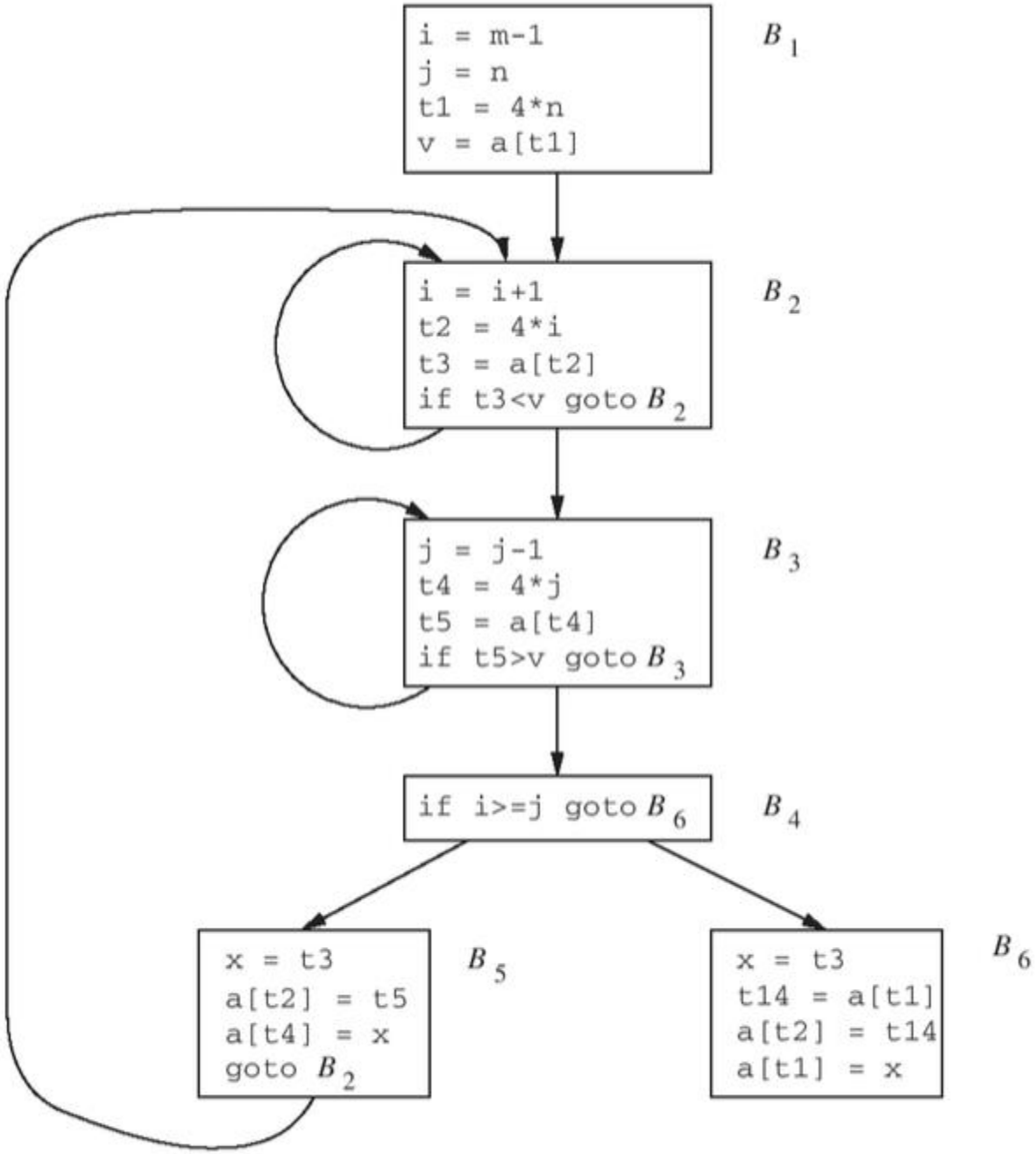
```
a[t6] = t5
```

Otimização de Código

- **Otimização de registros e de blocos sequenciais de código**
- **As principais fontes de otimização**
  - **Subexpressões comuns globais**
    - De modo semelhante, o valor atribuído a *x* no bloco *B5* da Fig. 4(b) é visto como sendo o mesmo que o valor atribuído a *t3* no bloco *B2*. O bloco *B5* na Fig. 5 é o resultado da eliminação das subexpressões comuns correspondentes aos valores das expressões em nível de fonte `a[i]` e `a[j]` de *B5* na Fig. 4(b).

    - Uma série de transformações semelhantes foi feita em *B6* na Fig. 5 a seguir:

* Fig. 5



B_5 e B_6 após a eliminação de subexpressão comum.

Otimização de Código

* **Otimização de registros e de blocos sequenciais de código**
* **As principais fontes de otimização**
  * **Subexpressões comuns globais**
    * A expressão `a[t1]` nos blocos *B1* e *B6* da Fig. 5 não é considerada subexpressão comum, embora *t1* possa ser usado nos dois lugares.

    * Depois que o controle sair de *B1* e antes de alcançar *B6*, ele pode passar por *B5*, onde existem atribuições para `a`. Logo, `a[t1]` pode não ter o mesmo valor ao atingir *B6*, como tinha ao sair de *B1*, e não é seguro tratar `a[t1]` como uma subexpressão comum.

Otimização de Código

* **Otimização de registros e de blocos sequenciais de código**
* **As principais fontes de otimização**
  * **Propagação de cópia**
    * O bloco *B5*, na Fig. 5, pode ser melhorado ainda mais eliminando-se x, por meio de duas novas transformações. Uma delas trata das atribuições da forma u = v, chamadas instruções de cópia, ou cópias, para abreviar.

    * Se tivéssemos entrado em mais detalhes no Exemplo 2, as cópias teriam surgido muito mais cedo, pois o algoritmo normal para eliminar subexpressões comuns as introduz, assim como vários outros algoritmos.

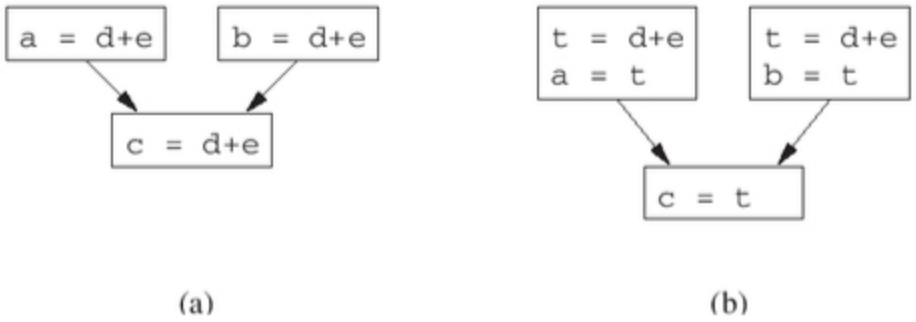
Otimização de Código

* **Otimização de registros e de blocos sequenciais de código**
* **As principais fontes de otimização**
  * **Propagação de cópia**
    * Exemplo 3: Para eliminar a subexpressão comum do comando c = d + e na Fig. 6(a), temos de usar uma nova variável *t* para conter o valor de d + e.

    * O valor da variável *t*, em vez daquele da expressão d + e, é atribuído a e na Fig. 6(b). Como o controle pode alcançar c = d + e depois da atribuição de a ou depois da atribuição de b, seria incorreto substituir c = d + e por c = a ou por c = b.

Otimização de Código

* **Otimização de registros e de blocos sequenciais de código**
* **As principais fontes de otimização**
  * **Propagação de cópia**
    * Abaixo, Fig. 6:



Cópias introduzidas durante a eliminação de subexpressão comum.

Otimização de Código

- **Otimização de registros e de blocos sequenciais de código**
- **As principais fontes de otimização**
  - **Propagação de cópia**
    - A ideia por trás da transformação por propagação de cópia é usar v ou u, sempre que possível, após o comando de cópia $u = v$.
    - Por exemplo, a atribuição `x = t3` no bloco *B5* da Fig. 5 é uma cópia. A propagação de cópia aplicada a *B5* gera o código da Fig. 7. Essa mudança pode não parecer uma melhoria, mas, conforme veremos a seguir, ela nos dá a oportunidade de eliminar a atribuição a `x`.

Otimização de Código

* **Otimização de registros e de blocos sequenciais de código**
* **As principais fontes de otimização**
  * **Propagação de cópia**
    * Abaixo, Fig. 7:

```
x = t3
a[t2] = t5
a[t4] = t3
goto B2
```

Bloco básico B_5 após propagação de cópia.

Otimização de Código

* **Otimização de registros e de blocos sequenciais de código**
* **As principais fontes de otimização**
  * **Eliminação de código morto**
    * Uma variável estará viva em algum ponto de um programa se seu valor puder ser usado posteriormente; caso contrário, ela está morta nesse ponto.

    * Uma ideia relacionada é o código morto (ou inútil) – comandos que capturam valores que nunca serão usados. Embora o programador provavelmente não introduza código morto intencionalmente, ele pode aparecer como resultado de transformações anteriores.

Otimização de Código

* **Otimização de registros e de blocos sequenciais de código**
* **As principais fontes de otimização**
  * **Eliminação de código morto**
    * Exemplo 4: Suponha que `debug` seja definido como TRUE ou FALSE em vários pontos do programa, e usado em comandos como:

```
if (debug) print...
```

    * Talvez seja possível para o compilador deduzir que, toda vez que o programa atinge esse comando, o valor de debug é FALSE.

Otimização de Código

* **Otimização de registros e de blocos sequenciais de código**
* **As principais fontes de otimização**
  * **Eliminação de código morto**
    * Usualmente, isso acontece porque existe um comando particular:

```
debug = FALSE
```

    * que deve ser a última atribuição para `debug` antes de quaisquer testes do valor de `debug`, não importa a sequência de desvios que o programa realmente tome.

Otimização de Código

* **Otimização de registros e de blocos sequenciais de código**
* **As principais fontes de otimização**
  * **Eliminação de código morto**
    * Se a propagação de cópia substituir `debug` por `FALSE`, **então o comando `print` estará morto, porque não poderá ser alcançado.**

    * Podemos eliminar o teste e a operação `print` do código objeto. Geralmente deduzir em tempo de compilação que o valor de uma expressão seja uma constante e usar a constante em seu lugar é algo conhecido como **desdobramento de constante**.

Otimização de Código

* **Otimização de registros e de blocos sequenciais de código**
* **As principais fontes de otimização**
  * **Eliminação de código morto**
    * Uma vantagem da propagação de cópia é que ela frequentemente transforma comandos de cópia em código morto. Por exemplo, a propagação de cópia seguida pela eliminação de código morto remove a atribuição a `x` e transforma o código da Fig. 7 em:

```
a[t2] = t5
a[t4] = t3
goto B2
```

    * Esse código é outra melhoria para o bloco *B5* da Fig. 5.

Otimização de Código

- **Ciclos de controle de fluxo**
- **As principais fontes de otimização**
  - **Movimentação de código**
    - Os *loops* constituem um local muito importante para otimizações, especialmente os *loops* internos, nos quais os programas costumam gastar a maior parte de seu tempo.
    - O tempo de execução de um programa pode ser melhorado se diminuirmos o número de instruções em um *loop* interno, mesmo se aumentarmos a quantidade de código fora desse *loop*.

Otimização de Código

- **Ciclos de controle de fluxo**
- **As principais fontes de otimização**
  - **Movimentação de código**
    - Uma modificação importante que diminui a quantidade de código em um *loop* é a movimentação de código.
    - Essa transformação pega uma expressão que gera o mesmo resultado, independentemente do número de vezes que um *loop* é executado (um cálculo do invariante do *loop*) e a avalia antes do *loop*. Observe que a noção 'antes do loop' assume a existência de uma entrada para o *loop*, ou seja, um bloco básico para o qual seguirão todos os desvios de fora do *loop*.

Otimização de Código

- **Ciclos de controle de fluxo**
- **As principais fontes de otimização**
  - **Movimentação de código**
    - Exemplo 5: A avaliação de *limit* - 2 é um cálculo do invariante do *loop* no comando while a seguir:

```
while(i <= limit - 2)/*comando não muda limit*/
```

    - A movimentação de código resultará no código equivalente:

```
t = limit - 2
while(i <= t)/*comando não muda limit nem t*/
```

Otimização de Código

* **Ciclos de controle de fluxo**
* **As principais fontes de otimização**
  * **Movimentação de código**
    * Agora, o cálculo de *limit* - 2 é realizado uma vez, antes de entrarmos no *loop*.
    * Anteriormente, haveria *n* + 1 cálculos de *limit* - 2 se tivéssemos *n* interações do corpo do *loop*.

Otimização de Código

- **Ciclos de controle de fluxo**
- **As principais fontes de otimização**
  - **Variáveis de indução e redução de força**
    - Outra otimização importante é encontrar variáveis de indução em *loops* e otimizar seu cálculo.
    - Uma variável *x* é considerada uma 'variável de indução' se houver uma constante positiva ou negativa *c* tal que, toda vez que *x* for atribuído, seu valor aumenta de acordo com *c*.
    - Por exemplo, *i* e *t2* são variáveis de indução no loop contendo *B2* da Fig. 5.

Otimização de Código

* **Ciclos de controle de fluxo**
* **As principais fontes de otimização**
  * **Variáveis de indução e redução de força**
    * Variáveis de indução podem ser computadas com um único incremento (adição ou subtração) a cada iteração de *loop*.
    * A transformação para substituir uma operação cara, como a multiplicação, por outra menos dispendiosa, como a adição, é conhecida como **redução de força**.

Otimização de Código

* **Ciclos de controle de fluxo**
* **As principais fontes de otimização**
  * **Variáveis de indução e redução de força**
    * Mas as variáveis de indução não apenas nos permitem efetuar às vezes uma redução de força.
    * Frequentemente, é possível eliminar tudo, menos um grupo de variáveis de indução cujos valores permanecem em passo sincronizado enquanto percorremos o *loop*.

Otimização de Código

* **Ciclos de controle de fluxo**
* **As principais fontes de otimização**
  * **Variáveis de indução e redução de força**
    * Quando o loop é processado, é útil trabalhar 'de dentro para fora'; ou seja, começar com os *loops* internos e prosseguir para os *loops* envolventes, progressivamente maiores.

    * Assim, veremos como essa otimização se aplica ao nosso exemplo de *quicksort*, começando com um dos *loops* mais internos: *B3*, isoladamente.

Otimização de Código

- **Ciclos de controle de fluxo**
- **As principais fontes de otimização**
  - **Variáveis de indução e redução de força**
    - Observe que os valores de j e $t4$ permanecem em sincronismo; toda vez que o valor de j diminui em 1, o valor de $t4$ diminui em 4, porque 4 * j é atribuído a $t4$.
    - Essas variáveis, j e 4, formam assim um bom exemplo de um par de variáveis de indução.

Otimização de Código

* **Ciclos de controle de fluxo**
* **As principais fontes de otimização**
  * **Variáveis de indução e redução de força**
    * Quando há duas ou mais variáveis de indução em um *loop*, pode ser possível livrar-se de todas menos de uma.

    * Para o loop interno de *B3* na Fig. 5, não podemos livrar-nos completamente nem de *j* nem de *t4*: *t4* é usada em *B3* e *j* é usada em *B4*. Contudo, podemos ilustrar a redução de força em uma parte do processo de eliminação da variável de indução. Por fim, *j* será eliminada quando o *loop* externo, consistindo nos blocos *B2*, *B3*, *B4* e *B5*, for considerado.

Otimização de Código

- **Ciclos de controle de fluxo**
- **As principais fontes de otimização**
  - **Variáveis de indução e redução de força**
    - Exemplo 6: Como o relacionamento $t4 = 4 * j$ certamente se mantém após a atribuição a $t4$ na Fig. 5, e $t4$ não é alterado em nenhum outro lugar no loop interno de $B3$, segue-se que, logo depois do comando $j = j - 1$, o relacionamento $t4 = 4 * j + 4$ deverá ser mantido.

    - Portanto, podemos substituir a atribuição $t4 = 4 * j$ por $t4 = t4 - 4$. O único problema é que $t4$ não tem um valor quando entramos no bloco $B3$ pela primeira vez.

Otimização de Código

* **Ciclos de controle de fluxo**
* **As principais fontes de otimização**
  * **Variáveis de indução e redução de força**
    * Como devemos manter o relacionamento *t4* = 4 * *j* na entrada do bloco *B3*, colocamos uma inicialização de *t4* no fim do bloco onde o próprio *j* é inicializado, mostrado pela adição da parte tracejada no bloco *B1* da Fig. 8.

    * Embora tenhamos incluído mais uma instrução, que é executada uma vez no bloco *B1*, a substituição de uma multiplicação por uma subtração acelerará o código objeto se a multiplicação exigir mais tempo que a adição ou subtração, como é o caso em muitas máquinas.

Otimização de Código

- **Ciclos de controle de fluxo**
- **As principais fontes de otimização**
  - **Variáveis de indução e redução de força**
    - Vejamos mais um caso de eliminação da variável de indução.
    - Este exemplo trata i e j no contexto do *loop* externo contendo *B2*, *B3*, *B4* e *B5*.

Otimização de Código

- **Ciclos de controle de fluxo**
- **As principais fontes de otimização**
  - **Variáveis de indução e redução de força**
    - Exemplo 7: Após a redução de força ser aplicada aos *loops* internos em torno de *B2* e *B3*, o único uso de i e j é determinar o resultado do teste no bloco *B4*.

    - Sabemos que os valores de i e *t2* satisfazem o relacionamento *t2* = 4 * i, enquanto os de j e *t4* satisfazem o relacionamento *t4* = 4 * j. Assim, o teste *t2* >= *t4* pode substituir i>=j.

Otimização de Código

* **Ciclos de controle de fluxo**
* **As principais fontes de otimização**
  * **Variáveis de indução e redução de força**
    * Quando essa substituição é feita, *i* no bloco *B2* e *j* no bloco *B3* se tornam variáveis mortas, e as atribuições a eles nesses blocos se tornam código morto, podendo ser eliminadas.

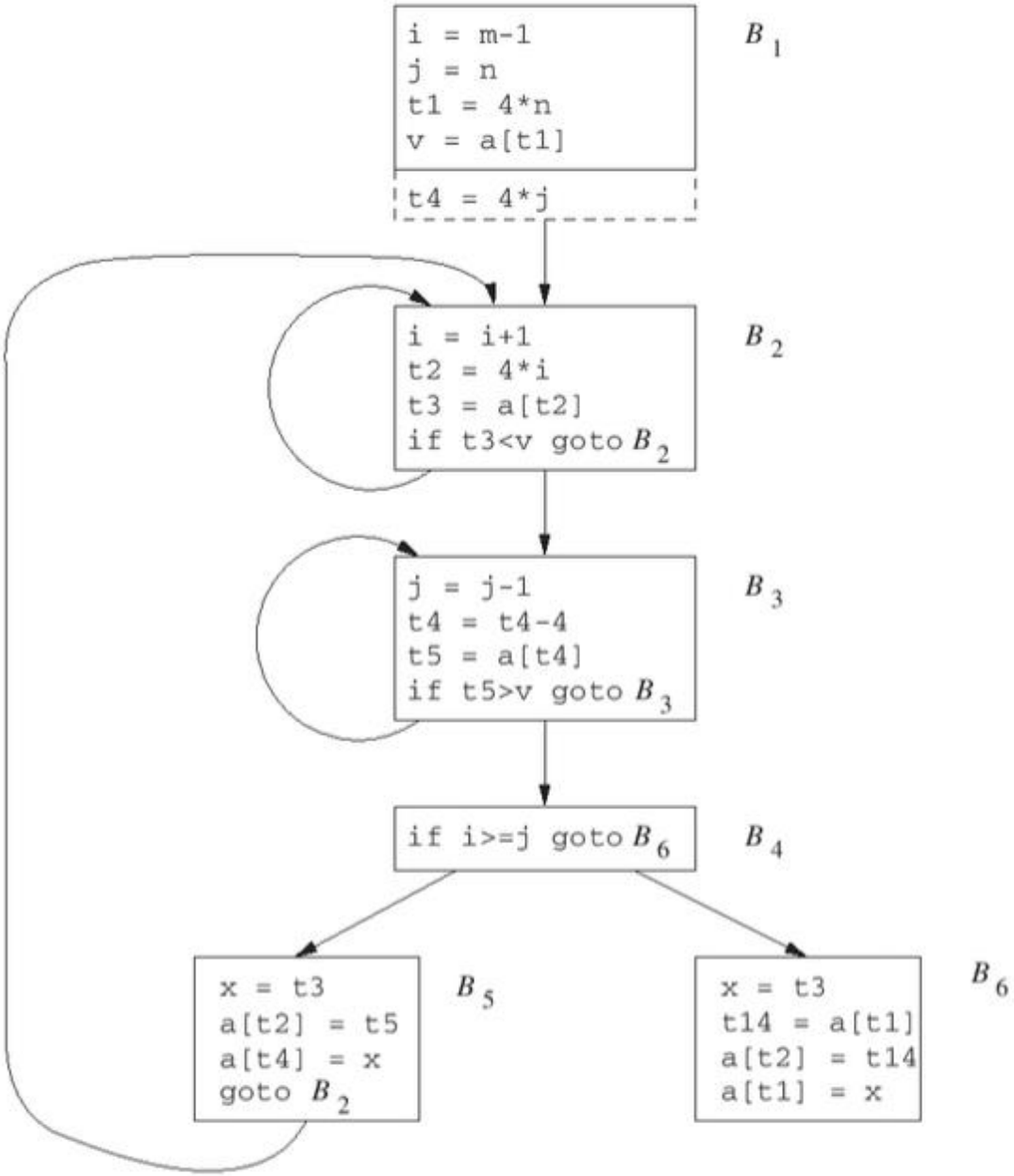
    * O grafo de fluxo resultante aparece na Fig. 9.

Otimização de Código

* **Ciclos de controle de fluxo**
* **As principais fontes de otimização**
  * **Variáveis de indução e redução de força**
    * As transformações para melhoria de código que discutimos foram eficazes.

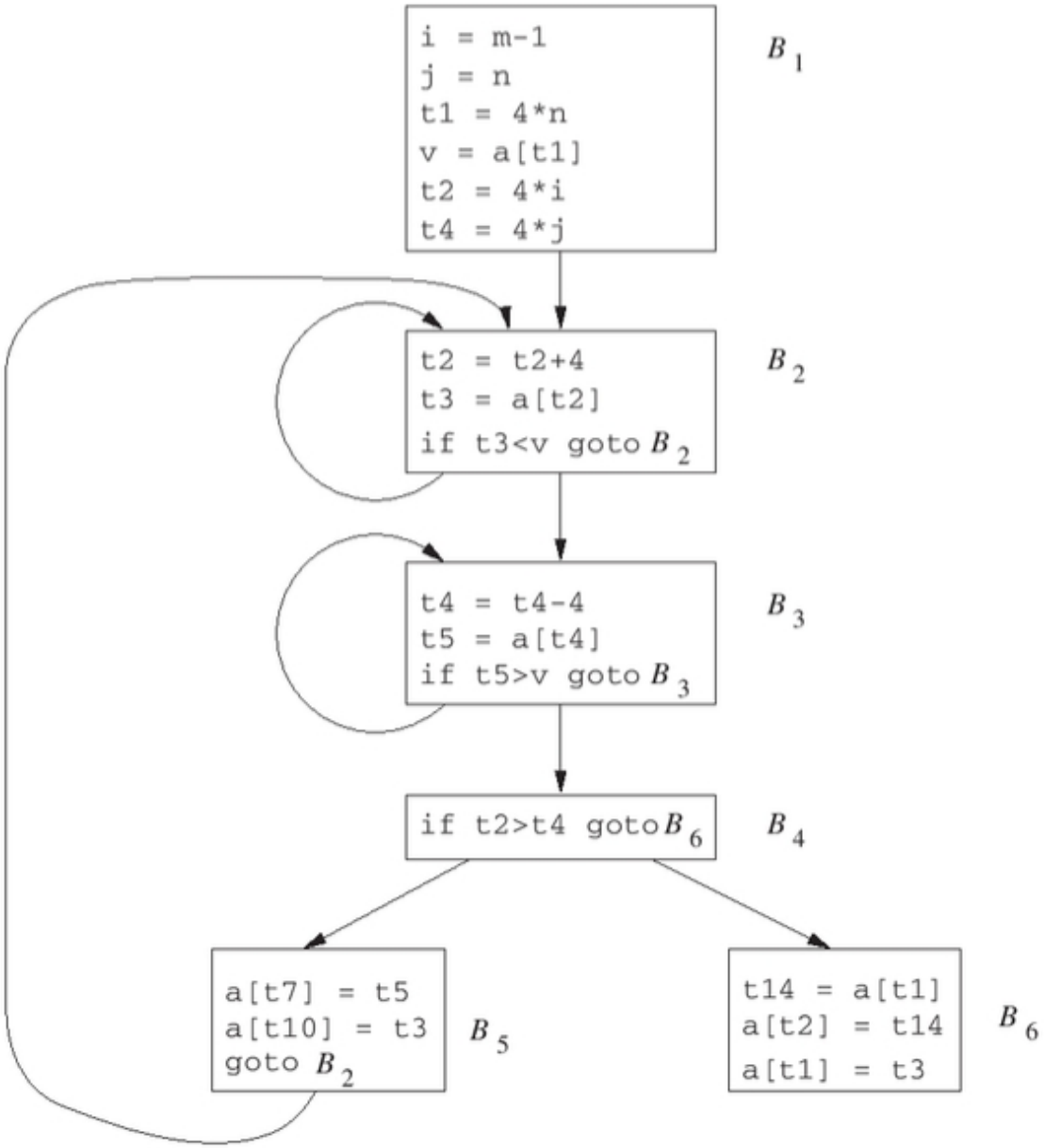
    * Na Fig. 9, o número de instruções nos blocos *B2* e *B3* foi reduzido de 4 para 3, em comparação com o grafo de fluxo original da Fig. 3. Em *B5*, o número foi reduzido de 9 para 3; e em *B6*, de 8 para 3. É verdade que *B1* aumentou de quatro instruções para seis, mas *B1* é executado apenas uma vez no fragmento de código, de modo que o tempo total de execução quase não é afetado pelo tamanho de *B1*.

* Fig. 8



Redução de força aplicada a 4 $\star$ j no bloco B_3.

* Fig. 9

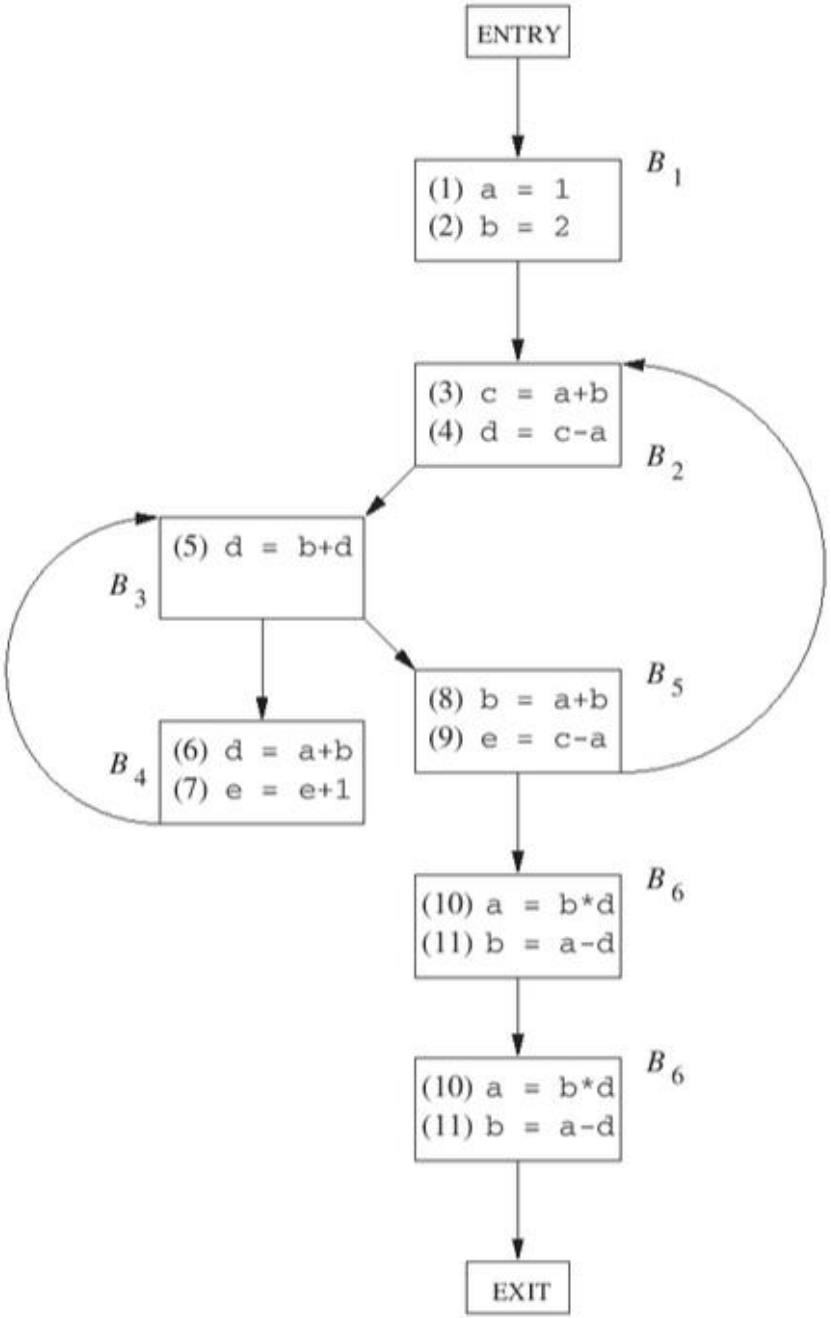


Grafo de fluxo após eliminação da variável de indução.

Otimização de Código

* **As principais fontes de otimização**
  * **Exercício 1**
    * Para o grafo de fluxo da Fig. 10, a seguir:
    * a) Identifique os *loops* do grafo de fluxo.
    * b) Os comandos (1) e (2) em *B1* são ambos comandos de cópia, em que a e b recebem valores constantes. Para quais usos de a e b podemos realizar propagação de cópia e substituir esses usos de variáveis por usos de uma constante? Faça isso, sempre que possível.
    * c) Identifique as subexpressões globais comuns para cada *loop*.
    * d) Identifique todas as variáveis de indução para cada *loop*. Não se esqueça de levar em conta todas as constantes introduzidas em (b).
    * e) identifique todos os cálculos de invariantes de *loop* para cada *loop*.

* Fig. 10



Grafo de fluxo

Otimização de Código

- **As principais fontes de otimização**
  - **Exercício 2**
    - Na Fig. 11 está o código intermediário para calcular o produto pontual de dois vetores A e B. Otimize esse código eliminando subexpressões comuns, realizando a redução de força nas variáveis de indução e eliminando todas as variáveis de indução que você puder.

```
    dp = 0.
    i = 0
L:  t1 = i*8
    t2 = A[t1]
    t3 = i*8
    t4 = B[t3]
    t5 = t2*t4
    dp = dp+t5
    i = i+1
    if i<n goto L
```

Código intermediário para calcular o produto pontual.

Otimização de Código

* **Análise de fluxo de dados**
  * Todas as otimizações introduzidas anteriormente dependem da **análise de fluxo de dados**. A análise de fluxo de dados refere-se a um conjunto de técnicas que derivam informações sobre o fluxo de dados ao longo dos caminhos de execução do programa.

  * Por exemplo, uma maneira de implementar a eliminação de subexpressão comum global requer que determinemos se duas expressões textualmente idênticas são avaliadas com o mesmo valor em qualquer caminho de execução possível do programa.

Otimização de Código

* **Análise de fluxo de dados**
  * Como outro exemplo, se o resultado de uma atribuição não for usado ao longo de algum caminho de execução subsequente, podemos eliminar a atribuição como código morto.

  * Estas e muitas questões importantes podem ser respondidas pela análise de fluxo de dados.

Otimização de Código

* **Análise de fluxo de dados**
  * Como vimos, a execução de um programa pode ser entendida como uma série de transformações do estado do programa, que consiste nos valores de todas as variáveis do programa, incluindo aquelas associadas aos registros de ativação abaixo do topo da pilha de execução.

  * Cada execução de um comando em código intermediário transforma um estado de entrada em um novo estado de saída. O estado de entrada está associado ao **ponto do programa antes do comando** e o estado de saída está associado ao **ponto do programa após o comando**.

Otimização de Código

* **Análise de fluxo de dados**
  * Quando analisamos o comportamento de um programa devemos considerar, usando um grafo de fluxo, todas as sequências possíveis de pontos do programa (‘caminhos’) que sua execução pode tomar.

  * Então, extraímos dos possíveis estados do programa em cada ponto a informação de que precisamos para o problema específico de análise de fluxo de dados que queremos solucionar.

Otimização de Código

* **Análise de fluxo de dados**
  * Nas análises mais complexas, devemos considerar caminhos que desviam para grafos de fluxo associados a diversos procedimentos, enquanto chamadas e retornos são executados.
  * Contudo, para iniciar nosso estudo, vamos concentrar-nos nos caminhos por um único grafo de fluxo para um único procedimento.

Otimização de Código

* **Análise de fluxo de dados**
  * Vejamos o que o grafo de fluxo nos diz sobre os possíveis caminhos de execução.
    * Em um bloco básico, o ponto do programa após um comando é o mesmo que o ponto do programa antes do próximo comando.
    * Se houver uma aresta do bloco *B1* para o bloco *B2*, o ponto do programa após o último comando de *B1* pode ser seguido imediatamente pelo ponto do programa antes do primeiro comando de *B2*.

Otimização de Código

* **Análise de fluxo de dados**
  * Assim, podemos definir um caminho de execução (ou apenas caminho) a partir do ponto *p1* para o ponto *pn* como sendo a sequência de pontos *p1*, *p2*, ..., *pn* tal que, para cada i = 1, 2, n - 1, ou:
    * 1. *pi* é o ponto imediatamente anterior a um comando e *pi* + 1 é o ponto imediatamente após esse mesmo comando, ou
    * 2. *pi* é o fim de algum bloco e *pi* + 1 é o início de um bloco sucessor.

Otimização de Código

* **Análise de fluxo de dados**
  * Geralmente, existe um número infinito de caminhos de execução possíveis ao longo de um programa, e não há um limite superior finito para o tamanho de um caminho de execução.
  * As análises de um programa resumem todos os estados de programa possíveis que podem ocorrer em um ponto do programa com um conjunto finito de fatos.
  * Diferentes análises podem escolher abstrair-se de informações diferentes e, em geral, nenhuma análise é necessariamente uma representação perfeita do estado.

Otimização de Código

* **Análise de fluxo de dados**
  * Exemplo 8: Até mesmo o programa simples da Fig. 12 descreve um número ilimitado de caminhos de execução.
  * Não entrando no loop em momento algum, o caminho de execução completo mais curto consiste nos pontos do programa (1, 2, 3, 4, 9).
  * O próximo caminho mais curto executa uma iteração do loop e consiste nos pontos (1, 2, 3, 4, 5, 6, 7, 8, 3, 4, 9).

Otimização de Código

* **Análise de fluxo de dados**
  * Sabemos que, por exemplo, a primeira vez em que o ponto (5) do programa é executado, o valor de a é 1 devido à definição *d1*.
  * Dizemos que *d1* alcança o ponto (5) na primeira iteração. Em iterações subsequentes, *d3* alcança o ponto (5) e o valor de a é 243.

Otimização de Código

* **Análise de fluxo de dados**
  * Em geral, não é possível manter o registro de todos os estados do programa para todos os caminhos possíveis.
  * Na análise de fluxo de dados, não distinguimos entre os caminhos tomados para alcançar um ponto do programa.
  * Além do mais, não registramos os estados inteiros; em vez disso, abstraímos certos detalhes, mantendo apenas os dados de que precisamos para efeito de análise.

Otimização de Código

* **Análise de fluxo de dados**
  * Dois exemplos ilustrarão como os mesmos estados de um programa podem conduzir a diferentes informações abstraídas em um ponto.

  * 1. Para auxiliar os usuários a depurar seus programas, podemos querer descobrir todos os valores que uma variável pode ter em um ponto do programa, e onde esses valores podem ser definidos. Por exemplo, podemos resumir todos os estados do programa no ponto (5) dizendo que o valor de *a* é um dentre {1,243}, e que ele pode ser definido por um dentre {*d1*, *d3*}. As definições que podem alcançar um ponto do programa ao longo de algum caminho são conhecidas como **definições de alcance**.

Otimização de Código

* **Análise de fluxo de dados**
  * Dois exemplos ilustrarão como os mesmos estados de um programa podem conduzir a diferentes informações abstraídas em um ponto.

  * 2. Suponha que, em vez disso, estejamos interessados em implementar o desdobramento de constante. Se em uso da variável *x* for alcançado apenas por uma definição, e essa definição atribuir uma constante a *x*, podemos simplesmente substituir *x* pela constante. Se, por outro lado, várias definições de *x* puderem alcançar um único ponto do programa, não podemos efetuar o desdobramento de constante para *x*. Assim, para o desdobramento de constante, queremos encontrar definições que sejam a definição única de sua variável a alcançar determinado ponto do programa, não importa qual seja o caminho de execução tomado.

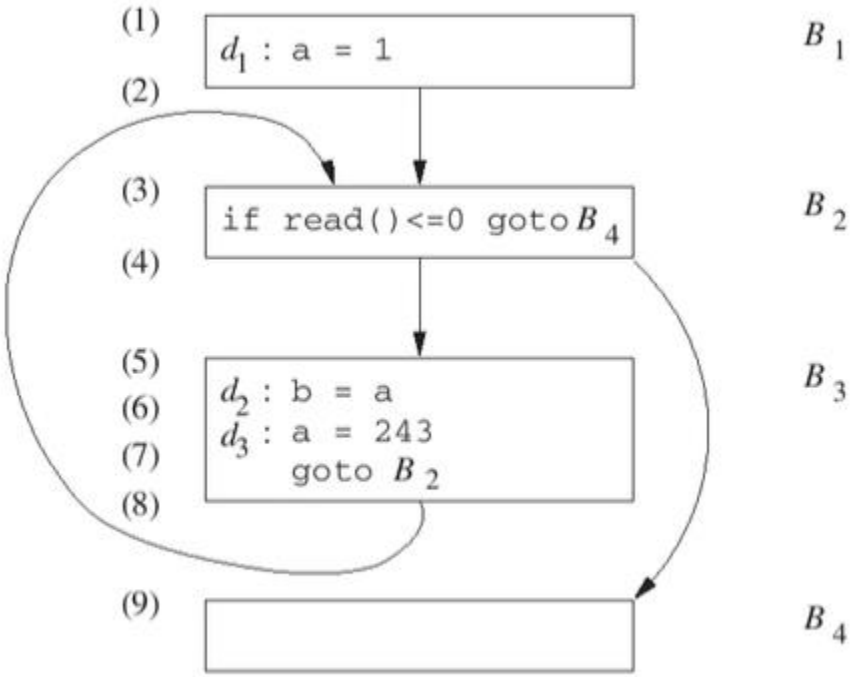
Otimização de Código

* **Análise de fluxo de dados**
  * Dois exemplos ilustrarão como os mesmos estados de um programa podem conduzir a diferentes informações abstraídas em um ponto.

  * 2. Para o ponto (5) da Fig. 12, não existe uma definição que deva ser a definição de a nesse ponto, de modo que esse conjunto é vazio para a no ponto (5). Mesmo que uma variável tenha uma única definição em um ponto, essa definição precisa atribuir uma constante à variável. Assim, podemos simplesmente descrever certas variáveis como 'não constantes', em vez de coletar todos os seus valores possíveis ou todas as suas definições possíveis.

Otimização de Código

* Abaixo, Fig. 12:



Exemplo de um programa ilustrando a abstração do fluxo de dados.

* Assim, vemos que a mesma informação pode ser resumida de formas diferentes, dependendo da finalidade da análise.

Otimização de Código

* **Resolução de equações de fluxo de dados**
  * Em cada aplicação da análise de fluxo de dados, associamos a todo ponto do programa um **valor de fluxo de dados** que representa uma abstração do conjunto de todos os estados do programa possíveis observáveis nesse ponto.

  * O conjunto dos valores de fluxo de dados possíveis é o **domínio** para essa aplicação. Por exemplo, o domínio dos valores de fluxo de dados para alcançar as definições é o conjunto de todos os subconjuntos de definições do programa.

Otimização de Código

* **Resolução de equações de fluxo de dados**
  * Um valor de fluxo de dados particular é um conjunto de definições, e queremos associar a cada ponto do programa o conjunto exato de definições que podem alcançar esse ponto.

  * Conforme discutimos anteriormente, a escolha da abstração depende do objetivo da análise; para ser eficiente, só registramos a informação que é relevante.

Otimização de Código

* **Resolução de equações de fluxo de dados**
  * Denotamos os valores de fluxo de dados antes e depois de cada comando s por IN[s] e OUT[s], respectivamente.
  * O **problema de fluxo de dados** é encontrar uma solução para um conjunto de restrições nos valores IN[s] e OUT[s], para todos os comandos s.
  * Existem dois conjuntos de restrições: aquelas baseadas na semântica do comando (‘funções de transferência”) e aquelas baseadas no fluxo de controle.

Otimização de Código

* **Resolução de equações de fluxo de dados**
  * **Funções de transferência**
    * Os valores de fluxo de dados antes e depois de um comando são restringidos pela semântica do comando.

    * Por exemplo, suponha que nossa análise de fluxo de dados envolva determinar o valor constante das variáveis nos pontos. Se a variável a tiver valor v antes de executar o comando $b = a$, tanto a quanto b terão o valor v após o comando. O relacionamento entre os valores de fluxo de dados antes e depois do comando de atribuição é conhecido como **função de transferência.**

Otimização de Código

* **Resolução de equações de fluxo de dados**
  * **Funções de transferência**
    * As funções de transferência são de dois tipos: a informação pode propagar-se para frente, ao longo dos caminhos de execução, ou pode fluir para trás, pelos caminhos de execução.

    * Em um problema de fluxo para frente, a função de transferência de um comando *s*, a qual usualmente denotamos como *fs*, pega o valor de fluxo de dados antes do comando e produz um novo valor de fluxo de dados após o comando.

Otimização de Código

* **Resolução de equações de fluxo de dados**
  * **Funções de transferência**
    * Ou seja,

$$\text{OUT}[s] = fs(\text{IN}[s]).$$

    * Por outro lado, em um problema de fluxo para trás, a função de transferência fs, para o comando s converte um valor de fluxo de dados após o comando para um novo valor de fluxo de dados antes do comando. Ou seja,

$$\text{IN}[s] = fs(\text{OUT}[s]).$$

Otimização de Código

* **Resolução de equações de fluxo de dados**
  * **Restrições do fluxo de controle**
    * O segundo conjunto de restrições sobre os valores do fluxo de dados é derivado do fluxo de controle. Em um bloco básico, o fluxo de controle é simples.

    * Se um bloco *B* consiste nos comandos *s1*, *s2*,..., *sn*, nessa ordem, então o valor do fluxo de controle saindo de *si* é o mesmo que o valor de fluxo de controle entrando em *si* + 1. Ou seja,

$$\text{IN}[si + 1] = \text{OUT}[si], \text{ para todo } i = 1, 2, \ldots, n - 1.$$

Otimização de Código

* **Resolução de equações de fluxo de dados**
  * **Restrições do fluxo de controle**
    * Contudo, as arestas do fluxo de controle entre os blocos básicos criam restrições mais complexas entre o último comando de um bloco básico e o primeiro comando do bloco seguinte.

    * Por exemplo, se estivermos interessados em coletar todas as definições que podem alcançar um ponto do programa, então o conjunto de definições alcançando o comando líder de um bloco básico é a união das definições após os últimos comandos de cada um dos blocos predecessores. Veremos agora os detalhes de como os dados fluem entre os blocos.

Otimização de Código

* **Resolução de equações de fluxo de dados**
  * Como um esquema de fluxo de dados tecnicamente envolve os valores de fluxo de dados em cada ponto no programa, podemos economizar tempo e espaço reconhecendo que, em geral, o que acontece em um bloco é muito simples.

  * O controle flui do início até o fim do bloco, sem interrupção ou desvio. Então, podemos redeclarar o esquema em termos dos valores de fluxo de dados entrando e saindo dos blocos. Denotamos os valores de fluxo de dados imediatamente antes e imediatamente após cada bloco básico *B* por IN[*B*] e OUT[*B*], respectivamente.

Otimização de Código

* **Resolução de equações de fluxo de dados**
  * As restrições envolvendo IN[*B*] e OUT[*B*] podem ser derivadas daquelas envolvendo IN[s] e OUT[s] para os diversos comandos *s* em *B* da forma a seguir.

  * Suponha que o bloco *B* consista nos comandos *s1*, ..., *sn*, nessa ordem. Se *s1* é o primeiro comando do bloco básico *B*, então IN[*B*] = IN[*s1*]. De modo semelhante, se *sn* é o último comando do bloco básico *B*, então OUT[*B*] = OUT[*sn*]. A função de transferência de um bloco básico *B*, a qual denotamos como fb, pode ser derivada pela composição das funções de transferência dos comandos no bloco.

Otimização de Código

* **Resolução de equações de fluxo de dados**
  * Ou seja, considere que *fsi* seja a função de transferência do comando *si*. Então, *fb* = *fsn*, ..., *fs2*, *fs1*. O relacionamento entre o início e o fim do bloco é:

$$\text{OUT}[B] = fb(\text{IN}[B]).$$

  * As restrições derivadas ao fluxo de controle entre os blocos básicos podem ser facilmente reescritas substituindo-se IN[*s1*] e OUT[*sn*] por IN[*B*] e OUT[*B*], respectivamente.

Otimização de Código

* **Resolução de equações de fluxo de dados**
  * Por exemplo, se os valores de fluxo de dados são informações sobre os conjuntos de constantes que podem ser atribuídas a uma variável, então temos um problema de fluxo para frente, no qual:

$$\text{IN}[B] = \bigcup_{P \text{ um predecessor de } B} \text{OUT}[P].$$

  * Quando o fluxo de dados é para trás, como veremos em breve na análise de variável viva, as equações são semelhantes, mas com papéis do INs e OUTs invertidos. Ou seja:

$$\text{IN}[B] = f_B(\text{OUT}[B])$$

$$\text{OUT}[B] = \bigcup_{S \text{ um sucessor de } B} \text{IN}[S].$$

Otimização de Código

* **Resolução de equações de fluxo de dados**
  * Ao contrário das equações aritméticas lineares, as equações de fluxo de dados não costumam possuir uma única solução.
  * Nosso objetivo é encontrar a solução mais 'precisa' que satisfaça os dois conjuntos de restrições: restrições de fluxo de controle e das funções de transferência.
  * Ou seja, precisamos de uma solução que **encoraje as melhorias de código válidas**, mas que não justifique transformações inseguras – aquelas que mudam o que o programa computa.

Otimização de Código

* **Resolução de equações de fluxo de dados**
  * Discutiremos agora, alguns dos exemplos mais importantes dos problemas que podem ser solucionados pela análise de fluxo de dados:
    * Definições de alcance
    * Análise de variável viva
    * Expressões disponíveis

Otimização de Código

* **Resolução de equações de fluxo de dados**
  * **Definições de alcance**
    * ‘Definições de alcance’ são um dos esquemas de fluxo de dados mais comuns e mais úteis.
    * Sabendo onde em um programa cada variável x pode ter sido definida quando o controle alcança cada ponto p, é possível determinar muitas informações sobre x.

Otimização de Código

* **Resolução de equações de fluxo de dados**
  * **Definições de alcance**
    * Citando apenas dois exemplos, um compilador pode saber se x é uma constante no ponto p, e um depurador pode dizer se é possível que x seja uma variável indefinida, caso x seja usado em p.

    * Dizemos que uma definição d alcança um ponto p se houver um caminho do ponto imediatamente após d para p, tal que d não seja 'morto' ao longo desse caminho.

Otimização de Código

* **Resolução de equações de fluxo de dados**
  * **Definições de alcance**
    * Matamos uma definição de uma variável x se houver qualquer outra definição de x em algum outro ponto ao longo do caminho. (Observe que o caminho pode ter *loops*, de modo que poderíamos chegar a outra ocorrência de d ao longo do caminho, o que não 'mata' d.

    * Intuitivamente, se uma definição d de alguma variável x alcançar o ponto p, então d poderia ser o local no qual o valor de x usado em p foi definido pela última vez.

Otimização de Código

- **Resolução de equações de fluxo de dados**
  - **Definições de alcance**
    - Uma definição de uma variável x é um comando que atribui (ou pode atribuir) um valor a x.
    - Parâmetros de procedimento, acessos a arranjo e referências indiretas podem ter sinônimos (aliases), e não é fácil saber se um comando está referindo-se a uma variável x em particular.

Otimização de Código

* **Resolução de equações de fluxo de dados**
  * **Definições de alcance**
    * A análise do programa precisa ser **conservadora**; se não sabemos se um comando s está atribuindo um valor a x, devemos considerar que ele pode atribuir, ou seja, a variável x após o comando s pode ter seu valor original antes de s ou o novo valor criado por s.

    * Por questão de simplicidade, consideraremos que estamos tratando apenas com variáveis que não possuem sinônimos.

Otimização de Código

* **Resolução de equações de fluxo de dados**
  * **Definições de alcance**
    * Essa classe de variáveis inclui todas as variáveis escalares locais na maioria das linguagens; no caso de C e C++, as variáveis locais cujos endereços foram calculados em algum ponto são excluídas.

Otimização de Código

* **Resolução de equações de fluxo de dados**
  * **Definições de alcance**
    * Vejamos agora, como usar uma solução do problema de definição de alcance para detectar os **usos antes da definição**.

    * O truque é introduzir uma definição fictícia para cada variável x na entrada do grafo de fluxo. Se a definição fictícia de x alcançar um ponto p onde x poderia ser usado, então pode haver uma oportunidade de usar x antes da definição.

Otimização de Código

* **Resolução de equações de fluxo de dados**
  * **Definições de alcance**
    * Observe que nunca estaremos absolutamente certos de que o programa tem um erro, pois pode haver algum motivo, possivelmente envolvendo um argumento lógico complexo, para que o caminho ao longo do qual p é alcançado nunca possa ser tomado sem uma definição real de x.

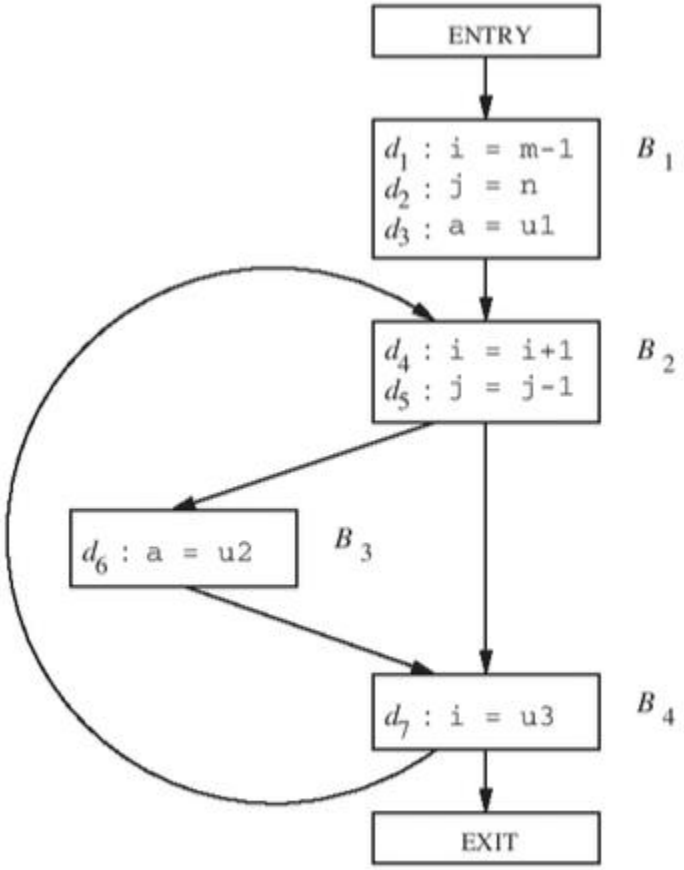
Otimização de Código

* **Resolução de equações de fluxo de dados**
  * **Definições de alcance**
    * Exemplo 9: A Fig. 13 mostra um grafo de fluxo com sete definições.

    * Vamos focalizar nas definições que alcançam o bloco *B2*. Todas as definições do bloco *B1* alcançar o início do bloco *B2*. A definição *d5*: $j = j - 1$ no bloco *B2* também alcançar o início do bloco *B2*, pois nenhuma outra definição de j pode ser encontrada no loop levando de volta a *B2*. Essa definição não alcança o início de *B2*, porque a variável i é sempre redefinida por *d7*: $i = u3$. Finalmente, a definição *d6*: a = *u2* também alcança o início do bloco *B2*.

Otimização de Código

* Fig. 13



Grafo de fluxo para ilustrar as definições de alcance.

Otimização de Código

* **Resolução de equações de fluxo de dados**
  * **Definições de alcance**
    * A determinação das definições de alcance às vezes gera imprecisões.
    * Contudo, todas elas estão em direção ‘segura’, ou ‘conservativa’. Observe nossa suposição de que todas as arestas em um grafo de fluxo podem ser atravessadas. Esta suposição pode não ser verdadeira na prática.

Otimização de Código

* **Resolução de equações de fluxo de dados**
  * **Definições de alcance**
    * Por exemplo, no fragmento de programa a seguir, independentemente dos valores de a e b, o fluxo de controle não pode realmente alcançar o comando 2.

```
if (a == b) comando 1; else if (a == b) comando 2;
```

    * Em geral, decidir se cada um dos caminhos em um grafo de fluxo pode ser seguido é um problema **indecidível**.

Otimização de Código

* **Resolução de equações de fluxo de dados**
  * **Definições de alcance**
    * Assim, simplesmente consideramos que todo caminho no grafo de fluxo pode ser seguido em alguma execução do programa. **Na maioria das aplicações de definições de alcance, a posição conservadora é considerar que uma definição pode alcançar um ponto, mesmo que não possa**.

    * Assim, podemos permitir caminhos que nunca são seguidos em nenhuma execução do programa, e podemos permitir que as definições passem com segurança por definições ambíguas da mesma variável.

Otimização de Código

* **Resolução de equações de fluxo de dados**
  * **Equações de fluxo de controle**
    * Em seguida, consideramos o conjunto de restrições derivadas do fluxo de controle entre os blocos básicos.

    * Como uma definição alcança um ponto no programa desde que exista pelo menos um caminho ao longo do qual a definição alcança, $\text{OUT}[P] \subseteq \text{IN}[B]$ sempre que houver uma aresta de fluxo de controle de *P* para *B*. Contudo, como uma definição não pode alcançar um ponto a menos que haja um caminho pelo qual ela o alcance, IN[*B*] não precisa ser maior do que a união das definições de alcance de todos os blocos predecessores.

Otimização de Código

* **Resolução de equações de fluxo de dados**
  * **Equações de fluxo de controle**
    * Ou seja, é seguro considerar que:

$$\text{IN}[B] = \bigcup_{P \text{ um predecessor de } B} \text{OUT}[P]$$

    * Nós nos referimos à união como o operador *meet* para as definições de alcance. Em qualquer esquema de fluxo de dados, o operador *meet* é aquele que usamos para criar um resumo das contribuições de diferentes caminhos na confluência desses caminhos.

Otimização de Código

- **Resolução de equações de fluxo de dados**
  - **Análise de variável viva**
    - Algumas transformações de melhoria de código dependem da informação calculada na direção oposta ao fluxo de controle de um programa; vamos, agora, examinar um exemplo.

    - Na análise de variável viva, queremos saber para a variável x e o ponto p se o valor de x em p poderia ser usado por algum caminho no grafo de fluxo que comece em p. Caso isso seja possível, dizemos que x está viva em p; caso contrário, x está morta em p.

Otimização de Código

* **Resolução de equações de fluxo de dados**
  * **Análise de variável viva**
    * Um importante uso da informação sobre variável viva é a alocação de registradores para os blocos básicos.

    * Os aspectos dessa questão foram introduzidos na AULA07. Após um valor ser calculado em um registrador, e presumivelmente usado em um bloco, não é necessário armazená-lo se ele estiver morto no fim do bloco. Além disso, se todos os registradores estiverem cheios e precisarmos de outro registrador, devemos favorecer o uso de um registrador com um valor morto, porque este não precisará ser armazenado.

Otimização de Código

* **Resolução de equações de fluxo de dados**
  * **Análise de variável viva**
    * Iremos definir as equações de fluxo de dados diretamente em termos de IN[*B*] e OUT[*B*], as quais representam o conjunto de variáveis vivas nos pontos imediatamente antes e depois do bloco *B*, respectivamente.

    * Essas equações também podem ser derivadas, definindo-se, primeiro, as funções de transferência dos comandos individuais e, depois, compondo-os para criar a função de transferência de um bloco básico.

Otimização de Código

* **Resolução de equações de fluxo de dados**
  * **Análise de variável viva**
    * Defina:
    * 1) *defB* como o conjunto de variáveis definidas (ou seja, valores atribuídos definitivamente) em *B* antes de qualquer uso dessa variável em *B*, e
    * 2) *useB* como o conjunto de variáveis cujos valores podem ser usados em *B* antes de qualquer definição da variável.

Otimização de Código

* **Resolução de equações de fluxo de dados**
  * **Análise de variável viva**
    * Exemplo 10: Por exemplo, o bloco *B2* da Fig. 13 definitivamente usa i. Ele também usa j antes de qualquer redefinição de j, a menos que seja possível que i e j sejam sinônimos um do outro.

    * Supondo que não existam sinônimos entre as variáveis da Fig. 13, *useB2* = $\{i, j\}$. Além disso, *B2* define claramente i e j. Supondo que não existam sinônimos, *defB2* = $\{i, j\}$ também.

Otimização de Código

* **Resolução de equações de fluxo de dados**
  * **Análise de variável viva**
    * Como consequência das definições, qualquer variável em *useB* deve ser considerada viva na entrada do bloco *B*, enquanto definições de variáveis em *defB* definitivamente estão mortas no início de *B*.

    * Com efeito, a inclusão como membro de *defB* 'mata' qualquer oportunidade de a variável estar viva por causa dos caminhos que começam em *B*.

Otimização de Código

* **Resolução de equações de fluxo de dados**
  * **Análise de variável viva**
    * Assim, as equações relacionando *def* e *use* aos desconhecidos IN e OUT são definidas da seguinte forma:

$$\text{IN}[\text{EXIT}] = \emptyset$$

    * e para todos os blocos básicos *B* diferentes de EXIT.

$$\text{IN}[B] = use_B \cup (\text{OUT}[B] - def_B)$$

$$\text{OUT}[B] = \bigcup_{\text{S um sucessor de B}} \text{IN}[S]$$

Otimização de Código

* **Resolução de equações de fluxo de dados**
  * **Análise de variável viva**
    * A **primeira equação** especifica a condição de contorno, a qual estabelece que nenhuma variável esteja viva na saída do programa.

    * A **segunda equação** diz que uma variável estará viva na entrada de um bloco se ela for usada antes de sua redefinição no bloco ou se estiver viva saindo do bloco e não for redefinida nele. A **terceira equação** diz que uma variável estará viva na saída de um bloco se e somente se ela estiver viva entrando em um de seus sucessores.

Otimização de Código

* **Resolução de equações de fluxo de dados**
  * **Análise de variável viva**
    * O relacionamento entre as equações de tempo de vida e as equações de definição de alcance deve ser observado:
      * Os dois conjuntos de equações têm a união como o operador *meet*. O motivo é que, em cada esquema de fluxo de dados, propagamos informações pelos caminhos, e só nos importamos em saber se existe algum caminho com propriedades desejadas, e não se algo é verdadeiro por todos os caminhos.

Otimização de Código

* **Resolução de equações de fluxo de dados**
  * **Análise de variável viva**
    * O relacionamento entre as equações de tempo de vida e as equações de definição de alcance deve ser observado:
      * Contudo, o fluxo de informações de tempo de vida trafega 'para trás', oposto à direção do fluxo de controle, porque, nesse problema, queremos ter certeza de que o uso de uma variável x em um ponto p é transmitido a todos os pontos antes de p em um caminho de execução, de modo que no ponto anterior possamos saber que x terá seu valor usado.

Otimização de Código

* **Resolução de equações de fluxo de dados**
  * **Análise de variável viva**
    * Para solucionar um problema para trás, em vez de inicializar OUT[ENTRY], inicializamos IN[EXIT]. Os conjuntos IN e OUT têm seus papéis trocados neste contexto.

    * Assim como para as definições de alcance, a solução para as equações de tempo de vida não é necessariamente **única**, e queremos a solução com os menores conjuntos de variáveis vivas.

Otimização de Código

- **Resolução de equações de fluxo de dados**
  - **Expressões Disponíveis**
    - Uma expressão $x + y$ está disponível em um ponto p se todo o caminho do nó de entrada para p avaliar $x + y$, e depois da última avaliação e antes de alcançar p, não houver atribuições subsequentes a x ou y.
    - Para um esquema de fluxo de dados de expressão disponíveis, dizemos que um bloco mata a expressão $x + y$ se ele atribuir (ou puder atribuir) x ou y e não recalcular $x + y$ posteriormente. Um bloco gera a expressão $x + y$ se ele avaliar definitivamente $x + y$ e não definir x ou y subsequentemente.

Otimização de Código

- **Resolução de equações de fluxo de dados**
  - **Expressões Disponíveis**
    - O principal uso da informação sobre expressão disponível é detectar subexpressões comuns globais.

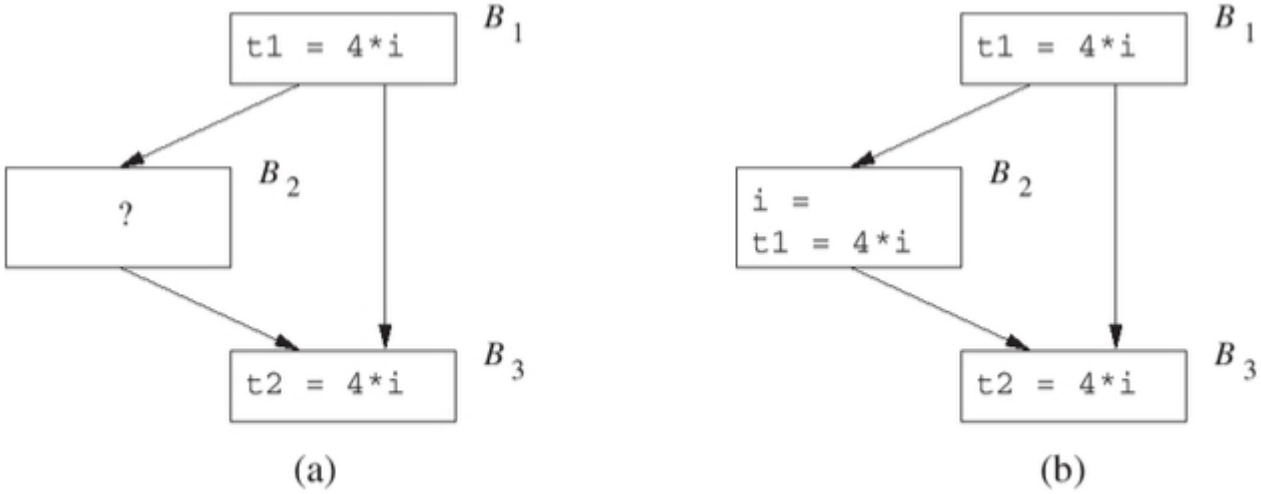
    - Por exemplo, na Fig. 14(a), a expressão 4 * i no bloco $B3$ será uma subexpressão comum se 4 * i estiver disponível no ponto de entrada do bloco $B3$. Ela estará disponível se i não receber um novo valor no bloco $B2$, ou se, como na Fig. 14(b), 4 * i for recalculado depois que i for atribuído em $B2$.

Otimização de Código

- **Resolução de equações de fluxo de dados**
  - **Expressões Disponíveis**
    - Podemos calcular o conjunto de expressões geradas para cada ponto em um bloco, trabalhando do início ao fim desse bloco. No ponto antes do bloco, nenhuma expressão é gerada. Se no ponto p o conjunto S de expressões estiver disponível, e q for o ponto após p, com o comando $x = y + z$ entre eles, então formamos o conjunto de expressões disponíveis em q pelos dois a seguir:
      - 1) Acrescente a S a expressão $y + z$.
      - 2) Remova de S qualquer expressão envolvendo a variável x.

Otimização de Código

* **Resolução de equações de fluxo de dados**
  * **Expressões Disponíveis**
    * Abaixo, Fig. 14.



Subexpressões comuns em potencial entre blocos.

Otimização de Código

* **Resolução de equações de fluxo de dados**
  * **Expressões Disponíveis**
    * Observe que os passos devem ser seguidos na ordem correta, pois x poderia ser o mesmo que y ou z.

    * Depois de alcançar o fim do bloco, S é o conjunto de expressões geradas pelo bloco. O conjunto de expressões mortas são todas as expressões, digamos $y + z$, tal que ou y ou z é definido no bloco, e $y + z$ não é gerado pelo bloco.

Otimização de Código

* **Resolução de equações de fluxo de dados**
  * **Expressões Disponíveis**
    * Exemplo 11: Considere os quatro comandos da Fig. 15 a seguir.
    * Após o primeiro, $b + c$ está disponível. Depois do segundo comando, $a - d$ torna-se disponível, mas $b + c$ não está mais disponível, porque b foi redefinido. O terceiro comando não torna $b + c$ disponível novamente, porque o valor de c é imediatamente alterado. Depois do último comando, $a - d$ não está mais disponível, pois d mudou. Assim, nenhuma expressão é gerada, e todas as expressões envolvendo a, b, c ou d são mortas.

Otimização de Código

* **Resolução de equações de fluxo de dados**
  * **Expressões Disponíveis**
    * Abaixo, Fig. 15:

Comando	Expressões disponíveis
	$\emptyset$
`a = b + c`	
	$\{b + c\}$
`b = a - d`	
	$\{a - d\}$
`c = b + c`	
	$\{a - d\}$
`d = a - d`	
	$\emptyset$

Cálculo de expressões disponíveis.

Otimização de Código

* **Transformações de código gerado**
  * Como todos os esquemas de fluxo de dados calculam aproximações de verdade básica (conforme definida por todos os caminhos de execução possíveis do programa), somos obrigados a garantir que quaisquer erros são na direção 'segura'.

  * Uma decisão de política é segura (ou conservativa) se nunca nos permitir mudar o que o programa calcula. Infelizmente, as políticas seguras podem fazer-nos perder algumas melhorias no código, os quais reteriam o significado do programa, mas em praticamente todas as otimizações de código **não existe uma política segura, que não perca nada.**

Otimização de Código

* **Transformações de código gerado**
  * Geralmente, seria inaceitável usar uma política insegura, ou seja, **que acelere o código, mas mude o que o programa calcula**.

  * Assim, ao projetar um esquema de fluxo de dados, devemos estar conscientes de como a informação será usada, e **garantir que quaisquer aproximações que fizermos estejam na direção 'conservativa' ou 'segura'.**

Otimização de Código

* **Transformações de código gerado**
  * Cada esquema e aplicação devem ser considerados independentemente. Por exemplo, se usarmos definições de alcance para o desdobramento de constante, é seguro pensar que uma definição é alcançada quando ela não é (poderíamos pensar que x não é uma constante, quando de fato ela é e poderia ter sido desdobrada).

  * Mas não é seguro pensar que uma definição não é alcançada quando ela é (poderíamos substituir x por uma constante, quando o programa, às vezes, teria um valor para x diferente dessa constante).

Resumo

* ***Subexpressões comuns globais***: Uma otimização importante é encontrar computações da mesma expressão em dois blocos básicos diferentes. Se um precede o outro, podemos armazenar o resultado da primeira vez que ele é computado e usar o resultado armazenado em ocorrências subsequentes.

* ***Propagação de cópia***: Um comando de cópia, $u = v$, atribui uma variável v a outra, u. Em algumas circunstâncias, podemos substituir todos os usos de u por v, eliminando dessa forma a ambos, tanto a atribuição quanto u.

* ***Movimentação de código***: Outra otimização é mover um cálculo para fora do loop em que ele aparece. Essa mudança só é correta se o cálculo produzir o mesmo valor cada vez que o loop é iterado.

Resumo

* ***Variáveis de indução***: Muitos *loops* possuem variáveis de indução, variáveis que recebem uma sequência linear de valores a cada passo do *loop*. Algumas delas são usadas apenas para contar iterações e, frequentemente, podem ser eliminadas, reduzindo assim o tempo gasto para iterar o *loop*.

* ***Análise de fluxo de dados***: Um esquema de análise de fluxo de dados define um valor em cada ponto do programa. Os comandos do programa têm funções de transferência associadas, que relacionam os valores antes e depois do comando. Comandos com mais de um predecessor devem ter seu valor definido pela combinação dos valores nos predecessores, usando um operador *meet* (ou confluência).

Resumo

* ***Análise de fluxo de dados em blocos básicos***: Como a propagação dos valores de fluxo de dados em um bloco costuma ser muito simples, as equações de fluxo de dados geralmente são projetadas para terem duas variáveis em cada bloco, chamadas IN e OUT. Estas duas variáveis representam os valores de fluxo de dados no início e no fim do bloco, respectivamente. As funções de transferência dos comandos em um bloco são compostas para obter a função de transferência para o bloco como um todo.

* ***Definição de alcance***: A estrutura de fluxo de dados de definições de alcance tem valores que são conjuntos de comando no programa que definem valores para uma ou mais variáveis. A função de transferência para um bloco mata as definições de variáveis que são definitivamente redefinidas no bloco e acrescenta (‘gera’) as definições de variáveis que ocorrem no bloco. O operador de confluência é a união, uma vez que as definições alcançam um ponto se alcançarem qualquer predecessor desse ponto.

Resumo

* ***Variáveis vivas***: Outra estrutura de fluxo de dados importante calcula as variáveis que estão vivas (serão usadas antes da redefinição) em cada ponto. A estrutura é semelhante às definições de alcance, exceto pelo fato que a função de transferência é executada do fim para o início. Uma variável está viva no início de um bloco se for usada antes da definição no bloco ou estiver viva no fim e não for redefinida no bloco.

* ***Expressões disponíveis***: Para descobrir as subexpressões comuns globais, determinamos as expressões disponíveis em cada ponto – expressões que foram calculadas, das quais nenhum argumento foi redefinido após o último cálculo. A estrutura de fluxo de dados é semelhante às definições de alcance, mas o operador de confluência é a interseção, em vez da união.

* ***Abstração de problemas de fluxo de dados***: Problemas de fluxo de dados comuns, como aqueles já mencionados, podem ser expressos em uma estrutura matemática comum, através de valores, operador de confluência e funções de transferência.

Obrigado.

joaopauloaramuni@gmail.com
joaopauloaramuni@fumec.br

UNIVERSIDADE FUMEC
50 Anos